Introduction

1 News

2 Example: Robust Multi-stage MPC

3 Example: Nonlinear MPC

4 Next steps

5 Table of contents

5.1 Getting started: MPC

5.1.1 Example system

5.1.2 Creating the model

5.1.2.1 Model variables

5.1.2.2 Query variables

5.1.2.3 Model parameters

5.1.2.4 Right-hand-side equation

5.1.3 Configuring the MPC controller

5.1.3.1 Optimizer parameters

5.1.3.2 Objective function

5.1.3.3 Constraints

5.1.3.4 Scaling

5.1.3.5 Uncertain Parameters

5.1.3.6 Setup

5.1.4 Configuring the Simulator

5.1.4.1 Simulator parameters

5.1.4.2 Uncertain parameters

5.1.4.3 Setup

5.1.5 Creating the control loop

5.1.5.1 Setting up the Graphic

5.1.5.2 Running the simulator

5.1.5.3 Running the optimizer

5.1.5.4 Changing the line appearance

5.1.5.5 Running the control loop

5.1.6 Data processing

5.1.6.1 Saving and retrieving results

5.1.6.2 Working with data objects

5.1.6.3 Animating results
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.11.6</td>
<td>data</td>
</tr>
<tr>
<td>5.11.6.1</td>
<td>Data</td>
</tr>
<tr>
<td>5.11.6.1.1</td>
<td>export</td>
</tr>
<tr>
<td>5.11.6.1.2</td>
<td>init_storage</td>
</tr>
<tr>
<td>5.11.6.1.3</td>
<td>set_meta</td>
</tr>
<tr>
<td>5.11.6.1.4</td>
<td>update</td>
</tr>
<tr>
<td>5.11.6.2</td>
<td>MPCData</td>
</tr>
<tr>
<td>5.11.6.2.1</td>
<td>export</td>
</tr>
<tr>
<td>5.11.6.2.2</td>
<td>init_storage</td>
</tr>
<tr>
<td>5.11.6.2.3</td>
<td>prediction</td>
</tr>
<tr>
<td>5.11.6.2.4</td>
<td>set_meta</td>
</tr>
<tr>
<td>5.11.6.2.5</td>
<td>update</td>
</tr>
<tr>
<td>5.11.6.3</td>
<td>load_results</td>
</tr>
<tr>
<td>5.11.6.4</td>
<td>save_results</td>
</tr>
<tr>
<td>5.11.7</td>
<td>graphics</td>
</tr>
<tr>
<td>5.11.7.1</td>
<td>Graphics</td>
</tr>
<tr>
<td>5.11.7.1.1</td>
<td>pred_lines</td>
</tr>
<tr>
<td>5.11.7.1.2</td>
<td>result_lines</td>
</tr>
<tr>
<td>5.11.7.1.3</td>
<td>add_line</td>
</tr>
<tr>
<td>5.11.7.1.4</td>
<td>clear</td>
</tr>
<tr>
<td>5.11.7.1.5</td>
<td>plot_predictions</td>
</tr>
<tr>
<td>5.11.7.1.6</td>
<td>plot_results</td>
</tr>
<tr>
<td>5.11.7.1.7</td>
<td>reset_predictions</td>
</tr>
<tr>
<td>5.11.7.1.8</td>
<td>reset_axes</td>
</tr>
<tr>
<td>5.11.7.2</td>
<td>animate</td>
</tr>
<tr>
<td>5.11.7.3</td>
<td>default_plot</td>
</tr>
<tr>
<td>5.11.8</td>
<td>samplingplanner</td>
</tr>
<tr>
<td>5.11.8.1</td>
<td>SamplingPlanner</td>
</tr>
<tr>
<td>5.11.8.1.1</td>
<td>data_dir</td>
</tr>
<tr>
<td>5.11.8.1.2</td>
<td>add_sampling_case</td>
</tr>
<tr>
<td>5.11.8.1.3</td>
<td>export</td>
</tr>
<tr>
<td>5.11.8.1.4</td>
<td>gen_sampling_plan</td>
</tr>
<tr>
<td>5.11.8.1.5</td>
<td>set_param</td>
</tr>
<tr>
<td>5.11.8.1.6</td>
<td>set_sampling_var</td>
</tr>
<tr>
<td>5.11.9</td>
<td>sampler</td>
</tr>
<tr>
<td>5.11.9.1</td>
<td>Sampler</td>
</tr>
<tr>
<td>5.11.9.1.1</td>
<td>data_dir</td>
</tr>
<tr>
<td>5.11.9.1.2</td>
<td>sample_data</td>
</tr>
<tr>
<td>5.11.9.1.3</td>
<td>sample_idx</td>
</tr>
<tr>
<td>5.11.9.1.4</td>
<td>set_param</td>
</tr>
<tr>
<td>5.11.9.1.5</td>
<td>set_sample_function</td>
</tr>
<tr>
<td>5.11.10</td>
<td>datahandler</td>
</tr>
<tr>
<td>5.11.10.1</td>
<td>DataHandler</td>
</tr>
<tr>
<td>5.11.10.1.1</td>
<td>data_dir</td>
</tr>
<tr>
<td>5.11.10.1.2</td>
<td>filter</td>
</tr>
<tr>
<td>5.11.10.1.3</td>
<td>set_param</td>
</tr>
<tr>
<td>5.11.10.1.4</td>
<td>set_post_processing</td>
</tr>
<tr>
<td>5.12</td>
<td>Release notes</td>
</tr>
<tr>
<td>5.12.1</td>
<td>v4.3.5</td>
</tr>
<tr>
<td>5.12.1.1</td>
<td>Minor fixes</td>
</tr>
<tr>
<td>5.12.2</td>
<td>v4.3.4</td>
</tr>
<tr>
<td>5.12.2.1</td>
<td>Minor fixes</td>
</tr>
<tr>
<td>5.12.3</td>
<td>v4.3.3</td>
</tr>
<tr>
<td>5.12.3.1</td>
<td>Major changes</td>
</tr>
</tbody>
</table>
5.12.15.1 Example files .................................................. 180
5.12.16 do-mpc v3.0.0 .................................................. 181
5.12.16.1 Main modifications .......................................... 181
5.12.17 do-mpc v2.0.0 .................................................. 181
5.12.18 do-mpc version 1.0.0 .......................................... 181
5.13 Batch Bioreactor ................................................... 181
  5.13.1 Model ............................................................. 181
    5.13.1.1 States and control inputs .............................. 182
    5.13.1.2 ODE and parameters .................................. 182
  5.13.2 Controller ..................................................... 183
    5.13.2.1 Objective .............................................. 183
    5.13.2.2 Constraints .......................................... 183
    5.13.2.3 Uncertain values ..................................... 184
  5.13.3 Estimator ..................................................... 184
  5.13.4 Simulator ..................................................... 184
    5.13.4.1 Realizations of uncertain parameters .............. 185
  5.13.5 Closed-loop simulation ..................................... 185
    5.13.5.1 Prepare visualization ................................. 185
    5.13.5.2 Run closed-loop ..................................... 186
    5.13.5.3 Results ............................................. 186
5.14 Continuous stirred tank reactor (CSTR) ......................... 187
  5.14.1 Model .......................................................... 187
    5.14.1.1 States and control inputs .......................... 188
    5.14.1.2 ODE and parameters ................................ 188
  5.14.2 Controller .................................................... 189
    5.14.2.1 Objective ............................................ 190
    5.14.2.2 Constraints ......................................... 190
    5.14.2.3 Uncertain values ................................... 191
  5.14.3 Estimator .................................................... 191
  5.14.4 Simulator .................................................... 191
    5.14.4.1 Realizations of uncertain parameters ............. 191
  5.14.5 Closed-loop simulation ..................................... 192
  5.14.6 Animating the results ...................................... 193
5.15 Industrial polymerization reactor ................................ 194
  5.15.1 Model .......................................................... 194
    5.15.1.1 System description .................................. 195
    5.15.1.2 Implementation ...................................... 197
  5.15.2 Controller .................................................... 199
    5.15.2.1 Objective ............................................ 199
    5.15.2.2 Constraints ......................................... 199
    5.15.2.3 Scaling ............................................ 202
    5.15.2.4 Uncertain values ................................... 203
  5.15.3 Estimator .................................................... 203
  5.15.4 Simulator .................................................... 203
    5.15.4.1 Realizations of uncertain parameters ............. 203
  5.15.5 Closed-loop simulation ..................................... 204
  5.15.6 Animating the results ...................................... 205
5.16 Oscillating masses ................................................ 206
  5.16.1 Model .......................................................... 206
    5.16.1.1 States and control inputs .......................... 208
  5.16.2 Controller .................................................... 208
    5.16.2.1 Objective ............................................ 209
    5.16.2.2 Constraints ......................................... 209
  5.16.3 Estimator .................................................... 210
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.16.4</td>
<td>Simulator</td>
<td>210</td>
</tr>
<tr>
<td>5.16.5</td>
<td>Closed-loop simulation</td>
<td>210</td>
</tr>
<tr>
<td>5.16.6</td>
<td>Displaying the results</td>
<td>211</td>
</tr>
<tr>
<td>5.17</td>
<td>Double inverted pendulum</td>
<td>211</td>
</tr>
<tr>
<td>5.17.1</td>
<td>Model</td>
<td>212</td>
</tr>
<tr>
<td>5.17.1.1</td>
<td>Parameters</td>
<td>212</td>
</tr>
<tr>
<td>5.17.1.2</td>
<td>Euler-Lagrangian equations</td>
<td>213</td>
</tr>
<tr>
<td>5.17.1.3</td>
<td>Differential algebraic equation (DAE)</td>
<td>213</td>
</tr>
<tr>
<td>5.17.1.4</td>
<td>Energy equations</td>
<td>214</td>
</tr>
<tr>
<td>5.17.2</td>
<td>Controller</td>
<td>215</td>
</tr>
<tr>
<td>5.17.2.1</td>
<td>Objective</td>
<td>215</td>
</tr>
<tr>
<td>5.17.2.2</td>
<td>Constraints</td>
<td>216</td>
</tr>
<tr>
<td>5.17.3</td>
<td>Estimator</td>
<td>216</td>
</tr>
<tr>
<td>5.17.4</td>
<td>Simulator</td>
<td>216</td>
</tr>
<tr>
<td>5.17.5</td>
<td>Closed-loop simulation</td>
<td>217</td>
</tr>
<tr>
<td>5.17.5.1</td>
<td>Prepare visualization</td>
<td>217</td>
</tr>
<tr>
<td>5.17.5.2</td>
<td>Run open-loop</td>
<td>219</td>
</tr>
<tr>
<td>5.17.5.3</td>
<td>Run closed-loop</td>
<td>225</td>
</tr>
<tr>
<td>5.17.5.4</td>
<td>Results</td>
<td>225</td>
</tr>
<tr>
<td>5.17.6</td>
<td>Controller with obstacle avoidance</td>
<td>226</td>
</tr>
<tr>
<td>5.18</td>
<td>Efficient data generation and handling with do-mpc</td>
<td>226</td>
</tr>
<tr>
<td>5.18.1</td>
<td>Toy example</td>
<td>227</td>
</tr>
<tr>
<td>5.18.2</td>
<td>Sampling closed-loop trajectories</td>
<td>231</td>
</tr>
</tbody>
</table>

6 Indices and tables 235

Bibliography 237

Python Module Index 239

Index 241
**do-mpc** is a comprehensive open-source toolbox for robust **model predictive control (MPC)** and **moving horizon estimation (MHE)**. **do-mpc** enables the efficient formulation and solution of control and estimation problems for nonlinear systems, including tools to deal with uncertainty and time discretization. The modular structure of **do-mpc** contains simulation, estimation and control components that can be easily extended and combined to fit many different applications.

In summary, **do-mpc** offers the following features:

- nonlinear and economic model predictive control
- support for differential algebraic equations (DAE)
- time discretization with orthogonal collocation on finite elements
- robust multi-stage model predictive control
- moving horizon state and parameter estimation
- modular design that can be easily extended

The **do-mpc** software is Python based and works therefore on any OS with a Python 3.x distribution. **do-mpc** has been developed by Sergio Lucia and Alexandru Tatulea at the DYN chair of the TU Dortmund lead by Sebastian Engell. The development is continued at the Laboratory of Process Automation Systems (PAS) of the TU Dortmund by Felix Fiedler and Sergio Lucia.
The first do-mpc developer conference was hosted from 13.09.-15.09.2021 at TU Dortmund. For everyone who missed the event, we are happy to share the recording below:
Example: Robust Multi-stage MPC

We showcase an example, where the control task is to regulate the rotating triple-mass-spring system as shown below:

Once excited, the uncontrolled system takes a long time to come to a rest. To influence the system, two stepper motors are connected to the outermost discs via springs. The designed controller will result in something like this:

Assume, we have modeled the system from first principles and identified the parameters in an experiment. We are especially unsure about the exact value of the inertia of the masses. With Multi-stage MPC, we can define different scenarios e.g. $\pm 10\%$ for each mass and predict as well as optimize multiple state and input trajectories. This family of trajectories will always obey to set constraints for states and inputs and can be visualized as shown below:
In the next example we showcase the capabilities of do-mpc to handle complex nonlinear systems. The task is to erect the classical double inverted pendulum (DIP) and navigate it around an obstacle.

The governing system equation is given as an implicit ODE:

\[ 0 = f(\dot{x}(t), x(t), u(t)), \]

which can be rewritten as:

\[ \dot{x} = z \\
0 = g(x(t), z(t), u(t)) \]

and thus constitutes a differential algebraic equation (DAE) which is fully supported by do-mpc.

The controller in this example is configured with an economic objective, where the task is to maximize the potential energy of the system while minimizing the kinetic energy.

An animation of the obtained controller results is shown below:

The code to recreate these results can be found in our example gallery.
We suggest you start by skimming over the selected examples below to get an first impression of the above mentioned features. A great further read for interested viewers is the `getting started: MPC` page, where we show how to setup `do-mpc` for the robust control task of a triple-mass-spring system. A state and parameter moving horizon estimator is configured and used for the same system in `getting started: MHE`.

To install `do-mpc` please see our installation instructions.
5.1 Getting started: MPC

In this Jupyter Notebook we illustrate the core functionalities of do-mpc.

Open an interactive online Jupyter Notebook with this content on Binder:

We start by importing the required modules, most notably do_mpc.

```python
import numpy as np

# Add do_mpc to path. This is not necessary if it was installed via pip.
import sys
sys.path.append('./..')

# Import do_mpc package:
import do_mpc
```

One of the essential paradigms of do-mpc is a modular architecture, where individual building bricks can be used independently or jointly, depending on the application.

In the following we will present the configuration, setup and connection between these blocks, starting with the model.
5.1.1 Example system

First, we introduce a simple system for which we setup do-mpc. We want to control a triple mass spring system as depicted below:

Three rotating discs are connected via springs and we denote their angles as $\phi_1, \phi_2, \phi_3$. The two outermost discs are each connected to a stepper motor with additional springs. The stepper motor angles ($\phi_{m,1}$ and $\phi_{m,2}$) are used as inputs to the system. Relevant parameters of the system are the inertia $\Theta$ of the three discs, the spring constants $c$ as well as the damping factors $d$. 
The second degree ODE of this system can be written as follows:

\[
\Theta_1 \ddot{\phi}_1 = -c_1 (\phi_1 - \phi_{m,1}) - c_2 (\phi_1 - \phi_2) - d_1 \dot{\phi}_1 \tag{5.1}
\]
\[
\Theta_2 \ddot{\phi}_2 = -c_2 (\phi_2 - \phi_1) - c_3 (\phi_2 - \phi_3) - d_2 \dot{\phi}_2 \tag{5.2}
\]
\[
\Theta_3 \ddot{\phi}_3 = -c_3 (\phi_3 - \phi_2) - c_4 (\phi_3 - \phi_{m,2}) - d_3 \dot{\phi}_3 \tag{5.3}
\]

The uncontrolled system, starting from a non-zero initial state will oscillate for an extended period of time, as shown below:

Later, we want to be able to use the motors efficiently to bring the oscillating masses to a rest. It will look something like this:

### 5.1.2 Creating the model

As indicated above, the model block is essential for the application of do-mpc. In mathematical terms the model is defined either as a continuous ordinary differential equation (ODE), a differential algebraic equation (DAE) or a discrete equation).

In the case of a DAE/ODE we write:

\[
\frac{\partial x}{\partial t} = f(x, u, z, p) \tag{5.4}
\]
\[
0 = g(x, u, z, p) \tag{5.5}
\]
\[
y = h(x, u, z, p) \tag{5.6}
\]

We denote \(x \in \mathbb{R}^{n_x}\) as the states, \(u \in \mathbb{R}^{n_u}\) as the inputs, \(z \in \mathbb{R}^{n_z}\) the algebraic states and \(p \in \mathbb{R}^{n_p}\) as parameters.

We reformulate the second order ODEs above as the following first order ODEs, by introducing the following states:

\[
x_1 = \phi_1 \tag{5.7}
\]
\[
x_2 = \phi_2 \tag{5.8}
\]
\[
x_3 = \phi_3 \tag{5.9}
\]
\[
x_4 = \dot{\phi}_1 \tag{5.10}
\]
\[
x_5 = \dot{\phi}_2 \tag{5.11}
\]
\[
x_6 = \dot{\phi}_3 \tag{5.12}
\]

and derive the right-hand-side function \(f(x, u, z, p)\) as:

\[
\dot{x}_1 = x_4 \tag{5.14}
\]
\[
\dot{x}_2 = x_5 \tag{5.15}
\]
\[
\dot{x}_3 = x_6 \tag{5.16}
\]
\[
\dot{x}_4 = -\frac{c_1}{\Theta_1} (x_1 - u_1) - \frac{c_2}{\Theta_1} (x_1 - x_2) - \frac{d_1}{\Theta_1} x_4 \tag{5.17}
\]
\[
\dot{x}_5 = -\frac{c_2}{\Theta_2} (x_2 - x_1) - \frac{c_3}{\Theta_2} (x_2 - x_3) - \frac{d_2}{\Theta_2} x_5 \tag{5.18}
\]
\[
\dot{x}_6 = -\frac{c_3}{\Theta_3} (x_3 - x_2) - \frac{c_4}{\Theta_3} (x_4 - u_2) - \frac{d_3}{\Theta_3} x_6 \tag{5.19}
\]

5.1. Getting started: MPC
With this theoretical background we can start configuring the `do-mpc` model object. First, we need to decide on the model type. For the given example, we are working with a continuous model.

```python
model_type = 'continuous'  # either 'discrete' or 'continuous'
model = do_mpc.model.Model(model_type)
```

### 5.1.2.1 Model variables

The next step is to define the model variables. It is important to define the variable type, name and optionally shape (default is scalar variable). The following types are available:

<table>
<thead>
<tr>
<th>Long name</th>
<th>short name</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>states</td>
<td>_x</td>
<td>Required</td>
</tr>
<tr>
<td>inputs</td>
<td>_u</td>
<td>Required</td>
</tr>
<tr>
<td>algebraic</td>
<td>_z</td>
<td>Optional</td>
</tr>
<tr>
<td>parameter</td>
<td>_p</td>
<td>Optional</td>
</tr>
<tr>
<td>timevarying_parameter</td>
<td>_tvp</td>
<td>Optional</td>
</tr>
</tbody>
</table>

```python
phi_1 = model.set_variable(var_type='_x', var_name='phi_1', shape=(1,1))
phi_2 = model.set_variable(var_type='_x', var_name='phi_2', shape=(1,1))
phi_3 = model.set_variable(var_type='_x', var_name='phi_3', shape=(1,1))

# Variables can also be vectors:
dphi = model.set_variable(var_type='_x', var_name='dphi', shape=(3,1))

# Two states for the desired (set) motor position:
phi_m_1_set = model.set_variable(var_type='_u', var_name='phi_m_1_set')
phi_m_2_set = model.set_variable(var_type='_u', var_name='phi_m_2_set')

# Two additional states for the true motor position:
phi_1_m = model.set_variable(var_type='_x', var_name='phi_1_m', shape=(1,1))
phi_2_m = model.set_variable(var_type='_x', var_name='phi_2_m', shape=(1,1))
```

Note that `model.set_variable()` returns the symbolic variable:

```python
print('phi_1=\{\}, with phi_1.shape=\{\}'.format(phi_1, phi_1.shape))
print('dphi=\{\}, with dphi.shape=\{\}'.format(dphi, dphi.shape))
```

```
phi_1=phi_1, with phi_1.shape=(1, 1)
dphi=[dphi_0, dphi_1, dphi_2], with dphi.shape=(3, 1)
```

### 5.1.2.2 Query variables

If at any time you need to obtain the model variables, e.g. if you create the model in a different file than additional `do-mpc` modules, you might need to retrieve the defined variables. `do-mpc` facilitates this process with the `Model` properties `x, u, z, p, tvp, y` and `aux`:

```python
model.x
```

```python
<casadi.tools.structure3 ssymStruct at 0x7fa718a27d30>
```

The properties itself a structured symbolic variables, which hold the user-defined variables. These can be accessed with indices:

```python
model.x['phi_1']
```
Note that this is identical to the output of `model.set_variable` from above:

```python
bool(model.x['phi_1'] == phi_1)
```

Further indices are possible in the case of variables with multiple elements:

```python
model.x['dphi',0]
```

Note that you can use the following methods:

- `.keys()`
- `.labels()`

to get more information from the symbolic structures:

```python
model.x.keys()
```

```python
['phi_1', 'phi_2', 'phi_3', 'dphi', 'phi_1_m', 'phi_2_m']
```

```python
model.x.labels()
```

```python
['[phi_1,0]', '[phi_2,0]', '[phi_3,0]', '[dphi,0]', '[dphi,1]', '[dphi,2]', '[phi_1_m,0]', '[phi_2_m,0]']
```

### 5.1.2.3 Model parameters

Next we define parameters. Known values can and should be hardcoded but with robust MPC in mind, we define uncertain parameters explicitly. We assume that the inertia is such an uncertain parameter and hardcode the spring constant and friction coefficient.

```python
# As shown in the table above, we can use Long names or short names for the variable type.
Theta_1 = model.set_variable('parameter', 'Theta_1')
Theta_2 = model.set_variable('parameter', 'Theta_2')
Theta_3 = model.set_variable('parameter', 'Theta_3')
c = np.array([2.697, 2.66, 3.05, 2.86]) * 1e-3
d = np.array([6.78, 8.01, 8.82]) * 1e-5
```

### 5.1.2.4 Right-hand-side equation

Finally, we set the right-hand-side of the model by calling `model.set_rhs(var_name, expr)` with the `var_name` from the state variables defined above and an expression in terms of \( x, u, z, p \).
From the vector valued state \( \text{dphi} \) we need to concatenate symbolic expressions. We import the symbolic library CasADi:

```python
from casadi import *
```

```python
dphi_next = vertcat(
    -c[0]/Theta_1*(phi_1-phi_1_m)-c[1]/Theta_1*(phi_1-phi_2)-d[0]/Theta_1*dphi[0],
    -c[1]/Theta_2*(phi_2-phi_1)-c[2]/Theta_2*(phi_2-phi_3)-d[1]/Theta_2*dphi[1],
    -c[2]/Theta_3*(phi_3-phi_2)-c[3]/Theta_3*(phi_3-phi_2_m)-d[2]/Theta_3*dphi[2],
)
```

```python
model.set_rhs('dphi', dphi_next)
```

The model setup is completed by calling `model.setup()`:

```python
model.setup()
```

After calling `model.setup()` we cannot define further variables etc.

### 5.1.3 Configuring the MPC controller

With the configured and setup model we can now create the optimizer for model predictive control (MPC). We start by creating the object (with the `model` as the only input)

```python
mpc = do_mpc.controller.MPC(model)
```

#### 5.1.3.1 Optimizer parameters

Next, we need to parametrize the optimizer. Please see the API documentation for `optimizer.set_param()` for a full description of available parameters and their meaning. Many parameters already have suggested default values. Most importantly, we need to set `n_horizon` and `t_step`. We also choose `n_robust=1` for this example, which would default to 0.

Note that by default the continuous system is discretized with collocation.

```python
setup_mpc = {
    'n_horizon': 20,
    't_step': 0.1,
    'n_robust': 1,
    'store_full_solution': True,
}
mpc.set_param(**setup_mpc)
```
5.1.3.2 Objective function

The MPC formulation is at its core an optimization problem for which we need to define an objective function:

\[ C = \sum_{k=0}^{n-1} \left( l(x_k, u_k, z_k, p) + \Delta u_k^T R \Delta u_k \right) + m(x_n) \]

We need to define the meyer term (mterm) and lagrange term (lterm). For the given example we set:

\( l(x_k, u_k, z_k, p) = \phi_1^2 + \phi_2^2 + \phi_3^2 \)

\( m(x_n) = \phi_1^2 + \phi_2^2 + \phi_3^2 \)

[mterm = phi_1**2 + phi_2**2 + phi_3**2
lterm = phi_1**2 + phi_2**2 + phi_3**2
mpc.set_objective(mterm=mterm, lterm=lterm)]

Part of the objective function is also the penalty for the control inputs. This penalty can often be used to smoothen the obtained optimal solution and is an important tuning parameter. We add a quadratic penalty on changes:

\[ \Delta u_k = u_k - u_{k-1} \]

and automatically supply the solver with the previous solution of \( u_{k-1} \) for \( \Delta u_0 \).

The user can set the tuning factor for these quadratic terms like this:

[20]: mpc.set_rterm(
    phi_m_1_set=1e-2,
    phi_m_2_set=1e-2
)

where the keyword arguments refer to the previously defined input names. Note that in the notation above \((\Delta u_k^T R \Delta u_k)\), this results in setting the diagonal elements of \( R \).

5.1.3.3 Constraints

It is an important feature of MPC to be able to set constraints on inputs and states. In do-mpc these constraints are set like this:

[21]: 
# Lower bounds on states:
mpc.bounds['lower', '_x', 'phi_1'] = -2*np.pi
mpc.bounds['lower', '_x', 'phi_2'] = -2*np.pi
mpc.bounds['lower', '_x', 'phi_3'] = -2*np.pi
# Upper bounds on states
mpc.bounds['upper', '_x', 'phi_1'] = 2*np.pi
mpc.bounds['upper', '_x', 'phi_2'] = 2*np.pi
mpc.bounds['upper', '_x', 'phi_3'] = 2*np.pi

# Lower bounds on inputs:
mpc.bounds['lower', '_u', 'phi_m_1_set'] = -2*np.pi
mpc.bounds['lower', '_u', 'phi_m_2_set'] = -2*np.pi
# Upper bounds on inputs:
mpc.bounds['upper', '_u', 'phi_m_1_set'] = 2*np.pi
mpc.bounds['upper', '_u', 'phi_m_2_set'] = 2*np.pi
5.1.3.4 Scaling

Scaling is an important feature if the OCP is poorly conditioned, e.g. different states have significantly different magnitudes. In that case the unscaled problem might not lead to a (desired) solution. Scaling factors can be introduced for all states, inputs and algebraic variables and the objective is to scale them to roughly the same order of magnitude. For the given problem, this is not necessary but we briefly show the syntax (note that this step can also be skipped).

```python
[22]: mpc.scaling['_x', 'phi_1'] = 2
mpc.scaling['_x', 'phi_2'] = 2
mpc.scaling['_x', 'phi_3'] = 2
```

5.1.3.5 Uncertain Parameters

An important feature of do-mpc is scenario based robust MPC. Instead of predicting and controlling a single future trajectory, we investigate multiple possible trajectories depending on different uncertain parameters. These parameters were previously defined in the model (the mass inertia). Now we must provide the optimizer with different possible scenarios.

This can be done in the following way:

```python
[23]: inertia_mass_1 = 2.25*1e-4*np.array([1., 0.9, 1.1])
inertia_mass_2 = 2.25*1e-4*np.array([1., 0.9, 1.1])
inertia_mass_3 = 2.25*1e-4*np.array([1.])

mpc.set_uncertainty_values(
    Theta_1 = inertia_mass_1,
    Theta_2 = inertia_mass_2,
    Theta_3 = inertia_mass_3
)
```

We provide a number of keyword arguments to the method optimizer.set_uncertain_parameter(). For each referenced parameter the value is a numpy.ndarray with a selection of possible values. The first value is the nominal case, where further values will lead to an increasing number of scenarios. Since we investigate each combination of possible parameters, the number of scenarios is growing rapidly. For our example, we are therefore only treating the inertia of mass 1 and 2 as uncertain and supply only one possible value for the mass of inertia 3.

5.1.3.6 Setup

The last step of configuring the optimizer is to call optimizer.setup, which finalizes the setup and creates the optimization problem. Only now can we use the optimizer to obtain the control input.

```python
[24]: mpc.setup()
```

5.1.4 Configuring the Simulator

In many cases a developed control approach is first tested on a simulated system. do-mpc responds to this need with the do_mpc.simulator class. The simulator uses state-of-the-art DAE solvers, e.g. Sundials CVODE to solve the DAE equations defined in the supplied do_mpc.model. This will often be the same model as defined for the optimizer but it is also possible to use a more complex model of the same system.

In this section we demonstrate how to setup the simulator class for the given example. We initilize the class with the previously defined model:
5.1.4.1 Simulator parameters

Next, we need to parametrize the simulator. Please see the API documentation for simulator.set_param() for a full description of available parameters and their meaning. Many parameters already have suggested default values. Most importantly, we need to set t_step. We choose the same value as for the optimizer.

```python
# Instead of supplying a dict with the splat operator (**), as with the optimizer.set_˓→param(),
# we can also use keywords (and call the method multiple times, if necessary):
simulator.set_param(t_step = 0.1)
```

5.1.4.2 Uncertain parameters

In the model we have defined the inertia of the masses as parameters, for which we have chosen multiple scenarios in the optimizer. The simulator is now parametrized to simulate with the “true” values at each timestep. In the most general case, these values can change, which is why we need to supply a function that can be evaluated at each time to obtain the current values. do-mpc requires this function to have a specific return structure which we obtain first by calling:

```python
p_template = simulator.get_p_template()
```

This object is a CasADi structure:

```python
type(p_template)
casadi.tools.structure3.DMStruct
```

which can be indexed with the following keys:

```python
p_template.keys()
['default', 'Theta_1', 'Theta_2', 'Theta_3']
```

We need to now write a function which returns this structure with the desired numerical values. For our simple case:

```python
def p_fun(t_now):
    p_template['Theta_1'] = 2.25e-4
    p_template['Theta_2'] = 2.25e-4
    p_template['Theta_3'] = 2.25e-4
    return p_template
```

This function is now supplied to the simulator in the following way:

```python
simulator.set_p_fun(p_fun)
```

5.1.4.3 Setup

Similarly to the optimizer we need to call simulator.setup() to finalize the setup of the simulator.

```python
simulator.setup()
```
5.1.5 Creating the control loop

In theory, we could now also create an estimator but for this concise example we just assume direct state-feedback. This means we are now ready to setup and run the control loop. The control loop consists of running the optimizer with the current state ($x_0$) to obtain the current control input ($u_0$) and then running the simulator with the current control input ($u_0$) to obtain the next state.

As discussed before, we setup a controller for regulating a triple-mass-spring system. To show some interesting control action we choose an arbitrary initial state $x_0 \neq 0$:

```python
x0 = np.pi*np.array([1, 1, -1.5, 1, -1, 1, 0, 0]).reshape(-1,1)
```

and use the $x0$ property to set the initial state.

```python
simulator.x0 = x0
mpc.x0 = x0
```

While we are able to set just a regular numpy array, this populates the state structure which was inherited from the model:

```python
mpc.x0
```

```python
<casadi.tools.structure3.DMStruct at 0x7fa71b5ee390>
```

We can thus easily obtain the value of particular states by calling:

```python
mpc.x0['phi_1']
```

```python
DM(3.14159)
```

Note that the properties $x0$ (as well as $u0$, $z0$ and $t0$) always display the values of the current variables in the class.

To set the initial guess of the MPC optimization problem we call:

```python
mpc.set_initial_guess()
```

The chosen initial guess is based on $x0$, $z0$ and $u0$ which are set for each element of the MPC sequence.

5.1.5.1 Setting up the Graphic

To investigate the controller performance **AND** the MPC predictions, we are using the do-mpc graphics module. This versatile tool allows us to conveniently configure a user-defined plot based on Matplotlib and visualize the results stored in the `mpc.data`, `simulator.data` (and if applicable `estimator.data`) objects.

We start by importing matplotlib:

```python
import matplotlib.pyplot as plt
import matplotlib as mpl
# Customizing Matplotlib:
mpl.rcParams['font.size'] = 18
mpl.rcParams['lines.linewidth'] = 3
mpl.rcParams['axes.grid'] = True
```

And initializing the graphics module with the data object of interest. In this particular example, we want to visualize both the `mpc.data as well as the `simulator.data`.

```python
mpc_graphics = do_mpc.graphics.Graphics(mpc.data)
sim_graphics = do_mpc.graphics.Graphics(simulator.data)
```
Next, we create a figure and obtain its axis object. Matplotlib offers multiple alternative ways to obtain an axis object, e.g. subplots, subplot2grid, or simply gca. We use subplots:

```python
# We just want to create the plot and not show it right now. This "inline magic"
# suppresses the output.
fig, ax = plt.subplots(2, sharex=True, figsize=(16,9))
fig.align_ylabels()
```

Most important API element for setting up the graphics module is `graphics.add_line`, which mimics the API of `model.add_variable`, except that we also need to pass an axis.

We want to show both the simulator and MPC results on the same axis, which is why we configure both of them identically:

```python
for g in [sim_graphics, mpc_graphics]:
    # Plot the angle positions (phi_1, phi_2, phi_2) on the first axis:
    g.add_line(var_type='_x', var_name='phi_1', axis=ax[0])
    g.add_line(var_type='_x', var_name='phi_2', axis=ax[0])
    g.add_line(var_type='_x', var_name='phi_3', axis=ax[0])

    # Plot the set motor positions (phi_m_1_set, phi_m_2_set) on the second axis:
    g.add_line(var_type='_u', var_name='phi_m_1_set', axis=ax[1])
    g.add_line(var_type='_u', var_name='phi_m_2_set', axis=ax[1])
```

![Figure](image)

**5.1.5.2 Running the simulator**

We start investigating the do-mpc simulator and the graphics package by simulating the autonomous system without control inputs ($u = 0$). This can be done as follows:

```python
u0 = np.zeros((2,1))
for i in range(200):
    simulator.make_step(u0)
```

We can visualize the resulting trajectory with the pre-defined graphic:

```python
sim_graphics.plot_results()
# Reset the limits on all axes in graphic to show the data.
sim_graphics.reset_axes()
# Show the figure:
fig
```
As desired, the motor angle (input) is constant at zero and the oscillating masses slowly come to a rest. Our control goal is to significantly shorten the time until the discs are stationary.

Remember the animation you saw above, of the uncontrolled system? This is where the data came from.

5.1.5.3 Running the optimizer

To obtain the current control input we call `optimizer.make_step(x0)` with the current state \( x_0 \):

\[
\text{u0 = mpc.make_step(x0)}
\]
Note that we obtained the output from IPOPT regarding the given optimal control problem (OCP). Most importantly we obtained Optimal Solution Found.

We can also visualize the predicted trajectories with the configure graphics instance. First we clear the existing lines from the simulator by calling:

```
[45]: sim_graphics.clear()
```

And finally, we can call plot_predictions to obtain:

5.1. Getting started: MPC
We are seeing the predicted trajectories for the states and the optimal control inputs. Note that we are seeing different scenarios for the configured uncertain inertia of the three masses.

We can also see that the solution is considering the defined upper and lower bounds. This is especially true for the inputs.

### 5.1.5.4 Changing the line appearance

Before we continue, we should probably improve the visualization a bit. We can easily obtain all line objects from the graphics module by using the `result_lines` and `pred_lines` properties:

```python
[47]: mpc_graphics.pred_lines
```

```python
[47]: <do_mpc.tools.structure.Structure at 0x7fa71b5ee860>
```

We obtain a structure that can be queried conveniently as follows:

```python
[48]: mpc_graphics.pred_lines['_x', 'phi_1']
```

```python
[48]: [<matplotlib.lines.Line2D at 0x7fa71c445828>,
     <matplotlib.lines.Line2D at 0x7fa71c6e7898>,
     <matplotlib.lines.Line2D at 0x7fa71c6e7978>,
     <matplotlib.lines.Line2D at 0x7fa71c7023c8>,
     <matplotlib.lines.Line2D at 0x7fa71c7023c8>,
     <matplotlib.lines.Line2D at 0x7fa71c7023c8>,
     <matplotlib.lines.Line2D at 0x7fa71c7022b0>,
     <matplotlib.lines.Line2D at 0x7fa71c7022b0>,
     <matplotlib.lines.Line2D at 0x7fa71c702630>,
     <matplotlib.lines.Line2D at 0x7fa71c702630>,
     <matplotlib.lines.Line2D at 0x7fa71c702798>,
     <matplotlib.lines.Line2D at 0x7fa71c702798>,
     <matplotlib.lines.Line2D at 0x7fa71c702c88>]
```
We obtain all lines for our first state. To change the color we can simply:

```python
# Change the color for the three states:
for line_i in mpc_graphics.pred_lines['_x', 'phi_1']:
    line_i.set_color('#1f77b4') # blue
for line_i in mpc_graphics.pred_lines['_x', 'phi_2']:
    line_i.set_color('#ff7f0e') # orange
for line_i in mpc_graphics.pred_lines['_x', 'phi_3']:
    line_i.set_color('#2ca02c') # green
# Change the color for the two inputs:
for line_i in mpc_graphics.pred_lines['_u', 'phi_m_1_set']:
    line_i.set_color('#1f77b4') # blue
for line_i in mpc_graphics.pred_lines['_u', 'phi_m_2_set']:
    line_i.set_color('#ff7f0e') # orange
# Make all predictions transparent:
for line_i in mpc_graphics.pred_lines.full:
    line_i.set_alpha(0.2)
```

Note that we can work in the same way with the `result_lines` property. For example, we can use it to create a legend:

```python
# Get line objects (note sum of lists creates a concatenated list)
lines = sim_graphics.result_lines['_x', 'phi_1'] + sim_graphics.result_lines['_x', 'phi_2'] + sim_graphics.result_lines['_x', 'phi_3']
ax[0].legend(lines, '123', title='disc')
# also set legend for second subplot:
lines = sim_graphics.result_lines['_u', 'phi_m_1_set'] + sim_graphics.result_lines['_u', 'phi_m_2_set']
ax[1].legend(lines, '12', title='motor')
```

### 5.1.5.5 Running the control loop

Finally, we are now able to run the control loop as discussed above. We obtain the input from the optimizer and then run the simulator.

To make sure we start fresh, we erase the history and set the initial state for the simulator:

```python
simulator.reset_history()
simulator.x0 = x0
mpc.reset_history()
```

This is the main-loop. We run 20 steps, which is identical to the prediction horizon. Note that we use “capture” again, to suppress the output from IPOPT.

It is usually suggested to display the output as it contains important information about the state of the solver.

```python
for i in range(20):
    u0 = mpc.make_step(x0)
    x0 = simulator.make_step(u0)
```

We can now plot the previously shown prediction from time $t = 0$, as well as the closed-loop trajectory from the simulator:
The simulated trajectory with the nominal value of the parameters follows almost exactly the nominal open-loop predictions. The simulated trajectory is bounded from above and below by further uncertain scenarios.

### 5.1.6 Data processing

#### 5.1.6.1 Saving and retrieving results

do-mpc results can be stored and retrieved with the methods `save_results` and `load_results` from the `do_mpc.data` module. We start by importing these methods:

```python
from do_mpc.data import save_results, load_results
```

The method `save_results` is passed a list of the do-mpc objects that we want to store. In our case, the optimizer and simulator are available and configured.

Note that by default results are stored in the subfolder `results` under the name `results.pkl`. Both can be changed and the folder is created if it doesn’t exist already.

```python
save_results([mpc, simulator])
```

We investigate the content of the newly created folder:

```bash
!ls ./results/
```
Automatically, the `save_results` call will check if a file with the given name already exists. To avoid overwriting, the method prepends an index. If we save again, the folder contains:

```
[58]: save_results([mpc, simulator])
!ls ./results/
001_results.pkl results.pkl
```

The pickled results can be loaded manually by writing:

```
with open(file_name, 'rb') as f:
    results = pickle.load(f)
```

or by calling `load_results` with the appropriate `file_name` (and path). `load_results` contains simply the code above for more convenience.

```
[59]: results = load_results('./results/results.pkl')
```

The obtained `results` is a dictionary with the data objects from the passed `do-mpc` modules. Such that: `results['optimizer']` and `optimizer.data` contain the same information (similarly for `simulator` and, if applicable, `estimator`).

### 5.1.6.2 Working with data objects

The `do_mpc.data.Data` objects also hold some very useful properties that you should know about. Most importantly, we can query them with indices, such as:

```
[60]: results['mpc']
[60]: <do_mpc.data.MPCData at 0x117c4df50>

[61]: x = results['mpc']['_x']
x.shape
[61]: (20, 8)
```

As expected, we have 20 elements (we ran the loop for 20 steps) and 8 states. Further indices allow to get selected states:

```
[62]: phi_1 = results['mpc']['_x','phi_1']
phi_1.shape
[62]: (20, 1)
```

For vector-valued states we can even query:

```
[63]: dphi_1 = results['mpc']['_x','dphi', 0]
dphi_1.shape
[63]: (20, 1)
```

Of course, we could also query inputs etc.
Furthermore, we can easily retrieve the predicted trajectories with the prediction method. The syntax is slightly different: The first argument is a tuple that mimics the indices shown above. The second index is the time instance. With the following call we obtain the prediction of \( \phi_1 \) at time 0:

```python
[64]: phi_1_pred = results['mpc'].prediction(('_x','phi_1'), t_ind=0)
```

```
phi_1_pred.shape
```

```
[64]: (1, 21, 9)
```

The first dimension shows that this state is a scalar, the second dimension shows the horizon and the third dimension refers to the nine uncertain scenarios that were investigated.

### 5.1.6.3 Animating results

Animating MPC results, to compare prediction and closed-loop trajectories, allows for a very meaningful investigation of the obtained results. **do-mpc** significantly facilitates this process while working hand in hand with Matplotlib for full customizability. Obtaining publication ready animations is as easy as writing the following short blocks of code:

```python
[66]: from matplotlib.animation import FuncAnimation, FFMpegWriter, ImageMagickWriter
def update(t_ind):
    sim_graphics.plot_results(t_ind)
    mpc_graphics.plot_predictions(t_ind)
    mpc_graphics.reset_axes()
```

The `graphics` module can also be used without restrictions with loaded **do-mpc** data. This allows for convenient data post-processing, e.g. in a Jupyter Notebook. We simply would have to initiate the `graphics` module with the loaded results from above.

```python
[69]: anim = FuncAnimation(fig, update, frames=20, repeat=False)
gif_writer = ImageMagickWriter(fps=3)
anim.save('anim.gif', writer=gif_writer)
```

Below we showcase the resulting gif file (not in real-time):

Thank you, for following through this short example on how to use **do-mpc**. We hope you find the tool and this documentation useful.

We suggest that you have a look at the API documentation for further details on the presented modules, methods and functions.

We also want to emphasize that we skipped over many details, further functions etc. in this introduction. Please have a look at our more complex examples to get a better impression of the possibilities with **do-mpc**.

### 5.2 Getting started: MHE

Open an interactive online Jupyter Notebook with this content on Binder:

In this Jupyter Notebook we illustrate application of the **do-mpc** moving horizon estimation module. Please follow first the general Getting Started guide, as we cover the sample example and skip over some previously explained details.
```python
import numpy as np
from casadi import *

# Add do_mpc to path. This is not necessary if it was installed via pip.
import sys
sys.path.append('..')

# Import do_mpc package:
import do_mpc
```

### 5.2.1 Creating the model

First, we need to decide on the model type. For the given example, we are working with a continuous model.

```python
model_type = 'continuous'  # either 'discrete' or 'continuous'
model = do_mpc.model.Model(model_type)
```

The model is based on the assumption that we have additive process and/or measurement noise:

\[
\dot{x}(t) = f(x(t), u(t), z(t), p(t), p_{tv}(t)) + w(t), \quad (5.21)
\]

\[
y(t) = h(x(t), u(t), z(t), p(t), p_{tv}(t)) + v(t), \quad (5.22)
\]

we are free to chose, which states and which measurements experience additive noise.

#### 5.2.1.1 Model variables

The next step is to define the model variables. It is important to define the variable type, name and optionally shape (default is scalar variable).

In contrast to the previous example, we now use vectors for all variables.

```python
phi = model.set_variable(var_type='_x', var_name='phi', shape=(3,1))
dphi = model.set_variable(var_type='_x', var_name='dphi', shape=(3,1))

# Two states for the desired (set) motor position:
phi_m_set = model.set_variable(var_type='_u', var_name='phi_m_set', shape=(2,1))

# Two additional states for the true motor position:
phi_m = model.set_variable(var_type='_x', var_name='phi_m', shape=(2,1))
```

#### 5.2.1.2 Model measurements

This step is essential for the state estimation task: We must define a measurable output. Typically, this is a subset of states (or a transformation thereof) as well as the inputs.

Note that some MHE implementations consider inputs separately.

As mentioned above, we need to define for each measurement if additive noise is present. In our case we assume noisy state measurements (\(\phi\)) but perfect input measurements.

```python
# State measurements
phi_meas = model.set_meas('phi_1_meas', phi, meas_noise=True)

# Input measurements
phi_m_set_meas = model.set_meas('phi_m_set_meas', phi_m_set, meas_noise=False)
```

---

5.2. Getting started: MHE
5.2.1.3 Model parameters

Next we define parameters. The MHE allows to estimate parameters as well as states. Note that not all parameters must be estimated (as shown in the MHE setup below). We can also hardcode parameters (such as the spring constants $c$).

[[5]]: Theta_1 = model.set_variable('parameter', 'Theta_1')
Theta_2 = model.set_variable('parameter', 'Theta_2')
Theta_3 = model.set_variable('parameter', 'Theta_3')

c = np.array([2.697, 2.66, 3.05, 2.86]) * 1e-3
d = np.array([6.78, 8.01, 8.82]) * 1e-5

5.2.1.4 Right-hand-side equation

Finally, we set the right-hand-side of the model by calling model.set_rhs(var_name, expr) with the var_name from the state variables defined above and an expression in terms of $x, u, z, p$.

Note that we can decide whether the individual states experience process noise. In this example we choose that the system model is perfect. This is the default setting, so we don’t need to pass this parameter explicitly.

[[6]]: model.set_rhs('phi', dphi)

dphi_next = vertcat(
    -c[0]/Theta_1*(phi[0]-phi_m[0])-c[1]/Theta_1*(phi[0]-phi[1])-d[0]/Theta_1*dphi[0],
    -c[1]/Theta_2*(phi[1]-phi[0])-c[2]/Theta_2*(phi[1]-phi[2])-d[1]/Theta_2*dphi[1],
    -c[2]/Theta_3*(phi[2]-phi[1])-c[3]/Theta_3*(phi[2]-phi_m[1])-d[2]/Theta_3*dphi[2],
)
model.set_rhs('dphi', dphi_next, process_noise = False)

tau = 1e-2
model.set_rhs('phi_m', 1/tau*(phi_m_set - phi_m))

The model setup is completed by calling model.setup():

[[7]]: model.setup()

After calling model.setup() we cannot define further variables etc.

5.2.2 Configuring the moving horizon estimator

The first step of configuring the moving horizon estimator is to call the class with a list of all parameters to be estimated. An empty list (default value) means that no parameters are estimated. The list of estimated parameters must be a subset (or all) of the previously defined parameters.

Note

So why did we define Theta_2 and Theta_3 if we do not estimate them?

In many cases we will use the same model for (robust) control and MHE estimation. In that case it is possible to have some external parameters (e.g. weather prediction) that are uncertain but cannot be estimated.
5.2.2.1 MHE parameters:

Next, we pass MHE parameters. Most importantly, we need to set the time step and the horizon. We also choose to obtain the measurement from the MHE data object. Alternatively, we are able to set a user defined measurement function that is called at each timestep and returns the $N$ previous measurements for the estimation step.

```
setup_mhe = {
    't_step': 0.1,
    'n_horizon': 10,
    'store_full_solution': True,
    'meas_from_data': True
}
mhe.set_param(**setup_mhe)
```

5.2.2.2 Objective function

The most important step of the configuration is to define the objective function for the MHE problem:

$$
\begin{align}
\min_{x_{0:N+1}, u_{0:N}, p, w_{0:N}} & \frac{1}{2} ||x_0 - \tilde{x}_0||_P^2 + \frac{1}{2} ||p - \tilde{p}||_P^2 + \sum_{k=0}^{N-1} \left( \frac{1}{2} ||v_k||_{P_v,k}^2 + \frac{1}{2} ||w_k||_{P_w,k}^2 \right), \\
\text{s.t.} & \quad x_{k+1} = f(x_k, u_k, z_k, p, p_{tv,k}) + w_k, \\
& \quad y_k = h(x_k, u_k, z_k, p, p_{tv,k}) + v_k, \\
& \quad g(x_k, u_k, z_k, p, p_{tv,k}) \leq 0, \\
& \quad k = 0, \ldots, N
\end{align}
$$

We typically consider the formulation shown above, where the user has to pass the weighting matrices $P_x$, $P_v$, $P_p$ and $P_w$. In our concrete example, we assume a perfect model without process noise and thus $P_w$ is not required.

We set the objective function with the weighting matrices shown below:

```
P_v = np.diag(np.array([1,1,1]))
P_x = np.eye(8)
P_p = 10*np.eye(1)
mhe.set_default_objective(P_x, P_v, P_p)
```

5.2.2.3 Fixed parameters

If the model contains parameters and if we estimate only a subset of these parameters, it is required to pass a function that returns the value of the remaining parameters at each time step.

