
do-mpc

Release 4.6.0

Sergio Lucia and Felix Fiedler

May 24, 2023

INTRODUCTION

1	Example: Robust Multi-stage MPC	3
2	Example: Nonlinear MPC	5
3	Next steps	7
4	Table of contents	9
4.1	Getting started: MPC	9
4.1.1	Example system	10
4.1.2	Creating the model	10
4.1.2.1	Model variables	11
4.1.2.2	Query variables	12
4.1.2.3	Model parameters	13
4.1.2.4	Right-hand-side equation	13
4.1.3	Configuring the MPC controller	14
4.1.3.1	Optimizer parameters	14
4.1.3.2	Objective function	14
4.1.3.3	Constraints	15
4.1.3.4	Scaling	15
4.1.3.5	Uncertain Parameters	16
4.1.3.6	Setup	16
4.1.4	Configuring the Simulator	16
4.1.4.1	Simulator parameters	17
4.1.4.2	Uncertain parameters	17
4.1.4.3	Setup	18
4.1.5	Creating the control loop	18
4.1.5.1	Setting up the Graphic	19
4.1.5.2	Running the simulator	20
4.1.5.3	Running the optimizer	21
4.1.5.4	Changing the line appearance	23
4.1.5.5	Running the control loop	24
4.1.6	Data processing	25
4.1.6.1	Saving and retrieving results	25
4.1.6.2	Working with data objects	26
4.1.6.3	Animating results	27
4.2	Getting started: MHE	27
4.2.1	Creating the model	28
4.2.1.1	Model variables	28
4.2.1.2	Model measurements	28
4.2.1.3	Model parameters	29

4.2.1.4	Right-hand-side equation	29
4.2.2	Configuring the moving horizon estimator	30
4.2.2.1	MHE parameters:	30
4.2.2.2	Objective function	30
4.2.2.3	Fixed parameters	31
4.2.2.4	Bounds	31
4.2.2.5	Setup	31
4.2.3	Configuring the Simulator	31
4.2.3.1	Simulator parameters	32
4.2.3.2	Parameters	32
4.2.3.3	Setup	32
4.2.4	Creating the loop	32
4.2.4.1	Setting up the Graphic	33
4.2.4.2	Running the loop	35
4.2.5	MHE Advantages	36
4.3	Orthogonal collocation on finite elements	39
4.3.1	Lagrange polynomials for ODEs	39
4.3.2	Deriving the integration equations	41
4.3.2.1	Collocation constraints	42
4.3.2.2	Continuity constraints	42
4.3.2.3	Solving the ODE problem	42
4.3.2.4	Collocation with orthogonal polynomials	43
4.3.3	Bibliography	43
4.4	Basics of model predictive control	43
4.4.1	System model	44
4.4.2	Model predictive control problem	44
4.4.3	Robust multi-stage NMPC	46
4.4.3.1	General description	46
4.4.3.2	Robust horizon	46
4.4.3.3	Mathematical formulation	47
4.5	Basics of moving horizon estimation	48
4.5.1	System model	48
4.5.2	Moving horizon estimation problem	49
4.5.2.1	Concept	49
4.5.2.2	Mathematical formulation	49
4.6	License	50
4.7	Installation	52
4.7.1	Requirements	52
4.7.2	Option 1: PIP	52
4.7.3	Option 2: Clone from Github	52
4.7.4	HSL linear solver for IPOPT	52
4.7.4.1	Option 1: Pre-compiled binaries	53
4.7.4.1.1	Linux	53
4.7.4.2	Option 2: Compile from source	53
4.8	Credit	54
4.9	FAQ	54
4.9.1	Time-varying parameters	54
4.9.1.1	Implementation	54
4.9.1.1.1	MPC configuration	55
4.9.1.1.2	MHE configuration	56
4.9.1.1.3	Simulator configuration	56
4.9.2	Feasibility issues	57
4.9.2.1	Is the initial state feasible?	57
4.9.2.2	Which constraints are violated?	57

