do-mpc

Release 4.0.0

Jun 25, 2020

Introduction

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
- 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 IOT chair of the TU Berlin by Felix Fiedler and Sergio Lucia.

CHAPTER 1

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:

CHAPTER 2

Next steps

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.

CHAPTER 3

Table of contents

3.1 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`.

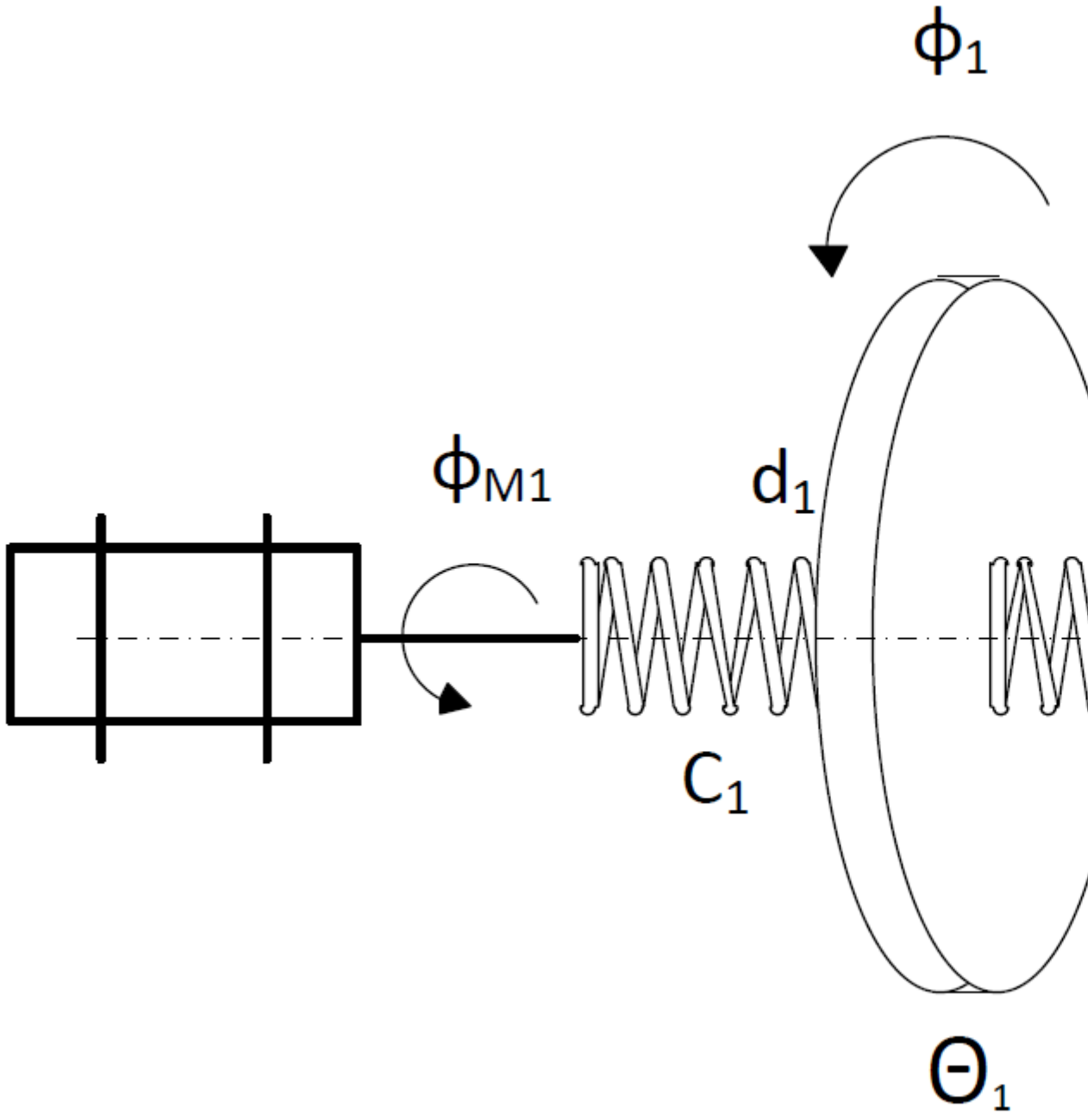
[1]:
```
import numpy as np

# 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 our jointly, depending on the application.

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

3.1.1 Example system

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



picted below:

Three rotating discs are connected via springs and we denote their angles as ϕ_1, ϕ_2, ϕ_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 Θ 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:

$$\Theta_1\ddot{\phi}_1 = -c_1\left(\phi_1 - \phi_{m,1}\right) - c_2\left(\phi_1 - \phi_2\right) - d_1\dot{\phi}_1 \tag{3.1}$$
$$\Theta_2\ddot{\phi}_2 = -c_2\left(\phi_2 - \phi_1\right) - c_3\left(\phi_2 - \phi_3\right) - d_2\dot{\phi}_2 \tag{3.2}$$
$$\Theta_3\ddot{\phi}_3 = -c_3\left(\phi_3 - \phi_2\right) - c_4\left(\phi_3 - \phi_{m,2}\right) - d_3\dot{\phi}_3 \tag{3.3}$$

The uncontrolled system, starting from a non-zero initial state will osciallate 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:

3.1.2 Creating the model

As indicated above, the `model` block is essential for the application of **do-mpc**. In mathmatical 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 an DAE/ODE we write:

$$\frac{\partial x}{\partial t} = f(x, u, z, p) \tag{3.4}$$
$$0 = g(x, u, z, p) \tag{3.5}$$
$$y = h(x, u, z, p) \tag{3.6}$$

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, be introducing the following states:

$$x_1 = \phi_1 \tag{3.7}$$
$$x_2 = \phi_2 \tag{3.8}$$
$$x_3 = \phi_3 \tag{3.9}$$
$$x_4 = \dot{\phi}_1 \tag{3.10}$$
$$x_5 = \dot{\phi}_2 \tag{3.11}$$
$$x_6 = \dot{\phi}_3 \tag{3.12}$$
$$\tag{3.13}$$

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

$$\dot{x}_1 = x_4 \tag{3.14}$$
$$\dot{x}_2 = x_5 \tag{3.15}$$
$$\dot{x}_3 = x_6 \tag{3.16}$$
$$\dot{x}_4 = -\frac{c_1}{\Theta_1}\left(x_1 - u_1\right) - \frac{c_2}{\Theta_1}\left(x1 - x_2\right) - \frac{d_1}{\Theta_1}x_4 \tag{3.17}$$
$$\dot{x}_5 = -\frac{c_2}{\Theta_2}\left(x_2 - x_1\right) - \frac{c_3}{\Theta_2}\left(x_2 - x_3\right) - \frac{d_2}{\Theta_2}x_5 \tag{3.18}$$
$$\dot{x}_6 = -\frac{c_3}{\Theta_3}\left(x_3 - x_2\right) - \frac{c_4}{\Theta_3}\left(x_4 - u_2\right) - \frac{d_3}{\Theta_3}x_6 \tag{3.19}$$
$$\tag{3.20}$$

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.

[2]:
```
model_type = 'continuous' # either 'discrete' or 'continuous'
model = do_mpc.model.Model(model_type)
```

3.1.2.1 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:

Long name	short name	Remark
`states`	`_x`	Required
`inputs`	`_u`	Required
`algebraic`	`_z`	Optional
`parameter`	`_p`	Optional
`timevarying_parameter`	`_tvp`	Optional

[3]:
```
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:

[4]:
```
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)
```

3.1.2.2 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`:

[5]:
```
model.x
```

[5]:
```
<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:

[6]:
```
model.x['phi_1']
```

[6]:
```
SX(phi_1)
```

Note that this is identical to the output of `model.set_variable` from above:

[7]:
```
bool(model.x['phi_1'] == phi_1)
```

[7]:
```
True
```

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

[8]:
```
model.x['dphi',0]
```

[8]:
```
SX(dphi_0)
```

Note that you can use the following methods:

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

to get more information from the symbolic structures:

[9]:
```
model.x.keys()
```

[9]:
```
['phi_1', 'phi_2', 'phi_3', 'dphi', 'phi_1_m', 'phi_2_m']
```

[10]:
```
model.x.labels()
```

[10]:
```
['[phi_1,0]',
 '[phi_2,0]',
 '[phi_3,0]',
 '[dphi,0]',
 '[dphi,1]',
 '[dphi,2]',
 '[phi_1_m,0]',
 '[phi_2_m,0]']
```

3.1.2.3 Model parameters

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

[11]:
```
# 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
```

3.1.2.4 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.

[12]:
```
model.set_rhs('phi_1', dphi[0])
model.set_rhs('phi_2', dphi[1])
model.set_rhs('phi_3', dphi[2])
```

For the vector valued state `dphi` we need to concatenate symbolic expressions. We import the symbolic library CasADi:

[13]:
```
from casadi import *
```

[14]:
```
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],
)

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

[15]:
```
tau = 1e-2
model.set_rhs('phi_1_m', 1/tau*(phi_m_1_set - phi_1_m))
model.set_rhs('phi_2_m', 1/tau*(phi_m_2_set - phi_2_m))
```

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

[16]:
```
model.setup()
```

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

3.1.3 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)

[17]:
```
mpc = do_mpc.controller.MPC(model)
```

3.1.3.1 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`.

[18]:
```
setup_mpc = {
    'n_horizon': 20,
    't_step': 0.1,
    'n_robust': 1,
    'store_full_solution': True,
}
mpc.set_param(**setup_mpc)
```

3.1.3.2 Objective function

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

$$C = \sum_{k=0}^{n-1} \left(\underbrace{l(x_k, u_k, z_k, p)}_{\text{lagrange term}} + \underbrace{\Delta u_k^T R \Delta u_k}_{\text{r-term}} \right) + \underbrace{m(x_n)}_{\text{meyer term}}$$

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

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

[19]:
```
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 **penality 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 Δu_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.

3.1.3.3 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
# Lower 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
```

3.1.3.4 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).

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

3.1.3.5 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:

[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.

3.1.3.6 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.

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

3.1.4 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`:

[25]:
```
simulator = do_mpc.simulator.Simulator(model)
```

3.1.4.1 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`.

[26]:
```
# 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)
```

3.1.4.2 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 evaluted 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:

[27]:
```
p_template = simulator.get_p_template()
```

This object is a CasADi structure:

[28]:
```
type(p_template)
```

[28]:
```
casadi.tools.structure3.DMStruct
```

which can be indexed with the following keys:

[29]:
```
p_template.keys()
```

[29]:
```
['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:

[30]:
```
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:

[31]:
```
simulator.set_p_fun(p_fun)
```

3.1.4.3 Setup

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

[32]:
```
simulator.setup()
```

3.1.5 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 $x_0 \neq 0$:

[33]:
```
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.

[34]:
```
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:

[35]:
```
mpc.x0
```

[35]:
```
<casadi.tools.structure3.DMStruct at 0x7fa71b5ee390>
```

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

[36]:
```
mpc.x0['phi_1']
```

[36]:
```
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:

[37]:
```
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.

3.1.5.1 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:

[38]:
```
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`.

[39]:
```
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`:

[40]:
```
%%capture
# We just want to create the plot and not show it right now. This "inline magic"␣
↪supresses 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:

[41]:
```
%%capture
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])


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

3.1.5.2 Running the simulator

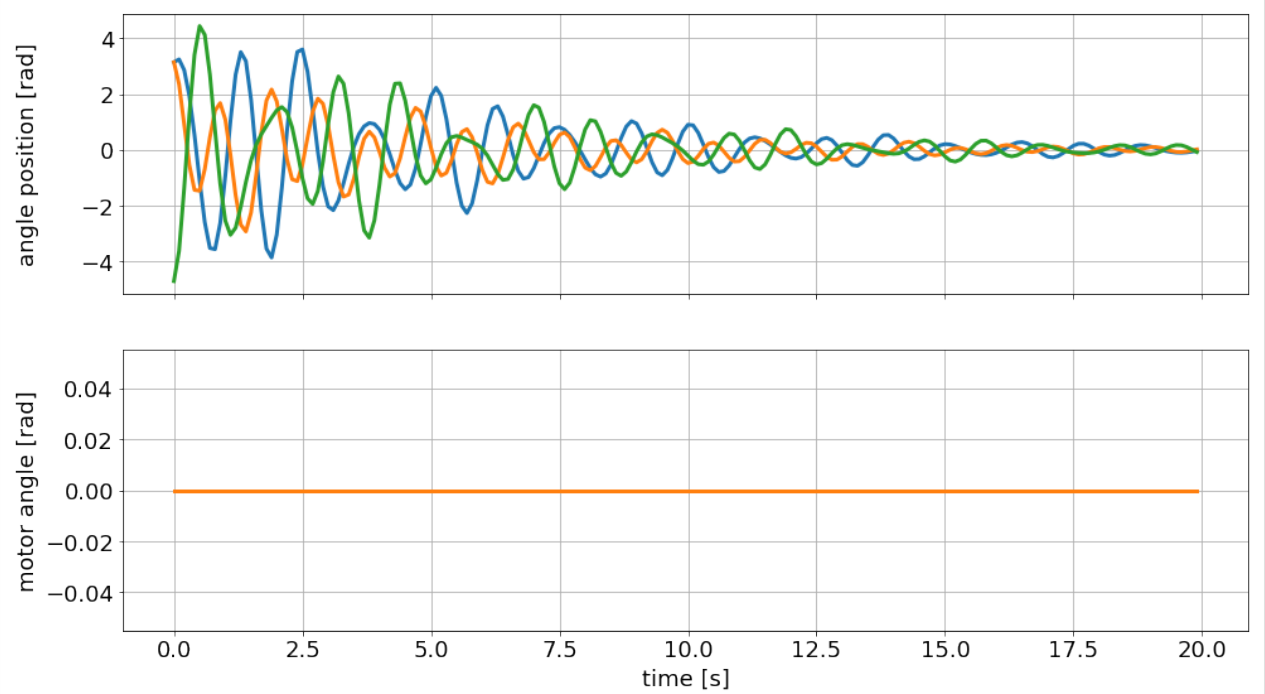
We start investigating the **do-mpc** simulator and the `graphics` package by simulating the autonomous system without control inputs ($u = 0$). This can be done as follows:

[42]:
```
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:

[43]:
```
sim_graphics.plot_results()
# Reset the limits on all axes in graphic to show the data.
sim_graphics.reset_axes()
# Show the figure:
fig
```

[43]:



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.

3.1.5.3 Running the optimizer

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

[44]:
```
u0 = mpc.make_step(x0)
```

```
******************************************************************************
This program contains Ipopt, a library for large-scale nonlinear optimization.
 Ipopt is released as open source code under the Eclipse Public License (EPL).
         For more information visit http://projects.coin-or.org/Ipopt
******************************************************************************

This is Ipopt version 3.12.3, running with linear solver mumps.
NOTE: Other linear solvers might be more efficient (see Ipopt documentation).