Furthermore, this function must return a specific structure, which is first obtained by calling:

```
p_template_mhe = mhe.get_p_template()
```

Using this structure, we then formulate the following function for the remaining (not estimated) parameters:
```python
[12]: `def` p_fun_mhe(t_now):
    p_template_mhe['Theta_2'] = 2.25e-4
    p_template_mhe['Theta_3'] = 2.25e-4
    `return` p_template_mhe

This function is finally passed to the mhe instance:

[13]: mhe.set_p_fun(p_fun_mhe)

5.2.2.4 Bounds

The MHE implementation also supports bounds for states, inputs, parameters which can be set as shown below. For the given example, it is especially important to set realistic bounds on the estimated parameter. Otherwise the MHE solution is a poor fit.

[14]: mhe.bounds['lower','_u', 'phi_m_set'] = -2*np.pi
    mhe.bounds['upper','_u', 'phi_m_set'] = 2*np.pi
    mhe.bounds['lower','_p_est', 'Theta_1'] = 1e-5
    mhe.bounds['upper','_p_est', 'Theta_1'] = 1e-3

5.2.2.5 Setup

Similar to the controller, simulator and model, we finalize the MHE configuration by calling:

[15]: mhe.setup()

5.2.3 Configuring the Simulator

In many cases, a developed control approach is first tested on a simulated system. do-mpc responds to this need with the do_mpc.simulator class. The simulator uses state-of-the-art DAE solvers, e.g. Sundials CVODE to solve the DAE equations defined in the supplied do_mpc.model. This will often be the same model as defined for the optimizer but it is also possible to use a more complex model of the same system.

In this section we demonstrate how to setup the simulator class for the given example. We initialize the class with the previously defined model:

[16]: simulator = do_mpc.simulator.Simulator(model)

5.2.3.1 Simulator parameters

Next, we need to parametrize the simulator. Please see the API documentation for simulator.set_param() for a full description of available parameters and their meaning. Many parameters already have suggested default values. Most importantly, we need to set `t_step`. We choose the same value as for the optimizer.

[17]: `#` Instead of supplying a dict with the splat operator (**), as with the optimizer.set_ ...
    `#` we can also use keywords (and call the method multiple times, if necessary):
    simulator.set_param(t_step = 0.1)
5.2.3.2 Parameters

In the model we have defined the inertia of the masses as parameters. The simulator is now parametrized to simulate using the “true” values at each timestep. In the most general case, these values can change, which is why we need to supply a function that can be evaluated at each time to obtain the current values. do-mpc requires this function to have a specific return structure which we obtain first by calling:

```python
[18]: p_template_sim = simulator.get_p_template()
```

We need to define a function which returns this structure with the desired numerical values. For our simple case:

```python
[19]: def p_fun_sim(t_now):
    p_template_sim['Theta_1'] = 2.25e-4
    p_template_sim['Theta_2'] = 2.25e-4
    p_template_sim['Theta_3'] = 2.25e-4
    return p_template_sim
```

This function is now supplied to the simulator in the following way:

```python
[20]: simulator.set_p_fun(p_fun_sim)
```

5.2.3.3 Setup

Finally, we call:

```python
[21]: simulator.setup()
```

5.2.4 Creating the loop

While the full loop should also include a controller, we are currently only interested in showcasing the estimator. We therefore estimate the states for an arbitrary initial condition and some random control inputs (shown below).

```python
[22]: x0 = np.pi*np.array([1, 1, -1.5, 1, -5, 5, 0, 0]).reshape(-1,1)
```

To make things more interesting we pass the estimator a perturbed initial state:

```python
[23]: x0_mhe = x0*(1+0.5*np.random.randn(8,1))
```

and use the `x0` property of the simulator and estimator to set the initial state:

```python
[24]: simulator.x0 = x0
    mhe.x0_mhe = x0_mhe
    mhe.p_est0 = 1e-4
```

It is also advised to create an initial guess for the MHE optimization problem. The simplest way is to base that guess on the initial state, which is done automatically when calling:

```python
[25]: mhe.set_initial_guess()
```

5.2.4.1 Setting up the Graphic

We are again using the do-mpc graphics module. This versatile tool allows us to conveniently configure a user-defined plot based on Matplotlib and visualize the results stored in the `mhe.data, simulator.data` objects.
We start by importing matplotlib:

```
[26]: import matplotlib.pyplot as plt
import matplotlib as mpl
# Customizing Matplotlib:
mpl.rcParams['font.size'] = 18
mpl.rcParams['lines.linewidth'] = 3
mpl.rcParams['axes.grid'] = True
```

And initializing the graphics module with the data object of interest. In this particular example, we want to visualize both the mpc.data as well as the simulator.data.

```
[27]: mhe_graphics = do_mpc.graphics.Graphics(mhe.data)
sim_graphics = do_mpc.graphics.Graphics(simulator.data)
```

Next, we create a figure and obtain its axis object. Matplotlib offers multiple alternative ways to obtain an axis object, e.g. subplots, subplot2grid, or simply gca. We use subplots:

```
[28]: %%capture
    fig, ax = plt.subplots(3, sharex=True, figsize=(16,9))
    fig.align_ylabels()
    # We create another figure to plot the parameters:
    fig_p, ax_p = plt.subplots(1, figsize=(16,4))
```

Most important API element for setting up the graphics module is `graphics.add_line`, which mimics the API of `model.add_variable`, except that we also need to pass an axis.

We want to show both the simulator and MHE results on the same axis, which is why we configure both of them identically:

```
[29]: %%capture
    for g in [sim_graphics, mhe_graphics]:
        # Plot the angle positions (phi_1, phi_2, phi_2) on the first axis:
        g.add_line(var_type='_x', var_name='phi', axis=ax[0])
        ax[0].set_prop_cycle(None)
        g.add_line(var_type='_x', var_name='dphi', axis=ax[1])
        ax[1].set_prop_cycle(None)

        # Plot the set motor positions (phi_m_1_set, phi_m_2_set) on the second axis:
        g.add_line(var_type='_u', var_name='phi_m_set', axis=ax[2])
        ax[2].set_prop_cycle(None)

        g.add_line(var_type='p', var_name='Theta_1', axis=ax_p)

    ax[0].set_ylabel('angle position [rad]
    ax[1].set_ylabel('angular velocity [rad/s])
    ax[2].set_ylabel('motor angle [rad]
    ax[2].set_xlabel('time [s])
```

Before we show any results we configure we further configure the graphic, by changing the appearance of the simulated lines. We can obtain line objects from any graphics instance with the `result_lines` property:

```
[30]: sim_graphics.result_lines
```
We obtain a structure that can be queried conveniently as follows:

```
# First element for state phi:
sim_graphics.result_lines['_x', 'phi', 0]
```

In this particular case we want to change all `result_lines` with:

```
for line_i in sim_graphics.result_lines.full:
    line_i.set_alpha(0.4)
    line_i.set_linewidth(6)
```

We furthermore use this property to create a legend:

```
ax[0].legend(sim_graphics.result_lines['_x', 'phi'], '123', title='Sim.', loc='center_right')
ax[1].legend(mhe_graphics.result_lines['_x', 'phi'], '123', title='MHE', loc='center_right')
```

and another legend for the parameter plot:

```
ax_p.legend(sim_graphics.result_lines['_p', 'Theta_1'] + mhe_graphics.result_lines['_p', 'Theta_1'], ['True', 'Estim.'])
```

5.2.4.2 Running the loop

We investigate the closed-loop MHE performance by alternating a simulation step (`y0=simulator.make_step(u0)`) and an estimation step (`x0=mhe.make_step(y0)`). Since we are lacking the controller which would close the loop (`u0=mpc.make_step(x0)`), we define a random control input function:

```
def random_u(u0):
    # Hold the current value with 80% chance or switch to new random value.
    u_next = (0.5-np.random.rand(2,1))*np.pi # New candidate value.
    switch = np.random.rand() >= 0.8 # switching? 0 or 1.
    u0 = (1-switch)*u0 + switch*u_next # Old or new value.
    return u0
```

The function holds the current input value with 80% chance or switches to a new random input value.

We can now run the loop. At each iteration, we perturb our measurements, for a more realistic scenario. This can be done by calling the simulator with a value for the measurement noise, which we defined in the model above.

```
np.random.seed(999) #make it repeatable
u0 = np.zeros((2,1))
for i in range(50):
    u0 = random_u(u0) # Control input
    v0 = 0.1*np.random.randn(model.n_v,1) # measurement noise
    y0 = simulator.make_step(u0, v0=v0) # MHE estimation step
```
We can visualize the resulting trajectory with the pre-defined graphic:

```python
[37]: sim_graphics.plot_results()
mhe_graphics.plot_results()
# Reset the limits on all axes in graphic to show the data.
mhe_graphics.reset_axes()

# Mark the time after a full horizon is available to the MHE.
ax[0].axvline(1)
ax[1].axvline(1)
ax[2].axvline(1)

# Show the figure:
fig

[37]:
```

Parameter estimation:

```python
[38]: ax_p.set_ylim(1e-4, 4e-4)
ax_p.set_ylabel('mass inertia')
ax_p.set_xlabel('time [s]')
fig_p

[38]:
```
5.2.5 MHE Advantages

One of the main advantages of moving horizon estimation is the possibility to set bounds for states, inputs and estimated parameters. As mentioned above, this is crucial in the presented example. Let’s see how the MHE behaves without realistic bounds for the estimated mass inertia of disc one.

We simply reconfigure the bounds:

```python
mhe.bounds['lower','_p_est', 'Theta_1'] = -np.inf
mhe.bounds['upper','_p_est', 'Theta_1'] = np.inf
```

And setup the MHE again. The backend is now recreating the optimization problem, taking into consideration the currently saved bounds.

```python
mhe.setup()
```

We reset the history of the estimator and simulator (to clear their data objects and start “fresh”).

```python
mhe.reset_history()
simulator.reset_history()
```

Finally, we run the exact same loop again obtaining new results.

```python
%%capture
np.random.seed(999)  # make it repeatable
u0 = np.zeros((2,1))
for i in range(50):
    u0 = random_u(u0)  # Control input
    v0 = 0.1*np.random.randn(model.n_v,1)  # measurement noise
    y0 = simulator.make_step(u0, v0=v0)
    x0 = mhe.make_step(y0)  # MHE estimation step
```

These results now look quite terrible:

```python
sim_graphics.plot_results()
mhe_graphics.plot_results()
# Reset the limits on all axes in graphic to show the data.
mhe_graphics.reset_axes()

# Mark the time after a full horizon is available to the MHE.
ax[0].axvline(1)
ax[1].axvline(1)
ax[2].axvline(1)

# Show the figure:
fig
```
Clearly, the main problem is a faulty parameter estimation, which is off by orders of magnitude:

```
ax_p.set_ylabel('mass inertia')
ax_p.set_xlabel('time [s]')
fig_p
```

Thank you, for following through this short example on how to use do-mpc. We hope you find the tool and this documentation useful.

We also want to emphasize that we skipped over many details, further functions etc. in this introduction. Please have a look at our more complex examples to get a better impression of the possibilities with do-mpc.

### 5.3 Orthogonal collocation on finite elements

A dynamic system model is at the core of all model predictive control (MPC) and moving horizon estimation (MHE) formulations. This model allows to predict and optimize the future behavior of the system (MPC) or establishes the relationship between past measurements and estimated states (MHE).
When working with do-mpc an essential question is whether a **discrete** or **continuous** model is supplied. The discrete time formulation:

\[
x_{k+1} = f(x_k, u_k, z_k, p_{tv,k}, p),
\]

\[
0 = g(x_k, u_k, z_k, p_{tv,k}, p),
\]

gives an explicit relationship for the future states \(x_{k+1}\) based on the current states \(x_k\), inputs \(u_k\), algebraic states \(z_k\) and further parameters \(p, p_{tv,k}\). It can be evaluated in a straight-forward fashion to recursively obtain the future states of the system, based on an initial state \(x_0\) and a sequence of inputs.

However, many dynamic model equations are given in the continuous time form as ordinary differential equations (ODE) or differential algebraic equations (DAE):

\[
\dot{x} = f(x(t), u(t), z(t), p_{tv}(t), p(t)),
\]

\[
0 = g(x(t), u(t), z(t), p_{tv}(t), p(t)).
\]

Incorporating the ODE/DAE is typically less straight-forward than their discrete-time counterparts and a variety of methods are applicable. An (incomplete!) overview and classification of commonly used methods is shown in the diagram below:

**do-mpc** is based on **orthogonal collocation on finite elements** which is a direct, simultaneous, full discretization approach.

- **Direct**: The continuous time variables are discretized to transform the infinite-dimensional optimal control problem to a finite dimensional nonlinear programming (NLP) problem.
- **Simultaneous**: Both the control inputs and the states are discretized.
- **Full discretization**: A discretization scheme is hand implemented in terms of symbolic variables instead of using an ODE/DAE solver.

The full discretization is realized with **orthogonal collocation on finite elements** which is discussed in the remainder of this post. The content is based on [Biegler2010].

### 5.3.1 Lagrange polynomials for ODEs

To simplify things, we now consider the following ODE:

\[
\dot{x} = f(x), \quad x(0) = x_0,
\]

Fundamental for orthogonal collocation is the idea that the solution of the ODE \(x(t)\) can be approximated accurately with a polynomial of order \(K + 1\):

\[
x^K_i(t) = \alpha_0 + \alpha_1 t + \cdots + \alpha_K t^K.
\]

This approximation should be valid on small time-intervals \(t \in [t_i, t_{i+1}]\), which are the **finite elements** mentioned in the title.

The interpolation is based on \(j = 0, \ldots, K\) interpolation points \((t_j, x_i,j)\) in the interval \([t_i, t_{i+1}]\). We are using the
Fig. 1: Approaching an ODE/DAE continuous model for MPC or MHE.
Lagrange interpolation polynomial:

\[ x^K_i(t) = \sum_{j=0}^{K} L_j(\tau) x_{i,j} \]

where:

\[ L_j(\tau) = \prod_{k=0}^{K} \frac{(\tau - \tau_k)}{(\tau_j - \tau_k)} \]

\[ = \frac{t - t_i}{\Delta t_i}, \quad \Delta t_i = t_{i+1} - t_i. \]

We call \( L_j(\tau) \) the Lagrangian basis polynomial with the dimensionless time \( \tau \in [0, 1] \). Note that the basis polynomial \( L_j(\tau) \) is constructed to be \( L_j(\tau_j) = 1 \) and \( L_j(\tau_i) = 0 \) for all other interpolation points \( i \neq j \).

This polynomial ensures that for the interpolation points \( x^K(t_{i,j}) = x_{i,j} \). Such a polynomial is fitted to all finite elements, as shown in the figure below.

Fig. 2: Lagrange polynomials representing the solution of an ODE on neighboring finite elements.

Note that the collocation points (round circles above) can be chosen freely while obeying \( \tau_0 = 0 \) and \( \tau_j < \tau_{j+1} \leq 1 \). There are, however, better choices than others which will be discussed in Collocation with orthogonal polynomials.

5.3.2 Deriving the integration equations

So far we have seen how to approximate an ODE solution with Lagrange polynomials given a set of values from the solution. This may seem confusing because we are looking for these values in the first place. However, it still helps us because we can now state conditions based on this polynomial representation that must hold for the desired solution:

\[ \frac{dx^K_i}{dt} \bigg|_{t_{i,k}} = f(x_{i,k}), \quad k = 1, \ldots, K. \]

This means that the time derivatives from our polynomial approximation evaluated at the collocation points must be equal to the original ODE at these same points.

Because we assumed a polynomial structure of \( x^K_i(t) \) the time derivative can be conveniently expressed as:

\[ \frac{dx^K_i}{dt} \bigg|_{t_{i,k}} = \sum_{j=0}^{K} x_{i,j} \frac{dL_j}{d\tau} \bigg|_{\tau \tau_{j,k}}, \]

for which we substituted \( t \) with \( \tau \). It is important to notice that for fixed collocation points the terms \( a_{j,k} \) are constants that can be pre-computed. The choice of these points is significant and will be discussed in Collocation with orthogonal polynomials.

5.3.2.1 Collocation constraints

The solution of the ODE, i.e. the values of \( x_{i,j} \) are now obtained by solving the following equations:

\[ \sum_{j=0}^{K} a_{j,k} \frac{x_{i,j}}{\Delta t} = f(x_{i,k}), \quad k = 1, \ldots, K. \]
5.3.2.2 Continuity constraints

The avid reader will have noticed that through the collocation constraints we obtain a system of \( K - 1 \) equations for \( K \) variables, which is insufficient.

The missing equation is used to ensure continuity between the finite elements shown in the figure above. We simply enforce equality between the final state of element \( i \), which we denote \( x_i^f \) and the initial state of the successive interval \( x_{i+1,0} \):

\[
x_{i+1,0} = x_i^f
\]

However, with our choice of collocation points \( \tau_0 = 0, \tau_j < \tau_{j+1} \leq 1, j = 0, \ldots, K - 1 \), we do not explicitly know \( x_i^f \) in the general case (unless \( \tau_K = 1 \)).

We thus evaluate the interpolation polynomial again and obtain:

\[
x_i^f = x_i^K(t_{i+1}) = \sum_{j=0}^{K} \underbrace{L_j(\tau = 1)}_{d_j} x_i,j,
\]

where similarly to the collocation coefficients \( a_{j,k} \), the continuity coefficient \( d_j \) can be precomputed.

5.3.2.3 Solving the ODE problem

It is important to note that orthogonal collocation on finite elements is an implicit ODE integration scheme, since we need to evaluate the ODE equation for yet to be determined future states of the system. While this seems inconvenient for simulation, it is straightforward to incorporate in a model predictive control (MPC) or moving horizon estimation (MHE) formulation, which are essentially large constrained optimization problems of the form:

\[
\begin{align*}
\min_z & \quad c(z) \\
\text{s.t.} & \quad h(z) = 0 \\
& \quad g(z) \leq 0
\end{align*}
\]

where \( z \) now denotes a generic optimization variable, \( c(z) \) a generic cost function and \( h(z) \) and \( g(z) \) the equality and inequality constraints.

Clearly, the equality constraints \( h(z) \) can be extended with the above mentioned collocation constraints, where the states \( x_{i,j} \) are then optimization variables of the problem.

Solving the MPC / MHE optimization problem then implicitly calculates the solution of the governing ODE which can be taken into consideration for cost, constraints etc.

5.3.2.4 Collocation with orthogonal polynomials

Finally we need to discuss how to choose the collocation points \( \tau_j, j = 0, \ldots, K \). Only for fixed values of the collocation points the collocation constraints become mere algebraic equations.

Just a short disclaimer: Ideal values for the collocation points are typically found in tables, e.g. in [Biegler2010]. The following simply illustrates how these suggested values are derived and are not implemented in practice.

We recall that the solution of the ODE can also be determined with:

\[
x(t_i) = x(t_{i-1}) + \int_{t_{i-1}}^{t_i} f(x(t))dt,
\]
which is solved numerically with the quadrature formula:

\[ x(t_i) = x(t_{i-1}) + \sum_{j=1}^{K} \omega_j \Delta t f(x(t_{i,j})) \]
\[ t_{i,j} = t_{i-1} + \tau_j \Delta t \]

The collocation points are now chosen such that the quadrature formula provides an exact solution for the original ODE if \( f(x(t)) \) is a polynomial in \( t \) of order \( 2K \). It shows that this is achieved by choosing \( \tau \) as the roots of a \( k \)-th degree polynomial \( P_K(\tau) \) which fulfills the orthogonal property:

\[ \int_0^1 P_i(\tau) P_j(\tau) = 0, \quad i = 0, \ldots, K - 1, \quad j = 1, \ldots, K \]

The resulting collocation points are called Legendre roots.

Similarly one can compute collocation points from the more general Gauss-Jacoby polynomial:

\[ \int_0^1 (1 - \tau)^{\alpha} \tau^{\beta} P_i(\tau) P_j(\tau) = 0, \quad i = 0, \ldots, K - 1, \quad j = 1, \ldots, K \]

which for \( \alpha = 0, \beta = 0 \) results exactly in the Legrendre polynomial from above where the truncation error is found to be \( O(\Delta t^{2K}) \). For \( \alpha = 1, \beta = 0 \) one can determine the Gauss-Radau collocation points with truncation error \( O(\Delta t^{2K-1}) \).

Both, Gauss-Radau and Legende roots are commonly used for orthogonal collocation and can be selected in do-mpc.

For more details about the procedure and the numerical values for the collocation points we refer to [Biegler2010].

5.3.3 Bibliography

5.4 Basics of model predictive control

Model predictive control (MPC) is a control scheme where a model is used for predicting the future behavior of the system over finite time window, the horizon. Based on these predictions and the current measured/estimated state of the system, the optimal control inputs with respect to a defined control objective and subject to system constraints is computed. After a certain time interval, the measurement, estimation and computation process is repeated with a shifted horizon. This is the reason why this method is also called receding horizon control (RHC).

Major advantages of MPC in comparison to traditional reactive control approaches, e.g. PID, etc. are

- **Proactive control action**: The controller is anticipating future disturbances, set-points etc.
- **Non-linear control**: MPC can explicitly consider non-linear systems without linearization
- **Arbitrary control objective**: Traditional set-point tracking and regulation or economic MPC
- **constrained formulation**: Explicitly consider physical, safety or operational system constraints

The MPC principle is visualized in the graphic above. The dotted line indicates the current prediction and the solid line represents the realized values. The graphic is generated using the innate plotting capabilities of do-mpc.

In the following, we will present the type of models, we can consider. Afterwards, the (basic) optimal control problem (OCP) is presented. Finally, multi-stage NMPC, the approach for robust NMPC used in do-mpc is explained.
5.4.1 System model

The system model plays a central role in MPC. do-mpc enables the optimal control of continuous and discrete-time nonlinear and uncertain systems. For the continuous case, the system model is defined by

\[
\dot{x}(t) = f(x(t), u(t), z(t), p(t), p_{tv}(t)), \\
y(t) = h(x(t), u(t), z(t), p(t), p_{tv}(t)),
\]

and for the discrete-time case by

\[
x_{k+1} = f(x_k, u_k, z_k, p_k, p_{tv,k}), \\
y_k = h(x_k, u_k, z_k, p_k, p_{tv,k}).
\]

The states of the systems are given by \(x(t), x_k\), the control inputs by \(u(t), u_k\), algebraic states by \(z(t), z_k\), (uncertain) parameters by \(p(t), p_k\), time-varying (but known) parameters by \(p_{tv}(t), p_{tv,k}\) and measurements by \(y(t), y_k\), respectively. The time is denoted as \(t\) for the continuous system and the time steps for the discrete system are indicated by \(k\).

5.4.2 Model predictive control problem

For the case of continuous systems, trying to solve OCP directly is in the general case computationally intractable because it is an infinite-dimensional problem. do-mpc uses a full discretization method, namely orthogonal collocation, to discretize the OCP. This means, that both the OCP for the continuous and the discrete system result in a similar discrete OCP.

For the application of MPC, the current state of the system needs to be known. In general, the measurement \(y_k\) does not contain the whole state vector, which means a state estimate \(\hat{x}_k\) needs to be computed. The state estimate can be derived e.g. via moving horizon estimation.
The OCP is then given by:

\[
\min_{x_{0:N+1}, u_{0:N}, z_{0:N}} \quad \begin{array}{l}
m(x_{N+1}) + \sum_{k=0}^{N} l(x_k, z_k, u_k, p_k, p_{tv,k}) \\
\end{array}
\]

subject to:

\[
x_0 = \hat{x}_0,
\]

\[
x_{k+1} = f(x_k, u_k, p_k, p_{tv,k}),
\forall k = 0, \ldots, N,
\]

\[
g(x_k, u_k, p_k, p_{tv,k}) \leq 0
\forall k = 0, \ldots, N,
\]

\[
x_{lb} \leq x_k \leq x_{ub},
\forall k = 0, \ldots, N,
\]

\[
u_{lb} \leq u_k \leq u_{ub},
\forall k = 0, \ldots, N,
\]

\[
z_{lb} \leq z_k \leq z_{ub},
\forall k = 0, \ldots, N,
\]

\[
g_{terminal}(x_{N+1}) \leq 0,
\]

where \(N\) is the prediction horizon and \(\hat{x}_0\) is the current state estimate, which is either measured (state-feedback) or estimated based on an incomplete measurement \((\hat{y}_k)\). Note that we introduce the bold letter notation, e.g. \(x_{0:N+1} = [x_0, x_1, \ldots, x_{N+1}]^T\) to represent sequences.

**do-mpc** allows to set upper and lower bounds for the states \(x_{lb}, x_{ub}\), inputs \(u_{lb}, u_{ub}\) and algebraic states \(z_{lb}, z_{ub}\). Terminal constraints can be enforced via \(g_{terminal}(\cdot)\) and general nonlinear constraints can be defined with \(g(\cdot)\), which can also be realized as soft constraints. The objective function consists of two parts, the mayer term \(m(\cdot)\) which gives the cost of the terminal state and the lagrange term \(l(\cdot)\) which is the cost of each stage \(k\).

This formulation is the basic formulation of the OCP, which is solved by **do-mpc**. In the next section, we will explain how **do-mpc** considers uncertainty to enable robust control.

---

**Note:** Please be aware, that due to the discretization in case of continuous systems, a feasible solution only means that the constraints are satisfied point-wise in time.

### 5.4.3 Robust multi-stage NMPC

One of the main features of **do-mpc** is robust control, i.e. the control action satisfies the system constraints under the presence of uncertainty. In particular, we apply the multi-stage approach which is described in the following.
5.4.3.1 General description

The basic idea for the multi-stage approach is to consider various scenarios, where a scenario is defined by one possible realization of all uncertain parameters at every control instant within the horizon. The family of all considered discrete scenarios can be represented as a tree structure, called the scenario tree:

where one scenario is one path from the root node on the left side to one leaf node on the right, e.g. the state evolution for the first scenario $S_4$ would be $x_0 \rightarrow x_2^1 \rightarrow x_4^2 \rightarrow \cdots \rightarrow x_5^5$. At every instant, the MPC problem at the root node $x_0$ is solved while explicitly taking into account the uncertain future evolution and the existence of future decisions, which can exploit the information gained throughout the evolution progress along the branches. Through this design, feedback information is considered in the open-loop optimization problem, which reduces the conservativeness of the multi-stage approach. Considering feedback information also means, that decisions $u$ branching from the same node need to be identical, because they are based on the same information, e.g. $u_1^4 = u_1^5 = u_1^6$.

The system equation for a discretized/discrete system in the multi-stage setting is given by:

$$x_{k+1}^j = f(x_p^k, u_k^j, z_p^{(j)}, r^{(j)}_k, p_{tv}, k),$$

where the function $p_{ij}$ refers to the parent state via $x_p^k$ and the considered realization of the uncertainty is given by $r^{(j)}_k$ via $d_{r ij}$. The set of all occurring exponent/index pairs $(j, k)$ are denoted as $I$.

5.4.3.2 Robust horizon

Because the uncertainty is modeled as a collection of discrete scenarios in the multi-stage approach, every node branches into $\prod_{i=1}^{n_p} v_i$ new scenarios, where $n_p$ is the number of parameters and $v_i$ is the number of explicit values considered for the $i$-th parameter. This leads to an exponential growth of the scenarios with respect to the horizon. To maintain the computational tractability of the multi-stage approach, the robust horizon $N_{\text{robust}}$ is introduced, which can be viewed as a tuning parameter. Branching is then only applied for the first $N_{\text{robust}}$ steps while the values of the uncertain parameters are kept constant for the last $N - N_{\text{robust}}$ steps. The number of considered scenarios is given by:

$$N_s = \left( \prod_{i=1}^{n_p} v_i \right)^{N_{\text{robust}}}$$

This results in $N_s = 9$ scenarios for the presented scenario tree above instead of 243 scenarios, if branching would be applied until the prediction horizon.

The impact of the robust horizon is in general minor, since MPC is based on feedback. This means the decisions are recomputed in every step after new information (measurements/state estimate) has been obtained and the branches are updated with respect to the current state.

Note: If the uncertainties $p$ are unknown but constant over time, $N_{\text{robust}} = 1$ is the suggested choice. In that case, branching of the scenario tree is only required for first time instant (since the uncertainties are constant) and the computational load is kept minimal.
### 5.4.3.3 Mathematical formulation

The formulation of the MPC problem for the multi-stage approach is given by:

\[
\begin{align*}
\min & \quad x^j_N, u^j_N, z^j_N, \forall (j, k) \in I \\
\sum_{j=1}^{N_s} & \omega_j J_j(x^j_0:N+1, u^j_0:N, z^j_0:N) \\
\text{subject to:} & \\
x_0 = \hat{x}_0 \\
x^j_{k+1} &= f(x^p(j)_k, u^j_k, z^p(j), r(j), p_{w,k}) \\
\forall (j, k) \in I \\
u^j_k &= u^i_k \text{ if } x^p(i)_k = x^p(j)_k, \\
\forall (i, k), (j, k) \in I \\
g(x^p(j)_k, u^j_k, z^p(j), r(j), p_{w,k}) &\leq 0 \\
\forall (j, k) \in I \\
x_{lb} &\leq x^j_k \leq x_{ub} \\
\forall (j, k) \in I \\
u_{lb} &\leq u^j_k \leq u_{ub} \\
\forall (j, k) \in I \\
z_{lb} &\leq z^j_k \leq z_{ub} \\
\forall (j, k) \in I \\
g_{\text{terminal}}(x^j_N, z^j_N) &\leq 0 \\
\forall (j, N) \in I,
\end{align*}
\]

The objective consists of one term for each scenario, which can be weighted according to the probability of the scenarios \( \omega_j, j = 1, \ldots, N_s \). The cost for each scenario \( J_j \) is given by:

\[
J_j = m(x^j_{N+1}) + \sum_{k=0}^{N} l(x^p(j)_k, u^j_k, z^p(j), r(j), p_{w,k}).
\]

For all scenarios, which are directly considered in the problem formulation, a feasible solution guarantees constraint satisfaction. This means if all uncertainties can only take discrete values and those are represented in the scenario tree, constraint satisfaction can be guaranteed.

For linear systems if \( p_{\text{min}} \leq p \leq p_{\text{max}} \), considering the extreme values of the uncertainties in the scenario tree guarantees constraint satisfaction, even if the uncertainties are continuous and time-varying. This design of the scenario
5.5 Basics of moving horizon estimation

Moving horizon estimation is an optimization-based state-estimation technique where the current state of the system is inferred based on a finite sequence of past measurements. In many ways it can be seen as the counterpart to model predictive control (MPC), which we are describing in our MPC article.

In comparison to more traditional state-estimation methods, e.g. the extended Kalman filter (EKF), MHE will often outperform the former in terms of estimation accuracy. This is especially true for non-linear dynamical systems, which are treated rigorously in MHE and where the EKF is known to work reliably only if the system is almost linear during updates.

Another advantage of MHE is the possible incorporation of further constraints on estimated variables. These can be used to enforce physical bounds, e.g. fractions between 0 and 1.

All of this comes at the cost of additional computational complexity. do-mpc mitigates this disadvantage through an efficient implementation which allows for very fast MHE estimation. Oftentimes, for moderately complex non-linear systems (~10 states) do-mpc will run at 10-100Hz.

5.5.1 System model

The system model plays a central role in MHE. do-mpc enables state-estimation for continuous and discrete-time nonlinear systems. For the continuous case, the system model is defined by

\[
\dot{x}(t) = f(x(t), u(t), z(t), p(t), p_{tv}(t)) + w(t),
\]

\[
y(t) = h(x(t), u(t), z(t), p(t), p_{tv}(t)) + v(t),
\]

and for the discrete-time case by

\[
x_{k+1} = f(x_k, u_k, z_k, p_k, p_{tv,k}) + w_k,
\]

\[
y_k = h(x_k, u_k, z_k, p_k, p_{tv,k}) + v_k.
\]

The states of the systems are given by \(x(t), x_k\), the control inputs by \(u(t), u_k\), algebraic states by \(z(t), z_k\), (possibly uncertain) parameters by \(p(t), p_k\), time-varying (but known) parameters by \(p_{tv}(t), p_{tv,k}\) and measurements by \(y(t), y_k\), respectively. The time is denoted as \(t\) for the continuous system and the time steps for the discrete system are indicated by \(k\).

Furthermore, we assume that the dynamic system equation is disturbed by additive (Gaussian) noise \(w(t), w_k\) and that we experience additive measurement noise \(v(t), v_k\). Note that do-mpc allows to activate or deactivate process and measurement noise explicitly for individual variables, e.g. we can express that inputs are exact and potentially measured states experience a measurement disturbance.

5.5.2 Moving horizon estimation problem

For the case of continuous systems, trying to solve the estimation problem directly is in the general case computationally intractable because it is an infinite-dimensional problem. do-mpc uses a full discretization method, namely orthogonal collocation, to discretize the OCP. This means, that both for continuous and discrete-time systems we formulate a discrete-time optimization problem to solve the estimation problem.
5.5.2.1 Concept

The fundamental idea of moving horizon estimation is that the current state of the system is inferred based on a finite sequence of \( N \) past measurements, while incorporating information from the dynamic system equation. This is formulated as an optimization problem, where the finite sequence of states, algebraic states and inputs are optimization variables. These sequences are determined, such that

1. The initial state of the sequence is coherent with the previous estimate
2. The computed measurements match the true measurements
3. The dynamic state equation is obeyed

This concept is visualized in the figure below.

Similarly to model predictive control, the MHE optimization problem is solved repeatedly at each sampling instance. At each estimation step, the new initial state is the second element from the previous estimation and we take into consideration the newest measurement while dropping the oldest. This can be seen in the figure below, which depicts the successive horizon.

5.5.2.2 Mathematical formulation

Following this concept, we formulate the MHE optimization problem as:

\[
\min_{x_0:N+1, u_0:N, p, w_0:N, v_0:N} \frac{1}{2} \| x_0 - \tilde{x}_0 \|^2 P_x + \frac{1}{2} \| p - \tilde{p} \|^2 P_p + \sum_{k=0}^{N-1} \left( \frac{1}{2} \| v_k \|^2 P_{v,k} + \frac{1}{2} \| w_k \|^2 P_{w,k} \right),
\]

\[
\text{s.t.} \quad \begin{aligned}
x_{k+1} &= f(x_k, u_k, z_k, p, p_{tv,k}) + w_k, \\
y_k &= h(x_k, u_k, z_k, p, p_{tv,k}) + v_k,
\end{aligned} \quad k = 0, \ldots, N
\]

\[
g(x_k, u_k, z_k, p, p_{tv,k}) \leq 0
\]

where we introduce the bold letter notation, e.g. \( x_{0:N+1} = [x_0, x_1, \ldots, x_{N+1}]^T \) to represent sequences and where \( \| x \|^2_P = x^TPx \) denotes the \( P \) weighted squared norm.

As mentioned above some states / measured variables do not experience additive noise, in which case their respective noise variables \( v_k, w_k \) do not appear in the optimization problem.

Also note that do-mpc allows to estimate parameters which are considered to be constant over the estimation horizon.

5.6 License

GNU LESSER GENERAL PUBLIC LICENSE  Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc.  <http://fsf.org/> Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

0. Additional Definitions.
As used herein, “this License” refers to version 3 of the GNU Lesser General Public License, and the “GNU GPL” refers to version 3 of the GNU General Public License.

“The Library” refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An “Application” is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A “Combined Work” is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the “Linked Version”.

The “Minimal Corresponding Source” for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The “Corresponding Application Code” for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or

b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.


The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License.

b) Accompany the object code with a copy of the GNU GPL and this license document.


You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.

b) Accompany the Combined Work with a copy of the GNU GPL and this license document.

c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.
d) Do one of the following:

0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.

1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.

e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)


You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.

b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License “or any later version” applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy’s public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.

5.7 Installation

*do-mpc* is a python 3.x package. Follow this guide to install *do-mpc*.

If you are new to Python, please read this article about Python environments. We recommend using a new Python environment for every project and to manage it with miniconda.

5.7.1 Requirements

*do-mpc* requires the following Python packages and their dependencies:
• numpy
• CasADi
• matplotlib

5.7.2 Option 1: PIP

Simply use PIP and install do-mpc from the terminal. This has the advantage that do-mpc is always in your Python path and can be used throughout your projects.

1. Install do-mpc:

   pip install do-mpc

Tested on Windows and Linux (Ubuntu 19.04).

PIP will also take care of dependencies and you are immediately ready to go.

Use this option if you plan to use do-mpc without altering the source code, e.g. write extensions.

2. Get example documents:

   All resources can be obtained from our release notes page. Please find the example files that match your currently installed do-mpc version in the respective section.

5.7.3 Option 2: Clone from Github

More experienced users are advised to clone or fork the most recent version of do-mpc from GitHub:

   git clone https://github.com/do-mpc/do-mpc.git

In this case, the dependencies from above must be manually taken care of. You have immediate access to our examples.

5.7.4 HSL linear solver for IPOPT

The standard configuration of do-mpc is based on IPOPT to solve the nonlinear constrained optimization problems that arise with the MPC and MHE formulation. The computational bottleneck of this method is repeatedly solving a large-scale linear systems for which IPOPT is offering a an interface to a variety of sparse symmetric indefinite linear solver. IPOPT and thus do-mpc comes by default with the MUMPS solver. It is suggested to try a different linear solver for IPOPT with do-mpc. Typically, a significant speed boost can be achieved with the HSL MA27 solver.

5.7.4.1 Option 1: Pre-compiled binaries

When installing CasADi via PIP or Anaconda (happens automatically when installing do-mpc via PIP), you obtain the pre-compiled CasADi package. To use MA27 (or other HSL solver in this setup) please follow these steps:

5.7.4.1.1 Linux

   (Tested on Ubuntu 19.10)

   1. Obtain the HSL shared library. Choose the personal licence.

   2. Unpack the archive and copy its content to a destination of your choice. (e.g. /home/username/Documents/coinhsl/)
3. Rename `libcoinhsl.so` to `libhsl.so`. CasADi is searching for the shared libraries under a depreciated name.

4. Locate your `.bashrc` file on your home directory (e.g. `/home/username/.bashrc`).

5. Add the previously created directory to your `LD_LIBRARY_PATH`, by adding the following line to your `.bashrc`:
   ```bash
   export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/home/ffiedler/Documents/coinhsl/lib"
   ```

6. Install `libgfortran` with Anaconda:
   ```bash
   conda install -c anaconda libgfortran
   ```

**Note**: To check if MA27 can be used as intended, please first change the solver according to `do_mpc.controller.MPC.set_param()`. When running the examples, inspect the IPOPT output in the console. Two possible errors are expected:

- **Tried to obtain MA27 from shared library "libhsl.so", but the following error occured:**
  `libhsl.so: cannot open shared object file: No such file or directory`

  This error suggests that step three above wasn’t executed or didn’t work.

- **Tried to obtain MA27 from shared library "libhsl.so", but the following error occured:**
  `libgfortran.so.3: cannot open shared object file: No such file or directory`

  This error suggests that step six wasn’t executed or didn’t work.

### 5.7.4.2 Option 2: Compile from source

Please see the comprehensive guide on the CasADi Github Wiki.

### 5.8 Credit

The developers of `do-mpc` own credit to CasADi and Ipopt which run at the core of our MPC and MHE implementation.

If you use `do-mpc` for published work please cite it as:


Please remember to properly cite other software that you might be using too if you use `do-mpc` (e.g. CasADi, IPOPT, …)

### 5.9 Structuring your project

In this guide we show you a suggested structure for your MPC or MHE project.

In general, we advice to use the provided templates from our GitHub repository as a starting point. We will explain the structure following the CSTR example. Simple projects can also be developed as presented in our introductory Jupyter Notebooks (`MPC`, `MHE`)
We split our MHE / MPC configuration into five separate files:

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>template_model.py</td>
<td>Define the dynamic model</td>
</tr>
<tr>
<td>template_mpc.py</td>
<td>Configure the MPC controller</td>
</tr>
<tr>
<td>template_simulator.py</td>
<td>Configure the DAE/ODE/discrete simulator</td>
</tr>
<tr>
<td>template_estimator.py</td>
<td>Configure the estimator (MHE / EKF / state-feedback)</td>
</tr>
<tr>
<td>main.py</td>
<td>Obtain all configured modules and run the loop.</td>
</tr>
</tbody>
</table>

The files all include a single function and return the configured `do_mpc.model.Model`, `do_mpc.controller.MPC`, `do_mpc.simulator.Simulator` or `do_mpc.estimator.MHE` objects, when called from a central `main.py` script.

### 5.9.1 template_model

The `do-mpc` model class is at the core of all other components and contains the mathematical description of the investigated dynamical system in the form of ordinary differential equations (ODE) or differential algebraic equations (DAE).

The `template_model.py` file will be structured as follows:

```python
def template_model():
    # Obtain an instance of the do-mpc model class
    # and select time discretization:
    model_type = 'continuous' # either 'discrete' or 'continuous'
    model = do_mpc.model.Model(model_type)

    # Introduce new states, inputs and other variables to the model, e.g.:
    C_b = model.set_variable(var_type='_x', var_name='C_b', shape=(1,1))
```

(continues on next page)
Q_dot = model.set_variable(var_type='_u', var_name='Q_dot')
...

# Set right-hand-side of ODE for all introduced states (_x).
# Names are inherited from the state definition.
model.set_rhs('C_b', ...)

# Setup model:
model.setup()

return model

5.9.2 template_mpc

With the configured model, it is possible to configure and setup the MPC controller. Note that the optimal control problem (OCP) is always given in the following form:

$$\min_{x,u,z} \sum_{k=0}^{N} \left( \sum_{i} l(x_k, u_k, z_k, p_i) + \Delta u_k^T R \Delta u_k \right) + m(x_{N+1})$$

subject to:

$$x_{lb} \leq x_k \leq x_{ub} \forall k = 0, \ldots, N + 1$$

$$u_{lb} \leq u_k \leq u_{ub} \forall k = 0, \ldots, N$$

$$z_{lb} \leq z_k \leq z_{ub} \forall k = 0, \ldots, N$$

$$m \left( x_k, u_k, z_k, p_k, p_{tv_k} \right) \leq m_{ub} \forall k = 0, \ldots, N$$

The configuration of the `do_mpc.controller.MPC` class in `template_mpc.py` can be done as follows:

```python
def template_mpc(model):
    # Obtain an instance of the do-mpc MPC class
    # and initiate it with the model:
    mpc = do_mpc.controller.MPC(model)

    # Set parameters:
    setup_mpc = {
        'n_horizon': 20,
        'n_robust': 1,
        't_step': 0.005,
        ...
    }
    mpc.set_param(**setup_mpc)

    # Configure objective function:
    mterm = (_x['C_b'] - 0.6)**2  # Setpoint tracking
```
lterm = (_x['C_b'] - 0.6)**2 # Setpoint tracking
mpc.set_objective(mterm=mterm, lterm=lterm)
mpc.set_rterm(F=0.1, Q_dot = 1e-3) # Scaling for quad. cost.

# State and input bounds:
mpc.bounds['lower', '_x', 'C_b'] = 0.1
mpc.bounds['upper', '_x', 'C_b'] = 2.0
...

mpc.setup()
return mpc

5.9.3 template_simulator

In many cases a developed control approach is first tested on a simulated system. do-mpc responds to this need with the simulator class. The simulator uses state-of-the-art DAE solvers, e.g. Sundials CVODE to solve the DAE equations defined in the supplied model. This will often be the same model as defined for the optimizer but it is also possible to use a more complex model of the same system.

The simulator is configured and setup with the supplied model in the template_simulator.py file, which is structured as follows:

```python
def template_simulator(model):
    # Obtain an instance of the do-mpc simulator class
    # and initiate it with the model:
    simulator = do_mpc.simulator.Simulator(model)

    # Set parameter(s):
    simulator.set_param(t_step = 0.005)

    # Optional: Set function for parameters and time-varying parameters.
    # Setup simulator:
    simulator.setup()

    return simulator
```

5.9.4 template_estimator

In the case that a dedicated estimator is required, another python file should be added to the project. Configuration and setup of the moving horizon estimator (MHE) will be structured as follows:

```python
def template(mhe):
    # Obtain an instance of the do-mpc MHE class
    # and initiate it with the model.
    # Optionally pass a list of parameters to be estimated.
    mhe = do_mpc.estimator.MHE(model)

    # Set parameters:
    setup_mhe = {
        'n_horizon': 10,
    }
    mhe.set_param(setup_mhe)

    # Setup estimator:
    mhe.setup()

    return mhe
```

(continues on next page)
Note that the cost function for the MHE can be freely configured using the available variables. Generally, we suggest to choose the typical MHE formulation:

\[
J = (x_0 - \tilde{x}_0)^T P_x (x_0 - \tilde{x}_0) + (p_0 - \tilde{p}_0)^T P_p (p_0 - \tilde{p}_0)
\]

\[
+ \sum_{k=0}^{n-1} (h(x_k, u_k, p_k) - y_k)^T P_y, k (h(x_k, u_k, p_k) - y_k)
\]

The measurement function must be defined in the model definition and typically contains the inputs. Inputs are not treated separately as in some other formulations.

### 5.9.5 main script

All previously defined functions are called from a single `main.py` file, e.g.:

```python
from template_model import template_model
from template_mpc import template_mpc
from template_simulator import template_simulator

model = template_model()
mpc = template_mpc(model)
simulator = template_simulator(model)
estimator = do_mpc.estimator.StateFeedback(model)
```

Simple configurations, as for the `do_mpc.estimator.StateFeedback` class above are often directly implemented in the `main.py` file.
### 5.9.5.1 Initial state & guess

Afterwards we set the initial state (true state) for the simulator. Note that in proper investigations we usually have a different initial state for the simulator (true state) and e.g. the estimator.

```python
# Set the initial state of simulator:
C_a_0 = 0.8
...
x0 = np.array([C_a_0, ...]).reshape(-1,1)
simulator.x0 = x0
```

We can set the initial guessed state for the MHE by modifying its attribute similarly as for the simulator shown above. The MPC initial guess is given when calling the function `do_mpc.controller.MPC.make_step()` for the first time.

### 5.9.5.2 Graphics configuration

Visualization the estimation and control results is key to evaluating performance and identifying potential problems. *do-mpc* has a powerful graphics library based on Matplotlib for quick and customizable graphics. After creating a blank class instance and initiating a figure object with:

```python
# Initialize graphic:
graphics = do_mpc.graphics.Graphics()
fig, ax = plt.subplots(5, sharex=True)
```

we need to configure where and what to plot, with the `graphics.Graphics.add_line()` method:

```python
graphics.add_line(var_type='_x', var_name='C_a', axis=ax[0])
# Fully customizable:
ax[0].set_ylabel('c [mol/l]')
ax[0].set_ylim(...)
...
```

Note that we are not plotting anything just yet.

### 5.9.5.3 closed-loop

As shown in Diagram *Project structure*, after obtaining the different *do-mpc* objects they can be used in the *main loop*. In code form the loop looks like this:

```python
for k in range(N_iterations):
    u0 = mpc.make_step(x0)
    y_next = simulator.make_step(u0)
    x0 = estimator.make_step(y_next)
```

Instead of running for a fixed number of iterations, we can also start an infinite loop with:

```python
while True:
    ...
```

or have some checks active:
While \( \text{mpc}._x0[\text{C_h}] \leq 0.8 \):

```python
...  
```

During or after the loop, we are using the previously configured graphics class. Open-loop predictions can be plotted at each sampling time:

```python
for k in range(N_iterations):
    u0 = mpc.make_step(x0)
    y_next = simulator.make_step(u0)
    x0 = estimator.make_step(y_next)
    graphics.reset_axes()
    graphics.plot_results(mpc.data, linewidth=3)
    graphics.plot_predictions(mpc.data, linestyle='--', linewidth=1)
    plt.show()
    input('next step')

Furthermore, we can obtain a visualization of the full closed-loop trajectory after the loop:

```python
graphics.plot_results(mpc.data)
```

## 5.10 FAQ

Some tips and tricks when you can’t rule them all.

### 5.10.1 Time-varying parameters

Time-varying parameters are an important feature of \textit{do-mpc}. But when do I need them, how are they implemented and what makes them different from regular parameters?

With model predictive control and moving horizon estimation we are considering finite future (control) or past (estimation) trajectories based on a model of our system. These finite sequences are shifting at each estimation and control step. \textbf{Time-varying parameters} are required, when:

- the model is subject to some exterior influence (e.g. weather prediction) that is varying at each element of the sequence.
- the MPC/MHE cost function contains elements (e.g. a reference for control) that is varying at each element of the sequence.

Both cases have in common that the parameters are \textbf{a priori known} and not constant over the prediction / estimation horizon. This is the main difference to regular \textbf{parameters} which typically only influence the model (not the cost function) and can be estimated with moving horizon estimation and considered as parametric uncertainties for robust model predictive control.

### 5.10.1.1 Implementation

Time-varying parameters are always introduced in the \texttt{do-mpc} \texttt{do_mpc.model.Model} with the \texttt{do_mpc.model.Model.set_variable} method. For example:

```python
model_type = 'continuous' # either 'discrete' or 'continuous'
model = do_mpc.model.Model(model_type)
```
# Introduce state temperature:
```python
temperature = model.set_variable(var_type='_x', var_name='temperature')
```

# Introduce tvp: Set-point for the temperature
```python
temperature_set_point = model.set_variable(var_type='_tvp', var_name='temperature_set_point')
```

# Introduce tvp: External temperature (disturbance)
```python
temperature_external = model.set_variable(var_type='_tvp', var_name='temperature_external')
```

The obtained time-varying parameters can be used throughout the model and all derived classes. In the shown example, we assume that the external temperature has an influence on our temperature state. We can thus incorporate this variable in the ODE:
```python
model.set_rhs('temperature', alpha*(temperature_external-temperature))
```

## 5.10.1.1 MPC configuration

Furthermore, we want to use the introduced set-point in a quadratic MPC cost function. To do this, we initiate an `do_mpc.controller.MPC` object with the configured model:
```python
mpc = do_mpc.controller.MPC(model)
mpc.set_param(n_horizon = 20, t_step = 60)
```

And then use the attributes `do_mpc.model.Model.x` and `do_mpc.model.Model.tvp` to formulate a quadratic tracking cost.
```python
lterm = (model.x['temperature']-model.tvp['temperature_set_point'])**2
mterm = lterm
mpc.set_objective(lterm=lterm, mterm=mterm)
```

**Note:** We assume here that the mpc controller is not configured in the same Python scope as the model. Thus the variables (e.g. `temperature_external`, `temperature`, ...) are not necessarily available. Instead, these variables are obtained from the model with the shown attributes.

After invoking the `do_mpc.controller.MPC.setup()` method this will create the following cost function:

$$J = \sum_{k=0}^{N+1} (T_k - T_{k,\text{set}})^2$$

The only problem that remains is: What are the values for the set-point for the temperature and the external temperature for the ODE equation? So far we have only introduced them as symbolic variables.

What makes the definition of these values so complicated is that at each control step, we need not only a single value for these variables but an entire sequence. Furthermore, these sequences are not necessarily the same (shifted) values at the next step.

To address this problem **do-mpc** allows the user to declare a **tvp-function** with `do_mpc.controller.MPC.set_tvp_fun()` which is internally invoked at each call of the MPC controller with `do_mpc.controller.MPC.make_step()`.

---

60 Chapter 5. Table of contents
The **tvp-function** returns numerical values for the currently valid sequences and passes them to the optimizer. Because the **tvp-function** is user-defined, the approach allows for the greatest flexibility.

**do-mpc** also ensures that the output of this function is consistent with the configuration of the model and controller. This is achieved by requiring the output of the **tvp-function** to be of a particular structure which can be obtained with **do_mpc.controller.MPC.get_tvp_template()**. This structure can be indexed with a time-step and the name of a previously introduced time-varying parameter. Through indexing these values can be obtained and set conveniently.

In the following we show how this works in practice. The first step is to obtain the **tvp_template**:

```python
tvp_template = mpc.get_tvp_template()
```

Afterwards, we define a function that takes as input the current time and returns the **tvp_template** filled with the currently valid sequences.

```python
def tvp_fun(t_now):
    for k in range(n_horizon+1):
        tvp_template['_tvp',k,'temperature_set_point'] = 10
        tvp_template['_tvp',k,'temperature_external'] = 20
    return tvp_template
```

**Note:** Within the **tvp_fun** above, the user is free to perform any operation. Typically, the data for the time-varying parameters is read from a numpy array or obtained as a function of the current time.

The function **tvp_fun** can now be treated similarly to a variable in the current python scope. The final step of the process is to pass this function with **do_mpc.controller.MPC.set_tvp_fun()**:

```python
mpc.set_tvp_fun(tvp_fun)
```

The configuration of the MPC controller is thus completed.

**5.10.1.1.2 MHE configuration**

The MHE configuration of the time-varying parameters is equivalent to the MPC configuration shown above.