4.9.2.3	Use soft-constraints.	58
4.9.3	Silence IPOPT	58
4.10	do_mpc	58
4.10.1	controller	60
4.10.1.1	LQR	60
4.10.1.1.1	Methods	61
4.10.1.1.2	Attributes	65
4.10.1.2	LQRSettings	67
4.10.1.2.1	Methods	68
4.10.1.2.2	Attributes	68
4.10.1.3	MPC	68
4.10.1.3.1	Methods	70
4.10.1.3.2	Attributes	79
4.10.1.4	MPCSettings	88
4.10.1.4.1	Methods	89
4.10.1.4.2	Attributes	90
4.10.2	data	93
4.10.2.1	load_results	93
4.10.2.2	save_results	93
4.10.2.3	Data	94
4.10.2.3.1	Methods	95
4.10.2.4	MPCData	96
4.10.2.4.1	Methods	98
4.10.3	differentiator	100
4.10.3.1	DoMPCDifferentiator	100
4.10.3.1.1	Methods	101
4.10.3.1.2	Attributes	101
4.10.3.2	NLPDifferentiator	102
4.10.3.2.1	Methods	103
4.10.3.2.2	Attributes	104
4.10.3.3	helper	104
4.10.3.3.1	NLPDifferentiatorSettings	104
4.10.3.3.2	NLPDifferentiatorStatus	106
4.10.4	estimator	108
4.10.4.1	EKF	108
4.10.4.1.1	Methods	108
4.10.4.1.2	Attributes	109
4.10.4.2	Estimator	110
4.10.4.2.1	Methods	111
4.10.4.2.2	Attributes	111
4.10.4.3	MHE	113
4.10.4.3.1	Methods	114
4.10.4.3.2	Attributes	124
4.10.4.4	MHESettings	133
4.10.4.4.1	Methods	134
4.10.4.4.2	Attributes	135
4.10.4.5	StateFeedback	137
4.10.4.5.1	Methods	137
4.10.4.5.2	Attributes	138
4.10.5	graphics	139
4.10.5.1	default_plot	140
4.10.5.2	animate	140
4.10.5.3	Graphics	141
4.10.5.3.1	Methods	142

4.10.5.3.2	Attributes	145
4.10.6	model	146
4.10.6.1	dae2odeconversion	146
4.10.6.2	linearize	147
4.10.6.3	IteratedVariables	148
4.10.6.3.1	Methods	148
4.10.6.3.2	Attributes	148
4.10.6.4	LinearModel	150
4.10.6.4.1	Methods	151
4.10.6.4.2	Attributes	155
4.10.6.5	Model	161
4.10.6.5.1	Methods	162
4.10.6.5.2	Attributes	167
4.10.7	opcua	172
4.10.7.1	ClientOpts	172
4.10.7.1.1	Methods	172
4.10.7.1.2	Attributes	172
4.10.7.2	Namespace	173
4.10.7.2.1	Methods	173
4.10.7.2.2	Attributes	173
4.10.7.3	NamespaceEntry	173
4.10.7.3.1	Methods	174
4.10.7.3.2	Attributes	174
4.10.7.4	RTBase	174
4.10.7.4.1	Methods	175
4.10.7.5	RTClient	177
4.10.7.5.1	Methods	178
4.10.7.6	RTServer	179
4.10.7.6.1	Methods	179
4.10.7.7	ServerOpts	180
4.10.7.7.1	Methods	180
4.10.7.7.2	Attributes	180
4.10.8	optimizer	181
4.10.8.1	Optimizer	181
4.10.8.1.1	Methods	181
4.10.8.1.2	Attributes	186
4.10.9	sampling	190
4.10.9.1	DataHandler	190
4.10.9.1.1	Methods	191
4.10.9.1.2	Attributes	194
4.10.9.2	Sampler	194
4.10.9.2.1	Methods	195
4.10.9.2.2	Attributes	197
4.10.9.3	SamplingPlanner	197
4.10.9.3.1	Methods	197
4.10.9.3.2	Attributes	200
4.10.10	simulator	200
4.10.10.1	Simulator	201
4.10.10.1.1	Methods	201
4.10.10.1.2	Attributes	206
4.10.10.2	SimulatorSettings	208
4.10.10.2.1	Methods	208
4.10.10.2.2	Attributes	209
4.10.10.3	ContinuousSimulatorSettings	209

