
do-mpc
Release 4.0.0

Mar 23, 2020

1	Example: Robust Multi-stage MPC	3
2	Next steps	5
3	Table of contents	7
3.1	Getting started: MPC	7
3.1.1	Example system	8
3.1.2	Creating the model	9
3.1.2.1	Model variables	10
3.1.2.2	Model parameters	10
3.1.2.3	Right-hand-side equation	11
3.1.3	Configuring the MPC controller	11
3.1.3.1	Optimizer parameters	11
3.1.3.2	Objective function	12
3.1.3.3	Constraints	12
3.1.3.4	Scaling	13
3.1.3.5	Uncertain Parameters	13
3.1.3.6	Setup	13
3.1.4	Configuring the Simulator	13
3.1.4.1	Simulator parameters	14
3.1.4.2	Uncertain parameters	14
3.1.4.3	Setup	14
3.1.5	Creating the control loop	14
3.1.5.1	Setting up the Graphic	15
3.1.5.2	Running the simulator	16
3.1.5.3	Running the optimizer	16
3.1.5.4	Running the control loop	19
3.1.6	Data processing	20
3.1.6.1	Saving and retrieving results	20
3.1.6.2	Animating results	20
3.2	Getting started: MHE	21
3.2.1	Creating the model	21
3.2.1.1	Model variables	22
3.2.1.2	Model measurements	22
3.2.1.3	Model parameters	22
3.2.1.4	Right-hand-side equation	22
3.2.2	Configuring the Moving Horizon Estimator	23

3.2.2.1	MHE parameters:	23
3.2.2.2	Cost Function	23
3.2.2.3	Fixed parameters	24
3.2.2.4	Bounds	25
3.2.2.5	Setup	25
3.2.3	Configuring the Simulator	25
3.2.3.1	Simulator parameters	25
3.2.3.2	Parameters	25
3.2.3.3	Setup	26
3.2.4	Creating the loop	26
3.2.4.1	Setting up the Graphic	26
3.2.4.2	Running the loop	27
3.2.5	MHE Advantages	29
3.3	License	31
3.4	Installation	33
3.4.1	Requirements	33
3.4.2	Option 1: PIP	33
3.4.3	Option 2: Clone from Github	33
3.4.4	HSL linear solver for IPOPT	33
3.4.4.1	Option 1: Pre-compiled binaries	34
3.4.4.1.1	Linux	34
3.4.4.2	Option 2: Compile from source	34
3.5	Credit	34
3.6	Structuring your project	35
3.6.1	template_model	35
3.6.2	template_mpc	36
3.6.3	template_simulator	37
3.6.4	template_estimator	37
3.6.5	main script	38
3.6.5.1	Initial state & guess	39
3.6.5.2	Graphics configuration	39
3.6.5.3	closed-loop	39
3.7	Debugging	40
3.7.1	Feasibility problems	40
3.7.1.1	Is the initial state feasible?	40
3.7.1.2	Which constraints are violated?	40
3.7.1.3	Use soft-constraints.	41
3.8	API Reference	41
3.8.1	model	42
3.8.1.1	Model	42
3.8.1.1.1	aux	43
3.8.1.1.2	p	44
3.8.1.1.3	tv _p	44
3.8.1.1.4	u	45
3.8.1.1.5	x	45
3.8.1.1.6	y	46
3.8.1.1.7	z	46
3.8.1.1.8	get_variables	47
3.8.1.1.9	set_expression	48
3.8.1.1.10	set_meas	49
3.8.1.1.11	set_rhs	49
3.8.1.1.12	set_variable	50
3.8.1.1.13	setup	51
3.8.1.1.14	setup_model	52

3.8.2	simulator	52
3.8.2.1	Simulator	52
3.8.2.1.1	get_p_template	53
3.8.2.1.2	get_tvp_template	53
3.8.2.1.3	make_step	53
3.8.2.1.4	reset_history	54
3.8.2.1.5	set_initial_state	54
3.8.2.1.6	set_p_fun	54
3.8.2.1.7	set_param	55
3.8.2.1.8	set_tvp_fun	56
3.8.2.1.9	setup	56
3.8.2.1.10	simulate	57
3.8.3	optimizer	57
3.8.3.1	Optimizer	58
3.8.3.1.1	bounds	58
3.8.3.1.2	scaling	58
3.8.3.1.3	get_tvp_template	59
3.8.3.1.4	reset_history	60
3.8.3.1.5	set_initial_state	60
3.8.3.1.6	set_nl_cons	61
3.8.3.1.7	set_tvp_fun	61
3.8.3.1.8	solve	62
3.8.4	controller	62
3.8.4.1	MPC	63
3.8.4.1.1	bounds	63
3.8.4.1.2	opt_p_num	64
3.8.4.1.3	opt_x_num	64
3.8.4.1.4	scaling	65
3.8.4.1.5	get_p_template	66
3.8.4.1.6	get_tvp_template	67
3.8.4.1.7	make_step	68
3.8.4.1.8	reset_history	68
3.8.4.1.9	set_initial_guess	68
3.8.4.1.10	set_initial_state	69
3.8.4.1.11	set_nl_cons	69
3.8.4.1.12	set_objective	70
3.8.4.1.13	set_p_fun	70
3.8.4.1.14	set_param	71
3.8.4.1.15	set_rterm	72
3.8.4.1.16	set_tvp_fun	73
3.8.4.1.17	set_uncertainty_values	74
3.8.4.1.18	setup	74
3.8.4.1.19	solve	75
3.8.5	estimator	75
3.8.5.1	EKF	75
3.8.5.1.1	make_step	76
3.8.5.1.2	reset_history	76
3.8.5.1.3	set_initial_state	76
3.8.5.2	Estimator	76
3.8.5.2.1	reset_history	77
3.8.5.2.2	set_initial_state	77
3.8.5.3	MHE	77
3.8.5.3.1	bounds	78
3.8.5.3.2	opt_p_num	79

3.8.5.3.3	opt_x_num	80
3.8.5.3.4	scaling	80
3.8.5.3.5	get_p_template	81
3.8.5.3.6	get_tvp_template	82
3.8.5.3.7	get_y_template	83
3.8.5.3.8	make_step	83
3.8.5.3.9	reset_history	84
3.8.5.3.10	set_default_objective	84
3.8.5.3.11	set_initial_guess	84
3.8.5.3.12	set_initial_state	85
3.8.5.3.13	set_nl_cons	85
3.8.5.3.14	set_objective	86
3.8.5.3.15	set_p_fun	87
3.8.5.3.16	set_param	87
3.8.5.3.17	set_tvp_fun	89
3.8.5.3.18	set_y_fun	89
3.8.5.3.19	setup	90
3.8.5.3.20	solve	90
3.8.5.4	StateFeedback	90
3.8.5.4.1	make_step	91
3.8.5.4.2	reset_history	91
3.8.5.4.3	set_initial_state	91
3.8.6	data	91
3.8.6.1	Data	91
3.8.6.1.1	export	92
3.8.6.1.2	init_storage	92
3.8.6.1.3	set_meta	92
3.8.6.1.4	update	92
3.8.6.2	load_results	93
3.8.6.3	save_results	94
3.8.7	graphics	94
3.8.7.1	Graphics	94
3.8.7.1.1	add_line	96
3.8.7.1.2	plot_predictions	96
3.8.7.1.3	plot_results	97
3.8.7.1.4	reset_axes	98
3.8.7.1.5	reset_prop_cycle	98
3.8.7.2	default_plot	98
4	Indices and tables	101
	Python Module Index	103
	Index	105

do-mpc is a comprehensive open-source toolbox for robust **Model Predictive Control (MPC)** and **Moving Horizon Estimation (MHE)**. It 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 at the DYN chair of the TU Dortmund by Sergio Lucia and Alexandru Tatulea. The development is continued at the IOT chair of the TU Berlin by Felix Fiedler and Sergio Lucia.

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](#).

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 to path:
import sys
sys.path.append('../..')

