# do-mpc

*Release 4.0.0*

**Apr 28, 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 Dormund 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**.

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

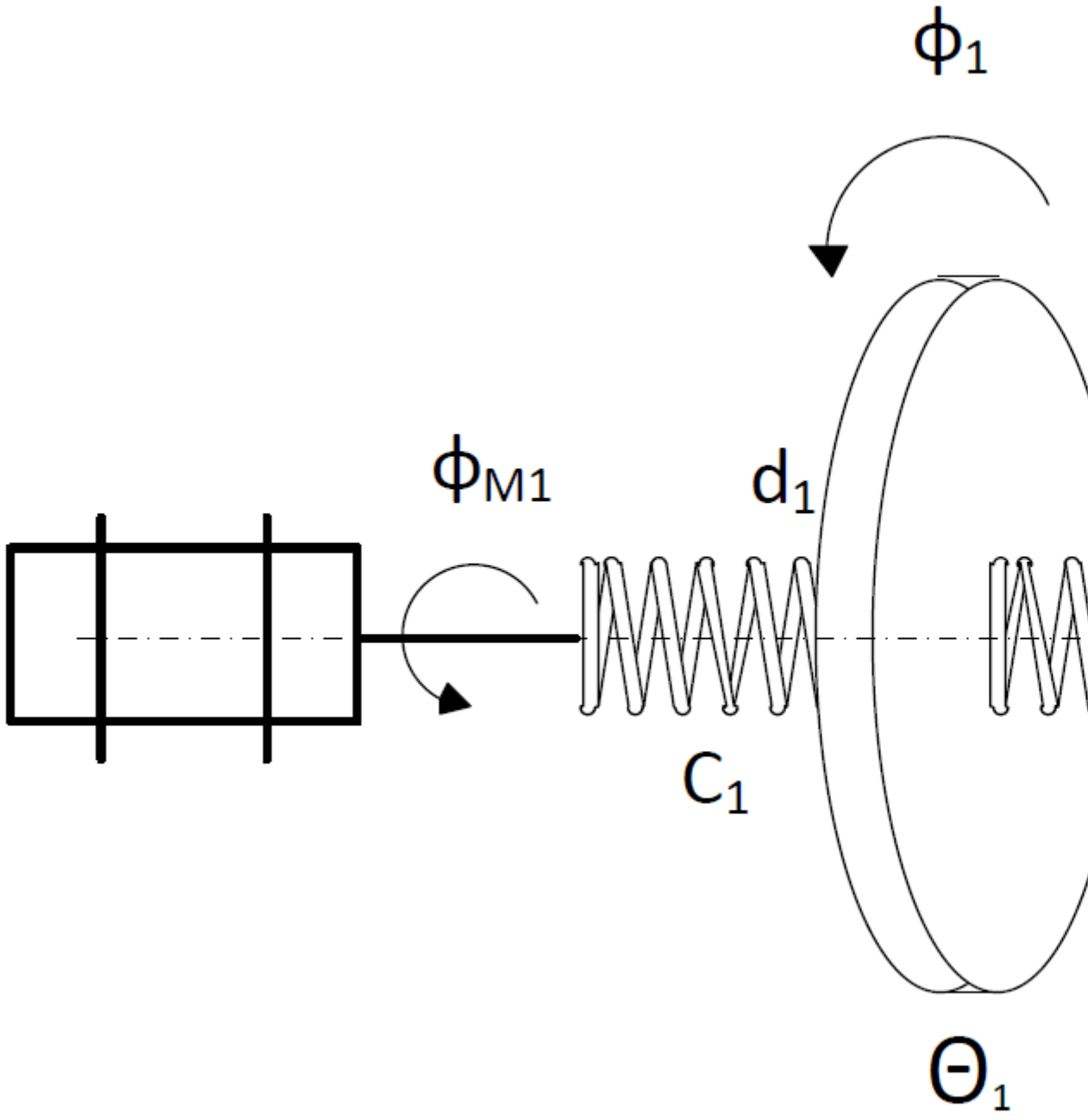
[1]:
```
import numpy as np

# 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 $\phi_1, \phi_2, \phi_3$. The two outermost discs are each connected to a stepper motor with additional springs. The stepper motor angles ($\phi_{m,1}$ and $\phi_{m,2}$ are used as inputs to the system. Relevant parameters of the system are the inertia $\Theta$ of the three discs, the spring constants $c$ as well as the damping factors $d$.

The second degree ODE of this system can be written as follows:

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

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:

$$\begin{aligned}\frac{\partial x}{\partial t} &= f(x,u,z,p)\\ 0 &= g(x,u,z,p)\\ y &= h(x,u,z,p)\end{aligned}$$

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:

$$\begin{aligned}x_1 &= \phi_1\\ x_2 &= \phi_2\\ x_3 &= \phi_3\\ x_4 &= \dot{\phi}_1\\ x_5 &= \dot{\phi}_2\\ x_6 &= \dot{\phi}_3\end{aligned}$$

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

$$\begin{aligned}\dot{x}_1 &= x_4\\ \dot{x}_2 &= x_5\\ \dot{x}_3 &= x_6\\ \dot{x}_4 &= -\frac{c_1}{\Theta_1}\left(x_1 - u_1\right) - \frac{c_2}{\Theta_1}\left(x1 - x_2\right) - \frac{d_1}{\Theta_1}x_4\\ \dot{x}_5 &= -\frac{c_2}{\Theta_2}\left(x_2 - x_1\right) - \frac{c_3}{\Theta_2}\left(x_2 - x_3\right) - \frac{d_2}{\Theta_2}x_5\\ \dot{x}_6 &= -\frac{c_3}{\Theta_3}\left(x_3 - x_2\right) - \frac{c_4}{\Theta_3}\left(x_4 - u_2\right) - \frac{d_3}{\Theta_3}x_6\end{aligned}$$

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.

[5]:
```
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 |

[6]:
```
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:

[7]:
```
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`:

[8]:
```
model.x
```

[8]:
```
<casadi.tools.structure3.ssymStruct at 0x106c36cf8>
```

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

[9]:
```
model.x['phi_1']
```

[9]:
```
SX(phi_1)
```

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

[10]:
```
bool(model.x['phi_1'] == phi_1)
```

[10]:
```
True
```

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

[11]:
```
model.x['dphi',0]
```

[11]:
```
SX(dphi_0)
```

Note that you can use the following methods:

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

to get more information from the symbolic structures:

[12]:
```
model.x.keys()
```

[12]:
```
['phi_1', 'phi_2', 'phi_3', 'dphi', 'phi_1_m', 'phi_2_m']
```

[13]:
```
model.x.labels()
```

[13]:
```
['[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.

[14]:
```
# 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 $x, u, z, p$.

[15]:
```
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:

[16]:
```
from casadi import *
```

[17]:
```
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)
```

[18]:
```
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()`:

[19]:
```
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)

[20]:
```
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`.

[21]:
```
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:

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

[22]:
```
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 $\Delta u_0$.

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

[23]:
```
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:

[24]:
```
# 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).

[25]:
```
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:

[26]:
```
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.

[27]:
```
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`:

[28]:
```
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`.

[29]:
```
# 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:

[30]:
```
p_template = simulator.get_p_template()
```

This object is a CasADi structure:

[31]:
```
type(p_template)
```

[31]:
```
casadi.tools.structure3.DMStruct
```

which can be indexed with the following keys:

[32]:
```
p_template.keys()
```

[32]:
```
['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:

[33]:
```
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:

[34]:
```
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.