Number of nonzeros in equality constraint Jacobian...:    19448
Number of nonzeros in inequality constraint Jacobian.:        0
Number of nonzeros in Lagrangian Hessian...:     1229

Total number of variables...:     6408
                     variables with only lower bounds:        0
                variables with lower and upper bounds:     2435
                     variables with only upper bounds:        0
Total number of equality constraints...:     5768
Total number of inequality constraints...:        0
        inequality constraints with only lower bounds:        0
```

(continues on next page)

(continued from previous page)

```
   inequality constraints with lower and upper bounds:        0
        inequality constraints with only upper bounds:        0

iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
   0  8.8086219e+02 1.65e+01 1.07e-01  -1.0 0.00e+00    -  0.00e+00 0.00e+00   0
   1  2.8794996e+02 2.32e+00 1.68e+00  -1.0 1.38e+01  -4.0 2.82e-01 8.60e-01f  1
   2  2.0017562e+02 1.87e-14 3.95e+00  -1.0 3.56e+00  -4.5 1.96e-01 1.00e+00f  1
   3  1.6039802e+02 1.48e-14 3.82e-01  -1.0 3.43e+00  -5.0 5.14e-01 1.00e+00f  1
   4  1.3046012e+02 2.04e-14 7.36e-02  -1.0 2.94e+00  -5.4 7.75e-01 1.00e+00f  1
   5  1.1452477e+02 2.04e-14 1.94e-02  -1.7 2.62e+00  -5.9 8.44e-01 1.00e+00f  1
   6  1.1247422e+02 1.87e-14 7.23e-03  -2.5 9.17e-01  -6.4 8.27e-01 1.00e+00f  1
   7  1.1235000e+02 1.69e-14 4.88e-08  -2.5 3.56e-01  -6.9 1.00e+00 1.00e+00f  1
   8  1.1230585e+02 1.87e-14 8.91e-09  -3.8 1.95e-01  -7.3 1.00e+00 1.00e+00f  1
   9  1.1229857e+02 1.83e-14 8.02e-10  -5.7 5.26e-02  -7.8 1.00e+00 1.00e+00f  1
iter    objective    inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
  10  1.1229833e+02 1.51e-14 6.08e-09  -5.7 1.20e+00  -8.3 1.00e+00 1.00e+00f  1
  11  1.1229831e+02 1.69e-14 3.25e-13  -8.6 1.92e-04  -8.8 1.00e+00 1.00e+00f  1

Number of Iterations...: 11

                                   (scaled)                 (unscaled)
Objective...:   1.1229831239969913e+02    1.1229831239969913e+02
Dual infeasibility...:   3.2479227640713759e-13    3.2479227640713759e-13
Constraint violation...:   1.6875389974302379e-14    1.6875389974302379e-14
Complementarity...:   4.2481089749952700e-09    4.2481089749952700e-09
Overall NLP error...:   4.2481089749952700e-09    4.2481089749952700e-09


Number of objective function evaluations             = 12
Number of objective gradient evaluations             = 12
Number of equality constraint evaluations            = 12
Number of inequality constraint evaluations          = 0
Number of equality constraint Jacobian evaluations   = 12
Number of inequality constraint Jacobian evaluations = 0
Number of Lagrangian Hessian evaluations             = 11
Total CPU secs in IPOPT (w/o function evaluations)   =      0.239
Total CPU secs in NLP function evaluations           =      0.006

EXIT: Optimal Solution Found.
           S  :   t_proc      (avg)   t_wall      (avg)    n_eval
       nlp_f  | 149.00us ( 12.42us) 145.00us ( 12.08us)        12
       nlp_g  |   2.16ms (180.17us)   1.83ms (152.33us)        12
    nlp_grad  | 377.00us (377.00us) 377.00us (377.00us)         1
  nlp_grad_f  | 504.00us ( 38.77us) 525.00us ( 40.38us)        13
  nlp_hess_l  | 138.00us ( 12.55us) 137.00us ( 12.45us)        11
   nlp_jac_g  |   3.27ms (251.31us)   3.26ms (251.00us)        13
       total  | 257.31ms (257.31ms) 256.06ms (256.06ms)         1
```

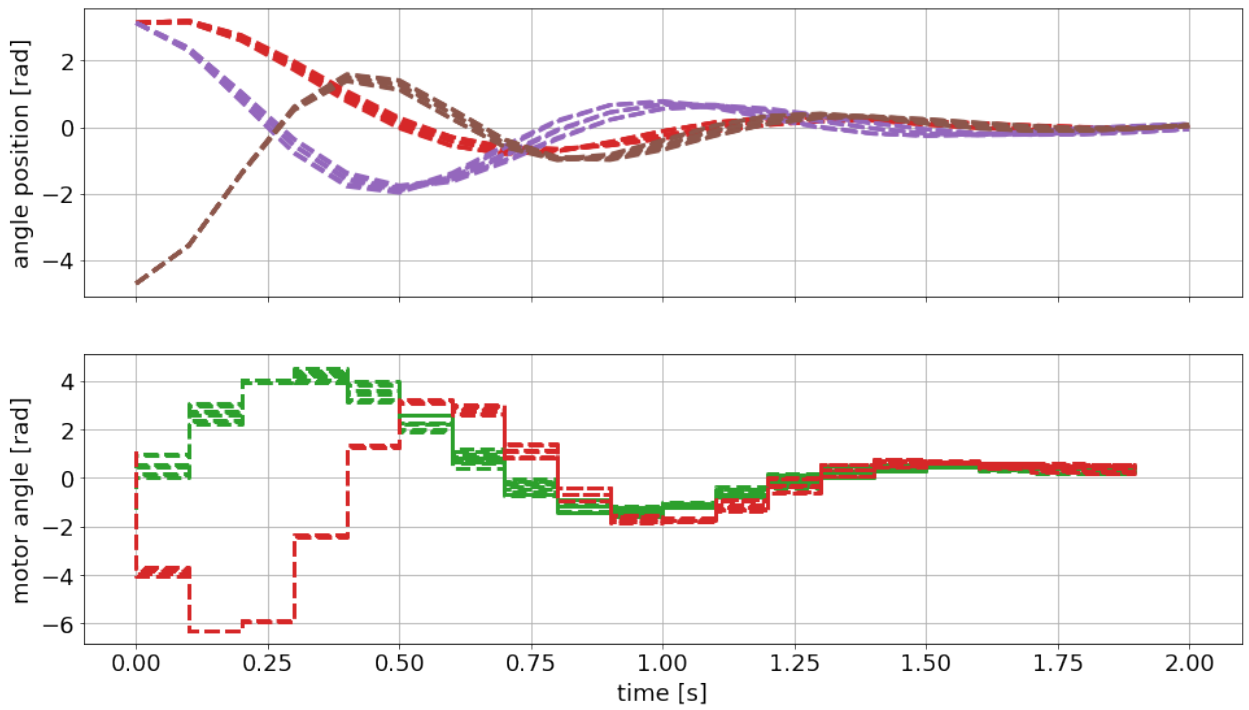
Note that we obtained the output from IPOPT regarding the given optimal control problem (OCP). Most importantly we obtained `Optimal Solution Found`.

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:

[46]:
```
mpc_graphics.plot_predictions()
mpc_graphics.reset_axes()
# Show the figure:
fig
```

[46]:


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.

3.1.5.4 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:

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

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

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

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

[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 0x7fa71c7022b0>,
 <matplotlib.lines.Line2D at 0x7fa71c702630>,
 <matplotlib.lines.Line2D at 0x7fa71c702978>,
 <matplotlib.lines.Line2D at 0x7fa71c702d30>,
 <matplotlib.lines.Line2D at 0x7fa71c702c88>]
```

We obtain all lines for our first state. To change the color we can simply:

[49]:
```
# Change the color for the three states:
for line_i in mpc_graphics.pred_lines['_x', 'phi_1']: line_i.set_color('#1f77b4') #␣
↪orange
for line_i in mpc_graphics.pred_lines['_x', 'phi_2']: line_i.set_color('#ff7f0e') #␣
↪blue
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
↪')
for line_i in mpc_graphics.pred_lines['_u', 'phi_m_2_set']: line_i.set_color('#ff7f0e
↪')

# 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:

[50]:
```
# 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')
```

[50]:
```
<matplotlib.legend.Legend at 0x7fa71c712eb8>
```

3.1.5.5 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 reset the initial state and erase the history:

[51]:
```
simulator.set_initial_state(x0, reset_history=True)
mpc.set_initial_state(x0, reset_history=True)
```

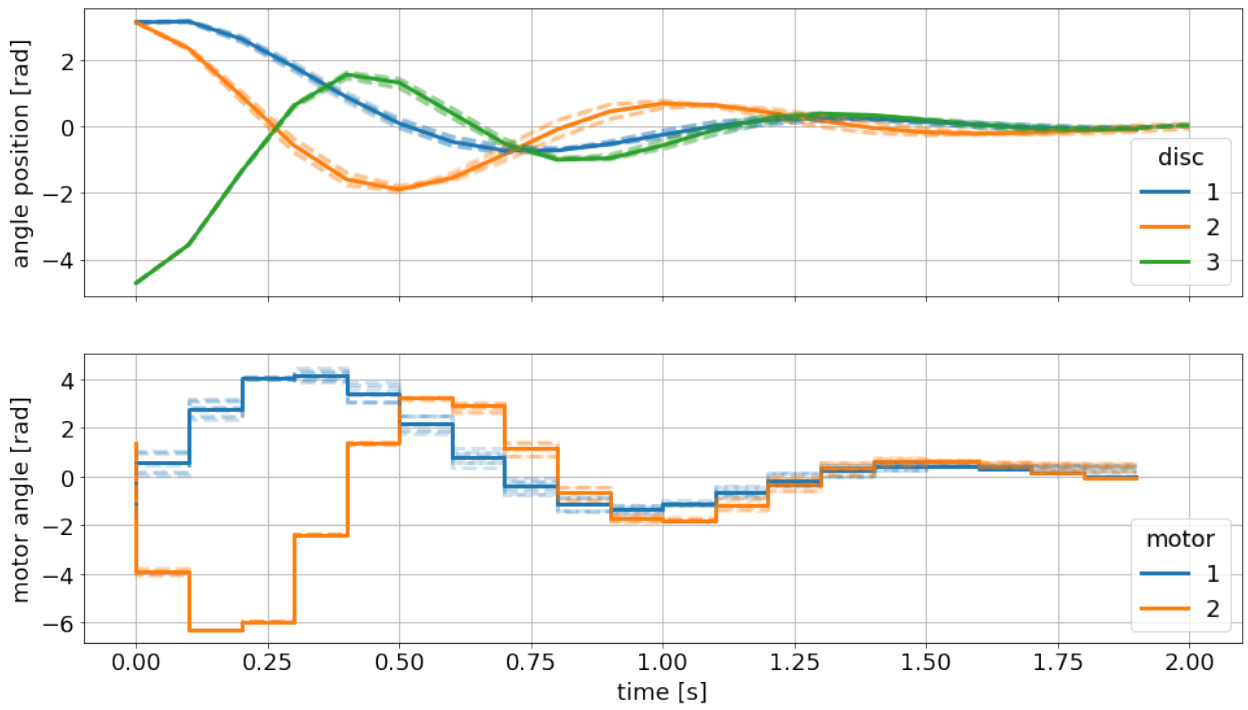
This is the main-loop. We run 20 steps, whic is identical to the prediction horizon. Note that we use "capture" again, to supress the output from IPOPT.

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

[52]:
```
%%capture
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:

[53]:
```
# Plot predictions from t=0
mpc_graphics.plot_predictions(t_ind=0)
# Plot results until current time
sim_graphics.plot_results()
sim_graphics.reset_axes()
fig
```

[53]:


The simulated trajectory with the nominal value of the parameters follows almost exactly the nominal open-loop predictions. The simulted trajectory is bounded from above and below by further uncertain scenarios.

3.1.6 Data processing

3.1.6.1 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:

[55]:
```
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.

[56]:
```
save_results([mpc, simulator])
```

We investigate the content of the newly created folder:

[57]:
```
!ls ./results/
```

```
results.pkl
```

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`).

3.1.6.2 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:

[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.

3.1.6.3 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:

[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.

[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**.

3.2 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.

[1]:
```
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
```

3.2.1 Creating the model

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

[2]:
```
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:

$$\dot{x}(t) = f(x(t), u(t), z(t), p(t), p_{\text{tv}}(t)) + w(t), \tag{3.21}$$
$$y(t) = h(x(t), u(t), z(t), p(t), p_{\text{tv}}(t)) + v(t), \tag{3.22}$$

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

3.2.1.1 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.

[3]:
```
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))
```

3.2.1.2 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 mentionned above, we need to define for each measurement if additive noise is present. In our case we assume noisy state measurements (ϕ) but perfect input measurements.

[4]:
```
# 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)
```

3.2.1.3 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
```

3.2.1.4 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 inidividual 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 explictly.

[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_model()`:

[7]:
```
model.setup_model()
```

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

3.2.2 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.

---

[8]:
```
mhe = do_mpc.estimator.MHE(model, ['Theta_1'])
```

3.2.2.1 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.

[9]:
```
setup_mhe = {
    't_step': 0.1,
    'n_horizon': 10,
    'store_full_solution': True,
    'meas_from_data': True
}
mhe.set_param(**setup_mhe)
```

3.2.2.2 Objective function

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

$$\min_{\substack{\mathbf{x}_{0:N+1},\,\mathbf{u}_{0:N},\,p,\\ \mathbf{w}_{0:N},\,\mathbf{v}_{0:N}}} \frac{1}{2}\|x_0-\tilde{x}_0\|^2_{P_x}+\frac{1}{2}\|p-\tilde{p}\|^2_{P_p}+\sum_{k=0}^{N-1}\left(\frac{1}{2}\|v_k\|^2_{P_{v,k}}+\frac{1}{2}\|w_k\|^2_{P_{w,k}}\right), \tag{3.23}$$

$$\text{s.t.}\quad \left.\begin{aligned} x_{k+1} &= f(x_k,u_k,z_k,p,p_{\text{tv},k})+w_k,\\ y_k &= h(x_k,u_k,z_k,p,p_{\text{tv},k})+v_k,\\ g(x_k,u_k,z_k,p_k,p_{\text{tv},k}) &\leq 0 \end{aligned}\right\} k=0,\ldots,N \tag{3.24}$$

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:

[10]:
```
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)
```

3.2.2.3 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:

[11]:
```
p_template_mhe = mhe.get_p_template()
```

Using this structure, we then formulate the following function for the remaining (not estimated) parameters:

[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)
```

3.2.2.4 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
```

3.2.2.5 Setup

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

[15]:
```
mhe.setup()
```

3.2.3 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)
```

3.2.3.1 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_
↪param(),
# we can also use keywords (and call the method multiple times, if necessary):
simulator.set_param(t_step = 0.1)
```

3.2.3.2 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 evaluted 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:

[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:

[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:

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

3.2.3.3 Setup

Finally, we call:

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

3.2.4 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).

[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:

[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:

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

It is also adviced 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:

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

3.2.4.1 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
# We just want to create the plot and not show it right now. This "inline magic"␣
↪surpresses the output.
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 \n 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
```

[30]:
```
<do_mpc.tools.structure.Structure at 0x7ff934280a20>
```

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

[31]:
```
# First element for state phi:
sim_graphics.result_lines['_x', 'phi', 0]
```

[31]:
```
[<matplotlib.lines.Line2D at 0x7ff934a54fd0>]
```

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

[32]:
```
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:

[33]:
```
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')
```

[33]:
```
<matplotlib.legend.Legend at 0x7ff934a8fcf8>
```

and another legend for the parameter plot:

[34]:
```
ax_p.legend(sim_graphics.result_lines['_p', 'Theta_1']+mhe_graphics.result_lines['_p',
↪ 'Theta_1'], ['True','Estim.'])
```

[34]:
```
<matplotlib.legend.Legend at 0x7ff934a8fe80>
```

3.2.4.2 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:

[35]:
```
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
```

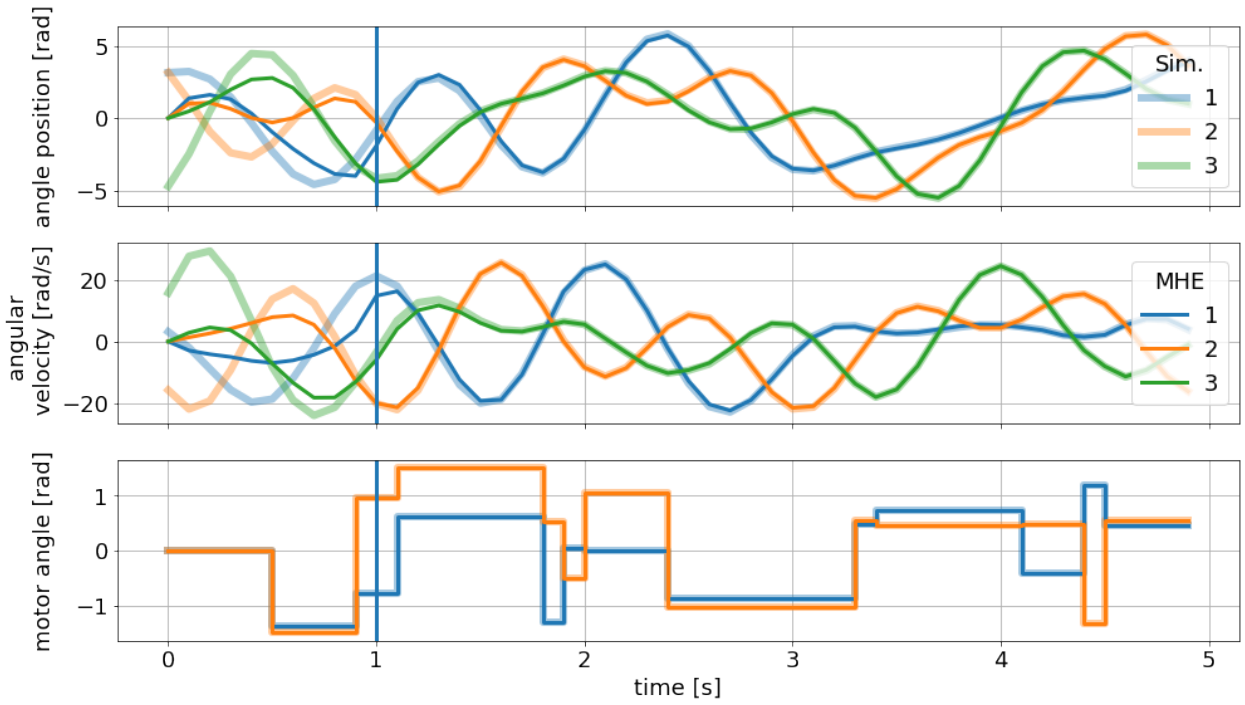
The function holds the current input value with 80% chance or switches to a new random input value.

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.

[36]:
```
%%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
```

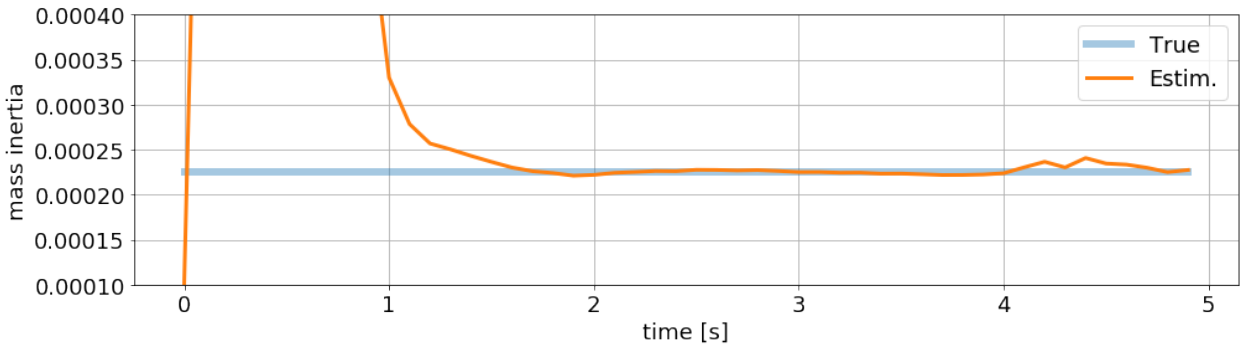
We can visualize the resulting trajectory with the pre-defined graphic:

[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:

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

[38]:


3.2.5 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:

[39]:
```
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.