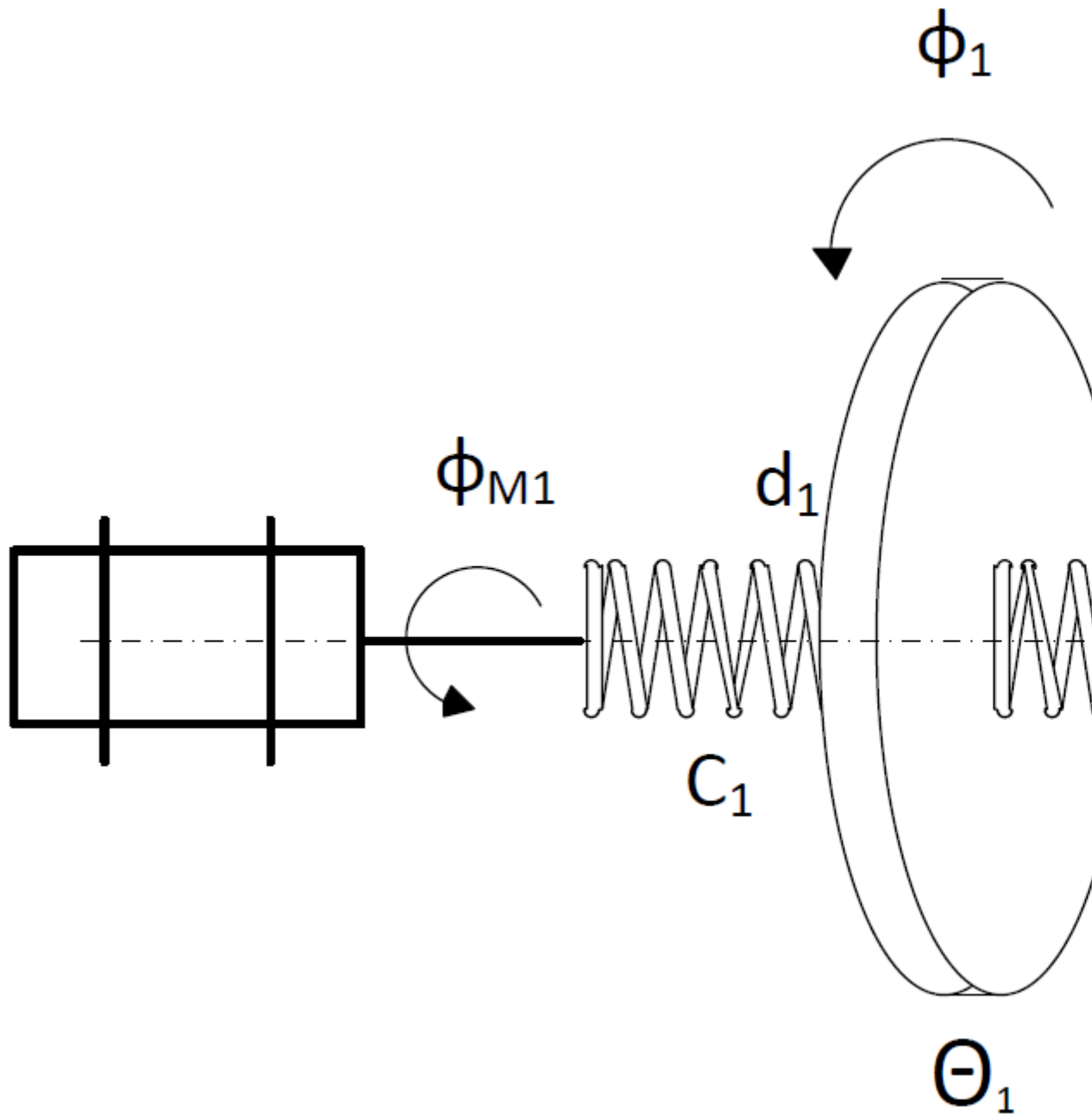
# Import do_mpc package:
import do_mpc
```

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

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

3.1.1 Example system

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



picted below:

Three rotating discs are connected via springs and we denote their angles as ϕ_1, ϕ_2, ϕ_3 . The two outermost discs are each connected to a stepper motor with additional springs. The stepper motor angles ($\phi_{m,1}$ and $\phi_{m,2}$) are used as inputs to the system. Relevant parameters of the system are the inertia Θ of the three discs, the spring constants c as well as the damping factors d .

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

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

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

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

3.1.2 Creating the model

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

In the case of 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, by 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:

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

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.

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

3.1.2.1 Model variables

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

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

```
[3]: phi_1 = model.set_variable(var_type='_x', var_name='phi_1', shape=(1,1))
phi_2 = model.set_variable(var_type='_x', var_name='phi_2', shape=(1,1))
phi_3 = model.set_variable(var_type='_x', var_name='phi_3', shape=(1,1))
# Variables can also be vectors:
dphi = model.set_variable(var_type='_x', var_name='dphi', shape=(3,1))
# Two states for the desired (set) motor position:
phi_m_1_set = model.set_variable(var_type='_u', var_name='phi_m_1_set')
phi_m_2_set = model.set_variable(var_type='_u', var_name='phi_m_2_set')
# Two additional states for the true motor position:
phi_1_m = model.set_variable(var_type='_x', var_name='phi_1_m', shape=(1,1))
phi_2_m = model.set_variable(var_type='_x', var_name='phi_2_m', shape=(1,1))
```

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

```
[4]: print('phi_1={}, with phi_1.shape={}'.format(phi_1, phi_1.shape))
print('dphi={}, with dphi.shape={}'.format(dphi, dphi.shape))
```

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

3.1.2.2 Model parameters

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

```
[5]: # 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
```


3.1.2.3 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_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:

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

```
[8]: 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)
```

```
[9]: 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_model()`:

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

After calling `model.setup_model()` 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)

```
[11]: 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`.

```
[12]: 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:

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

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

```
[13]: mterm = phi_1**2 + phi_2**2 + phi_3**2
lterm = phi_1**2 + phi_2**2 + phi_3**2

mpc.set_objective(mterm=mterm, lterm=lterm)
```

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

$$\Delta u_k = u_k - u_{k-1}$$

and automatically supply the solver with the previous solution of u_{k-1} for Δu_0 .

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

```
[14]: 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:

```
[15]: # 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).

```
[16]: 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:

```
[17]: 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([inertia_mass_1, inertia_mass_2, inertia_mass_3])
```

We provide a list to the method `optimizer.set_uncertain_parameter()`, where each entry is a `numpy.ndarray` with the 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.

```
[18]: 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 initialize the class with the previously defined `model`:

```
[19]: 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.

```
[20]: # 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 evaluated at each time to obtain the current values. **do-mpc** requires this function to have a specific return structure which we obtain first by calling:

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

This object is a CasADi structure:

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

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

which can be indexed with the following keys:

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

```
[23]: ['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:

```
[24]: 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:

```
[25]: 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.

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

```
[27]: 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()`).

```
[28]: 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 `optimizer.data`, `simulator.data` (and if applicable `estimator.data`) objects.

We start by importing matplotlib:

```
[29]: 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:

```
[30]: graphics = do_mpc.graphics.Graphics()
```

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`:

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

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

Note that we are only configuring the graphics module without showing anything just yet.

```
[32]: %%capture
# Plot the angle positions (phi_1, phi_2, phi_2) on the first axis:
graphics.add_line(var_type='_x', var_name='phi_1', axis=ax[0])
graphics.add_line(var_type='_x', var_name='phi_2', axis=ax[0])
graphics.add_line(var_type='_x', var_name='phi_3', axis=ax[0])
ax[0].set_ylabel('angle position [rad]')

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

(continues on next page)

(continued from previous page)

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

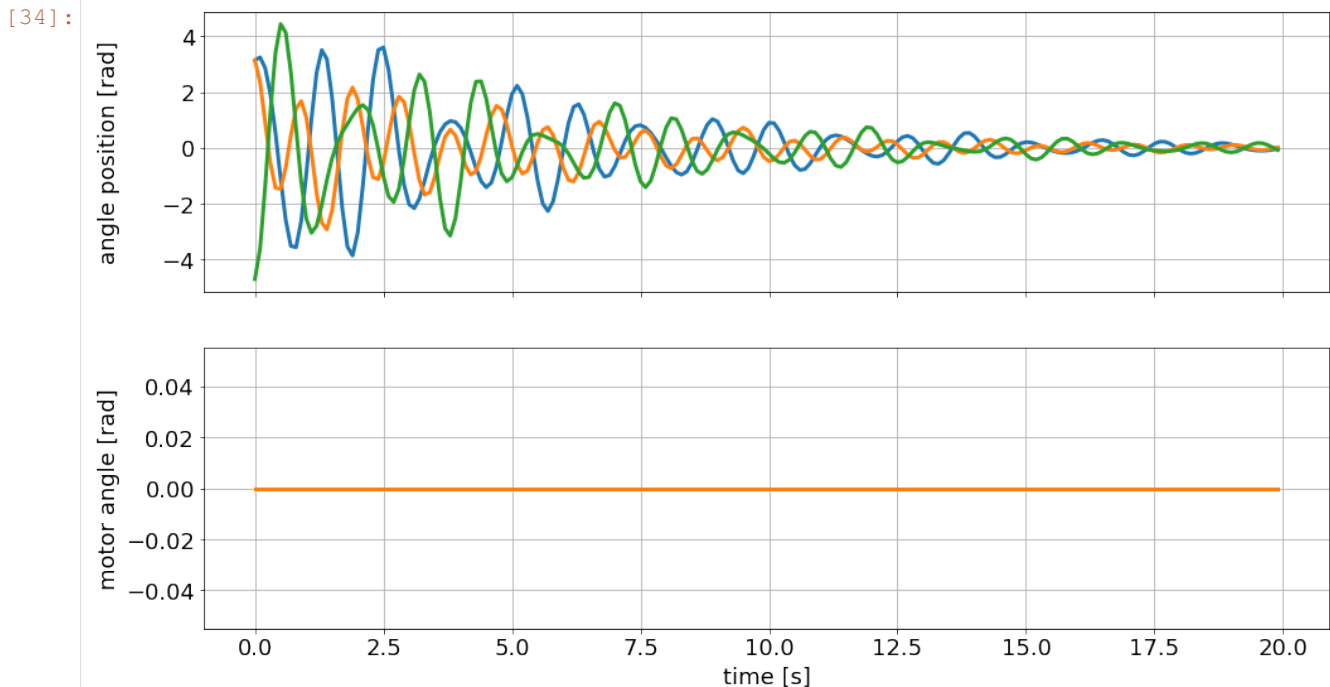
3.1.5.2 Running the simulator

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:

```
[33]: 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:

```
[34]: graphics.plot_results(simulator.data)
      # Show the figure:
      fig
```



As desired, the motor angle (input) is constant at zero and the oscillating masses slowly come to a rest. Our control goal is to significantly shorten the time until the discs are stationary.