**5.10.1.1.3 Simulator configuration**

The simulator also needs to be adapted for time-varying parameters because we cannot evaluate the previously introduced ODE without a numerical value for **temperature_external**.

The logic is the same as for the MPC controller and MHE estimator: We get the **tvp_template** with **do_mpc.simulator.Simulator.get_tvp_template()** define a function **tvp_fun** and pass it to the simulator with **do_mpc.simulator.Simulator.set_tvp_fun()**

The configuration of the simulator is significantly easier however, because we only need a single value of this parameter instead of a sequence:

```python
# Get simulator instance. The model contains _tvp.
simulator = do_mpc.simulator.Simulator(model)
# Set some required parameters
simulator.set_param(t_step = 60)
```

(continues on next page)
# Get the template
tvp_template = simulator.get_tvp_template()

# Define the function (indexing is much simpler ...)
def tvp_fun(t_now):
    tvp_template['temperature_external'] = ...
    return tvp_template

# Set the tvp_fun:
simulator.set_tvp_fun(tvp_fun)

Note: All time-varying parameters that are not explicitly set default to 0 in the tvp_template. Thus, if some parameters are not required (e.g. they were introduced for the controller), they don’t need to be set in the tvp_fun. This is shown here, where the simulator doesn’t need the set-point.

Note: From the perspective of the simulator there is no difference between time-varying parameters (_tvp) and regular parameters (_p). The difference is important only for the MPC controller and MHE estimator. These methods consider a finite sequence of future / past information, e.g. the weather, which can change over time. Parameters, on the other hand, are constant over the entire horizon.

5.10.2 Feasibility issues

A common problem with MPC control and MHE estimation are feasibility issues that arise when the solver cannot satisfy the constraints of the optimization problem.

5.10.2.1 Is the initial state feasible?

With MPC, a problem is infeasible if the initial state is infeasible. This can happen in the close-loop application, where the state prediction may vary from the true state evolution. The following tips may be used to diagnose and fix this (and other) problems.

5.10.2.2 Which constraints are violated?

Check which bound constraints are violated. Retrieve the (infeasible) “optimal” solution and compare it to the bounds:

```python
lb_bound_violation = mpc.opt_x.num.cat <= mpc.lb_opt_x
ub_bound_violation = mpc.opt_x.num.cat <= mpc.ub_opt_x
```

Retrieve the labels from the optimization variables and find those that are violating the constraints:

```python
opt_labels = mpc.opt_x.labels()
lb_labels_viol = np.array(opt_labels)[np.where(lb_bound_violation)]
ub_labels_viol = np.array(opt_labels)[np.where(ub_bound_violation)]
```

The arrays labels_lb_viol and labels_ub_viol indicate which variables are problematic.
5.10.2.3 Use soft-constraints.

Some control problems, especially with economic objective will lead to trajectories operating close to (some) con-
straints. Uncertainty or model inaccuracy may lead to constraint violations and thus infeasible (usually nonsense) solutions. Using soft-constraints may help in this case. Both the MPC controller and MHE estimator support this feature, which can be configured with (example for MPC):

```python
mpc.set_nl_cons('cons_name', expression, upper_bound, soft_constraint=True)
```

See the full feature documentation here: `do_mpc.optimizer.Optimizer.set_nl_cons`

5.11 API Reference

Find below a table of all do-mpc modules. Classes and functions of each module are shown on their respective page.

Core modules

The core modules are used to create the do-mpc control loop (click on elements to open documentation page):

Fig. 4: do-mpc control loop and interconnection of classes.
## 5.11.1 model

### Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IteratedVariables</td>
<td>Class to initiate properties and attributes for iterated variables.</td>
</tr>
<tr>
<td>Model</td>
<td>The do-mpc model class.</td>
</tr>
</tbody>
</table>

### 5.11.1.1 IteratedVariables

**Class** `do_mpc.model.IteratedVariables`

Class to initiate properties and attributes for iterated variables. This class is inherited to all iterating do-mpc classes and based on the `Model`.

**Warning:** This base class can not be used independently.

### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IteratedVariables.t0</td>
<td>Current time marker of the class.</td>
</tr>
<tr>
<td>IteratedVariables.u0</td>
<td>Initial input and current iterate.</td>
</tr>
<tr>
<td>IteratedVariables.x0</td>
<td>Initial state and current iterate.</td>
</tr>
<tr>
<td>IteratedVariables.z0</td>
<td>Initial algebraic state and current iterate.</td>
</tr>
</tbody>
</table>

#### 5.11.1.1.1 t0

Class attribute.

**IteratedVariables.t0**

Current time marker of the class. Use this property to set or query the time.

Set with `int`, `float`, `numpy.ndarray` or `casadi.DM` type.

This page is auto-generated. Page source is not available on Github.

#### 5.11.1.1.2 u0

Class attribute.

**IteratedVariables.u0**

Initial input and current iterate. This is the numerical structure holding the information about the current
input in the class. The property can be indexed according to the model definition.

Example:

```python
code
model = do_mpc.model.Model('continuous')
model.set_variable('_u','heating', shape=(4,1))
...
```

```python
code
mhe = do_mpc.estimator.MHE(model)
# or
mpc = do_mpc.estimator.MPC(model)

# Get or set current value of variable:
mpc.u0['heating', 0] # 0th element of variable
mpc.u0['heating'] # all elements of variable
mpc.u0['heating', 0:2] # 0th and 1st element
```

Useful CasADi symbolic structure methods:

- `.shape`
- `.keys()`
- `.labels()`

This page is auto-generated. Page source is not available on Github.

### 5.11.1.3 x0

Class attribute.

**IteratedVariables.x0**

Initial state and current iterate. This is the numerical structure holding the information about the current states in the class. The property can be indexed according to the model definition.

Example:

```python
code
model = do_mpc.model.Model('continuous')
model.set_variable('_x','temperature', shape=(4,1))
...
```

```python
code
mhe = do_mpc.estimator.MHE(model)
# or
mpc = do_mpc.estimator.MPC(model)

# Get or set current value of variable:
mpc.x0['temperature', 0] # 0th element of variable
mpc.x0['temperature'] # all elements of variable
mpc.x0['temperature', 0:2] # 0th and 1st element
```

Useful CasADi symbolic structure methods:

- `.shape`
- `.keys()`
- `.labels()`

This page is auto-generated. Page source is not available on Github.
5.11.1.4 z0

Class attribute.

IteratedVariables.z0

Initial algebraic state and current iterate. This is the numerical structure holding the information about the current algebraic states in the class. The property can be indexed according to the model definition.

Example:

```python
model = do_mpc.model.Model('continuous')
model.set_variable('_z','temperature', shape=(4,1))
...

mhe = do_mpc.estimator.MHE(model)
# or
mpc = do_mpc.estimator.MPC(model)

# Get or set current value of variable:
mpc.z0['temperature', 0]  # 0th element of variable
mpc.z0['temperature']     # all elements of variable
mpc.z0['temperature', 0:2]  # 0th and 1st element
```

Useful CasADi symbolic structure methods:

- `.shape`
- `.keys()`
- `.labels()`

This page is auto-generated. Page source is not available on Github.

Methods

This page is auto-generated. Page source is not available on Github.

5.11.1.2 Model

```python
class do_mpc.model.Model(model_type=None, symvar_type='SX')
```

The do-mpc model class. This class holds the full model description and is at the core of do_mpc.simulator.Simulator, do_mpc.controller.MPC and do_mpc.estimator.Estimator. The Model class is created with setting the model_type (continuous or discrete). A continuous model consists of an underlying ordinary differential equation (ODE) or differential algebraic equation (DAE):

\[
\dot{x}(t) = f(x(t), u(t), z(t), p(t), p_{tv}(t)) + w(t),
\]

\[
0 = g(x(t), u(t), z(t), p(t), p_{tv}(t))
\]

\[
y = h(x(t), u(t), z(t), p(t), p_{tv}(t)) + v(t)
\]

whereas a discrete model consists of a difference equation:

\[
x_{k+1} = f(x_k, u_k, z_k, p_k, p_{tv,k}) + w_k,
\]

\[
0 = g(x_k, u_k, z_k, p_k, p_{tv,k})
\]

\[
y_k = h(x_k, u_k, z_k, p_k, p_{tv,k}) + v_k
\]

The do-mpc model can be initiated with either SX or MX variable type. We refer to the CasADi documentation on the difference of these two types.
Note: SX vs. MX in a nutshell: In general use SX variables (default). If your model consists of scalar operations SX variables will be beneficial. Your implementation will most likely only benefit from MX variables if you use large(r)-scale matrix-vector multiplications.

Note: The option symvar_type will be inherited to all derived classes (e.g. do_mpc.simulator.Simulator, do_mpc.controller.MPC and do_mpc.estimator.Estimator). All symbolic variables in these classes will be chosen respectively.

Configuration and setup:
Configuring and setting up the Model involves the following steps:

1. Use set_variable() to introduce new variables to the model.
2. Optionally introduce “auxiliary” expressions as functions of the previously defined variables with set_expression(). The expressions can be used for monitoring or be reused as constraints, the cost function etc.
3. Optionally introduce measurement equations with set_meas(). The syntax is identical to set_expression(). By default state-feedback is assumed.
4. Define the right-hand-side of the discrete or continuous model as a function of the previously defined variables with set_rhs(). This method must be called once for each introduced state.
5. Call setup() to finalize the Model. No further changes are possible afterwards.

Note: All introduced model variables are accessible as Attributes of the Model. Use these attributes to query to variables, e.g. to form the cost function in a separate file for the MPC configuration.

Parameters

• model_type (str) – Set if the model is discrete or continuous.
• symvar_type (str) – Set if the model is configured with CasADi SX or MX variables.

Raises

• assertion – model_type must be string
• assertion – model_type must be either discrete or continuous

__getitem__(ind)
The Model class supports the __getitem__ method, which can be used to retrieve the model variables (see attribute list).

```python
# Query the states like this:
x = model.x
# or like this:
x = model['x']
```

This also allows to retrieve multiple variables simultaneously:

```python
x, u, z = model['x','u','z']
```
Attributes

<table>
<thead>
<tr>
<th>Model.aux</th>
<th>Auxiliary expressions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model.p</td>
<td>Static parameters.</td>
</tr>
<tr>
<td>Model.tvp</td>
<td>Time-varying parameters.</td>
</tr>
<tr>
<td>Model.u</td>
<td>Inputs.</td>
</tr>
<tr>
<td>Model.v</td>
<td>Measurement noise.</td>
</tr>
<tr>
<td>Model.w</td>
<td>Process noise.</td>
</tr>
<tr>
<td>Model.x</td>
<td>Dynamic states.</td>
</tr>
<tr>
<td>Model.y</td>
<td>Measurements.</td>
</tr>
<tr>
<td>Model.z</td>
<td>Algebraic states.</td>
</tr>
</tbody>
</table>

5.11.1.2.1 aux

Class attribute.

Model.aux

Auxiliary expressions. CasADi symbolic structure, can be indexed with user-defined variable names.

**Note:** Expressions are introduced with `Model.set_expression()` Use this property only to query variables.

**Example:**

```python
model = do_mpc.model.Model('continuous')
model.set_variable('_x', 'temperature', 4)  # 4 states
dt = model.x['temperature', 0] - model.x['temperature', 1]
model.set_expression('dtemp', dt)
# Query:
model.aux['dtemp', 0]  # 0th element of variable
model.aux['dtemp']     # all elements of variable
```

Useful CasADi symbolic structure methods:

- .shape
- .keys()
- .labels()

**Raises assertion** – Cannot set aux directly Use set_expression instead.

This page is auto-generated. Page source is not available on Github.

5.11.1.2.2 p

Class attribute.

Model.p

Static parameters. CasADi symbolic structure, can be indexed with user-defined variable names.
**Note:** Variables are introduced with `Model.set_variable()` Use this property only to query variables.

**Example:**

```python
model = do_mpc.model.Model('continuous')
model.set_variable('_p', 'temperature', shape=(4,1))
# Query:
model.p['temperature', 0] # 0th element of variable
model.p['temperature'] # all elements of variable
model.p['temperature', 0:2] # 0th and 1st element
```

Useful CasADi symbolic structure methods:
- `.shape`
- `.keys()`
- `.labels()`

**Raises assertion** – Cannot set model variables directly Use set_variable instead.

This page is auto-generated. Page source is not available on Github.

### 5.11.1.2.3 tvp

Class attribute.

**Model.tvp**

Time-varying parameters. CasADi symbolic structure, can be indexed with user-defined variable names.

**Note:** Variables are introduced with `Model.set_variable()` Use this property only to query variables.

**Example:**

```python
model = do_mpc.model.Model('continuous')
model.set_variable('_tvp', 'temperature', shape=(4,1))
# Query:
model.tvp['temperature', 0] # 0th element of variable
model.tvp['temperature'] # all elements of variable
model.tvp['temperature', 0:2] # 0th and 1st element
```

Useful CasADi symbolic structure methods:
- `.shape`
- `.keys()`
- `.labels()`

**Raises assertion** – Cannot set model variables directly Use set_variable instead.

This page is auto-generated. Page source is not available on Github.
5.11.1.2.4 u

Class attribute.

Model.u

Inputs. CasADi symbolic structure, can be indexed with user-defined variable names.

**Note:** Variables are introduced with `Model.set_variable()`. Use this property only to query variables.

**Example:**

```python
model = do_mpc.model.Model('continuous')
model.set_variable('_u','heating', shape=(4,1))
# Query:
model.u['heating', 0]  # 0th element of variable
model.u['heating']    # all elements of variable
model.u['heating', 0:2]  # 0th and 1st element
```

Useful CasADi symbolic structure methods:

- .shape
- .keys()
- .labels()

**Raises assertion** – Cannot set model variables directly. Use set_variable instead.

This page is auto-generated. Page source is not available on Github.

5.11.1.2.5 v

Class attribute.

Model.v

Measurement noise. CasADi symbolic structure, can be indexed with user-defined variable names.

The measurement noise structure is created automatically, whenever the `Model.set_meas()` method is called with the argument `meas_noise = True`.

**Note:** The measurement noise is used for the `do_mpc.estimator.MHE` and can be used to simulate a disturbed system in the `do_mpc.simulator.Simulator`.

Useful CasADi symbolic structure methods:

- .shape
- .keys()
- .labels()

**Raises assertion** – Cannot set v directly.

This page is auto-generated. Page source is not available on Github.
5.11.1.2.6 \texttt{w}

Class attribute.

\texttt{Model.w}

Process noise. CasADi symbolic structure, can be indexed with user-defined variable names.

The process noise structure is created automatically, whenever the \texttt{Model.set_rhs()} method is called with the argument \texttt{process_noise = True}.

\textbf{Note:} The process noise is used for the \texttt{do_mpc.estimator.MHE} and can be used to simulate a disturbed system in the \texttt{do_mpc.simulator.Simulator}.

Useful CasADi symbolic structure methods:

- \texttt{.shape}
- \texttt{.keys()}
- \texttt{.labels()}

\textbf{Raises assertion} – Cannot set \texttt{w} directly

This page is auto-generated. Page source is not available on Github.

5.11.1.2.7 \texttt{x}

Class attribute.

\texttt{Model.x}

Dynamic states. CasADi symbolic structure, can be indexed with user-defined variable names.

\textbf{Note:} Variables are introduced with \texttt{Model.set_variable()} Use this property only to query variables.

\textbf{Example:}

```python
model = do_mpc.model.Model('continuous')
model.set_variable('_x', 'temperature', shape=(4,1))
# Query:
model.x['temperature', 0]  # 0th element of variable
model.x['temperature']    # all elements of variable
model.x['temperature', 0:2]  # 0th and 1st element
```

Useful CasADi symbolic structure methods:

- \texttt{.shape}
- \texttt{.keys()}
- \texttt{.labels()}

\textbf{Raises assertion} – Cannot set model variables directly Use set_variable instead.

This page is auto-generated. Page source is not available on Github.
5.11.1.2.8 y

Class attribute.

Model.y

Measurements. CasADi symbolic structure, can be indexed with user-defined variable names.

Note: Measured variables are introduced with `Model.set_meas()` Use this property only to query variables.

Example:

```python
model = do_mpc.model.Model('continuous')
model.set_variable('_x','temperature', 4) # 4 states
model.set_meas('temperature', model.x['temperature',:2]) # first 2 measured
# Query:
model.y['temperature', 0] # 0th element of variable
model.y['temperature'] # all elements of variable
```

Useful CasADi symbolic structure methods:

- `shape`
- `keys()`
- `labels()`

Raises `assertion` – Cannot set model variables directly Use set_meas instead.

This page is auto-generated. Page source is not available on Github.

5.11.1.2.9 z

Class attribute.

Model.z

Algebraic states. CasADi symbolic structure, can be indexed with user-defined variable names.

Note: Variables are introduced with `Model.set_variable()` Use this property only to query variables.

Example:

```python
model = do_mpc.model.Model('continuous')
model.set_variable('_z','temperature', shape=(4,1))
# Query:
model.z['temperature', 0] # 0th element of variable
model.z['temperature'] # all elements of variable
model.z['temperature', 0:2] # 0th and 1st element
```

Useful CasADi symbolic structure methods:

- `shape`
- `keys()`
• .labels()

Raises assertion – Cannot set model variables directly Use set_variable instead.

This page is auto-generated. Page source is not available on Github.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model.set_alg</td>
<td>Introduce new algebraic equation to model.</td>
</tr>
<tr>
<td>Model.set_expression</td>
<td>Introduce new expression to the model class.</td>
</tr>
<tr>
<td>Model.set_meas</td>
<td>Introduce new measurable output to the model class.</td>
</tr>
<tr>
<td>Model.set_rhs</td>
<td>Formulate the right hand side (rhs) of the ODE:</td>
</tr>
<tr>
<td>Model.set_variable</td>
<td>Introduce new variables to the model class.</td>
</tr>
<tr>
<td>Model.setup</td>
<td>Setup method must be called to finalize the modelling process.</td>
</tr>
</tbody>
</table>

5.11.1.2.10 set_alg

Class method.

do_mpc.model.Model.set_alg(self, expr_name, expr)

Introduce new algebraic equation to model.

For the continuous time model, the expression must be formulated as

\[ 0 = g(x(t), u(t), z(t), p(t), p_v(t)) \]

or for a discrete model:

\[ 0 = g(x_k, u_k, z_k, p_k, p_{v,k}) \]

Note: For the introduced algebraic variables \( z \in \mathbb{R}^{n_z} \) it is required to introduce exactly \( n_z \) algebraic equations. Otherwise setup() will throw an error message.

Parameters

• expr_name (string) – Name of the introduced expression
• expr (CasADi SX or MX) – CasADi SX or MX function depending on \(_x, _u, _z, _tvp, _p\).

This page is auto-generated. Page source is not available on Github.

5.11.1.2.11 set_expression

Class method.

do_mpc.model.Model.set_expression(self, expr_name, expr)

Introduce new expression to the model class. Expressions are not required but can be used to extract further information from the model. Expressions must be formulated with respect to \(_x, _u, _z, _tvp, _p\).

Example:
Maybe you are interested in monitoring the product of two states?

```python
Introduce two scalar states:
```x_1 = model.set_variable('_x', 'x_1')
x_2 = model.set_variable('_x', 'x_2')
```# Introduce expression:
model.set_expression('x1x2', x_1*x_2)
```

This new expression `x1x2` is then available in all `do-mpc` modules utilizing this model instance. It can be set, e.g. as the cost function in `do-mpc.controller.MPC` or simply used in a graphical representation of the simulated / controlled system.

**Parameters**
- `expr_name (string)` – Arbitrary name for the given expression. Names are used for key word indexing.
- `expr (CasADi SX or MX)` – CasADi SX or MX function depending on `_x`, `_u`, `_z`, `_tvp`, `_p`.

**Raises**
- `assertion` – `expr_name` must be str
- `assertion` – `expr` must be a casadi SX or MX type
- `assertion` – Cannot call after `setup()`.

**Returns**
Returns the newly created expression. Expression can be used e.g. for the RHS.

**Return type**
casadi.SX

This page is auto-generated. Page source is not available on Github.

5.11.1.2.12 `set_meas`

Class method.

```python
do_mpc.model.Model.set_meas(self, meas_name, expr, meas_noise=True)
``` Introduce new measurable output to the model class.

\[
y = h(x(t), u(t), z(t), p(t), p_v(t)) + v(t)
\]

or in case of discrete dynamics:

\[
y_k = h(x_k, u_k, z_k, p_k, p_{tv,k}) + v_k
\]

By default, the model assumes state-feedback (all states are measured outputs). Expressions must be formulated with respect to `_x`, `_u`, `_z`, `_tvp`, `_p`.

Be default, it is assumed that the measurements experience additive noise \( v_k \). This can be deactivated for individual measured variables by changing the boolean variable `meas_noise` to `False`. Note that measurement noise is only meaningful for state-estimation and will not affect the controller. Furthermore, it can be set with each `do_mpc.simulator.Simulator` call to obtain imperfect outputs.

**Note:** For moving horizon estimation it is suggested to declare all inputs (`_u`) and e.g. a subset of states (`_x`) as measurable output. Some other MHE formulations treat inputs separately.
Note: It is often suggested to deactivate measurement noise for "measured" inputs (\( _u \)). These can typically seen as certain variables.

Example:

```python
# Introduce states:
x_meas = model.set_variable('_x', 'x', 3)  # 3 measured states (vector)
x_est = model.set_variable('_x', 'x', 3)  # 3 estimated states (vector)
# and inputs:
u = model.set_variable('_u', 'u', 2)  # 2 inputs (vector)

# define measurements:
model.set_meas('x_meas', x_meas)
model.set_meas('u', u)
```

Parameters

- **expr_name** (string) – Arbitrary name for the given expression. Names are used for key word indexing.
- **expr** (CasADi SX or MX) – CasADi SX or MX function depending on \(_x, _u, _z, _tvp, _p\).
- **meas_noise** (bool) – Set if the measurement equation is disturbed by additive noise.

Raises

- **assertion** – expr_name must be str
- **assertion** – expr must be a casadi SX or MX type
- **assertion** – Cannot call after `setup()`.

Returns Returns the newly created measurement expression.

Return type casadi.SX

This page is auto-generated. Page source is not available on Github.

5.11.1.2.13 set_rhs

Class method.

```python
do_mpc.model.Model.set_rhs(self, var_name, expr, process_noise=False)
```

Formulate the right hand side (rhs) of the ODE:

\[
\dot{x}(t) = f(x(t), u(t), z(t), p(t), p_v(t)) + w(t),
\]

or the update equation in case of discrete dynamics:

\[
x_{k+1} = f(x_k, u_k, z_k, p_k, p_v, k) + w_k,
\]

Each defined state variable must have a respective equation (of matching dimension) for the rhs. Match the rhs with the state by choosing the corresponding names. rhs must be formulated with respect to \(_x, _u, _z, _tvp, _p\).

Example:
tank_level = model.set_variable('states', 'tank_level')
tank_temp = model.set_variable('states', 'tank_temp')

tank_level_next = 0.5*tank_level
tank_temp_next = ...

model.set_rhs('tank_level', tank_level_next)
model.set_rhs('tank_temp', tank_temp_next)

Optionally, set `process_noise = True` to introduce an additive process noise variable. This is meaningful for the `do_mpc.estimator.MHE` (See `do_mpc.estimator.MHE.set_default_objective()` for more details). Furthermore, it can be set with each `do_mpc.simulator.Simulator` call to obtain imperfect (realistic) simulation results.

**Parameters**

- `var_name (string)` – Reference to previously introduced state names (with `Model.set_variable()`)
- `expr (CasADi SX or MX)` – CasADi SX or MX function depending on `_x`, `_u`, `_z`, `_tvp`, `_p`.
- `process_noise (boolean)` – (optional) Make the respective state variable non-deterministic.

**Raises**

- `assertion` – `var_name` must be str
- `assertion` – `expr` must be a casadi SX or MX type
- `assertion` – `var_name` must refer to the previously defined states

**Returns** None

**Return type** None

This page is auto-generated. Page source is not available on Github.

5.11.1.2.14 set_variable

Class method.

`do_mpc.model.Model.set_variable(self, var_type, var_name, shape=(I, I))`

Introduce new variables to the model class. Define variable type, name and shape (optional).

**Example:**

```python
# States struct (optimization variables):
C_a = model.set_variable(var_type='_x', var_name='C_a', shape=(1,1))
T_K = model.set_variable(var_type='_x', var_name='T_K', shape=(1,1))

# Input struct (optimization variables):
Q_dot = model.set_variable(var_type='_u', var_name='Q_dot')

# Fixed parameters:
alpha = model.set_variable(var_type='_p', var_name='alpha')
```
Note: var_type allows a shorthand notation e.g. _x which is equivalent to states.

Parameters

- **var_type** *(string)* – Declare the type of the variable. The following types are valid (long or short name is possible):

<table>
<thead>
<tr>
<th>Long name</th>
<th>short name</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>states</td>
<td>_x</td>
<td>Required</td>
</tr>
<tr>
<td>inputs</td>
<td>_u</td>
<td>optional</td>
</tr>
<tr>
<td>algebraic</td>
<td>_z</td>
<td>Optional</td>
</tr>
<tr>
<td>parameter</td>
<td>_p</td>
<td>Optional</td>
</tr>
<tr>
<td>timevarying_parameter</td>
<td>_tvp</td>
<td>Optional</td>
</tr>
</tbody>
</table>

- **var_name** – Set a user-defined name for the parameter. The names are reused throughout do_mpc.

- **shape** *(int or tuple of length 2.)* – Shape of the current variable (optional), defaults to 1.

Raises

- **assertion** – var_type must be string
- **assertion** – var_name must be string
- **assertion** – shape must be tuple or int
- **assertion** – Cannot call after setup().

Returns Returns the newly created symbolic variable.

Return type casadi.SX

This page is auto-generated. Page source is not available on Github.

5.11.1.2.15 setup

Class method.

do_mpc.model.Model.setup(self)

Setup method must be called to finalize the modelling process. All required model variables must be declared. The right hand side expression for _x must have been set with set_rhs().

Sets default measurement function (state feedback) if set_meas() was not called.

**Warning:** After calling setup(), the model is locked and no further variables, expressions etc. can be set.

Raises **assertion** – Definition of right hand side (rhs) is incomplete

Returns None

Return type None
5.11.2 simulator

Classes

| Simulator | A class for simulating systems. |

5.11.2.1 Simulator

class do_mpc.simulator.Simulator(model)

A class for simulating systems. Discrete-time and continuous systems can be considered.

do-mpc uses the CasADi interface to popular state-of-the-art tools such as Sundials CVODES for the integration of ODE/DAE equations.

**Configuration and setup:**

Configuring and setting up the simulator involves the following steps:

1. Set parameters with `set_param()`, e.g. the sampling time.
2. Set parameter function with `get_p_template()` and `set_p_fun()`.
3. Set time-varying parameter function with `get_tvp_template()` and `set_tvp_fun()`.
4. Setup simulator with `setup()`.

During runtime, call the simulator `make_step()` method with current input \( u \). This computes the next state of the system and the respective measurement. Optionally, pass (sampled) random variables for the process \( w \) and measurement noise \( v \) (if they were defined in :py:class:`do_mpc.model.Model`)

**Attributes**

| Simulator.t0 | Current time marker of the class. |
| Simulator.u0 | Initial input and current iterate. |
| Simulator.x0 | Initial state and current iterate. |
| Simulator.z0 | Initial algebraic state and current iterate. |

5.11.2.1.1 t0

Class attribute.

Simulator.t0

Current time marker of the class. Use this property to set of query the time.

Set with `int, float, numpy.ndarray` or `casadi.DM` type.

This page is auto-generated. Page source is not available on Github.
5.11.2.1.2 u0

Class attribute.

**Simulator.u0**

Initial input and current iterate. This is the numerical structure holding the information about the current input in the class. The property can be indexed according to the model definition.

**Example:**

```python
model = do_mpc.model.Model('continuous')
model.set_variable('_u','heating', shape=(4,1))
...

mhe = do_mpc.estimator.MHE(model)
# or
mpc = do_mpc.estimator.MPC(model)

# Get or set current value of variable:
mpc.u0['heating', 0] # 0th element of variable
mpc.u0['heating']    # all elements of variable
mpc.u0['heating', 0:2] # 0th and 1st element
```

Useful CasADi symbolic structure methods:

- .shape
- .keys()
- .labels()

This page is auto-generated. Page source is not available on Github.

5.11.2.1.3 x0

Class attribute.

**Simulator.x0**

Initial state and current iterate. This is the numerical structure holding the information about the current states in the class. The property can be indexed according to the model definition.

**Example:**

```python
model = do_mpc.model.Model('continuous')
model.set_variable('_x','temperature', shape=(4,1))
...

mhe = do_mpc.estimator.MHE(model)
# or
mpc = do_mpc.estimator.MPC(model)

# Get or set current value of variable:
mpc.x0['temperature', 0] # 0th element of variable
mpc.x0['temperature']    # all elements of variable
mpc.x0['temperature', 0:2] # 0th and 1st element
```

Useful CasADi symbolic structure methods:

- .shape
5.11.2.1.4 \( z_0 \)

Class attribute.

**Simulator.z0**

Initial algebraic state and current iterate. This is the numerical structure holding the information about the current algebraic states in the class. The property can be indexed according to the model definition.

**Example:**

```python
model = do_mpc.model.Model('continuous')
model.set_variable('_z','temperature', shape=(4,1))
...

mhe = do_mpc.estimator.MHE(model)
# or
mpc = do_mpc.estimator.MPC(model)

# Get or set current value of variable:
mpc.z0['temperature', 0] # 0th element of variable
mpc.z0['temperature'] # all elements of variable
mpc.z0['temperature', 0:2] # 0th and 1st element
```

Useful CasADi symbolic structure methods:

- `.shape`
- `.keys()`
- `.labels()`

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Simulator.get_p_template</code></td>
<td>Obtain output template for <code>set_p_fun()</code>.</td>
</tr>
<tr>
<td><code>Simulator.get_tvp_template</code></td>
<td>Obtain the output template for <code>set_tvp_fun()</code>.</td>
</tr>
<tr>
<td><code>Simulator.make_step</code></td>
<td>Main method of the simulator class during control runtime.</td>
</tr>
<tr>
<td><code>Simulator.reset_history</code></td>
<td>Reset the history of the simulator.</td>
</tr>
<tr>
<td><code>Simulator.set_initial_guess</code></td>
<td>Initial guess for DAE variables.</td>
</tr>
<tr>
<td><code>Simulator.set_p_fun</code></td>
<td>Method to set the function which gives the values of the parameters.</td>
</tr>
<tr>
<td><code>Simulator.set_param</code></td>
<td>Set the parameters for the simulator.</td>
</tr>
<tr>
<td><code>Simulator.set_tvp_fun</code></td>
<td>Method to set the function which returns the values of the time-varying parameters.</td>
</tr>
<tr>
<td><code>Simulator.setup</code></td>
<td>Sets up the simulator and finalizes the simulator configuration.</td>
</tr>
<tr>
<td><code>Simulator.simulate</code></td>
<td>Call the CasADi simulator.</td>
</tr>
</tbody>
</table>
5.11.2.1.5 get_p_template

Class method.

do_mpc.simulator.Simulator.get_p_template(self)

Obtain output template for set_p_fun(). Use this method in conjunction with set_p_fun() to define the function for retrieving the parameters at each sampling time.

See set_p_fun() for more details.

Returns numerical CasADi structure

Return type struct_SX

This page is auto-generated. Page source is not available on Github.

5.11.2.1.6 get_tvp_template

Class method.

do_mpc.simulator.Simulator.get_tvp_template(self)

Obtain the output template for set_tvp_fun(). Use this method in conjunction with set_tvp_fun() to define the function for retrieving the time-varying parameters at each sampling time.

Returns numerical CasADi structure

Return type struct_SX

This page is auto-generated. Page source is not available on Github.

5.11.2.1.7 make_step

Class method.

do_mpc.simulator.Simulator.make_step(self, u0, v0=None, w0=None)

Main method of the simulator class during control runtime. This method is called at each timestep and computes the next state or the current control input u0. The method returns the resulting measurement, as defined in do_mpc.model.Model.set_meas.

The initial state x0 is stored as a class attribute. Use this attribute x0 to change the initial state. It is also possible to supply an initial guess for the algebraic states through the attribute z0 and by calling set_initial_guess().

Finally, the method can be called with values for the process noise w0 and the measurement noise v0 that were (optionally) defined in the do_mpc.model.Model. Typically, these values should be sampled from a random distribution, e.g. np.random.randn for a random normal distribution.

The method prepares the simulator by setting the current parameters, calls simulator.simulate() and updates the do_mpc.data object.

Parameters

- u0 (numpy.ndarray) – Current input to the system.
- v0 (numpy.ndarray (optional)) – Additive measurement noise
- w0 (numpy.ndarray (optional)) – Additive process noise

Returns y_next
5.11.2.1.8 reset_history

Class method.

do_mpc.simulator.Simulator.reset_history(self)

Reset the history of the simulator.

5.11.2.1.9 set_initial_guess

Class method.

do_mpc.simulator.Simulator.set_initial_guess(self)

Initial guess for DAE variables. Use the current class attribute `z0` to create the initial guess for the DAE algebraic equations.

The simulator uses “warmstarting” to solve the continuous/discrete DAE system by using the previously computed algebraic states as an initial guess. Thus, this method is typically only invoked once.

**Warning:** If no initial values for `z0` were supplied during setup, they default to zero.

5.11.2.1.10 set_p_fun

Class method.

do_mpc.simulator.Simulator.set_p_fun(self, p_fun)

Method to set the function which gives the values of the parameters. This function must return a CasADi structure which can be obtained with `get_p_template()`.

**Example:**

In the `do_mpc.model.Model` we have defined the following parameters:

```python
Theta_1 = model.set_variable('parameter', 'Theta_1')
Theta_2 = model.set_variable('parameter', 'Theta_2')
Theta_3 = model.set_variable('parameter', 'Theta_3')
```

To integrate the ODE or evaluate the discrete dynamics, the simulator needs to obtain the numerical values of these parameters at each timestep. In the most general case, these values can change, which is why a function must be supplied that can be evaluated at each timestep to obtain the current values.

`do-mpc` requires this function to have a specific return structure which we obtain first by calling:

```python
p_template = simulator.get_p_template()
```

The parameter function can look something like this:
```python
p_template['Theta_1'] = 2.25e-4
p_template['Theta_2'] = 2.25e-4
p_template['Theta_3'] = 2.25e-4

def p_fun(t_now):
    return p_template

simulator.set_p_fun(p_fun)
```

which results in constant parameters.

A more “interesting” variant could be this random-walk:

```python
p_template['Theta_1'] = 2.25e-4
p_template['Theta_2'] = 2.25e-4
p_template['Theta_3'] = 2.25e-4

def p_fun(t_now):
    p_template['Theta_1'] += 1e-6*np.random.randn()
    p_template['Theta_2'] += 1e-6*np.random.randn()
    p_template['Theta_3'] += 1e-6*np.random.randn()
    return p_template
```

Parameters

**p_fun** *(python function)* – A function which gives the values of the parameters

Raises

**assert** – p must have the right structure

Returns

None

Return type

None

This page is auto-generated. Page source is not available on Github.

### 5.11.2.11 set_param

Class method.

do_mpc.simulator.Simulator.set_param*(self, **kwargs)*

Set the parameters for the simulator. Setting the simulation time step t_step is necessary for setting up the simulator via setup_simulator.

Parameters

- **integration_tool** *(string)* – Sets which integration tool is used, defaults to cvodes (only continuous)
- **abstol** *(float)* – gives the maximum allowed absolute tolerance for the integration, defaults to 1e-10 (only continuous)
- **reltol** – gives the maximum allowed relative tolerance for the integration, defaults to 1e-10 (only continuous)
- **t_step** *(float)* – Sets the time step for the simulation

Returns

None

Return type

None

This page is auto-generated. Page source is not available on Github.
5.11.2.1.12 set_tvp_fun

Class method.

do_mpc.simulator.Simulator.set_tvp_fun(self, tvp_fun)

Method to set the function which returns the values of the time-varying parameters. This function must return a CasADi structure which can be obtained with get_tvp_template().

In the do_mpc.model.Model we have defined the following parameters:

```
a = model.set_variable('_tvp', 'a')
```

The integrate the ODE or evaluate the discrete dynamics, the simulator needs to obtain the numerical values of these parameters at each timestep. In the most general case, these values can change, which is why a function must be supplied that can be evaluated at each timestep to obtain the current values.

do-mpc requires this function to have a specific return structure which we obtain first by calling:

```
tvp_template = simulator.get_tvp_template()
```

The time-varying parameter function can look something like this:

```
def tvp_fun(t_now):
    tvp_template['a'] = 3
    return tvp_template

simulator.set_tvp_fun(tvp_fun)
```

which results in constant parameters.

Note: From the perspective of the simulator there is no difference between time-varying parameters and regular parameters. The difference is important only for the MPC controller and MHE estimator. These methods consider a finite sequence of future / past information, e.g. the weather, which can change over time. Parameters, on the other hand, are constant over the entire horizon.

Parameters tvp_fun (function) – Function which gives the values of the time-varying parameters

Raises

- assertion – tvp_fun has incorrect return type.
- assertion – Incorrect output of tvp_fun. Use get_tvp_template to obtain the required structure.

Returns None

Return type None

This page is auto-generated. Page source is not available on Github.

5.11.2.1.13 setup

Class method.
do_mpc.simulator.Simulator.setup(self)
Sets up the simulator and finalizes the simulator configuration. Only after the setup, the make_step() method becomes available.

Raises assertion – t_step must be set

Returns None

Return type None

This page is auto-generated. Page source is not available on Github.

5.11.2.1.14 simulate

Class method.

do_mpc.simulator.Simulator.simulate(self)
Call the CasADi simulator.

Warning: simulate() can be used as part of the public API but is typically called from within make_step() which wraps this method and sets the required values to the sim_x_num and sim_p_num structures automatically.

Numerical values for sim_x_num and sim_p_num need to be provided beforehand in order to simulate the system for one time step:
• states sim_c_num['_x']
• algebraic states sim_z_num['_z']
• inputs sim_p_num['_u']
• parameter sim_p_num['_p']
• time-varying parameters sim_p_num['_tvp']

The function returns the new state of the system.

Returns x_new

Return type numpy array

This page is auto-generated. Page source is not available on Github.

This page is auto-generated. Page source is not available on Github.

5.11.3 optimizer

Classes

Optimizer
The base class for the optimization based state estimation (MHE) and predictive controller (MPC).
5.11.3.1 Optimizer

class do_mpc.optimizer.Optimizer

The base class for the optimization based state estimation (MHE) and predictive controller (MPC). This class establishes the jointly used attributes, methods and properties.

**Warning:** The Optimizer base class can not be used independently.

### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimizer.bounds</td>
<td>Query and set bounds of the optimization variables.</td>
</tr>
<tr>
<td>Optimizer.lb_opt_x</td>
<td>Query and modify the lower bounds of all optimization variables ( \text{opt}_x ).</td>
</tr>
<tr>
<td>Optimizer.nlp_cons</td>
<td>Query and modify (symbolically) the NLP constraints.</td>
</tr>
<tr>
<td>Optimizer.nlp_cons_lb</td>
<td>Query and modify the lower bounds of the ( \text{nlp}_\text{cons} ).</td>
</tr>
<tr>
<td>Optimizer.nlp_cons_ub</td>
<td>Query and modify the upper bounds of the ( \text{nlp}_\text{cons} ).</td>
</tr>
<tr>
<td>Optimizer.nlp_obj</td>
<td>Query and modify (symbolically) the NLP objective function.</td>
</tr>
<tr>
<td>Optimizer.scaling</td>
<td>Query and set scaling of the optimization variables.</td>
</tr>
<tr>
<td>Optimizer.ub_opt_x</td>
<td>Query and modify the lower bounds of all optimization variables ( \text{opt}_x ).</td>
</tr>
</tbody>
</table>

### 5.11.3.1.1 bounds

Class attribute.

**Optimizer.bounds**

Query and set bounds of the optimization variables. The `bounds()` method is an indexed property, meaning getting and setting this property requires an index and calls this function. The power index (elements are separated by commas) must contain at least the following elements:

<table>
<thead>
<tr>
<th>order</th>
<th>index name</th>
<th>valid options</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>bound type</td>
<td>lower and upper</td>
</tr>
<tr>
<td>2</td>
<td>variable type</td>
<td>(_x), (_u) and (_z) (and (_p) _est for MHE)</td>
</tr>
<tr>
<td>3</td>
<td>variable name</td>
<td>Names defined in do_mpc.model.Model,</td>
</tr>
</tbody>
</table>

Further indices are possible (but not necessary) when the referenced variable is a vector or matrix.

**Example:**

```python
# Set with:
optimizer.bounds['lower', '_x', 'phi_1'] = -2*np.pi
optimizer.bounds['upper', '_x', 'phi_1'] = 2*np.pi

# Query with:
optimizer.bounds['lower', '_x', 'phi_1']
```

This page is auto-generated. Page source is not available on Github.
5.11.3.1.2 lb_opt_x

Class attribute.

Optimizer.lb_opt_x

Query and modify the lower bounds of all optimization variables opt_x. This is a more advanced method of setting bounds on optimization variables of the MPC/MHE problem. Users with less experience are advised to use bounds instead.

The attribute returns a nested structure that can be indexed using powerindexing. Please refer to opt_x for more details.

Note: The attribute automatically considers the scaling variables when setting the bounds. See scaling for more details.

Note: Modifications must be done after calling prepare_nlp() or setup() respectively.

This page is auto-generated. Page source is not available on Github.

5.11.3.1.3 nlp_cons

Class attribute.

Optimizer.nlp_cons

Query and modify (symbolically) the NLP constraints. Use the variables in opt_x and opt_p.

Prior to calling create_nlp() this attribute returns a list of symbolic constraints. After calling create_nlp() this attribute returns the concatenation of this list and the attribute cannot be altered anymore.

It is advised to append to the current list of nlp_cons:

```python
mpc.prepare_nlp()

# Create new constraint: Input at timestep 0 and 1 must be identical.
extra_cons = mpc.opt_x["_u", 0, 0]-mpc.opt_x["_u", 1, 0]
mpc.nlp_cons.append(extra_cons)

# Create appropriate upper and lower bound (here they are both 0 to create an equality constraint)
mpc.nlp_cons_lb.append(np.zeros(extra_cons.shape))
mpc.nlp_cons_ub.append(np.zeros(extra_cons.shape))

mpc.create_nlp()
```

See the documentation of opt_x and opt_p on how to query these attributes.

Warning: This is a VERY low level feature and should be used with extreme caution. It is easy to break the code.

Be especially careful NOT to accidentally overwrite the default objective.
Note: Modifications must be done after calling `prepare_nlp()` and before calling `create_nlp()`

This page is auto-generated. Page source is not available on Github.

### 5.11.3.1.4 nlp_cons_lb

Class attribute.

Optimizer.

Query and modify the lower bounds of the `nlp_cons`.

Prior to calling `create_nlp()` this attribute returns a list of lower bounds matching the list of constraints obtained with `nlp_cons`. After calling `create_nlp()` this attribute returns the concatenation of this list.

Values for lower (and upper) bounds MUST be added when adding new constraints to `nlp_cons`.

**Warning:** This is a VERY low level feature and should be used with extreme caution. It is easy to break the code.

Note: Modifications must be done after calling `prepare_nlp()`

This page is auto-generated. Page source is not available on Github.

### 5.11.3.1.5 nlp_cons_ub

Class attribute.

Optimizer.

Query and modify the upper bounds of the `nlp_cons`.

Prior to calling `create_nlp()` this attribute returns a list of upper bounds matching the list of constraints obtained with `nlp_cons`. After calling `create_nlp()` this attribute returns the concatenation of this list.

Values for upper (and lower) bounds MUST be added when adding new constraints to `nlp_cons`.

**Warning:** This is a VERY low level feature and should be used with extreme caution. It is easy to break the code.

Note: Modifications must be done after calling `prepare_nlp()`

This page is auto-generated. Page source is not available on Github.

### 5.11.3.1.6 nlp_obj

Class attribute.
Optimizer.nlp_obj
Query and modify (symbolically) the NLP objective function. Use the variables in opt_x and opt_p.

It is advised to add to the current objective, e.g.:

```python
mpc.prepare_nlp()
# Modify the objective
mpc.nlp_obj += sum1(vertcat(*mpc.opt_x['_x', -1, 0])**2)
# Finish creating the NLP
mpc.create_nlp()
```

See the documentation of opt_x and opt_p on how to query these attributes.

**Warning:** This is a VERY low level feature and should be used with extreme caution. It is easy to break the code.

Be especially careful NOT to accidentally overwrite the default objective.

**Note:** Modifications must be done after calling `prepare_nlp()` and before calling `create_nlp()`

This page is auto-generated. Page source is not available on Github.

### 5.11.3.1.7 scaling

Class attribute.

**Optimizer.scaling**
Query and set scaling of the optimization variables. The `Optimizer.scaling()` method is an indexed property, meaning getting and setting this property requires an index and calls this function. The power index (elements are separated by commas) must contain at least the following elements:

<table>
<thead>
<tr>
<th>order</th>
<th>index name</th>
<th>valid options</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>variable type</td>
<td>_x, _u and _z (and _p_est for MHE)</td>
</tr>
<tr>
<td>2</td>
<td>variable name</td>
<td>Names defined in do_mpc.model.Model</td>
</tr>
</tbody>
</table>

Further indices are possible (but not necessary) when the referenced variable is a vector or matrix.

**Example:**

```python
# Set with:
optimizer.scaling['_x', 'phi_1'] = 2
optimizer.scaling['_x', 'phi_2'] = 2

# Query with:
optimizer.scaling['_x', 'phi_1']
```

Scaling factors $a$ affect the MHE / MPC optimization problem. The optimization variables are scaled variables:

$$
\bar{\phi} = \frac{\phi}{a_\phi} \quad \forall \phi \in [x, u, z, p_{est}]
$$
Scaled variables are used to formulate the bounds $\bar{\phi}_b \leq \bar{\phi}_u$ and for the evaluation of the ODE. For the objective function and the nonlinear constraints the unscaled variables are used. The algebraic equations are also not scaled.

**Note:** Scaling the optimization problem is suggested when states and / or inputs take on values which differ by orders of magnitude.

This page is auto-generated. Page source is not available on Github.

### 5.11.3.1.8 ub_opt_x

Class attribute.

**Optimizer.ub_opt_x**

Query and modify the lower bounds of all optimization variables `opt_x`. This is a more advanced method of setting bounds on optimization variables of the MPC/MHE problem. Users with less experience are advised to use `bounds` instead.

The attribute returns a nested structure that can be indexed using powerindexing. Please refer to `opt_x` for more details.

**Note:** The attribute automatically considers the scaling variables when setting the bounds. See `scaling` for more details.

**Note:** Modifications must be done after calling `prepare_nlp()` or `setup()` respectively.

This page is auto-generated. Page source is not available on Github.

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Optimizer.create_nlp</code></td>
<td>Create the optimization problem.</td>
</tr>
<tr>
<td><code>Optimizer.get_tvp_template</code></td>
<td>Obtain output template for <code>set_tvp_fun()</code>.</td>
</tr>
<tr>
<td><code>Optimizer.prepare_nlp</code></td>
<td>Prepare the optimization problem.</td>
</tr>
<tr>
<td><code>Optimizer.reset_history</code></td>
<td>Reset the history of the optimizer.</td>
</tr>
<tr>
<td><code>Optimizer.set_nl_cons</code></td>
<td>Introduce new constraint to the class.</td>
</tr>
<tr>
<td><code>Optimizer.set_tvp_fun</code></td>
<td>Set function which returns time-varying parameters.</td>
</tr>
<tr>
<td><code>Optimizer.solve</code></td>
<td>Solves the optimization problem.</td>
</tr>
</tbody>
</table>

### 5.11.3.1.9 create_nlp

Class method.

`do_mpc.optimizer.Optimizer.create_nlp(self)`

Create the optimization problem. Typically, this method is called internally from `setup()`.

Users should only call this method if they intend to modify the objective with `nlp_obj`, the constraints with `nlp_cons, nlp_cons_lb` and `nlp_cons_ub`.

To finish the setup process, users MUST call `create_nlp()` afterwards.
Note: Do NOT call `setup()` if you intend to go the manual route with `prepare_nlp()` and `create_nlp()`.

Note: Only AFTER calling `prepare_nlp()` the previously mentioned attributes `nlp_obj`, `nlp_cons`, `nlp_cons_lb`, `nlp_cons_ub` become available.

This page is auto-generated. Page source is not available on Github.

5.11.3.1.10 `get_tvp_template`

Class method.

```python
do_mpc.optimizer.Optimizer.get_tvp_template(self)
```

Obtain output template for `set_tvp_fun()`.

The method returns a structured object with `n_horizon+1` elements, and a set of time-varying parameters (as defined in `do_mpc.model.Model`) for each of these instances. The structure is initialized with all zeros. Use this object to define values of the time-varying parameters.

This structure (with numerical values) should be used as the output of the `tvp_fun` function which is set to the class with `set_tvp_fun()`. Use the combination of `get_tvp_template()` and `set_tvp_fun()`.

Example:

```python
# in model definition:
alpha = model.set_variable(var_type='_tvp', var_name='alpha')
beta = model.set_variable(var_type='_tvp', var_name='beta')
...
# in optimizer configuration:
tvp_temp_1 = optimizer.get_tvp_template()
tvp_temp_1['_tvp', :] = np.array([1,1])
tvp_temp_2 = optimizer.get_tvp_template()
tvp_temp_2['_tvp', :] = np.array([0,0])

def tvp_fun(t_now):
    if t_now<10:
        return tvp_temp_1
    else:
        tvp_temp_2
optimizer.set_tvp_fun(tvp_fun)
```

Returns None

Return type None

This page is auto-generated. Page source is not available on Github.
5.11.3.1.11 prepare_nlp

Class method.

do_mpc.optimizer.Optimizer.prepare_nlp(self)

Prepare the optimization problem. Typically, this method is called internally from setup().

Users should only call this method if they intend to modify the objective with nlp_obj, the constraints with nlp_cons, nlp_cons_lb and nlp_cons_ub.

To finish the setup process, users MUST call create_nlp() afterwards.

**Note:** Do NOT call setup() if you intend to go the manual route with prepare_nlp() and create_nlp().

**Note:** Only AFTER calling prepare_nlp() the previously mentionned attributes nlp_obj, nlp_cons, nlp_cons_lb, nlp_cons_ub become available.

This page is auto-generated. Page source is not available on Github.

5.11.3.1.12 reset_history

Class method.

do_mpc.optimizer.Optimizer.reset_history(self)

Reset the history of the optimizer. All data from the do_mpc.data.Data instance is removed.

This page is auto-generated. Page source is not available on Github.

5.11.3.1.13 set_nl_cons

Class method.

do_mpc.optimizer.Optimizer.set_nl_cons(self, expr_name, expr, ub=inf, soft_constraint=False, penalty_term_cons=1, maximum_violation=inf)

Introduce new constraint to the class. Further constraints are optional. Expressions must be formulated with respect to _x, _u, _z, _tvp, _p. They are implemented as:

\[ m(x, u, z, p_v, p) \leq m_{ub} \]

Setting the flag soft_constraint=True will introduce slack variables \( \epsilon \), such that:

\[ m(x, u, z, p_v, p) - \epsilon \leq m_{ub}, \quad 0 \leq \epsilon \leq \epsilon_{max}, \]

Slack variables are added to the cost function and multiplied with the supplied penalty term. This formulation makes constraints soft, meaning that a certain violation is tolerated and does not lead to infeasibility. Typically, high values for the penalty are suggested to avoid significant violation of the constraints.