[40]:
```
mhe.setup()
```

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

[41]:
```
mhe.reset_history()
simulator.reset_history()
```

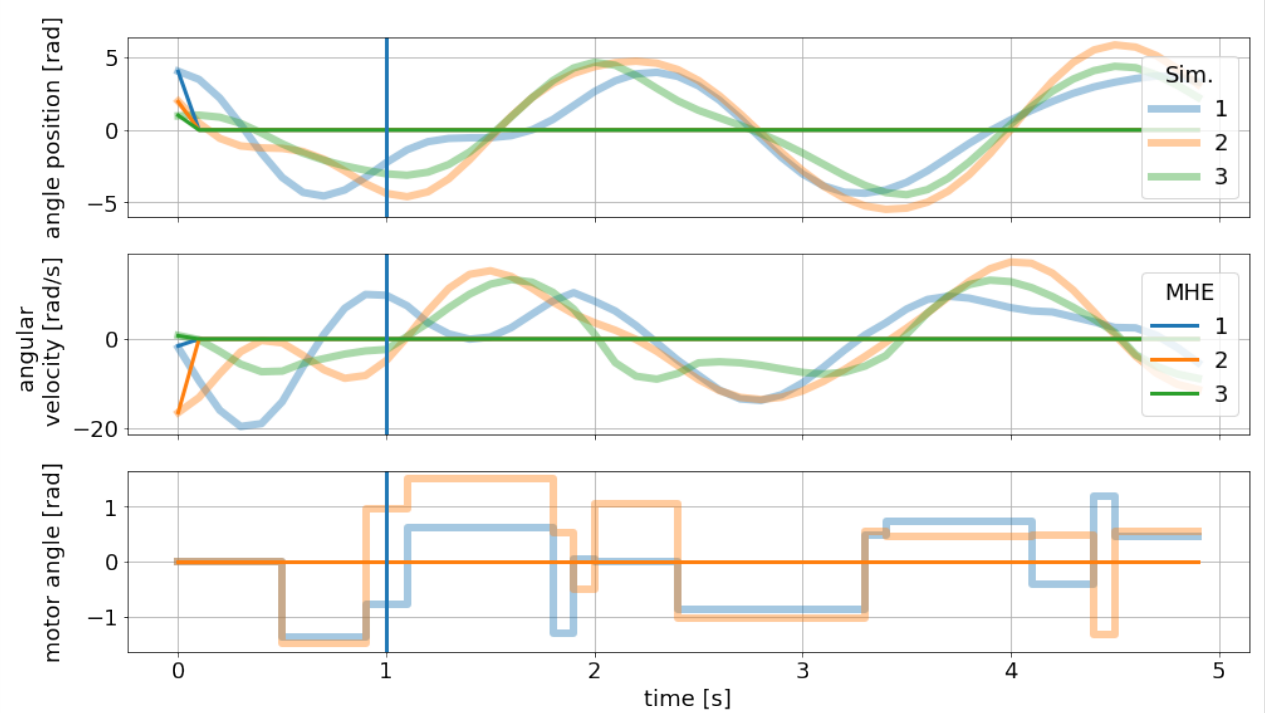
Finally, we run the exact same loop again obtaining new results.

[42]:
```
%%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:

[43]:
```
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
```

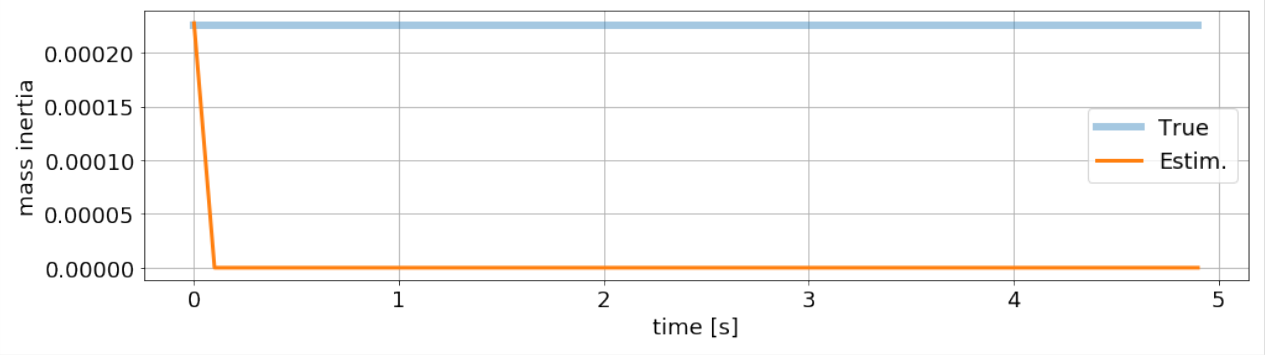
[43]:



Clearly, the main problem is a faulty parameter estimation, which is off by orders of magnitude:

[44]:
```
ax_p.set_ylabel('mass inertia')
ax_p.set_xlabel('time [s]')
fig_p
```

[44]:



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**.

3.3 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].

3.3.1 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 $K + 1$:

$$x_i^K(t) = \alpha_0 + \alpha_1 t + \cdots + \alpha_K t^K.$$

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

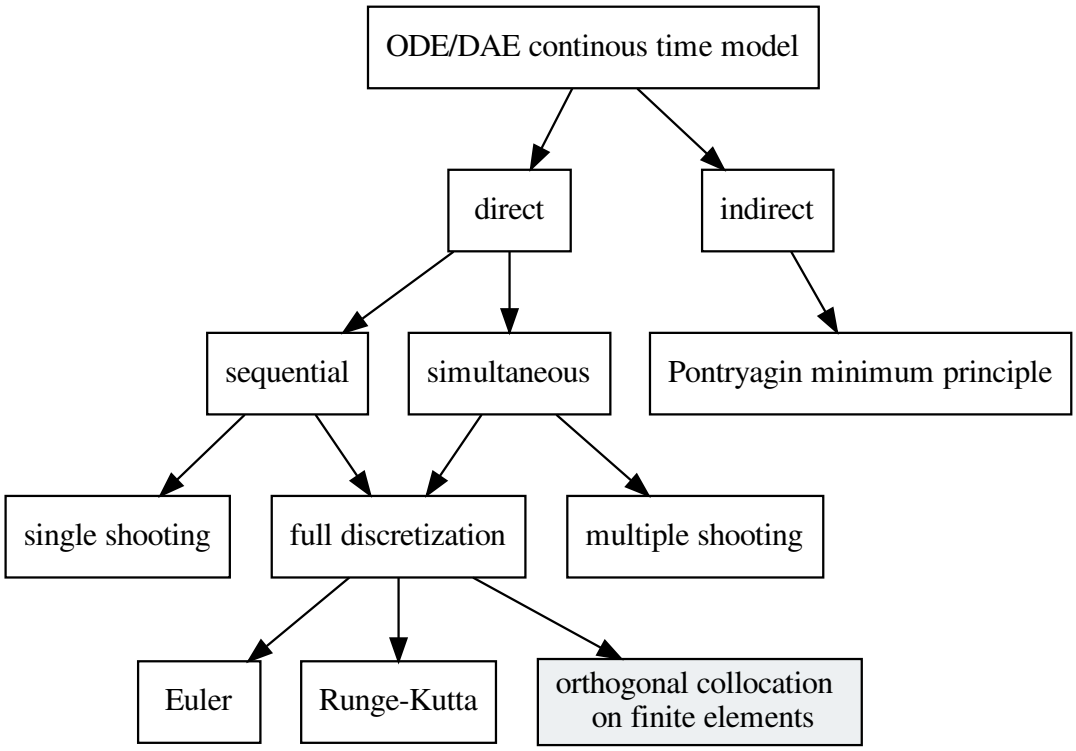


Fig. 1: Approaching an ODE/DAE continuous model for MPC or MHE.

Lagrange interpolation polynomial:

$$x_i^K(t) = \sum_{j=0}^{K} L_j(\tau) x_{i,j}$$

$$\text{where:} \quad L_j(\tau) = \prod_{\substack{k=0 \\ k \neq j}}^{K} \frac{(\tau - \tau_k)}{(\tau_j - \tau_k)}, \quad \tau$$

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

We call $L_j(\tau)$ the Lagrangrian 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 choosen 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*.

3.3.2 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**:

$$\left.\frac{dx_i^K}{dt}\right|_{t_{i,k}} = f(x_{i,k}), \quad k = 1, \ldots, K.$$

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 $x_i^K(t)$ the time derivative can be conveniently expressed as:

$$\left.\frac{dx_i^K}{dt}\right|_{t_{i,k}} = \sum_{j=0}^{K} \frac{x_{i,j}}{\Delta t} \underbrace{\left.\frac{dL_j}{d\tau}\right|_{\tau_k}}_{a_{j,k}},$$

for which we substituted t with τ. 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*.

3.3.2.1 Collocation constraints

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

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

3.3.2.2 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.

3.3.2.3 Solving the ODE problem

It is important to note that orthogonal collocation on finite elements is an **implict 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:

$$\begin{aligned} \min_{z} \quad & c(z) \\ \text{s.t.:} \quad & h(z) = 0 \\ & g(z) \leq 0 \end{aligned}$$

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 implictly calculates the solution of the governing ODE which can be taken into consideration for cost, constraints etc.

3.3.2.4 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 τ as the roots of a k-th degree polynomial $P_K(\tau)$ which fulfils the **orthogonal property**:

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

The resulting collocation points are called **Legendre roots**.

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

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

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

Both, Gauss-Radau and Legrende 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].

3.3.3 Bibliography

3.4 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.

3.4.1 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

$$\begin{aligned}\dot{x}(t) &= f(x(t), u(t), z(t), p(t), p_{\text{tv}}(t)),\\ y(t) &= h(x(t), u(t), z(t), p(t), p_{\text{tv}}(t)),\end{aligned}$$

and for the discrete-time case by

$$\begin{aligned}x_{k+1} &= f(x_k, u_k, z_k, p_k, p_{\text{tv},k}),\\ y_k &= h(x_k, u_k, z_k, p_k, p_{\text{tv},k}).\end{aligned}$$

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 $p_{\text{tv}}(t), p_{\text{tv},k}$ 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.

3.4.2 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:

$$\min_{\mathbf{x}_{0:N+1},\mathbf{u}_{0:N},\mathbf{z}_{0:N}} m(x_{N+1}) + \sum_{k=0}^{N} l(x_k, z_k, u_k, p_k, p_{\text{tv},k})$$

subject to:

$$x_0 = \hat{x}_0,$$

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

$$g(x_k, u_k, p_k, p_{\text{tv},k}) \leq 0 \quad \forall k = 0, \ldots, N,$$

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

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

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

$$g_{\text{terminal}}(x_{N+1}) \leq 0,$$

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 (y_k). Note that we introduce the bold letter notation, e.g. $\mathbf{x}_{0:N+1} = [x_0, x_1, \ldots, x_{N+1}]^T$ to represent sequences.

do-mpc allows to set upper and lower bounds for the states $x_{\text{lb}}, x_{\text{ub}}$, inputs $u_{\text{lb}}, u_{\text{ub}}$ and algebraic states $z_{\text{lb}}, z_{\text{ub}}$. Terminal constraints can be enforced via $g_{\text{terminal}}(\cdot)$ 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.

3.4.3 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.

3.4.3.1 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 $x_0 \rightarrow x_1^2 \rightarrow x_2^4 \rightarrow \cdots \rightarrow x_5^4$. 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:

$$x_{k+1}^j = f(x_k^{p(j)}, u_k^j, z_k^{p(j)}, p_k^{r(j)}, p_{\text{tv},k}),$$

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

3.4.3.2 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_{robust} is introduced, which can be viewed as a tuning parameter. Branching is then only applied for the first N_{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:

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

This results in $N_{\text{s}} = 9$ 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: It 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.

3.4.3.3 Mathematical formulation

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

$$\min_{x_k^j, u_k^j, z_k^j \ \forall (j,k) \in I} \sum_{j=1}^{N_\mathrm{s}} \omega_i J_j(\mathbf{x}_{0:N+1}^j, \mathbf{u}_{0:N}^j, \mathbf{z}_{0:N}^j)$$

subject to:

$$x_0 = \hat{x}_0$$

$$x_{k+1}^j = f(x_k^{p(j)}, u_k^j, z_k^{p(j)}, p_k^{r(j)}, p_{\mathrm{tv},k}) \quad \forall (j,k) \in I$$

$$u_k^i = u_k^j \text{ if } x_k^{p(i)} = x_k^{p(j)}, \quad \forall (i,k),(j,k) \in I$$

$$g(x_k^{p(j)}, u_k^j, z_k^{p(j)}, p_k^{r(j)}, p_{\mathrm{tv},k}) \leq 0 \quad \forall (j,k) \in I$$

$$x_{\mathrm{lb}} \leq x_k^j \leq x_{\mathrm{ub}} \quad \forall (j,k) \in I$$

$$u_{\mathrm{lb}} \leq u_k^j \leq u_{\mathrm{ub}} \quad \forall (j,k) \in I$$

$$z_{\mathrm{lb}} \leq z_k^j \leq z_{\mathrm{ub}} \quad \forall (j,k) \in I$$

$$g_{\mathrm{terminal}}(x_N^j, z_N^j) \leq 0 \quad \forall (j,N) \in I,$$

The objective consists of one term for each scenario, which can be weighted according to the probability of the scenarios ω_j, $j = 1, \ldots, N_\mathrm{s}$. The cost for each scenario J_i is given by:

$$J_j = m(x_{N+1}^j) + \sum_{k=0}^{N} l(x_k^{p(j)}, u_k^j, z_k^{p(j)}, p_k^{r(j)}, p_{\mathrm{tv},k}).$$

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 $p_{\mathrm{min}} \leq p \leq p_{\mathrm{max}}$, 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

tree for nonlinear systems does not guarantee constraint satisfaction for all $p \in [p_{\text{min}}, p_{\text{max}}]$. However, also for nonlinear systems the worst-case scenarios are often at the boundaries of the uncertainty intervals $[p_{\text{min}}, p_{\text{max}}]$. In practice, considering only the extreme values for nonlinear systems provides good results.

Other commonly used robust MPC schemes, such as tube-based MPC, are not currently implemented in **do-mpc** but planned for the near future. Please check our development roadmap on Github for details and updates.

3.5 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.

3.5.1 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

$$\begin{aligned}\dot{x}(t) &= f(x(t), u(t), z(t), p(t), p_{\text{tv}}(t)) + w(t),\\ y(t) &= h(x(t), u(t), z(t), p(t), p_{\text{tv}}(t)) + v(t),\end{aligned}$$

and for the discrete-time case by

$$\begin{aligned}x_{k+1} &= f(x_k, u_k, z_k, p_k, p_{\text{tv},k}) + w_k,\\ y_k &= h(x_k, u_k, z_k, p_k, p_{\text{tv},k}) + v_k.\end{aligned}$$

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 $p_{\text{tv}}(t), p_{\text{tv},k}$ 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.

3.5.2 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.

3.5.2.1 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.

3.5.2.2 Mathematical formulation

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

$$\min_{\substack{\mathbf{x}_{0:N+1},\,\mathbf{u}_{0:N},\,p,\\ \mathbf{w}_{0:N},\,\mathbf{v}_{0:N}}} \frac{1}{2}\|x_0-\tilde{x}_0\|^2_{P_x}+\frac{1}{2}\|p-\tilde{p}\|^2_{P_p}+\sum_{k=0}^{N-1}\left(\frac{1}{2}\|v_k\|^2_{P_{v,k}}+\frac{1}{2}\|w_k\|^2_{P_{w,k}}\right),$$

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

where we introduce the bold letter notation, e.g. $\mathbf{x}_{0:N+1}=[x_0,x_1,\ldots,x_{N+1}]^T$ 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**.

3.6 License

GNU LESSER GENERAL PUBLIC LICENSE Version 3, 29 June 2007

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.

3. Object Code Incorporating Material from Library Header Files.

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.

4. Combined Works.

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.)

5. Combined Libraries.

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.

3.7 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.

3.7.1 Requirements

do-mpc requires the following Python packages and their dependencies:

- numpy
- CasADi
- matplotlib

3.7.2 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.

3.7.3 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.

3.7.4 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.

3.7.4.1 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:

3.7.4.1.1 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`

```
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/home/ffiedler/Documents/coinhsl/lib"
```

6. Install `libgfortran` with Anaconda:

```
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.