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

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):

```
[35]: 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 3.12.12, 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	1.73e-14	3.95e+00	-1.0	3.56e+00	-4.5	1.96e-01	1.00e+00f	1
3	1.6039802e+02	1.87e-14	3.82e-01	-1.0	3.43e+00	-5.0	5.14e-01	1.00e+00f	1
4	1.3046012e+02	1.55e-14	7.36e-02	-1.0	2.94e+00	-5.4	7.75e-01	1.00e+00f	1
5	1.1452477e+02	2.58e-14	1.94e-02	-1.7	2.62e+00	-5.9	8.44e-01	1.00e+00f	1
6	1.1247422e+02	2.33e-14	7.23e-03	-2.5	9.17e-01	-6.4	8.27e-01	1.00e+00f	1
7	1.1235000e+02	1.42e-14	4.88e-08	-2.5	3.56e-01	-6.9	1.00e+00	1.00e+00f	1
8	1.1230585e+02	1.42e-14	8.91e-09	-3.8	1.95e-01	-7.3	1.00e+00	1.00e+00f	1
9	1.1229857e+02	1.95e-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	1.60e-14	6.08e-09	-5.7	1.20e+00	-8.3	1.00e+00	1.00e+00f	1
11	1.1229831e+02	1.75e-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...:	3.2478880696018564e-13	3.2478880696018564e-13
Constraint violation...:	1.7541523789077473e-14	1.7541523789077473e-14
Complementarity...:	4.2481089750730680e-09	4.2481089750730680e-09
Overall NLP error...:	4.2481089750730680e-09	4.2481089750730680e-09

```

Number of objective function evaluations = 12
Number of objective gradient evaluations = 12
Number of equality constraint evaluations = 12
Number of inequality constraint evaluations = 0
Number of equality constraint Jacobian evaluations = 12
Number of inequality constraint Jacobian evaluations = 0
Number of Lagrangian Hessian evaluations = 11
Total CPU secs in IPOPT (w/o function evaluations) = 2.790

```

(continues on next page)

(continued from previous page)

```
Total CPU secs in NLP function evaluations      =      0.017
```

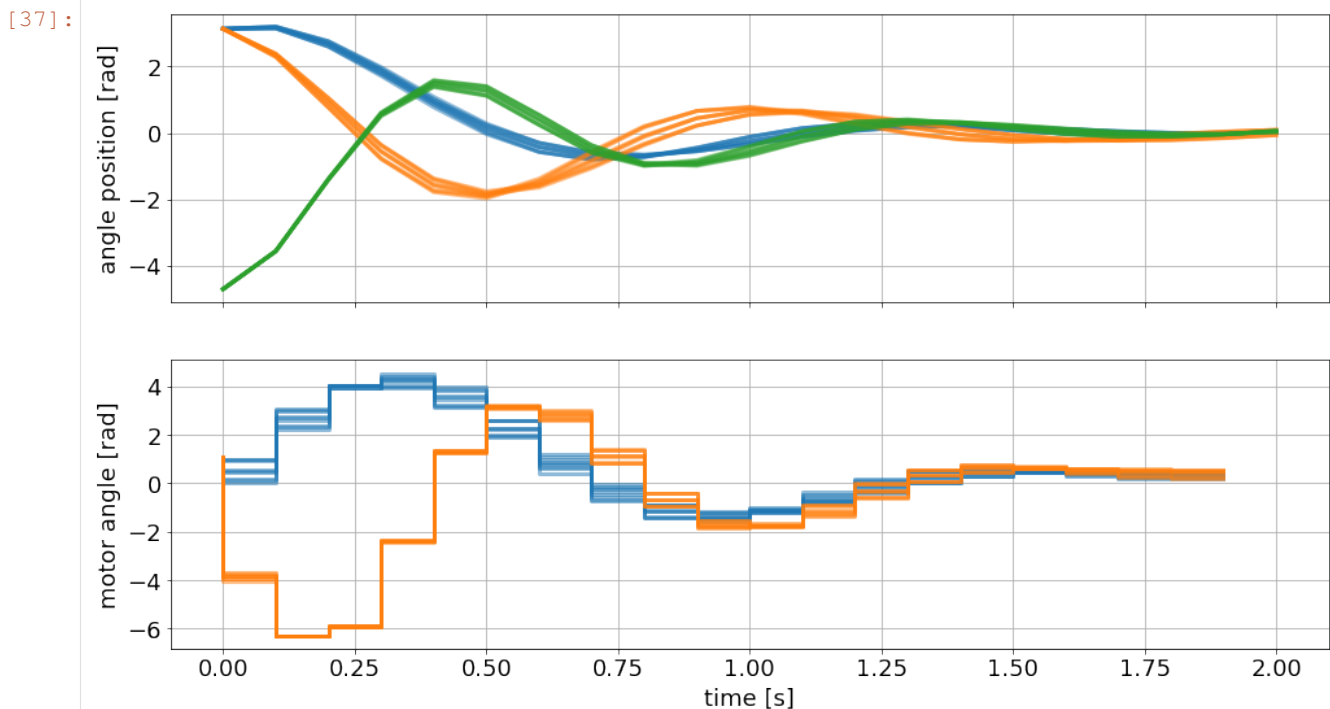
```
EXIT: Optimal Solution Found.
```

	t_proc [s]	t_wall [s]	n_eval
S	3.25	1.63	1
nlp_f	0.000461	0.000235	12
nlp_g	0.00565	0.00275	12
nlp_grad	0.000843	0.00042	1
nlp_grad_f	0.00142	0.00073	13
nlp_hess_l	0.000392	0.000186	11
nlp_jac_g	0.00948	0.00465	13

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

The configured graphics module is very useful to investigate the obtained solution by plotting the predicted system behavior. We simply need to call:

```
[37]: graphics.reset_axes()
lines = graphics.plot_predictions(mpc.data, alpha=0.5)
# Autoscale the axis:
ax[0].relim()
ax[0].autoscale()
ax[1].relim()
ax[1].autoscale()
# Show the figure:
fig
```



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.

As a conclusion, the results look as expected and can be assessed in the closed-loop application.

3.1.5.4 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:

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

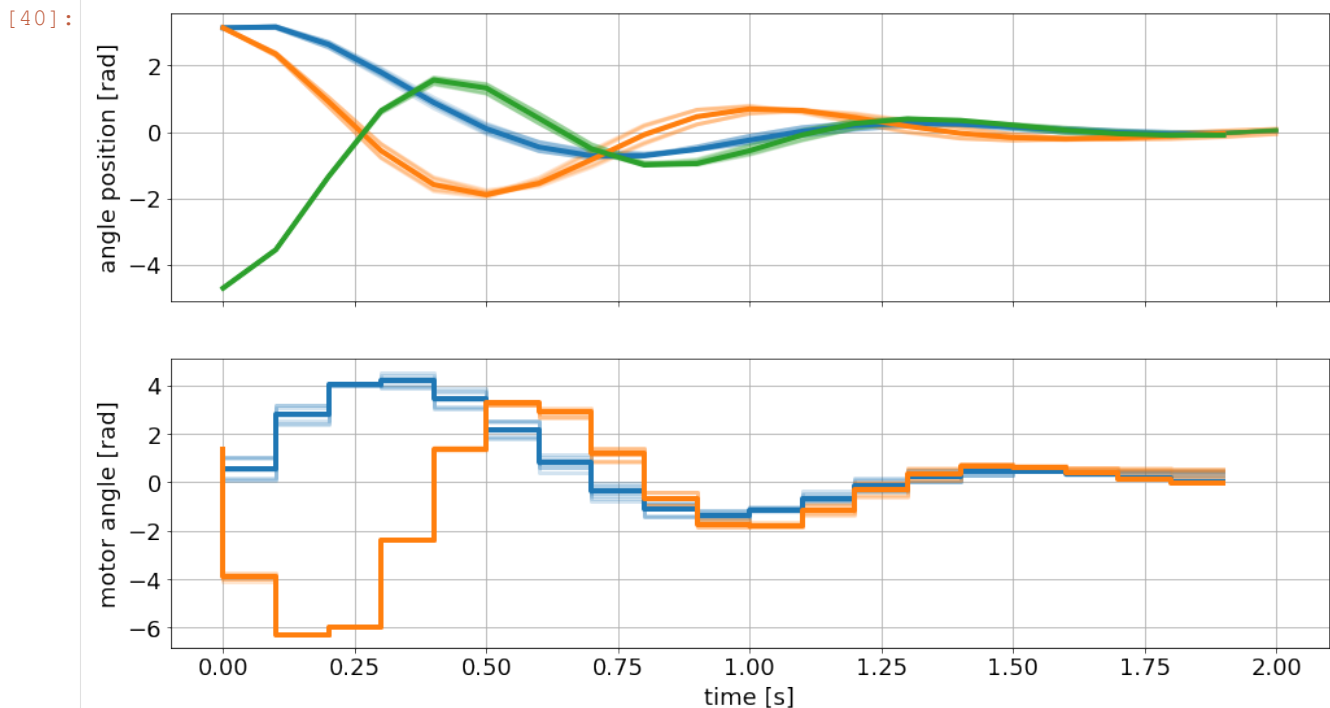
This is the main-loop. We run 20 steps, which is identical to the prediction horizon. Note that we use “capture” again, to suppress the output from IPOPT.

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