[35]:
```
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 $x_0 \neq 0$:

[36]:
```
x0 = np.pi*np.array([1, 1, -1.5, 1, -1, 1, 0, 0]).reshape(-1,1)
```

and use the `.set_initial_state()` method to set the initial state (which also calls `optimizer.set_initial_guess()`.

[37]:
```
simulator.set_initial_state(x0, reset_history=True)
mpc.set_initial_state(x0, reset_history=True)
```

#### 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])
```

(continues on next page)

(continued from previous page)

```
    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

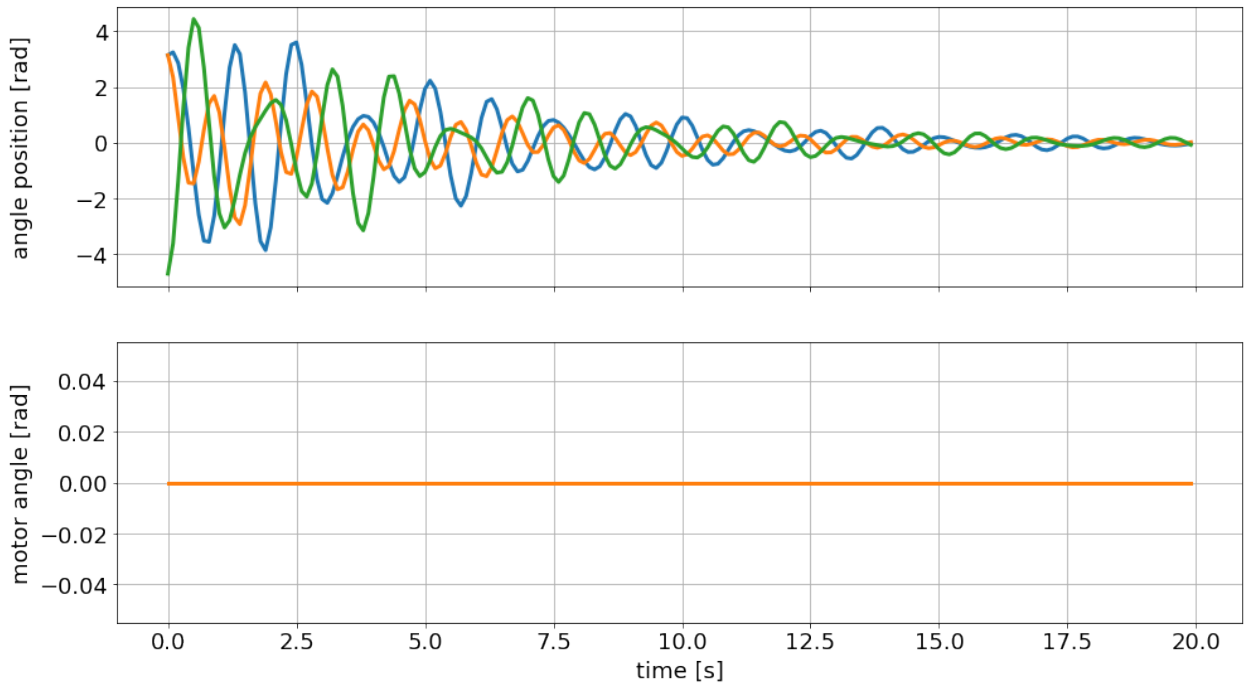
We start investigating the **do-mpc** simulator and the `graphics` package by simulating the autonomous system without control inputs ($u = 0$). This can be done as follows:

[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
   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
```

(continues on next page)

(continued from previous page)

```
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)   =      0.251
Total CPU secs in NLP function evaluations           =      0.006

EXIT: Optimal Solution Found.
           S  :   t_proc      (avg)   t_wall      (avg)     n_eval
       nlp_f  | 139.00us ( 11.58us) 140.00us ( 11.67us)         12
       nlp_g  |   1.80ms (150.33us)   1.69ms (140.92us)         12
    nlp_grad  | 367.00us (367.00us) 367.00us (367.00us)          1
  nlp_grad_f  | 469.00us ( 36.08us) 470.00us ( 36.15us)         13
  nlp_hess_l  | 129.00us ( 11.73us) 129.00us ( 11.73us)         11
   nlp_jac_g  |   3.20ms (246.31us)   3.24ms (248.92us)         13
       total  | 268.13ms (268.13ms) 267.36ms (267.36ms)          1
```

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

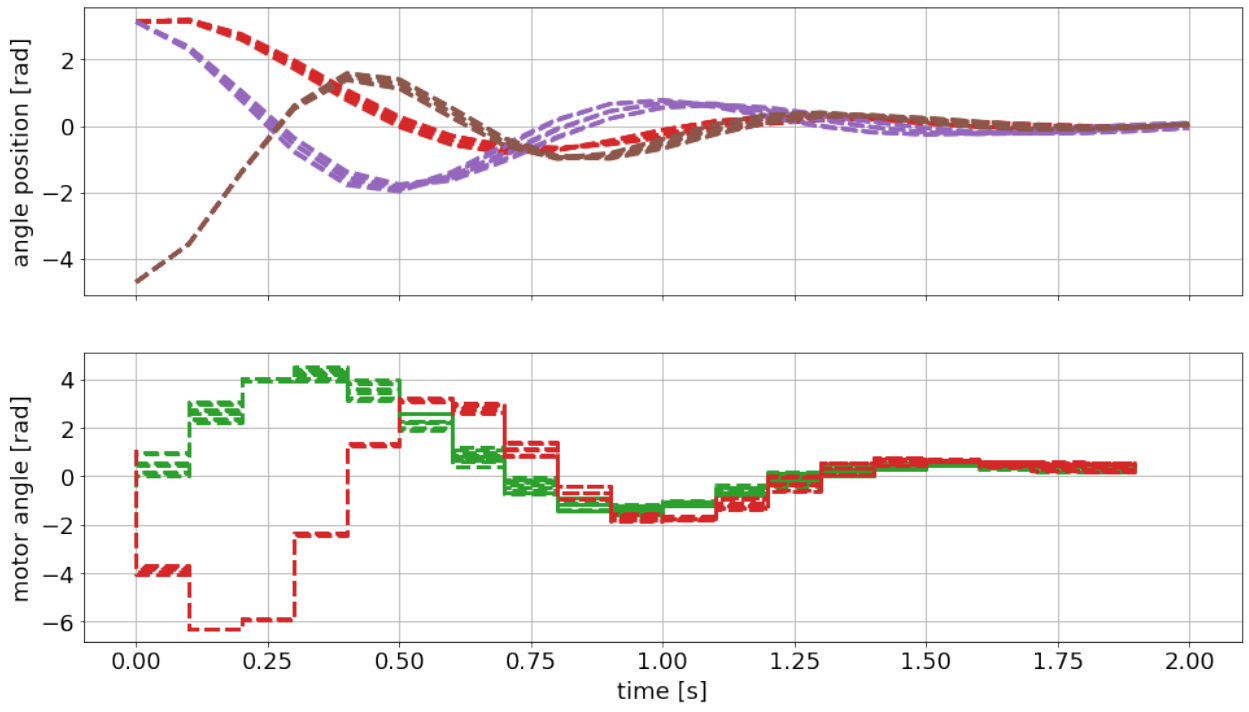
We can also visualize the predicted trajectories with the configure `graphics` instance. First we clear the existing lines from the simulator by calling:

[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 0x1156f6ac8>
```

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

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

[48]:
```
[<matplotlib.lines.Line2D at 0x116b02d68>,
 <matplotlib.lines.Line2D at 0x116b02630>,
 <matplotlib.lines.Line2D at 0x116b02320>,
 <matplotlib.lines.Line2D at 0x116b027b8>,
 <matplotlib.lines.Line2D at 0x116b024e0>,
 <matplotlib.lines.Line2D at 0x116b02e80>,
 <matplotlib.lines.Line2D at 0x116b09358>,
 <matplotlib.lines.Line2D at 0x116b090f0>,
 <matplotlib.lines.Line2D at 0x116b09940>]
```

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:
```

(continues on next page)