3.7.4.2 Option 2: Compile from source

Please see the comprehensive guide on the CasADi Github Wiki.

3.8 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:

> S. Lucia, A. Tatulea-Codrean, C. Schoppmeyer, and S. Engell. Rapid development of modular and sustainable nonlinear model predictive control solutions. Control Engineering Practice, 60:51-62, 2017

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

3.9 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*)

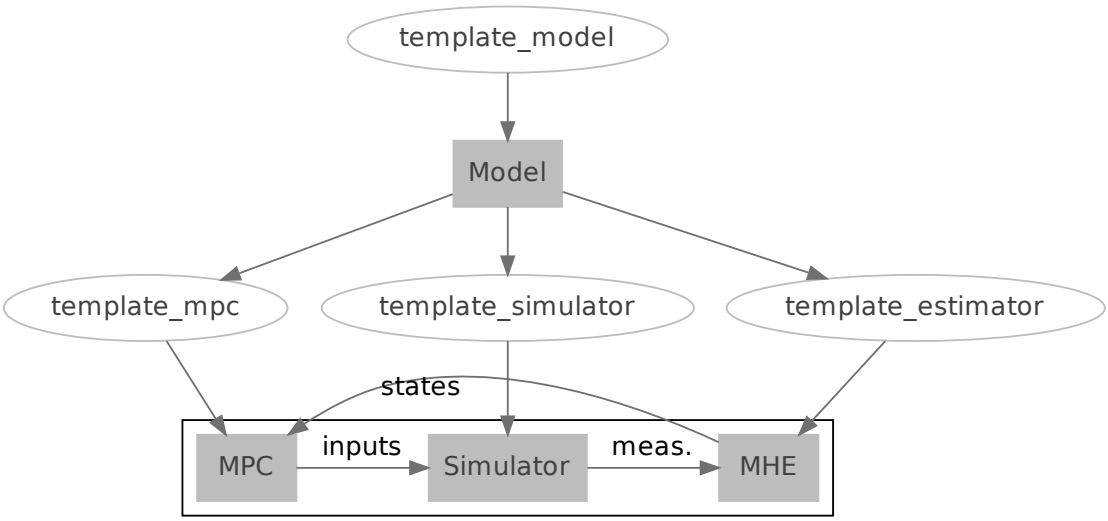


Fig. 3: Project structure

We split our MHE / MPC configuration into five separate files:

`template_model.py`	Define the dynamic model
`template_mpc.py`	Configure the MPC controller
`template_simulator.py`	Configure the DAE/ODE/discrete simulator
`template_estimator.py`	Configure the estimator (MHE / EKF / state-feedback)
`main.py`	**Obtain all configured modules and run the loop.**

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.

3.9.1 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:

```
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)

(continued from previous 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
```

3.9.2 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:

$$\min_{x,u,z} \sum_{k=0}^{N} \left(\underbrace{l(x_k, u_k, z_k, p)}_{\text{lagrange term}} + \underbrace{\Delta u_k^T R \Delta u_k}_{\text{r-term}} \right) + \underbrace{m(x_{N+1})}_{\text{meyer term}}$$

subject to:

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

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

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

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

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

```
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
```

(continues on next page)

(continued from previous page)

```
    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
```

3.9.3 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:

```
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
```

3.9.4 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:

```
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,
```

(continues on next page)

(continued from previous page)

```
        't_step': 0.1,
        'meas_from_data': True,
    }
    mhe.set_param(**setup_mhe)

    # Set custom objective function
    # based on:
    y_meas = mhe._y_meas
    y_calc = mhe._y_calc

    # and (for the arrival cost):
    x_0 = mhe._x
    x_prev = mhe._x_prev

    ...
    mhe.set_objective(...)

    # Set bounds for states, parameters, etc.
    mhe.bounds[...] = ...

    # [Optional] Set measurement function.
    # Measurements are read from data object by default.

    mhe.setup()

    return mhe
```

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:

$$J = \underbrace{(x_0 - \tilde{x}_0)^T P_x (x_0 - \tilde{x}_0)}_{\text{arrival cost states}} + \underbrace{(p_0 - \tilde{p}_0)^T P_p (p_0 - \tilde{p}_0)}_{\text{arrival cost params.}} + \sum_{k=0}^{n-1} \underbrace{(h(x_k, u_k, p_k) - y_k)^T P_{y,k} (h(x_k, u_k, p_k) - y_k)}_{\text{stage cost}}$$

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.

3.9.5 main script

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

```
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.

3.9.5.1 Initial state & guess

Afterwards we set the initial state (and guess for MPC/MHE) for all objects. Note that in proper investigations we usually have a different initial state for the `simulator` (true state) and e.g. the estimator.

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

mpc.set_initial_state(x0, reset_history=True)
simulator.set_initial_state(x0, reset_history=True)
```

The initial guess is automatically set with *do_mpc.controller.MPC.set_initial_state()* as can be seen in the documentation.

3.9.5.2 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:

```
# 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:

```
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.

3.9.5.3 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:

```
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:

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

or have some checks active:

```
while mpc._x0['C_b'] <= 0.8:
    ...
```

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

```
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:

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

3.10 Debugging

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

3.10.1 Feasibility problems

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.

3.10.1.1 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.

3.10.1.2 Which constraints are violated?

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

```
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:

```
opt_labels = mpc.opt_x.labels()
labels_lb_viol =np.array(opt_labels)[np.where(lb_viol)[0]]
labels_ub_viol =np.array(opt_labels)[np.where(lb_viol)[0]]
```

The arrays `labels_lb_viol` and `labels_ub_viol` indicate which variables are problematic.

3.10.1.3 Use soft-constraints.

Some control problems, especially with economic objective will lead to trajectories operating close to (some) constraints. 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):

```
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*

3.11 API Reference

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

Note the following important inheritance of **do-mpc** classes:

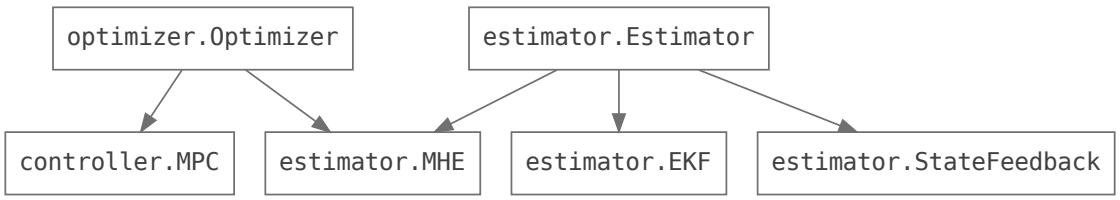


Fig. 4: Class inheritance. Click on classes for more information.

Modules

model
simulator
optimizer
controller
estimator
data
graphics

3.11.1 model

Classes

IteratedVariables	Class to initiate properties and attributes for iterated variables.
Model	The **do-mpc** model class.

3.11.1.1 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

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

3.11.1.1.1 t0

Class attribute.

IteratedVariables.**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.

3.11.1.1.2 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:

```
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
```

Usefull CasADi symbolic structure methods:

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

- `.labels()`

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

3.11.1.1.3 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:

```
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
```

Usefull CasADi symbolic structure methods:

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

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

3.11.1.1.4 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:

```
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
```

(continues on next page)

(continued from previous page)

```
mpc.z0['temperature']     # all elements of variable
mpc.z0['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.

Methods

—

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

3.11.1.2 Model

class do_mpc.model.**Model**(*model_type=None*)

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 `continous` model consists of an underlying ordinary differential equation (ODE) or differential algebraic equation (DAE):

$$\begin{aligned} \dot{x}(t) &= f(x(t), u(t), z(t), p(t), p_{\text{tv}}(t)) + w(t), \\ 0 &= g(x(t), u(t), z(t), p(t), p_{\text{tv}}(t)) \\ y &= h(x(t), u(t), z(t), p(t), p_{\text{tv}}(t)) + v(t) \end{aligned}$$

whereas a `discrete` model consists of a difference equation:

$$\begin{aligned} x_{k+1} &= f(x_k, u_k, z_k, p_k, p_{\text{tv},k}) + w_k, \\ 0 &= g(x_k, u_k, z_k, p_k, p_{\text{tv},k}) \\ y_k &= h(x_k, u_k, z_k, p_k, p_{\text{tv},k}) + v_k \end{aligned}$$

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 seperate file for the MPC configuration.

Parameters **model_type** – Set if the model is `discrete` or `continuous`.

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).

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

This also allows to retrieve multiple variables simultaneously:

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

Attributes

Model.aux	Auxiliary expressions.
Model.p	Static parameters.
Model.tvp	Time-varying parameters.
Model.u	Inputs.
Model.v	Measurement noise.
Model.w	Process noise.
Model.x	Dynamic states.
Model.y	Measurements.
Model.z	Algebraic states.

3.11.1.2.1 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:

```
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
```

Usefull CasADi symbolic structure methods:

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

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

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

3.11.1.2.2 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:

```
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
```

Usefull CasADi symbolic structure methods:

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

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

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

3.11.1.2.3 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:

```
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
```

Usefull CasADi symbolic structure methods:

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

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

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

3.11.1.2.4 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:

```
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
```

Usefull CasADi symbolic structure methods:

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

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

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

3.11.1.2.5 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*.

Usefull CasADi symbolic structure methods:

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

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

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

3.11.1.2.6 w

Class attribute.

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

The process noise structure is created automatically, whenever the *Model.set_rhs()* method is called with the argument `process_noise = True`.

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

Usefull CasADi symbolic structure methods:

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

Raises **assertion** – Cannot set w direcly.

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

3.11.1.2.7 x

Class attribute.

`Model.x`
Dynamic 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:

```
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
```

Usefull CasADi symbolic structure methods:

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

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

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

3.11.1.2.8 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:

```
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
```

Usefull CasADi symbolic structure methods:

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

Raises **assertion** – Cannot set model variables direcly. Use set_meas instead.

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

3.11.1.2.9 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:

```
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
```

Usefull CasADi symbolic structure methods:

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

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

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

Methods

Model.set_expression	Introduce new expression to the model class.
Model.set_meas	Introduce new measurable output to the model class.
Model.set_rhs	Formulate the right hand side (rhs) of the ODE:
Model.set_variable	Introduce new variables to the model class.
Model.setup	Setup method must be called to finalize the modelling process.
Model.setup_model	Legacy method.

3.11.1.2.10 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?

```
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.

3.11.1.2.11 set_meas

Class method.

`do_mpc.model.Model.`**set_meas** (*self*, *meas_name*, *expr*, *meas_noise=True*)

Introduce new measurable output to the model class.

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

or in case of discrete dynamics:

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

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:

```
# 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.

3.11.1.2.12 set_rhs

Class method.

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

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

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

or the update equation in case of discrete dynamics:

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

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
- **assertion** – Cannot call after :py:func‘setup‘.

Returns None

Return type None

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

3.11.1.2.13 set_variable

Class method.

`do_mpc.model.Model.`**`set_variable`**(*self*, *var_type*, *var_name*, *shape=(1, 1)*)

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

Example:

```
# 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):

Long name	short name	Remark
states	_x	Required
inputs	_u	optional
algebraic	_z	Optional
parameter	_p	Optional
timevarying_parameter	_tvp	Optional

- **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.

3.11.1.2.14 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

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

3.11.1.2.15 setup_model

Class method.

do_mpc.model.Model.**setup_model**(*self*)
Legacy method.

> **Warning:** The method is depreciated and will be removed in a later version. Please call *setup()* instead.

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.

3.11.2 simulator

Classes

Simulator	A class for simulating systems.

3.11.2.1 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.

3.11.2.1.1 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.

3.11.2.1.2 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:

```
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
```

Usefull CasADi symbolic structure methods:

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

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

3.11.2.1.3 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:

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

...
```

(continues on next page)

(continued from previous page)

```
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
```

Usefull CasADi symbolic structure methods:

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

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

3.11.2.1.4 z0

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:

```
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
```

Usefull CasADi symbolic structure methods:

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

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

Methods

Simulator.get_p_template	Obtain output template for *set_p_fun()*.
Simulator.get_tvp_template	Obtain the output template for *set_tvp_fun()*.

Continued on next page

Table 9 – continued from previous page

Simulator.make_step	Main method of the simulator class during control runtime.
Simulator.reset_history	Reset the history of the simulator.
Simulator.set_initial_state	Set the intial state of the simulator.
Simulator.set_p_fun	Method to set the function which gives the values of the parameters.
Simulator.set_param	Set the parameters for the simulator.
Simulator.set_tvp_fun	Method to set the function which returns the values of the time-varying parameters.
Simulator.setup	Sets up the simulator and finalizes the simulator configuration.
Simulator.simulate	Call the CasADi simulator.

3.11.2.1.5 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.

3.11.2.1.6 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.

3.11.2.1.7 make_step

Class method.

do_mpc.simulator.Simulator.**make_step**(*self*, *u0*, *x0=None*, *z0=None*, *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 but can optionally be supplied. The algebraic states *z0* can also be supplied, if they are defined in the model but are only used as an intial 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.
- **x0** (*numpy.ndarray (optional)*) – Current state of the system.
- **z0** (*numpy.ndarray (optional)*) – Initial guess for current algebraic states
- **v0** (*numpy.ndarray (optional)*) – Additive measurement noise
- **w0** (*numpy.ndarray (optional)*) – Additive process noise

Returns x_nsext

Return type numpy.ndarray

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

3.11.2.1.8 reset_history

Class method.

do_mpc.simulator.Simulator.**reset_history**(*self*)
Reset the history of the simulator.

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

3.11.2.1.9 set_initial_state

Class method.

do_mpc.simulator.Simulator.**set_initial_state**(*self*, *x0*, *reset_history=False*)
Set the intial state of the simulator. Optionally resets the history. The history is empty upon creation of the simulator.

> **Warning:** This method is depreciated. Use the *x0* property of the class to set the intial values instead.

Parameters

- **x0** (*numpy array*) – Initial state
- **reset_history** (*bool (optional)*) – Resets the history of the simulator, defaults to False

Returns None

Return type None

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

3.11.2.1.10 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:

```
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 evaluted 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:

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

The parameter function can look something like this:

```
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:

```
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.

3.11.2.1.11 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.

3.11.2.1.12 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 evaluted 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.

3.11.2.1.13 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.

3.11.2.1.14 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_x_num['_x']`
- algebraic states `sim_x_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.

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

3.11.3 optimizer

Classes

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

3.11.3.1 Optimizer

class do_mpc.optimizer.**Optimizer**

The base clase 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

Optimizer.bounds	Query and set bounds of the optimization variables.
Optimizer.scaling	Query and set scaling of the optimization variables.