**Parameters**

- **expr_name (string)** – Arbitrary name for the given expression. Names are used for key word indexing.
- **expr** *(CasADi SX or MX)* – CasADi SX or MX function depending on \_x, \_u, \_z, \_tvp, \_p.

**Raises**
- **assertion** – expr_name must be str
- **assertion** – expr must be a casadi SX or MX type

**Returns** Returns the newly created expression. Expression can be used e.g. for the RHS.

**Return type** casadi.SX or casadi.MX

This page is auto-generated. Page source is not available on Github.

### 5.11.3.1.14 set_tvp_fun

Class method.

**do_mpc.optimizer.Optimizer.set_tvp_fun**(self, tvp_fun)

Set function which returns time-varying parameters.

The tvp_fun is called at each optimization step to get the current prediction of the time-varying parameters. The supplied function must be callable with the current time as the only input. Furthermore, the function must return a CasADi structured object which is based on the horizon and on the model definition. The structure can be obtained with `get_tvp_template()`.

**Example:**

```python
# in model definition:
alpha = model.set_variable(var_type='\_tvp', var_name='alpha')
beta = model.set_variable(var_type='\_tvp', var_name='beta')
...
# in optimizer configuration:
tvp_temp_1 = optimizer.get_tvp_template()
tvp_temp_1[\_tvp', :] = np.array([[1,1]])
tvp_temp_2 = optimizer.get_tvp_template()
tvp_temp_2[\_tvp', :] = np.array([[0,0]])

def tvp_fun(t_now):
    if t_now<10:
        return tvp_temp_1
    else:
        tvp_temp_2
optimizer.set_tvp_fun(tvp_fun)
```

**Note:** The method `set_tvp_fun()` must be called prior to setup IF time-varying parameters are defined in the model. It is not required to call the method if no time-varying parameters are defined.

**Parameters tvp_fun** *(function)* – Function that returns the predicted tvp values at each timestep. Must have single input (float) and return a `structure3.DMStruct` (obtained with `get_tvp_template()`).

This page is auto-generated. Page source is not available on Github.
5.11.3.15 solve

Class method.

\texttt{do_mpc.opt}\texttt{imizer.Optimizer.solve(self)}

Solves the optimization problem.

The current problem is defined by the parameters in the \texttt{opt_p_num} CasADi structured Data.

Typically, \texttt{opt_p_num} is prepared for the current iteration in the \texttt{make_step()} method. It is, however, valid and possible to directly set parameters in \texttt{opt_p_num} before calling \texttt{solve()}.

The method updates the \texttt{opt_p_num} and \texttt{opt_x_num} attributes of the class. By resetting \texttt{opt_x_num} to the current solution, the method implicitly enables \textit{warmstarting the optimizer} for the next iteration, since this vector is always used as the initial guess.

\textbf{Warning:} The method is part of the public API but it is generally not advised to use it. Instead we recommend to call \texttt{make_step()} at each iterations, which acts as a wrapper for \texttt{solve()}.

\begin{itemize}
  \item \textbf{Raises} assertion – Optimizer was not setup yet.
  \item \textbf{Returns} None
  \item \textbf{Return type} None
\end{itemize}

This page is auto-generated. Page source is not available on Github.

This page is auto-generated. Page source is not available on Github.

This page is auto-generated. Page source is not available on Github.

5.11.4 controller

Classes

\begin{itemize}
  \item \texttt{MPC} Model predictive controller.
\end{itemize}

5.11.4.1 MPC

\texttt{class do_mpc.controller.MPC(model)}

Model predictive controller.

For general information on model predictive control, please read our background article.

The MPC controller extends the \texttt{do_mpc.optimizer.Optimizer} base class (which is also used for the \texttt{do_mpc.estimator.MHE} estimator).

Use this class to configure and run the MPC controller based on a previously configured \texttt{do_mpc.model. Model} instance.

Configuration and setup:

Configuring and setting up the MPC controller involves the following steps:

1. Use \texttt{set_param()} to configure the \texttt{MPC} instance.
2. Set the objective of the control problem with \texttt{set_objective()} and \texttt{set_rterm()}

94 Chapter 5. Table of contents
3. Set upper and lower bounds with `bounds` (optional).
4. Set further (non-linear) constraints with `set_nl_cons()` (optional).
5. Use the low-level API (`get_p_template()` and `set_p_fun()` or high level API (`set_uncertainty_values()`) to create scenarios for robust MPC (optional).
6. Use `get_tvp_template()` and `set_tvp_fun()` to create a method to obtain new time-varying parameters at each iteration.
7. To finalize the class configuration there are two routes. The default approach is to call `setup()`. For deep customization use the combination of `prepare_nlp()` and `create_nlp()`. See graph below for an illustration of the process.

![Route to setting up the MPC class.](image)

**Warning:** Before running the controller, make sure to supply a valid initial guess for all optimized variables (states, algebraic states and inputs). Simply set the initial values of \( x_0 \), \( z_0 \) and \( u_0 \) and then call `set_initial_guess()`.

To take full control over the initial guess, modify the values of `opt_x_num`.

During runtime call `make_step()` with the current state \( x \) to obtain the optimal control input \( u \).

### Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MPC.Bounds</code></td>
<td>Query and set bounds of the optimization variables.</td>
</tr>
<tr>
<td><code>MPC.lb_opt_x</code></td>
<td>Query and modify the lower bounds of all optimization variables <code>opt_x</code>.</td>
</tr>
<tr>
<td><code>MPC.nlp_cons</code></td>
<td>Query and modify (symbolically) the NLP constraints.</td>
</tr>
<tr>
<td><code>MPC.nlp_cons_lb</code></td>
<td>Query and modify the lower bounds of the <code>nlp_cons</code>.</td>
</tr>
<tr>
<td><code>MPC.nlp_cons_ub</code></td>
<td>Query and modify the upper bounds of the <code>nlp_cons</code>.</td>
</tr>
</tbody>
</table>

Continued on next page
### 5.11.4.1.1 bounds

Class attribute.

**MPC.bounds**

Query and set bounds of the optimization variables. The `bounds()` method is an indexed property, meaning getting and setting this property requires an index and calls this function. The power index (elements are separated by commas) must contain at least the following elements:

<table>
<thead>
<tr>
<th>order</th>
<th>index name</th>
<th>valid options</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>bound type</td>
<td>lower and upper</td>
</tr>
<tr>
<td>2</td>
<td>variable type</td>
<td>_x, _u and _z (and _p_est for MHE)</td>
</tr>
<tr>
<td>3</td>
<td>variable name</td>
<td>Names defined in <code>do_mpc.model.Model</code>.</td>
</tr>
</tbody>
</table>

Further indices are possible (but not necessary) when the referenced variable is a vector or matrix.

**Example:**

```python
# Set with:
optimizer.bounds['lower', '_x', 'phi_1'] = -2*np.pi
optimizer.bounds['upper', '_x', 'phi_1'] = 2*np.pi

# Query with:
optimizer.bounds['lower', '_x', 'phi_1']
```

This page is auto-generated. Page source is not available on Github.

### 5.11.4.1.2 lb_opt_x

Class attribute.

**MPC.lb_opt_x**

Query and modify the lower bounds of all optimization variables `opt_x`. This is a more advanced method of setting bounds on optimization variables of the MPC/MHE problem. Users with less experience are advised to use `bounds` instead.
The attribute returns a nested structure that can be indexed using powerindexing. Please refer to `opt_x` for more details.

**Note:** The attribute automatically considers the scaling variables when setting the bounds. See `scaling` for more details.

**Note:** Modifications must be done after calling `prepare_nlp()` or `setup()` respectively.

This page is auto-generated. Page source is not available on Github.

### 5.11.4.1.3 nlp_cons

Class attribute.

```python
MPC.nlp_cons
```

Query and modify (symbolically) the NLP constraints. Use the variables in `opt_x` and `opt_p`.

Prior to calling `create_nlp()` this attribute returns a list of symbolic constraints. After calling `create_nlp()` this attribute returns the concatenation of this list and the attribute cannot be altered anymore.

It is advised to append to the current list of `nlp_cons`:

```python
mpc.prepare_nlp()

# Create new constraint: Input at timestep 0 and 1 must be identical.
extra_cons = mpc.opt_x["_u", 0, 0] - mpc.opt_x["_u", 1, 0]
mpc.nlp_cons.append(
    extra_cons
)

# Create appropriate upper and lower bound (here they are both 0 to create an equality constraint)
mpc.nlp_cons_lb.append(np.zeros(extra_cons.shape))
mpc.nlp_cons_ub.append(np.zeros(extra_cons.shape))
mpc.create_nlp()
```

See the documentation of `opt_x` and `opt_p` on how to query these attributes.

**Warning:** This is a VERY low level feature and should be used with extreme caution. It is easy to break the code.

Be especially careful NOT to accidentally overwrite the default objective.

**Note:** Modifications must be done after calling `prepare_nlp()` and before calling `create_nlp()`.
5.11.4.1.4 nlp_cons_lb

Class attribute.

MPC.nlp_cons_lb
Query and modify the lower bounds of the nlp_cons.

Prior to calling create_nlp() this attribute returns a list of lower bounds matching the list of constraints obtained with nlp_cons. After calling create_nlp() this attribute returns the concatenation of this list.

Values for lower (and upper) bounds MUST be added when adding new constraints to nlp_cons.

**Warning:** This is a VERY low level feature and should be used with extreme caution. It is easy to break the code.

**Note:** Modifications must be done after calling prepare_nlp()
mpc.prepare_nlp()
# Modify the objective
mpc.nlp_obj += sum1(vertcat(*mpc.opt_x['x', -1, 0])**2)
# Finish creating the NLP
mpc.create_nlp()

See the documentation of opt_x and opt_p on how to query these attributes.

**Warning:** This is a VERY low level feature and should be used with extreme caution. It is easy to break the code.
Be especially careful NOT to accidentally overwrite the default objective.

**Note:** Modifications must be done after calling prepare_nlp() and before calling create_nlp()

This page is auto-generated. Page source is not available on Github.

### 5.11.4.1.7 opt_p

Class attribute.

**MPC.opt_p**

Full structure of (symbolic) MPC parameters.

The attribute is a CasADi numeric structure with nested power indices. It can be indexed as follows:

```py
# initial state:
opt_p['x0', _x_name]
# uncertain scenario parameters
opt_p['p', scenario, _p_name]
# time-varying parameters:
opt_p['tvp', time_step, _tvp_name]
# input at time k-1:
opt_p['u_prev', time_step, scenario]
```

The names refer to those given in the do_mpc.model.Model configuration. Further indices are possible, if the variables are itself vectors or matrices.

**Warning:** Do not tweak or overwrite this attribute unless you known what you are doing.

**Note:** The attribute is populated when calling setup() or prepare_nlp()

This page is auto-generated. Page source is not available on Github.

### 5.11.4.1.8 opt_p_num

Class attribute.

This page is auto-generated. Page source is not available on Github.
MPC.\texttt{opt\_p\_num}

Full MPC parameter vector.

This attribute is used when calling the MPC solver to pass all required parameters, including

- initial state
- uncertain scenario parameters
- time-varying parameters
- previous input sequence

do-mpc handles setting these parameters automatically in the \texttt{make\_step()} method. However, you can set these values manually and directly call \texttt{solve()}. The attribute is a CasADi numeric structure with nested power indices. It can be indexed as follows:

\begin{verbatim}
# initial state:
opt\_p\_num['_x0', _x_name]
# uncertain scenario parameters
opt\_p\_num['_p', scenario, _p_name]
# time-varying parameters:
opt\_p\_num['_tvp', time_step, _tvp_name]
# input at time k-1:
opt\_p\_num['_u\_prev', time_step, scenario]
\end{verbatim}

The names refer to those given in the \texttt{do_mpc.model.Model} configuration. Further indices are possible, if the variables are itself vectors or matrices.

\textbf{Warning:} Do not tweak or overwrite this attribute unless you known what you are doing.

\textbf{Note:} The attribute is populated when calling \texttt{setup()}

This page is auto-generated. Page source is not available on Github.

\section*{5.11.4.1.9 \texttt{opt\_x}}

Class attribute.

MPC.\texttt{opt\_x}

Full structure of (symbolic) MPC optimization variables.

The attribute is a CasADi symbolic structure with nested power indices. It can be indexed as follows:

\begin{verbatim}
# dynamic states:
opt\_x['_x', time_step, scenario, collocation_point, _x_name]
# algebraic states:
opt\_x['_z', time_step, scenario, collocation_point, _z_name]
# inputs:
opt\_x['_u', time_step, scenario, _u_name]
# slack variables for soft constraints:
opt\_x['_eps', time_step, scenario, _nl_cons_name]
\end{verbatim}

The names refer to those given in the \texttt{do_mpc.model.Model} configuration. Further indices are possible, if the variables are itself vectors or matrices.
The attribute can be used to alter the objective function or constraints of the NLP.

How to query?

Querying the structure is more complicated than it seems at first look because of the scenario-tree used for robust MPC. To obtain all collocation points for the finite element at time-step \( k \) and scenario \( b \) use:

```matlab
doctoc(\{mpc.opt_x['x',k,b,-1]\}+mpc.opt_x['x',k+1,b,:-1])
```

Due to the multi-stage formulation at any given time \( k \) we can have multiple future scenarios. However, there is only exactly one scenario that lead to the current node in the tree. Thus the collocation points associated to the finite element \( k \) lie in the past.

The concept is illustrated in the figure below:

---

**Note:** The attribute \( \text{opt}_x \) carries the scaled values of all variables.

---

**Note:** The attribute is populated when calling \( \text{setup()} \) or \( \text{prepare_nlp()} \).

This page is auto-generated. Page source is not available on Github.

### 5.11.4.1.10 \( \text{opt}_x \text{num} \)

Class attribute.

\( \text{MPC} \).\( \text{opt}_x \text{num} \)

Full MPC solution and initial guess.

This is the core attribute of the MPC class. It is used as the initial guess when solving the optimization problem and then overwritten with the current solution.

The attribute is a CasADi numeric structure with nested power indices. It can be indexed as follows:

```matlab
# dynamic states:
opt_x_num['x', time_step, scenario, collocation_point, _x_name]
# algebraic states:
opt_x_num['z', time_step, scenario, collocation_point, _z_name]
# inputs:
opt_x_num['u', time_step, scenario, _u_name]
# slack variables for soft constraints:
opt_x_num['eps', time_step, scenario, _nl_cons_name]
```

The names refer to those given in the \( \text{do_mpc.model.Model} \) configuration. Further indices are possible, if the variables are itself vectors or matrices.

The attribute can be used **to manually set a custom initial guess or for debugging purposes**.

How to query?

Querying the structure is more complicated than it seems at first look because of the scenario-tree used for robust MPC. To obtain all collocation points for the finite element at time-step \( k \) and scenario \( b \) use:

```matlab
doctoc(\{mpc.opt_x_num['x',k,b,-1]\}+mpc.opt_x_num['x',k+1,b,:-1])
```
Due to the multi-stage formulation at any given time $k$ we can have multiple future scenarios. However, there is only exactly one scenario that lead to the current node in the tree. Thus the collocation points associated to the finite element $k$ lie in the past.

The concept is illustrated in the figure below:

**Note:** The attribute `opt_x_num` carries the scaled values of all variables. See `opt_x_num_unscaled` for the unscaled values (these are not used as the initial guess).

**Warning:** Do not tweak or overwrite this attribute unless you known what you are doing.

**Note:** The attribute is populated when calling `setup()`

This page is auto-generated. Page source is not available on Github.

### 5.11.4.1.11 scaling

Class attribute.

**MPC.scaling**

Query and set scaling of the optimization variables. The `Optimizer.scaling()` method is an indexed property, meaning getting and setting this property requires an index and calls this function. The power index (elements are separated by commas) must contain at least the following elements:

<table>
<thead>
<tr>
<th>order</th>
<th>index name</th>
<th>valid options</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>variable type</td>
<td>_x, _u and _z (and _p_est for MHE)</td>
</tr>
<tr>
<td>2</td>
<td>variable name</td>
<td>Names defined in <code>do_mpc.model.Model</code></td>
</tr>
</tbody>
</table>

Further indices are possible (but not necessary) when the referenced variable is a vector or matrix.

**Example:**

```python
# Set with:
optimizer.scaling['\_x', 'phi\_1'] = 2
optimizer.scaling['\_x', 'phi\_2'] = 2

# Query with:
optimizer.scaling['\_x', 'phi\_1']
```

Scaling factors $a$ affect the MHE / MPC optimization problem. The optimization variables are scaled variables:

$$
\bar{\phi} = \frac{\phi}{\alpha_{\phi}} \quad \forall \phi \in [x, u, z, p_{est}]
$$

Scaled variables are used to formulate the bounds $\bar{\phi}_l \leq \bar{\phi}_u$ and for the evaluation of the ODE. For the objective function and the nonlinear constraints the unscaled variables are used. The algebraic equations are also not scaled.
Note: Scaling the optimization problem is suggested when states and/or inputs take on values which differ by orders of magnitude.

This page is auto-generated. Page source is not available on Github.

5.11.4.1.12 \texttt{t0}

Class attribute.

\texttt{MPC.t0}

Current time marker of the class. Use this property to set or query the time.

Set with \texttt{int, float, numpy.ndarray or casadi.DM} type.

This page is auto-generated. Page source is not available on Github.

5.11.4.1.13 \texttt{terminal_bounds}

Class attribute.

\texttt{MPC.terminal_bounds}

Query and set the terminal bounds for the states. The \texttt{terminal_bounds()} method is an indexed property, meaning getting and setting this property requires an index and calls this function. The power index (elements are separated by commas) must contain at least the following elements:

<table>
<thead>
<tr>
<th>order</th>
<th>index name</th>
<th>valid options</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>bound type</td>
<td>lower and upper</td>
</tr>
<tr>
<td>2</td>
<td>variable name</td>
<td>Names defined in do_mpc.model.Model.</td>
</tr>
</tbody>
</table>

Further indices are possible (but not necessary) when the referenced variable is a vector or matrix.

Example:

```python
# Set with:
optimizer.terminal_bounds[\{'lower', 'phi_1'\}] = -2*np.pi
optimizer.terminal_bounds[\{'upper', 'phi_1'\}] = 2*np.pi

# Query with:
optimizer.terminal_bounds[\{'lower', 'phi_1'\}]
```

This page is auto-generated. Page source is not available on Github.

5.11.4.1.14 \texttt{u0}

Class attribute.

\texttt{MPC.u0}

Initial input and current iterate. This is the numerical structure holding the information about the current input in the class. The property can be indexed according to the model definition.

Example:
model = do_mpc.model.Model('continuous')
model.set_variable('_u', 'heating', shape=(4,1))
...
mhe = do_mpc.estimator.MHE(model)
# or
mpc = do_mpc.estimator.MPC(model)

# Get or set current value of variable:
mpc.u0['heating', 0]  # 0th element of variable
mpc.u0['heating']     # all elements of variable
mpc.u0['heating', 0:2]  # 0th and 1st element

Useful CasADi symbolic structure methods:

- .shape
- .keys()
- .labels()

This page is auto-generated. Page source is not available on Github.

5.11.4.1.15 ub_opt_x

Class attribute.

MPC.ub_opt_x

Query and modify the lower bounds of all optimization variables opt_x. This is a more advanced method of setting bounds on optimization variables of the MPC/MHE problem. Users with less experience are advised to use bounds instead.

The attribute returns a nested structure that can be indexed using powerindexing. Please refer to opt_x for more details.

Note: The attribute automatically considers the scaling variables when setting the bounds. See scaling for more details.

Note: Modifications must be done after calling prepare_nlp() or setup() respectively.

This page is auto-generated. Page source is not available on Github.

5.11.4.1.16 x0

Class attribute.

MPC.x0

Initial state and current iterate. This is the numerical structure holding the information about the current states in the class. The property can be indexed according to the model definition.

Example:
model = do_mpc.model.Model('continuous')
model.set_variable('_x','temperature', shape=(4,1))

...

mhe = do_mpc.estimator.MHE(model)
# or
mpc = do_mpc.estimator.MPC(model)

# Get or set current value of variable:
mpc.x0['temperature', 0]  # 0th element of variable
mpc.x0['temperature']     # all elements of variable
mpc.x0['temperature', 0:2] # 0th and 1st element

Useful CasADi symbolic structure methods:

- .shape
- .keys()
- .labels()

This page is auto-generated. Page source is not available on Github.

5.11.4.17 z0

Class attribute.

MPC.z0

Initial algebraic state and current iterate. This is the numerical structure holding the information about the current algebraic states in the class. The property can be indexed according to the model definition.

Example:

```python
model = do_mpc.model.Model('continuous')
model.set_variable('_z','temperature', shape=(4,1))

...

mhe = do_mpc.estimator.MHE(model)
# or
mpc = do_mpc.estimator.MPC(model)

# Get or set current value of variable:
mpc.z0['temperature', 0]  # 0th element of variable
mpc.z0['temperature']     # all elements of variable
mpc.z0['temperature', 0:2] # 0th and 1st element
```

Useful CasADi symbolic structure methods:

- .shape
- .keys()
- .labels()

This page is auto-generated. Page source is not available on Github.

Methods
5.11.4.1.18 create_nlp

Class method.

do_mpc.controller.MPC.create_nlp(self)
Create the optimization problem. Typically, this method is called internally from setup().

Users should only call this method if they intend to modify the objective with nlp_obj, the constraints with nlp_cons, nlp_cons_lb and nlp_cons_ub.

To finish the setup process, users MUST call create_nlp() afterwards.

Note: Do NOT call setup() if you intend to go the manual route with prepare_nlp() and create_nlp().

Note: Only AFTER calling prepare_nlp() the previously mentioned attributes nlp_obj, nlp_cons, nlp_cons_lb, nlp_cons_ub become available.

This page is auto-generated. Page source is not available on Github.
This structure (with numerical values) should be used as the output of the \texttt{p\_fun} function which is set to the class with \texttt{set\_p\_fun()}.

Use the combination of \texttt{get\_p\_template()} and \texttt{set\_p\_template()} as a more adaptable alternative to \texttt{set\_uncertainty\_values()}.

\textbf{Note:} We advice less experienced users to use \texttt{set\_uncertainty\_values()} as an alternative way to configure the scenario-tree for robust multi-stage MPC.

\begin{verbatim}
# in model definition:
alpha = model.set_variable(var_type='\_p', var_name='alpha')
beta = model.set_variable(var_type='\_p', var_name='beta')

...# in MPC configuration:
n_combinations = 3
p_template = MPC.get_p_template(n_combinations)
p_template[\_p,0] = np.array([1,1])
p_template[\_p,1] = np.array([0.9, 1.1])
p_template[\_p,2] = np.array([1.1, 0.9])

def p_fun(t_now):
    return p_template
MPC.set_p_fun(p_fun)
\end{verbatim}

Note the nominal case is now: \texttt{alpha = 1 beta = 1} which is determined by the order in the arrays above (first element is nominal).

\textbf{Parameters} \texttt{n\_combinations} (int) – Define the number of combinations for the uncertain parameters for robust MPC.

\textbf{Returns} None

\textbf{Return type} None

This page is auto-generated. Page source is not available on Github.

\subsection{5.11.4.1.20 get\_tvp\_template}

Class method.

\begin{verbatim}
do_mpc.controller.MPC.get_tvp_template(self)
\end{verbatim}

The method returns a structured object with \texttt{n\_horizon+1} elements, and a set of time-varying parameters (as defined in \texttt{do_mpc.model.Model}) for each of these instances. The structure is initialized with all zeros. Use this object to define values of the time-varying parameters.

This structure (with numerical values) should be used as the output of the \texttt{tvp\_fun} function which is set to the class with \texttt{set\_tvp\_fun()}. Use the combination of \texttt{get\_tvp\_template()} and \texttt{set\_tvp\_fun()}.

\textbf{Example:}
In model definition:

```python
alpha = model.set_variable(var_type='_tvp', var_name='alpha')
beta = model.set_variable(var_type='_tvp', var_name='beta')
...
```

In optimizer configuration:

```python
tvp_temp_1 = optimizer.get_tvp_template()
tvp_temp_1['_tvp', :] = np.array([1,1])
tvp_temp_2 = optimizer.get_tvp_template()
tvp_temp_2['_tvp', :] = np.array([0,0])

def tvp_fun(t_now):
    if t_now<10:
        return tvp_temp_1
    else:
        tvp_temp_2
optimizer.set_tvp_fun(tvp_fun)
```

Returns

None

Return type

None

This page is auto-generated. Page source is not available on Github.

5.11.4.1.21 make_step

Class method.

```python
do_mpc.controller.MPC.make_step(self, x0)
```

Main method of the class during runtime. This method is called at each timestep and returns the control input for the current initial state `x0`.

The method prepares the MHE by setting the current parameters, calls `solve()` and updates the `do_mpc.data.Data` object.

Parameters

- `x0` (numpy.ndarray or casadi.DM) – Current state of the system.

Returns

`u0`

Return type

numpy.ndarray

This page is auto-generated. Page source is not available on Github.

5.11.4.1.22 prepare_nlp

Class method.

```python
do_mpc.controller.MPC.prepare_nlp(self)
```

Prepare the optimization problem. Typically, this method is called internally from `setup()`.

Users should only call this method if they intend to modify the objective with `nlp_obj`, the constraints with `nlp_cons`, `nlp_cons_lb` and `nlp_cons_ub`.

To finish the setup process, users MUST call `create_nlp()` afterwards.
Note: Do NOT call `setup()` if you intend to go the manual route with `prepare_nlp()` and `create_nlp()`.

Note: Only AFTER calling `prepare_nlp()` the previously mentioned attributes `nlp_obj`, `nlp_cons`, `nlp_cons_lb`, `nlp_cons_ub` become available.

This page is auto-generated. Page source is not available on Github.

### 5.11.4.1.23 reset_history

Class method.

```python
do_mpc.controller.MPC.reset_history(self)
```

Reset the history of the optimizer. All data from the `do_mpc.data.Data` instance is removed.

This page is auto-generated. Page source is not available on Github.

### 5.11.4.1.24 set_initial_guess

Class method.

```python
do_mpc.controller.MPC.set_initial_guess(self)
```

Initial guess for optimization variables. Uses the current class attributes `x0`, `z0` and `u0` to create the initial guess. The initial guess is simply the initial values for all $k = 0, \ldots, N$ instances of $x_k, u_k$ and $z_k$.

**Warning:** If no initial values for `x0`, `z0` and `u0` were supplied during setup, these default to zero.

**Note:** The initial guess is fully customizable by directly setting values on the class attribute: `opt_x_num`.

This page is auto-generated. Page source is not available on Github.

### 5.11.4.1.25 set_nl_cons

Class method.

```python
do_mpc.controller.MPC.set_nl_cons(self, expr_name, expr, ub=inf, soft_constraint=False, penalty_term_cons=1, maximum_violation=inf)
```

Introduce new constraint to the class. Further constraints are optional. Expressions must be formulated with respect to `_x`, `_u`, `_z`, `_tvp`, `_p`. They are implemented as:

$$ m(x, u, z, p_v, p) \leq m_{ub} $$

Setting the flag `soft_constraint=True` will introduce slack variables $\epsilon$, such that:

$$ m(x, u, z, p_v, p) - \epsilon \leq m_{ub}, $$

$$ 0 \leq \epsilon \leq \epsilon_{max}, $$

5.11.  API Reference 109
Slack variables are added to the cost function and multiplied with the supplied penalty term. This formulation makes constraints soft, meaning that a certain violation is tolerated and does not lead to infeasibility. Typically, high values for the penalty are suggested to avoid significant violation of the constraints.

**Parameters**

- `expr_name (string)` – Arbitrary name for the given expression. Names are used for key word indexing.
- `expr (CasADi SX or MX)` – CasADi SX or MX function depending on `_x`, `_u`, `_z`, `_tvp`, `_p`.

**Raises**

- `assertion` – `expr_name` must be str
- `assertion` – `expr` must be a casadi SX or MX type

**Returns**

Returns the newly created expression. Expression can be used e.g. for the RHS.

**Return type**

`casadi.SX` or `casadi.MX`

This page is auto-generated. Page source is not available on Github.

### 5.11.4.1.26 set_objective

Class method:

```python
do_mpc.controller.MPC.set_objective(self, mterm=None, lterm=None)
```

Sets the objective of the optimal control problem (OCP). We introduce the following cost function:

\[
J(x, u, z) = \sum_{k=0}^{N} \left( l(x_k, z_k, u_k, p_{k, k}) + \Delta u_k^T R \Delta u_k \right) + m(x_{N+1})
\]

which is applied to the discrete-time model **AND** the discretized continuous-time model. For discretization we use orthogonal collocation on finite elements. The cost function is evaluated only on the first collocation point of each interval.

`set_objective()` is used to set the \( l(x_k, z_k, u_k, p_{k, k}) \) (lterm) and \( m(x_{N+1}) \) (mterm), where \( N \) is the prediction horizon. Please see `set_rterm()` for the penalization of the control inputs.

**Parameters**

- `lterm (CasADi SX or MX)` – Stage cost - scalar symbolic expression with respect to `_x`, `_u`, `_z`, `_tvp`, `_p`
- `mterm (CasADi SX or MX)` – Terminal cost - scalar symbolic expression with respect to `_x` and `_p`

**Raises**

- `assertion` – `mterm` must have `shape=(1,1)` (scalar expression)
- `assertion` – `lterm` must have `shape=(1,1)` (scalar expression)

**Returns**

None

**Return type**

None

This page is auto-generated. Page source is not available on Github.
5.11.4.1.27 set_p_fun

Class method.

do_mpc.controller.MPC.set_p_fun(self, p_fun)

Set function which returns parameters. The `p_fun` is called at each optimization step to get the current values of the (uncertain) parameters.

This is the low-level API method to set user defined scenarios for robust multi-stage MPC by defining an arbitrary number of combinations for the parameters defined in the model. For more details on robust multi-stage MPC please read our background article.

The method takes as input a function, which MUST return a structured object, based on the defined parameters and the number of combinations. The defined function has time as a single input.

Obtain this structured object first, by calling `get_p_template()`.

Use the combination of `get_p_template()` and `set_p_fun()` as a more adaptable alternative to `set_uncertainty_values()`.

**Note:** We advise less experienced users to use `set_uncertainty_values()` as an alternative way to configure the scenario-tree for robust multi-stage MPC.

**Example:**

```python
# in model definition:
alpha = model.set_variable(var_type='p', var_name='alpha')
beta = model.set_variable(var_type='p', var_name='beta')
...

# in MPC configuration:
n_combinations = 3
p_template = MPC.get_p_template(n_combinations)
p_template["p",0] = np.array([1,1])
p_template["p",1] = np.array([0.9, 1.1])
p_template["p",2] = np.array([1.1, 0.9])

def p_fun(t_now):
    return p_template

MPC.set_p_fun(p_fun)
```

Note the nominal case is now: `alpha = 1, beta = 1` which is determined by the order in the arrays above (first element is nominal).

**Parameters**

- **p_fun** (`function`) – Function which returns a structure with numerical values.
  - Must be the same structure as obtained from `get_p_template()`. Function must have a single input (time).

**Returns** None

**Return type** None

---

5.11.4.1.28 set_param

Class method.
do_mpc.controller.MPC.set_param(self, **kwargs)

Set the parameters of the MPC class. Parameters must be passed as pairs of valid keywords and respective argument. For example:

```python
cmpc.set_param(n_horizon = 20)
```

It is also possible and convenient to pass a dictionary with multiple parameters simultaneously as shown in the following example:

```python
setup_mpc = {
    'n_horizon': 20,
    't_step': 0.5,
}
mpc.set_param(**setup_mpc)
```

This makes use of the python “unpack” operator. See more details here.

**Note:** The only required parameters are `n_horizon` and `t_step`. All other parameters are optional.

**Note:** `set_param()` can be called multiple times. Previously passed arguments are overwritten by successive calls.

The following parameters are available:

**Parameters**

- **n_horizon (int)** – Prediction horizon of the optimal control problem. Parameter must be set by user.
- **n_robust (int, optional)** – Robust horizon for robust scenario-tree MPC, defaults to 0. Optimization problem grows exponentially with `n_robust`.
- **open_loop (bool, optional)** – Setting for scenario-tree MPC: If the parameter is False, for each timestep AND scenario an individual control input is computed. If set to True, the same control input is used for each scenario. Defaults to False.
- **t_step (float)** – Timestep of the mpc.
- **use_terminal_bounds (bool)** – Choose if terminal bounds for the states are used. Defaults to True. Set terminal bounds with `terminal_bounds`.
- **state_discretization (str)** – Choose the state discretization for continuous models. Currently only 'collocation' is available. Defaults to 'collocation'. Has no effect if model is created in discrete type.
- **collocation_type (str)** – Choose the collocation type for continuous models with collocation as state discretization. Currently only 'radau' is available. Defaults to 'radau'.
- **collocation_deg (int)** – Choose the collocation degree for continuous models with collocation as state discretization. Defaults to 2.
- **collocation_ni (int)** – For orthogonal collocation choose the number of finite elements for the states within a time-step (and during constant control input). Defaults to 1. Can be used to avoid high-order polynomials.
- **nl_cons_check_colloc_points** *(bool)* – For orthogonal collocation choose whether the nonlinear bounds set with `set_nl_cons()` are evaluated once per finite Element or for each collocation point. Defaults to `False` (once per collocation point).

- **nl_cons_single_slack** *(bool)* – If `True`, soft-constraints set with `set_nl_cons()` introduce only a single slack variable for the entire horizon. Defaults to `False`.

- **cons_check_colloc_points** *(bool)* – For orthogonal collocation choose whether the linear bounds set with `bounds` are evaluated once per finite Element or for each collocation point. Defaults to `True` (for all collocation points).

- **store_full_solution** *(bool)* – Choose whether to store the full solution of the optimization problem. This is required for animating the predictions in post processing. However, it drastically increases the required storage. Defaults to `False`.

- **store_lagr_multiplier** *(bool)* – Choose whether to store the lagrange multipliers of the optimization problem. Increases the required storage. Defaults to `True`.

- **store_solver_stats** *(dict)* – Choose which solver statistics to store. Must be a list of valid statistics. Defaults to `['success','t_wall_total']`.

- **nlpsol_opts** – Dictionary with options for the CasADi solver call `nlpsol` with plugin `ipopt`. All options are listed [here](#).

**Note:** We highly suggest to change the linear solver for IPOPT from `mumps` to `MA27`. In many cases this will drastically boost the speed of `do-mpc`. Change the linear solver with:

```python
MPC.set_param(nlpsol_opts = {'ipopt.linear_solver': 'MA27'})
```

**Note:** To suppress the output of IPOPT, please use:

```python
suppress_ipopt = {'ipopt.print_level':0, 'ipopt.sb': 'yes', 'print_time':0}
MPC.set_param(nlpsol_opts = suppress_ipopt)
```

This page is auto-generated. Page source is not available on Github.

### 5.11.4.1.29 set_rterm

Class method.

```python
do_mpc.controller.MPC.set_rterm(self, **kwargs)
```

Set the penality factor for the inputs. Call this function with keyword argument refering to the input names in `model` and the penalty factor as the respective value.

We define for $i \in I$, where $I$ is the set of inputs and all $k = 0, \ldots, N$ where $N$ denotes the horizon:

$$
\Delta u_{k,i} = u_{k,i} - u_{k-1,i}
$$

and add:

$$
\sum_{k=0}^{N} \sum_{i \in I} r_i \Delta u_{k,i}^2,
$$

the weighted squared cost to the MPC objective function.
Example:

```python
# in model definition:
Q_heat = model.set_variable(var_type='_u', var_name='Q_heat')
F_flow = model.set_variable(var_type='_u', var_name='F_flow')
...
# in MPC configuration:
MPC.set_rterm(Q_heat = 10)
MPC.set_rterm(F_flow = 10)
# or alternatively:
MPC.set_rterm(Q_heat = 10, F_flow = 10)
```

In the above example we set $r_{Q_{\text{heat}}} = 10$ and $r_{F_{\text{flow}}} = 10$.

**Note:** For $k = 0$ we obtain $u_{-1}$ from the previous solution.

This page is auto-generated. Page source is not available on Github.

### 5.11.4.1.30 set_tvp_fun

Class method.

do_mpc.controller.MPC.set_tvp_fun(self, tvp_fun)

Set function which returns time-varying parameters.

The `tvp_fun` is called at each optimization step to get the current prediction of the time-varying parameters. The supplied function must be callable with the current time as the only input. Furthermore, the function must return a CasADi structured object which is based on the horizon and on the model definition. The structure can be obtained with `get_tvp_template()`.

Example:

```python
# in model definition:
alpha = model.set_variable(var_type='_tvp', var_name='alpha')
beta = model.set_variable(var_type='_tvp', var_name='beta')
...
# in optimizer configuration:
tvp_temp_1 = optimizer.get_tvp_template()
tvp_temp_1['_tvp', :] = np.array([1,1])
tvp_temp_2 = optimizer.get_tvp_template()
tvp_temp_2['_tvp', :] = np.array([0,0])

def tvp_fun(t_now):
    if t_now<10:
        return tvp_temp_1
    else:
        tvp_temp_2
optimizer.set_tvp_fun(tvp_fun)
```

**Note:** The method `set_tvp_fun()` must be called prior to setup IF time-varying parameters are
defined in the model. It is not required to call the method if no time-varying parameters are defined.

**Parameters** tvp_fun *(function)* – Function that returns the predicted tvp values at each timestep. Must have single input (float) and return a `structure3.DMStruct` (obtained with `get_tvp_template()`).

This page is auto-generated. Page source is not available on Github.

### 5.11.4.1.31 set_uncertainty_values

Class method.

```python
do_mpc.controller.MPC.set_uncertainty_values(self, **kwargs)
```

Define scenarios for the uncertain parameters. High-level API method to conveniently set all possible scenarios for multistage MPC. For more details on robust multi-stage MPC please read our background article.

Pass a number of keyword arguments, where each keyword refers to a user defined parameter name from the model definition. The value for each parameter must be an array (or list), with an arbitrary number of possible values for this parameter. The first element is the nominal case.

**Example:**

```python
# in model definition:
alpha = model.set_variable(var_type='_p', var_name='alpha')
beta = model.set_variable(var_type='_p', var_name='beta')
gamma = model.set_variable(var_type='_p', var_name='gamma')
...
# in MPC configuration:
alpha_var = np.array([1., 0.9, 1.1])
beta_var = np.array([1., 1.05])
MPC.set_uncertainty_values(
    alpha = alpha_var,
    beta = beta_var
)
```

**Note:** Parameters that are not important for the MPC controller (e.g. MHE tuning matrices) can be ignored with the new interface (see `gamma` in the example above).

Note the nominal case is now: `alpha = 1, beta = 1` which is determined by the order in the arrays above (first element is nominal).

**Parameters** kwargs – Arbitrary number of keyword arguments.

**Returns** None

**Return type** None

This page is auto-generated. Page source is not available on Github.

### 5.11.4.1.32 setup

Class method.
do_mpc.controller.MPC.setup(self)
    Setup the MPC class. Internally, this method will create the MPC optimization problem under consideration of the supplied dynamic model and the given MPC class instance configuration.

    The setup() method can be called again after changing the configuration (e.g. adapting bounds) and will simply overwrite the previous optimization problem.

    **Note:** After this call, the solve() and make_step() method is applicable.

    **Warning:** The setup() method may take a while depending on the size of your MPC problem. Note that especially for robust multi-stage MPC with a long robust horizon and many possible combinations of the uncertain parameters very large problems will arise.

    For more details on robust multi-stage MPC please read our background article.

This page is auto-generated. Page source is not available on Github.

5.11.4.1.33 solve

Class method.

do_mpc.controller.MPC.solve(self)
    Solves the optimization problem.

    The current problem is defined by the parameters in the opt_p_num CasADi structured Data.

    Typically, opt_p_num is prepared for the current iteration in the make_step() method. It is, however, valid and possible to directly set parameters in opt_p_num before calling solve().

    The method updates the opt_p_num and opt_x_num attributes of the class. By resetting opt_x_num to the current solution, the method implicitly enables warmstarting the optimizer for the next iteration, since this vector is always used as the initial guess.

    **Warning:** The method is part of the public API but it is generally not advised to use it. Instead we recommend to call make_step() at each iterations, which acts as a wrapper for solve().

    **Raises** assertion – Optimizer was not setup yet.

    **Returns** None

    **Return type** None

This page is auto-generated. Page source is not available on Github.

5.11.5 estimator

Classes
**5.11.5.1 EKF**

_class do_mpc.estimator.EKF (model)_

Extended Kalman Filter. Setup this class and use :meth:`EKF.make_step` during runtime to obtain the currently estimated states given the measurements \( y_0 \).

**Warning:** Not currently implemented.

**Attributes**

<table>
<thead>
<tr>
<th>EKF.t0</th>
<th>Current time marker of the class.</th>
</tr>
</thead>
<tbody>
<tr>
<td>EKF.u0</td>
<td>Initial input and current iterate.</td>
</tr>
<tr>
<td>EKF.x0</td>
<td>Initial state and current iterate.</td>
</tr>
<tr>
<td>EKF.z0</td>
<td>Initial algebraic state and current iterate.</td>
</tr>
</tbody>
</table>

**5.11.5.1.1 t0**

Class attribute.

**EKF.t0**

Current time marker of the class. Use this property to set or query the time.


**5.11.5.1.2 u0**

Class attribute.

**EKF.u0**

Initial input and current iterate. This is the numerical structure holding the information about the current input in the class. The property can be indexed according to the model definition.

**Example:**

```python
model = do_mpc.model.Model('continuous')
model.set_variable('_u', 'heating', shape=(4,1))
...

mhe = do_mpc.estimator.MHE(model)
# or
mpc = do_mpc.estimator.MPC(model)

# Get or set current value of variable:
mpc.u0['heating', 0] # 0th element of variable
```
Useful CasADi symbolic structure methods:

- .shape
- .keys()
- .labels()

This page is auto-generated. Page source is not available on Github.

### 5.11.5.1.3 \textbf{x0}

Class attribute.

\textbf{EKF.} \textbf{x0}

Initial state and current iterate. This is the numerical structure holding the information about the current states in the class. The property can be indexed according to the model definition.

\textbf{Example:}

```python
model = do_mpc.model.Model('continuous')
model.set_variable('_x', 'temperature', shape=(4,1))
...
```

```python
mhe = do_mpc.estimator.MHE(model)
# or
mpc = do_mpc.estimator.MPC(model)

# Get or set current value of variable:
mpc.x0['temperature', 0] # 0th element of variable
mpc.x0['temperature'] # all elements of variable
mpc.x0['temperature', 0:2] # 0th and 1st element
```

Useful CasADi symbolic structure methods:

- .shape
- .keys()
- .labels()

This page is auto-generated. Page source is not available on Github.

### 5.11.5.1.4 \textbf{z0}

Class attribute.

\textbf{EKF.} \textbf{z0}

Initial algebraic state and current iterate. This is the numerical structure holding the information about the current algebraic states in the class. The property can be indexed according to the model definition.

\textbf{Example:}
```python
model = do_mpc.model.Model('continuous')
model.set_variable('_z','temperature', shape=(4,1))

... 
mhe = do_mpc.estimator.MHE(model)
# or
mpc = do_mpc.estimator.MPC(model)

# Get or set current value of variable:
mpc.z0['temperature', 0]  # 0th element of variable
mpc.z0['temperature']    # all elements of variable
mpc.z0['temperature', 0:2]  # 0th and 1st element
```

Useful CasADi symbolic structure methods:

- `.shape`
- `.keys()`
- `.labels()`

This page is auto-generated. Page source is not available on Github.

## Methods

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>EKF.make_step</code></td>
<td>Main method during runtime.</td>
</tr>
<tr>
<td><code>EKF.reset_history</code></td>
<td>Reset the history of the estimator</td>
</tr>
</tbody>
</table>

### 5.11.5.1.5 make_step

Class method.

```python
do_mpc.estimator.EKF.make_step(self, y0)
```

Main method during runtime. Pass the most recent measurement and retrieve the estimated state.

This page is auto-generated. Page source is not available on Github.

### 5.11.5.1.6 reset_history

Class method.

```python
do_mpc.estimator.EKF.reset_history(self)
```

Reset the history of the estimator

This page is auto-generated. Page source is not available on Github.

## 5.11.5.2 Estimator

```python
class do_mpc.estimator.Estimator(model)
```

The `Estimator` base class. Used for `StateFeedback`, `EKF` and `MHE`. This class cannot be used independently.
Note: The methods `Estimator.set_initial_state()` and `Estimator.reset_history()` are overwritten when using the MHE by the methods defined in `do_mpc.optimizer.Optimizer`.

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Estimator.t0</code></td>
<td>Current time marker of the class.</td>
</tr>
<tr>
<td><code>Estimator.u0</code></td>
<td>Initial input and current iterate.</td>
</tr>
<tr>
<td><code>Estimator.x0</code></td>
<td>Initial state and current iterate.</td>
</tr>
<tr>
<td><code>Estimator.z0</code></td>
<td>Initial algebraic state and current iterate.</td>
</tr>
</tbody>
</table>

5.11.5.2.1 t0

Class attribute.

`Estimator.t0`

Current time marker of the class. Use this property to set or query the time.

Set with `int`, `float`, `numpy.ndarray` or `casadi.DM` type.

This page is auto-generated. Page source is not available on Github.

5.11.5.2.2 u0

Class attribute.

`Estimator.u0`

Initial input and current iterate. This is the numerical structure holding the information about the current input in the class. The property can be indexed according to the model definition.

Example:

```python
model = do_mpc.model.Model('continuous')
model.set_variable('_u','heating', shape=(4,1))
...

mhe = do_mpc.estimator.MHE(model)
# or
mpc = do_mpc.estimator.MPC(model)

# Get or set current value of variable:
mpc.u0['heating', 0] # 0th element of variable
mpc.u0['heating'] # all elements of variable
mpc.u0['heating', 0:2] # 0th and 1st element
```

Useful CasADi symbolic structure methods:

- `shape`
- `keys()`
- `labels()`

This page is auto-generated. Page source is not available on Github.
5.11.5.2.3 \texttt{x0}

Class attribute.

\texttt{Estimator.x0}

Initial state and current iterate. This is the numerical structure holding the information about the current states in the class. The property can be indexed according to the model definition.

\textbf{Example:}

```python
model = do_mpc.model.Model('continuous')
model.set_variable('_x','temperature', shape=(4,1))
...

mhe = do_mpc.estimator.MHE(model)
# or
mpc = do_mpc.estimator.MPC(model)

# Get or set current value of variable:
mhe.x0['temperature', 0] # 0th element of variable
mpc.x0['temperature']    # all elements of variable
mpc.x0['temperature', 0:2] # 0th and 1st element
```

Useful CasADi symbolic structure methods:

- .shape
- .keys()
- .labels()

This page is auto-generated. Page source is not available on Github.

5.11.5.2.4 \texttt{z0}

Class attribute.

\texttt{Estimator.z0}

Initial algebraic state and current iterate. This is the numerical structure holding the information about the current algebraic states in the class. The property can be indexed according to the model definition.

\textbf{Example:}

```python
model = do_mpc.model.Model('continuous')
model.set_variable('_z','temperature', shape=(4,1))
...

mhe = do_mpc.estimator.MHE(model)
# or
mpc = do_mpc.estimator.MPC(model)

# Get or set current value of variable:
mhe.z0['temperature', 0] # 0th element of variable
mpc.z0['temperature']    # all elements of variable
mpc.z0['temperature', 0:2] # 0th and 1st element
```

Useful CasADi symbolic structure methods:

- .shape
Methods

**Estimator.reset_history** Reset the history of the estimator

### 5.11.5.2.5 reset_history

Class method.

```python
do_mpc.estimator.Estimator.reset_history(self)
```

Reset the history of the estimator

---

### 5.11.5.3 MHE

**class do_mpc.estimator.MHE***(model, p_est_list=[])**

Moving horizon estimator.

For general information on moving horizon estimation, please read our [background article](https://example.com).

The MHE estimator extends the `do_mpc.optimizer.Optimizer` base class (which is also used for `do_mpc.controller.MPC`), as well as the `Estimator` base class. Use this class to configure and run the MHE based on a previously configured `do_mpc.model.Model` instance.

The class is initiated by passing a list of the **parameters that should be estimated**. This must be a subset (or all) of the parameters defined in `do_mpc.model.Model`. This allows to define parameters in the model that influence the model externally (e.g. weather predictions), and those that are internal e.g. system parameters and can be estimated. Passing an empty list (default) value, means that no parameters are estimated.

**Note**: Parameters are influencing the model equation at all timesteps but are constant over the entire horizon. Parameters could also be introduced as states without dynamic but this would increase the total number of optimization variables.

**Configuration and setup:**

Configuring and setting up the MHE involves the following steps:

1. Use `set_param()` to configure the MHE. See docstring for details.
2. Set the objective of the control problem with `set_default_objective()` or use the low-level interface `set_objective()`.
3. Set upper and lower bounds.
4. Optionally, set further (non-linear) constraints with `set_nl_cons()`.
5. Use `get_p_template()` and `set_p_fun()` to set the function for the (not estimated) parameters.
8. Use `get_tvp_template()` and `set_tvp_fun()` to create a method to obtain new time-varying parameters at each iteration.

9. To finalize the class configuration there are two routes. The default approach is to call `setup()`. For deep customization use the combination of `prepare_nlp()` and `create_nlp()`. See graph below for an illustration of the process.

![Fig. 6: Route to setting up the MHE class.](image)

**Warning:** Before running the estimator, make sure to supply a valid initial guess for all estimated variables (states, algebraic states, inputs and parameters). Simply set the initial values of $x_0, z_0, u_0$ and $p_{est0}$ and then call `set_initial_guess()`.

To take full control over the initial guess, modify the values of `opt_x_num`.

During runtime use `make_step()` with the most recent measurement to obtain the estimated states.

**Parameters**

- `model (do_mpc.model.Model)` – A configured and setup `do_mpc.model.Model`
- `p_est_list (list)` – List with names of parameters (`_p`) defined in `model`

**Attributes**

- `MHE.bounds` Query and set bounds of the optimization variables.
- `MHE.lb_opt_x` Query and modify the lower bounds of all optimization variables `opt_x`.
- `MHE.nlp_cons` Query and modify (symbolically) the NLP constraints.
- `MHE.nlp_cons_lb` Query and modify the lower bounds of the `nlp_cons`.
- `MHE.nlp_cons_ub` Query and modify the upper bounds of the `nlp_cons`.