(continued from previous page)

```
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 0x117523b38>
```

#### 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.

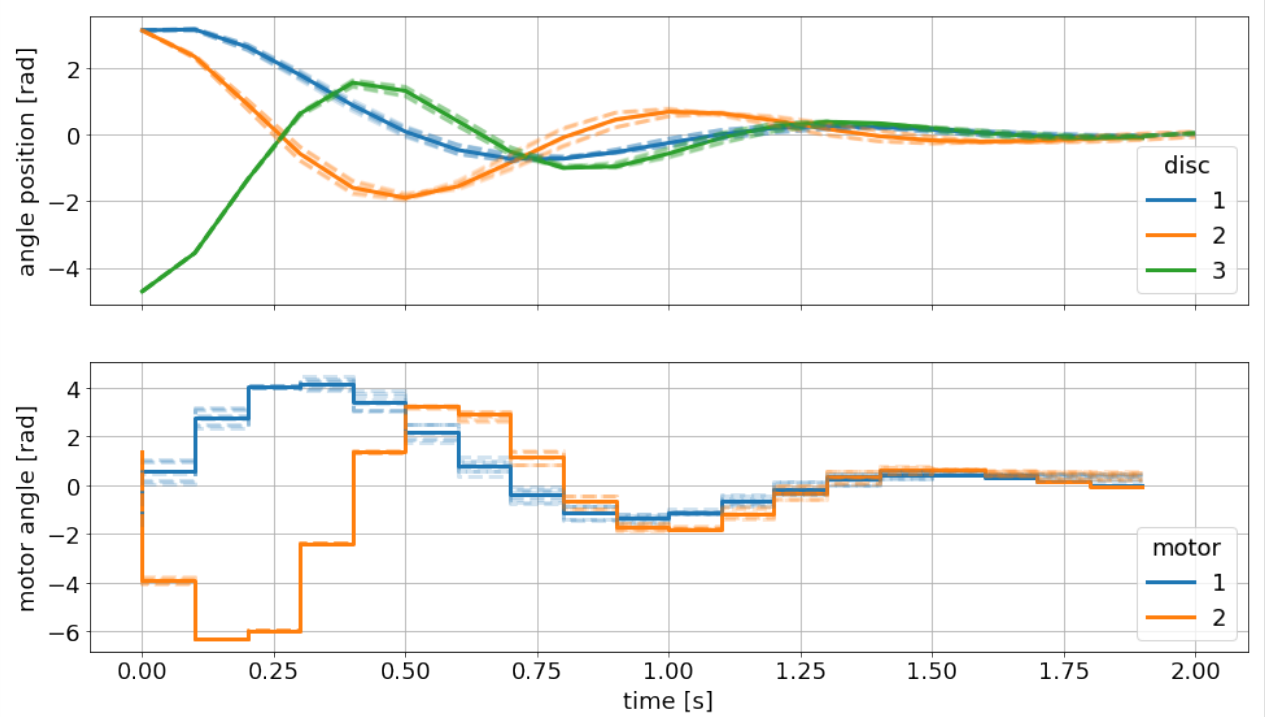
It is usually suggested to display the output as it contains important information about the state of the solver.

[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:

[75]:
```
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.

[76]:
```
save_results([mpc, simulator])
```

We investigate the content of the newly created folder:

[77]:
```
!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:

[78]:
```
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.

[79]:
```
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:

[80]:
```
results['mpc']
```

[80]:
```
<do_mpc.data.MPCData at 0x7f58bf7be6d0>
```

[84]:
```
x = results['mpc']['_x']
x.shape
```

[84]:
```
(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:

[86]:
```
phi_1 = results['mpc']['_x','phi_1']

phi_1.shape
```

[86]:
```
(20, 1)
```

For vector-valued states we can even query:

[90]:
```
dphi_1 = results['mpc']['_x','dphi', 0]

dphi_1.shape
```

[90]:
```
(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:

[97]:
```
phi_1_pred = results['mpc'].prediction(('_x','phi_1'), t_ind=0)

phi_1_pred.shape
```

[97]:
```
(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:

[99]:
```
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.

[100]:
```
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

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)
```

#### 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.

[4]:
```
# State measurements
phi_meas = model.set_meas('phi_1_meas', phi)

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

#### 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 $x, u, z, p$.

[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)

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:

$$\min_{\mathbf{x}_{0:N},\mathbf{u}_{0:N-1},\mathbf{w}_{0:N-1}} \underbrace{(x_0-\tilde{x}_0)^T P_x (x_0-\tilde{x}_0)}_{\text{arrival cost states}} + \underbrace{(p_0-\tilde{p}_0)^T P_p (p_0-\tilde{p}_0)}_{\text{arrival cost params.}}$$
$$+ \sum_{k=0}^{n-1} \underbrace{(h(x_k,u_k,p_k)-y_k)^T P_{y,k} (h(x_k,u_k,p_k)-y_k) + w_k^T P_w w_k}_{\text{stage cost}}$$
$$\text{s.t.:} \quad x_{k+1} = f(x_k,u_k,z_k,p_k,p_{tv,k}) + w_k$$

We typically consider the formulation shown above, where the user has to pass the weighting matrices `P_x`, `P_y`, `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_y = np.diag(np.array([1,1,1,20,20]))
      P_x = np.eye(8)
      P_p = 10*np.eye(1)

      mhe.set_default_objective(P_x, P_y, 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 `.set_initial_state()` method to set the initial state:

[24]:
```
simulator.set_initial_state(x0, reset_history=True)
mhe.set_initial_state(x0_mhe ,p_est0=np.array([1e-4]), reset_history=True)
```

#### 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:

[25]:
```
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`.

[26]:
```
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`:

[27]:
```
%%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:

[28]:
```
%%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:

[29]:
```
sim_graphics.result_lines
```

[29]:
```
<do_mpc.tools.structure.Structure at 0x11948fa90>
```

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

[30]:
```
# First element for state phi:
sim_graphics.result_lines['_x', 'phi', 0]
```

[30]:
```
[<matplotlib.lines.Line2D at 0x11a32f240>]
```

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

[31]:
```
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:

[32]:
```
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')
```

[32]:
```
<matplotlib.legend.Legend at 0x11a3885c0>
```

and another legend for the parameter plot:

[33]:
```
ax_p.legend(sim_graphics.result_lines['_p', 'Theta_1']+mhe_graphics.result_lines['_p',
↪ 'Theta_1'], ['True','Estim.'])
```