3.11.3.1.1 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 seperated by comas) must contain atleast the following elements:

order	index name	valid options
1	bound type	`lower` and `upper`
2	variable type	`_x`, `_u` and `_z` (and `_p_est` for MHE)
3	variable name	Names defined in *do_mpc.model.Model*.

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

Example:

```
# 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.

3.11.3.1.2 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 seperated by comas) must contain atleast the following elements:

order	index name	valid options
1	variable type	`_x`, `_u` and `_z` (and `_p_est` for MHE)
2	variable name	Names defined in *`do_mpc.model.Model`*.

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

Example:

```
# Set with:
optimizer.scaling['_x', 'phi_1'] = 2
optimizer.scaling['_x', 'phi_2'] = 2

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

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.

Methods

`Optimizer.get_tvp_template`	Obtain output template for *`set_tvp_fun()`*.
`Optimizer.reset_history`	Reset the history of the optimizer.
`Optimizer.set_initial_state`	Set the intial state of the optimizer.
`Optimizer.set_nl_cons`	Introduce new constraint to the class.
`Optimizer.set_tvp_fun`	Set function which returns time-varying parameters.
`Optimizer.solve`	Solves the optmization problem.

3.11.3.1.3 get_tvp_template

Class method.

do_mpc.optimizer.Optimizer.**get_tvp_template**(*self*)

Obtain output template for *set_tvp_fun()*.

The method returns a structured object with `n_horizon` 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:

```
# 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.

3.11.3.1.4 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.

3.11.3.1.5 set_initial_state

Class method.

do_mpc.optimizer.Optimizer.**set_initial_state**(*self*, *x0*, *p_est0=None*, *reset_history=False*, *set_intial_guess=True*)

Set the intial state of the optimizer. Optionally resets the history. The history is empty upon creation of the optimizer.

Optionally update the initial guess. The initial guess is first created with the `.setup()` method (MHE/MPC) and uses the class attributes `x0`, `u0`, `z0` for all time instances, collocation points (if applicable) and scenarios (if applicable). If these values were not explicitly set by the user, they default to all zeros.

Warning: This method is depreciated. Use the *x0* (and *p_est0* for MHE) property of the class to set the intial values instead.

Parameters

- **x0** (*numpy array*) – Initial state
- **reset_history** (*bool (,optional)*) – Resets the history of the optimizer, defaults to False
- **set_intial_guess** (*bool (,optional)*) – Setting the initial state also updates the intial guess for the optimizer.

Returns None

Return type None

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

3.11.3.1.6 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:

$$m(x, u, z, p_{\text{tv}}, p) \leq m_{\text{ub}}$$

Setting the flag `soft_constraint=True` will introduce slack variables ϵ, such that:

$$\begin{aligned} m(x, u, z, p_{\text{tv}}, p) - \epsilon &\leq m_{\text{ub}}, \\ 0 \leq \epsilon &\leq \epsilon_{\text{max}}, \end{aligned}$$

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

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

3.11.3.1.7 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:

```
# 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.

3.11.3.1.8 solve

Class method.

do_mpc.optimizer.Optimizer.**solve**(*self*)

Solves the optmization 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 paramters 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 **asssertion** – Optimizer was not setup yet.

Returns None

Return type 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.

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

3.11.4 controller

Classes

MPC	Model predictive controller.

3.11.4.1 MPC

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 *`do_mpc.optimizer.Optimizer`* base class (which is also used for the *`do_mpc.estimator.MHE`* estimator).

Use this class to configure and run the MPC controller based on a previously configured *`do_mpc.model.Model`* instance.

Configuration and setup:

Configuring and setting up the MPC controller involves the following steps:

1. Use *`set_param()`* to configure the *`MPC`* instance.
2. Set the objective of the control problem with *`set_objective()`* and *`set_rterm()`*
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. Finally, call *`setup()`*.

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 *x0*, *z0* and *u0* 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

MPC.bounds	Query and set bounds of the optimization variables.
MPC.opt_p_num	Full MPC parameter vector.
MPC.opt_x_num	Full MPC solution and initial guess.
MPC.scaling	Query and set scaling of the optimization variables.
MPC.t0	Current time marker of the class.
MPC.u0	Initial input and current iterate.
MPC.x0	Initial state and current iterate.
MPC.z0	Initial algebraic state and current iterate.

3.11.4.1.1 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 seperated by comas) must contain atleast the following elements:

order	index name	valid options
1	bound type	`lower` and `upper`
2	variable type	`_x`, `_u` and `_z` (and `_p_est` for MHE)
3	variable name	Names defined in *do_mpc.model.Model*.

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

Example:

```
# 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.

3.11.4.1.2 opt_p_num

Class attribute.

MPC.**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 *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:

```
# 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]
```

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.

3.11.4.1.3 opt_x_num

Class attribute.

MPC.**opt_x_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:

```
# 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 *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.

3.11.4.1.4 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 seperated by comas) must contain atleast the following elements:

order	index name	valid options
1	variable type	`_x`, `_u` and `_z` (and `_p_est` for MHE)
2	variable name	Names defined in *do_mpc.model.Model*.

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

Example:

```
# Set with:
optimizer.scaling['_x', 'phi_1'] = 2
optimizer.scaling['_x', 'phi_2'] = 2

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

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.

3.11.4.1.5 t0

Class attribute.

MPC.**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.

3.11.4.1.6 u0

Class attribute.

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
```

Usefull CasADi symbolic structure methods:

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

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

3.11.4.1.7 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)
```

(continues on next page)

(continued from previous page)

```
# 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
```

Usefull CasADi symbolic structure methods:

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

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

3.11.4.1.8 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:

```
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
```

Usefull CasADi symbolic structure methods:

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

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

Methods

MPC.get_p_template	Obtain output template for *set_p_fun()*.
MPC.get_tvp_template	Obtain output template for *set_tvp_fun()*.
MPC.make_step	Main method of the class during runtime.
MPC.reset_history	Reset the history of the optimizer.
MPC.set_initial_guess	Initial guess for optimization variables.
MPC.set_initial_state	Set the intial state of the optimizer.

Continued on next page

Table 15 – continued from previous page

MPC.set_nl_cons	Introduce new constraint to the class.
MPC.set_objective	Sets the objective of the optimal control problem (OCP).
MPC.set_p_fun	Set function which returns parameters.
MPC.set_param	Set the parameters of the *MPC* class.
MPC.set_rterm	Set the penality factor for the inputs.
MPC.set_tvp_fun	Set function which returns time-varying parameters.
MPC.set_uncertainty_values	Define scenarios for the uncertain parameters.
MPC.setup	Setup the MPC class.
MPC.solve	Solves the optmization problem.

3.11.4.1.9 get_p_template

Class method.

do_mpc.controller.MPC.**get_p_template**(*self*, *n_combinations*)

Obtain output template for *set_p_fun()*.

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 returns a structured object which is initialized with all zeros. Use this object to define values of the parameters for an arbitrary number of scenarios (defined by `n_combinations`).

This structure (with numerical values) should be used as the output of the `p_fun` function which is set to the class with *set_p_fun()*.

Use the combination of *get_p_template()* and `set_p_template()` as a more adaptable alternative to *set_uncertainty_values()*.

Note: We advice less experienced users to use *set_uncertainty_values()* as an alterntive way to configure the scenario-tree for robust multi-stage MPC.

Example:

```
# 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 **n_combinations** (*int*) – Define the number of combinations for the uncertain parameters for robust MPC.

Returns None

Return type None

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

3.11.4.1.10 get_tvp_template

Class method.

do_mpc.controller.MPC.**get_tvp_template**(*self*)

Obtain output template for *set_tvp_fun()*.

The method returns a structured object with n_horizon 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:

```
# 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.

3.11.4.1.11 make_step

Class method.

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.

3.11.4.1.12 reset_history

Class method.

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.

3.11.4.1.13 set_initial_guess

Class method.

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.

3.11.4.1.14 set_initial_state

Class method.

do_mpc.controller.MPC.**set_initial_state**(*self*, *x0*, *p_est0=None*, *reset_history=False*, *set_intial_guess=True*)

Set the intial state of the optimizer. Optionally resets the history. The history is empty upon creation of the optimizer.

Optionally update the initial guess. The initial guess is first created with the `.setup()` method (MHE/MPC) and uses the class attributes `x0`, `u0`, `z0` for all time instances, collocation points (if applicable) and scenarios (if applicable). If these values were not explicitly set by the user, they default to all zeros.

Warning: This method is depreciated. Use the *x0* (and *p_est0* for MHE) property of the class to set the intial values instead.

Parameters

- **x0** (*numpy array*) – Initial state
- **reset_history** (*bool (,optional)*) – Resets the history of the optimizer, defaults to False
- **set_intial_guess** (*bool (,optional)*) – Setting the initial state also updates the intial guess for the optimizer.

Returns None

Return type None

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

3.11.4.1.15 set_nl_cons

Class method.

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:

$$m(x, u, z, p_{\text{tv}}, p) \leq m_{\text{ub}}$$

Setting the flag soft_constraint=True will introduce slack variables ϵ, such that:

$$\begin{aligned} m(x, u, z, p_{\text{tv}}, p) - \epsilon &\leq m_{\text{ub}}, \\ 0 \leq \epsilon &\leq \epsilon_{\text{max}}, \end{aligned}$$

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

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

3.11.4.1.16 set_objective

Class method.

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:

$$J(x,u,z) = \sum_{k=0}^{N} \left(\underbrace{l(x_k, z_k, u_k, p_k, p_{\text{tv},k})}_{\text{lagrange term}} + \underbrace{\Delta u_k^T R \Delta u_k}_{\text{r-term}} \right) + \underbrace{m(x_{N+1})}_{\text{meyer term}}$$

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 $l(x_k, z_k, u_k, p_k, p_{\text{tv},k})$ (`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`

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.

3.11.4.1.17 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 advice less experienced users to use *set_uncertainty_values()* as an alterntive way to configure the scenario-tree for robust multi-stage MPC.

Example:

```
# 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

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

3.11.4.1.18 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:

```
mpc.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:

```
setup_mpc = {
    'n_horizon': 20,
    't_step': 0.5,
}
mpc.set_param(**setup_mpc)
```

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.
- **state_discretization** (*str*) – Choose the state discretization for continuous models. Currently only `'collocation'` is available. Defaults to `'collocation'`.
- **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.
- **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','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:

```
MPC.set_param(nlpsol_opts = {'ipopt.linear_solver': 'MA27'})
```

Note: To surpress the output of IPOPT, please use:

```
surpress_ipopt = {'ipopt.print_level':0, 'ipopt.sb': 'yes', 'print_time':0}
MPC.set_param(nlpsol_opts = surpress_ipopt)
```

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

3.11.4.1.19 set_rterm

Class method.

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 $i \in \mathbb{I}$, where $\mathbb{I}$ 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:

$$\sum_{k=0}^{N} \sum_{i \in \mathbb{I}} r_i \Delta u_{k,i}^2,$$

the weighted squared cost to the MPC objective function.

Example:

```
# 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.

3.11.4.1.20 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:

```
# 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])
```

(continues on next page)

(continued from previous page)

```
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.

3.11.4.1.21 set_uncertainty_values

Class method.

`do_mpc.controller.MPC.`**`set_uncertainty_values`**(*self*, *uncertainty_values=None*, ***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:

```
# 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 imporant for the MPC controller (e.g. MHE tuning matrices) can be ignored with the new interface (see `gamma` in the example above).

Legacy interface: Pass a list of arrays for the uncertain parameters. This list must have the same number of elements as uncertain parameters in the model definition. The first element is the nominal case. Each list element can be an array or list of possible values for the respective parameter. Note that the order of elements determine the assignment.

Example:

```
# 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:
alpha_var = np.array([1., 0.9, 1.1])
beta_var = np.array([1., 1.05])
MPC.set_uncertainty_values([alpha_var, beta_var])
```

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.
- **uncertainty_values** (*list*) – (Depreciated) List of lists / numpy arrays with the same number of elements as number of parameters in model.

Returns None

Return type None

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

3.11.4.1.22 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.

3.11.4.1.23 solve

Class method.

do_mpc.controller.MPC.**solve**(*self*)

Solves the optmization 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 paramters 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 **asssertion** – Optimizer was not setup yet.

Returns None

Return type 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.

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

3.11.5 estimator

Classes

EKF	Extended Kalman Filter.
Estimator	The Estimator base class.
MHE	Moving horizon estimator.
StateFeedback	Simple state-feedback "estimator".

3.11.5.1 EKF

class do_mpc.estimator.**EKF**(*model*)

Extended Kalman Filter. Setup this class and use *EKF.make_step()* during runtime to obtain the currently estimated states given the measurements `y0`.

> **Warning:** Not currently implemented.

Attributes

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

3.11.5.1.1 t0

Class attribute.

EKF.**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.

3.11.5.1.2 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:

```
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
```

Usefull CasADi symbolic structure methods:

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

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

3.11.5.1.3 x0

Class attribute.

EKF.**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
```

Usefull CasADi symbolic structure methods:

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

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

3.11.5.1.4 z0

Class attribute.

EKF.**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:

```
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
```

Usefull CasADi symbolic structure methods:

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

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

Methods

EKF.make_step	Main method during runtime.
EKF.reset_history	Reset the history of the estimator
EKF.set_initial_state	Set the intial state of the estimator.

3.11.5.1.5 make_step

Class method.

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.

3.11.5.1.6 reset_history

Class method.

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.

3.11.5.1.7 set_initial_state

Class method.

do_mpc.estimator.EKF.**set_initial_state**(*self*, *x0*, *reset_history=False*)
Set the intial state of the estimator. Optionally resets the history. The history is empty upon creation of the estimator. This method is overwritten for the MHE from *do_mpc.optimizer.Optimizer*.

> **Warning:** This method is depreciated. Use the *x0* property of the class to set the intial state instead.

Parameters

- **x0** (*numpy array*) – Initial state
- **reset_history** (*bool (,optional)*) – Resets the history of the estimator, defaults to False

Returns None

Return type 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.

3.11.5.2 Estimator

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

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

3.11.5.2.1 t0

Class attribute.

Estimator.**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.

3.11.5.2.2 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:

```
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
```

Usefull CasADi symbolic structure methods:

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

- `.labels()`

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

3.11.5.2.3 x0

Class attribute.

`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.

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
```

Usefull CasADi symbolic structure methods:

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

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

3.11.5.2.4 z0

Class attribute.

`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.

Example:

```
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
```

(continues on next page)

(continued from previous page)

```
mpc.z0['temperature']     # all elements of variable
mpc.z0['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.

Methods

Estimator.reset_history	Reset the history of the estimator
Estimator.set_initial_state	Set the intial state of the estimator.

3.11.5.2.5 reset_history

Class method.

do_mpc.estimator.Estimator.**reset_history**(*self*)
Reset the history of the estimator

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

3.11.5.2.6 set_initial_state

Class method.

do_mpc.estimator.Estimator.**set_initial_state**(*self*, *x0*, *reset_history=False*)
Set the intial state of the estimator. Optionally resets the history. The history is empty upon creation of the estimator. This method is overwritten for the MHE from *do_mpc.optimizer.Optimizer*.

> **Warning:** This method is depreciated. Use the *x0* property of the class to set the intial state instead.

Parameters

- **x0** (*numpy array*) – Initial state
- **reset_history** (*bool (,optional)*) – Resets the history of the estimator, defaults to False

Returns None

Return type 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.

3.11.5.3 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.

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()*.

5. Set upper and lower bounds.
6. Optionally, set further (non-linear) constraints with *set_nl_cons()*.
7. Use *get_p_template()* and *set_p_fun()* to set the function for the parameters.
8. Finally, call *setup()*.

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 intial values of *x0*, *z0*, *u0* 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.opt_p_num	Full MHE parameter vector.

Continued on next page

Table 21 – continued from previous page

MHE.opt_x_num	Full MHE solution and initial guess.
MHE.p_est0	Initial value of estimated parameters and current iterate.
MHE.scaling	Query and set scaling of the optimization variables.
MHE.t0	Current time marker of the class.
MHE.u0	Initial input and current iterate.
MHE.x0	Initial state and current iterate.
MHE.z0	Initial algebraic state and current iterate.

3.11.5.3.1 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 seperated by comas) must contain atleast the following elements:

order	index name	valid options
1	bound type	`lower` and `upper`
2	variable type	`_x`, `_u` and `_z` (and `_p_est` for MHE)
3	variable name	Names defined in *do_mpc.model.Model*.

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

Example:

```
# 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.

3.11.5.3.2 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:

```
# 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.

3.11.5.3.3 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:

```
# 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.