4.10.10.3.1	Methods	209
4.10.10.3.2	Attributes	209
4.10.11	sysid	210
4.10.11.1	ONNXConversion	210
4.10.11.1.1	Methods	212
4.10.11.2	ONNXOperations	212
4.10.11.2.1	Methods	213
4.10.12	tools	214
4.10.12.1	load_pickle	215
4.10.12.2	printProgressBar	215
4.10.12.3	save_pickle	215
4.10.12.4	IndexedProperty	216
4.10.12.4.1	Methods	216
4.10.12.5	Structure	216
4.10.12.5.1	Methods	217
4.10.12.5.2	Attributes	217
4.10.12.6	Timer	218
4.10.12.6.1	Methods	218
4.11	Release notes	218
4.11.1	v4.5.1	219
4.11.2	v4.5.0	219
4.11.2.1	Major changes	219
4.11.2.1.1	Linear control	219
4.11.2.1.2	Data-based system identification with neural networks	219
4.11.2.2	Minor changes	219
4.11.2.2.1	Compile NLP	219
4.11.2.2.2	Improved sampling tools for data generation	219
4.11.2.2.3	Simulator	220
4.11.2.3	Bug fixes	220
4.11.2.4	Backend	220
4.11.3	v4.4.0	220
4.11.3.1	Major changes	220
4.11.3.2	Minor changes	221
4.11.4	v4.3.5	221
4.11.4.1	Minor fixes	221
4.11.5	v4.3.4	221
4.11.5.1	Minor fixes	221
4.11.6	v4.3.3	221
4.11.6.1	Major changes	221
4.11.6.2	Minor changes	221
4.11.6.3	Documentation	221
4.11.6.4	Example files	221
4.11.7	v4.3.2	222
4.11.7.1	Major fixes	222
4.11.8	v4.3.1	222
4.11.9	v4.3.0	222
4.11.9.1	Major changes	222
4.11.9.1.1	do-mpc sampling tools	222
4.11.9.2	Minor changes	222
4.11.9.3	Example files	222
4.11.10	v4.2.5	222
4.11.10.1	Major changes	222
4.11.10.2	Backend	223
4.11.10.2.1	Model	223

4.11.11	v4.2.0	223
4.11.11.1	Major changes	223
4.11.11.1.1	MX support	223
4.11.11.2	Minor changes	224
4.11.11.3	Bug fixes	224
4.11.12	v4.1.1	224
4.11.12.1	Major changes	224
4.11.12.1.1	Adapted time-varying parameters for MPC object	224
4.11.12.2	Documentation	224
4.11.13	do-mpc v4.1.0	224
4.11.13.1	Major changes	224
4.11.13.1.1	DAE support	224
4.11.13.1.2	Constraints with MPC / MHE with orthogonal collocation	225
4.11.13.1.3	Terminal bounds for MPC	226
4.11.13.2	Minor changes	226
4.11.13.3	Documentation	226
4.11.13.4	Example files	226
4.11.14	do-mpc v4.0.0	226
4.11.14.1	Major changes	227
4.11.14.1.1	New properties for Simulator, Estimator and MPC	227
4.11.14.1.2	Measurement noise	227
4.11.14.1.3	Simulator with disturbances	228
4.11.14.2	Documentation	228
4.11.14.3	Minor changes	228
4.11.14.4	Example files	228
4.11.15	do-mpc v4.0.0-beta3	228
4.11.15.1	Major changes	228
4.11.15.1.1	Data	228
4.11.15.1.2	Graphics	229
4.11.15.1.3	Process noise	229
4.11.15.1.4	Symbolic variables for MHE weighting matrices	229
4.11.15.2	Example files	230
4.11.16	do-mpc v4.0.0-beta2	230
4.11.17	do-mpc v4.0.0-beta1	230
4.11.17.1	Major changes	230
4.11.17.2	Bug fixes	230
4.11.17.3	Other changes	230
4.11.17.4	Example files	230
4.11.18	do-mpc v4.0.0-beta	230
4.11.18.1	Example files	231
4.11.19	do-mpc v3.0.0	231
4.11.19.1	Main modifications	231
4.11.20	do-mpc v2.0.0	231
4.11.21	do-mpc version 1.0.0	231
4.12	Batch Bioreactor	231
4.12.1	Model	232
4.12.1.1	States and control inputs	232
4.12.1.2	ODE and parameters	232
4.12.2	Controller	233
4.12.2.1	Objective	234
4.12.2.2	Constraints	234
4.12.2.3	Uncertain values	234
4.12.3	Estimator	235
4.12.4	Simulator	235