[33]:
```
<matplotlib.legend.Legend at 0x11a3884a8>
```

#### 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:

[34]:
```
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.

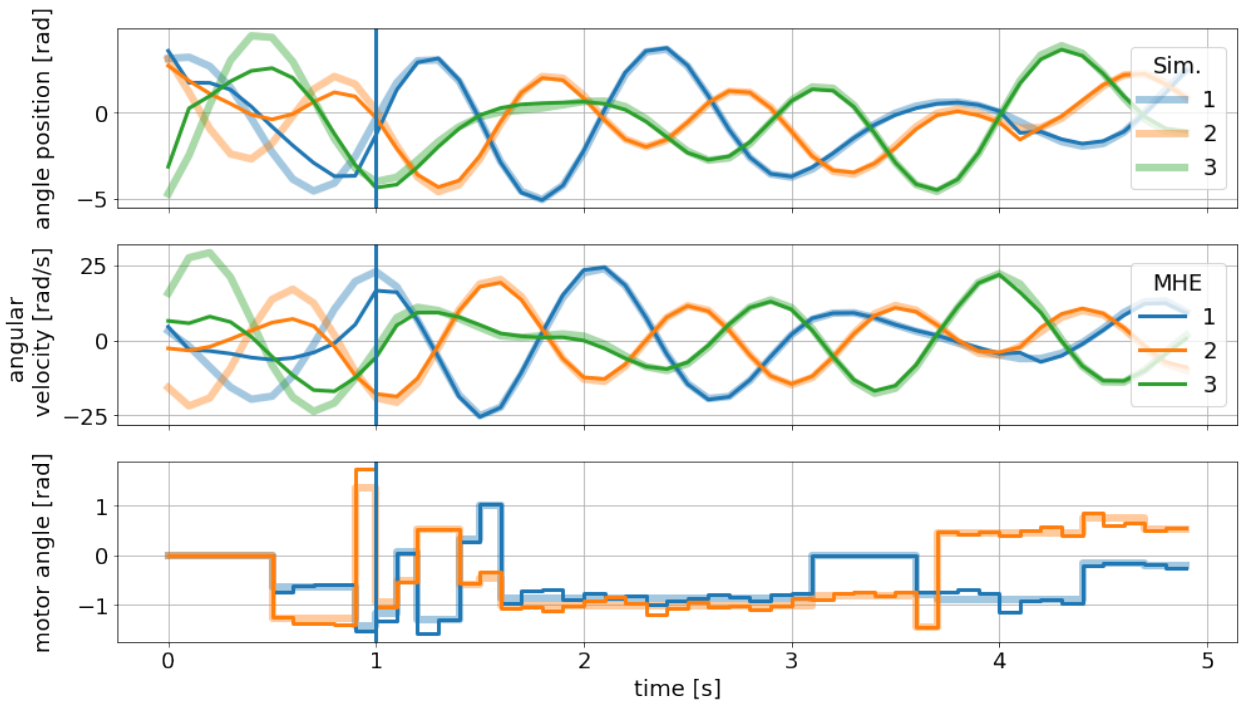
We can now run the loop. At each iteration, we perturb our measurements with additive Gaussian noise, for a more realistic scenario.

[35]:
```
%%capture
np.random.seed(999) #make it repeatable

u0 = np.zeros((2,1))
for i in range(50):
    u0 = random_u(u0) # Control input
    y0 = simulator.make_step(u0) # true measurement
    y0 = y0*(1+0.1*np.random.randn(model.n_y,1)) # perturbed measurement
    x0 = mhe.make_step(y0) # MHE estimation step
```

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

[36]:
```
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
```

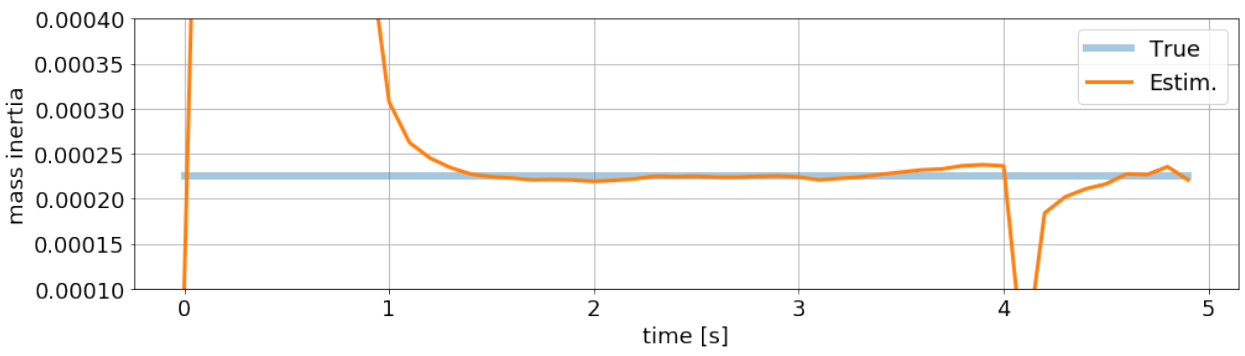
[36]:




Parameter estimation:

[37]:
```
ax_p.set_ylim(1e-4, 4e-4)
ax_p.set_ylabel('mass inertia')
ax_p.set_xlabel('time [s]')
fig_p
```

[37]:




### 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:

[45]:
```
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.

[46]:
```
mhe.setup()
```

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

[47]:
```
mhe.reset_history()
simulator.reset_history()
```

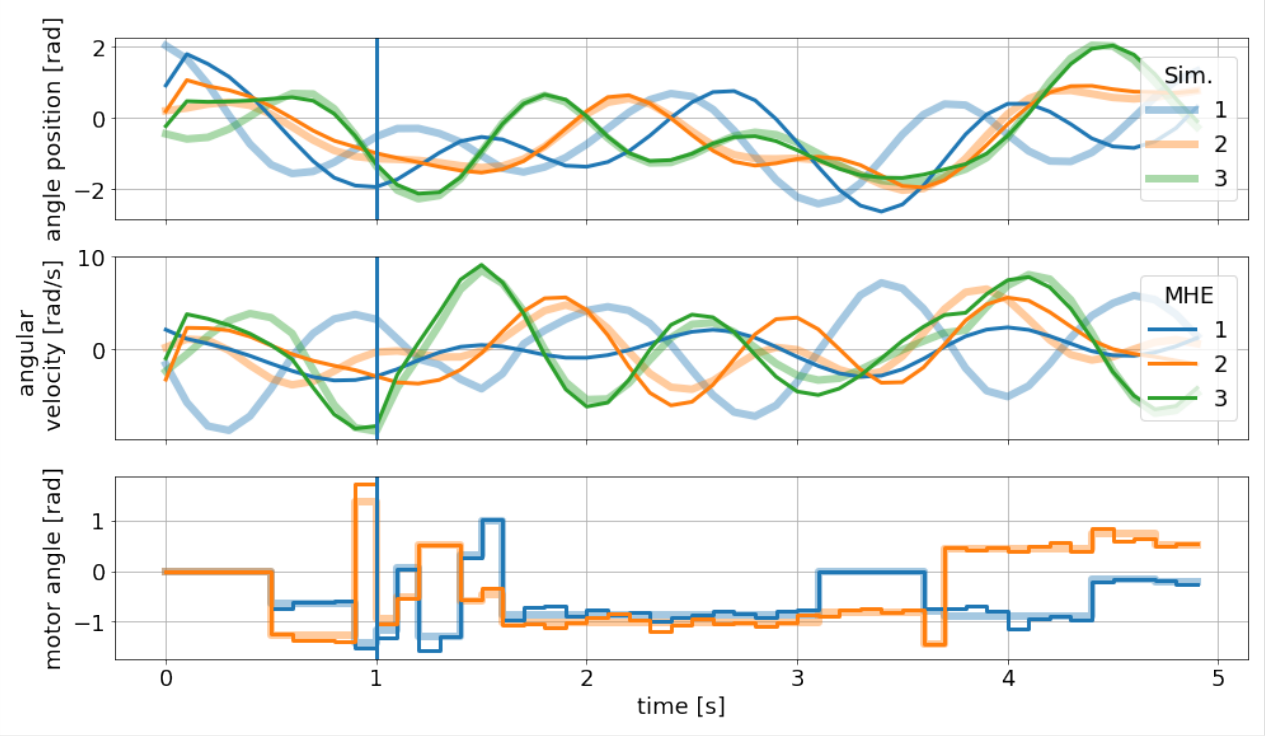
Finally, we run the exact same loop again obtaining new results.