```
[39]: %%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:

```
[40]: # Clear previous lines
      graphics.reset_axes()
      # Plot predictions from t=0
      pred_lines = graphics.plot_predictions(mpc.data, t_ind=0, linewidth=3, alpha=.2)
      # Plot results until current time
      res_lines = graphics.plot_results(simulator.data, linewidth=4, alpha=1)
      fig
```



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

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:

```
[41]: 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.

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

We investigate the content of the newly created folder:

```
[43]: import os
      os.listdir('./results/')
```

```
[43]: ['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:

```
[44]: save_results([mpc, simulator])
      os.listdir('./results/')
```

```
[44]: ['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.

```
[45]: 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 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:

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

def update(t_ind):
    graphics.reset_axes()
    graphics.plot_results(simulator.data, t_ind=t_ind, linewidth=5)
    graphics.plot_predictions(optimizer.data, t_ind=t_ind, linestyle='--',
    ↪linewidth=3)
```

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. With the `results` dict from above we have to write:

```
[49]: def update(t_ind):
    graphics.reset_axes()
    graphics.plot_results(results['simulator'], t_ind=t_ind, linewidth=5)
    graphics.plot_predictions(results['mpc'], t_ind=t_ind, linestyle='--',
    ↪linewidth=3)
```

```
[50]: 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 to path:
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],
```

(continues on next page)

(continued from previous page)

```

)

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 Cost Function

The most important step of the configuration is to define the MHE cost function. The **do-mpc** MHE implementation allows maximum flexibility in that sense. We retrieve the variables `_y_meas`, `_y_calc`, `_x_prev`, `_x`, `_p_est`, `_p_est_prev` and formulate the stage cost (as a function of `_y_meas`, `_y_calc`) and the arrival cost (as a function

of $(_x_prev, _x, _p_est, _p_est_prev)$. Typically, we want to formulate the following objective:

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

```
[10]: y_meas = mhe._y_meas
      y_calc = mhe._y_calc

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

      stage_cost = dy.T@P_y@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

      P_x = np.eye(8)
      P_p = np.eye(1)

      arrival_cost = 1e-4*dx.T@P_x@dx + 1e-4*dp.T@P_p@dp

      mhe.set_objective(stage_cost, arrival_cost)
```

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 evaluated at each time to obtain the current values. **do-mpc** requires this function to have a specific return structure which we obtain first by calling:

```
[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:

```
[26]: graphics = do_mpc.graphics.Graphics()
```

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.

Note that we are only configuring the graphics module without showing anything just yet.

```
[28]: %%capture
# Plot the angle positions (phi_1, phi_2, phi_2) on the first axis:
graphics.add_line(var_type='_x', var_name='phi', axis=ax[0])
ax[0].set_ylabel('angle \n position [rad]')

graphics.add_line(var_type='_x', var_name='dphi', axis=ax[1])
ax[1].set_ylabel('angular \n velocity [rad/s]')

# Plot the set motor positions (phi_m_1_set, phi_m_2_set) on the second axis:
graphics.add_line(var_type='_u', var_name='phi_m_set', axis=ax[2])
ax[2].set_ylabel('motor \n angle [rad]')

ax[2].set_xlabel('time [s]')

graphics.add_line(var_type='_p', var_name='Theta_1', axis=ax_p)
graphics.add_line(var_type='_p', var_name='Theta_2', axis=ax_p)
graphics.add_line(var_type='_p', var_name='Theta_3', axis=ax_p)
```

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:

```
[29]: 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.

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

u0 = np.zeros((2,1))
for i in range(50):
    u0 = random_u(u0) # Control input
```

(continues on next page)

(continued from previous page)

```

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

```

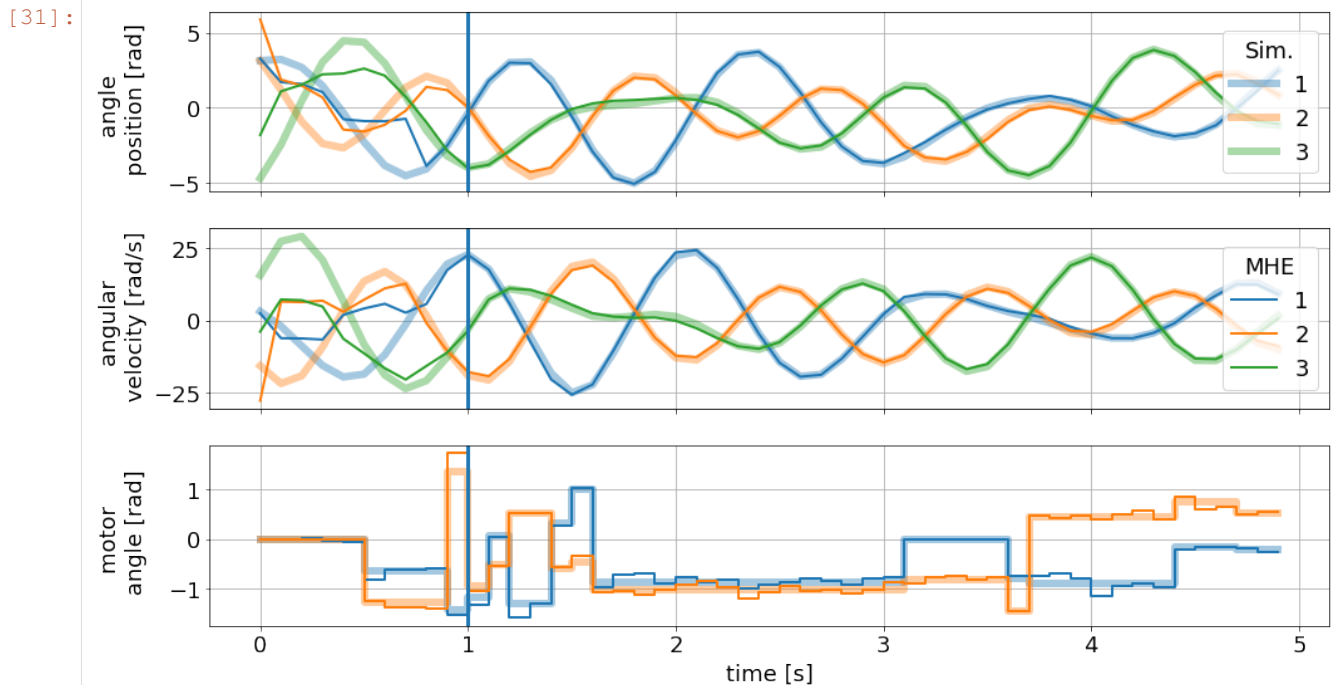
We can visualize the resulting trajectory with the pre-defined graphic:

```

[31]: graphics.reset_axes()
sim_lines = graphics.plot_results(simulator.data, linewidth = 6, alpha = 0.4,
↳ linestyle='-')
ax[0].legend(sim_lines[:3], '123', title='Sim.', loc='center right')
mhe_lines = graphics.plot_results(mhe.data, linewidth = 2, alpha = 1, linestyle='-')
ax[1].legend(mhe_lines[:3], '123', title='MHE', loc='center right')
# 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

```

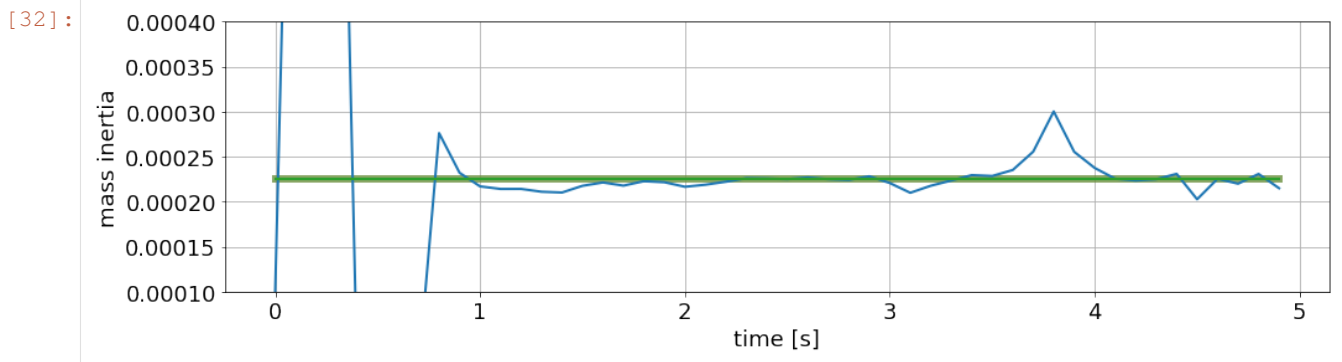


Parameter estimation:

```

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

```



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:

```
[33]: 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.