3.11.5.3.4 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:

```
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.

3.11.5.3.5 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 comas) must contain atleast the following elements:

order	index name	valid options
1	variable type	`_x`, `_u` and `_z` (and `_p_est` for MHE)
2	variable name	Names defined in *do_mpc.model.Model*.

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

Example:

```
# Set with:
optimizer.scaling['_x', 'phi_1'] = 2
optimizer.scaling['_x', 'phi_2'] = 2

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

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.

3.11.5.3.6 t0

Class attribute.

`MHE.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.

3.11.5.3.7 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:

```
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
```

Usefull CasADi symbolic structure methods:

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

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

3.11.5.3.8 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:

```
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
```

Usefull CasADi symbolic structure methods:

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

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

3.11.5.3.9 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:

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

...
mhe = do_mpc.estimator.MHE(model)
# or
```

(continues on next page)

(continued from previous page)

```
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
```

Usefull CasADi symbolic structure methods:

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

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

Methods

MHE.get_p_template	Obtain output template for *set_p_fun()*.
MHE.get_tvp_template	Obtain output template for *set_tvp_fun()*.
MHE.get_y_template	Obtain output template for *set_y_fun()*.
MHE.make_step	Main method of the class during runtime.
MHE.reset_history	Reset the history of the optimizer.
MHE.set_default_objective	Configure the suggested default MHE formulation.
MHE.set_initial_guess	Initial guess for optimization variables.
MHE.set_initial_state	Set the intial state of the optimizer.
MHE.set_nl_cons	Introduce new constraint to the class.
MHE.set_objective	Set the stage cost $l(\cdot)$ and arrival cost $m(\cdot)$ function for the MHE problem:
MHE.set_p_fun	Set function which returns parameters..
MHE.set_param	Method to set the parameters of the *MHE* class.
MHE.set_tvp_fun	Set function which returns time-varying parameters.
MHE.set_y_fun	Set the measurement function.
MHE.setup	The setup method finalizes the MHE creation.
MHE.solve	Solves the optmization problem.

3.11.5.3.10 get_p_template

Class method.

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* 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.

3.11.5.3.11 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` 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:

```
# 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.

3.11.5.3.12 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.:
y_template['y_meas', :, 'meas_name']
```

where `k` runs from `0` to `N_horizon` and `meas_name` refers to the user-defined names in *do_mpc.model.Model*.

Note: The structure is ordered, sucht that `k=0` is the "oldest measurement" and `k=N_horizon` is the newest measurement.

By default, the following measurement function is choosen:

```
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 `MHE.data` object.

Returns y_template

Return type struct_symSX

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

3.11.5.3.13 make_step

Class method.

`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 `y0`.

The method prepares the MHE by setting the current parameters, calls *solve()* and updates the *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.

3.11.5.3.14 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.

3.11.5.3.15 set_default_objective

Class method.

do_mpc.estimator.MHE.**set_default_objective**(*self*, *P_x*, *P_v=None*, *P_p=None*, *P_w=None*, *P_y=None*)

Configure the suggested default MHE formulation.

Use this method to pass tuning matrices for the MHE optimization problem:

$$\min_{\substack{\mathbf{x}_{0:N+1},\,\mathbf{u}_{0:N},\,p,\\ \mathbf{w}_{0:N},\,\mathbf{v}_{0:N}}} \quad m(x_0,\tilde{x}_0,p,\tilde{p})+\sum_{k=0}^{N-1} l(v_k,w_k,p,p_{\text{tv},k}),$$

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

where we introduce the bold letter notation, e.g. $\mathbf{x}_{0:N+1}=[x_0,x_1,\ldots,x_{N+1}]^T$ to represent sequences and where $\|x\|_P^2=x^TPx$ 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 (_tvp) for P_v and P_w and parameters (_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 $l \times l$ ($p_{\text{est}} \in \mathbb{R}^l$))
- **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.

3.11.5.3.16 set_initial_guess

Class method.

do_mpc.estimator.MHE.**set_initial_guess**(*self*)

Initial guess for optimization variables. Uses the current class attributes *x0*, *z0* and *u0*, *p_est0* 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 *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.

3.11.5.3.17 set_initial_state

Class method.

do_mpc.estimator.MHE.**set_initial_state**(*self*, *x0*, *p_est0=None*, *reset_history=False*, *set_intial_guess=True*)

Set the intial state of the optimizer. Optionally resets the history. The history is empty upon creation of the optimizer.

Optionally update the initial guess. The initial guess is first created with the `.setup()` method (MHE/MPC) and uses the class attributes `x0`, `u0`, `z0` for all time instances, collocation points (if applicable) and scenarios (if applicable). If these values were not explicitly set by the user, they default to all zeros.

> **Warning:** This method is depreciated. Use the *x0* (and *p_est0* for MHE) property of the class to set the intial values instead.

Parameters

- **x0** (*numpy array*) – Initial state
- **reset_history** (*bool (,optional)*) – Resets the history of the optimizer, defaults to False

- **set_intial_guess** (*bool (,optional)*) – Setting the initial state also updates the intial guess for the optimizer.

Returns None

Return type None

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

3.11.5.3.18 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:

$$m(x, u, z, p_{\text{tv}}, p) \leq m_{\text{ub}}$$

Setting the flag soft_constraint=True will introduce slack variables ϵ, such that:

$$\begin{aligned} m(x, u, z, p_{\text{tv}}, p) - \epsilon &\leq m_{\text{ub}}, \\ 0 \leq \epsilon &\leq \epsilon_{\text{max}}, \end{aligned}$$

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

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

3.11.5.3.19 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:

$$\min_{\substack{\mathbf{x}_{0:N+1},\,\mathbf{u}_{0:N},\,p,\\ \mathbf{w}_{0:N},\,\mathbf{v}_{0:N}}} \quad m(x_0,\tilde{x}_0,p,\tilde{p})+\sum_{k=0}^{N-1} l(v_k,w_k,p,p_{\text{tv},k}),$$

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

Use the class attributes:

- `mhe._w` as w_k
- `mhe._v` as v_k
- `mhe._x_prev` as $\tilde{x}_0$
- `mhe._x` as x_0
- `mhe._p_est_prev` as $\tilde{p}_0$
- `mhe._p_est` as p_0

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.:

```
x1 = mhe._x['state_1']
```

For a vector of all states, use the `.cat` method as shown in the example below.

Example:

```
# 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.

3.11.5.3.20 set_p_fun

Class method.

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.

3.11.5.3.21 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:

```
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:

```
setup_mhe = {
    'n_horizon': 20,
    't_step': 0.5,
}
mhe.set_param(**setup_mhe)
```

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'.
- **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.
- **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','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:

```
optimizer.set_param(nlpsol_opts = {'ipopt.linear_solver': 'MA27'})
```

Note: To surpress the output of IPOPT, please use:

```
surpress_ipopt = {'ipopt.print_level':0, 'ipopt.sb': 'yes', 'print_time':0}
optimizer.set_param(nlpsol_opts = surpress_ipopt)
```

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

3.11.5.3.22 set_tvp_fun

Class method.

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:

```
# 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.

3.11.5.3.23 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.

3.11.5.3.24 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.

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

3.11.5.3.25 solve

Class method.

do_mpc.estimator.MHE.**solve**(*self*)

Solves the optmization 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 paramters 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 **asssertion** – Optimizer was not setup yet.

Returns None

Return type 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.

3.11.5.4 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

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

3.11.5.4.1 t0

Class attribute.

StateFeedback.**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.

3.11.5.4.2 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:

```
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
```

Usefull CasADi symbolic structure methods:

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

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

3.11.5.4.3 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:

```
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
```

Usefull CasADi symbolic structure methods:

- `.shape`

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

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

3.11.5.4.4 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:

```
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
```

Usefull CasADi symbolic structure methods:

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

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

Methods

StateFeedback.make_step	Return the measurement `y0`.
StateFeedback.reset_history	Reset the history of the estimator
StateFeedback.set_initial_state	Set the intial state of the estimator.

3.11.5.4.5 make_step

Class method.

`do_mpc.estimator.StateFeedback.`**`make_step`**(*self*, *y0*)

Return the measurement `y0`.

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

3.11.5.4.6 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.

3.11.5.4.7 set_initial_state

Class method.

do_mpc.estimator.StateFeedback.**set_initial_state**(*self*, *x0*, *reset_history=False*)
Set the intial state of the estimator. Optionally resets the history. The history is empty upon creation of the estimator. This method is overwritten for the MHE from *do_mpc.optimizer.Optimizer*.

> **Warning:** This method is depreciated. Use the *x0* property of the class to set the intial state instead.

Parameters

- **x0** (*numpy array*) – Initial state
- **reset_history** (*bool (,optional)*) – Resets the history of the estimator, defaults to False

Returns None

Return type 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.

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

3.11.6 data

Classes

Data	**do-mpc** data container.
MPCData	**do-mpc** data container for the *do_mpc.controller.MPC* instance.

3.11.6.1 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
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_S']                # optimizer runtime
# These do not allow further indices.
```

Returns Returns the queried data field (for all time instances)

Return type numpy.ndarray

Methods

Data.export	The export method returns a dictionary of the stored data.
Data.init_storage	Create new (empty) arrays for all variables.
Data.set_meta	Set meta data for the current instance of the data object.
Data.update	Update value(s) of the data structure with key word arguments.

3.11.6.1.1 export

Class method.

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.

3.11.6.1.2 init_storage

Class method.

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.

3.11.6.1.3 set_meta

Class method.

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.

3.11.6.1.4 update

Class method.

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:

```
_x = np.ones((1, 3))
_u = np.ones((1, 2))
data.update('_x': _x, '_u': _u)

or:
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.

3.11.6.2 MPCData

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.

__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
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_S']               # optimizer runtime
# These do not allow further indices.
```

Returns Returns the queried data field (for all time instances)

Return type numpy.ndarray

Methods

MPCData.export	The export method returns a dictionary of the stored data.
MPCData.init_storage	Create new (empty) arrays for all variables.
MPCData.prediction	Query the MPC trajectories.
MPCData.set_meta	Set meta data for the current instance of the data object.
MPCData.update	Update value(s) of the data structure with key word arguments.

3.11.6.2.1 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.

3.11.6.2.2 init_storage

Class method.

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.

3.11.6.2.3 prediction

Class method.

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.:

```
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:

```
arr = data.prediction(('_x', 'x_1'))
arr.shape
>> (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.

3.11.6.2.4 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.

3.11.6.2.5 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 objective. See self.data_fields for a complete list of data fields.

Example:

```
_x = np.ones((1, 3))
_u = np.ones((1, 2))
data.update('_x': _x, '_u': _u)

or:
data.update('_x': _x)
data.update('_u': _u)

Alternatively:
data_dict = {
    '_x':np.ones((1, 3)),
    '_u':np.ones((1, 2))
```

(continues on next page)

(continued from previous page)

```
}

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

load_results	Simple wrapper to open and unpickle a file.
save_results	Exports the data objects from the **do-mpc** modules in `save_list` as a pickled file.

3.11.6.3 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.

3.11.6.4 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** – results_path must be a string.
- **assertion** – overwrite must be boolean.
- **Exception** – save_list contains object which is neither do_mpc simulator, optimizer nor estimator.

Returns None

Return type None

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

3.11.7 graphics

Classes

Graphics	Graphics module to present the results of **do-mpc**.

3.11.7.1 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 an *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.:

```
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])
```

(continues on next page)

(continued from previous page)

```
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 developement process.

Animations can be setup with the follwing loop:

```
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

Graphics.pred_lines	Structure that holds the prediction line objects.
Graphics.result_lines	Structure that holds the result line objects.

3.11.7.1.1 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:

```
# 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 obects

Returns List of line objects.

Return type list

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

3.11.7.1.2 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:

```
# 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 obects

Returns List of line objects.

Return type list

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

Methods

Graphics.add_line	add_line is called during setting up the *Graphics* class.
Graphics.clear	Clears all data from lines.
Graphics.plot_predictions	Plots the predicted trajectories for the plot configuration.
Graphics.plot_results	Plots the results stored in the data object.
Graphics.reset_axes	Relimits and scales all axes.
Graphics.reset_prop_cycle	Resets the property cycle for all axes which were passed with *Graphics.add_line()*.

3.11.7.1.3 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 .plot 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.

3.11.7.1.4 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.

3.11.7.1.5 plot_predictions

Class method.

do_mpc.graphics.Graphics.**plot_predictions**(*self*, *t_ind=-1*, ***pltkwargs*)
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.

3.11.7.1.6 plot_results

Class method.

do_mpc.graphics.Graphics.**plot_results**(*self*, *t_ind=-1*, ***pltkwargs*)
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.

3.11.7.1.7 reset_axes

Class method.

do_mpc.graphics.Graphics.**reset_axes**(*self*)

Relimits and scales all axes. This method calls

```
ax.relim()
ax.autoscale()
```

on all axes instances in the class.

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

3.11.7.1.8 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.

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

Functions

animate	Animation helper function.
default_plot	Pass a *do_mpc.data.Data* object and create a default **do-mpc** plot.

3.11.7.2 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

3.11.7.3 default_plot

do_mpc.graphics.**default_plot**(*data*, *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.
- **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.

3.12 Release notes

This content is autogenereated from our Github release notes.

3.12.1 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!

3.12.1.1 Major changes

3.12.1.1.1 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.

3.12.1.1.2 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:

$$\min_{\substack{\mathbf{x}_{0:N},\mathbf{u}_{0:N-1},p,\\ \mathbf{w}_{0:N-1},\mathbf{v}_{0:N-1}}} \frac{1}{2}\|x_0-\tilde{x}_0\|^2_{P_x}+\frac{1}{2}\|p-\tilde{p}\|^2_{P_p}+\sum_{k=0}^{N-1}\left(\frac{1}{2}\|v_k\|^2_{P_{v,k}}+\frac{1}{2}\|w_k\|^2_{P_{w,k}}\right),$$

$$\text{s.t.}\quad \left.\begin{aligned} x_{k+1} &= f(x_k,u_k,z_k,p,p_{\text{tv},k})+w_k,\\ y_k &= h(x_k,u_k,z_k,p,p_{\text{tv},k})+v_k,\\ g(x_k,u_k,z_k,p_k,p_{\text{tv},k}) &\leq 0 \end{aligned}\right\} k=0,\ldots,N-1$$

image

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.

3.12.1.1.3 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..

3.12.1.2 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.

3.12.1.3 Minor changes

- Renamed `model.setup_model()` -> `model.setup()` in all examples. This adresses #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.

3.12.1.4 Example files

Please download the example files for release do-mpc v4.0.0 here.

3.12.2 do-mpc v4.0.0-beta3

3.12.2.1 Major changes

3.12.2.1.1 Data

- New `__getitem__` method to conveniently retrieve values from `Data` object (details here)
- New `MPCData` class (which inherits form `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.

3.12.2.1.2 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.)

3.12.2.1.3 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.

3.12.2.1.4 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

3.12.2.2 Example files

Please download the example files for release do-mpc v4.0.0-beta3 here.

3.12.3 do-mpc v4.0.0-beta2

Error in release. Immediately replaced with beta3.

3.12.4 do-mpc v4.0.0-beta1

3.12.4.1 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.

3.12.4.2 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.

3.12.4.3 Other changes

- Spelling in documentation
- New guide about installing HSL linear solver
- Credits in documentation

3.12.4.4 Example files

Please download the example files for release do-mpc v4.0.0-beta1 here.

3.12.5 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.

3.12.5.1 Example files

Please download the example files for release do-mpc v4.0.0-beta here.

3.12.6 do-mpc v3.0.0

3.12.6.1 Main modifications

- Support for CasADi version 3.4.4
- Support for time-varying parameters
- Support for discrete-time systems

3.12.7 do-mpc v2.0.0

Compatible with CasADi 3.0.0

3.12.8 do-mpc version 1.0.0

3.13 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**.

[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
```

3.13.1 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:

[2]:
```
model_type = 'continuous' # either 'discrete' or 'continuous'
model = do_mpc.model.Model(model_type)
```

3.13.1.1 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]:
```
# States struct (optimization variables):
X_s = model.set_variable('_x',  'X_s')
S_s = model.set_variable('_x',  'S_s')
P_s = model.set_variable('_x',  'P_s')
V_s = model.set_variable('_x',  'V_s')
```

The control input is the feed flow rate u_{inp} of S_s:

[4]:
```
# Input struct (optimization variables):
inp = model.set_variable('_u',  'inp')
```