[48]:
```
%%capture
np.random.seed(999) #make it repeatable

u0 = np.zeros((2,1))
for i in range(50):
    u0 = random_u(u0) # Control input
    y0 = simulator.make_step(u0) # true measurement
    y0 = y0*(1+0.1*np.random.randn(model.n_y,1)) # perturbed measurement
    x0 = mhe.make_step(y0) # MHE estimation step
```

These results now look quite terrible:

[53]:
```
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
```

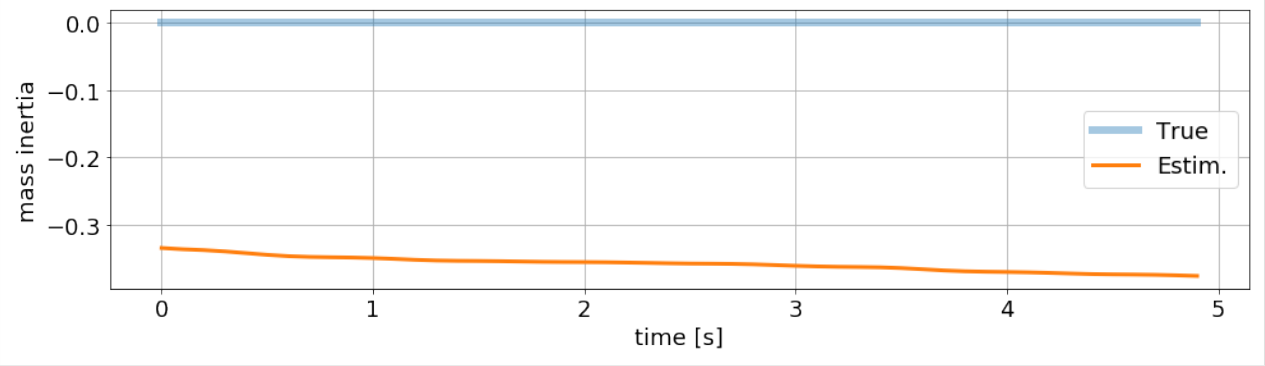
[53]:




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

[54]:
```
ax_p.set_ylabel('mass inertia')
ax_p.set_xlabel('time [s]')
fig_p
```

[54]:




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 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.

## 3.4 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.

### 3.4.1 Requirements

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

- numpy
- CasADi
- matplotlib

### 3.4.2 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 GitHub repository. Clone or download the repository and obtain the files from the examples directory.

### 3.4.3 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.

### 3.4.4 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.

#### 3.4.4.1 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:

#### 3.4.4.1.1 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.

---

### 3.4.4.2 Option 2: Compile from source

Please see the comprehensive guide on the CasADi Github Wiki.

## 3.5 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, ...)

## 3.6 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*)

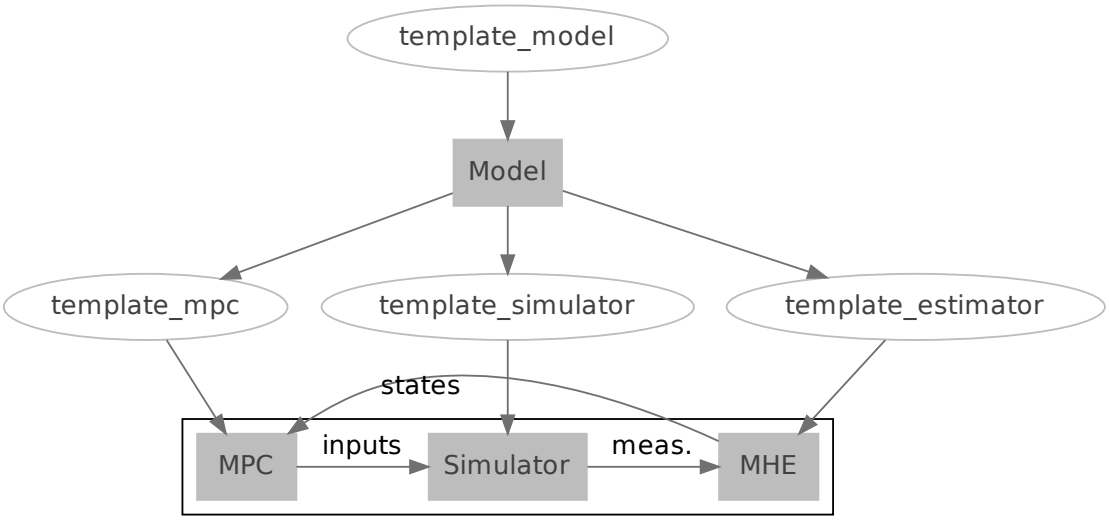



Fig. 1: Project structure

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.

### 3.6.1 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))
    ...

    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
```

### 3.6.2 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,
```

(continues on next page)

(continued from previous page)

```
        't_step': 0.005,
        ...
    }
    mpc.set_param(**setup_mpc)

    # Configure objective function:
    mterm = (_x['C_b'] - 0.6)**2    # Setpoint tracking
    lterm = (_x['C_b'] - 0.6)**2    # Setpoint tracking

    mpc.set_objective(mterm=mterm, lterm=lterm)
    mpc.set_rterm(F=0.1, Q_dot = 1e-3) # Scaling for quad. cost.

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

    mpc.setup()

    return mpc
```

### 3.6.3 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
```

### 3.6.4 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
```

(continues on next page)

(continued from previous page)

```
    # 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,
        '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.

### 3.6.5 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
```

(continues on next page)

(continued from previous page)

```
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.

#### 3.6.5.1 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.

#### 3.6.5.2 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.

#### 3.6.5.3 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)
```

(continues on next page)

(continued from previous page)

```
    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)
```

## 3.7 Debugging

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

### 3.7.1 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.

#### 3.7.1.1 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.

#### 3.7.1.2 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.

#### 3.7.1.3 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*

## 3.8 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:

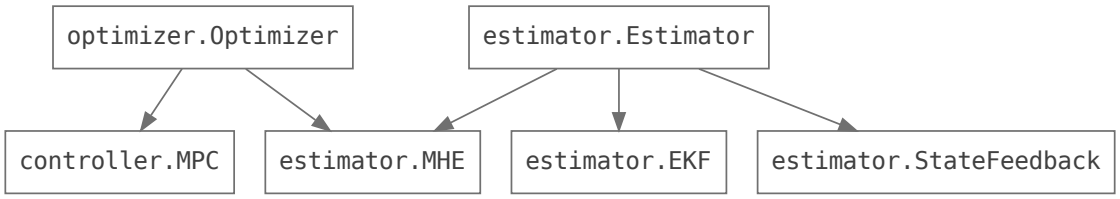



Fig. 2: Class inheritance. Click on classes for more information.

### Modules

| |
|---|
| *model* |
| *simulator* |
| *optimizer* |
| *controller* |
| *estimator* |
| *data* |
| *graphics* |

## 3.8.1 model

### Classes

| | |
|---|---|
| *Model* | The **do-mpc** model class. |

#### 3.8.1.1 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):

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

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

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

**Configuration and setup:**

Configuring and setting up the optimizer involves the following steps:

1. Use *set_variable()* to introduce new variables to the model. It is important to declare the variable type (states, inputs, etc.).
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.

**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.w* | Process noise. |
| *Model.x* | Dynamic states. |
| *Model.y* | Measurements. |
| *Model.z* | Algebraic states. |

#### 3.8.1.1.1 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.y['dtemp', 0] # 0th element of variable
model.y['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.

#### 3.8.1.1.2 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.

#### 3.8.1.1.3 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.

#### 3.8.1.1.4 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.

#### 3.8.1.1.5 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`.

---

**Note:** The process noise is currently only used for the *`do_mpc.estimator.MHE`*.

---

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.

#### 3.8.1.1.6 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.

#### 3.8.1.1.7 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:
```

(continues on next page)

(continued from previous page)

```
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.

#### 3.8.1.1.8 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.get_variables* | Method to retrieve the variables of the model. |
| *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: |

Continued on next page

Table 4 – continued from previous page

| | |
|---|---|
| *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. |

#### 3.8.1.1.9 get_variables

Class method.

do_mpc.model.Model.**get_variables**(*self*)

Method to retrieve the variables of the model.

> **Warning:** The method is depreciated and will be removed in a later version. Please query class attributes (variables) directly.