Continued on next page
Table 21 – continued from previous page

<table>
<thead>
<tr>
<th>MHE.nlp_obj</th>
<th>Query and modify (symbolically) the NLP objective function.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MHE.opt_p</td>
<td>Full structure of (symbolic) MHE parameters.</td>
</tr>
<tr>
<td>MHE.opt_p_num</td>
<td>Full MHE parameter vector.</td>
</tr>
<tr>
<td>MHE.opt_x</td>
<td>Full structure of the (symbolic) MHE optimization variables.</td>
</tr>
<tr>
<td>MHE.opt_x_num</td>
<td>Full MHE solution and initial guess.</td>
</tr>
<tr>
<td>MHE.p_est0</td>
<td>Initial value of estimated parameters and current iterate.</td>
</tr>
<tr>
<td>MHE.scaling</td>
<td>Query and set scaling of the optimization variables.</td>
</tr>
<tr>
<td>MHE.t0</td>
<td>Current time marker of the class.</td>
</tr>
<tr>
<td>MHE.u0</td>
<td>Initial input and current iterate.</td>
</tr>
<tr>
<td>MHE.ub_opt_x</td>
<td>Query and modify the lower bounds of all optimization variables opt_x.</td>
</tr>
<tr>
<td>MHE.x0</td>
<td>Initial state and current iterate.</td>
</tr>
<tr>
<td>MHE.z0</td>
<td>Initial algebraic state and current iterate.</td>
</tr>
</tbody>
</table>

### 5.11.5.3.1 bounds

Class attribute.

**MHE.bounds**

Query and set bounds of the optimization variables. The `bounds()` method is an indexed property, meaning getting and setting this property requires an index and calls this function. The power index (elements are separated by commas) must contain at least the following elements:

<table>
<thead>
<tr>
<th>order</th>
<th>index name</th>
<th>valid options</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>bound type</td>
<td>lower and upper</td>
</tr>
<tr>
<td>2</td>
<td>variable type _x, _u and _z (and _p_est for MHE)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>variable name</td>
<td>Names defined in do_mpc.model.Model.</td>
</tr>
</tbody>
</table>

Further indices are possible (but not necessary) when the referenced variable is a vector or matrix.

**Example:**

```python
# Set with:
optimizer.bounds['lower', '_x', 'phi_1'] = -2*np.pi
optimizer.bounds['upper', '_x', 'phi_1'] = 2*np.pi

# Query with:
optimizer.bounds['lower', '_x', 'phi_1']
```

This page is auto-generated. Page source is not available on Github.

### 5.11.5.3.2 lb_opt_x

Class attribute.

**MHE.lb_opt_x**

Query and modify the lower bounds of all optimization variables `opt_x`. This is a more advanced method of setting bounds on optimization variables of the MPC/MHE problem. Users with less experience are advised to use `bounds` instead.
The attribute returns a nested structure that can be indexed using powerindexing. Please refer to `opt_x` for more details.

**Note:** The attribute automatically considers the scaling variables when setting the bounds. See `scaling` for more details.

**Note:** Modifications must be done after calling `prepare_nlp()` or `setup()` respectively.

This page is auto-generated. Page source is not available on Github.

### 5.11.5.3.3 nlp_cons

Class attribute.

**MHE.nlp_cons**

Query and modify (symbolically) the NLP constraints. Use the variables in `opt_x` and `opt_p`.

Prior to calling `create_nlp()` this attribute returns a list of symbolic constraints. After calling `create_nlp()` this attribute returns the concatenation of this list and the attribute cannot be altered anymore.

It is advised to append to the current list of `nlp_cons`:

```python
mpc.prepare_nlp()

# Create new constraint: Input at timestep 0 and 1 must be identical.
extra_cons = mpc.opt_x['_u', 0, 0] - mpc.opt_x['_u', 1, 0]
mpc.nlp_cons.append(extra_cons)

# Create appropriate upper and lower bound (here they are both 0 to create an equality constraint)
mpc.nlp_cons_lb.append(np.zeros(extra_cons.shape))
mpc.nlp_cons_ub.append(np.zeros(extra_cons.shape))

mpc.create_nlp()
```

See the documentation of `opt_x` and `opt_p` on how to query these attributes.

**Warning:** This is a VERY low level feature and should be used with extreme caution. It is easy to break the code.

Be especially careful NOT to accidentally overwrite the default objective.

**Note:** Modifications must be done after calling `prepare_nlp()` and before calling `create_nlp()`

This page is auto-generated. Page source is not available on Github.
5.11.5.3.4 nlp_cons_lb

Class attribute.

MHE. nlp_cons_lb
Query and modify the lower bounds of the nlp_cons.

Prior to calling create_nlp() this attribute returns a list of lower bounds matching the list of constraints obtained with nlp_cons. After calling create_nlp() this attribute returns the concatenation of this list.

Values for lower (and upper) bounds MUST be added when adding new constraints to nlp_cons.

Warning: This is a VERY low level feature and should be used with extreme caution. It is easy to break the code.

Note: Modifications must be done after calling prepare_nlp()
mpc.prepare_nlp()
# Modify the objective
mpc.nlp_obj += sum1(vertcat(*mpc.opt_x['_x', -1, 0])**2)
# Finish creating the NLP
mpc.create_nlp()

See the documentation of `opt_x` and `opt_p` on how to query these attributes.

**Warning:** This is a **VERY** low level feature and should be used with extreme caution. It is easy to break the code.

Be especially careful NOT to accidentally overwrite the default objective.

**Note:** Modifications must be done after calling `prepare_nlp()` and before calling `create_nlp()`

This page is auto-generated. Page source is not available on Github.

### 5.11.5.3.7 opt_p

Class attribute.

**MHE.** `opt_p`

Full structure of (symbolic) MHE parameters.

The attribute can be used to alter the objective function or constraints of the NLP.

The attribute is a CasADi numeric structure with nested power indices. It can be indexed as follows:

```python
# previously estimated state:
opt_p['_x_prev', _x_name]
# previously estimated parameters:
opt_p['_p_est_prev', _x_name]
# known parameters
opt_p['_p_set', _p_name]
# time-varying parameters:
opt_p['_tvp', time_step, _tvp_name]
# sequence of measurements:
opt_p['_y_meas', time_step, _y_name]
```

The names refer to those given in the `do_mpc.model.Model` configuration. Further indices are possible, if the variables are itself vectors or matrices.

**Warning:** Do not tweak or overwrite this attribute unless you known what you are doing.

**Note:** The attribute is populated when calling `setup()` or `create_nlp()`.

This page is auto-generated. Page source is not available on Github.
5.11.5.3.8 opt_p_num

Class attribute.

MHE.opt_p_num

Full MHE parameter vector.

This attribute is used when calling the solver to pass all required parameters, including

- previously estimated state(s)
- previously estimated parameter(s)
- known parameters
- sequence of time-varying parameters
- sequence of measurements parameters

do-mpc handles setting these parameters automatically in the make_step() method. However, you can set these values manually and directly call solve().

The attribute is a CasADi numeric structure with nested power indices. It can be indexed as follows:

```python
# previously estimated state:
opt_p_num['x_prev', _x_name]
# previously estimated parameters:
opt_p_num['p_est_prev', _x_name]
# known parameters
opt_p_num['p_set', _p_name]
# time-varying parameters:
opt_p_num['tvp', time_step, _tvp_name]
# sequence of measurements:
opt_p_num['y_meas', time_step, _y_name]
```

The names refer to those given in the do_mpc.model.Model configuration. Further indices are possible, if the variables are itself vectors or matrices.

**Warning:** Do not tweak or overwrite this attribute unless you known what you are doing.

**Note:** The attribute is populated when calling setup().

This page is auto-generated. Page source is not available on Github.

5.11.5.3.9 opt_x

Class attribute.

MHE.opt_x

Full structure of the (symbolic) MHE optimization variables.

The attribute is a CasADi numeric structure with nested power indices. It can be indexed as follows:

```python
# dynamic states:
opt_x['x', time_step, collocation_point, _x_name]
# algebraic states:
```
The names refer to those given in the `do_mpc.model.Model` configuration. Further indices are possible, if the variables are itself vectors or matrices.

The attribute can be used to alter the objective function or constraints of the NLP.

**Note:** The attribute `opt_x` carries the scaled values of all variables.

**Warning:** Do not tweak or overwrite this attribute unless you known what you are doing.

**Note:** The attribute is populated when calling `setup()` or `prepare_nlp()`.

This page is auto-generated. Page source is not available on Github.

### 5.11.5.3.10 `opt_x_num`

Class attribute.

**MHE. opt_x_num**

Full MHE solution and initial guess.

This is the core attribute of the MHE class. It is used as the initial guess when solving the optimization problem and then overwritten with the current solution.

The attribute is a CasADi numeric structure with nested power indices. It can be indexed as follows:

```python
# dynamic states:
opt_x_num['x', time_step, collocation_point, _x_name]
# algebraic states:
opt_x_num['z', time_step, collocation_point, _z_name]
# inputs:
opt_x_num['u', time_step, _u_name]
# estimated parameters:
opt_x_NUM['p_est', _p_names]
# slack variables for soft constraints:
opt_x_num['eps', time_step, _nl_cons_name]
```

The names refer to those given in the `do_mpc.model.Model` configuration. Further indices are possible, if the variables are itself vectors or matrices.

The attribute can be used to manually set a custom initial guess or for debugging purposes.

**Note:** The attribute `opt_x_num` carries the scaled values of all variables. See `opt_x_num_unscaled`.
for the unscaled values (these are not used as the initial guess).

**Warning:** Do not tweak or overwrite this attribute unless you known what you are doing.

**Note:** The attribute is populated when calling `setup()`

This page is auto-generated. Page source is not available on Github.

### 5.11.5.3.11 p_est0

Class attribute.

**MHE.p_est0**

Initial value of estimated parameters and current iterate. This is the numerical structure holding the information about the current estimated parameters in the class. The property can be indexed according to the model definition.

**Example:**

```python
model = do_mpc.model.Model('continuous')
model.set_variable('_p', 'temperature', shape=(4,1))

# Initiate MHE with list of estimated parameters:
mhe = do_mpc.estimator.MHE(model, ['temperature'])

# Get or set current value of variable:
mhe.p_est0['temperature', 0] # 0th element of variable
mhe.p_est0['temperature'] # all elements of variable
mhe.p_est0['temperature', 0:2] # 0th and 1st element
```

Usefull CasADi symbolic structure methods:

- `.shape`
- `.keys()`
- `.labels()`

This page is auto-generated. Page source is not available on Github.

### 5.11.5.3.12 scaling

Class attribute.

**MHE.scaling**

Query and set scaling of the optimization variables. The `Optimizer.scaling()` method is an indexed property, meaning getting and setting this property requires an index and calls this function. The power index (elements are seperated by commas) must contain atleast the following elements:

<table>
<thead>
<tr>
<th>order</th>
<th>index name</th>
<th>valid options</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>variable type</td>
<td>_x, _u and _z (and _p_est for MHE)</td>
</tr>
<tr>
<td>2</td>
<td>variable name</td>
<td>Names defined in <code>do_mpc.model.Model</code>.</td>
</tr>
</tbody>
</table>

This page is auto-generated. Page source is not available on Github.
Further indices are possible (but not necessary) when the referenced variable is a vector or matrix.

**Example:**

```python
# Set with:
optimizer.scaling['_x', 'phi_1'] = 2
optimizer.scaling['_x', 'phi_2'] = 2

# Query with:
optimizer.scaling['_x', 'phi_1']
```

Scaling factors $a$ affect the MHE / MPC optimization problem. The optimization variables are scaled variables:

$$
\tilde{\phi} = \frac{\phi}{a_\phi}, \ \forall \phi \in [x, u, z, p_{est}]
$$

Scaled variables are used to formulate the bounds $\tilde{\phi}_{lb} \leq \tilde{\phi}_{ub}$ and for the evaluation of the ODE. For the objective function and the nonlinear constraints the unscaled variables are used. The algebraic equations are also not scaled.

**Note:** Scaling the optimization problem is suggested when states and / or inputs take on values which differ by orders of magnitude.

This page is auto-generated. Page source is not available on Github.

### 5.11.5.3.13 t0

Class attribute.

**MHE.t0**

Current time marker of the class. Use this property to set or query the time.

Set with `int`, `float`, `numpy.ndarray` or `casadi.DM` type.

This page is auto-generated. Page source is not available on Github.

### 5.11.5.3.14 u0

Class attribute.

**MHE.u0**

Initial input and current iterate. This is the numerical structure holding the information about the current input in the class. The property can be indexed according to the model definition.

**Example:**

```python
model = do_mpc.model.Model('continuous')
model.set_variable('_u','heating', shape=(4,1))
...

mhe = do_mpc.estimator.MHE(model)
# or
mpc = do_mpc.estimator.MPC(model)
```

(continues on next page)
Useful CasADi symbolic structure methods:

- .shape
- .keys()
- .labels()

This page is auto-generated. Page source is not available on Github.

### 5.11.5.3.15 ub_opt_x

Class attribute.

MHE.ub_opt_x

Query and modify the lower bounds of all optimization variables opt_x. This is a more advanced method of setting bounds on optimization variables of the MPC/MHE problem. Users with less experience are advised to use bounds instead.

The attribute returns a nested structure that can be indexed using powerindexing. Please refer to opt_x for more details.

**Note:** The attribute automatically considers the scaling variables when setting the bounds. See scaling for more details.

**Note:** Modifications must be done after calling prepare_nlp() or setup() respectively.

This page is auto-generated. Page source is not available on Github.

### 5.11.5.3.16 x0

Class attribute.

MHE.x0

Initial state and current iterate. This is the numerical structure holding the information about the current states in the class. The property can be indexed according to the model definition.

**Example:**

```python
model = do_mpc.model.Model('continuous')
model.set_variable('_x','temperature', shape=(4,1))
...
```

```python
mhe = do_mpc.estimator.MHE(model)
# or
mpc = do_mpc.estimator.MPC(model)
```

(continues on next page)
mpc.x0['temperature', 0]  # 0th element of variable
mpc.x0['temperature']    # all elements of variable
mpc.x0['temperature', 0:2]  # 0th and 1st element

Useful CasADi symbolic structure methods:

- .shape
- .keys()
- .labels()

This page is auto-generated. Page source is not available on Github.

### 5.11.5.3.17 z0

Class attribute.

**MHE.z0**

Initial algebraic state and current iterate. This is the numerical structure holding the information about the current algebraic states in the class. The property can be indexed according to the model definition.

**Example:**

```python
model = do_mpc.model.Model('continuous')
model.set_variable('_z','temperature', shape=(4,1))
...

mhe = do_mpc.estimator.MHE(model)
# or
mpc = do_mpc.estimator.MPC(model)

# Get or set current value of variable:
mpc.z0['temperature', 0]  # 0th element of variable
mpc.z0['temperature']    # all elements of variable
mpc.z0['temperature', 0:2]  # 0th and 1st element
```

Useful CasADi symbolic structure methods:

- .shape
- .keys()
- .labels()

This page is auto-generated. Page source is not available on Github.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MHE.create_nlp</td>
<td>Create the optimization problem.</td>
</tr>
<tr>
<td>MHE.get_p_template</td>
<td>Obtain output template for set_p_fun().</td>
</tr>
<tr>
<td>MHE.get_tvp_template</td>
<td>Obtain output template for set_tvp_fun().</td>
</tr>
<tr>
<td>MHE.get_y_template</td>
<td>Obtain output template for set_y_fun().</td>
</tr>
<tr>
<td>MHE.make_step</td>
<td>Main method of the class during runtime.</td>
</tr>
<tr>
<td>MHE.prepare_nlp</td>
<td>Prepare the optimization problem.</td>
</tr>
</tbody>
</table>

Continued on next page
### Table 22 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MHE.reset_history</td>
<td>Reset the history of the optimizer.</td>
</tr>
<tr>
<td>MHE.set_default_objective</td>
<td>Configure the suggested default MHE formulation.</td>
</tr>
<tr>
<td>MHE.set_initial_guess</td>
<td>Initial guess for optimization variables.</td>
</tr>
<tr>
<td>MHE.set_nl_cons</td>
<td>Introduce new constraint to the class.</td>
</tr>
<tr>
<td>MHE.set_objective</td>
<td>Set the stage cost $l(\cdot)$ and arrival cost $m(\cdot)$ function for the MHE problem:</td>
</tr>
<tr>
<td>MHE.set_p_fun</td>
<td>Set function which returns parameters.</td>
</tr>
<tr>
<td>MHE.set_param</td>
<td>Method to set the parameters of the MHE class.</td>
</tr>
<tr>
<td>MHE.set_tvp_fun</td>
<td>Set function which returns time-varying parameters.</td>
</tr>
<tr>
<td>MHE.set_y_fun</td>
<td>Set the measurement function.</td>
</tr>
<tr>
<td>MHE.setup</td>
<td>The setup method finalizes the MHE creation.</td>
</tr>
<tr>
<td>MHE.solve</td>
<td>Solves the optimization problem.</td>
</tr>
</tbody>
</table>

#### 5.11.5.3.18 create_nlp

Class method.

```python
do_mpc.estimator.MHE.create_nlp(self)
```

Create the optimization problem. Typically, this method is called internally from `setup()`.

Users should only call this method if they intend to modify the objective with `nlp_obj`, the constraints with `nlp_cons`, `nlp_cons_lb`, and `nlp_cons_ub`.

To finish the setup process, users MUST call `create_nlp()` afterwards.

**Note:** Do NOT call `setup()` if you intend to go the manual route with `prepare_nlp()` and `create_nlp()`.

**Note:** Only AFTER calling `prepare_nlp()` the previously mentioned attributes `nlp_obj`, `nlp_cons`, `nlp_cons_lb`, and `nlp_cons_ub` become available.

This page is auto-generated. Page source is not available on Github.

#### 5.11.5.3.19 get_p_template

Class method.

```python
do_mpc.estimator.MHE.get_p_template(self)
```

Obtain output template for `set_p_fun()`. This is used to set the (not estimated) parameters. Use this structure as the return of a user defined parameter function (`p_fun`) that is called at each MHE step. Pass this function to the MHE by calling `set_p_fun()`.

**Note:** The combination of `get_p_template()` and `set_p_fun()` is identical to the `do_mpc.simulator.Simulator.Simulator` methods, if the MHE is not estimating any parameters.

**Returns** `p_template`

**Return type** `struct_symSX`

This page is auto-generated. Page source is not available on Github.
5.11.5.3.20 get_tvp_template

Class method.

do_mpc.estimator.MHE.get_tvp_template(self)

Obtain output template for set_tvp_fun().

The method returns a structured object with n_horizon+1 elements, and a set of time-varying parameters (as defined in do_mpc.model.Model) for each of these instances. The structure is initialized with all zeros. Use this object to define values of the time-varying parameters.

This structure (with numerical values) should be used as the output of the tvp_fun function which is set to the class with set_tvp_fun(). Use the combination of get_tvp_template() and set_tvp_fun().

Example:

```python
# in model definition:
alpha = model.set_variable(var_type='_tvp', var_name='alpha')
beta = model.set_variable(var_type='_tvp', var_name='beta')
...

# in optimizer configuration:
tvp_temp_1 = optimizer.get_tvp_template()
tvp_temp_1['_tvp', :] = np.array([1,1])
tvp_temp_2 = optimizer.get_tvp_template()
tvp_temp_2['_tvp', :] = np.array([0,0])

def tvp_fun(t_now):
    if t_now<10:
        return tvp_temp_1
    else:
        tvp_temp_2

optimizer.set_tvp_fun(tvp_fun)
```

**Returns** None

**Return type** None

This page is auto-generated. Page source is not available on Github.

5.11.5.3.21 get_y_template

Class method.

do_mpc.estimator.MHE.get_y_template(self)

Obtain output template for set_y_fun().

Use this structure as the return of a user defined parameter function (y_fun) that is called at each MHE step. Pass this function to the MHE by calling set_y_fun().

The structure carries a set of measurements for each time step of the horizon and can be accessed as follows:
y_template['y_meas', k, 'meas_name']
# Slicing is possible, e.g.:

where $k$ runs from 0 to $N_{\text{horizon}}$ and $\text{meas\_name}$ refers to the user-defined names in $\text{do\_mpc.\ model.\ Model}$. 

**Note:** The structure is ordered, such that $k=0$ is the “oldest measurement” and $k=N_{\text{horizon}}$ is the newest measurement.

By default, the following measurement function is choosen:

```python
y_template = self.get_y_template()
def y_fun(t_now):
    n_steps = min(self.data._y.shape[0], self.n_horizon)
    for k in range(-n_steps, 0):
        y_template['y_meas', k] = self.data._y[k]
    try:
        for k in range(self.n_horizon-n_steps):
            y_template['y_meas', k] = self.data._y[-n_steps]
    except:
        None
    return y_template
```

Which simply reads the last results from the $\text{MHE.data}$ object.

**Returns** $y_{\text{template}}$

**Return type** struct_symSX

This page is auto-generated. Page source is not available on Github.

### 5.11.5.3.22 make_step

Class method.

$\text{do\_mpc.\ estimator.\ MHE.make\_step}\ (self, y0)$

Main method of the class during runtime. This method is called at each timestep and returns the current state estimate for the current measurement $y_0$.

The method prepares the MHE by setting the current parameters, calls $\text{solve()}$ and updates the $\text{do\_mpc.\ data.\ Data}$ object.

**Warning:** Moving horizon estimation will only work reliably once a full sequence of measurements corresponding to the set horizon ist available.

**Parameters** $y0$ (numpy.ndarray) — Current measurement.

**Returns** $x0$, estimated state of the system.

**Return type** numpy.ndarray

This page is auto-generated. Page source is not available on Github.
5.11.5.3.23 prepare_nlp

Class method.

do_mpc.estimator.MHE.prepare_nlp(self)

Prepare the optimization problem. Typically, this method is called internally from setup().

Users should only call this method if they intend to modify the objective with nlp_obj, the constraints with nlp_cons, nlp_cons_lb and nlp_cons_ub.

To finish the setup process, users MUST call create_nlp() afterwards.

Note: Do NOT call setup() if you intend to go the manual route with prepare_nlp() and create_nlp().

Note: Only AFTER calling prepare_nlp() the previously mentionned attributes nlp_obj, nlp_cons, nlp_cons_lb, nlp_cons_ub become available.

This page is auto-generated. Page source is not available on Github.

5.11.5.3.24 reset_history

Class method.

do_mpc.estimator.MHE.reset_history(self)

Reset the history of the optimizer. All data from the do_mpc.data.Data instance is removed.

This page is auto-generated. Page source is not available on Github.

5.11.5.3.25 set_default_objective

Class method.

do_mpc.estimator.MHE.set_default_objective(self, P_x, P_v=None, P_p=None, P_w=None)

Configure the suggested default MHE formulation.

Use this method to pass tuning matrices for the MHE optimization problem:

\[
\begin{align*}
\min_{x_0:N+1, u_0:N, p, v_0:N} & \quad m(x_0, x_1, p, \tilde{p}) + \sum_{k=0}^{N-1} l(v_k, w_k, p, p_{tv,k}), \\
\text{s.t.} & \quad x_{k+1} = f(x_k, u_k, z_k, p, p_{tv,k}) + w_k, \\
& \quad y_k = h(x_k, u_k, z_k, p, p_{tv,k}) + v_k, \\
& \quad g(x_k, u_k, z_k, p, p_{tv,k}) \leq 0
\end{align*}
\]

where we introduce the bold letter notation, e.g. \( x_{0:N+1} = [x_0, x_1, \ldots, x_{N+1}]^T \) to represent sequences and where \( \|x\|_P^2 = x^T P x \) denotes the \( P \) weighted squared norm.

Pass the weighting matrices \( P_x, P_p \) and \( P_v \) and \( P_w \). The matrices must be of appropriate dimension and array-like.
Note: It is possible to pass parameters or time-varying parameters defined in the do_mpc.model.Model as weighting. You’ll probably choose time-varying parameters (\(_\text{tvp}\)) for \(P_v\) and \(P_w\) and parameters (\(_\text{p}\)) for \(P_x\) and \(P_p\). Use set_p_fun() and set_tvp_fun() to configure how these values are determined at each time step.

General remarks:

• In the case that no parameters are estimated, the weighting matrix \(P_p\) is not required.
• In the case that the do_mpc.model.Model is configured without process-noise (see do_mpc.model.Model.set_rhs()) the parameter \(P_w\) is not required.
• In the case that the do_mpc.model.Model is configured without measurement-noise (see do_mpc.model.Model.set_meas()) the parameter \(P_v\) is not required.

The respective terms are not present in the MHE formulation in that case.

Note: Use set_objective() as a low-level alternative for this method, if you want to use a custom objective function.

Parameters

- \(P_x\) (numpy.ndarray, casadi.SX, casadi.DM) – Tuning matrix \(P_x\) of dimension \(n \times n\)\((x \in \mathbb{R}^n)\)
- \(P_v\) (numpy.ndarray, casadi.SX, casadi.DM) – Tuning matrix \(P_v\) of dimension \(m \times m\)\((v \in \mathbb{R}^m)\)
- \(P_p\) (numpy.ndarray, casadi.SX, casadi.DM) – Tuning matrix \(P_p\) of dimension \(l \times l\)\((\text{est} \in \mathbb{R}^l)\)
- \(P_w\) (numpy.ndarray, casadi.SX, casadi.DM) – Tuning matrix \(P_w\) of dimension \(k \times k\)\((w \in \mathbb{R}^k)\)

This page is auto-generated. Page source is not available on Github.

5.11.5.3.26 set_initial_guess

Class method.

do_mpc.estimator.MHE.set_initial_guess(self)
Initial guess for optimization variables. Uses the current class attributes \(x_0, z_0\) and \(u_0, p_{\text{est}0}\) to create an initial guess for the MHE. The initial guess is simply the initial values for all \(k = 0, \ldots, N\) instances of \(x_k, u_k\) and \(z_k, p_{\text{est},k}\).

Warning: If no initial values for \(x_0, z_0\) and \(u_0\) were supplied during setup, these default to zero.

Note: The initial guess is fully customizable by directly setting values on the class attribute: \(\text{opt}_x\)num.

This page is auto-generated. Page source is not available on Github.
5.11.5.3.27 set_nl_cons

Class method.

do_mpc.estimator.MHE.set_nl_cons(self, expr_name, expr, ub=inf, soft_constraint=False, penalty_term_cons=1, maximum_violation=inf)

Introduce new constraint to the class. Further constraints are optional. Expressions must be formulated with respect to \(_x, _u, _z, _tvp, _p\). They are implemented as:

\[
m(x, u, z, pv, p) \leq m_{ub}
\]

Setting the flag soft_constraint=True will introduce slack variables \(\epsilon\), such that:

\[
m(x, u, z, pv, p) - \epsilon \leq m_{ub},
\]

\[0 \leq \epsilon \leq \epsilon_{max},\]

Slack variables are added to the cost function and multiplied with the supplied penalty term. This formulation makes constraints soft, meaning that a certain violation is tolerated and does not lead to infeasibility. Typically, high values for the penalty are suggested to avoid significant violation of the constraints.

**Parameters**

- **expr_name** (string) – Arbitrary name for the given expression. Names are used for key word indexing.
- **expr** (CasADi SX or MX) – CasADi SX or MX function depending on \(_x, _u, _z, _tvp, _p\).

**Raises**

- **assertion** – expr_name must be str
- **assertion** – expr must be a casadi SX or MX type

**Returns**

Returns the newly created expression. Expression can be used e.g. for the RHS.

**Return type** casadi.SX or casadi.MX

This page is auto-generated. Page source is not available on Github.

5.11.5.3.28 set_objective

Class method.

do_mpc.estimator.MHE.set_objective(self, stage_cost, arrival_cost)

Set the stage cost \(l(\cdot)\) and arrival cost \(m(\cdot)\) function for the MHE problem:

\[
\min_{x_0:N+1, u_0:N, p, w_0:N, v_0:N} m(x_0, \tilde{x}_0, p, \tilde{p}) + \sum_{k=0}^{N-1} l(v_k, w_k, p, pv_k),
\]

\[
\text{s.t. } \begin{align*}
    x_{k+1} &= f(x_k, u_k, z_k, p, pv_k) + w_k, \\
    y_k &= h(x_k, u_k, z_k, p, pv_k) + v_k, \\
    g(x_k, u_k, z_k, p_k, pv_k) &\leq 0
\end{align*} \quad k = 0, \ldots, N
\]

Use the class attributes:

- **mhe._w** as \(w_k\)
- **mhe._v** as \(v_k\)
To formulate the objective function and pass the stage cost and arrival cost independently.

Note: The retrieved attributes are symbolic structures, which can be queried with the given variable names, e.g.:

```python
x1 = mhe._x['state_1']
```

For a vector of all states, use the `.cat` method as shown in the example below.

Example:

```python
# Get variables:
v = mhe._v.cat
stage_cost = v.T@np.diag(np.array([1,1,1,20,20]))@v
x_0 = mhe._x
x_prev = mhe._x_prev
p_0 = mhe._p_est
p_prev = mhe._p_est_prev
dx = x_0.cat - x_prev.cat
dp = p_0.cat - p_prev.cat
arrival_cost = 1e-4*dx.T@dx + 1e-4*dp.T@dp
mhe.set_objective(stage_cost, arrival_cost)
```

Note: Use `set_default_objective()` as a high-level wrapper for this method, if you want to use the default MHE objective function.

Parameters

- `stage_cost` *(CasADi expression)* – Stage cost that is added to the MHE objective at each age.

- `arrival_cost` *(CasADi expression)* – Arrival cost that is added to the MHE objective at the initial state.

Returns None

Return type None

This page is auto-generated. Page source is not available on Github.
do_mpc.estimator.MHE.set_p_fun(self, p_fun)

Set function which returns parameters. The p_fun is called at each MHE time step and returns the (fixed) parameters. The function must return a numerical CasADi structure, which can be retrieved with get_p_template().

Parameters

- **p_fun** (function) – Parameter function.

This page is auto-generated. Page source is not available on Github.

### 5.11.5.3.30 set_param

Class method.

do_mpc.estimator.MHE.set_param(self, **kwargs)

Method to set the parameters of the MHE class. Parameters must be passed as pairs of valid keywords and respective argument. For example:

```python
mhe.set_param(n_horizon = 20)
```

It is also possible and convenient to pass a dictionary with multiple parameters simultaneously as shown in the following example:

```python
setup_mhe = {'n_horizon': 20,
             't_step': 0.5,
            }
mhe.set_param(**setup_mhe)
```

This makes use of the python “unpack” operator. See more details here.

**Note:** The only required parameters are **n_horizon** and **t_step**. All other parameters are optional.

**Note:** set_param() can be called multiple times. Previously passed arguments are overwritten by successive calls.

The following parameters are available:

**Parameters**

- **n_horizon** (*int*) – Prediction horizon of the optimal control problem. Parameter must be set by user.
- **t_step** (*float*) – Timestep of the mhe.
- **meas_from_data** (*bool*) – Default option to retrieve past measurements for the MHE optimization problem. The set_y_fun() is called during setup.
- **state_discretization** (*str*) – Choose the state discretization for continuous models. Currently only 'collocation' is available. Defaults to 'collocation'. Has no effect if model is created in discrete type.
- **collocation_type** (*str*) – Choose the collocation type for continuous models with collocation as state discretization. Currently only 'radau' is available. Defaults to 'radau'.
• `collocation_deg` (*int*) – Choose the collocation degree for continuous models with collocation as state discretization. Defaults to 2.

• `collocation_ni` (*int*) – For orthogonal collocation, choose the number of finite elements for the states within a time-step (and during constant control input). Defaults to 1. Can be used to avoid high-order polynomials.

• `nl_cons_check_colloc_points` (*bool*) – For orthogonal collocation choose whether the bounds set with `set_nl_cons()` are evaluated once per finite element or for each collocation point. Defaults to `False` (once per collocation point).

• `cons_check_colloc_points` (*bool*) – For orthogonal collocation choose whether the linear bounds set with `bounds` are evaluated once per finite element or for each collocation point. Defaults to `True` (for all collocation points).

• `nl_cons_single_slack` (*bool*) – If `True`, soft-constraints set with `set_nl_cons()` introduce only a single slack variable for the entire horizon. Defaults to `False`.

• `store_full_solution` (*bool*) – Choose whether to store the full solution of the optimization problem. This is required for animating the predictions in post processing. However, it drastically increases the required storage. Defaults to `False`.

• `store_lagr_multiplier` (*bool*) – Choose whether to store the lagrange multipliers of the optimization problem. Increases the required storage. Defaults to `True`.

• `store_solver_stats` (*dict*) – Choose which solver statistics to store. Must be a list of valid statistics. Defaults to `['success', 't_wall_S']`.

• `nlpsol_opts` – Dictionary with options for the CasADi solver call `nlpsol` with plugin `ipopt`. All options are listed here.

**Note:** We highly suggest to change the linear solver for IPOPT from `mumps` to `MA27`. In many cases this will drastically boost the speed of `do-mpc`. Change the linear solver with:

```python
optimizer.set_param(nlpsol_opts = {'ipopt.linear_solver': 'MA27'})
```

**Note:** To suppress the output of IPOPT, please use:

```python
suppress_ipopt = {'ipopt.print_level':0, 'ipopt.sb': 'yes', 'print_time':0}
optimizer.set_param(nlpsol_opts = suppress_ipopt)
```

This page is auto-generated. Page source is not available on Github.

### 5.11.5.3.31 set_tvp_fun

Class method.

```python
do_mpc.estimator.MHE.set_tvp_fun(self, tvp_fun)
```

Set function which returns time-varying parameters.

The `tvp_fun` is called at each optimization step to get the current prediction of the time-varying parameters. The supplied function must be callable with the current time as the only input. Furthermore, the function must return a CasADi structured object which is based on the horizon and on the model definition. The structure can be obtained with `get_tvp_template()`.
Example:

```python
# in model definition:
alpha = model.set_variable(var_type='_tvp', var_name='alpha')
beta = model.set_variable(var_type='_tvp', var_name='beta')
...
# in optimizer configuration:
tvp_temp_1 = optimizer.get_tvp_template()
tvp_temp_1['_tvp', :] = np.array([1,1])
tvp_temp_2 = optimizer.get_tvp_template()
tvp_temp_2['_tvp', :] = np.array([0,0])

def tvp_fun(t_now):
    if t_now<10:
        return tvp_temp_1
    else:
        tvp_temp_2
optimizer.set_tvp_fun(tvp_fun)
```

**Note:** The method `set_tvp_fun()` must be called prior to setup IF time-varying parameters are defined in the model. It is not required to call the method if no time-varying parameters are defined.

**Parameters** tvp_fun *(function)* – Function that returns the predicted tvp values at each timestep. Must have single input (float) and return a `structure3.DMStruct` (obtained with `get_tvp_template()`).

This page is auto-generated. Page source is not available on Github.

### 5.11.5.3.32 set_y_fun

Class method.

`do_mpc.estimator.MHE.set_y_fun(self, y_fun)`

Set the measurement function. The function must return a CasADi structure which can be obtained from `get_y_template()`. See the respective doc string for details.

**Parameters** y_fun *(function)* – measurement function.

This page is auto-generated. Page source is not available on Github.

### 5.11.5.3.33 setup

Class method.

`do_mpc.estimator.MHE.setup(self)`

The setup method finalizes the MHE creation. The optimization problem is created based on the configuration of the module.

**Note:** After this call, the `solve()` and `make_step()` method is applicable.
5.11.5.34 solve

Class method.

do_mpc.estimator.MHE.solve(self)

Solves the optimization problem.

The current problem is defined by the parameters in the opt_p_num CasADi structured Data.

Typically, opt_p_num is prepared for the current iteration in the make_step() method. It is, however, valid and possible to directly set parameters in opt_p_num before calling solve().

The method updates the opt_p_num and opt_x_num attributes of the class. By resetting opt_x_num to the current solution, the method implicitly enables warmstarting the optimizer for the next iteration, since this vector is always used as the initial guess.

Warning: The method is part of the public API but it is generally not advised to use it. Instead we recommend to call make_step() at each iterations, which acts as a wrapper for solve().

Raises assertion – Optimizer was not setup yet.

Returns None

Return type None

5.11.5.4 StateFeedback

class do_mpc.estimator.StateFeedback(model)

Simple state-feedback “estimator”. The main method StateFeedback.make_step() simply returns the input. Why do you even bother to use this class?

Attributes

<table>
<thead>
<tr>
<th>StateFeedback.t0</th>
<th>Current time marker of the class.</th>
</tr>
</thead>
<tbody>
<tr>
<td>StateFeedback.u0</td>
<td>Initial input and current iterate.</td>
</tr>
<tr>
<td>StateFeedback.x0</td>
<td>Initial state and current iterate.</td>
</tr>
<tr>
<td>StateFeedback.z0</td>
<td>Initial algebraic state and current iterate.</td>
</tr>
</tbody>
</table>

5.11.5.4.1 t0

Class attribute.

StateFeedback.t0

Current time marker of the class. Use this property to set or query the time.

Set with int, float, numpy.ndarray or casadi.DM type.
5.11.5.4.2 u0

Class attribute.

StateFeedback.u0
Initial input and current iterate. This is the numerical structure holding the information about the current input in the class. The property can be indexed according to the model definition.

Example:

```python
define = do_mpc.model.Model('continuous')
define.set_variable('_u','heating', shape=(4,1))
...
define = do_mpc.estimator.MHE(define)
# or
define = do_mpc.estimator.MPC(define)

# Get or set current value of variable:
define.u0['heating', 0] # 0th element of variable
define.u0['heating']   # all elements of variable
define.u0['heating', 0:2] # 0th and 1st element
```

Useful CasADi symbolic structure methods:

- .shape
- .keys()
- .labels()

5.11.5.4.3 x0

Class attribute.

StateFeedback.x0
Initial state and current iterate. This is the numerical structure holding the information about the current states in the class. The property can be indexed according to the model definition.

Example:

```python
define = do_mpc.model.Model('continuous')
define.set_variable('_x','temperature', shape=(4,1))
...
define = do_mpc.estimator.MHE(define)
# or
define = do_mpc.estimator.MPC(define)

# Get or set current value of variable:
define.x0['temperature', 0] # 0th element of variable
define.x0['temperature']   # all elements of variable
define.x0['temperature', 0:2] # 0th and 1st element
```
Useful CasADi symbolic structure methods:
  • .shape
  • .keys()
  • .labels()

This page is auto-generated. Page source is not available on Github.

5.11.5.4.4 z0

Class attribute.

**StateFeedback. z0**

Initial algebraic state and current iterate. This is the numerical structure holding the information about the current algebraic states in the class. The property can be indexed according to the model definition.

**Example:**

```python
model = do_mpc.model.Model('continuous')
model.set_variable('_z','temperature', shape=(4,1))
...

mhe = do_mpc.estimator.MHE(model)
# or
mpc = do_mpc.estimator.MPC(model)

# Get or set current value of variable:
mpc.z0['temperature', 0]  # 0th element of variable
mpc.z0['temperature']     # all elements of variable
mpc.z0['temperature', 0:2]  # 0th and 1st element
```

Useful CasADi symbolic structure methods:
  • .shape
  • .keys()
  • .labels()

This page is auto-generated. Page source is not available on Github.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StateFeedback.make_step</td>
<td>Return the measurement ( y_0 ).</td>
</tr>
<tr>
<td>StateFeedback.reset_history</td>
<td>Reset the history of the estimator</td>
</tr>
</tbody>
</table>

5.11.5.4.5 make_step

Class method.

do_mpc.estimator.StateFeedback.make_step \( (self, y_0) \)

  Return the measurement \( y_0 \).

This page is auto-generated. Page source is not available on Github.
5.11.5.6 reset_history

Class method.

do_mpc.estimator.StateFeedback.reset_history(self)

Reset the history of the estimator

This page is auto-generated. Page source is not available on Github.

5.11.6 data

Classes

<table>
<thead>
<tr>
<th>Data</th>
<th>do-mpc data container.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPCData</td>
<td>do-mpc data container for the do_mpc.controller.MPC instance.</td>
</tr>
</tbody>
</table>

5.11.6.1 Data

class do_mpc.data.Data(model)

do-mpc data container. An instance of this class is created for the active do-mpc classes, e.g. do_mpc.simulator.Simulator, do_mpc.estimator.MHE.

The class is initialized with an instance of the do_mpc.model.Model which contains all information about variables (e.g. states, inputs etc.).

The Data class has a public API but is mostly used by other do-mpc classes, e.g. updated in the .make_step calls.

__getitem__(ind)

Query data fields. This method can be used to obtain the stored results in the Data instance.

The full list of available fields can be inspected with:

```
print(data.data_fields)
```

The dict also denotes the dimension of each field.

The method allows for power indexing the results for the fields _x, _u, _z, _tvp, _p, _aux, _y where further indices refer to the configured variables in the do_mpc.model.Model instance.

Example:

```
# Assume the following model was used (excerpt):
model = do_mpc.model.Model('continuous')
model.set_variable('_x', 'Temperature', shape=(5,1)) # Vector
model.set_variable('_p', 'disturbance', shape=(3,3)) # Matrix
model.set_variable('_u', 'heating') # scalar
...
# the model was used (among others) for the MPC controller
```

(continues on next page)
mpc = do_mpc.controller.MPC(model)
...

# Query the mpc.data instance:
mpc.data[['_x']]  # Return all states
mpc.data[['_x', 'Temperature']]  # Return the 5 temp states
mpc.data[['_x', 'Temperature', :2]]  # Return the first 2 temp. states
mpc.data[['_p', 'disturbance', 0, 2]]  # Matrix allows for further indices

# Other fields can also be queried, e.g.:
mpc.data['_time']  # current time
mpc.data['t_wall_total']  # optimizer runtime
# These do not allow further indices.

**Returns**  Returns the queried data field (for all time instances)

**Return type**  numpy.ndarray

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Data.export</code></td>
<td>The export method returns a dictionary of the stored data.</td>
</tr>
<tr>
<td><code>Data.init_storage</code></td>
<td>Create new (empty) arrays for all variables.</td>
</tr>
<tr>
<td><code>Data.set_meta</code></td>
<td>Set meta data for the current instance of the data object.</td>
</tr>
<tr>
<td><code>Data.update</code></td>
<td>Update value(s) of the data structure with key word arguments.</td>
</tr>
</tbody>
</table>

### 5.11.6.1.1 export

Class method.

```python
do_mpc.data.Data.export(self)
```

The export method returns a dictionary of the stored data.

**Returns**  Dictionary of the currently stored data.

**Return type**  dict

This page is auto-generated. Page source is not available on Github.

### 5.11.6.1.2 init_storage

Class method.

```python
do_mpc.data.Data.init_storage(self)
```

Create new (empty) arrays for all variables. The variables of interest are listed in the `data_fields` dictionary, with their respective dimension. This dictionary may be updated. The `do_mpc.controller.MPC` class adds for example optimizer information.

This page is auto-generated. Page source is not available on Github.
5.11.6.1.3 set_meta

Class method.

```python
do_mpc.data.Data.set_meta(self, **kwargs)
```

Set meta data for the current instance of the data object.

This page is auto-generated. Page source is not available on Github.

5.11.6.1.4 update

Class method.

```python
do_mpc.data.Data.update(self, **kwargs)
```

Update value(s) of the data structure with key word arguments. These key word arguments must exist in the data fields of the data objective. See self.data_fields for a complete list of data fields.

Example:

```python
_x = np.ones((1, 3))
_u = np.ones((1, 2))
data.update('_x': _x, '_u': _u)
```

or:

```python
data.update('_x': _x)
data.update('_u': _u)
```

Alternatively:

```python
data_dict = {
    '_x': np.ones((1, 3)),
    '_u': np.ones((1, 2))
}
data.update(**data_dict)
```

**Parameters**

- **kwargs** *(casadi.DM or numpy.ndarray)* – Arbitrary number of key word arguments for data fields that should be updated.

**Raises**

- **assertion** – Keyword must be in existing data_fields.

**Returns**

None

This page is auto-generated. Page source is not available on Github.

5.11.6.2 MPCData

```python
class do_mpc.data.MPCData(model)
```

do_mpc data container for the do_mpc.controller.MPC instance. This method inherits from Data and extends it to query the MPC predictions.

```python
__getitem__(ind)
```

Query data fields. This method can be used to obtain the stored results in the Data instance.

The full list of available fields can be inspected with:
print(data.data_fields)

The dict also denotes the dimension of each field.

The method allows for power indexing the results for the fields _x, _u, _z, _tvp, _p, _aux, _y where further indices refer to the configured variables in the do_mpc.model.Model instance.

Example:

```python
# Assume the following model was used (excerpt):
model = do_mpc.model.Model('continuous')
model.set_variable('_x', 'Temperature', shape=(5,1))  # Vector
model.set_variable('_p', 'disturbance', shape=(3,3))  # Matrix
model.set_variable('_u', 'heating')  # scalar
...

# the model was used (among others) for the MPC controller
mpc = do_mpc.controller.MPC(model)
...

# Query the mpc.data instance:
mpc.data['_x']  # Return all states
mpc.data['_x', 'Temperature']  # Return the 5 temp states
mpc.data['_x', 'Temperature', :2]  # Return the first 2 temp. states
mpc.data['_p', 'disturbance', 0, 2]  # Matrix allows for further indices

# Other fields can also be queried, e.g.:
mpc.data['_time']  # current time
mpc.data['t_wall_total']  # optimizer runtime

# These do not allow further indices.
```

**Returns**  Returns the queried data field (for all time instances)

**Return type**  numpy.ndarray

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPCData.export</td>
<td>The export method returns a dictionary of the stored data.</td>
</tr>
<tr>
<td>MPCData.init_storage</td>
<td>Create new (empty) arrays for all variables.</td>
</tr>
<tr>
<td>MPCData.prediction</td>
<td>Query the MPC trajectories.</td>
</tr>
<tr>
<td>MPCData.set_meta</td>
<td>Set meta data for the current instance of the data object.</td>
</tr>
<tr>
<td>MPCData.update</td>
<td>Update value(s) of the data structure with key word arguments.</td>
</tr>
</tbody>
</table>

#### 5.11.6.2.1 export

Class method.

do_mpc.data.MPCData.export(self)

The export method returns a dictionary of the stored data.
**Returns** Dictionary of the currently stored data.

**Return type** dict

This page is auto-generated. Page source is not available on Github.

### 5.11.6.2.2 init_storage

Class method.

```python
do_mpc.data.MPCData.init_storage(self)
```

Create new (empty) arrays for all variables. The variables of interest are listed in the `data_fields` dictionary, with their respective dimension. This dictionary may be updated. The `do_mpc.controller.MPC` class adds for example optimizer information.

This page is auto-generated. Page source is not available on Github.

### 5.11.6.2.3 prediction

Class method.

```python
do_mpc.data.MPCData.prediction(self, ind, t_ind=-1)
```

Query the MPC trajectories. Use this method to obtain specific MPC trajectories from the data object.

**Warning:** This method requires that the optimal solution is stored in the `do_mpc.data.MPCData` instance. Storing the optimal solution must be activated with `do_mpc.controller.MPC.set_param()`.

Querying predicted trajectories requires the use of power indices, which is passed as tuple e.g.:

```python
data.prediction((var_type, var_name, i), t_ind)
```

where

- `var_type` refers to \_x, \_u, \_z, \_tvp, \_p, \_aux
- `var_name` refers to the user-defined names in the `do_mpc.model.Model`
- Use `i` to index vector valued variables.

The method returns a multidimensional numpy.ndarray. The dimensions refer to:

```python
arr = data.prediction(('\_x', '\x\_1'))
arr.shape
```

```python
>>> (n_size, n_horizon, n_scenario)
```

with:

- `n_size` denoting the number of elements in \_x\_1, where `n_size = 1` is a scalar variable.
- `n_horizon` is the MPC horizon defined with `do_mpc.controller.MPC.set_param()`.
- `n_scenario` refers to the number of uncertain scenarios (for robust MPC).

Additional to the power index tuple, a time index (`t_ind`) can be passed to access the prediction for a certain time.

**Parameters** `ind` *(tuple)* – Power index to query the prediction of a specific variable.
Returns Predicted trajectories for the queries variable.

Return type numpy.ndarray

This page is auto-generated. Page source is not available on Github.

5.11.6.2.4 set_meta

Class method.

do_mpc.data.MPCData.set_meta(self, **kwargs)

Set meta data for the current instance of the data object.

This page is auto-generated. Page source is not available on Github.

5.11.6.2.5 update

Class method.

do_mpc.data.MPCData.update(self, **kwargs)

Update value(s) of the data structure with key word arguments. These key word arguments must exist in the data fields of the data object. See self.data_fields for a complete list of data fields.