4.12.4.1	Realizations of uncertain parameters	235
4.12.5	Closed-loop simulation	236
4.12.5.1	Prepare visualization	236
4.12.5.2	Run closed-loop	237
4.12.5.3	Results	237
4.13	Continuous stirred tank reactor (CSTR)	238
4.13.1	Model	238
4.13.1.1	States and control inputs	238
4.13.1.2	ODE and parameters	239
4.13.2	Controller	240
4.13.2.1	Objective	241
4.13.2.2	Constraints	241
4.13.2.3	Uncertain values	242
4.13.3	Estimator	242
4.13.4	Simulator	242
4.13.4.1	Realizations of uncertain parameters	243
4.13.5	Closed-loop simulation	243
4.13.6	Animating the results	244
4.14	Industrial polymerization reactor	245
4.14.1	Model	246
4.14.1.1	System description	246
4.14.1.2	Implementation	247
4.14.2	Controller	250
4.14.2.1	Objective	250
4.14.2.2	Constraints	251
4.14.2.3	Scaling	252
4.14.2.4	Uncertain values	252
4.14.3	Estimator	253
4.14.4	Simulator	253
4.14.4.1	Realizations of uncertain parameters	253
4.14.5	Closed-loop simulation	254
4.14.6	Animating the results	254
4.15	Oscillating masses	256
4.15.1	Model	256
4.15.1.1	States and control inputs	257
4.15.2	Controller	258
4.15.2.1	Objective	258
4.15.2.2	Constraints	258
4.15.3	Estimator	259
4.15.4	Simulator	259
4.15.5	Closed-loop simulation	259
4.15.6	Displaying the results	260
4.16	Double inverted pendulum	261
4.16.1	Model	261
4.16.1.1	Parameters	261
4.16.1.2	Euler-Lagrangian equations	262
4.16.1.3	Differential algebraic equation (DAE)	263
4.16.1.4	Energy equations	263
4.16.2	Controller	264
4.16.2.1	Objective	265
4.16.2.2	Constraints	265
4.16.3	Estimator	266
4.16.4	Simulator	266
4.16.5	Closed-loop simulation	266

4.16.5.1	Prepare visualization	267
4.16.5.2	Run open-loop	268
4.16.5.3	Run closed-loop	277
4.16.5.4	Results	277
4.16.6	Controller with obstacle avoidance	278
4.17	Efficient data generation and handling with do-mpc	278
4.17.1	Toy example	279
4.17.2	Sampling closed-loop trajectories	283
4.18	Continuous stirred tank reactor (CSTR) - LQR	286
4.18.1	Model	287
4.18.1.1	States and control inputs	287
4.18.1.2	ODE and parameters	288
4.18.2	Controller	289
4.18.2.1	Objective	290
4.18.3	Estimator	290
4.18.4	Simulator	291
4.18.5	Closed-loop simulation	291
4.18.6	Plotting	292
5	Indices and tables	293
	Bibliography	295
	Python Module Index	297
	Index	299

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

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

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

The **do-mpc** software is Python based and works therefore on any OS with a Python 3.x distribution. **do-mpc** has been developed by Sergio Lucia and Alexandru Tatulea at the DYN chair of the TU Dortmund lead by Sebastian Engell. The development is continued at the [Laboratory of Process Automation Systems \(PAS\)](#) of the TU Dortmund by Felix Fiedler and Sergio Lucia.