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

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

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

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

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

u0 = np.zeros((2,1))
for i in range(50):
    u0 = random_u(u0) # Control input
    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:

```
[37]: graphics.reset_axes()
sim_lines = graphics.plot_results(simulator.data, linewidth = 6, alpha =0.4,
↳linestyle='-')
ax[0].legend(sim_lines[:3], '123', title='Sim.', loc='center right')
mhe_lines = graphics.plot_results(mhe.data, linewidth = 2, alpha = 1, linestyle='-')
ax[1].legend(mhe_lines[:3], '123', title='MHE', loc='center right')
# Mark the time after a full horizon is available to the MHE.
```

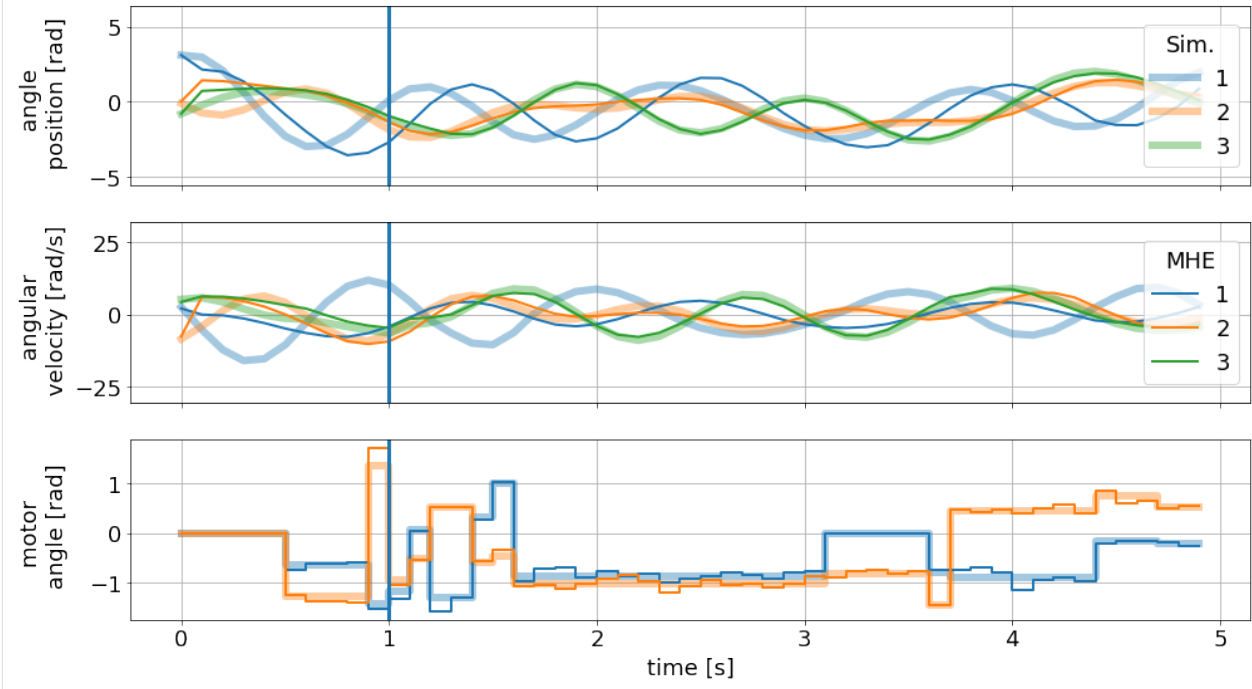
(continues on next page)

(continued from previous page)

```
ax[0].axvline(1)
ax[1].axvline(1)
ax[2].axvline(1)
```

```
# Show the figure:
fig
```

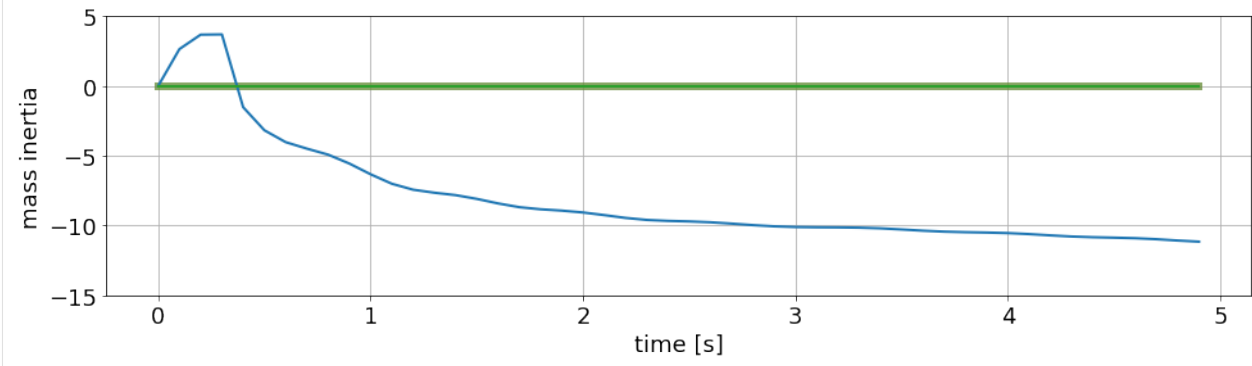
[37]:



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

```
[51]: ax_p.set_ylim(-15, 5)
ax_p.set_ylabel('mass inertia')
ax_p.set_xlabel('time [s]')
fig_p
```

[51]:



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

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

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

0. Additional Definitions.

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

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

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

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

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

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

1. Exception to Section 3 of the GNU GPL.

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

2. Conveying Modified Versions.

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

- a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or
- b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

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:

Download our examples [here](#). You can also get the [MPC](#) Jupyter Notebook and the [MHE](#) Jupyter Notebook from the introduction.

All these resources can also be obtained by simply cloning the repository as shown below.

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 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 occurred:  
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 occurred:  
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

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](#))

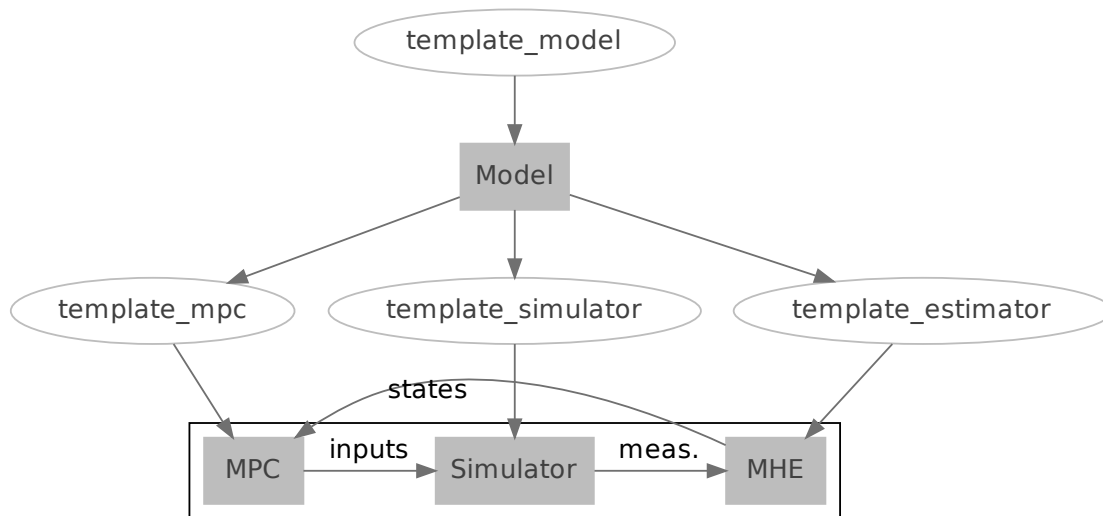


Fig. 1: Project structure

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:

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

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

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

$$m(x_k, u_k, z_k, p_k, p_k^v) \leq m_{\text{ub}} \forall k = 0, \dots, N$$

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,
```

(continues on next page)

(continued from previous page)