Example:

```python
_x = np.ones((1, 3))
_u = np.ones((1, 2))
data.update('_x': _x, '_u': _u)
```

or:

```python
data.update('_x': _x)
data.update('_u': _u)
```

Alternatively:

data_dict = {
    '_x': np.ones((1, 3)),
    '_u': np.ones((1, 2))
}

data.update(**data_dict)
```

Parameters kwargs (casadi.DM or numpy.ndarray) – Arbitrary number of key word arguments for data fields that should be updated.

Raises assertion – Keyword must be in existing data_fields.

Returns None

This page is auto-generated. Page source is not available on Github.

This page is auto-generated. Page source is not available on Github.

Functions

<table>
<thead>
<tr>
<th>load_results</th>
<th>Simple wrapper to open and unpickle a file.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Continued on next page</td>
</tr>
</tbody>
</table>
save_results
Exports the data objects from the do-mpc modules in save_list as a pickled file.

5.11.6.3 load_results

**do_mpc.data.load_results(file_name)**

Simple wrapper to open and unpickle a file. If used for do-mpc results, this will return a dictionary with the stored do-mpc modules:

- do_mpc.controller.MPC
- do_mpc.simulator.Simulator
- do_mpc.estimator.Estimator

**Parameters**

- **file_name (str)** – File name (including path) for the file to be opened and unpickled.

5.11.6.4 save_results

**do_mpc.data.save_results(save_list, result_name='results', result_path='./results/', overwrite=False)**

Exports the data objects from the do-mpc modules in save_list as a pickled file. Supply any, all or a selection of (as a list):

- do_mpc.controller.MPC
- do_mpc.simulator.Simulator
- do_mpc.estimator.Estimator

These objects can be used in post-processing to create graphics with the do_mpc.graphics_backend.

**Parameters**

- **save_list (list)** – List of the objects to be stored.
- **result_name (string, optional)** – Name of the result file, defaults to ‘result’.
- **result_path (string, optional)** – Result path, defaults to ‘./results’.
- **overwrite (bool, optional)** – Option to overwrite existing results, defaults to False. Index will be appended if file already exists.

**Raises**

- **assertion** – save_list must be a list.
- **assertion** – result_name must be a string.
- **assertion** – result_path must be a string.
- **assertion** – overwrite must be boolean.
- **Exception** – save_list contains object which is neither do_mpc simulator, optimizier nor estimator.

**Returns**

None

**Return type** None

This page is auto-generated. Page source is not available on Github.
5.11.7 graphics

Classes

| Graphics | Graphics module to present the results of do-mpc. |

5.11.7.1 Graphics

class do_mpc.graphics.Graphics(data)

Graphics module to present the results of do-mpc. The module is independent of all other modules and can be used optionally. The module can also be used with pickled result files in post-processing for flexible and custom graphics.

The graphics module is based on Matplotlib and allows for fully customizable, publication ready graphics and animations.

The Graphics module is initialized with a do_mpc.data.Data or do_mpc.data.MPCData module and will showcase this data.

User defined graphics are configured prior to plotting results, e.g.:

```python
mpc = do_mpc.controller.MPC(model)
...
# Initialize graphic:
graphics = do_mpc.graphics.Graphics(mpc.data)

# Create figure with arbitrary Matplotlib method
fig, ax = plt.subplots(5, sharex=True)
# Configure plot (pass the previously obtained ax objects):
graphics.add_line(var_type='_x', var_name='C_a', axis=ax[0])
graphics.add_line(var_type='_x', var_name='C_b', axis=ax[0])
graphics.add_line(var_type='_x', var_name='T_R', axis=ax[1])
graphics.add_line(var_type='_x', var_name='T_K', axis=ax[1])
graphics.add_line(var_type='_aux', var_name='T_dif', axis=ax[2])
graphics.add_line(var_type='_u', var_name='Q_dot', axis=ax[3])
graphics.add_line(var_type='_u', var_name='F', axis=ax[4])
# Optional configuration of the plot(s) with matplotlib:
ax[0].set_ylabel('c [mol/l]')
ax[1].set_ylabel('Temperature [K]')
ax[2].set_ylabel('\Delta T [K]')
ax[3].set_ylabel('Q_heat [kW]')
ax[4].set_ylabel('Flow [l/h]')
fig.align_ylabels()
```

After initializing the Graphics module, the Graphics.add_line() method is used to define which results are to be plotted on which existing axes object. The method created (empty) line objects for each plotted variable. The graphic is updated with the most recent data with Graphics.plot_results(). Furthermore, the module contains the Graphics.plot_predictions() method which is applicable only for do_mpc.data.MPCData, and can be used to show the predicted trajectories.

**Note:** A high-level API for obtaining a configured Graphics module is the default_plot() function. Use this function and the obtained Graphics module in the development process.
Animations can be setup with the following loop:

```python
for k in range(50):
    u0 = mpc.make_step(x0)
    y_next = simulator.make_step(u0)
    x0 = estimator.make_step(y_next)

    graphics.plot_results()
    graphics.plot_predictions()
    graphics.reset_axes()
plt.show()
plt.pause(0.01)
```

**Parameters** `data` (*do_mpc.data.Data or do_mpc.data.MPCData*) – Data object from the `do-mpc` modules (simulator, estimator, controller)

**Attributes**

<table>
<thead>
<tr>
<th>Graphics.pred_lines</th>
<th>Structure that holds the prediction line objects.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphics.result_lines</td>
<td>Structure that holds the result line objects.</td>
</tr>
</tbody>
</table>

### 5.11.7.1.1 pred_lines

Class attribute.

**Graphics.pred_lines**

Structure that holds the prediction line objects. Query this structure with power indices. The power indices must have the following order:

`pred_lines[var_type, var_name, i, k]`

where

- `var_type` refers to `_x`, `_u`, `_z`, `_tvp`, `_p`, `_aux`
- `var_name` refers to the user-defined names in the `do_mpc.model.Model`
- Use `i` to index vector valued variables (choose 0 for scalars).
- Use `k` to select the `k`th scenario (for robust MPC). Note the `k=0` is the nominal case.

Note that (e.g.) `pred_lines['_x']` will return all lines for all states and `pred_lines.full` can be used to retrieve all line objects.

This property can be used to query and configure specific lines in the current graphic.

**Example:**

```python
# Update properties for all lines:
for line_i in graphics.pred_lines.full:
    line_i.set_linewidth(2)
    line_i.set_alpha(0.5)
```

An extensive list of all line properties can be found [here](#).

**Parameters** `powerind` (*tuple*) – Tuple of indices (power indices) to obtain the desired line objects.
**Returns**  List of line objects.

**Return type**  list

This page is auto-generated. Page source is not available on Github.

### 5.11.7.1.2 result_lines

Class attribute.

**Graphics.result_lines**

Structure that holds the result line objects. Query this structure with power indices. The power indices must have the following order:

```
result_lines[var_type, var_name, i]
```

where

- **var_type** refers to \_x, \_u, \_z, \_tvp, \_p, \_aux
- **var_name** refers to the user-defined names in the do_mpc.model.Model
- Index \(i\) is applicable if the selecte variable is vector valued.

Note that (e.g.) `result_lines['\_x']` will return all lines for all states and `result_lines.full` can be used to retrieve all line objects.

This property can be used to query and configure specific lines in the current graphic.

**Example:**

```python
# Update properties for all lines:
for line_i in graphics.result_lines.full:
    line_i.set_linewidth(2)
    line_i.set_alpha(0.5)
```

An extensive list of all line properties can be found here.

**Parameters**  `powerind` *(tuple)* – Tuple of indices (power indices) to obtain the desired line objects

**Returns**  List of line objects.

**Return type**  list

This page is auto-generated. Page source is not available on Github.

### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Graphics.add_line</strong></td>
<td><code>add_line</code> is called during setting up the Graphics class.</td>
</tr>
<tr>
<td><strong>Graphics.clear</strong></td>
<td>Clears all data from lines.</td>
</tr>
<tr>
<td><strong>Graphics.plot_predictions</strong></td>
<td>Plots the predicted trajectories for the plot configuration.</td>
</tr>
<tr>
<td><strong>Graphics.plot_results</strong></td>
<td>Plots the results stored in the data object.</td>
</tr>
<tr>
<td><strong>Graphics.reset_axes</strong></td>
<td>Relimits and scales all axes.</td>
</tr>
</tbody>
</table>
| **Graphics.reset_prop_cycle**  | Resets the property cycle for all axes which were passed with `Graphics.add_line()`.


5.11.7.1.3 add_line

Class method.

do_mpc.graphics.Graphics.add_line(self, var_type, var_name, axis, **pltkwargs)

add_line is called during setting up the Graphics class. This is typically the last step of configuring do-mpc. Each call of Graphics.add_line() adds a line to the passed axis according to the variable type (_x, _u, _z, _tvp, _p, _aux) and its name (as defined in the do_mpc.model.Model). Furthermore, all valid matplotlib.pyplot arguments can be passed as optional keyword arguments, e.g.: linewidth, color, alpha.

**Note:** Lines can also be configured after adding them with this method. Use the result_lines() and pred_lines() attributes for this purpose.

**Parameters**

- **var_type** (string) – Variable type to be plotted. Valid arguments are _x, _u, _z, _tvp, _p, _aux.
- **var_name** (string) – Variable name. Must reference the names defined in the model for the given variable type.
- **axis** (matplotlib.axes.Axes object) – Axis object on which to plot the line(s).
- **pltkwargs** (optional) – Valid matplotlib.pyplot keyword arguments (e.g.: linewidth, color, alpha)

**Raises**

- **assertion** – var_type argument must be a string
- **assertion** – var_name argument must be a string
- **assertion** – var_type argument must reference to the valid var_types of do-mpc models.
- **assertion** – axis argument must be matplotlib axes object.

This page is auto-generated. Page source is not available on Github.

5.11.7.1.4 clear

Class method.

do_mpc.graphics.Graphics.clear(self, lines=None)

Clears all data from lines.

This page is auto-generated. Page source is not available on Github.

5.11.7.1.5 plot_predictions

Class method.

do_mpc.graphics.Graphics.plot_predictions(self, t_ind=-1)

Plots the predicted trajectories for the plot configuration. The predicted trajectories are part of the optimal solution at each timestep and are optionally stored in the do_mpc.data.MPCData object.
Warning: This method requires that the optimal solution is stored in the `do_mpc.data.MPCData` instance. Storing the optimal solution must be activated with `do_mpc.controller.MPC.set_param()`.

The `plot_predictions` method can only be called with data from the `do_mpc.controller.MPC` object and raises an error if called with data from other objects. Use the `t_ind` parameter to plot the prediction for the given time instance. This can be used in post-processing for animations.

**Parameters** `t_ind` (*int*) – Plot predictions at this time index.

**Raises**

- **assertion** – Can only call `plot_predictions` with data object from do-mpc optimizer
- **Exception** – Cannot plot predictions if full solution is not stored or supplied when calling the method
- **assertion** – `t_ind` argument must be a int
- **assertion** – `t_ind` argument must not exceed the length of the results

**Returns** None

This page is auto-generated. Page source is not available on Github.

### 5.11.7.1.6 plot_results

Class method.

```python
do_mpc.graphics.Graphics.plot_results(self, t_ind=-1)
```

Plots the results stored in the data object. Use the `t_ind` parameter to plot only until the given time index. This can be used in post-processing for animations.

**Parameters** `t_ind` (*int*) – Plot results up until this time index.

**Raises**

- **assertion** – `t_ind` argument must be a int
- **assertion** – `t_ind` argument must not exceed the length of the results

**Returns** None.

This page is auto-generated. Page source is not available on Github.

### 5.11.7.1.7 reset_axes

Class method.

```python
do_mpc.graphics.Graphics.reset_axes(self)
```

Relimits and scales all axes. This method calls

```python
ax.relim()
ax.autoscale()
```

on all axes instances in the class.

This page is auto-generated. Page source is not available on Github.
5.11.7.1.8 reset_prop_cycle

Class method.

```
do_mpc.graphics.Graphics.reset_prop_cycle(self)
```

Resets the property cycle for all axes which were passed with Graphics.add_line(). The matplotlib color cycler is restarted.

This page is auto-generated. Page source is not available on Github.

5.11.7.2 animate

```
do_mpc.graphics.animate(graphics, fig, n_steps=None, export_path='.', export_name='animation', overwrite=False, format='gif', fps=5, writer=None)
```

Animation helper function.

Call this function with a configured Graphics instance and the respective figure. This function will export an animation with the results from the do_mpc.data.Data object.

Either specify format and fps or supply a configured writer (e.g. ImageMagickWriter for gifs).

Parameters

- **graphics** (Graphics) – Configured Graphics instance.
- **fig** (Matplotlib Figure) – Matplotlib Figure.
- **n_steps** (int) – (Optional) number of time steps for the animation.
- **export_path** (str) – (Optional) Path where to export the animation. Directory will be created if it doesn’t exist.
- **export_name** (str) – (Optional) Name of the resulting animation (gif/mp4) file.
- **overwrite** (bool) – (Optional) Check if export_name already exists in the supplied directory and overwrite or alter export_name.
- **format** (str) – (Optional) Choose between gif or mp4.
- **fps** (int) – (Optional) Frames per second for the resulting animation.
- **writer** (writer class) – (Optional) If supplied, the fps and format argument are discarded. Use this to configure your own writer.

Returns

None

5.11.7.3 default_plot

```
do_mpc.graphics.default_plot(data, states_list=None, dae_states_list=None, inputs_list=None, aux_list=None, **kwargs)
```

Pass a do_mpc.data.Data object and create a default do-mpc plot. By default all states, inputs and auxiliary
expressions are plotted on individual axes. Pass lists of states, inputs and aux names (string) to plot only a subset of these trajectories.

Returns a figure, axis and configured Graphics object.

Parameters

- **model** (**do_mpc.data.Data** or **do_mpc.data.MPCData**) – do-mpc data instance.
- **states_list** (**list**) – List of strings containing a subset of state names defined in py:class:**do_mpc.model.Model**. These states are plotted.
- **dae_states_list** (**list**) – List of strings containing a subset of dae states (_z) names defined in py:class:**do_mpc.model.Model**. These states are plotted.
- **inputs_list** (**list**) – List of strings containing a subset of input names defined in py:class:**do_mpc.model.Model**. These inputs are plotted.
- **aux_list** (**list**) – List of strings containing a subset of auxiliary expression names defined in py:class:**do_mpc.model.Model**. These values are plotted.
- **kwargs** – Further arguments are passed to the call of plt.subplots(n_plot, 1, sharex=True, **kwargs).

Returns

- **fig** (**Matplotlib figure**)
- **ax** (**Matplotlib axes**)
- **configured** Graphics object (Graphics)

This page is auto-generated. Page source is not available on Github.

### Sampling tools

- **samplingplanner**
- **sampler**
- **datahandler**

#### 5.11.8 samplingplanner

**Classes**

- **SamplingPlanner** A class for generating sampling plans.

#### 5.11.8.1 SamplingPlanner

**class** do_mpc.sampling.samplingplanner.SamplingPlanner

A class for generating sampling plans. These sampling plans will be executed by do_mpc.sampling.sampler.Sampler to generate data.

Configuration and sampling plan generation:

1. Set variables which should be sampled with **set_sampling_var()**.
2. (Optional) Set further options of the SamplingPlanner with **set_param()**
3. Generate the sampling plan with **gen_sampling_plan()**.
4. And/or: Add specific sampling case with `add_sampling_case()`.
5. Export the plan with all sampling cases with `export()`.

**Attributes**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SamplingPlanner.data_dir</code></td>
<td>Set the save directory for the sampling plan.</td>
</tr>
</tbody>
</table>

### 5.11.8.1.1 data_dir

Class attribute.

**`SamplingPlanner.data_dir`**

Set the save directory for the sampling plan. If the directory does not exist yet, it is created. If the directory is nested all (non-existing) parent folders are also created.

**Example:**

```python
sp = do_mpc.sampling.SamplingPlanner()
sp.data_dir = './samples/experiment_1/'
```

This will set the directory to the indicated path. If the path does not exist, all folders are created.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
</table>
| `SamplingPlanner.add_sampling_case` | Manually add sampling case with user-defined values. Create a sampling case by choosing values for the previously introduced sampling variables (with `set_sampling_var()`). Method takes arbitrary (keyword, argument) pairs, where the keywords must refer to previously introduced sampling variables. `add_sampling_case()` will automatically augment the sampling case with values for variables that are not passed as arguments. This only works if these variables were created with the argument `fun_var_pdf`. **Example:**

**5.11.8.1.2 add_sampling_case**

Class method.

```python
do_mpc.sampling.samplingplanner.SamplingPlanner.add_sampling_case(self, **kwargs)
```
sp = do_mpc.sampling.SamplingPlanner()

# Plan with two variables alpha and beta:
sp.set_sampling_var('alpha', np.random.randn)
sp.set_sampling_var('beta', lambda: np.random.randint(0,5))

# Create two new sampling cases, missing variable is auto-generated:
sp.add_sampling_case(alpha=1)
sp.add_sampling_case(beta= 0)

This page is auto-generated. Page source is not available on Github.

### 5.11.8.1.3 export

**Class method.**

do_mpc.sampling.samplingplanner.SamplingPlanner.

#### .export

Export SamplingPlan in pickle format. Pass `sampling_plan_name` without any path. File extension can be added (but will be stripped automatically). Change the path with `data_dir`.

**Parameters**

- **sampling_plan_name** (str) – Name of the exported sampling plan file.

**Raises**

- assertion – `sampling_plan_name` must be string.

This page is auto-generated. Page source is not available on Github.

### 5.11.8.1.4 gen_sampling_plan

**Class method.**

do_mpc.sampling.samplingplanner.SamplingPlanner.

#### .gen_sampling_plan

Generate the sampling plan. The generated plan contains `n_samples` samples based on the defined variables and the corresponding evaluation functions.

**Parameters**

- **n_samples** (int) – The number of generated samples

**Raises**

- assertion – `n_samples` must be int

**Returns**

Returns the newly created sampling plan.

**Return type**

list

This page is auto-generated. Page source is not available on Github.

### 5.11.8.1.5 set_param

**Class method.**

do_mpc.sampling.samplingplanner.SamplingPlanner.

#### .set_param

Set the parameters of the SamplingPlanner class. Parameters must be passed as pairs of valid keywords and respective argument. For example:

```python
sp.set_param(overwrite = True)
```
It is also possible and convenient to pass a dictionary with multiple parameters simultaneously as shown in the following example:

```python
setup_dict = {
   'overwrite': True,
   'save_format': pickle,
}
sp.set_param(**setup_dict)
```

This makes use of the python “unpack” operator. See more details here.

**Note:** `set_param()` can be called multiple times. Previously passed arguments are overwritten by successive calls.

The following parameters are available:

**Parameters**

- **overwrite** *(bool)* – Overwrites existing sampling plan under the same name, if set to True.
- **id_precision** *(string)* – Padding for IDs of created samples. Defaults to 3. This means sample 20 will be denoted as 020.

This page is auto-generated. Page source is not available on Github.

### 5.11.8.1.6 set_sampling_var

Class method.

```python
do_mpc.sampling.samplingplanner.SamplingPlanner.set_sampling_var(self, name, fun_var_pdf=None)
```

Introduce new sampling variables to the `SamplingPlanner`. Define variable name. Optionally add a function to generate values for the sampled variable (e.g. following some distribution). The parameter `fun_var_pdf` defaults to `None`.

**Note:** If no value-generating function is passed (for any of the introduced variables), all sampling cases must be created manually with `add_sampling_case()`.

**Note:** Value generating function `fun_var_pdf` must not require inputs.

**Example:**

```python
sp = do_mpc.sampling.SamplingPlanner()

# Plan with two variables alpha and beta:
sp.set_sampling_var('alpha', np.random.randn)
sp.set_sampling_var('beta', lambda: np.random.randint(0,5))
```

In the example we have passed a `BuiltinFunction` for the introduced variable `alpha`. We use the function that created values from the random normal distribution with zero mean and unity covariance. For the variable `beta` we created a new lambda function that draws random integers from 0 to 5.
Parameters

- **name**: (string) – Name of the sampled variable
- **fun_var_pdf**: (*Function of BuiltinFunction*) – Declare the value-generating function of the sampled variable

 Raises

- **assertion**: name must be string
- **assertion**: fun_var_pdf must be Function or BuiltinFunction

This page is auto-generated. Page source is not available on Github.

This page is auto-generated. Page source is not available on Github.

This page is auto-generated. Page source is not available on Github.

5.11.9 sampler

Classes

| Sampler | Generate samples based on a sampling plan. |

5.11.9.1 Sampler

**class** do_mpc.sampling.sampler.Sampler(*sampling_plan*)

Generate samples based on a sampling plan. Initiate the class by passing a do_mpc.sampling.samplingplanner.SamplingPlanner(*sampling_plan*) object. The class can be configured to create samples based on the defined cases in the *sampling_plan*.

Configuration and sampling:

1. (Optional) use `set_param()` to configure the class. Use `data_dir` to choose the save location for the samples.
2. Set the sample generating function with `set_sample_function()`. This function is executed for each of the samples in the *sampling_plan*.
3. Use `sample_data()` to generate all samples defined in the *sampling_plan*. A new file is written for each sample.
4. Or: Create an individual sample result with `sample_idx()`, where an index (int) referring to the *sampling_plan* determines the sampled case.

**Note:** By default, the Sampler will only create samples that do not already exist in the chosen `data_dir`.

**Example:**

```python
sp = do_mpc.sampling.SamplingPlanner()

# Plan with two variables alpha and beta:
sp.set_sampling_var('alpha', np.random.randn)
sp.set_sampling_var('beta', lambda: np.random.randint(0,5))
plan = sp.gen_sampling_plan(n_samples=10)
```

(continues on next page)
sampler = do_mpc.sampling.Sampler(plan)

# Sampler computes the product of two variables alpha and beta
# that were created in the SamplingPlanner:

def sample_function(alpha, beta):
    return alpha*beta

sampler.set_sample_function(sample_function)
sampler.sample_data()

Attributes

| **Sampler.data_dir** | Set the save directory for the results. |

5.11.9.1.1 data_dir

Class attribute.

**Sampler.data_dir**
Set the save directory for the results. If the directory does not exist yet, it is created. If the directory is nested all (non-existing) parent folders are also created.

**Example:**

```python
sampler = do_mpc.sampling.Sampler()
sampler.data_dir = './samples/experiment_1/'
```

This will set the directory to the indicated path. If the path does not exist, all folders are created.

Methods

| **Sampler.sample_data** | Sample data after having configured the Sampler. |
| **Sampler.sample_idx** | Sample case based on the index of the sample. |
| **Sampler.set_param** | Configure the Sampler class. |
| **Sampler.set_sample_function** | Set sample generating function. |

5.11.9.1.2 sample_data

Class method.

```python
do_mpc.sampling.sampler.Sampler.sample_data(self)
```
Sample data after having configured the Sampler. No user input is required and the method will iterate through all the items defined in the sampling_plan (obtained with `do_mpc.sampling.samplingplanner.SamplingPlanner`).

**Note:** Depending on your sample_function (set with `set_sample_function()`) and the total
number of samples, executing this method may take some time.

Note: If sampler.set_param(overwrite = False) (default) data will only be sampled for instances that do not yet exist.

This page is auto-generated. Page source is not available on Github.

5.11.9.1.3 sample_idx

Class method.

do_mpc.sampling.sampler.Sampler.sample_idx(self, idx)

Sample case based on the index of the sample.

Parameters

idx (int) – Index of the sampling_plan for which the sample should be created.

Raises

• assertion – Index must be between 0 and n_samples.
• assertion – sample_function must be set prior to sampling data.

This page is auto-generated. Page source is not available on Github.

5.11.9.1.4 set_param

Class method.

do_mpc.sampling.sampler.Sampler.set_param(self, **kwargs)

Configure the Sampler class.

Parameters must be passed as pairs of valid keywords and respective argument. For example:

sampler.set_param(overwrite = True)

Parameters

• overwrite (bool) – Should previously created results be overwritten. Default is False.
• sample_name (str) – Naming scheme for samples.

Save_format Choose either pickle or mat.

Print_progress Print progress-bar to terminal. Default is True.

This page is auto-generated. Page source is not available on Github.

5.11.9.1.5 set_sample_function

Class method.
do_mpc.sampling.sampler.Sampler.set_sample_function(self, sample_function)

Set sample generating function. The sampling function produces a sample result for each sample definition in the sampling_plan and is called in the method sample_data().

It is important that the sample function only uses keyword arguments with the same name as previously defined in the sampling_plan.

Example:

```python
sp = do_mpc.sampling.SamplingPlanner()
sp.set_sampling_var('alpha', np.random.randn)
sp.set_sampling_var('beta', lambda: np.random.randint(0,5))
sampler = do_mpc.sampling.Sampler(plan)

def sample_function(alpha, beta):
    return alpha*beta
sampler.set_sample_function(sample_function)
```

Parameters sample_function (FunctionType) – Function to create each sample of the sampling plan.

This page is auto-generated. Page source is not available on Github.

This page is auto-generated. Page source is not available on Github.

This page is auto-generated. Page source is not available on Github.

5.11.10 datahandler

Classes

DataHandler  Post-processing data created from a sampling plan.

5.11.10.1 DataHandler

class do_mpc.sampling.datahandler.DataHandler(sampling_plan)

Post-processing data created from a sampling plan. Data (individual samples) were created with do_mpc.sampling.sampler.Sampler. The list of all samples originates from do_mpc.sampling.samplingplanner.SamplingPlanner and is used to initiate this class (sampling_plan).

Configuration and retrieving processed data:

1. Initiate the object with the sampling_plan originating from do_mpc.sampling.samplingplanner.SamplingPlanner.
2. Set parameters with set_param(). Most importantly, the directory in which the individual samples are located should be passe with data_dir argument.
3. (Optional) set one (or multiple) post-processing functions. These functions are applied to each loaded sample and can, e.g., extract or compile important information.
4. Load and return samples either by indexing with the __getitem__() method or by filtering with filter().

5.11. API Reference
Example:

```python
sp = do_mpc.sampling.SamplingPlanner()

# Plan with two variables alpha and beta:
sp.set_sampling_var('alpha', np.random.randn)
sp.set_sampling_var('beta', lambda: np.random.randint(0, 5))

plan = sp.gen_sampling_plan(n_samples=10)
sampler = do_mpc.sampling.Sampler(plan)

# Sampler computes the product of two variables alpha and beta
# that were created in the SamplingPlanner:

def sample_function(alpha, beta):
    return alpha*beta

sampler.set_sample_function(sample_function)
sampler.sample_data()

# Create DataHandler object with same plan:
dh = do_mpc.sampling.DataHandler(plan)

# Assume you want to compute the square of the result of each sample
dh.set_post_processing('square', lambda res: res**2)

# As well as the value itself:
dh.set_post_processing('default', lambda res: res)

# Query all post-processed results with:
dh[:]
```

**Attributes**

<table>
<thead>
<tr>
<th>DataHandler.data_dir</th>
<th>Set the directory where the results are stored.</th>
</tr>
</thead>
</table>

**5.11.10.1.1 data_dir**

Class attribute.

DataHandler.data_dir

Set the directory where the results are stored.

This page is auto-generated. Page source is not available on Github.

**Methods**

<table>
<thead>
<tr>
<th>DataHandler.filter</th>
<th>Filter data from the DataHandler.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataHandler.set_param</td>
<td>Set the parameters of the DataHandler.</td>
</tr>
<tr>
<td>DataHandler.set_post_processing</td>
<td>Set a post processing function.</td>
</tr>
</tbody>
</table>
5.11.10.1.2 filter

Class method.

```
do_mpc.sampling.datahandler.DataHandler.filter(self, input_filter=None, output_filter=None)
```

Filter data from the DataHandler. Filters can be applied to inputs or to results that were obtained with the post-processing functions. Filtering returns only a subset from the created samples based on arbitrary conditions.

**Example:**

```
sp = do_mpc.sampling.SamplingPlanner()

# SamplingPlanner with two variables alpha and beta:
sp.set_sampling_var('alpha', np.random.randn)
sp.set_sampling_var('beta', lambda: np.random.randint(0,5))
plan = sp.gen_sampling_plan()

... 

dh = do_mpc.sampling.DataHandler(plan)
dh.set_post_processing('square', lambda res: res**2)

# Return all samples with alpha < 0 and beta > 2
dh.filter(input_filter = lambda alpha, beta: alpha < 0 and beta > 2)
# Return all samples for which the computed value square < 5
dh.filter(output_filter = lambda square: square < 5)
```

**Parameters**

- **filter_fun** – Function to filter the data.

**Raises**

- **assertion** – No post processing function is set

- **assertion** – filter_fun must be either Function or BuiltinFunction or Method

**Returns**

Returns the post processed samples that satisfy the filter

**Return type**

dict

This page is auto-generated. Page source is not available on Github.

5.11.10.1.3 set_param

Class method.

```
do_mpc.sampling.datahandler.DataHandler.set_param(self, **kwargs)
```

Set the parameters of the DataHandler.

Parameters must be passed as pairs of valid keywords and respective argument. For example:

```
datahandler.set_param(overwrite = True)
```

**Parameters**

- **data_dir** (bool) – Directory where the data can be found (as defined in the do_mpc.sampling.sampler.Sampler).
• **sample_name** *(str)* – Naming scheme for samples (as defined in the *do_mpc*sampling.sampler.Sampler).

**Save_format** Choose either pickle or mat (as defined in the *do_mpc*sampling.sampler.Sampler).

This page is auto-generated. Page source is not available on Github.

5.11.10.1.4 set_post_processing

Class method.

do_mpc.sampling.datahandler.DataHandler.set_post_processing(*self, name, post_processing_function*)

Set a post processing function. The post processing function is applied to all loaded samples, e.g. with
__getitem__() or filter(). Users can set an arbitrary amount of post processing functions by
repeatedly calling this method.

The post_processing_function can have two possible signatures:

1. post_processing_function(case_definition, sample_result)
2. post_processing_function(sample_result)

Where case_definition is a dict of all variables introduced in the *do_mpc*sampling.
samplingplanner.SamplingPlanner and sample_results is the result obtained from the
function introduced with *do_mpc*sampling.sampler.Sampler.set_sample_function.

**Note:** Setting a post processing function with an already existing name will overwrite the previously set
post processing function.

Example:

```python
sp = do_mpc.sampling.SamplingPlanner()

# Plan with two variables alpha and beta:
sp.set_sampling_var('alpha', np.random.randn)
sp.set_sampling_var('beta', lambda: np.random.randint(0,5))

plan = sp.gen_sampling_plan(n_samples=10)
sampler = do_mpc.sampling.Sampler(plan)

# Sampler computes the product of two variables alpha and beta
# that were created in the SamplingPlanner:

def sample_function(alpha, beta):
    return alpha*beta

sampler.set_sample_function(sample_function)
sampler.sample_data()

# Create DataHandler object with same plan:
dh = do_mpc.sampling.DataHandler(plan)
```

(continues on next page)
# Assume you want to compute the square of the result of each sample

dh.set_post_processing('square', lambda res: res**2)

# As well as the value itself:

dh.set_post_processing('default', lambda res: res)

# Query all post-processed results with:

dh[ : ]

## Parameters

- **name** *(string)* – Name of the output of the post-processing operation

- **post_processing_function** – The post processing function to be evaluated

## Raises

- **assertion** – name must be string

- **assertion** – post_processing_function must be either Function of BuiltinFunction

This page is auto-generated. Page source is not available on Github.

This content is autogenereated from our Github release notes.

5.12.1 v4.3.5

5.12.1.1 Minor fixes

- Setup for release in v4.3.4 was incomplete.

5.12.2 v4.3.4

5.12.2.1 Minor fixes

- `model.aux` can now be queried before calling `model.setup()`

  - Some typos in documentation

5.12.3 v4.3.3

5.12.3.1 Major changes

- `DataHandler` class can now create `post_processing_function` considering inputs from the case-definition as created in the `SamplingPlanner`.
5.12.3.2 Minor changes

- All cost terms that are continuously appended to are now initialised with value $D M(0)$. If nothing is appended (i.e. the term is not active), this avoids unclear error messages. This should fix #214 and #86

5.12.3.3 Documentation

Minor fixes.

5.12.3.4 Example files

Please download the example files for release v4.3.3 here.

5.12.4 v4.3.2.

5.12.4.1 Major fixes

- Solved #215
- Hoping to solve #233

5.12.5 v4.3.1

Fixed an issue with release 4.3.0 where sampling tools where not included on pypi.

5.12.6 v4.3.0

5.12.6.1 Major changes

5.12.6.1.1 do-mpc sampling tools

With this release we are integrated a major new feature in do-mpc which was originally started at the do-mpc developer conference 2021. To learn more about the new feature we have prepared a video tutorial.

5.12.6.2 Minor changes

- Fixed an issue with saving computation time in MHE/MPC in data object.
- New example: Kite systems

5.12.6.3 Example files

Please download the example files for release v4.3.0 here.
5.12.7 v4.2.5

5.12.7.1 Major changes

Full customization of the MPC or MHE optimization problem is now possible. Instead of using `MPC.setup()` to finalize the MPC optimization problem, an alternative two step process is now possible:

- `MPC.prepare_nlp()`
- `MPC.create_nlp()`

In between these two calls, users can add custom constraints and terms to the cost function using state, input etc. variables from different time-steps, collocation points or scenarios. A typical example would be to constrain changes of inputs or two enforce a cyclic behavior over the course of the horizon.

The new feature is fully documented and we suggest to have a look at the API reference for the MPC or MHE object.

5.12.7.2 Backend

5.12.7.2.1 Model

Internal functions in `do_mpc.model.Model` class have now properly named inputs and outputs. These inputs/outputs were previously automatically named `i0`, `i1`, .... They are now name e.g. `_x`, `_u`, `_z` ....

Here is an example (from the backend):

```python
self._rhs_fun = Function('rhs_fun',
                         [_x, _u, _z, _tvp, _p, _w], [self._rhs],
                         ### variables
                         "_["_x", "_u", "_z", "_tvp", "_p", "_w"]", ["_rhs\]"))
                         ### names
```

This may help for debugging because we now have that:

```python
model = do_mpc.model.Model("continuous")
....
model.setup()
print(model._rhs_fun)
```

Returns

```python
Function(rhs_fun:(_x[6],_u,_z[3],_tvp,_p[2],_w[0])->(_rhs[6]) SXFunction)
```

5.12.8 v4.2.0

5.12.8.1 Major changes

5.12.8.1.1 MX support

This addresses #34. The do-mpc model class can now be created with the `symvar_type` argument, defining whether the model is using CasADi SX or MX optimization variables.

```python
model = do_mpc.model.Model('continuous', 'MX')
```
all classes (MPC, MHE, Simulator ...) created from a MX model will also use MX variables. From a users-perspective
the change has no significant influence on the experience. It does, however, allow for significantly faster matrix vector
operations, which is main motivation to use the MX support.

The new feature resulted in some major changes to the backend. This is because CasADi does not allow (e.g.):

```python
x = MX.sym('x')
struct_symMX([
    entry('x', sym=x)
])
```

on which the model configuration heavily relied on.

Most importantly:

- The Model class attributes _x, _u etc. are now dicts prior to calling Model.setup.
- Calling model['x'] still works prior to calling Model.setup but works differently internally
- a new method _convert2struct converts dicts (e.g. of all the states) to symbolic structures (used in
Model.setup). The only problem: These structs hold variables with the same name but which are differ-
ent.
- a new method _substitute_struct_vars is introduced and substitutes the variables in the dicts in all
expressions (e.g. _rhs with the new variables from the symbolic structs.
- the MHE also required some major internal changes. The problem is that we split the parameters (_p) for the
MHE into estimated and set parameters. Splitting symbolic variables with the MX type is problematic.

5.12.8.2 Minor changes

- Solved #149: Option to only have a single slack variable (for each soft-constraint) over the entire horizon

5.12.8.3 Bug fixes

- Resolves #89. Discrete-time model now inherits its properties to MHE/MPC etc.

5.12.9 v4.1.1

5.12.9.1 Major changes

5.12.9.1.1 Adapted time-varying parameters for MPC object

Time-varying parameters (tvp) are now defined for k=0, ..., N+1 as opposed to k=0, ..., N in the previous ver-
sion. The main consequence is that the mterm for mpc.set_objective can now include the tvp in its expression.
This is beneficial e.g. for set-point tracking.

5.12.9.2 Documentation

Time-varying parameters are also described in greater detail now in this article.
5.12.10 do-mpc v4.1.0

5.12.10.1 Major changes

5.12.10.1.1 DAE support

This addresses the long overdue #3 (and closes #36). DAE works for both discrete time and continuous time formulations.

- DAE’s are introduced in the model with the set_alg method.
- Algebraic states are introduced with the set_variable method and have the var_type='_z'.
- The model checks that for each newly introduced algebraic state there must be one new algebraic equation. Otherwise the problem is under-determined.
- Algebraic states can be scaled and bounded in both MHE and MPC similar to states, inputs etc. The algebraic equations itself are not automatically scaled. This is different for the ODE which is scaled with the scaling factor for its respective state.

Continuous time (orthogonal collocation)

When using DAEs with continuous time models the DAE equation is added as an additional constraint at each collocation point (both for MHE/MPC).

The simulator must use the idas integration tool (or some other tool supporting DAEs). The default tool cvodes does not support DAE equations.

Discrete time

When using DAEs with continuous time models the DAE equation is added as an additional constraint at each time-step (both for MHE/MPC).

The simulator cannot simply evaluate the discrete time equation to obtain the next state as it is an expression of the unknown algebraic states. Thus we first solve the algebraic equation with the current state, input etc (using nlpsol) and then evaluate the discrete time equation with the obtained algebraic states.

5.12.10.1.2 Constraints with MPC / MHE with orthogonal collocation

Added a flag that can be set during MPC / MHE setup to choose whether inequality constraints are evaluated for each collocation point or only on the beginning of the finite Element. The flag is set during setup of the MPC / MHE with the set_param method:

```python
mpc = do_mpc.controller.MPC(model)
setup_mpc = {
    'n_horizon': 20,
    't_step': 0.005,
    'nl_cons_check_colloc_points': True,
}
mpc.set_param(**setup_mpc)
```

Currently defaults to False.

5.12. Release notes
Added a flag that can be set during MPC / MHE setup to choose whether bounds (lower and upper) are evaluated for each collocation point or only on the beginning of the finite Element. The flag is set during setup of the MPC / MHE with the **set_param** method:

```python
mpc = do_mpc.controller.MPC(model)
setup_mpc = {
    'n_horizon': 20,
    't_step': 0.005,
    'cons_check_colloc_points': True,
}
mpc.set_param(**setup_mpc)
```

Currently defaults to True.

### 5.12.10.1.3 Terminal bounds for MPC

This fixes #35.

- The MPC controller now supports terminal bounds for the states which can be different to the generic state constraints set with the **bounds** attribute.
- Set terminal bounds with the new **terminal_bounds** attribute.
- If no terminal bounds are explicitly set, they default to the regular state bounds (this means that previously working examples won’t have to add terminal bounds to obtain the same results).
- If this behavior is undesired (e.g. terminal state should be unbounded even though all other states are bounded) set the parameter **use_terminal_bounds=False** during MPC setup.

### 5.12.10.2 Minor changes

- **MPC.set_objective**: The mterm (terminal cost) now allows parameters (_p) in the formulation.
- **Simulator.set_initial_guess**: Introduced this method to set the initial guess for the algebraic variables. The guess is based on the class attributes _z0 which is inline with how the estimator and controller work.
- **Simulator.make_step**: No longer takes the initial value/guess for _x0 and _z0 as arguments. The initial state _x0 can be changed via its class attribute whereas the initial guess for _z0 is changed as described above.
- Addressed #71: The initial state is no longer constrained through upper and lower bounds.
- Addressed #65 and removed depreciated methods from all modules.

### 5.12.10.3 Documentation

- New non-linear example on the front page (double inverted pendulum with obstacle avoidance). This addresses #70.
- Fixed documentation of **MPC.opt_x_num**. This fixes #72

### 5.12.10.4 Example files

Please download the example files for release do-mpc v4.1.0 here.
5.12.11 do-mpc v4.0.0

We are finally out of beta with do-mpc v4.0.0. Thanks to everyone who has contributed, for the feedback and all the interest. This release includes some important changes and bugfixes and also significantly extends our homepage do-mpc.

We hope you will like the new features and content. Development will now continue with work on version 4.1.0 (and potentially some in between versions with minor features). Stay tuned on our Github page and feel free to open issues or join the discussion!

5.12.11.1 Major changes

5.12.11.1.1 New properties for Simulator, Estimator and MPC

Inheriting from the new class IteratedVariables these classes now obtain the attributes _x0, _u0, _z0 (and _p_est0). Users can access these attributes with the properties with x0, u0, z0 (and p_est0), which are listed in the documentation and have sanity checks etc. when setting them. This fixes e.g. #55. These new properties are used for two things:

Set initial values

For the simulator the initial state is self explanatory and a very important attribute. For the MHE and MPC class the attributes are used when calling the important set_initial_guess method, which does exactly that: Set the initial guess of the optimization problem.

Obtain the current values of the iterated variables

This is very useful for conditional MPC loops: E.g. stop the controller and simulation when a certain state has reached a certain value.

5.12.11.1.2 Measurement noise

Currently, the do_mpc.model.Model.set_rhs method allows to set an additive process noise. This is used for the MHE optimization problem. In a similar fashion, the do_mpc.model.Model.set_meas method now allows to set an additive measurement noise.

In the MHE the measurement noise is introduced as a new optimization variable and the measurement equation is added as an additional constraint. The full optimization problem now looks like this:

\[
\min_{x_0, u, z_0, p, \ldots} \frac{1}{2} \| x_0 - \bar{x}_0 \|_P^2 + \frac{1}{2} \| p - \bar{p} \|_P^2 + \sum_{k=0}^{N-1} \left( \frac{1}{2} \| u_k \|_{P_{uk}}^2 + \frac{1}{2} \| w_k \|_{P_{wk}}^2 \right),
\]

s. t. \[
\begin{align*}
    x_{k+1} &= f(x_k, u_k, z_k, p, p_{tv,k}) + u_k, \\
    y_k &= h(x_k, u_k, z_k, p, p_{tv,k}) + v_k, \\
    g(x_k, u_k, z_k, p_k, p_{tv,k}) &\leq 0
\end{align*}
\]

This change makes it possible for the user to decide, which measurements are enforced and which can be perturbed. A typical example would be to ensure that input “measurements” are completely trusted.
5.12.11.3 Simulator with disturbances

The newly introduced measurement noise and the existing process noise are now used within the simulator. With each call of `Simulator.make_step` values can be passed to obtain an imperfectly simulated and measured system.

5.12.11.2 Documentation

- Release notes are now included in the documentation. They are autogenerated from the Github release notes which can be accessed via Rest API.
- The release notes are appended with a section that includes a download link for the example files that were written for the respective versions.
- Installation instructions now refer to these download links. This solves #62.
- Added new section Example gallery, explaining the supplied examples in `do-mpc` in Jupyter Notebooks (rendered on readthedocs)
- Added new section Background with various articles explaining the mathematics behind `do-mpc`.
- Parameter `collocation_ni` in MPC/MHE is now explained more clearly.

5.12.11.3 Minor changes

- Renamed `model.setup_model()` -> `model.setup()` in all examples. This addresses #38
- `opt_p_num` and `opt_x_num` for MHE/MPC are now instance properties instead of class attributes. They still appear in the documentation and can be used as before. Having them as class attributes can lead to problems when multiple classes are live during the same session.

5.12.11.4 Example files

Please download the example files for release do-mpc v4.0.0 here.

5.12.12 do-mpc v4.0.0-beta3

5.12.12.1 Major changes

5.12.12.1.1 Data

- New `__getitem__` method to conveniently retrieve values from `Data` object (details here)
- New `MPCData` class (which inherits from `Data`). This adds the `prediction` method, which can be used to query the optimal trajectories. Details here.

Both methods were previously (in a slightly different form) in the Graphics module. They are still used in this class but can also be convenient under different circumstances.

5.12.12.1.2 Graphics

The Graphics module is now initialized with a specific `Data` instance (e.g. `mpc.data`). Each `Data` class has their own Graphics class (if it is supposed to be displayed). Compared to the previous implementation, we now initialize
all lines that are supposed to be plotted (and store them in `pred_lines` and `result_lines`). During runtime, the data on these lines is getting updated.

- Added new structure class in `do_mpc.tools`. Used for tracking the new Graphics properties: `pred_lines` and `result_lines`.
- The properties `pred_lines` and `result_lines` can be used to retrieve line instances with power indices. Line instances can be easily configured (linestyle, alpha, color etc.)

### 5.12.12.1.3 Process noise

Process noise can be added to rhs of `Model` class: [link](#)

This is solving issue #53.

This change was necessary to allow for the more natural MHE formulation where the process noise is penalised in the cost function. The user can define for each state (vector) individually if this is intended or not.

As a consequence of this change I had to introduce the new variable $w$ throughout `do-mpc`. For the MPC and simulator module this is without effect.

The main difference is here

**Remark:** The change also allows to estimate parameters that change over time (e.g. environmental influences). Our regular estimated parameters are constant over the entire MHE horizon, which is not always valid. To estimate varying parameters, they should be defined as states with unknown dynamics. Concretely, their RHS is zero (for ODEs) and they have a high process noise.

### 5.12.12.1.4 Symbolic variables for MHE weighting matrices

As originally intended, it is now possible to have symbolic matrices as MHE tuning factors. The result of this change can be seen in the [rotating_oscillating_masses example](#).

The symbolic variables are defined in the `do_mpc Model` where typically, you want to have $P\_x$ and $P\_p$ as parameters and $P\_y$ and $P\_w$ as time-varying parameters. Example of their definition.

and here they are used.

The purpose of using symbolic weighting is of course to update them at each iteration. Since they are parameters and time-varying parameters respectively, this is done with the `set_p_fun` and `set_tvp_fun` method of the MHE: [link](#)

Note that in the example above, we don’t actually need varying weighting matrices and the returned values are in fact constant. This can be seen as a proof of concept.

This change had some other implications. Most notably, having additional parameters interferes with the multi-stage robust MPC module. Where we previously had to pass a number of scenarios for each defined parameter. Since parameters for the MHE are irrelevant for MPC the API for the call `set_uncertainty_values` has changed: [link](#)

**The new API is fully backwards compatible.** However, it is much more intuitive now. The function is called with keyword arguments, where each keyword refers to one uncertain parameter (note that we can ignore the parameters that are irrelevant). In practice this looks something like [this](#)

### 5.12.12.2 Example files

Please download the example files for release do-mpc v4.0.0-beta3 [here](#).
5.12.13 do-mpc v4.0.0-beta2

Error in release. Immediately replaced with beta3.

5.12.14 do-mpc v4.0.0-beta1

5.12.14.1 Major changes

- We are now explicitly pointing out attributes of the Model such as states, inputs, etc. These should be used to obtain these attributes and replace the previous get_variables method which is now depreciated. The Model also supports a __get_variable__ call now to conveniently select items.

- setup_model is replaced by setup to be more consistent with other setup methods. The old method is still available and shows a depreciation warning.

- The MHE now supports the set_default_objective method.

5.12.14.2 Bug fixes

- The MHE formulation had an error in the make_step method. We used the wrong time step from the previous solution to compute the arrival cost.

5.12.14.3 Other changes

- Spelling in documentation

- New guide about installing HSL linear solver

- Credits in documentation

5.12.14.4 Example files

Please download the example files for release do-mpc v4.0.0-beta1 here.

5.12.15 do-mpc v4.0.0-beta

do-mpc has undergone a massive overhaul and comes with a completely new interface, new features and a comprehensive documentation.

Please note that previously written code is not compatible with do-mpc 4.0.0. If you want to continue working with older code please use version 3.0.0.

This is the beta release of version 4.0.0. We expect minor modifications and bug fixes in the near future.

Please see our documentation on our new project homepage www.do-mpc.com for a full list of features.

5.12.15.1 Example files

Please download the example files for release do-mpc v4.0.0-beta here.
5.12.16 do-mpc v3.0.0

5.12.16.1 Main modifications

- Support for CasADi version 3.4.4
- Support for time-varying parameters
- Support for discrete-time systems

5.12.17 do-mpc v2.0.0

Compatible with CasADi 3.0.0

5.12.18 do-mpc version 1.0.0

5.13 Batch Bioreactor

In this Jupyter Notebook we illustrate the example batch_reactor.

Open an interactive online Jupyter Notebook with this content on Binder:

The example consists of the three modules template_model.py, which describes the system model, template_mpc.py, which defines the settings for the control and template_simulator.py, which sets the parameters for the simulator. The modules are used in main.py for the closed-loop execution of the controller.

In the following the different parts are presented. But first, we start by importing basic modules and do-mpc.

```python
[1]:
import numpy as np
import sys
from casadi import *

# Add do_mpc to path. This is not necessary if it was installed via pip
sys.path.append('..../..../')

# Import do_mpc package:
import do_mpc
```

5.13.1 Model

In the following we will present the configuration, setup and connection between these blocks, starting with the model.

The considered model of the batch bioreactor is continuous and has 4 states and 1 control input, which are depicted below:

The model is initiated by:

```python
[2]:
model_type = 'continuous'  # either 'discrete' or 'continuous'
model = do_mpc.model.Model(model_type)
```
5.13.1.1 States and control inputs

The four states are concentration of the biomass $X_s$, the concentration of the substrate $S_s$, the concentration of the product $P_s$ and the volume $V_s$:

$$[3]: \begin{align*}
\text{# States struct (optimization variables):} \\
X_s &= \text{model.set_variable('}_x', 'X_s') \\
S_s &= \text{model.set_variable('}_x', 'S_s') \\
P_s &= \text{model.set_variable('}_x', 'P_s') \\
V_s &= \text{model.set_variable('}_x', 'V_s') \\
\end{align*}$$

The control input is the feed flow rate $u_{\text{inp}}$ of $S_s$:

$$[4]: \begin{align*}
\text{# Input struct (optimization variables):} \\
inp &= \text{model.set_variable('}_u', 'inp') \\
\end{align*}$$

5.13.1.2 ODE and parameters

The system model is described by the ordinary differential equation:

$$\begin{align*}
\dot{X}_s &= \mu(S_s)X_s - \frac{u_{\text{inp}}}{V_s} X_s, \quad (5.25) \\
\dot{S}_s &= -\frac{\mu(S_s)X_s}{Y_s} - \frac{vX_s}{V_p} + \frac{u_{\text{inp}}}{V_s} (S_m - S_s), \quad (5.26) \\
\dot{P}_s &= vX_s - \frac{u_{\text{inp}}}{V_s} P_s, \quad (5.27) \\
\dot{V}_s &= u_{\text{inp}}, \quad (5.28)
\end{align*}$$

where:

$$\mu(S_s) = \frac{\mu_m S_s}{K_m + S_s + (S_s^2/K_i)}, \quad (5.30)$$

$S_m$ is the inlet substrate concentration, $\mu_m$, $K_m$, $K_i$ and $v$ are kinetic parameters $Y_s$ and $Y_p$ are yield coefficients. The inlet substrate concentration $S_m$ and the $Y_s$ are uncertain while the rest of the parameters is considered certain:

$$\begin{align*}
\text{# Certain parameters} \\
\mu_m &= 0.02 \\
K_m &= 0.05 \\
K_i &= 5.0 \\
v_{\text{par}} &= 0.004 \\
Y_p &= 1.2 \\
\text{# Uncertain parameters:} \\
Y_x &= \text{model.set_variable('}_p', 'Y_x') \\
S_in &= \text{model.set_variable('}_p', 'S_in') \\
\end{align*}$$

In the next step, the ODE for each state is set:

$$\begin{align*}
\text{# Auxiliary term} \\
\mu_{\text{S}} &= \mu_m S_s / (K_m + S_s + (S_s^2/K_i)) \\
\text{# Differential equations} \\
\text{model.set_rhs('}_s', \mu_{\text{S}} X_s - inp/V_s X_s) \\
\end{align*}$$
Finally, the model setup is completed:

```python
model.set_rhs('S_s', -mu_S*X_s/Y_x - v_par*X_s/Y_p + inp/V_s*(S_in-S_s))
model.set_rhs('P_s', v_par*X_s - inp/V_s*P_s)
model.set_rhs('V_s', inp)
```

### 5.13.2 Controller

Next, the controller is configured. First, one member of the mpc class is generated with the prediction model defined above:

```python
mpc = do_mpc.controller.MPC(model)
```

We choose the prediction horizon `n_horizon`, set the robust horizon `n_robust` to 3. The time step `t_step` is set to one second and parameters of the applied discretization scheme orthogonal collocation are as seen below:

```python
setup_mpc = {
    'n_horizon': 20,
    'n_robust': 1,
    'open_loop': 0,
    't_step': 1.0,
    'state_discretization': 'collocation',
    'collocation_type': 'radau',
    'collocation_deg': 2,
    'collocation_ni': 2,
    'store_full_solution': True,
    # Use MA27 linear solver in ipopt for faster calculations:
    #'nlpsol_opts': {'ipopt.linear_solver': 'MA27'}
}
mpc.set_param(**setup_mpc)
```

#### 5.13.2.1 Objective

The batch bioreactor is used to produce penicillin. Hence, the objective of the controller is to maximize the concentration of the product $P_s$. Additionally, we add a penalty on input changes, to obtain a smooth control performance.