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:

EXAMPLE: NONLINEAR MPC

In the next example we showcase the capabilities of **do-mpc** to handle complex nonlinear systems. The task is to erect the classical **double inverted pendulum (DIP)** and navigate it around an obstacle.

The governing system equation is given as an implicit ODE:

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

which can be rewritten as:

$$\begin{aligned}\dot{x} &= z \\ 0 &= g(x(t), z(t), u(t))\end{aligned}$$

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

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

An animation of the obtained controller results is shown below:

The code to recreate these results can be found in our [example gallery](#).

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](#).

TABLE OF CONTENTS

4.1 Getting started: MPC

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

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

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

```
[1]: import numpy as np

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

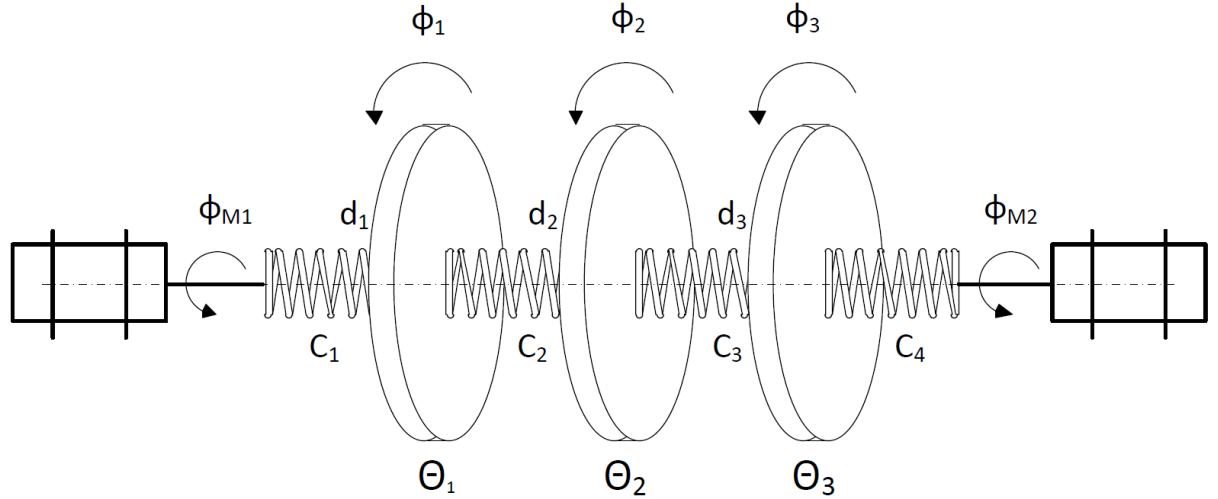
# Import do_mpc package:
import do_mpc
```

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`.

4.1.1 Example system

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



picted below:

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

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

$$\Theta_1 \ddot{\phi}_1 = -c_1 (\phi_1 - \phi_{m,1}) - c_2 (\phi_1 - \phi_2) - d_1 \dot{\phi}_1 \quad (4.1)$$

$$\Theta_2 \ddot{\phi}_2 = -c_2 (\phi_2 - \phi_1) - c_3 (\phi_2 - \phi_3) - d_2 \dot{\phi}_2 \quad (4.2)$$

$$\Theta_3 \ddot{\phi}_3 = -c_3 (\phi_3 - \phi_2) - c_4 (\phi_3 - \phi_{m,2}) - d_3 \dot{\phi}_3 \quad (4.3)$$