3.13.1.2 ODE and parameters

The system model is described by the ordinary differential equation:

$$\dot{X}_s = \mu(S_s)X_s - \frac{u_{inp}}{V_s}X_s, \tag{3.25}$$

$$\dot{S}_s = -\frac{\mu(S_s)X_s}{Y_x} - \frac{vX_s}{Y_p} + \frac{u_{inp}}{V_s}(S_{in} - S_s), \tag{3.26}$$

$$\dot{P}_s = vX_s - \frac{u_{inp}}{V_s}P_s, \tag{3.27}$$

$$\dot{V}_s = u_{inp}, \tag{3.28}$$

$$\tag{3.29}$$

where:

$$\mu(S_s) = \frac{\mu_m S_s}{K_m + S_s + (S_s^2/K_i)}, \tag{3.30}$$

S_{in} is the inlet substrate concentration, μ_m, K_m, K_i and v are kinetic parameters Y_x and Y_p are yield coefficients. The inlet substrate concentration S_{in} and the Y_x are uncertain while the rest of the parameters is considered certain:

[5]:
```
# Certain parameters
mu_m  = 0.02
K_m   = 0.05
K_i   = 5.0
v_par = 0.004
Y_p   = 1.2

# Uncertain parameters:
Y_x  = model.set_variable('_p',  'Y_x')
S_in = model.set_variable('_p', 'S_in')
```

In the next step, the ODE for each state is set:

[6]:
```
# Auxiliary term
mu_S = mu_m*S_s/(K_m+S_s+(S_s**2/K_i))

# Differential equations
model.set_rhs('X_s', mu_S*X_s - inp/V_s*X_s)
```

(continues on next page)

(continued from previous page)

```
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)
```

Finally, the model setup is completed:

[7]:
```
# Build the model
model.setup()
```

3.13.2 Controller

Next, the controller is configured. First, one member of the mpc class is generated with the prediction model defined above:

[8]:
```
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:

[9]:
```
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)
```

3.13.2.1 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.

[10]:
```
mterm = -model.x['P_s'] # stage cost
lterm = -model.x['P_s'] # terminal cost

mpc.set_objective(mterm=mterm, lterm=lterm)
mpc.set_rterm(inp=1.0) # penalty on input changes
```

3.13.2.2 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:

[11]:
```
# lower bounds of the states
mpc.bounds['lower', '_x', 'X_s'] = 0.0
mpc.bounds['lower', '_x', 'S_s'] = -0.01
mpc.bounds['lower', '_x', 'P_s'] = 0.0
mpc.bounds['lower', '_x', 'V_s'] = 0.0

# upper bounds of the states
mpc.bounds['upper', '_x','X_s'] = 3.7
mpc.bounds['upper', '_x','P_s'] = 3.0

# upper and lower bounds of the control input
mpc.bounds['lower','_u','inp'] = 0.0
mpc.bounds['upper','_u','inp'] = 0.2
```

3.13.2.3 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:

[12]:
```
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_values, S_in_values])
```

This means with `n_robust=1`, that 9 different scenarios are considered. The setup of the MPC controller is concluded by:

[13]:
```
mpc.setup()
```

3.13.3 Estimator

We assume, that all states can be directly measured (state-feedback):

[14]:
```
estimator = do_mpc.estimator.StateFeedback(model)
```

3.13.4 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:

[15]:
```
simulator = do_mpc.simulator.Simulator(model)
```

For the simulation, we use the time step `t_step` as for the optimizer:

[16]:
```
params_simulator = {
    'integration_tool': 'cvodes',
    'abstol': 1e-10,
    'reltol': 1e-10,
    't_step': 1.0
}

simulator.set_param(**params_simulator)
```

3.13.4.1 Realizations of uncertain parameters

For the simulatiom, it is necessary to define the numerical realizations of the uncertain parameters in `p_num`. First, we get the structure of the uncertain parameters:

[17]:
```
p_num = 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_now`):

[18]:
```
p_num['Y_x'] = 0.4
p_num['S_in'] = 200.0

# 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_fun` as above, the function will always return the same values. To finish the configuration of the simulator, call:

[19]:
```
simulator.setup()
```

3.13.5 Closed-loop simulation

For the simulation of the MPC configured for the batch bioreactor, we inspect the file **main.py**. We define the initial state of the system and set for all parts of the closed-loop configuration:

[20]:
```
# Initial state
X_s_0 = 1.0 # Concentration biomass [mol/l]
S_s_0 = 0.5 # Concentration substrate [mol/l]
P_s_0 = 0.0 # Concentration product [mol/l]
V_s_0 = 120.0 # Volume inside tank [m^3]
x0 = np.array([X_s_0, S_s_0, P_s_0, V_s_0])

# Set for controller, simulator and estimator
mpc.x0 = x0
simulator.x0 = x0
estimator.x0 = x0
mpc.set_initial_guess()
```

3.13.5.1 Prepare visualization

For the visualization of the control performance, we first import matplotlib and change some basic settings:

[21]:
```
import matplotlib.pyplot as plt
plt.ion()
from matplotlib import rcParams
rcParams['text.usetex'] = True
rcParams['text.latex.preamble'] = [r'\usepackage{amsmath}',r'\usepackage{siunitx}']
rcParams['axes.grid'] = True
```

(continues on next page)

(continued from previous page)

```
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.

[22]:
```
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:

[23]:
```
%%capture
fig, ax = plt.subplots(5, sharex=True, figsize=(16,9))
fig.align_ylabels()

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')


ax[0].set_ylabel(r'$X_s~[\si[per-mode=fraction]{\mole\per\litre}]$')
ax[1].set_ylabel(r'$S_s~[\si[per-mode=fraction]{\mole\per\litre}]$')
ax[2].set_ylabel(r'$P_s~[\si[per-mode=fraction]{\mole\per\litre}]$')
ax[3].set_ylabel(r'$V_s~[\si[per-mode=fraction]{\mole\per\litre}]$')
ax[4].set_ylabel(r'$u_{\text{inp}}~[\si[per-mode=fraction]{\cubic\metre\per\minute}]$
↪')
ax[4].set_xlabel(r'$t~[\si[per-mode=fraction]{\minute}]$')
```

3.13.5.2 Run closed-loop

The closed-loop system is now simulated for 50 steps (and the ouput of the optimizer is suppressed):

[24]:
```
%%capture
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)
```

3.13.5.3 Results

The next cell converts the results of the closed-loop MPC simulation into a gif (might take a few minutes):

[25]:
```
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()


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:

3.14 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**.

[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

import matplotlib.pyplot as plt
```

3.14.1 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:

[2]:
```
model_type = 'continuous' # either 'discrete' or 'continuous'
model = do_mpc.model.Model(model_type)
```

3.14.1.1 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):

[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 $\dot{Q}$:

[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')
```

3.14.1.2 ODE and parameters

The system model is described by the ordinary differential equation:

$$\dot{C}_\mathrm{A} = F \cdot (C_{\mathrm{A},0} - C_\mathrm{A}) - k_1 \cdot C_\mathrm{A} - k_3 \cdot C_\mathrm{A}^2, \tag{3.31}$$

$$\dot{C}_\mathrm{B} = F \cdot C_\mathrm{B} + k_1 \cdot C_\mathrm{A} - k_2 \cdot C_\mathrm{B}, \tag{3.32}$$

$$\dot{T}_\mathrm{R} = \frac{k_1 \cdot C_\mathrm{A} \cdot H_{\mathrm{R},ab} + k_2 \cdot C_\mathrm{B} \cdot H_{\mathrm{R},bc} + k_3 \cdot C_\mathrm{A}^2 \cdot H_{\mathrm{R},ad}}{-\rho \cdot c_p} \tag{3.33}$$

$$+ F \cdot (T_\mathrm{in} - T_\mathrm{R}) + \frac{K_w \cdot A_\mathrm{R} \cdot (T_\mathrm{K} - T_\mathrm{R})}{\rho \cdot c_p \cdot V_\mathrm{R}}, \tag{3.34}$$

$$\dot{T}_\mathrm{K} = \frac{\dot{Q} + k_w \cdot A_\mathrm{R} \cdot T_\mathrm{dif}}{m_k \cdot C_{p,k}}, \tag{3.35}$$

where

$$k_1 = \beta \cdot k_{0,\mathrm{ab}} \cdot \exp\left(\frac{-E_{\mathrm{A,ab}}}{T_\mathrm{R} + 273.15}\right), \tag{3.36}$$

$$k_1 = k_{0,\mathrm{bc}} \cdot \exp\left(\frac{-E_{\mathrm{A,bc}}}{T_\mathrm{R} + 273.15}\right), \tag{3.37}$$

$$k_3 = k_{0,\mathrm{ad}} \cdot \exp\left(\frac{-\alpha \cdot E_{\mathrm{A,ad}}}{T_\mathrm{R} + 273.15}\right). \tag{3.38}$$

The parameters α and β are uncertain while the rest of the parameters is considered certain:

[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]
```

(continues on next page)

(continued from previous page)

```
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.
↪5 [mol/l]

# 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:

[6]:
```
# Auxiliary terms
K_1 = beta * K0_ab * exp((-E_A_ab)/((T_R+273.15)))
K_2 =  K0_bc * exp((-E_A_bc)/((T_R+273.15)))
K_3 = K0_ad * exp((-alpha*E_A_ad)/((T_R+273.15)))
```

Additionally, we define an artificial variable of interest, that is not a state of the system, but will be later used for plotting:

[7]:
```
T_dif = model.set_expression(expr_name='T_dif', expr=T_R-T_K)
```

WIth the help ot the k_i-s and T_{dif} we can define the ODEs:

[8]:
```
model.set_rhs('C_a', F*(C_A0 - C_a) -K_1*C_a - K_3*(C_a**2))
model.set_rhs('C_b', -F*C_b + K_1*C_a - K_2*C_b)
model.set_rhs('T_R', ((K_1*C_a*H_R_ab + K_2*C_b*H_R_bc + K_3*(C_a**2)*H_R_ad)/(-
↪Rou*Cp)) + F*(T_in-T_R) +(((K_w*A_R)*(-T_dif))/(Rou*Cp*V_R)))
model.set_rhs('T_K', (Q_dot + K_w*A_R*(T_dif))/(m_k*Cp_k))
```

Finally, the model setup is completed:

[9]:
```
# Build the model
model.setup()
```

3.14.2 Controller

Next, the model predictive controller is configured. First, one member of the mpc class is generated with the prediction model defined above:

[10]:
```
mpc = do_mpc.controller.MPC(model)
```

We choose the prediction horizon `n_horizon`, set the robust horizon `n_robust` to 1. The time step `t_step` is set to one second and parameters of the applied discretization scheme orthogonal collocation are as seen below:

[11]:
```
setup_mpc = {
    'n_horizon': 20,
    'n_robust': 1,
    'open_loop': 0,
    't_step': 0.005,
```

(continues on next page)

(continued from previous page)

```
    '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)
```

Because the magnitude of the states and inputs is very different, we introduce scaling factors:

[12]:
```
mpc.scaling['_x', 'T_R'] = 100
mpc.scaling['_x', 'T_K'] = 100
mpc.scaling['_u', 'Q_dot'] = 2000
mpc.scaling['_u', 'F'] = 100
```

3.14.2.1 Objective

The goal of the CSTR is to obtain a mixture with a concentration of $C_{\mathrm{B,ref}} = 0.6$ mol/l. Additionally, we add a penalty on input changes for both control inputs, to obtain a smooth control performance.

[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
```

3.14.2.2 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:

[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()`.

[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))
```

3.14.2.3 Uncertain values

The explicit values of the two uncertain parameters α and β, which are considered in the scenario tree, are given by:

[16]:
```
alpha_var = np.array([1., 1.05, 0.95])
beta_var = np.array([1., 1.1, 0.9])

mpc.set_uncertainty_values([alpha_var, beta_var])
```

This means with `n_robust=1`, that 9 different scenarios are considered. The setup of the MPC controller is concluded by:

[17]:
```
mpc.setup()
```

3.14.3 Estimator

We assume, that all states can be directly measured (state-feedback):

[18]:
```
estimator = do_mpc.estimator.StateFeedback(model)
```

3.14.4 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:

[19]:
```
simulator = do_mpc.simulator.Simulator(model)
```

For the simulation, we use the same time step `t_step` as for the optimizer:

[20]:
```
params_simulator = {
    'integration_tool': 'cvodes',
    'abstol': 1e-10,
    'reltol': 1e-10,
    't_step': 0.005
}

simulator.set_param(**params_simulator)
```

3.14.4.1 Realizations of uncertain parameters

For the simulatiom, 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:

[21]:
```
p_num = simulator.get_p_template()
tvp_num = simulator.get_tvp_template()
```

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_now`):

[22]:
```
# 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 custum functions are used in the simulation via:

[23]:
```
simulator.set_tvp_fun(tvp_fun)
simulator.set_p_fun(p_fun)
```

By defining `p_fun` as above, the function will always return the value 1.0 for both α and β. To finish the configuration of the simulator, call:

[24]:
```
simulator.setup()
```

3.14.5 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:

[25]:
```
# 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`):

[26]:
```
%%capture
for k in range(50):
    u0 = mpc.make_step(x0)
    y_next = simulator.make_step(u0)
    x0 = estimator.make_step(y_next)
```

3.14.6 Animating the results

To animate the results, we first configure the **do-mpc** graphics object, which is initiated with the respective data object:

[27]:
```
mpc_graphics = do_mpc.graphics.Graphics(mpc.data)
```

We quickly configure Matplotlib.

[28]:
```
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.

[29]:
```
%%capture
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`.

[30]:
```
# 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:

[31]:
```
from matplotlib.animation import FuncAnimation, ImageMagickWriter
```

We obtain the animation with:

[32]:
```
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)

gif_writer = ImageMagickWriter(fps=5)
anim.save('anim_CSTR.gif', writer=gif_writer)
```