```python
mterm = -model.x['P_s'] # terminal cost
lterm = -model.x['P_s'] # stage cost
mpc.set_objective(mterm=mterm, lterm=lterm)
mpc.set_rterm(inp=1.0) # penalty on input changes
```

#### 5.13.2.2 Constraints

In the next step, the constraints of the control problem are set. In this case, there are only upper and lower bounds for each state and the input:

```python
```
5.13.2.3 Uncertain values

The explicit values of the two uncertain parameters $Y_x$ and $S_{in}$, which are considered in the scenario tree, are given by:

```python
Y_x_values = np.array([0.5, 0.4, 0.3])
S_in_values = np.array([200.0, 220.0, 180.0])
mpc.set_uncertainty_values(Y_x = Y_x_values, S_in = S_in_values)
```

This means with $n_{robust}=1$, that 9 different scenarios are considered. The setup of the MPC controller is concluded by:

```python
mpc.setup()
```

5.13.3 Estimator

We assume, that all states can be directly measured (state-feedback):

```python
estimator = do_mpc.estimator.StateFeedback(model)
```

5.13.4 Simulator

To create a simulator in order to run the MPC in a closed-loop, we create an instance of the `do-mpc` simulator which is based on the same model:

```python
simulator = do_mpc.simulator.Simulator(model)
```

For the simulation, we use the time step $t_{step}$ as for the optimizer:

```python
params_simulator = {
    'integration_tool': 'cvodes',
    'abstol': 1e-10,
    'reltol': 1e-10,
    't_step': 1.0
}
simulator.set_param(**params_simulator)
```
5.13.4.1 Realizations of uncertain parameters

For the simulation, it is necessary to define the numerical realizations of the uncertain parameters in \( p_{\text{num}} \). First, we get the structure of the uncertain parameters:

\[
\text{p}_{\text{num}} = \text{simulator.get_p_template()}
\]

We define a function which is called in each simulation step, which gives the current realization of the uncertain parameters, with respect to defined inputs (in this case \( t_{\text{now}} \)):

\[
\begin{align*}
\text{p}_{\text{num}}[\text{'Y}_x'] &= 0.4 \\
\text{p}_{\text{num}}[\text{'S}_{\text{in}}'] &= 200.0 \\
\end{align*}
\]

# function definition
def p_fun(t_now):
    return p_num

# Set the user-defined function above as the function for the realization of the uncertain parameters
simulator.set_p_fun(p_fun)

By defining \( p_{\text{fun}} \) as above, the function will always return the same values. To finish the configuration of the simulator, call:

\[
\text{simulator.setup()}
\]

5.13.5 Closed-loop simulation

For the simulation of the MPC configured for the batch bioreactor, we inspect the file \texttt{main.py}. We define the initial state of the system and set for all parts of the closed-loop configuration:

\[
\begin{align*}
\text{X}_s_0 &= 1.0 & \text{Concentration biomass [mol/l]} \\
\text{S}_s_0 &= 0.5 & \text{Concentration substrate [mol/l]} \\
\text{P}_s_0 &= 0.0 & \text{Concentration product [mol/l]} \\
\text{V}_s_0 &= 120.0 & \text{Volume inside tank [m}^3\text{]} \\
\text{x0} &= \text{np.array([X}_s_0, \text{S}_s_0, \text{P}_s_0, \text{V}_s_0]) \\
\end{align*}
\]

# Set for controller, simulator and estimator
mpc.x0 = x0
simulator.x0 = x0
estimator.x0 = x0
mpc.set_initial_guess()

5.13.5.1 Prepare visualization

For the visualization of the control performance, we first import matplotlib and change some basic settings:

\[
\begin{align*}
\text{import matplotlib.pyplot as plt} \\
\text{plt.ion()} \\
\text{from matplotlib import rcParams} \\
\text{rcParams['text.usetex'] = True} \\
\text{rcParams['text.latex.preamble'] = [r'\usepackage{amsmath},r'\usepackage{siunitx}'] } \\
\text{rcParams['axes.grid'] = True} \\
\end{align*}
\]

(continues on next page)
We use the plotting capabilities, which are included in do-mpc. The mpc_graphics contain information like the current estimated state and the predicted trajectory of the states and inputs based on the solution of the control problem. The sim_graphics contain the information about the simulated evaluation of the system.

```python
mpc_graphics = do_mpc.graphics.Graphics(mpc.data)
sim_graphics = do_mpc.graphics.Graphics(simulator.data)
```

A figure containing the 4 states and the control input are created:

```python
for g in [sim_graphics,mpc_graphics]:
    # Plot the state on axis 1 to 4:
    g.add_line(var_type='_x', var_name='X_s', axis=ax[0], color='#1f77b4')
    g.add_line(var_type='_x', var_name='S_s', axis=ax[1], color='#1f77b4')
    g.add_line(var_type='_x', var_name='P_s', axis=ax[2], color='#1f77b4')
    g.add_line(var_type='_x', var_name='V_s', axis=ax[3], color='#1f77b4')

    # Plot the control input on axis 5:
    g.add_line(var_type='_u', var_name='inp', axis=ax[4], color='#1f77b4')
```

5.13.5.2 Run closed-loop

The closed-loop system is now simulated for 50 steps (and the output of the optimizer is suppressed):

```python
n_steps = 100
for k in range(n_steps):
    u0 = mpc.make_step(x0)
    y_next = simulator.make_step(u0)
    x0 = estimator.make_step(y_next)
```

5.13.5.3 Results

The next cell converts the results of the closed-loop MPC simulation into a gif (might take a few minutes):
from matplotlib.animation import FuncAnimation, FFMpegWriter, ImageMagickWriter

The function describing the gif:

def update(t_ind):
    sim_graphics.plot_results(t_ind)
    mpc_graphics.plot_predictions(t_ind)
    mpc_graphics.reset_axes()

if False:
    anim = FuncAnimation(fig, update, frames=n_steps, repeat=False)
    gif_writer = ImageMagickWriter(fps=10)
    anim.save('anim_batch_reactor_final.gif', writer=gif_writer)

The result is shown below, where solid lines are the recorded trajectories and dashed lines are the predictions of the scenarios:

5.14 Continuous stirred tank reactor (CSTR)

In this Jupyter Notebook we illustrate the example CSTR.

Open an interactive online Jupyter Notebook with this content on Binder:

The example consists of the three modules template_model.py, which describes the system model, template_mpc.py, which defines the settings for the control and template_simulator.py, which sets the parameters for the simulator. The modules are used in main.py for the closed-loop execution of the controller. The file post_processing.py is used for the visualization of the closed-loop control run. One exemplary result will be presented at the end of this tutorial as a gif.

In the following the different parts are presented. But first, we start by importing basic modules and do-mpc.

import numpy as np
import sys
from casadi import *

# Add do_mpc to path. This is not necessary if it was installed via pip
sys.path.append('..//..//..//')

# Import do_mpc package:
import do_mpc

import matplotlib.pyplot as plt

5.14.1 Model

In the following we will present the configuration, setup and connection between these blocks, starting with the model. The considered model of the CSTR is continuous and has 4 states and 2 control inputs. The model is initiated by:

define_model = 'continuous' # either 'discrete' or 'continuous'
model = do_mpc.model.Model(model_type)
5.14.1.1 States and control inputs

The four states are concentration of reactant A \( (C_A) \), the concentration of reactant B \( (C_B) \), the temperature inside the reactor \( (T_R) \) and the temperature of the cooling jacket \( (T_K) \):

```plaintext
[3]: # States struct (optimization variables):
C_a = model.set_variable(var_type='_x', var_name='C_a', shape=(1,1))
C_b = model.set_variable(var_type='_x', var_name='C_b', shape=(1,1))
T_R = model.set_variable(var_type='_x', var_name='T_R', shape=(1,1))
T_K = model.set_variable(var_type='_x', var_name='T_K', shape=(1,1))
```

The control inputs are the feed \( F \) and the heat flow \( \dot{Q} \):

```plaintext
[4]: # Input struct (optimization variables):
F = model.set_variable(var_type='_u', var_name='F')
Q_dot = model.set_variable(var_type='_u', var_name='Q_dot')
```

5.14.1.2 ODE and parameters

The system model is described by the ordinary differential equation:

\[
\dot{C}_A = F \cdot (C_{A,0} - C_A) - k_1 \cdot C_A - k_2 \cdot C_A^2 \tag{5.31}
\]

\[
\dot{C}_B = F \cdot C_B + k_1 \cdot C_A - k_2 \cdot C_B \tag{5.32}
\]

\[
\dot{T}_R = \frac{k_1 \cdot C_A \cdot H_{R,ab} + k_2 \cdot C_B \cdot H_{R,bc} + k_3 \cdot C_A^2 \cdot H_{R,ad}}{-\rho \cdot c_p} + F \cdot (T_{in} - T_R) + \frac{K_w \cdot A_R \cdot (T_K - T_R)}{\rho \cdot c_p \cdot V_R}, \tag{5.33}
\]

\[
\dot{T}_K = \frac{\dot{Q} + k_w \cdot A_R \cdot T_{df}}{m_k \cdot C_{p,k}}, \tag{5.35}
\]

where

\[
k_1 = \beta \cdot k_{0,ab} \cdot \exp \left( \frac{-E_{A,ab}}{T_R + 273.15} \right), \tag{5.36}
\]

\[
k_1 = k_{0,bc} \cdot \exp \left( \frac{-E_{A,bc}}{T_R + 273.15} \right), \tag{5.37}
\]

\[
k_3 = k_{0,ad} \cdot \exp \left( \frac{-\alpha \cdot E_{A,ad}}{T_R + 273.15} \right). \tag{5.38}
\]

The parameters \( \alpha \) and \( \beta \) are uncertain while the rest of the parameters is considered certain:

```plaintext
[5]: # Certain parameters
K0_ab = 1.287e12 # K0 [h^-1]
K0_bc = 1.287e12 # K0 [h^-1]
K0_ad = 9.043e9 # K0 [l/mol.h]
R_gas = 8.3144621e-3 # Universal gas constant
E_A_ab = 9758.3*1.00 #* R_gas# [kj/mol]
E_A_bc = 9758.3*1.00 #* R_gas# [kj/mol]
E_A_ad = 8560.0*1.0 #* R_gas# [kj/mol]
H_R_ab = 4.2 # [kj/mol A]
H_R_bc = -11.0 # [kj/mol B] Exothermic
H_R_ad = -41.85 # [kj/mol A] Exothermic
Rou = 0.9342 # Density [kg/l]
```
Cp = 3.01 # Specific Heat capacity [kJ/Kg.K]
Cp_k = 2.0 # Coolant heat capacity [kJ/kg.k]
A_R = 0.215 # Area of reactor wall [m^2]
V_R = 10.01 #0.01 # Volume of reactor [l]
m_k = 5.0 # Coolant mass[kg]
T_in = 130.0 # Temp of inflow [Celsius]
K_w = 4032.0 # [kj/h.m^2.K]
C_A0 = (5.7+4.5)/2.0*1.0 # Concentration of A in input Upper bound 5.7 lower bound 4.
# Uncertain parameters:
alpha = model.set_variable(var_type='_p', var_name='alpha')
beta = model.set_variable(var_type='_p', var_name='beta')

In the next step, we formulate the \( k_i \)-s:

\[
K_1 = \beta \times K0_ab \times \exp\left(\frac{-E_A_ab}{(T_R+273.15)}\right)
\]
\[
K_2 = K0_bc \times \exp\left(\frac{-E_A_bc}{(T_R+273.15)}\right)
\]
\[
K_3 = K0_ad \times \exp\left(\frac{-\alpha \times E_A_ad}{(T_R+273.15)}\right)
\]

Additionally, we define an artificial variable of interest, that is not a state of the system, but will be later used for plotting:

\[
T_{\text{dif}} = \text{model.set_expression(expr_name='T_{\text{dif}}', expr=T_R-T_K)}
\]

With the help of the \( k_i \)-s and \( T_{\text{dif}} \) we can define the ODEs:

\[
\text{model.set_rhs('C_a', F \times (C_a0 - C_a) - K_1 \times C_a - K_3 \times (C_a^2))}
\]
\[
\text{model.set_rhs('C_b', -F \times C_b + K_1 \times C_a - K_2 \times C_b)}
\]
\[
\text{model.set_rhs('T_R', (\frac{K_1 \times C_a \times H_{R_ab} + K_2 \times C_b \times H_{R_bc} + K_3 \times (C_a^2) \times H_{R_ad}}{-\color{red}{Rou \times Cp}}) + F \times (T_{\text{in}} - T_R) + (((K_w \times A_R) \times (-T_{\text{dif}}))/(Rou \times Cp \times V_R))))}
\]
\[
\text{model.set_rhs('T_K', (Q_dot + K_w \times A_R \times (T_{\text{dif}}))/(m_k \times Cp_k))}
\]

Finally, the model setup is completed:

\[
\text{# Build the model}
\text{model.setup()}
\]

5.14.2 Controller

Next, the model predictive controller is configured. First, one member of the mpc class is generated with the prediction model defined above:

\[
\text{mpc = do_mpc.controller.MPC(model)}
\]

We choose the prediction horizon \( n_{\text{horizon}} \), set the robust horizon \( n_{\text{robust}} \) to 1. The time step \( t_{\text{step}} \) is set to one second and parameters of the applied discretization scheme orthogonal collocation are as seen below:

\[
\text{setup_mpc = }
\{
'\text{'n_{horizon}': 20},
'\text{'n_{robust}': 1},
'\text{'open_{loop}': 0},
'\text{'t_{step}': 0.005},
\}
\]
Because the magnitude of the states and inputs is very different, we introduce scaling factors:

```python
[12]:
mpc.scaling['_x', 'T_R'] = 100
mpc.scaling['_x', 'T_K'] = 100
mpc.scaling['_u', 'Q_dot'] = 2000
mpc.scaling['_u', 'F'] = 100
```

### 5.14.2.1 Objective

The goal of the CSTR is to obtain a mixture with a concentration of $C_{B,ref} = 0.6$ mol/l. Additionally, we add a penalty on input changes for both control inputs, to obtain a smooth control performance.

```python
[13]:
_x = model.x
mterm = (_x['C_b'] - 0.6)**2  # terminal cost
lterm = (_x['C_b'] - 0.6)**2  # stage cost
mpc.set_objective(mterm=mterm, lterm=lterm)
mpc.set_rterm(F=0.1, Q_dot = 1e-3)  # input penalty
```

### 5.14.2.2 Constraints

In the next step, the constraints of the control problem are set. In this case, there are only upper and lower bounds for each state and the input:

```python
[14]:
# lower bounds of the states
mpc.bounds['lower', '_x', 'C_a'] = 0.1
mpc.bounds['lower', '_x', 'C_b'] = 0.1
mpc.bounds['lower', '_x', 'T_R'] = 50
mpc.bounds['lower', '_x', 'T_K'] = 50

# upper bounds of the states
mpc.bounds['upper', '_x', 'C_a'] = 2
mpc.bounds['upper', '_x', 'C_b'] = 2
mpc.bounds['upper', '_x', 'T_K'] = 140

# lower bounds of the inputs
mpc.bounds['lower', '_u', 'F'] = 5
mpc.bounds['lower', '_u', 'Q_dot'] = -8500

# upper bounds of the inputs
mpc.bounds['upper', '_u', 'F'] = 100
mpc.bounds['upper', '_u', 'Q_dot'] = 0.0
```
If a constraint is not critical, it is possible to implement it as a soft constraint. This means, that a small violation of the constraint does not render the optimization infeasible. Instead, a penalty term is added to the objective. Soft constraints can always be applied, if small violations can be accepted and it might even be necessary to apply MPC in a safe way (by avoiding avoiding numerical instabilities). In this case, we define the upper bounds of the reactor temperature as a soft constraint by using `mpc.set_nl_cons()`.

```python
[15]: mpc.set_nl_cons('T_R', _x['T_R'], ub=140, soft_constraint=True, penalty_term_cons=1e2)
[15]: SX((T_R-eps_T_R))
```

### 5.14.2.3 Uncertain values

The explicit values of the two uncertain parameters $\alpha$ and $\beta$, which are considered in the scenario tree, are given by:

```python
[16]: alpha_var = np.array([1., 1.05, 0.95])
beta_var = np.array([1., 1.1, 0.9])
mpc.set_uncertainty_values(alpha = alpha_var, beta = beta_var)
```

This means with $n_{robust}=1$, that 9 different scenarios are considered. The setup of the MPC controller is concluded by:

```python
[17]: mpc.setup()
```

### 5.14.3 Estimator

We assume, that all states can be directly measured (state-feedback):

```python
[18]: estimator = do_mpc.estimator.StateFeedback(model)
```

### 5.14.4 Simulator

To create a simulator in order to run the MPC in a closed-loop, we create an instance of the `do_mpc` simulator which is based on the same model:

```python
[19]: simulator = do_mpc.simulator.Simulator(model)
```

For the simulation, we use the same time step $t_{step}$ as for the optimizer:

```python
[20]: params_simulator = {
    'integration_tool': 'cvodes',
    'abstol': 1e-10,
    'reltol': 1e-10,
    't_step': 0.005
}

simulator.set_param(**params_simulator)
```

### 5.14.4.1 Realizations of uncertain parameters

For the simulation, it is necessary to define the numerical realizations of the uncertain parameters in $p_{num}$ and the time-varying parameters in $tvp_{num}$. First, we get the structure of the uncertain and time-varying parameters:

### 5.14. Continuous stirred tank reactor (CSTR)
We define two functions which are called in each simulation step, which return the current realizations of the parameters, with respect to defined inputs (in this case $t_{\text{now}}$):

```python
# function for time-varying parameters
def tvp_fun(t_now):
    return tvp_num

# uncertain parameters
p_num['alpha'] = 1
p_num['beta'] = 1
def p_fun(t_now):
    return p_num
```

These two custom functions are used in the simulation via:

```python
simulator.set_tvp_fun(tvp_fun)
simulator.set_p_fun(p_fun)
```

By defining $p_{\text{fun}}$ as above, the function will always return the value 1.0 for both $\alpha$ and $\beta$. To finish the configuration of the simulator, call:

```python
simulator.setup()
```

### 5.14.5 Closed-loop simulation

For the simulation of the MPC configured for the CSTR, we inspect the file `main.py`. We define the initial state of the system and set it for all parts of the closed-loop configuration:

```python
# Set the initial state of mpc, simulator and estimator:
C_a_0 = 0.8 # This is the initial concentration inside the tank [mol/l]
C_b_0 = 0.5 # This is the controlled variable [mol/l]
T_R_0 = 134.14 #[C]
T_K_0 = 130.0 #[C]
x0 = np.array([C_a_0, C_b_0, T_R_0, T_K_0]).reshape(-1,1)
mpc.x0 = x0
simulator.x0 = x0
estimator.x0 = x0
mpc.set_initial_guess()
```

Now, we simulate the closed-loop for 50 steps (and suppress the output of the cell with the magic command `%%capture`):

```python
for k in range(50):
    u0 = mpc.make_step(x0)
    y_next = simulator.make_step(u0)
    x0 = estimator.make_step(y_next)
```
5.14.6 Animating the results

To animate the results, we first configure the do-mpc graphics object, which is initiated with the respective data object:

```python
mpc_graphics = do_mpc.graphics.Graphics(mpc.data)
```

We quickly configure Matplotlib.

```python
from matplotlib import rcParams
rcParams['axes.grid'] = True
rcParams['font.size'] = 18
```

We then create a figure, configure which lines to plot on which axis and add labels.

```python
fig, ax = plt.subplots(5, sharex=True, figsize=(16,12))
# Configure plot:
mpc_graphics.add_line(var_type='_x', var_name='C_a', axis=ax[0])
mpc_graphics.add_line(var_type='_x', var_name='C_b', axis=ax[0])
mpc_graphics.add_line(var_type='_x', var_name='T_R', axis=ax[1])
mpc_graphics.add_line(var_type='_x', var_name='T_K', axis=ax[1])
mpc_graphics.add_line(var_type='_aux', var_name='T_dif', axis=ax[2])
mpc_graphics.add_line(var_type='_u', var_name='Q_dot', axis=ax[3])
mpc_graphics.add_line(var_type='_u', var_name='F', axis=ax[4])
ax[0].set_ylabel('c [mol/l]')
ax[1].set_ylabel('T [K]')
ax[2].set_ylabel('$\Delta$ T [K]')
ax[3].set_ylabel('Q [kW]')
ax[4].set_ylabel('Flow [l/h]')
ax[4].set_xlabel('time [h]')
```

Some “cosmetic” modifications are easily achieved with the structure `pred_lines` and `result_lines`.

```python
# Update properties for all prediction lines:
for line_i in mpc_graphics.pred_lines.full:
    line_i.set_linewidth(2)
# Highlight nominal case:
for line_i in np.sum(mpc_graphics.pred_lines['_x', :, :,0]):
    line_i.set_linewidth(5)
for line_i in np.sum(mpc_graphics.pred_lines['_u', :, :,0]):
    line_i.set_linewidth(5)
for line_i in np.sum(mpc_graphics.pred_lines['_aux', :, :,0]):
    line_i.set_linewidth(5)

# Add labels
label_lines = mpc_graphics.result_lines['_x', 'C_a']+mpc_graphics.result_lines['_x', 'C_b']
ax[0].legend(label_lines, ['C_a', 'C_b'])
label_lines = mpc_graphics.result_lines['_x', 'T_R']+mpc_graphics.result_lines['_x', 'T_K']
ax[1].legend(label_lines, ['T_R', 'T_K'])
fig.align_ylabels()
```

After importing the necessary package:

```python
from matplotlib.animation import FuncAnimation, ImageMagickWriter
```

We obtain the animation with:

```python
```
5.15 Industrial polymerization reactor

In this Jupyter Notebook we illustrate the example `industrial_poly`.

Open an interactive online Jupyter Notebook with this content on Binder:

The example consists of the three modules `template_model.py`, which describes the system model, `template_mpc.py`, which defines the settings for the control and `template_simulator.py`, which sets the parameters for the simulator. The modules are used in `main.py` for the closed-loop execution of the controller.

In the following the different parts are presented. But first, we start by importing basic modules and `do-mpc`.

```python
import numpy as np
import matplotlib.pyplot as plt
import sys
from casadi import *

# Add do_mpc to path. This is not necessary if it was installed via pip
sys.path.append('..//..//..//')

# Import do_mpc package:
import do_mpc
```

5.15.1 Model

In the following we will present the configuration, setup and connection between these blocks, starting with the model. The considered model of the industrial reactor is continuous and has 10 states and 3 control inputs. The model is initiated by:

```python
model_type = 'continuous' # either 'discrete' or 'continuous'
model = do_mpc.model.Model(model_type)
```
5.15.1.1 System description

The system consists of a reactor into which nonomer is fed. The monomer turns into a polymer via a very exothermic chemical reaction. The reactor is equipped with a jacket and with an External Heat Exchanger (EHE) that can both be used to control the temperature inside the reactor. A schematic representation of the system is presented below:

The process is modeled by a set of 8 ordinary differential equations (ODEs):

\[
\begin{align*}
\dot{m}_W &= \dot{m}_F \omega_{WF} \\
\dot{m}_A &= \dot{m}_F \omega_{AF} - k_{R1} m_{A,R} - k_{R2} m_{AWT} m_A/m_{ges}, \\
\dot{m}_p &= k_{R1} m_{A,R} + p_1 k_{R2} m_{AWT} m_A/m_{ges}, \\
\dot{T}_R &= 1/(c_{p,R} m_{ges}) \left[ \dot{m}_F c_{p,F} (T_F - T_R) + \Delta H_R k_{R1} m_{A,R} - k_K A (T_R - T_S) \right] \\
&\quad - \dot{m}_{AWT} c_{p,R} (T_R - T_{EK}), \\
\dot{T}_S &= 1/(c_{p,S} m_S) \left[ k_K A (T_R - T_S) - k_K A (T_S - T_M) \right], \\
\dot{T}_M &= 1/(c_{p,WM} m_{MW}) \left[ \dot{m}_{MM} c_{p,W} (T_{IN}^M - T_M) \\
&\quad + k_K A (T_S - T_M) \right] + k_K A (T_S - T_M), \\
\dot{T}_{EK} &= 1/(c_{p,R} m_{AWT}) \left[ \dot{m}_{AWT} c_{p,W} (T_R - T_{EK}) - \alpha (T_{EK} - T_{AWT}) \right] \\
&\quad + k_{R2} m_A m_{AWT} \Delta H_R / m_{ges}, \\
\dot{T}_{AWT} &= \dot{m}_{AWT} c_{p,W} (T_{IN}^M - T_{AWT}) - \alpha (T_{AWT} - T_{EK}) / (c_{p,W} m_{AWT}),
\end{align*}
\]

where

\[
\begin{align*}
U &= m_p/(m_A + m_p), \\
m_{ges} &= m_W + m_A + m_p, \\
k_{R1} &= k_U \frac{m_{R1}}{m_{AWT}} e^{E_R / RT} (k_{U1} (1 - U) + k_{U2} U), \\
k_{R2} &= k_U \frac{m_{R2}}{m_{AWT}} e^{E_R / RT} (k_{U1} (1 - U) + k_{U2} U), \\
k_K &= (m_W k_W + m_A k_{AS} + m_p k_p) / m_{ges}, \\
m_{A,R} &= m_A - m_A m_{AWT} / m_{ges}.
\end{align*}
\]

The model includes mass balances for the water, monomer and product hold-ups \((m_W, m_A, m_p)\) and energy balances for the reactor \((T_R)\), the vessel \((T_S)\), the jacket \((T_M)\), the mixture in the external heat exchanger \((T_{EK})\) and the coolant leaving the external heat exchanger \((T_{AWT})\). The variable \(U\) denotes the polymer-monomer ratio in the reactor, \(m_{ges}\) represents the total mass, \(k_{R1}\) is the reaction rate inside the reactor and \(k_{R2}\) is the reaction rate in the external heat exchanger. The total heat transfer coefficient of the mixture inside the reactor is denoted as \(k_K\) and \(m_{A,R}\) represents the current amount of monomer inside the reactor.

The available control inputs are the feed flow \(\dot{m}_F\), the coolant temperature at the inlet of the jacket \(T_{IN}^M\) and the coolant temperature at the inlet of the external heat exchanger \(T_{IN}^{AWT}\).

An overview of the parameters are listed below:
<table>
<thead>
<tr>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R$</td>
</tr>
<tr>
<td>$C_{p,W}$</td>
</tr>
<tr>
<td>$c_{p,S}$</td>
</tr>
<tr>
<td>$C_{p,F}$</td>
</tr>
<tr>
<td>$c_{p,R}$</td>
</tr>
<tr>
<td>$k_{WS}$</td>
</tr>
<tr>
<td>$T_F$</td>
</tr>
<tr>
<td>$A$</td>
</tr>
<tr>
<td>$m_{M,KW}$</td>
</tr>
<tr>
<td>$m_S$</td>
</tr>
<tr>
<td>$m_{AWT}$</td>
</tr>
<tr>
<td>$m_{AWT,KW}$</td>
</tr>
<tr>
<td>$\dot{m}_{M,KW}$</td>
</tr>
<tr>
<td>$\dot{m}_{AWT,KW}$</td>
</tr>
<tr>
<td>$\dot{m}_{AWT}$</td>
</tr>
<tr>
<td>$E_a$</td>
</tr>
<tr>
<td>$\Delta H_R$</td>
</tr>
<tr>
<td>$k$</td>
</tr>
</tbody>
</table>
5.15.1.2 Implementation

First, we set the certain parameters:

```plaintext
# Certain parameters
R = 8.314  # gas constant
T_F = 25 + 273.15  # feed temperature
E_a = 8500.0  # activation energy
delH_R = 950.0*1.00  # sp reaction enthalpy
A_tank = 65.0  # area heat exchanger surface jacket 65
k_0 = 7.0*1.00  # sp reaction rate
k_U2 = 32.0  # reaction parameter 1
k_U1 = 4.0  # reaction parameter 2
w_WF = .333  # mass fraction water in feed
w_AF = .667  # mass fraction of A in feed
m_M_KW = 5000.0  # mass of coolant in jacket
fm_M_KW = 300000.0  # coolant flow in jacket 300000;
m_AWT_KW = 1000.0  # mass of coolant in EHE
fm_AWT_KW = 100000.0  # coolant flow in EHE
m_AWT = 200.0  # mass of product in EHE
fm_AWT = 200000.0  # product flow in EHE
m_S = 39000.0  # mass of reactor steel
c_pW = 4.2  # sp heat cap coolant
c_pS = .47  # sp heat cap steel
c_pF = 3.0  # sp heat cap feed
c_pR = 5.0  # sp heat cap reactor contents
k_WS = 17280.0  # heat transfer coeff water-steel
k_AS = 3600.0  # heat transfer coeff monomer-steel
k_PS = 360.0  # heat transfer coeff product-steel
alfa = 5*20e4*3.6
p_1 = 1.0
```

and afterwards the uncertain parameters:

```plaintext
# Uncertain parameters:
delH_R = model.set_variable('_p', 'delH_R')
k_0 = model.set_variable('_p', 'k_0')
```

The 10 states of the control problem stem from the 8 ODEs, `accum_monom` models the amount that has been fed to the reactor via \( \dot{m}_{ac}^{\text{acc}} = \dot{m}_{F} \) and \( \dot{T}_{\text{adiab}} (\dot{T}_{\text{adiab}} = \frac{\Delta H_k}{c_{pR} m_{A}} + T_R, \text{ hence } \dot{T}_{\text{adiab}} = \frac{\Delta H_k}{m_{ac}c_{pR}} m_A - \dot{m}_W - \dot{m}_A - \dot{m}_P ) \) is a virtual variable that is important for safety aspects, as we will explain later. All states are created in `do-mpc` via:

```plaintext
# States struct (optimization variables):
m_W = model.set_variable('_x', 'm_W')
m_A = model.set_variable('_x', 'm_A')
m_P = model.set_variable('_x', 'm_P')
T_R = model.set_variable('_x', 'T_R')
T_S = model.set_variable('_x', 'T_S')
T_EK = model.set_variable('_x', 'T_EK')
```

(continues on next page)
and the control inputs via:

\[ \text{[6]: \# Input struct (optimization variables):} \]

- \[ m_{\text{dot}}_f = \text{model.set_variable('}_u\text{', 'm_{\text{dot}}_f')} \]
- \[ T_{\text{in}}_M = \text{model.set_variable('}_u\text{', 'T_{\text{in}}_M')} \]
- \[ T_{\text{in}}_E K = \text{model.set_variable('}_u\text{', 'T_{\text{in}}_E K')} \]

Before defining the ODE for each state variable, we create auxiliary terms:

\[ \text{[7]: \# algebraic equations} \]

\[
\begin{align*}
U_m &= \frac{m_P}{m_A + m_P} \\
m_{\text{ges}} &= m_W + m_A + m_P \\
k_{R1} &= k_0 \exp\left(- \frac{E_a}{(R \cdot T_R)}\right) \cdot ((k_{U1} \cdot (1 - U_m)) + (k_{U2} \cdot U_m)) \\
k_{R2} &= k_0 \exp\left(- \frac{E_a}{(R \cdot T_{E K})}\right) \cdot ((k_{U1} \cdot (1 - U_m)) + (k_{U2} \cdot U_m)) \\
k_K &= ((m_W / m_{\text{ges}}) \cdot k_{WS}) + ((m_A/m_{\text{ges}}) \cdot k_{AS}) + ((m_P/m_{\text{ges}}) \cdot k_{PS})
\end{align*}
\]

The auxiliary terms are used for the more readable definition of the ODEs:

\[ \text{[8]: \# Differential equations} \]

\[
\begin{align*}
\text{dot}_m_W &= m_{\text{dot}}_f \cdot w_{WF} \\
\text{model.set_rhs('}_m\text{', \text{dot}_m_W')} \\
\text{dot}_m_A &= (m_{\text{dot}}_f \cdot w_{AF}) - (k_{R1} \cdot (m_A-((m_A*m_{AWT})/(m_W+m_A+m_P)))) - (p_1 \cdot k_{R2} \cdot (m_{A}/m_{\text{ges}}) \cdot m_{AWT}) \\
\text{model.set_rhs('}_m\text{', \text{dot}_m_A')} \\
\text{dot}_m_P &= (k_{R1} \cdot (m_A-((m_A*m_{AWT})/(m_W+m_A+m_P)))) + (p_1 \cdot k_{R2} \cdot (m_{A}/m_{\text{ges}}) \cdot m_{AWT}) \\
\text{model.set_rhs('}_m\text{', \text{dot}_m_P')} \\
\text{dot}_T_R &= \frac{1}{c_pR \cdot m_{\text{ges}}} \cdot ((m_{\text{dot}}_f \cdot c_{pF} \cdot (T_F - T_R)) - (k_K \cdot A_{tank} \cdot (T_R - T_S)) - (f_{m_AWT} \cdot c_{pR} \cdot (T_R - T_{E K})) + (\text{delH}_R \cdot k_{R1} \cdot (m_A-((m_A*m_{AWT})/(m_W+m_{AWT})))) \\
\text{model.set_rhs('}_T\text{', \text{dot}_T_R')} \\
\text{dot}_T_S &= \frac{1}{c_pS \cdot m_S} \cdot (((k_K \cdot A_{tank} \cdot (T_R - T_S)) - (k_K \cdot A_{tank} \cdot (T_S - T_S))) - (k_K \cdot A_{tank} \cdot (T_S - T_{M})) \\
\text{model.set_rhs('}_T\text{', \text{dot}_T_S')} \\
\text{dot}_T_{A W T} &= (f_{m_AWT} \cdot c_{pS} \cdot (T_{in} - T_{A W T})) + ((f_{m_{MKW}} \cdot c_{pW} \cdot (T_{in} - T_{M})) + (k_K \cdot A_{tank} \cdot (T_S - T_{M}))) \\
\text{model.set_rhs('}_T\text{', \text{dot}_T_{A W T}')} \\
\text{dot}_T_{E K} &= (f_{m_AWT} \cdot c_{pR} \cdot (T_{in} - T_{E K})) - (f_{m_{E K}} \cdot c_{pR} \cdot (T_{in} - T_{E K})) - (\text{alfa} \cdot (T_{E K} - T_{A W T})) + ((f_{m_{MKW}} \cdot c_{pW} \cdot (T_{in} - T_{M})) + (k_K \cdot A_{tank} \cdot (T_S - T_{M}))) \\
\text{model.set_rhs('}_T\text{', \text{dot}_T_{E K}')} \\
\text{dot}_m_{AWT} &= (f_{m_{AWT}} \cdot c_{pW} \cdot (T_{in} - T_{AWT})) - (f_{m_{E K}} \cdot c_{pR} \cdot (T_{in} - T_{E K})) - (\text{alfa} \cdot (T_{A W T} - T_{E K})) + (\text{alfa} \cdot (T_{A W T} - T_{E K}))) \\
\text{model.set_rhs('}_m\text{', \text{dot}_m_{AWT}')} \\
\text{dot}_m_{adiab} &= \text{delH}_R \cdot (m_{\text{ges}} \cdot c_{pR} \cdot (m_{A} - (m_{A} + m_{\text{adiab}}) + m_{\text{adiab}}) \cdot \text{delH}_R \cdot m_{\text{adiab}}) + \text{dot}_T_R \\
\text{model.set_rhs('}_T\text{', \text{dot}_m_{adiab})}
\end{align*}
\]

Finally, the model setup is completed:

\[ \text{[9]: \# Build the model} \]

\[ \text{model.setup()} \]
5.15.2 Controller

Next, the model predictive controller is configured (in template_mpc.py). First, one member of the mpc class is generated with the prediction model defined above:

```python
[10]: mpc = do_mpc.controller.MPC(model)
```

Real processes are also subject to important safety constraints that are incorporated to account for possible failures of the equipment. In this case, the maximum temperature that the reactor would reach in the case of a cooling failure is constrained to be below 109°C. The temperature that the reactor would achieve in the case of a complete cooling failure is $T_{\text{adiab}}$, hence it needs to stay beneath 109°C.

We choose the prediction horizon $n_{\text{horizon}}$, set the robust horizon $n_{\text{robust}}$ to 1. The time step $t_{\text{step}}$ is set to one second and parameters of the applied discretization scheme orthogonal collocation are as seen below:

```python
[11]: setup_mpc = {
    'n_horizon': 20,
    'n_robust': 1,
    'open_loop': 0,
    't_step': 50.0/3600.0,
    'state_discretization': 'collocation',
    'collocation_type': 'radau',
    'collocation_deg': 2,
    'collocation_ni': 2,
    'store_full_solution': True,
    # Use MA27 linear solver in ipopt for faster calculations:
    # 'nlpsol_opts': {'ipopt.linear_solver': 'MA27'}
}
```

```python
mpc.set_param(**setup_mpc)
```

5.15.2.1 Objective

The goal of the economic NMPC controller is to produce 20680 kg of $m_P$ as fast as possible. Additionally, we add a penalty on input changes for all three control inputs, to obtain a smooth control performance.

```python
[12]: _x = model.x
mterm = - _x['m_P'] # terminal cost
lterm = - _x['m_P'] # stage cost
mpc.set_objective(mterm=mterm, lterm=lterm)
mpc.set_rterm(m_dot_f=0.002, T_in_M=0.004, T_in_EK=0.002) # penalty on control input changes
```

5.15.2.2 Constraints

The temperature at which the polymerization reaction takes place strongly influences the properties of the resulting polymer. For this reason, the temperature of the reactor should be maintained in a range of $\pm 2.0^\circ$C around the desired reaction temperature $T_{\text{set}} = 90^\circ$C in order to ensure that the produced polymer has the required properties.

The initial conditions and the bounds for all states are summarized in:
<table>
<thead>
<tr>
<th>State</th>
<th>Init. con</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m_W$</td>
<td>10,000</td>
</tr>
<tr>
<td>$m_A$</td>
<td>853</td>
</tr>
<tr>
<td>$m_P$</td>
<td>26.5</td>
</tr>
<tr>
<td>$T_R$</td>
<td>90.0</td>
</tr>
<tr>
<td>$T_S$</td>
<td>90.0</td>
</tr>
<tr>
<td>$T_M$</td>
<td>90.0</td>
</tr>
<tr>
<td>$T_{EK}$</td>
<td>35.0</td>
</tr>
<tr>
<td>$T_{AWT}$</td>
<td>35.0</td>
</tr>
<tr>
<td>$T_{adiab}$</td>
<td>104.897</td>
</tr>
<tr>
<td>$m_{acc}$</td>
<td>0</td>
</tr>
</tbody>
</table>
and set via:

```python
# auxiliary term
temp_range = 2.0

# lower bound states
mpc.bounds['lower', '_x', 'm_W'] = 0.0
mpc.bounds['lower', '_x', 'm_A'] = 0.0
mpc.bounds['lower', '_x', 'm_P'] = 26.0
mpc.bounds['lower', '_x', 'T_R'] = 363.15 - temp_range
mpc.bounds['lower', '_x', 'T_S'] = 298.0
mpc.bounds['lower', '_x', 'Tout_M'] = 298.0
mpc.bounds['lower', '_x', 'T_EK'] = 288.0
mpc.bounds['lower', '_x', 'Tout_AWT'] = 288.0
mpc.bounds['lower', '_x', 'accum_monom'] = 0.0

# upper bound states
mpc.bounds['upper', '_x', 'T_S'] = 400.0
mpc.bounds['upper', '_x', 'Tout_M'] = 400.0
mpc.bounds['upper', '_x', 'T_EK'] = 400.0
mpc.bounds['upper', '_x', 'Tout_AWT'] = 400.0
mpc.bounds['upper', '_x', 'accum_monom'] = 30000.0
mpc.bounds['upper', '_x', 'T_adiab'] = 382.15
```

The upper bound of the reactor temperature is set via a soft-constraint:

```python
mpc.set_nl_cons('T_R_UB', _x['T_R'], ub=363.15+temp_range, soft_constraint=True, penalty_term_cons=1e4)
```

The bounds of the inputs are summarized below:
Control

\[
\begin{align*}
\dot{m}_F \\
T_{IN}^M \\
T_{IN}^{AWT}
\end{align*}
\]

and set via:

```python
# lower bound inputs
mpc.bounds['lower', '_u', 'm_dot_f'] = 0.0
mpc.bounds['lower', '_u', 'T_in_M'] = 333.15
mpc.bounds['lower', '_u', 'T_in_EK'] = 333.15

# upper bound inputs
mpc.bounds['upper', '_u', 'm_dot_f'] = 3.0e4
mpc.bounds['upper', '_u', 'T_in_M'] = 373.15
mpc.bounds['upper', '_u', 'T_in_EK'] = 373.15
```

5.15.2.3 Scaling

Because the magnitudes of the states and inputs are very different, the performance of the optimizer can be enhanced by properly scaling the states and inputs:

```python
# states
mpc.scaling['_x', 'm_W'] = 10
mpc.scaling['_x', 'm_A'] = 10
mpc.scaling['_x', 'm_P'] = 10
mpc.scaling['_x', 'accum_monom'] = 10

# control inputs
mpc.scaling['_u', 'm_dot_f'] = 100
```
5.15.2.4 Uncertain values

In a real system, usually the model parameters cannot be determined exactly, what represents an important source of uncertainty. In this work, we consider that two of the most critical parameters of the model are not precisely known and vary with respect to their nominal value. In particular, we assume that the specific reaction enthalpy $\Delta H_R$ and the specific reaction rate $k_0$ are constant but uncertain, having values that can vary $\pm 30\%$ with respect to their nominal values.

```python
[16]:
def H_R_var = np.array([950.0, 950.0 * 1.30, 950.0 * 0.70])
k_0_var = np.array([7.0 * 1.00, 7.0 * 1.30, 7.0 * 0.70])
mpc.set_uncertainty_values(delH_R = delH_R_var, k_0 = k_0_var)
```

This means with $n_{\text{robust}}=1$, that 9 different scenarios are considered. The setup of the MPC controller is concluded by:

```python
[17]:
mpc.setup()
```

5.15.3 Estimator

We assume, that all states can be directly measured (state-feedback):

```python
[18]:
estimator = do_mpc.estimator.StateFeedback(model)
```

5.15.4 Simulator

To create a simulator in order to run the MPC in a closed-loop, we create an instance of the do-mpc simulator which is based on the same model:

```python
[19]:
simulator = do_mpc.simulator.Simulator(model)
```

For the simulation, we use the same time step $t_{\text{step}}$ as for the optimizer:

```python
[20]:
params_simulator = {
    'integration_tool': 'cvodes',
    'abstol': 1e-10,
    'rehtol': 1e-10,
    't_step': 50.0/3600.0
}
simulator.set_param(**params_simulator)
```

5.15.4.1 Realizations of uncertain parameters

For the simulation, it is necessary to define the numerical realizations of the uncertain parameters in $p_{\text{num}}$. First, we get the structure of the uncertain parameters:

```python
[21]:
p_num = simulator.get_p_template()
tvp_num = simulator.get_tvp_template()
```

We define a function which is called in each simulation step, which returns the current realizations of the parameters with respect to defined inputs (in this case $t_{\text{now}}$):

5.15. Industrial polymerization reactor

203
# uncertain parameters

```python
p_num['delH_R'] = 950 * np.random.uniform(0.75, 1.25)
p_num['k_0'] = 7 * np.random.uniform(0.75, 1.25)

def p_fun(t_now):
    return p_num

simulator.set_p_fun(p_fun)
```

By defining `p_fun` as above, the function will return a constant value for both uncertain parameters within a range of \(\pm 25\%\) of the nominal value. To finish the configuration of the simulator, call:

```python
simulator.setup()
```

## 5.15.5 Closed-loop simulation

For the simulation of the MPC configured for the CSTR, we inspect the file `main.py`. We define the initial state of the system and set it for all parts of the closed-loop configuration:

```python
# Set the initial state of the controller and simulator:
# assume nominal values of uncertain parameters as initial guess
delH_R_real = 950.0
c_pR = 5.0

# x0 is a property of the simulator - we obtain it and set values.
x0 = simulator.x0

x0['m_W'] = 10000.0
x0['m_A'] = 853.0
x0['m_P'] = 26.5

x0['T_R'] = 90.0 + 273.15
x0['T_S'] = 90.0 + 273.15
x0['Tout_M'] = 90.0 + 273.15
x0['T_EK'] = 35.0 + 273.15
x0['Tout_AWT'] = 35.0 + 273.15
x0['accum_monom'] = 300.0
x0['Tadiab'] = x0['m_A'] * delH_R_real / ((x0['m_W'] + x0['m_A'] + x0['m_P']) * c_pR) + x0['T_R']

mpc.x0 = x0
simulator.x0 = x0
estimator.x0 = x0

mpc.set_initial_guess()
```

Now, we simulate the closed-loop for 100 steps (and suppress the output of the cell with the magic command `%%capture`):

```python
for k in range(100):
    u0 = mpc.make_step(x0)
    y_next = simulator.make_step(u0)
    x0 = estimator.make_step(y_next)
```
5.15.6 Animating the results

To animate the results, we first configure the do-mpc graphics object, which is initiated with the respective data object:

```
mpc_graphics = do_mpc.graphics.Graphics(mpc.data)
```

We quickly configure Matplotlib.

```
from matplotlib import rcParams
rcParams['axes.grid'] = True
rcParams['font.size'] = 18
```

We then create a figure, configure which lines to plot on which axis and add labels.

```
%%capture
fig, ax = plt.subplots(5, sharex=True, figsize=(16,12))
plt.ion()
# Configure plot:
mpc_graphics.add_line(var_type='_x', var_name='T_R', axis=ax[0])
mpc_graphics.add_line(var_type='_x', var_name='accum_monom', axis=ax[1])
mpc_graphics.add_line(var_type='_u', var_name='m_dot_f', axis=ax[2])
mpc_graphics.add_line(var_type='_u', var_name='T_in_M', axis=ax[3])
mpc_graphics.add_line(var_type='_u', var_name='T_in_EK', axis=ax[4])
ax[0].set_ylabel('T_R [K]')
ax[1].set_ylabel('acc. monom')
ax[2].set_ylabel('m_dot_f')
ax[3].set_ylabel('T_in_M [K]')
ax[4].set_ylabel('T_in_EK [K]')
ax[4].set_xlabel('time')
fig.align_ylabels()
```

After importing the necessary package:

```
from matplotlib.animation import FuncAnimation, ImageMagickWriter
```

We obtain the animation with:

```
def update(t_ind):
    print('Writing frame: \'.format(t_ind), end='\r')
    mpc_graphics.plot_results(t_ind=t_ind)
    mpc_graphics.plot_predictions(t_ind=t_ind)
    mpc_graphics.reset_axes()
    lines = mpc_graphics.result_lines.full
    return lines
```

```
n_steps = mpc.data['_time'].shape[0]
anim = FuncAnimation(fig, update, frames=n_steps, blit=True)
```

Writing frame: 99.
We are displaying recorded values as solid lines and predicted trajectories as dashed lines. Multiple dashed lines exist for different realizations of the uncertain scenarios.

The most interesting behavior here can be seen in the state $T_R$, which has the upper bound:

```python
[38]: mpc.bounds['upper', '_x', 'T_R']
[38]: DM(375.15)
```

Due to robust control, we are approaching this value but hold a certain distance as some possible trajectories predict a temperature increase. As the reaction finishes we can safely increase the temperature because a rapid temperature change due to uncertainty is impossible.

## 5.16 Oscillating masses

In this Jupyter Notebook we illustrate the example `oscillating_masses_discrete`.

Open an interactive online Jupyter Notebook with this content on Binder:

The example consists of the three modules `template_model.py`, which describes the system model, `template_mpc.py`, which defines the settings for the control and `template_simulator.py`, which sets the parameters for the simulator. The modules are used in `main.py` for the closed-loop execution of the controller. One exemplary result will be presented at the end of this tutorial as a gif.

In the following the different parts are presented. But first, we start by importing basic modules and `do-mpc`.

```python
[1]: import numpy as np
    import sys
    from casadi import *

    # Add do_mpc to path. This is not necessary if it was installed via pip
    sys.path.append('..//..//..//')

    # Import do_mpc package:
    import do_mpc
```

### 5.16.1 Model

In the following we will present the configuration, setup and connection between these blocks, starting with the model. The considered model are two horizontally oscillating masses interconnected via a spring where each one is connected via a spring to a wall, as shown below:
The states of each mass are its position \( s_i \) and velocity \( v_i \), \( i = 1, 2 \). A force \( u_1 \) can be applied to the right mass. The via first-order hold and a sampling time of 0.5 seconds discretized model \( x_{k+1} = Ax_k + Bu_k \) is given by:

\[
A = \begin{bmatrix}
0.763 & 0.460 & 0.115 & 0.020 \\
0.899 & 0.763 & 0.420 & 0.115 \\
0.115 & 0.020 & 0.763 & 0.460 \\
0.420 & 0.115 & 0.899 & 0.763 \\
\end{bmatrix}, \quad B = \begin{bmatrix}
0.014 \\
0.063 \\
0.221 \\
0.367 \\
\end{bmatrix},
\]

where \( x = [s_1, v_1, s_2, v_2]^T \) and \( u = [u_1] \).

The discrete model is initiated by:

```
[2]: model_type = 'discrete' # either 'discrete' or 'continuous'
model = do_mpc.model.Model(model_type)
```

### 5.16.1.1 States and control inputs

The states and the inputs are directly created as vectors:

```
[3]: _x = model.set_variable(var_type='_x', var_name='x', shape=(4,1))
_u = model.set_variable(var_type='_u', var_name='u', shape=(1,1))
```

Afterwards the discrete-time LTI model is added:

```
[4]: A = np.array([[0.763, 0.460, 0.115, 0.020],
                  [-0.899, 0.763, 0.420, 0.115],
                  [0.115, 0.020, 0.763, 0.460],
                  [0.420, 0.115, -0.899, 0.763]])
B = np.array([[0.014],
              [0.063],
              [0.221],
              [0.367]])

x_next = A@_x + B@_u
model.set_rhs('x', x_next)
```

Additionally, we will define an expression, which represents the stage and terminal cost of our control problem. This term will be later used as the cost in the MPC formulation and can be used to directly plot the trajectory of the cost of each state.

```
[5]: model.set_expression(expr_name='cost', expr=sum1(_x**2))
[5]: SX((((sq(x_0)+sq(x_1))+sq(x_2))+sq(x_3)))
```

The model setup is completed via:

```
[6]: # Build the model
model.setup()
```

### 5.16.2 Controller

Next, the model predictive controller is configured. First, one member of the mpc class is generated with the prediction model defined above:
We choose the prediction horizon \( n_{\text{horizon}} \) to 7 and set the robust horizon \( n_{\text{robust}} \) to zero, because no uncertainties are present. The time step \( t_{\text{step}} \) is set to 0.5 seconds (like the discretization time step)). There is no need to apply a discretization scheme, because the system is discrete:

```python
setup_mpc = {
    'n_robust': 0,
    'n_horizon': 7,
    't_step': 0.5,
    'state_discretization': 'discrete',
    'store_full_solution': True,
    # Use MA27 linear solver in ipopt for faster calculations:
    #'nlpsol_opts': {'ipopt.linear_solver': 'MA27'}
}
mpc.set_param(**setup_mpc)
```

### 5.16.2.1 Objective

The goal of the controller is to bring the system to the origin, hence we apply a quadratic cost with weight one to every state and penalty on input changes for a smooth operation. This is here done by using the cost expression defined in the model:

```python
mterm = model.aux['cost'] # terminal cost
lterm = model.aux['cost'] # terminal cost
    # stage cost
mpc.set_objective(mterm=mterm, lterm=lterm)
mpc.set_rterm(u=1e-4) # input penalty
```

### 5.16.2.2 Constraints

In the next step, the constraints of the control problem are set. In this case, there are only upper and lower bounds for each state and the input. The displacement has to fulfill \(-4m \leq s_i \leq 4m\), the velocity \(-10\text{ms}^{-1} \leq v_i \leq 10\text{ms}^{-1}\) and the force cannot exceed \(-0.5N \leq u_i \leq 0.5N\):

```python
max_x = np.array([[-4.0, 10.0, [4.0], [10.0]])
    # lower bounds of the states
mpc.bounds['lower','_x','x'] = -max_x

    # upper bounds of the states
mpc.bounds['upper','_x','x'] = max_x

    # lower bounds of the input
mpc.bounds['lower','_u','u'] = -0.5

    # upper bounds of the input
mpc.bounds['upper','_u','u'] = 0.5
```

The setup of the MPC controller is concluded by:

#### 5.16. Oscillating masses
5.16.3 Estimator

We assume, that all states can be directly measured (state-feedback):

```python
estimator = do_mpc.estimator.StateFeedback(model)
```

5.16.4 Simulator

To create a simulator in order to run the MPC in a closed-loop, we create an instance of the do-mpc simulator which is based on the same model:

```python
simulator = do_mpc.simulator.Simulator(model)
```

Because the model is discrete, we do not need to specify options for the integration necessary for simulating the system. We only set the time step `t_step` which is identical to the one used for the optimization and finish the setup of the simulator:

```python
simulator.set_param(t_step = 0.1)
simulator.setup()
```

5.16.5 Closed-loop simulation

For the simulation of the MPC configured for the oscillating masses, we inspect the file `main.py`. We set the initial state of the system (randomly seeded) and set it for all parts of the closed-loop configuration:

```python
# Seed
np.random.seed(99)

# Initial state
e = np.ones([model.n_x,1])
x0 = np.random.uniform(-3*e,3*e)  # Values between +3 and +3 for all states
mpc.x0 = x0
simulator.x0 = x0
estimator.x0 = x0

# Use initial state to set the initial guess.
mpc.set_initial_guess()
```

Now, we simulate the closed-loop for 50 steps (and suppress the output of the cell with the magic command `%%capture`):

```python
for k in range(50):
    u0 = mpc.make_step(x0)
y_next = simulator.make_step(u0)
x0 = estimator.make_step(y_next)
```
5.16.6 Displaying the results

After some slight configuration of matplotlib:

```
from matplotlib import rcParams
rcParams['axes.grid'] = True
rcParams['font.size'] = 18
```

We use the convenient `default_plot` function of the `graphics` module, to obtain the graphic below.