```

    'n_robust': 1,
    't_step': 0.005,
    ...
}
mpc.set_param(**setup_mpc)

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

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

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

mpc.setup()

return mpc

```

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
    # 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:

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

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)
    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(ub_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:

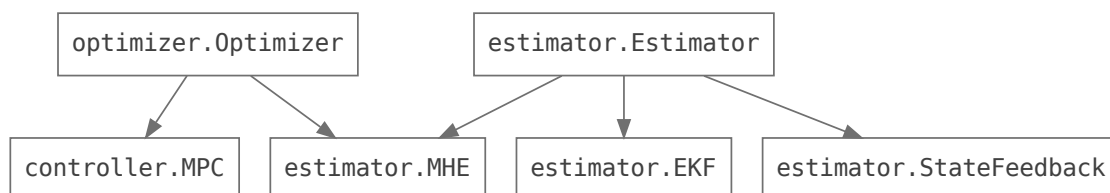


Fig. 2: Class inheritance. Click on classes for more information.

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 continuous 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

<i>Model.aux</i>	Auxiliary expressions.
<i>Model.p</i>	Static parameters.
<i>Model.tvp</i>	Time-varying parameters.
<i>Model.u</i>	Inputs.
<i>Model.x</i>	Dynamic states.
<i>Model.y</i>	Measurements.
<i>Model.z</i>	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 directly. 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 directly. 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 directly. 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 directly. Use `set_variable` instead.

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

3.8.1.1.5 `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
```

(continues on next page)

(continued from previous page)

```
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 directly. Use `set_variable` instead.

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

3.8.1.1.6 `y`

Class attribute.

`Model.y`

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

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

Example:

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

Usefull CasADi symbolic structure methods:

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

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

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

3.8.1.1.7 `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 directly. Use `set_variable` instead.

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

Methods

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

3.8.1.1.8 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.

3.8.1.1.9 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.

3.8.1.1.10 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.

3.8.1.1.11 set_rhs

Class method.

`do_mpc.model.Model.set_rhs` (*self*, *var_name*, *expr*)

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)
```

Parameters

- **var_name** (*string*) – Name of any of the previously defined states with: `model.set_variable('states', [NAME])`
- **expr** (*CasADi SX or MX*) – CasADi SX or MX function depending on `_x`, `_u`, `_z`, `_tvp`, `_p`.

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.

3.8.1.1.12 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	<code>_x</code>	Required
inputs	<code>_u</code>	Required
algebraic	<code>_z</code>	Optional
parameter	<code>_p</code>	Optional
timevarying_parameter	<code>_tvp</code>	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.

3.8.1.13 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.

3.8.1.1.14 `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

<code>Simulator.get_p_template</code>	Obtain output template for <code>set_p_fun()</code> .
<code>Simulator.get_tvp_template</code>	Obtain the output template for <code>set_tvp_fun()</code> .
<code>Simulator.make_step</code>	Main method of the simulator class during control runtime.
<code>Simulator.reset_history</code>	Reset the history of the simulator.
<code>Simulator.set_initial_state</code>	Set the initial state of the simulator.
<code>Simulator.set_p_fun</code>	Function to set the function which gives the values of the parameters.
<code>Simulator.set_param</code>	Set the parameters for the simulator.
<code>Simulator.set_tvp_fun</code>	Function to set the function which gives the values of the time-varying parameters.
<code>Simulator.setup</code>	Sets up the simulator and finalizes the simulator configuration.
<code>Simulator.simulate</code>	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 z_0 can also be supplied, if they are defined in the model but are only used as an initial 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_nnext`

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 initial 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')
```

To integrate the ODE or evaluate the discrete dynamics, the simulator needs to obtain the numerical values of these parameters at each timestep. In the most general case, these values can change, which is why we need to supply a function that can be evaluated at each time to obtain the current values. **do-mpc** requires this function to have a specific return structure which we obtain first by calling:

```
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')
```

To integrate the ODE or evaluate the discrete dynamics, the simulator needs to obtain the numerical values of these parameters at each timestep. In the most general case, these values can change, which is why we need to supply a function that can be evaluated at each time to obtain the current values. **do-mpc** requires this function to have a specific return structure which we obtain first by calling:

```
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 class for the optimization based state estimation (MHE) and predictive controller (MPC).

3.8.3.1 Optimizer

class `do_mpc.optimizer.Optimizer`

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

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

Attributes

<code>Optimizer.bounds</code>	Queries and sets the bounds of the optimization variables for the optimizer.
<code>Optimizer.scaling</code>	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 separated by commas) must contain at least the following elements:

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

Further indices are possible (but not necessary) 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 separated by commas) must contain at least the following elements:

order	index name	valid options
1	variable type	<code>_x</code> , <code>_u</code> and <code>_z</code> (and <code>_p_est</code> for MHE)
2	variable name	Names defined in <code>do_mpc.model</code> .

Further indices are possible (but not necessary) 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

<code>Optimizer.get_tvp_template</code>	The method returns a structured object with <code>n_horizon</code> elements, and a set of time varying parameters (as defined in model) for each of these instances.
<code>Optimizer.reset_history</code>	Reset the history of the optimizer.
<code>Optimizer.set_initial_state</code>	Set the initial state of the optimizer.
<code>Optimizer.set_nl_cons</code>	Introduce new constraint to the class.
<code>Optimizer.set_tvp_fun</code>	Set the <code>tvp_fun</code> which is called at each optimization step to get the current prediction of the time-varying parameters.
<code>Optimizer.solve</code>	Solves the optimization 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')
```

(continues on next page)

(continued from previous page)

```

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_initial_guess=True)`

Set the initial 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_initial_guess** (*bool* (*, optional*)) – Setting the initial state also updates the initial 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:

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

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

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

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 optimization 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 parameters 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 parameters 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 `assertion` – 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

<i>MPC</i>	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

<code>MPC.bounds</code>	Queries and sets the bounds of the optimization variables for the optimizer.
<code>MPC.opt_p_num</code>	Full MPC parameter vector.
<code>MPC.opt_x_num</code>	Full MPC solution and initial guess.
<code>MPC.scaling</code>	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 separated by comas) must contain atleast the following elements:

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

Further indices are possible (but not necessary) 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 know 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 know 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 separated by commas) must contain atleast the following elements:

order	index name	valid options
1	variable type	<code>_x</code> , <code>_u</code> and <code>_z</code> (and <code>_p_est</code> for MHE)
2	variable name	Names defined in <code>do_mpc.model</code> .

Further indices are possible (but not necessary) 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

<code>MPC.get_p_template</code>	Obtain output template for <code>set_p_fun()</code> .
<code>MPC.get_tvp_template</code>	The method returns a structured object with <code>n_horizon</code> elements, and a set of time varying parameters (as defined in model) for each of these instances.
<code>MPC.make_step</code>	Main method of the class during runtime.
<code>MPC.reset_history</code>	Reset the history of the optimizer.
<code>MPC.set_initial_guess</code>	Initial guess for optimization variables.
<code>MPC.set_initial_state</code>	Set the initial state of the optimizer.
<code>MPC.set_nl_cons</code>	Introduce new constraint to the class.
<code>MPC.set_objective</code>	Sets the objective of the optimal control problem (OCP).
<code>MPC.set_p_fun</code>	Set parameter function for MPC.
<code>MPC.set_param</code>	Set the parameters of the <code>MPC</code> class.
<code>MPC.set_rterm</code>	Set the penalty factor for the inputs.
<code>MPC.set_tvp_fun</code>	Set the <code>tvp_fun</code> which is called at each optimization step to get the current prediction of the time-varying parameters.
<code>MPC.set_uncertainty_values</code>	Define scenarios for the uncertain parameters.
<code>MPC.setup</code>	Setup the MPC class.
<code>MPC.solve</code>	Solves the optimization 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])
```

(continues on next page)

(continued from previous page)

```

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 x_0 .

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 x_0 (*numpy.ndarray*) – Current state of the system.

Returns u_0

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_initial_guess=True*)

Set the initial 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_initial_guess`** (*bool (, optional)*) – Setting the initial state also updates the initial 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:

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

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

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

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 suppress 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 penalty factor for the inputs. Call this function with keyword argument referring 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 $k = 0, \dots, N$ 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.

3.8.4.16 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])
```

(continues on next page)

(continued from previous page)

```

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*)

Define scenarios for the uncertain parameters. High-level API method to conveniently set all possible scenarios for multistage MPC, given a list of uncertainty values. 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 optimizer configuration:
alpha_var = np.array([1., 0.9, 1.1])
beta_var = np.array([1., 1.05])
optimizer.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 `uncertainty_values` (*list*) – List of lists / numpy arrays with the same number of elements as number of parameters in model.