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

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

4.1.2 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:

$$\frac{\partial x}{\partial t} = f(x, u, z, p) \quad (4.4)$$

$$0 = g(x, u, z, p) \quad (4.5)$$

$$y = h(x, u, z, p) \quad (4.6)$$

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

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

$$x_1 = \phi_1 \quad (4.7)$$

$$x_2 = \dot{\phi}_2 \quad (4.8)$$

$$x_3 = \dot{\phi}_3 \quad (4.9)$$

$$x_4 = \ddot{\phi}_1 \quad (4.10)$$

$$x_5 = \ddot{\phi}_2 \quad (4.11)$$

$$x_6 = \ddot{\phi}_3 \quad (4.12)$$

$$(4.13)$$

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

$$\dot{x}_1 = x_4 \quad (4.14)$$

$$\dot{x}_2 = x_5 \quad (4.15)$$

$$\dot{x}_3 = x_6 \quad (4.16)$$

$$\dot{x}_4 = -\frac{c_1}{\Theta_1} (x_1 - u_1) - \frac{c_2}{\Theta_1} (x_1 - x_2) - \frac{d_1}{\Theta_1} x_4 \quad (4.17)$$

$$\dot{x}_5 = -\frac{c_2}{\Theta_2} (x_2 - x_1) - \frac{c_3}{\Theta_2} (x_2 - x_3) - \frac{d_2}{\Theta_2} x_5 \quad (4.18)$$

$$\dot{x}_6 = -\frac{c_3}{\Theta_3} (x_3 - x_2) - \frac{c_4}{\Theta_3} (x_4 - u_2) - \frac{d_3}{\Theta_3} x_6 \quad (4.19)$$

$$(4.20)$$

With this theoretical background we can start configuring the **do-mpc** `model` object.

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

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

4.1.2.1 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	<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

```
[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')
```

(continues on next page)

(continued from previous page)

```

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)

```

4.1.2.2 Query variables

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

```

[5]: model.x
[5]: <casadi.tools.structure3.ssymStruct at 0x7fa718a27d30>

```

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

```

[6]: model.x['phi_1']
[6]: SX(phi_1)

```

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

```

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

```

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

```

[8]: model.x['dphi', 0]
[8]: SX(dphi_0)

```

Note that you can use the following methods:

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

to get more information from the symbolic structures:

```

[9]: model.x.keys()
[9]: ['phi_1', 'phi_2', 'phi_3', 'dphi', 'phi_1_m', 'phi_2_m']

[10]: model.x.labels()

```

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

4.1.2.3 Model parameters

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

```
[11]: # As shown in the table above, we can use Long names or short names for the variable_
      ↪ type.
      Theta_1 = model.set_variable('parameter', 'Theta_1')
      Theta_2 = model.set_variable('parameter', 'Theta_2')
      Theta_3 = model.set_variable('parameter', 'Theta_3')

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

4.1.2.4 Right-hand-side equation

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

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

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

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

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

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

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

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

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

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

4.1.3 Configuring the MPC controller

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

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

4.1.3.1 Optimizer parameters

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

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

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

4.1.3.2 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:

$$\begin{aligned} l(x_k, u_k, z_k, p) &= \phi_1^2 + \phi_2^2 + \phi_3^2 \\ m(x_n) &= \phi_1^2 + \phi_2^2 + \phi_3^2 \end{aligned}$$

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

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

Part of the objective function is also the **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:

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

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

4.1.3.3 Constraints

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

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

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

4.1.3.4 Scaling

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

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

4.1.3.5 Uncertain Parameters

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

This can be done in the following way:

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

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

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

4.1.3.6 Setup

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

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

4.1.4 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:

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

4.1.4.1 Simulator parameters

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

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

4.1.4.2 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:

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

This object is a CasADi structure:

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

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

which can be indexed with the following keys:

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

```
[29]: ['default', 'Theta_1', 'Theta_2', 'Theta_3']
```

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

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

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

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

4.1.4.3 Setup

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

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

4.1.5 Creating the control loop

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

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

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

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

```
[34]: simulator.x0 = x0
      mpc.x0 = x0
```

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

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

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

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

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

```
[36]: DM(3.14159)
```

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

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