```
import matplotlib.pyplot as plt
fig, ax, graphics = do_mpc.graphics.default_plot(mpc.data, figsize=(16,9))
graphics.plot_results()
graphics.reset_axes()
plt.show()
```

We can see that the control objective was successfully fulfilled and that bounds, e.g. for the control inputs are satisfied.

5.17 Double inverted pendulum

In this Jupyter Notebook we illustrate the example DIP. This example illustrates how to use DAE models in do-mpc.

Open an interactive online Jupyter Notebook with this content on Binder:

The example consists of the three modules `template_model.py`, which describes the system model, `template_mpc.py`, which defines the settings for the control and `template_simulator.py`, which sets the parameters for the simulator. The modules are used in `main.py` for the closed-loop execution of the controller.

We start by importing basic modules and do-mpc.
import numpy as np
import sys
from casadi import *

# Add do_mpc to path. This is not necessary if it was installed via pip
sys.path.append('..//..//..//')

# Import do_mpc package:
import do_mpc

5.17.1 Model

In the following we will present the configuration, setup and connection between these blocks, starting with the model.

In this example we consider the double pendulum on a card as depicted below:

The system is described in terms of its horizontal position $x$ and the two angles $\theta$, where $\theta_1 = \theta_2 = 0$ denotes the upright position.

We will formulate a continuous dynamic model for this system and start by initiating a do_mpc Model instance:

```python
model_type = 'continuous' # either 'discrete' or 'continuous'
model = do_mpc.model.Model(model_type)
```

5.17.1.1 Parameters

The model is configured with the following (certain) parameters:

```python
m0 = 0.6  # kg, mass of the cart
m1 = 0.2  # kg, mass of the first rod
m2 = 0.2  # kg, mass of the second rod
L1 = 0.5  # m, length of the first rod
L2 = 0.5  # m, length of the second rod

G = 9.80665  # m/s^2, Gravity
```

We furthermore introduce the following derived parameters to conveniently formulate the model equations below.

```python
l1 = L1/2  # m,
l2 = L2/2  # m,
J1 = (m1 * l1**2) / 3  # Inertia
J2 = (m2 * l2**2) / 3  # Inertia

h1 = m0 + m1 + m2
h2 = m1*l1 + m2*L1
h3 = m2*l2
h4 = m1*l1**2 + m2*L1**2 + J1
h5 = m2*l2*L1
h6 = m2*l2*l2 + J2
h7 = (m1*l1 + m2*L1) * g
h8 = m2*l2*g
```
5.17.1.2 Euler-Lagrangian equations

The dynamics of the double pendulum can be derived from the Euler-Lagrangian equations, which yield:

\[
\begin{align*}
\ddot{x} + h_2 \dot{\theta}_1 \cos(\theta_1) + h_3 \dot{\theta}_2 \cos(\theta_2) &= (h_2 \dot{\theta}_1^2 \sin(\theta_1) + h_3 \dot{\theta}_2^2 \sin(\theta_2) + u) \\
\dot{\theta}_1 &= h_2 \cos(\theta_1) \ddot{x} + h_4 \dot{\theta}_1 + h_5 \cos(\theta_1 - \theta_2) \dot{\theta}_2 = (h_7 \sin(\theta_1) - h_5 \dot{\theta}_2^2 \sin(\theta_1 - \theta_2)) \\
\dot{\theta}_2 &= h_3 \cos(\theta_2) \ddot{x} + h_5 \cos(\theta_1 - \theta_2) \dot{\theta}_1 + h_6 \dot{\theta}_2 = (h_5 \dot{\theta}_1^2 \sin(\theta_1 - \theta_2) + h_8 \sin(\theta_2))
\end{align*}
\]

we introduce the states

\[x = [x, \theta_1, \theta_2, \dot{x}, \dot{\theta}_1, \dot{\theta}_2]^T\]

and input:

\[u = f\]

which is the horizontal force applied to the cart.

At this point we have two options. The typical approach would be to rewrite the system as:

\[M(x) \ddot{x} = A(x)x + Bu\]

where it can be shown that

\[\det(M) > 0, \forall x\]

such that we can obtain the ODE:

\[\dot{x} = M(x)^{-1}(A(x)x + Bu)\]

**do-mpc** fully supports this option but it requires some nasty reformulations of the above equations and yields a very complex expression for the ODE.

Instead we suggest ...

5.17.1.3 Differential algebraic equation (DAE)

We introduce new algebraic states

\[z = [\dddot{x}, \dddot{\theta}_1, \dddot{\theta}_2]^T\]

which makes it very convenient to formulate the ODE in terms of \(x, u, z\):

\[\dot{x} = [\dot{x}, \dot{\theta}_1, \dot{\theta}_2]^T\]
The only remaining challenge is to implement the relationship between $x, u$ and $z$, in the form of:

$$g(x, u, z) = 0$$

with do-mpc this is easily achieved:

```python
[8]: euler_lagrange = vertcat(
    # 1
    h1*dpos + h2*ddtheta[0]*cos(theta[0]) - h2*dtheta[0]**2*sin(theta[0]) - h3*ddtheta[1]*cos(theta[0]) + h3*dtheta[1]**2*sin(theta[1]) + u),
    # 2
    h2*cos(theta[0])*dpos + h4*ddtheta[0] + h5*cos(theta[0]-theta[1])*ddtheta[1] - h5*dtheta[1]**2*sin(theta[0]-theta[1]),
    # 3
    h3*cos(theta[1])*dpos + h5*cos(theta[0]-theta[1])*ddtheta[0] + h6*ddtheta[1] - h5*dtheta[0]**2*sin(theta[0]-theta[1]) + h8*sin(theta[1])
)
```

with just a few lines of code we have defined the dynamics of the double pendulum!

### 5.17.1.4 Energy equations

The next step is to introduce new “auxiliary” expressions to do-mpc for the kinetic and potential energy of the system. This is required in this example, because we will need these expressions for the formulation of the MPC controller.

Introducing these expressions has the additional advantage that do-mpc saves and exports the calculated values, which is great for monitoring and debugging.

For the kinetic energy, we have:

$$E_{\text{kin, cart}} = \frac{1}{2} m_0 \dot{x}^2$$

$$E_{\text{kin, p1}} = \frac{1}{2} m_1 ((\ddot{x} + l_1 \dot{\theta}_1 \cos(\theta_1))^2 + (l_1 \dot{\theta}_1 \sin(\theta_1))^2) + \frac{1}{2} J_1 \dot{\theta}_1^2$$

$$E_{\text{kin, p2}} = \frac{1}{2} m_2 ((\ddot{x} + L_1 \dot{\theta}_1 \cos(\theta_1) + l_2 \dot{\theta}_2 \cos(\theta_2))^2$$

$$+ (L_1 \dot{\theta}_1 \sin(\theta_1) + l_2 \dot{\theta}_2 \sin(\theta_2))^2) + \frac{1}{2} J_2 \dot{\theta}_2^2$$

and for the potential energy:

$$E_{\text{pot.}} = m_1 g l_1 \cos(\theta_1) + m_2 g (L_1 \cos(\theta_1) + l_2 \cos(\theta_2))$$

It only remains to formulate the expressions and set them to the model:

```python
[9]: E_kin_cart = 1 / 2 * m0 * dpos**2
E_kin_p1 = 1 / 2 * m1 * (
    (dpos + l1 * dtheta[0] * cos(theta[0]))**2 +
    (l1 * dtheta[0] * sin(theta[0]))**2) + 1 / 2 * J1 * dtheta[0]**2
```
\[
\begin{align*}
E_{\text{kin\_p2}} &= \frac{1}{2} m_2 \left( (d\text{pos} + L_1 \cdot d\theta[0] \cdot \cos(\theta[0]) + l_2 \cdot d\theta[1] \cdot \cos(\theta[1]))^2 + (L_1 \cdot d\theta[0] \cdot \sin(\theta[0]) + l_2 \cdot d\theta[1] \cdot \sin(\theta[1]))^2 + 1 / 2 \cdot J_2 \cdot d\theta[0]^2 \right) \\
E_{\text{kin}} &= E_{\text{kin\_cart}} + E_{\text{kin\_p1}} + E_{\text{kin\_p2}} \\
E_{\text{pot}} &= m_1 \cdot g \cdot l_1 \cdot \cos(\theta[0]) + m_2 \cdot g \cdot (L_1 \cdot \cos(\theta[0]) + l_2 \cdot \cos(\theta[1]))
\end{align*}
\]

Finally, the model setup is completed:

```python
model.set_expression('E_kin', E_kin)
model.set_expression('E_pot', E_pot)
```

\[(9)\] \(\text{SX}(((0.490333\cdot\cos(\theta_0))+(1.96133*((0.5\cdot\cos(\theta_0))+(0.25\cdot\cos(\theta_1)))))\)

5.17.2 Controller

Next, the controller is configured. First, an instance of the MPC class is generated with the prediction model defined above:

```python
mpc = do_mpc.controller.MPC(model)
```

We choose the prediction horizon \(n_{\text{horizon}}=100\), set the robust horizon \(n_{\text{robust}} = 0\). The time step \(t_{\text{step}}\) is set to 0.04s and parameters of the applied discretization scheme orthogonal collocation are as seen below:

```python
setup_mpc = {
    'n_horizon': 100,
    'n_robust': 0,
    'open_loop': 0,
    't_step': 0.04,
    'state_discretization': 'collocation',
    'collocation_type': 'radau',
    'collocation_deg': 3,
    'collocation_ni': 1,
    'store_full_solution': True,
    # Use MA27 linear solver in ipopt for faster calculations:
    'nlpsol_opts': {'ipopt.linear_solver': 'mumps'}
}
mpc.set_param(**setup_mpc)
```

5.17.2.1 Objective

The control objective is to erect the double pendulum and to stabilize it in the up-up position. It is not straight-forward to formulate an objective which yields this result. Classical set-point tracking, e.g. with the set-point:

\[\theta_s = [0, 0, 0]\]
and a quadratic cost:

\[
J = \sum_{k=0}^{N} \left( \theta - \theta_s \right)^2
\]

is **known to work very poorly**. Clearly, the problem results from the fact that \( \theta_s = 2\pi n, \ n \in \mathbb{Z} \) is also a valid solution.

Instead we will use an energy-based formulation for the objective. If we think of energy in terms of potential and kinetic energy it is clear that we want to maximize the potential energy (up-up position) and minimize the kinetic energy (stabilization).

Since we have already introduced the expressions for the potential and kinetic energy in the model, we can now simply reuse these expressions for the formulation of the objective function, as shown below:

```python
mterm = model.aux['E_kin'] - model.aux['E_pot']  # terminal cost
lterm = model.aux['E_kin'] - model.aux['E_pot']  # stage cost

mpc.set_objective(mterm=mterm, lterm=lterm)
# Input force is implicitly restricted through the objective.
mpc.set_rterm(force=0.1)
```

### 5.17.2.2 Constraints

In the next step, the constraints of the control problem are set. In this case, there is only an upper and lower bounds for the input.

```python
mpc.bounds['lower', '_u', 'force'] = -4
mpc.bounds['upper', '_u', 'force'] = 4
```

We can now setup the controller.

```python
mpc.setup()
```

### 5.17.3 Estimator

We assume, that all states can be directly measured (state-feedback):

```python
estimator = do_mpc.estimator.StateFeedback(model)
```

### 5.17.4 Simulator

To create a simulator in order to run the MPC in a closed-loop, we create an instance of the `do_mpc` simulator which is based on the same model:

```python
simulator = do_mpc.simulator.Simulator(model)
```

For the simulation, we use the time step `t_step` as for the optimizer:

```python
params_simulator = {
    # Note: cvode doesn't support DAE systems.
    'integration_tool': 'idas',
    'abstol': 1e-10,
    ...
}
```
For the simulation of the MPC configured for the DIP, we inspect the file `main.py`. We define the initial state of the system and set for all parts of the closed-loop configuration:

```python
simulator.x0['theta'] = 0.99*np.pi
x0 = simulator.x0.cat.full()
mpc.x0 = x0
estimator.x0 = x0
mpc.set_initial_guess()
```

Note that `mpc.set_initial_guess()` is very important in this example. Also note that we didn’t set the initial state at exactly $\pi$ which results in unfavorable numerical issues (it will work however).

## 5.17.5.1 Prepare visualization

For the visualization of the control performance, we first import matplotlib and change some basic settings:

```python
import matplotlib.pyplot as plt
plt.ion()
from matplotlib import rcParams
rcParams['text.usetex'] = False
rcParams['axes.grid'] = True
rcParams['lines.linewidth'] = 2.0
rcParams['axes.labelsize'] = 'xx-large'
rcParams['xtick.labelsize'] = 'xx-large'
rcParams['ytick.labelsize'] = 'xx-large'
```

We use the plotting capabilities, which are included in do-mpc. The `mpc_graphics` contain information like the current estimated state and the predicted trajectory of the states and inputs based on the solution of the control problem. The `sim_graphics` contain the information about the simulated evaluation of the system.

```python
mpc_graphics = do_mpc.graphics.Graphics(mpc.data)
```

For the example of the DIP we create a new function which takes as input the states (at a given time $k$) and returns the x and y positions of the two bars (the arms of the pendulum).

```python
def pendulum_bars(x):
    x = x.flatten()
    # Get the x,y coordinates of the two bars for the given state x.
    line_1_x = np.array([x[0],
```
```python
x[0]+L1*np.sin(x[1])
])

line_1_y = np.array([0,
    L1*np.cos(x[1])
])

line_2_x = np.array([line_1_x[1],
    line_1_x[1] + L2*np.sin(x[2])
])

line_2_y = np.array([line_1_y[1],
    line_1_y[1] + L2*np.cos(x[2])
])

line_1 = np.stack((line_1_x, line_1_y))
line_2 = np.stack((line_2_x, line_2_y))

return line_1, line_2
```

We then setup a graphic:

```python
fig = plt.figure(figsize=(16,9))
ax1 = plt.subplot2grid((4, 2), (0, 0), rowspan=4)
ax2 = plt.subplot2grid((4, 2), (0, 1))
ax3 = plt.subplot2grid((4, 2), (1, 1))
ax4 = plt.subplot2grid((4, 2), (2, 1))
ax5 = plt.subplot2grid((4, 2), (3, 1))

ax2.set_ylabel('$E_{kin}$ [J]')
ax3.set_ylabel('$E_{pot}$ [J]')
ax4.set_ylabel('Angle [rad]')
ax5.set_ylabel('Input force [N]')

# Axis on the right.
for ax in [ax2, ax3, ax4, ax5]:
    ax.yaxis.set_label_position("right")
    ax.yaxis.tick_right()
    if ax != ax5:
        ax.xaxis.set_ticklabels([])

ax5.set_xlabel('time [s]')

mpc_graphics.add_line(var_type='_aux', var_name='E_kin', axis=ax2)
mpc_graphics.add_line(var_type='_aux', var_name='E_pot', axis=ax3)
mpc_graphics.add_line(var_type='_x', var_name='theta', axis=ax4)
mpc_graphics.add_line(var_type='_u', var_name='force', axis=ax5)

ax1.axhline(0,color='black')

bar1 = ax1.plot([],[], '-o', linewidth=5, markersize=10)
```

(continues on next page)
5.17.5.2 Run open-loop

Before we test the closed loop case, let's plot one open-loop prediction to check how the resulting graphic looks like.

```python
bar2 = ax1.plot([],[], '-o', linewidth=5, markersize=10)
ax1.set_xlim(-1.8,1.8)
ax1.set_ylim(-1.2,1.2)
ax1.set_axis_off()
fig.align_ylabels()
fig.tight_layout()
```

This is Ipopt version 3.12.3, running with linear solver mumps.

Number of nonzeros in equality constraint Jacobian...: 19406
Number of nonzeros in inequality constraint Jacobian.: 0
Number of nonzeros in Lagrangian Hessian....: 6814

```
<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>iter</td>
<td>objective</td>
<td>inf_pr</td>
<td>inf_du</td>
<td>lg(mu)</td>
<td></td>
<td></td>
<td>d</td>
</tr>
<tr>
<td>-----</td>
<td>------------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>------</td>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td>0</td>
<td>1.9799658e+02</td>
<td>4.62e-02</td>
<td>1.00e-03</td>
<td>-1.0</td>
<td>0.00e+00</td>
<td>-</td>
<td>0.00e+00</td>
</tr>
<tr>
<td>1</td>
<td>1.9809206e+02</td>
<td>7.33e-05</td>
<td>3.25e-04</td>
<td>-1.0</td>
<td>1.53e+00</td>
<td>0.0</td>
<td>1.00e+00</td>
</tr>
<tr>
<td>2</td>
<td>1.9808344e+02</td>
<td>8.59e-05</td>
<td>1.22e-04</td>
<td>-2.5</td>
<td>4.56e-01</td>
<td>1.2</td>
<td>1.00e+00</td>
</tr>
<tr>
<td>3</td>
<td>1.9753270e+02</td>
<td>2.78e+00</td>
<td>5.70e-01</td>
<td>-3.8</td>
<td>2.32e+01</td>
<td>4.1</td>
<td>6.69e-01</td>
</tr>
<tr>
<td>4</td>
<td>1.9586898e+02</td>
<td>2.82e-01</td>
<td>3.12e-01</td>
<td>-3.8</td>
<td>1.14e-01</td>
<td>3.8</td>
<td>3.78e-01</td>
</tr>
<tr>
<td>5</td>
<td>1.9526975e+02</td>
<td>1.38e+00</td>
<td>1.17e+00</td>
<td>-3.8</td>
<td>5.85e+01</td>
<td>4.1</td>
<td>4.40e-01</td>
</tr>
<tr>
<td>6</td>
<td>1.9266817e+02</td>
<td>2.02e-01</td>
<td>3.44e-01</td>
<td>-3.8</td>
<td>9.02e-02</td>
<td>3.7</td>
<td>1.63e-02</td>
</tr>
<tr>
<td>7</td>
<td>1.9230233e+02</td>
<td>1.47e-01</td>
<td>2.58e-01</td>
<td>-3.8</td>
<td>2.10e-01</td>
<td>4.2</td>
<td>9.32e-01</td>
</tr>
<tr>
<td>8</td>
<td>1.9168748e+02</td>
<td>3.42e-02</td>
<td>2.53e-02</td>
<td>-3.8</td>
<td>6.96e-02</td>
<td>3.7</td>
<td>1.00e+00</td>
</tr>
<tr>
<td>9</td>
<td>1.9143723e+02</td>
<td>3.64e-02</td>
<td>1.08e-01</td>
<td>-3.8</td>
<td>2.48e-01</td>
<td>4.2</td>
<td>1.00e+00</td>
</tr>
</tbody>
</table>
```
| iter | objective | inf_pr | inf_du | lg(mu) | ||d|| | lg(rg) | alpha_du | alpha_pr | ls |
|------|-----------|--------|--------|--------|-------|--------|----------|----------|----|
| 15   | 1.8497700e+02 | 2.04e-01 | 2.37e-01 | -3.8 | 1.23e+01 | -3.5 | 4.49e-01 | 1.00e+00 | h 1 |
| 16   | 1.8432030e+02 | 1.32e+00 | 8.74e-01 | -3.8 | 2.65e+02 | -3.9 | 4.70e-01 | 1.13e-01f | 1 |
| 17   | 1.8419900e+02 | 1.26e+00 | 8.39e-01 | -3.8 | 1.56e+02 | -4.4 | 4.11e-01 | 4.08e-02h | 1 |
| 18   | 1.8356256e+02 | 1.91e-01 | 3.51e-01 | -3.8 | 1.42e+01 | -4.0 | 4.33e-01 | 1.00e+00h | 1 |
| 19   | 1.8341217e+02 | 8.79e-02 | 1.36e-01 | -3.8 | 1.80e+01 | -3.6 | 3.87e-01 | 6.81e-01h | 1 |

(continued from previous page)

| iter | objective | inf_pr | inf_du | lg(mu) | ||d|| | lg(rg) | alpha_du | alpha_pr | ls |
|------|-----------|--------|--------|--------|-------|--------|----------|----------|----|
| 20   | 1.8278812e+02 | 1.76e-01 | 8.50e-02 | -3.8 | 1.52e+01 | -4.0 | 5.64e-01 | 1.00e+00h | 1 |
| 21   | 1.8138293e+02 | 1.20e+00 | 1.92e-01 | -3.8 | 4.27e+01 | -4.5 | 7.99e-01 | 1.00e+00h | 1 |
| 22   | 1.8054638e+02 | 9.70e-01 | 2.77e-01 | -3.8 | 2.47e+02 | -5.0 | 1.00e+00 | 1.90e-01h | 1 |
| 23   | 1.7935640e+02 | 6.65e-01 | 6.44e-01 | -3.8 | 7.49e+01 | -4.6 | 1.00e+00 | 1.00e+00h | 1 |
| 24   | 1.7911838e+02 | 2.49e-02 | 6.02e-02 | -3.8 | 2.17e+01 | -4.1 | 8.37e+00 | 1.00e+00h | 1 |
| 25   | 1.7853437e+02 | 1.68e-01 | 8.35e-02 | -3.8 | 3.72e+01 | -4.6 | 1.00e+00 | 1.00e+00h | 1 |
| 26   | 1.7811649e+02 | 2.22e-01 | 1.28e-01 | -3.8 | 1.79e+02 | -5.1 | 2.68e-01 | 1.72e-01h | 1 |
| 27   | 1.7728888e+02 | 7.04e-01 | 7.44e-01 | -3.8 | 8.19e+01 | -4.7 | 1.00e+00 | 1.00e+00h | 1 |
| 28   | 1.7707269e+02 | 2.61e-02 | 7.21e-02 | -3.8 | 2.92e+01 | -4.2 | 7.54e-01 | 1.00e+00h | 1 |
| 29   | 1.7653474e+02 | 7.15e-02 | 7.34e-02 | -3.8 | 5.09e+01 | -4.7 | 1.00e+00 | 5.45e-01h | 1 |
| 30   | 1.7656004e+02 | 8.96e-02 | 1.00e-01 | -3.8 | 3.03e+02 | -5.2 | 6.98e-02 | 6.52e-02h | 1 |
| 31   | 1.7563497e+02 | 5.41e-01 | 5.17e-01 | -3.8 | 9.26e+01 | -4.8 | 1.00e+00 | 1.00e+00h | 1 |
| 32   | 1.7525047e+02 | 3.61e-01 | 2.32e-01 | -3.8 | 9.44e+01 | -4.3 | 1.00e+00 | 4.12e-01h | 1 |
| 33   | 1.7495933e+02 | 3.41e-02 | 1.10e-01 | -3.8 | 3.39e+01 | -3.9 | 1.00e+00 | 1.00e+00h | 1 |
| 34   | 1.7432319e+02 | 2.26e-02 | 2.67e-01 | -3.8 | 9.19e+01 | -4.4 | 1.00e+00 | 6.21e-01h | 1 |
| 35   | 1.7390163e+02 | 3.10e-02 | 2.47e-01 | -3.8 | 2.92e+01 | -4.0 | 7.87e-01 | 1.00e+00h | 1 |
| 36   | 1.7313617e+02 | 2.11e-01 | 2.71e-01 | -3.8 | 5.51e+01 | -4.4 | 9.59e-01 | 1.00e+00h | 1 |
| 37   | 1.7299030e+02 | 1.06e-01 | 2.56e-01 | -3.8 | 1.38e+01 | -4.9 | 2.75e-01 | 6.47e-02h | 1 |
| 38   | 1.7267489e+02 | 1.08e-01 | 2.52e-01 | -3.8 | 3.03e+01 | -5.4 | 2.44e-01 | 7.04e-02h | 1 |
| 39   | 1.7196110e+02 | 1.73e-01 | 2.89e-01 | -3.8 | 1.32e+01 | -5.0 | 2.15e-01 | 4.92e-01h | 1 |

(continues on next page)
| iter | objective | inf_pr | inf_du | lg(mu) | ||d|| | lg(rg) | alpha_du | alpha_pr | ls |
|------|-----------|-------|--------|--------|------|-------|---------|---------|----|
| 67   | 1.6206843e+02 | 7.95e-01 | 1.90e+00 | -3.8 | 1.86e+02 | -4.8 | 5.50e-01 | 1.00e+00h | 1 |
| 68   | 1.6177105e+02 | 4.34e-01 | 1.27e+00 | -3.8 | 9.70e+01 | -4.4 | 1.00e+00 | 6.91e-01h | 1 |
| 69   | 1.6040557e+02 | 2.79e+00 | 3.27e+00 | -3.8 | 6.57e+02 | -4.8 | 2.32e-01 | 2.08e-01f | 1 |

continues on next page...
| iter | objective | inf.pr | inf.du | lg(mu) | ||d|| | lg(rg) | alpha.du | alpha.pr | ls |
|------|-----------|--------|--------|--------|-------|-------|---------|---------|----|
| 119  | 1.3490818e+02 | 1.40e+00 | 1.44e+00 | -3.8 | 3.72e+01 | -3.9 | 7.39e-01 | 9.14e-02 | 1  |
| 120  | 1.3454765e+02 | 7.88e-01 | 6.30e-01 | -3.8 | 1.07e+02 | -4.4 | 1.00e+00 | 4.45e-01 | 1  |
| 121  | 1.3382891e+02 | 8.02e-01 | 1.05e+00 | -3.8 | 4.07e+02 | -4.8 | 2.62e-02 | 1.37e-01 | 1  |
| 122  | 1.3320144e+02 | 8.37e-01 | 1.06e+00 | -3.8 | 7.34e+03 | -5.3 | 1.93e-02 | 6.24e-03 | 1  |
| 123  | 1.3250550e+02 | 5.02e-01 | 4.48e-01 | -3.8 | 1.34e+02 | -4.9 | 1.00e+00 | 4.90e-01 | 1  |
| 124  | 1.3248374e+02 | 4.59e-01 | 4.09e-01 | -3.8 | 4.23e+01 | -4.5 | 1.00e+00 | 8.59e-02 | 1  |
| 125  | 1.3200480e+02 | 4.21e-01 | 3.63e-01 | -3.8 | 1.87e+02 | -4.9 | 5.69e-01 | 2.05e-01 | 1  |
| 126  | 1.3169215e+02 | 2.48e-01 | 1.67e-01 | -3.8 | 7.07e+01 | -4.5 | 1.00e+00 | 4.82e-01 | 1  |
| 127  | 1.3151459e+02 | 5.34e-02 | 1.28e-01 | -3.8 | 3.19e+01 | -4.1 | 1.00e+00 | 1.00e+00 | 1  |
| 128  | 1.3083326e+02 | 3.20e-01 | 7.78e-01 | -3.8 | 1.21e+02 | -4.6 | 6.13e-01 | 4.90e-01 | 1  |
| 129  | 1.3062369e+02 | 1.98e-01 | 4.53e-01 | -3.8 | 4.02e+01 | -4.1 | 1.00e+00 | 4.12e-01 | 1  |

(continues on next page)
| Iter | Objective | Inf Pr   | Inf Du   | Ln(Mu) | ||d||   | Ln(RG) | Alpha Du | Alpha Pr | LS |
|------|-----------|----------|----------|--------|--------|--------|----------|----------|----|
| 170  | 1.0781742e+02 | 8.09e-01 | 2.02e+00 | -3.8   | 9.29e+01 | -4.7   | 5.53e-01 | 1.00e+00 | 1   |
| 171  | 1.0770287e+02 | 5.68e-01 | 1.42e+00 | -3.8   | 9.29e+01 | -4.3   | 7.05e-01 | 2.98e-01 | 1   |
| 172  | 1.0733575e+02 | 4.23e-01 | 1.02e+00 | -3.8   | 9.29e+01 | -4.8   | 2.96e-01 | 3.11e-01 | 1   |
| 173  | 1.0672927e+02 | 4.68e-01 | 7.80e-01 | -3.8   | 2.71e+02 | -5.3   | 1.60e-01 | 1.87e-01 | 1   |
| 174  | 1.0643176e+02 | 3.85e-01 | 5.25e-01 | -3.8   | 1.32e+02 | -4.8   | 4.35e-01 | 2.45e-01 | 1   |
| 175  | 1.0574448e+02 | 6.53e-01 | 7.64e-01 | -3.8   | 9.20e+02 | -5.3   | 7.55e-02 | 6.55e-02 | 2   |
| 176  | 1.0506841e+02 | 6.97e-01 | 7.23e-01 | -3.8   | 2.14e+02 | -4.9   | 3.64e-01 | 2.81e-01 | 1   |
| 177  | 1.0501340e+02 | 4.67e-01 | 5.44e-01 | -3.8   | 5.16e+01 | -4.5   | 1.96e-01 | 3.26e-01 | 1   |
| 178  | 1.0485871e+02 | 4.34e-01 | 5.71e-01 | -3.8   | 2.19e+02 | -4.9   | 5.23e-01 | 6.34e-02 | 2   |
| 179  | 1.0479127e+02 | 2.99e-01 | 4.53e-01 | -3.8   | 4.78e+01 | -4.5   | 1.00e+00 | 2.96e-01 | 1   |

5.17. Double inverted pendulum

(continues on next page)
Number of Iterations...: 227
(scaled) (unscaled)
Objective...: 1.0149767957695215e+02 1.0149767957695215e+02
Dual infeasibility...: 2.4670754328326439e-10 2.4670754328326439e-10
Constraint violation...: 2.65307858488954e-10 2.65307858488954e-10
Complementarity...: 2.5715300299692318e-09 2.5715300299692318e-09
Overall NLP error...: 2.5715300299692318e-09 2.5715300299692318e-09

Number of objective function evaluations = 248
Number of objective gradient evaluations = 228
Number of equality constraint evaluations = 248
Number of inequality constraint evaluations = 0
Number of equality constraint Jacobian evaluations = 228
Number of inequality constraint Jacobian evaluations = 0
Number of Lagrangian Hessian evaluations = 227
Total CPU secs in IPOPT (w/o function evaluations) = 3.739
Total CPU secs in NLP function evaluations = 0.239

EXIT: Optimal Solution Found.

The first optimization will take rather long (4 seconds) but in the end we get:

EXIT: Optimal Solution Found.

which tells us that we found an optimal solution. Note that follow-up optimizations take around 100 ms due to warmstarting.

We can visualize the open-loop prediction with:

```python
[27]: line1, line2 = pendulum_bars(x0)
bar1[0].set_data(line1[0],line1[1])
bar2[0].set_data(line2[0],line2[1])
mpc_graphics.plot_predictions()
mpc_graphics.reset_axes()
fig
```
The open-loop prediction looks perfectly fine! We see that within the horizon the potential energy settles on a plateau greater than zero, while the kinetic energy becomes zero. This indicates our desired up-up position. Both angles seem to reach $2\pi$.

### 5.17.5.3 Run closed-loop

The closed-loop system is now simulated for 100 steps (and the output of the optimizer is suppressed):

```python
[30]: %%capture
    # Quickly reset the history of the MPC data object.
    mpc.reset_history()

    n_steps = 100
    for k in range(n_steps):
        u0 = mpc.make_step(x0)
        y_next = simulator.make_step(u0)
        x0 = estimator.make_step(y_next)
```

### 5.17.5.4 Results

The next cell converts the results of the closed-loop MPC simulation into a gif (might take a few minutes):

```python
[31]: from matplotlib.animation import FuncAnimation, FFMpegWriter, ImageMagickWriter
    
    # The function describing the gif:
    x_arr = mpc.data['_x']
    def update(t_ind):
        line1, line2 = pendulum_bars(x_arr[t_ind])
        barl[0].set_data(line1[0],line1[1])
        (continues on next page)```
bar2[0].set_data(line2[0],line2[1])
mpc_graphics.plot_results(t_ind)
mpc_graphics.plot_predictions(t_ind)
mpc_graphics.reset_axes()

anim = FuncAnimation(fig, update, frames=n_steps, repeat=False)
gif_writer = ImageMagickWriter(fps=20)
anim.save('anim_dip.gif', writer=gif_writer)

The result is shown below, where solid lines are the recorded trajectories and dashed lines are the predictions of the scenarios:

5.17.6 Controller with obstacle avoidance

To make the example even more interesting it is possible to add obstacles and include a set-point tracking task, where the pendulum must be erect at a desired location.

Please note that the animation below now shows the pendulum position (of the cart) as well as the desired setpoint instead of the angles.

The code to create this animation is included in the do-mpc example files and is just an extension of the example shown above.
The necessary changes include the detection of obstacle intersection and an adapted objective function that includes the set-point tracking for the position.

5.18 Efficient data generation and handling with do-mpc

This notebook was used in our video tutorial on data generation and handling with do-mpc.
We start by importing basic modules and do-mpc.

```python
import numpy as np
import sys
from casadi import *
import os
import time

# Add do_mpc to path. This is not necessary if it was installed via pip
sys.path.append('..../..../')

# Import do_mpc package:
import do_mpc
import matplotlib.pyplot as plt
import pandas as pd
```
5.18.1 Toy example

Step 1: Create the sampling plan with the SamplingPlanner.

Define sampled variables, e.g.:

\[
\begin{align*}
\alpha, \beta \\
01 & \quad -1.221 & 4 & \ldots \\
02 & \quad 0.034 & 0 & \ldots \\
\vdots & \quad \vdots & \vdots & \vdots \\
\text{n}_{\text{samples}} & \quad -0.689 & 3 & \ldots
\end{align*}
\]

The planner is initiated and we set some (optional) parameters.

\[\text{sp} = \text{do_mpc.sampling.SamplingPlanner()}
\text{sp.set_param(overwrite = } \text{True})
\text{# This generates the directory, if it does not exist already.}
\text{sp.data_dir = './sampling_test/'}
\]

We then introduce new variables to the SamplingPlanner which will later jointly define a sampling case. Think of header rows in a table (see figure above).

These variables can themselves be sampled from a generating function or we add user defined cases one by one. If we want to sample variables to define the sampling case, we need to pass a sample generating function as shown below:

\[\text{sp.set_sampling_var('alpha', np.random.randn)}
\text{sp.set_sampling_var('beta', lambda: np.random.randint(0,5))}
\]

In this example we have two variables alpha and beta. We have:

\[\alpha \sim \mathcal{N}(\mu, \sigma)\]

and

\[\beta \sim \mathcal{U}([0,5])\]

Having defined generating functions for all of our variables, we can now generate a sampling plan with an arbitrary amount of cases:

\[\text{SamplingPlanner.gen_sampling_plan(n_samples)}\]

\[\text{plan} = \text{sp.gen_sampling_plan(n_samples=10)}\]

We can inspect the plan conveniently by converting it to a pandas DataFrame. Natively, the plan is a list of dictionaries.
If we do not wish to automatically generate a sampling plan, we can also add sampling cases one by one with:

```python
[6]: plan = sp.add_sampling_case(alpha=1, beta=-0.5)
    print(plan[-1])
{'alpha': 1, 'beta': -0.5, 'id': '010'}
```

Typically, we finish the process of generating the sampling plan by saving it to the disc. This is simply done with:

```python
sp.export(sampling_plan_name)
```

The save directory was already set with `sp.data_dir = ....`

**Step 2:** Create the `Sampler` object by providing the `sampling_plan`:

```python
[7]: sampler = do_mpc.sampling.Sampler(plan)
    sampler.set_param(overwrite = True)
```

Most important setting of the sampler is the `sample_function`. This function takes as arguments previously the defined `sampling_var` (from the configuration of the `SamplingPlanner`).
It this example, we create a dummy sampling generating function, where:

\[ f(\alpha, \beta) = \alpha \cdot \beta \]

```python
[8]: def sample_function(alpha, beta):
    time.sleep(0.1)
    return alpha*beta

sampler.set_sample_function(sample_function)
```

Before we sample, we want to set the directory for the created files and a name:

```python
[9]: sampler.data_dir = './sampling_test/'
sampler.set_param(sample_name = 'dummy_sample')
```

Now we can actually create all the samples:

```python
[10]: sampler.sample_data()
Progress: || 100.0% Complete
```

The sampler will now create the sampling results as a new file for each result and store them in a subfolder with the same name as the `sampling_plan`:

```python
[11]: ls = os.listdir('./sampling_test/')
ls.sort()
ls
```

```python
[11]: ['dummy_sample_000.pkl',
    'dummy_sample_001.pkl',
    'dummy_sample_002.pkl',
    'dummy_sample_003.pkl',
    'dummy_sample_004.pkl',
    'dummy_sample_005.pkl',
    'dummy_sample_006.pkl',
    'dummy_sample_007.pkl',
    'dummy_sample_008.pkl',
    'dummy_sample_009.pkl',
    'dummy_sample_010.pkl',
    'dummy_sample_011.pkl',
    'dummy_sample_012.pkl']
```

**Step 3:** Process data in the data handler class.
The first step is to initiate the class with the `sampling_plan`:

```
[12]: dh = do_mpc.sampling.DataHandler(plan)
```

We then need to point out where the data is stored and how the samples are called:

```
[13]: dh.data_dir = './sampling_test/'
   dh.set_param(sample_name = 'dummy_sample')
```

Next, we define the post-processing functions. For this toy example we do some “dummy” post-processing and request to compute two results:

```
[14]: dh.set_post_processing('res_1', lambda x: x)
   dh.set_post_processing('res_2', lambda x: x**2)
```

The interface of DataHandler.set_post_processing requires a name that we will see again later and a function that processes the output of the previously defined sample_function.

We can now obtain **obtain processed data** from the DataHandler in two ways. Note that we convert the returned list of dictionaries directly to a DataFrame for a better visualization.

1. **Indexing:**

```
[15]: pd.DataFrame(dh[:3])
```

```
   alpha  beta  id  res_1  res_2
0  0.105326  0.000000  0.000000  0.000000
1  0.784304  2.001568608  2.460532
2  0.257489  0.0257489  0.066301
```

Or we use a more complex filter with the `DataHandler.filter` method. This method requires either an input or an output filter in the form of a function.

Let’s retrieve all samples, where \( \alpha < 0 \):
Or we can filter by outputs, e.g. with:

```python
pd.DataFrame(dh.filter(output_filter = lambda res_2: res_2>10))
```

```
alpha  beta  id  res_1  res_2
0  1.04107  4  005  4.164281  17.341236
```

5.18.2 Sampling closed-loop trajectories

A more reasonable use-case in the scope of do-mpc is to sample closed-loop trajectories of a dynamical system with a (MPC) controller.

The approach is almost identical to our toy example above. The main difference lies in the `sample_function` that is passed to the Sampler and the `post_processing` in the DataHandler.

In the presented example, we will sample the oscillating mass system which is part of the do-mpc example library.

```
sys.path.append('../../../examples/oscillating_masses_discrete/')
from template_model import template_model
from template_mpc import template_mpc
from template_simulator import template_simulator
```

**Step 1: Create the sampling plan with the SamplingPlanner**

We want to generate various closed-loop trajectories of the system starting from random initial states, hence we design the SamplingPlanner as follows:

```
# Initialize sampling planner
sp = do_mpc.sampling.SamplingPlanner()
sp.set_param(overwrite=True)

# Sample random feasible initial states
def gen_initial_states():
    x0 = np.random.uniform(-3*np.ones((4,1)),3*np.ones((4,1)))
    return x0

# Add sampling variable including the corresponding evaluation function
sp.set_sampling_var('X0', gen_initial_states)
```

This implementation is sufficient to generate the sampling plan:

```
plan = sp.gen_sampling_plan(n_samples=9)
```

Since we want to run the system in the closed-loop in our sample function, we need to load the corresponding configuration:

```
model = template_model()
mpc = template_mpc(model)
estimator = do_mpc.estimator.StateFeedback(model)
simulator = template_simulator(model)
```
We can now define the sampling function:

```python
[22]: def run_closed_loop(X0):
    mpc.reset_history()
    simulator.reset_history()
    estimator.reset_history()

    # set initial values and guess
    x0 = X0
    mpc.x0 = x0
    simulator.x0 = x0
    estimator.x0 = x0

    mpc.set_initial_guess()

    # run the closed loop for 150 steps
    for k in range(100):
        u0 = mpc.make_step(x0)
        y_next = simulator.make_step(u0)
        x0 = estimator.make_step(y_next)

    # we return the complete data structure that we have obtained during the closed-loop run
    return simulator.data
```

Now we have all the ingredients to make our sampler:

```python
[23]: %%capture
    # Initialize sampler with generated plan
    sampler = do_mpc.sampling.Sampler(plan)

    # Set directory to store the results:
    sampler.data_dir = './sampling_closed_loop/'
    sampler.set_param(overwrite=True)

    # Set the sampling function
    sampler.set_sample_function(run_closed_loop)

    # Generate the data
    sampler.sample_data()
```

**Step 3:** Process data in the data handler class. The first step is to initiate the class with the `sampling_plan`:

```python
[24]: # Initialize DataHandler
    dh = do_mpc.sampling.DataHandler(plan)
    dh.data_dir = './sampling_closed_loop/'
```

In this case, we are interested in the states and the inputs of all trajectories. We define the following post processing functions:

```python
[25]: dh.set_post_processing('input', lambda data: data[['_u', 'u']])
    dh.set_post_processing('state', lambda data: data[['_x', 'x']])
```

To retrieve all post-processed data from the datahandler we use slicing. The result is stored in `res`.

```python
[26]: res = dh[:]
```

To inspect the sampled closed-loop trajectories, we create an array of plots where in each plot $x_2$ is plotted over $x_1$. This shows the different behavior, based on the sampled initial conditions:
\[
\text{n\_res} = \min(\text{len(res)}, 80)
\]
\[
\text{n\_row} = \text{int}(\text{np.ceil}(\text{np.sqrt(n\_res)}))
\]
\[
\text{n\_col} = \text{int}(\text{np.ceil(n\_res/n\_row)})
\]

```python
fig, ax = plt.subplots(n_row, n_col, sharex=True, sharey=True, figsize=(8,8))
for i, res_i in enumerate(res):
    ax[i//n_col, np.mod(i,n_col)].plot(res_i['state'][:,1],res_i['state'][:,0])
for i in range(ax.size):
    ax[i//n_col, np.mod(i,n_col)].axis('off')
fig.tight_layout(pad=0)
```

5.18. Efficient data generation and handling with do-mpc
CHAPTER 6

Indices and tables

- genindex
- search
d

do_mpc.controller, 94
do_mpc.data, 152
do_mpc.estimator, 116
do_mpc.graphics, 159
do_mpc.model, 64
do_mpc.optimizer, 85
do_mpc.sampling.datahandler, 167
do_mpc.sampling.sampler, 164
do_mpc.sampling.samplingplanner, 160
do_mpc.simulator, 78
Symbols

__getitem__() (do_mpc.data.Data method), 147
__getitem__() (do_mpc.data.MPCData method), 149
__getitem__() (do_mpc.model.Model method), 67

A
add_line() (in module do_mpc.graphics.Graphics), 157
add_sampling_case() (in module do_mpc.sampling.samplingplanner.SamplingPlanner), 161
animate() (in module do_mpc.graphics), 159
aux (do_mpc.model.Model attribute), 68

B
bounds (do_mpc.controller.MPC attribute), 96
bounds (do_mpc.estimator.MHE attribute), 124
bounds (do_mpc.optimizer.Optimizer attribute), 86

clear() (in module do_mpc.graphics.Graphics), 157
create_nlp() (in module do_mpc.controller.MPC), 106
create_nlp() (in module do_mpc.estimator.MHE), 134
create_nlp() (in module do_mpc.optimizer.Optimizer), 90

default_plot() (in module do_mpc.graphics), 159
do_mpc.controller (module), 94
do_mpc.data (module), 147, 152
do_mpc.estimator (module), 116
do_mpc.graphics (module), 156
do_mpc.model (module), 64
do_mpc.optimizer (module), 85
do_mpc.sampling.datahandler (module), 167
(do_mpc.sampling.sampler (module), 164
do_mpc.sampling.samplingplanner (module), 160

do_mpc.simulator (module), 78

E
EKF (class in do_mpc.estimator), 117
Estimator (class in do_mpc.estimator), 119
export () (in module do_mpc.data.Data), 148
export () (in module do_mpc.data.MPCData), 150
export () (in module do_mpc.sampling.samplingplanner.SamplingPlanner), 162

F
filter() (in module do_mpc.sampling.datahandler.DataHandler), 169

generate_sampling_plan() (in module do_mpc.sampling.samplingplanner.SamplingPlanner), 162
get_p_template() (in module do_mpc.controller.MPC), 106
get_p_template() (in module do_mpc.estimator.MHE), 134
get_p_template() (in module do_mpc.simulator.Simulator), 81
get_tvp_template() (in module do_mpc.controller.MPC), 107
sample_idx() (in module do_mpc.sampling.Sampler), 166
Sampler (class in do_mpc.sampling.Sampler), 164
SamplingPlanner (class in do_mpc.sampling.SamplingPlanner), 160
save_results() (in module do_mpc.data), 153
scaling (do_mpc.controller.MPC attribute), 102
scaling (do_mpc.estimator.MHE attribute), 130
set_alg() (in module do_mpc.model.Model), 73
set_default_objective() (in module do_mpc.estimator.MHE), 137
set_expression() (in module do_mpc.model.Model), 73
set_initial_guess() (in module do_mpc.controller.MPC), 109
set_initial_guess() (in module do_mpc.estimator.MHE), 138
set_initial_guess() (in module do_mpc.simulator.Simulator), 82
set_meas() (in module do_mpc.model.Model), 74
set_meta() (in module do_mpc.data.Data), 149
set_meta() (in module do_mpc.data.MPCData), 152
set_nl_cons() (in module do_mpc.controller.MPC), 109
set_nl_cons() (in module do_mpc.estimator.MHE), 139
set_nl_cons() (in module do_mpc.optimizer.Optimizer), 92
set_objective() (in module do_mpc.controller.MPC), 110
set_objective() (in module do_mpc.estimator.MHE), 139
set_p_fun() (in module do_mpc.controller.MPC), 111
set_p_fun() (in module do_mpc.estimator.MHE), 140
set_p_fun() (in module do_mpc.simulator.Simulator), 82
set_param() (in module do_mpc.controller.MPC), 111
set_param() (in module do_mpc.estimator.MHE), 141
set_param() (in module do_mpc.optimizer.Optimizer), 92
set_p_fun() (in module do_mpc.controller.MPC), 111
set_p_fun() (in module do_mpc.estimator.MHE), 140
set_p_fun() (in module do_mpc.simulator.Simulator), 82
set_post_processing() (in module do_mpc.sampling.samplingplanner.SamplingPlanner), 166
set_rterm() (in module do_mpc.controller.MPC), 113
set_sample_function() (in module do_mpc.sampling.Sampler), 166
set_sampling_var() (in module do_mpc.sampling.SamplingPlanner), 163
set_tvp_fun() (in module do_mpc.controller.MPC), 114
set_tvp_fun() (in module do_mpc.estimator.MHE), 142
set_tvp_fun() (in module do_mpc.optimizer.Optimizer), 93
set_tvp_fun() (in module do_mpc.simulator.Simulator), 84
set_uncertainty_values() (in module do_mpc.controller.MPC), 115
set_variable() (in module do_mpc.model.Model), 76
set_y_fun() (in module do_mpc.estimator.MHE), 143
setup() (in module do_mpc.controller.MPC), 115
setup() (in module do_mpc.estimator.MHE), 143
setup() (in module do_mpc.model.Model), 77
simulate() (in module do_mpc.simulator.Simulator), 84
StateFeedback (class in do_mpc.estimator), 144
T

t0 (do_mpc.controller.MPC attribute), 103
t0 (do_mpc.estimator.EKF attribute), 117
t0 (do_mpc.estimator.Estimator attribute), 120
t0 (do_mpc.estimator.MHE attribute), 131
t0 (do_mpc.model.IteratedVariables attribute), 64
t0 (do_mpc.simulator.Simulator attribute), 78
terminal_bounds (do_mpc.controller.MPC attribute), 103
tvp (do_mpc.model.Model attribute), 69
U

u0 (do_mpc.controller.MPC attribute), 103
u0 (do_mpc.estimator.EKF attribute), 117
u0 (do_mpc.estimator.Estimator attribute), 120
u0 (do_mpc.estimator.MHE attribute), 131
u0 (do_mpc.estimator.StateFeedback attribute), 145
u0 (do_mpc.model.IteratedVariables attribute), 64
u0 (do_mpc.simulator.Simulator attribute), 79
ub_opt_x (do_mpc.controller.MPC attribute), 104
ub_opt_x (do_mpc.estimator.MHE attribute), 132
ub_opt_x (do_mpc.optimizer.Optimizer attribute), 90
update() (in module do_mpc.data.Data), 149
update() (in module do_mpc.data.MPCData), 152
V
v (do_mpc.model.Model attribute), 70
W
w (do_mpc.model.Model attribute), 71
X
x (do_mpc.model.Model attribute), 71
x0 (do_mpc.controller.MPC attribute), 104
x0 (do_mpc.estimator.EKF attribute), 118
x0 (do_mpc.estimator.Estimator attribute), 121
x0 (do_mpc.estimator.MHE attribute), 132
x0 (do_mpc.estimator.StateFeedback attribute), 145
x0 (do_mpc.model.IteratedVariables attribute), 65
x0 (do_mpc.simulator.Simulator attribute), 79
Y
y (do_mpc.model.Model attribute), 72
Z
z (do_mpc.model.Model attribute), 72
z0 (do_mpc.controller.MPC attribute), 105
z0 (do_mpc.estimator.EKF attribute), 118
z0 (do_mpc.estimator.Estimator attribute), 121
z0 (do_mpc.estimator.MHE attribute), 133
z0 (do_mpc.estimator.StateFeedback attribute), 146
z0 (do_mpc.model.IteratedVariables attribute), 66
z0 (do_mpc.simulator.Simulator attribute), 80