```
Writing frame: 49.
```

Recorded trajectories are shown as solid lines, whereas predictions are dashed. We highlight the nominal prediction with a thicker line.

3.15 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**.

[29]:
```
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
```

3.15.1 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:

[2]:
```
model_type = 'continuous' # either 'discrete' or 'continuous'
model = do_mpc.model.Model(model_type)
```

3.15.1.1 System description

The system consists of a reactor into which nonomer is fed. The monomerturns 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):

$$\dot{m}_{\mathrm{W}} = \dot{m}_{\mathrm{F}}\,\omega_{\mathrm{W,F}} \tag{3.39}$$

$$\dot{m}_{\mathrm{A}} = \dot{m}_{\mathrm{F}}\omega_{\mathrm{A,F}} - k_{\mathrm{R1}}\,m_{\mathrm{A,R}} - k_{\mathrm{R2}}\,m_{\mathrm{AWT}}\,m_{\mathrm{A}}/m_{\mathrm{ges}}, \tag{3.40}$$

$$\dot{m}_{\mathrm{P}} = k_{\mathrm{R1}}\,m_{\mathrm{A,R}} + p_1\,k_{\mathrm{R2}}\,m_{\mathrm{AWT}}\,m_{\mathrm{A}}/m_{\mathrm{ges}}, \tag{3.41}$$

$$\dot{T}_{\mathrm{R}} = 1/(c_{\mathrm{p,R}}m_{\mathrm{ges}})\,[\dot{m}_{\mathrm{F}}\,c_{\mathrm{p,F}}\,(T_{\mathrm{F}} - T_{\mathrm{R}}) + \Delta H_{\mathrm{R}}k_{\mathrm{R1}}m_{\mathrm{A,R}} - k_{\mathrm{K}}A\,(T_{\mathrm{R}} - T_{\mathrm{S}}) \tag{3.42}$$

$$- \dot{m}_{\mathrm{AWT}}\,c_{\mathrm{p,R}}\,(T_{\mathrm{R}} - T_{\mathrm{EK}})],$$

$$\dot{T}_S = 1/(c_{\mathrm{p,S}}m_{\mathrm{S}})\,[k_{\mathrm{K}}A\,(T_{\mathrm{R}} - T_{\mathrm{S}}) - k_{\mathrm{K}}A\,(T_{\mathrm{S}} - T_{\mathrm{M}})],$$

$$\dot{T}_{\mathrm{M}} = 1/(c_{\mathrm{p,W}}m_{\mathrm{M,KW}})\,[\dot{m}_{\mathrm{M,KW}}\,c_{\mathrm{p,W}}\,\left(T_{\mathrm{M}}^{\mathrm{IN}} - T_{\mathrm{M}}\right) \tag{3.43}$$

$$+ k_{\mathrm{K}}A\,(T_{\mathrm{S}} - T_{\mathrm{M}})] + k_{\mathrm{K}}A\,(T_{\mathrm{S}} - T_{\mathrm{M}})], \tag{3.44}$$

$$\dot{T}_{\mathrm{EK}} = 1/(c_{\mathrm{p,R}}m_{\mathrm{AWT}})\,[\dot{m}_{\mathrm{AWT}}c_{\mathrm{p,W}}\,(T_{\mathrm{R}} - T_{\mathrm{EK}}) - \alpha\,(T_{\mathrm{EK}} - T_{\mathrm{AWT}}) \tag{3.45}$$

$$+ k_{\mathrm{R2}}\,m_{\mathrm{A}}\,m_{\mathrm{AWT}}\Delta H_{\mathrm{R}}/m_{\mathrm{ges}}],$$

$$\dot{T}_{\mathrm{AWT}} = [\dot{m}_{\mathrm{AWT,KW}}\,c_{\mathrm{p,W}}\,(T_{\mathrm{AWT}}^{\mathrm{IN}} - T_{\mathrm{AWT}}) - \alpha\,(T_{\mathrm{AWT}} - T_{\mathrm{EK}})]/(c_{\mathrm{p,W}}m_{\mathrm{AWT,KW}}), \tag{3.46}$$

where

$$U = m_{\mathrm{P}}/(m_{\mathrm{A}} + m_{\mathrm{P}}), \tag{3.47}$$

$$m_{\mathrm{ges}} = m_{\mathrm{W}} + m_{\mathrm{A}} + m_{\mathrm{P}}, \tag{3.48}$$

$$k_{\mathrm{R1}} = k_0 e^{\frac{-E_a}{R(T_{\mathrm{R}}+273.15)}}\,(k_{\mathrm{U1}}\,(1-U) + k_{\mathrm{U2}}U)\,, \tag{3.49}$$

$$k_{\mathrm{R2}} = k_0 e^{\frac{-E_a}{R(T_{\mathrm{EK}}+273.15)}}\,(k_{\mathrm{U1}}\,(1-U) + k_{\mathrm{U2}}U)\,, \tag{3.50}$$

$$k_{\mathrm{K}} = (m_{\mathrm{W}}k_{\mathrm{WS}} + m_{\mathrm{A}}k_{\mathrm{AS}} + m_{\mathrm{P}}k_{\mathrm{PS}})/m_{\mathrm{ges}}, \tag{3.51}$$

$$m_{\mathrm{A,R}} = m_{\mathrm{A}} - m_{\mathrm{A}}m_{\mathrm{AWT}}/m_{\mathrm{ges}}. \tag{3.52}$$

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 $m_{\mathrm{A,R}}$ represents the current amount of monomer inside the reactor.

The available control inputs are the feed flow $\dot{m}_{\mathrm{F}}$, the coolant temperature at the inlet of the jacket $T_{\mathrm{M}}^{\mathrm{IN}}$ and the coolant temperature at the inlet of the external heat exchanger $T_{\mathrm{AWT}}^{\mathrm{IN}}$.

An overview of the parameters are listed below:

Parameter	
R	
$c_{p,W}$	
$c_{p,S}$	
$c_{p,F}$	
$c_{p,R}$	
k_{WS}	
T_F	
A	
$m_{M,KW}$	
m_S	
m_{AWT}	
$m_{AWT,KW}$	
$\dot{m}_{M,KW}$	
$\dot{m}_{AWT,KW}$	
$\dot{m}_{AWT}$	
E_a	
ΔH_R	

3.15.1.2 Implementation

First, we set the certain parameters:

```
[3]: # 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      = 20000.0               #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:

```
[4]: # 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 $\dot{m}_{\mathrm{F}}^{\mathrm{acc}} = \dot{m}_{\mathrm{F}}$ and `T_adiab` ($T_{\mathrm{adiab}} = \frac{\Delta H_{\mathrm{R}}}{c_{\mathrm{p,R}}} \frac{m_{\mathrm{A}}}{m_{\mathrm{ges}}} + T_{\mathrm{R}}$, hence $\dot{T}_{\mathrm{adiab}} = \frac{\Delta H_{\mathrm{R}}}{m_{\mathrm{ges}} c_{\mathrm{p,R}}} \dot{m}_{\mathrm{A}} - (\dot{m}_{\mathrm{W}} + \dot{m}_{\mathrm{A}} + \dot{m}_{\mathrm{P}}) \left(\frac{m_{\mathrm{A}} \Delta H_{\mathrm{R}}}{m_{\mathrm{ges}}^2 c_{\mathrm{p,R}}} \right) + \dot{T}_{\mathrm{R}}$) is a virtual variable that is important for safety aspects, as we will explain later. All states are created in **do-mpc** via:

```
[5]: # 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')
Tout_M =        model.set_variable('_x', 'Tout_M')
T_EK =          model.set_variable('_x', 'T_EK')
```

(continues on next page)

(continued from previous page)

```
Tout_AWT =    model.set_variable('_x', 'Tout_AWT')
accum_monom = model.set_variable('_x', 'accum_monom')
T_adiab =     model.set_variable('_x', 'T_adiab')
```

and the control inputs via:

[6]:
```
# Input struct (optimization variables):
m_dot_f = model.set_variable('_u', 'm_dot_f')
T_in_M =  model.set_variable('_u', 'T_in_M')
T_in_EK = model.set_variable('_u', 'T_in_EK')
```

Before defining the ODE for each state variable, we create auxiliary terms:

[7]:
```
# algebraic equations
U_m    = m_P / (m_A + m_P)
m_ges  = m_W + m_A + m_P
k_R1   = k_0 * exp(- E_a/(R*T_R)) * ((k_U1 * (1 - U_m)) + (k_U2 * U_m))
k_R2   = k_0 * exp(- E_a/(R*T_EK))* ((k_U1 * (1 - U_m)) + (k_U2 * U_m))
k_K    = ((m_W / m_ges) * k_WS) + ((m_A/m_ges) * k_AS) + ((m_P/m_ges) * k_PS)
```

The auxiliary terms are used for the more readable definition of the ODEs:

[8]:
```
# Differential equations
dot_m_W = m_dot_f * w_WF
model.set_rhs('m_W', dot_m_W)
dot_m_A = (m_dot_f * w_AF) - (k_R1 * (m_A-((m_A*m_AWT)/(m_W+m_A+m_P)))) - (p_1 * k_R2␣
↪* (m_A/m_ges) * m_AWT)
model.set_rhs('m_A', dot_m_A)
dot_m_P = (k_R1 * (m_A-((m_A*m_AWT)/(m_W+m_A+m_P)))) + (p_1 * k_R2 * (m_A/m_ges) * m_
↪AWT)
model.set_rhs('m_P', dot_m_P)

dot_T_R = 1./(c_pR * m_ges)    * ((m_dot_f * c_pF * (T_F - T_R)) - (k_K *A_tank* (T_R -
↪ T_S)) - (fm_AWT * c_pR * (T_R - T_EK)) + (delH_R * k_R1 * (m_A-((m_A*m_AWT)/(m_W+m_
↪A+m_P)))))
model.set_rhs('T_R', dot_T_R)
model.set_rhs('T_S', 1./(c_pS * m_S)      * ((k_K *A_tank* (T_R - T_S)) - (k_K *A_
↪tank* (T_S - Tout_M))))
model.set_rhs('Tout_M', 1./(c_pW * m_M_KW)   * ((fm_M_KW * c_pW * (T_in_M - Tout_M)) +␣
↪(k_K *A_tank* (T_S - Tout_M))))
model.set_rhs('T_EK', 1./(c_pR * m_AWT)    * ((fm_AWT * c_pR * (T_R - T_EK)) - (alfa *␣
↪(T_EK - Tout_AWT)) + (p_1 * k_R2 * (m_A/m_ges) * m_AWT * delH_R)))
model.set_rhs('Tout_AWT', 1./(c_pW * m_AWT_KW)* ((fm_AWT_KW * c_pW * (T_in_EK - Tout_
↪AWT)) - (alfa * (Tout_AWT - T_EK))))
model.set_rhs('accum_monom', m_dot_f)
model.set_rhs('T_adiab', delH_R/(m_ges*c_pR)*dot_m_A-(dot_m_A+dot_m_W+dot_m_P)*(m_
↪A*delH_R/(m_ges*m_ges*c_pR))+dot_T_R)
```

Finally, the model setup is completed:

[9]:
```
# Build the model
model.setup()
```

3.15.2 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:

[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_{adiab}, hence it needs to stay beneath $109°C$.

We choose the prediction horizon `n_horizon`, set the robust horizon `n_robust` to 1. The time step `t_step` is set to one second and parameters of the applied discretization scheme orthogonal collocation are as seen below:

[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'}
}

mpc.set_param(**setup_mpc)
```

3.15.2.1 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.

[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
```

3.15.2.2 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 $\pm 2.0°C$ around the desired reaction temperature $T_{\text{set}} = 90°C$ in order to ensure that the produced polymer has the required properties.

The initial conditions and the bounds for all states are summarized in:

State	Init. con
m_W	10,000
m_A	853
m_P	26.5
T_R	90.0
T_S	90.0
T_M	90.0
T_{EK}	35.0
T_{AWT}	35.0
T_{adiab}	104.897
m_F^{acc}	0

and set via:

[13]:
```
# 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_R'] = 363.15 + temp_range + 10.0
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 + 10.0
```

The bounds of the inputsare summarized below:

Control
$\dot{m}_{\mathrm{F}}$
$T^{\mathrm{IN}}_{\mathrm{M}}$
$T^{\mathrm{IN}}_{\mathrm{AWT}}$

and set via:

[14]:
```
# 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
```

3.15.2.3 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:

[15]:
```
# 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
```

3.15.2.4 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 ΔH_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

[16]:
```
delH_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_var, k_0_var])
```

This means with `n_robust=1`, that 9 different scenarios are considered. The setup of the MPC controller is concluded by:

[17]:
```
mpc.setup()
```

3.15.3 Estimator

We assume, that all states can be directly measured (state-feedback):

[18]:
```
estimator = do_mpc.estimator.StateFeedback(model)
```

3.15.4 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:

[19]:
```
simulator = do_mpc.simulator.Simulator(model)
```

For the simulation, we use the same time step `t_step` as for the optimizer:

[20]:
```
params_simulator = {
    'integration_tool': 'cvodes',
    'abstol': 1e-10,
    'reltol': 1e-10,
    't_step': 50.0/3600.0
}

simulator.set_param(**params_simulator)
```

3.15.4.1 Realizations of uncertain parameters

For the simulatiom, it is necessary to define the numerical realizations of the uncertain parameters in `p_num`. First, we get the structure of the uncertain parameters:

[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_now`):

[22]:
```
# uncertain parameters
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 nomimal value. To finish the configuration of the simulator, call:

[23]:
```
simulator.setup()
```

3.15.5 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:

[24]:
```
# 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
```

(continues on next page)

(continued from previous page)

```
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['T_adiab'] = 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`):

[25]:
```
%%capture
for k in range(100):
    u0 = mpc.make_step(x0)
    y_next = simulator.make_step(u0)
    x0 = estimator.make_step(y_next)
```

3.15.6 Animating the results

To animate the results, we first configure the **do-mpc** graphics object, which is initiated with the respective data object:

[48]:
```
mpc_graphics = do_mpc.graphics.Graphics(mpc.data)
```

We quickly configure Matplotlib.

[49]:
```
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.

[50]:
```
%%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')
```

(continues on next page)

(continued from previous page)

```
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:

[43]:
```
from matplotlib.animation import FuncAnimation, ImageMagickWriter
```

We obtain the animation with:

[51]:
```
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)

gif_writer = ImageMagickWriter(fps=5)
anim.save('anim_poly_batch.gif', writer=gif_writer)
```

```
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:

[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 uncertainy is impossible.

3.16 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**.

[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
```

3.16.1 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:

$$A = \begin{bmatrix} 0.763 & 0.460 & 0.115 & 0.020 \\ 0.899 & 0.763 & 0.420 & 0.115 \\ 0.115 & 0.020 & 0.763 & 0.460 \\ 0.420 & 0.115 & 0.899 & 0.763 \end{bmatrix}, \qquad B = \begin{bmatrix} 0.014 \\ 0.063 \\ 0.221 \\ 0.367 \end{bmatrix},$$

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)
```

3.16.1.1 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()
```

3.16.2 Controller

Next, the model predictive controller is configured. First, one member of the mpc class is generated with the prediction model defined above:

[7]:
```
mpc = do_mpc.controller.MPC(model)
```

We choose the prediction horizon `n_horizon` to 7 and set the robust horizon `n_robust` to zero, because no uncertainties are present. The time step `t_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:

[8]:
```
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)
```

3.16.2.1 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 the cost expression defined in the model:

[9]:
```
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
```

3.16.2.2 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 $-4\text{m} \leq s_i \leq 4\text{m}$, the velocity $-10\text{ms}^{-1} \leq v_i \leq 10\text{ms}^{-1}$ and the force cannot exceed $-0.5\text{N} \leq u_1 \leq 0.5\text{N}$:

[10]:
```
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:

[11]:
```
mpc.setup()
```

3.16.3 Estimator

We assume, that all states can be directly measured (state-feedback):

[12]:
```
estimator = do_mpc.estimator.StateFeedback(model)
```

3.16.4 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:

[13]:
```
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:

[14]:
```
simulator.set_param(t_step = 0.1)
simulator.setup()
```

3.16.5 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:

[15]:
```
# 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`):

[16]:
```
%%capture
for k in range(50):
    u0 = mpc.make_step(x0)
    y_next = simulator.make_step(u0)
    x0 = estimator.make_step(y_next)
```

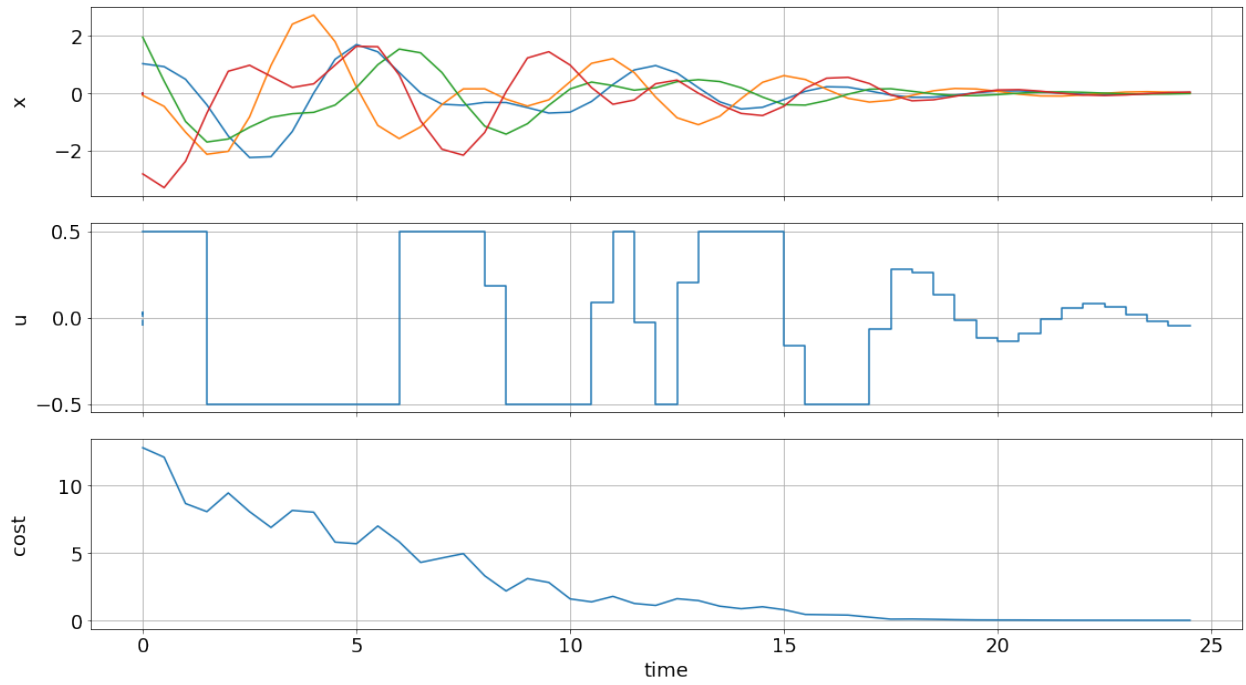
3.16.6 Displaying the results

After some slight configuration of matplotlib:

[17]:
```
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.

[18]:
```
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 sucessfully fulfilled and that bounds, e.g. for the control inputs are satisfied.

CHAPTER 4

Indices and tables

- genindex
- search

Bibliography

[Biegler2010] L.T. Biegler. Nonlinear Programming: Concepts, Algorithms, and Applications to Chemical Processes. SIAM, 2010.

Python Module Index

d

Index

Symbols

A

B

C

D

E

G

I

L

M

T

U

V

W

X

Y

Z