This method is convenient when creating the model in a different file than the, e.g. the *do_mpc.optimizer*. Returns the variables as a list with the following order:

- _x (states)
- _u (inputs)
- _z (algebraic variables)
- _tvp (time varying parameters)
- _p (uncertain parameters)
- _aux (auxiliary expressions)
- _y (measurements, measured)
- _y_expression (measurements, calculated)

The method cannot be called prior to `model.setup()`.

Variables can also be retrieved independently of this method with, e.g.:

```
x = model.x
```

Note that structures can be indexed with the previously defined variable names:

```
_x['var_name']
```

**Raises** **assertion** – Model was not setup. Finish model creation by calling model.setup_model().

**Returns** List of model variables (_x, _u, _z, _tvp, _p, _aux, _y, _y_expression)

**Return type** list

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

#### 3.8.1.1.10 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. They can also be used for the objective function or constraints. 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)
```

**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 Model.setup_model().

**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.

#### 3.8.1.1.11 set_meas

Class method.

do_mpc.model.Model.**set_meas**(*self*, *meas_name*, *expr*)

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

---

**Note:** For moving horizon estimation it is suggested to declare all inputs (_u) and a subset of states (_x) as measurable output. Some other MHE formulations treat inputs separately.

---

**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.

**Raises**

- **assertion** – expr_name must be str
- **assertion** – expr must be a casadi SX or MX type
- **assertion** – Cannot call after .setup_model.

**Returns** Returns the newly created measurement expression.

**Return type** casadi.SX

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

#### 3.8.1.1.12 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:

$$\dot{x} = f(\dots),$$

or the update equation in case of discrete dynamics:

$$x_{k+1} = f(\dots).$$

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 currently only meaningful for the *do_mpc.estimator.MHE*. See *do_mpc.estimator.MHE.set_default_objective()* for more details.

**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 .setup_model.

**Returns** None

**Return type** None

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

#### 3.8.1.1.13 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 | Required |
| 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 `Model.setup_model()`.

**Returns** Returns the newly created symbolic variable.

**Return type** casadi.SX

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

#### 3.8.1.1.14 setup

Class method.

`do_mpc.model.Model.`**`setup`**(*self*)

Setup method must be called to finalize the modelling process. All required model variables `_x`, `_u`, `_z`, `_tvp`, `_p` 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_model()`*, 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.

#### 3.8.1.1.15 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.

## 3.8.2 simulator

### Classes

| | |
|---|---|
| *Simulator* | A class for simulating systems. |

#### 3.8.2.1 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.

### Methods

| | |
|---|---|
| *Simulator.get_p_template* | Obtain output template for *set_p_fun()*. |
| *Simulator.get_tvp_template* | Obtain the output template for *set_tvp_fun()*. |
| *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* | Function to set the function which gives the values of the parameters. |
| *Simulator.set_param* | Set the parameters for the simulator. |
| *Simulator.set_tvp_fun* | Function to set the function which gives the values of the time-varying parameters. |
| *Simulator.setup* | Sets up the simulator and finalizes the simulator configuration. |
| *Simulator.simulate* | Call the CasADi simulator. |

##### 3.8.2.1.1 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.

#### 3.8.2.1.2 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.

#### 3.8.2.1.3 make_step

Class method.

do_mpc.simulator.Simulator.**make_step**(*self*, *u0*, *x0=None*, *z0=None*)

Main method of the simulator class during control runtime. This method is called at each timestep and returns the next state for the current control input `u0`. 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.

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

**Returns** x_nsext

**Return type** numpy.ndarray

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

#### 3.8.2.1.4 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.

#### 3.8.2.1.5 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.

**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.

#### 3.8.2.1.6 set_p_fun

Class method.

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

Function 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')
```

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 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:

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

The parameter function can look something like this:

```
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

simulator.set_p_fun(p_fun)
```

which results in constant parameters.

**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.

#### 3.8.2.1.7 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.

#### 3.8.2.1.8 set_tvp_fun

Class method.

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

Function to set the function which gives 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('parameter', '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 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:

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

The 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 incorporate a finite set of future / past information, e.g. regarding 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.

#### 3.8.2.1.9 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.

#### 3.8.2.1.10 simulate

Class method.

do_mpc.simulator.Simulator.**simulate**(*self*)

Call the CasADi simulator. 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.

> **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.

**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.

## 3.8.3 optimizer

### Classes

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

#### 3.8.3.1 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* | Queries and sets the bounds of the optimization variables for the optimizer. |
| *Optimizer.scaling* | Queries and sets the scaling of the optimization variables for the optimizer. |

###### 3.8.3.1.1 bounds

Class attribute.

Optimizer.**bounds**

Queries and sets the bounds of the optimization variables for the optimizer. The *Optimizer.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*. |

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.

#### 3.8.3.1.2 scaling

Class attribute.

`Optimizer.`**`scaling`**

Queries and sets the scaling of the optimization variables for the optimizer. 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*. |

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* | The method returns a structured object with n_horizon elements, and a set of time varying parameters (as defined in model) for each of these instances. |
| *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 the tvp_fun which is called at each optimization step to get the current prediction of the time-varying parameters. |
| *Optimizer.solve* | Solves the optmization problem. |

#### 3.8.3.1.3 get_tvp_template

Class method.

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

The method returns a structured object with n_horizon elements, and a set of time varying parameters (as defined in 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 `Optimizer.set_tvp_fun()`. Use the combination of `Optimizer.get_tvp_template()` and `Optimizer.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.

#### 3.8.3.1.4 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.

#### 3.8.3.1.5 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.

**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.

#### 3.8.3.1.6 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 $\epsilon$, 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.

#### 3.8.3.1.7 set_tvp_fun

Class method.

do_mpc.optimizer.Optimizer.**set_tvp_fun**(*self*, *tvp_fun*)

Set the tvp_fun which 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 `Optimizer.get_tvp_template()`.

```
# 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)
```

The method `Optimizer.set_tvp_fun()`. must be called prior to setup IF time-varying parameters are defined in the model.

**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.

#### 3.8.3.1.8 solve

Class method.

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

Solves the optmization problem. The current time-step is defined by the parameters in the `self.opt_p_num` CasADi structured Data. These include the initial condition, the parameters, the time-varying paramters and the previous input. Typically, `self.opt_p_num` is prepared for the current iteration in the `.make_step()` (in MHE/MPC) method. It is, however, valid and possible to directly set paramters in `self.opt_p_num` before calling `.solve()`.

Solve updates the `opt_x_num`, and `lam_g_num` attributes of the class. In 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.

**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.

### 3.8.4 controller

#### Classes

| | |
|---|---|
| *MPC* | Model predictive controller. |

#### 3.8.4.1 MPC

**class** do_mpc.controller.**MPC**(*model*)

Model predictive controller. The MPC controller extends the *do_mpc.optimizer.Optimizer* base class (which is also used for the MHE estimator). Use this class to configure and run the MPC controller based on a previously configured *do_mpc.model* instance.

**Configuration and setup:**

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

1. Use *MPC.set_param()* to configure the *MPC* instance.
2. Set the objective of the control problem with *MPC.set_objective()* and *MPC.set_rterm()*
3. Set upper and lower bounds with *do_mpc.optimizer.Optimizer.bounds()* (optional).
4. Set further (non-linear) constraints with *do_mpc.optimizer.Optimizer.set_nl_cons()* (optional).
5. Use the low-level API (*MPC.get_p_template()* and *MPC.set_p_fun()*) or high level API (*MPC.set_uncertainty_values()*) to create scenarios for robust MPC (optional).
6. Finally, call *MPC.setup()*.

### Attributes

| | |
|---|---|
| *MPC.bounds* | Queries and sets the bounds of the optimization variables for the optimizer. |
| *MPC.opt_p_num* | Full MPC parameter vector. |
| *MPC.opt_x_num* | Full MPC solution and initial guess. |
| *MPC.scaling* | Queries and sets the scaling of the optimization variables for the optimizer. |

#### 3.8.4.1.1 bounds

Class attribute.