Raises `assertion` – uncertainty values must be of type list

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 optimization 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 parameters 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 parameters 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 `AssertionError` – 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

<code>EKF</code>	Extended Kalman Filter.
<code>Estimator</code>	The Estimator base class.
<code>MHE</code>	Moving horizon estimator.
<code>StateFeedback</code>	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

<code>EKF.make_step</code>	Main method during runtime.
<code>EKF.reset_history</code>	Reset the history of the estimator
<code>EKF.set_initial_state</code>	Set the initial 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 initial 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

<code>Estimator.reset_history</code>	Reset the history of the estimator
<code>Estimator.set_initial_state</code>	Set the initial 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 initial 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

<code>MHE.bounds</code>	Queries and sets the bounds of the optimization variables for the optimizer.
<code>MHE.opt_p_num</code>	Full MHE parameter vector.
<code>MHE.opt_x_num</code>	Full MHE solution and initial guess.
<code>MHE.scaling</code>	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 separated by commas) must contain at least the following elements:

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

Further indices are possible (but not necessary) 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.

3.8.5.3.2 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 know 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 know 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 separated by commas) must contain at least the following elements:

order	index name	valid options
1	variable type	<code>_x</code> , <code>_u</code> and <code>_z</code> (and <code>_p_est</code> for MHE)
2	variable name	Names defined in <code>do_mpc.model</code> .

Further indices are possible (but not necessary) 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

<code>MHE.get_p_template</code>	Obtain the a numerical copy of the structure of the (not estimated) parameters.
<code>MHE.get_tvp_template</code>	The method returns a structured object with <code>n_horizon</code> elements, and a set of time varying parameters (as defined in model) for each of these instances.
<code>MHE.get_y_template</code>	Obtain the a numerical copy of the structure of the measurements for the set horizon.
<code>MHE.make_step</code>	Main method of the class during runtime.
<code>MHE.reset_history</code>	Reset the history of the optimizer.
<code>MHE.set_default_objective</code>	Wrapper function to set the suggested default MHE objective:
<code>MHE.set_initial_guess</code>	Uses the current class attributes <code>_x0</code> , <code>_z0</code> and <code>_u0</code> , <code>_p_est0</code> to create an initial guess for the mhe.
<code>MHE.set_initial_state</code>	Set the intial state of the optimizer.
<code>MHE.set_nl_cons</code>	Introduce new constraint to the class.
<code>MHE.set_objective</code>	Set the objective function for the MHE problem.
<code>MHE.set_p_fun</code>	Set the parameter function which is called at each MHE time step and returns the (not) estimated parameters.
<code>MHE.set_param</code>	Method to set the parameters of the <code>MHE</code> class.
<code>MHE.set_tvp_fun</code>	Set the <code>tvp_fun</code> which is called at each optimization step to get the current prediction of the time-varying parameters.
<code>MHE.set_y_fun</code>	Set the measurement function.
<code>MHE.setup</code>	The setup method finalizes the MHE creation.
<code>MHE.solve</code>	Solves the optimization 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, such that `k=0` is the “oldest measurement” and `k=N_horizon-1` is the newest measurement.

By default, the following measurement function is chosen:

```
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 x_0 , 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)`

Wrapper function to set the suggested default MHE objective:

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

Pass the weighting matrices P_x , P_p and P_y . The matrices must be of appropriate dimension (and numpy nd.arrays). In the case that no parameters are estimated, the weighting matrix P_p is not required.

Note: Use `MHE.set_objective()` as a low-level alternative for this method, if you want to use a custom objective function.

Parameters

- **P_x** (`numpy.ndarray`) – Tuning matrix P_x of dimension $n \times n$ ($x \in \mathbb{R}^n$)
- **P_y** (`numpy.ndarray`) – Tuning matrix P_y of dimension $m \times m$ ($y \in \mathbb{R}^m$)
- **P_p** (`numpy.ndarray, optional`) – Tuning matrix P_p of dimension $l \times l$ ($p_{extest} \in \mathbb{R}^l$)

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.estimate.MHE.set_initial_state` (*self*, *x0*, *p_est0=None*, *reset_history=False*, *set_initial_guess=True*)

Set the initial 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_initial_guess`** (*bool (, optional)*) – Setting the initial state also updates the initial 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.estimate.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:

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

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

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

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
```

(continues on next page)

(continued from previous page)

```

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 suppress 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 optimization 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 `assertion` – 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

<code>StateFeedback.make_step</code>	Return the measurement <code>y0</code> .
<code>StateFeedback.reset_history</code>	Reset the history of the estimator
<code>StateFeedback.set_initial_state</code>	Set the inital state of the estimator.

3.8.5.4.1 make_step

Class method.

`do_mpc.estimator.StateFeedback.make_step(self, y0)`
Return the measurement y_0 .

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 initial 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.

3.8.6.1 Data

class `do_mpc.data.Data(model)`

do-mpc data container. An instance of this class is created for all active **do-mpc** classes, e.g. `do_mpc.simulator.Simulator`, `do_mpc.controller.MPC`, `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.

Methods

<i>Data.export</i>	The export method returns a dictionary of the stored data.
<i>Data.init_storage</i>	Create new (empty) arrays for all variables.
<i>Data.set_meta</i>	Set meta data for the current instance of the data object.
<i>Data.update</i>	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.

Functions

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

3.8.6.2 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.

3.8.6.3 save_results

`do_mpc.data.save_results` (*save_list*, *result_name='results'*, *result_path='./results/'*, *overwrite=False*)

Exports the data objects from the **do-mpc** modules in *save_list* as a pickled file. Supply any, all or a selection of (as a list):

- `do_mpc.controller.MPC`
- `do_mpc.simulator.Simulator`
- `do_mpc.estimator.Estimator`

These objects can be used in post-processing to create graphics with the `do_mpc.graphics_backend`.

Parameters

- **save_list** (*list*) – List of the objects to be stored.
- **result_name** (*string, optional*) – Name of the result file, defaults to 'result'.
- **result_path** (*string, optional*) – Result path, defaults to './results/'.
- **overwrite** (*bool, optional*) – Option to overwrite existing results, defaults to False. Index will be appended if file already exists.

Raises

- **assertion** – *save_list* must be a list.
- **assertion** – *result_name* must be a string.
- **assertion** – *result_path* must be a string.
- **assertion** – *overwrite* must be boolean.
- **Exception** – *save_list* contains object which is neither `do_mpc` simulator, optimizer 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`

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. User defined graphics are configured prior to plotting results, e.g.:

```

# Initialize graphic:
graphics = do_mpc.graphics.Graphics()

# 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. Note that `Graphics.add_line()` does not create a graphic or plots results. The graphic is obtained with the `Graphics.plot_results()` method, which takes a `do_mpc.data.Data` object as input. Each module (e.g.: `do_mpc.simulator.Simulator`, `do_mpc.estimator.MHE`, `do_mpc.controller.MPC`) has its own `do_mpc.data.Data` object. Furthermore, the module contains the `Graphics.plot_predictions()` method, which can be used to show the predicted trajectories for the `do_mpc.controller.MPC`.

Note: A high-level API for obtaining a configured `Graphics` module is the `default_plot()` function. Use this function and the obtained `Graphics` module in the development process.

Animations can be setup with the following loop:

```

for k in range(50):
    # do-mpc loop:
    u0 = mpc.make_step(x0)
    y_next = simulator.make_step(u0)
    x0 = estimator.make_step(y_next)

    # Reset and replot results and predictions with Graphics:
    graphics.reset_axes()
    graphics.plot_results(mpc.data, linewidth=3)
    # The second and third argument can be omitted if this information is stored_
    ↪ in the data object (optional setting).
    graphics.plot_predictions(mpc.data, mpc.opt_x_num, mpc.opt_aux_num)
    plt.show()

```

Methods

<code>Graphics.add_line</code>	add_line is called during setting up the <code>Graphics</code> class.
<code>Graphics.plot_predictions</code>	Plots the predicted trajectories for the plot configuration.
<code>Graphics.plot_results</code>	Plots the results stored in the passed data object for the plot configuration.
<code>Graphics.reset_axes</code>	Clears the lines on all axes which were passed with <code>Graphics.add_line()</code> .
<code>Graphics.reset_prop_cycle</code>	Resets the property cycle for all axes which were passed with <code>Graphics.add_line()</code> .

3.8.7.1.1 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`.

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.*) – Variable name. Must reference the names defined in the model for the given variable type.
- **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.

3.8.7.1.2 plot_predictions

Class method.

`do_mpc.graphics.Graphics.plot_predictions` (*self*, *data*, *opt_x_num=None*, *opt_aux_num=None*, *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 can be passed either as the optional argument (`opt_x_num`)

or they are part of the data structure, if the optimizer was set to store the optimal solution (see `:py:func: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 is mostly used in post-processing for animations. Optionally pass an arbitrary number of valid `pyplot.plot` arguments (e.g. `linewidth, color, alpha`), which is applied to ALL lines.

Parameters

- **data** (*do-mpc (optimizer) data object*) – do-mpc (optimizer) data object. Either from unpickled results or the created modules.
- **t_ind** (*int*) – Plot predictions at this time index.
- **pltkwargs** (*, optional*) – Valid matplotlib pyplot keyword arguments (e.g.: `linewidth, color, alpha`)

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

Returns All plotted lines on all supplied axes.