```
[37]: mpc.set_initial_guess()
```

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

4.1.5.1 Setting up the Graphic

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

We start by importing matplotlib:

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

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

```
[39]: mpc_graphics = do_mpc.graphics.Graphics(mpc.data)
sim_graphics = do_mpc.graphics.Graphics(simulator.data)
```

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

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

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

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

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

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

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

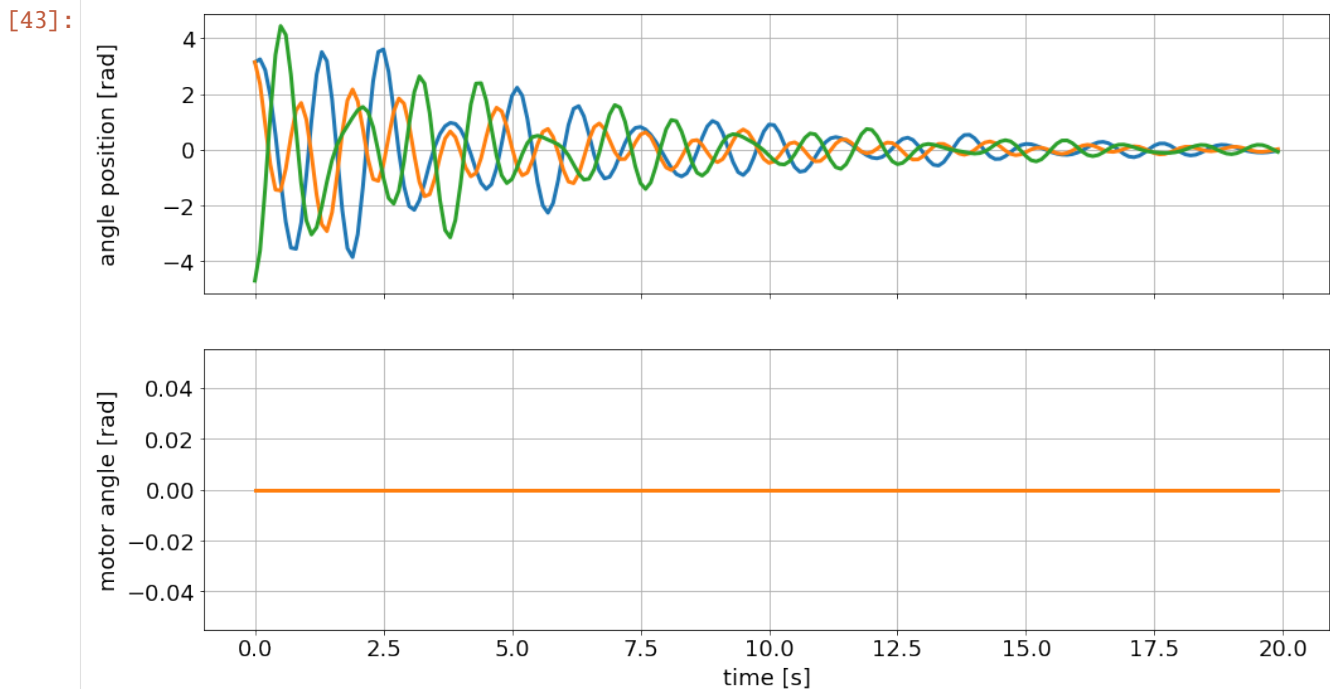
4.1.5.2 Running the simulator

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

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

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

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



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.

4.1.5.3 Running the optimizer

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

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

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

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

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

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

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

Number of Iterations...: 11

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

(continues on next page)

(continued from previous page)