MPC.**bounds**

Queries and sets the bounds of the optimization variables for the optimizer. The `Optimizer.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*. |

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.

#### 3.8.4.1.2 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 *MPC.make_step()* method. However, you can set these values manually and directly call *MPC.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 *MPC.setup()*

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

#### 3.8.4.1.3 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 *MPC.setup()*

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

#### 3.8.4.1.4 scaling

Class attribute.

MPC.**scaling**

Queries and sets the scaling of the optimization variables for the optimizer. 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*. |

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

| | |
|---|---|
| *MPC.get_p_template* | Obtain output template for *set_p_fun()*. |
| *MPC.get_tvp_template* | The method returns a structured object with n_horizon elements, and a set of time varying parameters (as defined in model) for each of these instances. |
| *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. |
| *MPC.set_nl_cons* | Introduce new constraint to the class. |
| *MPC.set_objective* | Sets the objective of the optimal control problem (OCP). |

Continued on next page

Table 12 – continued from previous page

| | |
|---|---|
| *MPC.set_p_fun* | Set parameter function for MPC. |
| *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 the tvp_fun which is called at each optimization step to get the current prediction of the 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. |

#### 3.8.4.1.5 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 MPC but defining an arbitrary number of combinations for the parameters defined in the model. The method returns a structured object which is initialized with all zeros. Use this object to define value of the parameters for an arbitrary number of scenarios (defined by n_scenarios).

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 (see doc string).

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

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 optimizer configuration:
n_combinations = 3
p_template = optimizer.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

optimizer.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.

#### 3.8.4.1.6 get_tvp_template

Class method.

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

The method returns a structured object with n_horizon elements, and a set of time varying parameters (as defined in 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 `Optimizer.set_tvp_fun()`. Use the combination of `Optimizer.get_tvp_template()` and `Optimizer.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.

#### 3.8.4.1.7 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 *do_mpc.optimizer.Optimizer.solve()* and updates the *do_mpc.data.Data* object.

**Parameters** **x0** (*numpy.ndarray*) – Current state of the system.

**Returns** u0

**Return type** numpy.ndarray

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

#### 3.8.4.1.8 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.

#### 3.8.4.1.9 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 instances of x, u and z. The method is automatically evoked when calling the `MPC.setup()` method. 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.

#### 3.8.4.1.10 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.

**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.

#### 3.8.4.1.11 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 $\epsilon$, 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.

#### 3.8.4.1.12 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:

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

MPC.set_objective() is used to set the $l(x_k, u_k, z_k, p)$ (lterm) and $m(x_N)$ (mterm), where N is the prediction horizon. Please see MPC.set_rterm() for the rterm.

**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.

#### 3.8.4.1.13 set_p_fun

Class method.

do_mpc.controller.MPC.**set_p_fun**(*self*, *p_fun*)

Set parameter function for MPC. Low level API method to set user defined scenarios for robust MPC but defining an arbitrary number of combinations for the parameters defined in the model. 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 `MPC.get_p_template()`.

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

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 optimizer configuration:
n_combinations = 3
p_template = optimizer.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

optimizer.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 `MPC.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.

#### 3.8.4.1.14 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:** MPC.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*) – Choose the collocation ni for continuous models with collocation as state discretization. Defaults to 1.
- **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.

#### 3.8.4.1.15 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 optimizer configuration:
optimizer.set_rterm(Q_heat = 10)
optimizer.set_rterm(F_flow = 10)
# or alternatively:
optimizer.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.

#### 3.8.4.1.16 set_tvp_fun

Class method.

do_mpc.controller.MPC.**set_tvp_fun**(*self*, *tvp_fun*)

Set the tvp_fun which 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 `Optimizer.get_tvp_template()`.

```
# 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)
```

The method `Optimizer.set_tvp_fun()`. must be called prior to setup IF time-varying parameters are defined in the model.

**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.

#### 3.8.4.1.17 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.

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')
```

(continues on next page)

(continued from previous page)

```
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.

#### 3.8.4.1.18 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 method also sets the initial guess with `MPC.set_initial_guess()`. The `MPC.setup()` method can be called again after changing the configuration (e.g. adapting bounds) and will simply overwrite the previous optimization problem.

After setup, the `MPC` instance enables the *`make_step()`* method. This method is used during runtime to obtain the MPC current control input given the current state.

The method sets `flags['setup'] = True`.

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

#### 3.8.4.1.19 solve

Class method.

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

Solves the optmization problem. The current time-step is defined by the parameters in the `self.opt_p_num` CasADi structured Data. These include the initial condition, the parameters, the time-varying paramters and the previous input. Typically, `self.opt_p_num` is prepared for the current iteration in the `.make_step()` (in MHE/MPC) method. It is, however, valid and possible to directly set paramters in `self.opt_p_num` before calling `.solve()`.

Solve updates the `opt_x_num`, and `lam_g_num` attributes of the class. In 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.

**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.

## 3.8.5 estimator

### Classes

| | |
|---|---|
| *`EKF`* | Extended Kalman Filter. |
| *`Estimator`* | The Estimator base class. |
| *`MHE`* | Moving horizon estimator. |
| *`StateFeedback`* | Simple state-feedback “estimator”. |

#### 3.8.5.1 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.

### 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. |

#### 3.8.5.1.1 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.

#### 3.8.5.1.2 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.

#### 3.8.5.1.3 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*.

**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.

### 3.8.5.2 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*.

#### Methods

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

##### 3.8.5.2.1 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.

##### 3.8.5.2.2 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*.

**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.

### 3.8.5.3 MHE

**class** do_mpc.estimator.**MHE**(*model*, *p_est_list=[]*)
Moving horizon estimator. THE MHE estimator extends the *do_mpc.optimizer.Optimizer* base class (which is also used for the MPC controller), 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 *MHE.set_param()* to configure the *MHE*. See docstring for details.
2. Obtain the following variables from the class: `MHE._y_meas`, `MHE._y_calc`, `MHE._x_prev`, `MHE._x0`, `MHE._p_est_prev`, `MHE._p_est0`
3. Set the objective of the control problem with *MHE.set_objective()* or use the high-level interface, *MHE.set_default_objective()*

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

During runtime use *MHE.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* | Queries and sets the bounds of the optimization variables for the optimizer. |
| *MHE.opt_p_num* | Full MHE parameter vector. |
| *MHE.opt_x_num* | Full MHE solution and initial guess. |
| *MHE.scaling* | Queries and sets the scaling of the optimization variables for the optimizer. |

#### 3.8.5.3.1 bounds

Class attribute.