Return type list

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

3.8.7.1.3 plot_results

Class method.

`do_mpc.graphics.Graphics.plot_results` (*self, data, t_ind=None, **pltkwargs*)

Plots the results stored in the passed data object for the plot configuration. Each **do-mpc** module has an individual data object. Use the `t_ind` parameter to plot only until the given time index. This is mostly used in post-processing for animations. Optionally pass an arbitrary number of valid `pyplot.plot` arguments (e.g. `linewidth, color, alpha`), which is applied to ALL lines.

Parameters

- **data** (*do-mpc data object*) – do-mpc data object. Either from unpickled results or the created modules. The data object is updated at each `make_step` call.
- **t_ind** (*int*) – Plot results up until this time index.
- **pltkwargs** (*optional*) – Valid matplotlib pyplot keyword arguments (e.g.: `linewidth, color, alpha`)

Raises

- **assertion** – `t_ind` argument must be a int
- **assertion** – `t_ind` argument must not exceed the length of the results

Returns All plotted lines on all supplied axes.

Return type list

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

3.8.7.1.4 reset_axes

Class method.

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

Clears the lines on all axes which were passed with `Graphics.add_line()`. Method is called internally, before each plot.

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

3.8.7.1.5 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 cyler 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

`default_plot`

Pass a `do_mpc.model.Model` object and create a default **do-mpc** plot.

3.8.7.2 default_plot

`do_mpc.graphics.default_plot(model, states_list=None, inputs_list=None, aux_list=None, **kwargs)`

Pass a `do_mpc.model.Model` 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.model.Model`) – Configured model that contains all information about states, inputs etc.
- **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`

d

do_mpc, 41
do_mpc.controller, 62
do_mpc.data, 91
do_mpc.estimator, 75
do_mpc.graphics, 94
do_mpc.model, 42
do_mpc.optimizer, 57
do_mpc.simulator, 52

Symbols

`__getitem__()` (*do_mpc.model.Model* method), 42

A

`add_line()` (*in module do_mpc.graphics.Graphics*), 96

`aux` (*do_mpc.model.Model* attribute), 43

B

`bounds` (*do_mpc.controller.MPC* attribute), 63

`bounds` (*do_mpc.estimator.MHE* attribute), 78

`bounds` (*do_mpc.optimizer.Optimizer* attribute), 58

D

`Data` (*class in do_mpc.data*), 91

`default_plot()` (*in module do_mpc.graphics*), 98

`do_mpc` (*module*), 41

`do_mpc.controller` (*module*), 62

`do_mpc.data` (*module*), 91

`do_mpc.estimator` (*module*), 75

`do_mpc.graphics` (*module*), 94

`do_mpc.model` (*module*), 42

`do_mpc.optimizer` (*module*), 57

`do_mpc.simulator` (*module*), 52

E

`EKF` (*class in do_mpc.estimator*), 75

`Estimator` (*class in do_mpc.estimator*), 76

`export()` (*in module do_mpc.data.Data*), 92

G

`get_p_template()` (*in module do_mpc.controller.MPC*), 66

`get_p_template()` (*in module do_mpc.estimator.MHE*), 81

`get_p_template()` (*in module do_mpc.simulator.Simulator*), 53

`get_tvp_template()` (*in module do_mpc.controller.MPC*), 67

`get_tvp_template()` (*in module do_mpc.estimator.MHE*), 82

`get_tvp_template()` (*in module do_mpc.optimizer.Optimizer*), 59

`get_tvp_template()` (*in module do_mpc.simulator.Simulator*), 53

`get_variables()` (*in module do_mpc.model.Model*), 47

`get_y_template()` (*in module do_mpc.estimator.MHE*), 83

`Graphics` (*class in do_mpc.graphics*), 94

I

`init_storage()` (*in module do_mpc.data.Data*), 92

L

`load_results()` (*in module do_mpc.data*), 93

M

`make_step()` (*in module do_mpc.controller.MPC*), 68

`make_step()` (*in module do_mpc.estimator.EKF*), 76

`make_step()` (*in module do_mpc.estimator.MHE*), 83

`make_step()` (*in module do_mpc.estimator.StateFeedback*), 91

`make_step()` (*in module do_mpc.simulator.Simulator*), 53

`MHE` (*class in do_mpc.estimator*), 77

`Model` (*class in do_mpc.model*), 42

`MPC` (*class in do_mpc.controller*), 63

O

`opt_p_num` (*do_mpc.controller.MPC* attribute), 64

`opt_p_num` (*do_mpc.estimator.MHE* attribute), 79

`opt_x_num` (*do_mpc.controller.MPC* attribute), 64

`opt_x_num` (*do_mpc.estimator.MHE* attribute), 80

`Optimizer` (*class in do_mpc.optimizer*), 58

P

`p` (*do_mpc.model.Model* attribute), 44

plot_predictions() (in
do_mpc.graphics.Graphics), 96
plot_results() (in
do_mpc.graphics.Graphics), 97

R

reset_axes() (in
do_mpc.graphics.Graphics), 98
reset_history() (in
do_mpc.controller.MPC), 68
reset_history() (in
do_mpc.estimator.EKF), 76
reset_history() (in
do_mpc.estimator.Estimator), 77
reset_history() (in
do_mpc.estimator.MHE), 84
reset_history() (in
do_mpc.estimator.StateFeedback), 91
reset_history() (in
do_mpc.optimizer.Optimizer), 60
reset_history() (in
do_mpc.simulator.Simulator), 54
reset_prop_cycle() (in
do_mpc.graphics.Graphics), 98

S

save_results() (in module do_mpc.data), 94
scaling (do_mpc.controller.MPC attribute), 65
scaling (do_mpc.estimator.MHE attribute), 80
scaling (do_mpc.optimizer.Optimizer attribute), 58
set_default_objective() (in module
do_mpc.estimator.MHE), 84
set_expression() (in
do_mpc.model.Model), 48
set_initial_guess() (in
do_mpc.controller.MPC), 68
set_initial_guess() (in
do_mpc.estimator.MHE), 84
set_initial_state() (in
do_mpc.controller.MPC), 69
set_initial_state() (in
do_mpc.estimator.EKF), 76
set_initial_state() (in
do_mpc.estimator.Estimator), 77
set_initial_state() (in
do_mpc.estimator.MHE), 85
set_initial_state() (in
do_mpc.estimator.StateFeedback), 91
set_initial_state() (in
do_mpc.optimizer.Optimizer), 60
set_initial_state() (in
do_mpc.simulator.Simulator), 54
set_meas() (in module do_mpc.model.Model), 49
set_meta() (in module do_mpc.data.Data), 92

module set_nl_cons() (in module do_mpc.controller.MPC),
69
module set_nl_cons() (in module do_mpc.estimator.MHE),
85
set_nl_cons() (in module
do_mpc.optimizer.Optimizer), 61
module set_objective() (in module
do_mpc.controller.MPC), 70
module set_objective() (in module
do_mpc.estimator.MHE), 86
module set_p_fun() (in module do_mpc.controller.MPC), 70
set_p_fun() (in module do_mpc.estimator.MHE), 87
module set_p_fun() (in module
do_mpc.simulator.Simulator), 54
module set_param() (in module do_mpc.controller.MPC), 71
set_param() (in module do_mpc.estimator.MHE), 87
module set_param() (in module
do_mpc.simulator.Simulator), 55
module set_rhs() (in module do_mpc.model.Model), 49
set_rterm() (in module do_mpc.controller.MPC), 72
module set_tvp_fun() (in module do_mpc.controller.MPC),
73
module set_tvp_fun() (in module do_mpc.estimator.MHE),
89
set_tvp_fun() (in module
do_mpc.optimizer.Optimizer), 61
set_tvp_fun() (in module
do_mpc.simulator.Simulator), 56
set_uncertainty_values() (in module
do_mpc.controller.MPC), 74
set_variable() (in module do_mpc.model.Model),
50
module set_y_fun() (in module do_mpc.estimator.MHE), 89
setup() (in module do_mpc.controller.MPC), 74
module setup() (in module do_mpc.estimator.MHE), 90
module setup() (in module do_mpc.model.Model), 51
module setup() (in module do_mpc.simulator.Simulator), 56
setup_model() (in module do_mpc.model.Model), 52
module simulate() (in module do_mpc.simulator.Simulator),
57
module Simulator (class in do_mpc.simulator), 52
solve() (in module do_mpc.controller.MPC), 75
module solve() (in module do_mpc.estimator.MHE), 90
module solve() (in module do_mpc.optimizer.Optimizer), 62
module StateFeedback (class in do_mpc.estimator), 90

T

tvp (do_mpc.model.Model attribute), 44

U

u (do_mpc.model.Model attribute), 45
update() (in module do_mpc.data.Data), 92

X

x (*do_mpc.model.Model attribute*), 45

Y

y (*do_mpc.model.Model attribute*), 46

Z

z (*do_mpc.model.Model attribute*), 46