`MHE.`**`bounds`**

Queries and sets the bounds of the optimization variables for the optimizer. The `Optimizer.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*. |

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
```

(continues on next page)

(continued from previous page)

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

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

#### 3.8.5.3.2 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 *MHE.make_step()* method. However, you can set these values manually and directly call *MHE.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 *MHE.setup()*

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

#### 3.8.5.3.3 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 *MHE.setup()*

---

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

#### 3.8.5.3.4 scaling

Class attribute.

MHE.**scaling**
Queries and sets the scaling of the optimization variables for the optimizer. 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*. |

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

| | |
|---|---|
| *MHE.get_p_template* | Obtain the a numerical copy of the structure of the (not estimated) parameters. |
| *MHE.get_tvp_template* | The method returns a structured object with n_horizon elements, and a set of time varying parameters (as defined in model) for each of these instances. |
| *MHE.get_y_template* | Obtain the a numerical copy of the structure of the measurements for the set horizon. |
| *MHE.make_step* | Main method of the class during runtime. |
| *MHE.reset_history* | Reset the history of the optimizer. |
| *MHE.set_default_objective* | Wrapper function to set the suggested default MHE formulation: |
| *MHE.set_initial_guess* | Uses the current class attributes `_x0`, `_z0` and `_u0`, `_p_est0` to create an initial guess for the mhe. |
| *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 objective function for the MHE problem. |
| *MHE.set_p_fun* | Set the parameter function which is called at each MHE time step and returns the (not) estimated parameters. |
| *MHE.set_param* | Method to set the parameters of the *MHE* class. |
| *MHE.set_tvp_fun* | Set the tvp_fun which is called at each optimization step to get the current prediction of the 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. |

#### 3.8.5.3.5 get_p_template

Class method.

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

Obtain the a numerical copy of the structure of 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 `MHE.set_p_fun()`.

**Note:** The combination of `MHE.get_p_template()` and `MHE.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.

#### 3.8.5.3.6 get_tvp_template

Class method.

`do_mpc.estimator.MHE.`**`get_tvp_template`**(*self*)

The method returns a structured object with n_horizon elements, and a set of time varying parameters (as defined in 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 `Optimizer.set_tvp_fun()`. Use the combination of `Optimizer.get_tvp_template()` and `Optimizer.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.

#### 3.8.5.3.7 get_y_template

Class method.

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

Obtain the a numerical copy of the structure of the measurements for the set horizon. 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 `MHE.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-1` 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-1` 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.

#### 3.8.5.3.8 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 *do_mpc.optimizer.Optimizer.solve()* and updates the *do_mpc.data.Data* object.

**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.

#### 3.8.5.3.9 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.

#### 3.8.5.3.10 set_default_objective

Class method.

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

Wrapper function to set the suggested default MHE formulation:

$$\begin{aligned}\min_{\mathbf{x}_{0:N},\mathbf{u}_{0:N-1},\mathbf{w}_{0:N-1}} & \underbrace{(x_0-\tilde{x}_0)^T P_x (x_0-\tilde{x}_0)}_{\text{arrival cost states}} + \underbrace{(p_0-\tilde{p}_0)^T P_p (p_0-\tilde{p}_0)}_{\text{arrival cost params.}} \\ & + \sum_{k=0}^{n-1} \underbrace{(h(x_k,u_k,p_k)-y_k)^T P_{y,k} (h(x_k,u_k,p_k)-y_k) + w_k^T P_w w_k}_{\text{stage cost}} \\ \text{s.t.:} \quad & x_{k+1} = f(x_k,u_k,z_k,p_k,p_{tv,k}) + w_k\end{aligned}$$

Pass the weighting matrices $P_x$, $P_p$ and $P_y$ 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_y 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.

---

In the case that no parameters are estimated, the weighting matrix $P_p$ is not required. Furthermore, 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. 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_y** (*numpy.ndarray, casadi.SX, casadi.DM*) – Tuning matrix $P_y$ of dimension $m \times m$ ($y \in \mathbb{R}^m$)
- **P_p** (*numpy.ndarray, casadi.SX, casadi.DM*) – Tuning matrix $P_p$ of dimension $l \times l$ ($p_{\text{est}} \in \mathbb{R}^l$))

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

#### 3.8.5.3.11 set_initial_guess

Class method.

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

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 instances of x, u and z, p_est. The method is automatically evoked when calling the MHE.setup() method. However, if no initial values for x, u and z were supplied during setup, these default to zero.

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

#### 3.8.5.3.12 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.

**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.

#### 3.8.5.3.13 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 $\epsilon$, 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.

#### 3.8.5.3.14 set_objective

Class method.

do_mpc.estimator.MHE.**set_objective**(*self*, *stage_cost*, *arrival_cost*)

Set the objective function for the MHE problem. We suggest to formulate the MHE objective ($J$) such that:

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

Use the class attributes:

- mhe._y_meas as $y_k$
- mhe._y_calc as $h(x_k, u_k, p_k)$ (function is defined in *do_mpc.model.Model*)
- 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:
y_meas = mhe._y_meas
y_calc = mhe._y_calc

dy = y_meas.cat-y_calc.cat
stage_cost = dy.T@np.diag(np.array([1,1,1,20,20]))@dy

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 `MHE.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.

#### 3.8.5.3.15 set_p_fun

Class method.

do_mpc.estimator.MHE.**set_p_fun**(*self*, *p_fun*)

Set the parameter function which is called at each MHE time step and returns the (not) estimated parameters. The function must return a numerical CasADi structure, which can be retrieved with `MHE.get_p_template()`.

**Parameters** **p_fun** (*function*) – Parameter function.

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

#### 3.8.5.3.16 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:** `mhe.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 `MHE.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*) – Choose the collocation ni for continuous models with collocation as state discretization. Defaults to `1`.
- **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.

#### 3.8.5.3.17 set_tvp_fun

Class method.

do_mpc.estimator.MHE.**set_tvp_fun**(*self*, *tvp_fun*)

Set the tvp_fun which 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 `Optimizer.get_tvp_template()`.

```
# 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)
```

The method `Optimizer.set_tvp_fun()`. must be called prior to setup IF time-varying parameters are defined in the model.

**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.

#### 3.8.5.3.18 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 `MHE.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.

#### 3.8.5.3.19 setup

Class method.

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

The setup method finalizes the MHE creation. After this call, the *do_mpc.optimizer.Optimizer.solve()* method is applicable. The method wraps the following calls:

- `do_mpc.optimizer.Optimizer._setup_nl_cons()`
- `MHE._check_validity()`
- `MHE._setup_mhe_optim_problem()`
- `MHE.set_initial_guess()`

and sets the setup flag = True.

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

#### 3.8.5.3.20 solve

Class method.

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

Solves the optmization problem. The current time-step is defined by the parameters in the `self.opt_p_num` CasADi structured Data. These include the initial condition, the parameters, the time-varying paramters and the previous input. Typically, `self.opt_p_num` is prepared for the current iteration in the `.make_step()` (in MHE/MPC) method. It is, however, valid and possible to directly set paramters in `self.opt_p_num` before calling `.solve()`.

Solve updates the `opt_x_num`, and `lam_g_num` attributes of the class. In 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.

**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.

#### 3.8.5.4 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?

##### 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. |

###### 3.8.5.4.1 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.

###### 3.8.5.4.2 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.

###### 3.8.5.4.3 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*.

**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.

## 3.8.6 data

### Classes

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

#### 3.8.6.1 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. |

#### 3.8.6.1.1 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.

#### 3.8.6.1.2 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.

#### 3.8.6.1.3 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.

#### 3.8.6.1.4 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.

#### 3.8.6.2 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

...
```

(continues on next page)

(continued from previous page)

```
# 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. |

#### 3.8.6.2.1 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.

#### 3.8.6.2.2 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.

#### 3.8.6.2.3 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.

#### 3.8.6.2.4 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.

#### 3.8.6.2.5 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))
}

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. |

#### 3.8.6.3 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.

#### 3.8.6.4 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, optimizizer nor estimator.

**Returns** None

**Return type** None

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

### 3.8.7 graphics

#### Classes

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

#### 3.8.7.1 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])
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. |

##### 3.8.7.1.1 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.

##### 3.8.7.1.2 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()*. |

#### 3.8.7.1.3 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.

#### 3.8.7.1.4 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.

#### 3.8.7.1.5 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.

#### 3.8.7.1.6 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.

#### 3.8.7.1.7 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.

#### 3.8.7.1.8 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. |

#### 3.8.7.2 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** – (Optional) If supplied, the `fps` and ‘‘format argument are discarded. Use this to configure your own writer.

**Returns** None

#### 3.8.7.3 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.

CHAPTER 4

# Indices and tables

- genindex
- search

# Python Module Index

## d

# Index

## Symbols



## A



## B



## C



## D



## E



## G



## I



## L



## M

## O



## P



## R



## S

## T



## U



## W



## X



## Y



## Z