```

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

```

EXIT: Optimal Solution Found.

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

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

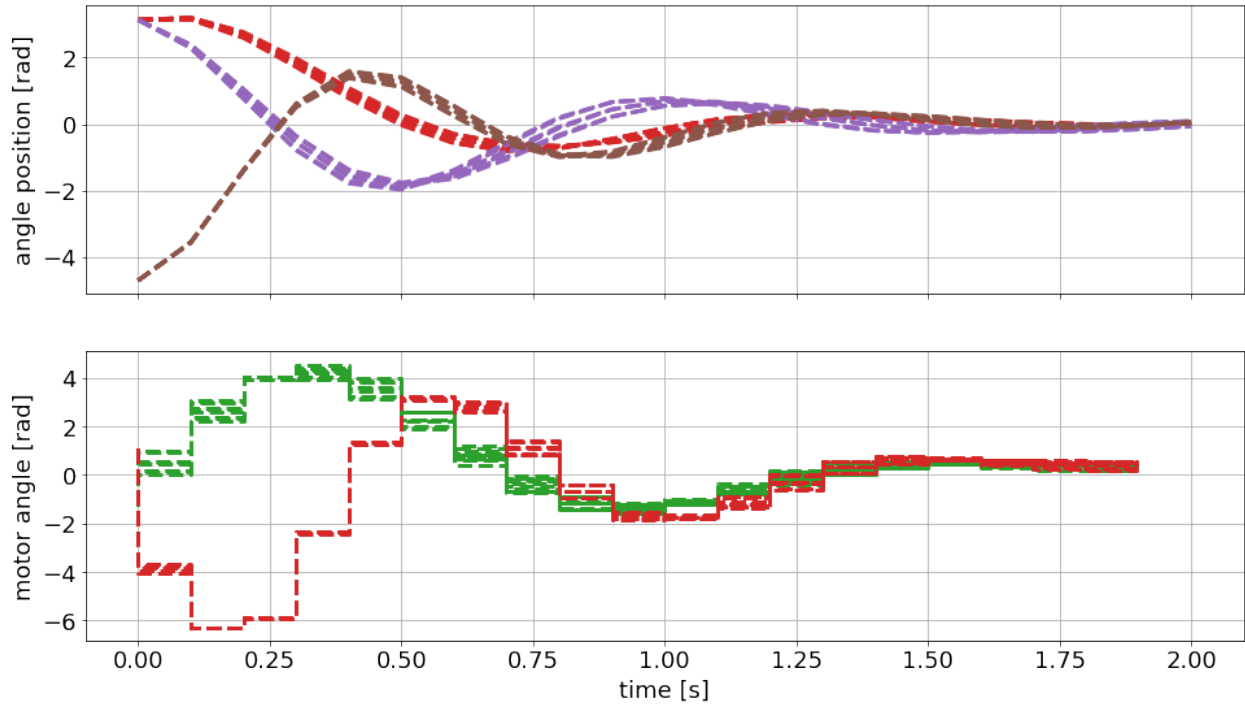
We can also visualize the predicted trajectories with the configure `graphics` instance. First we clear the existing lines from the simulator by calling:

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

And finally, we can call `plot_predictions` to obtain:

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


[46]:



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

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

4.1.5.4 Changing the line appearance

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

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

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

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

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

```
[48]: [<matplotlib.lines.Line2D at 0x7fa71c445828>,
<matplotlib.lines.Line2D at 0x7fa71c6e7898>,
<matplotlib.lines.Line2D at 0x7fa71c6e7978>,
<matplotlib.lines.Line2D at 0x7fa71c7023c8>,
<matplotlib.lines.Line2D at 0x7fa71c7022b0>,
<matplotlib.lines.Line2D at 0x7fa71c702630>,
<matplotlib.lines.Line2D at 0x7fa71c702978>,
<matplotlib.lines.Line2D at 0x7fa71c702d30>,
<matplotlib.lines.Line2D at 0x7fa71c702c88>]
```

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

```
[49]: # Change the color for the three states:
for line_i in mpc_graphics.pred_lines['_x', 'phi_1']: line_i.set_color('#1f77b4') # blue
for line_i in mpc_graphics.pred_lines['_x', 'phi_2']: line_i.set_color('#ff7f0e') # ↵
↪orange
for line_i in mpc_graphics.pred_lines['_x', 'phi_3']: line_i.set_color('#2ca02c') # green
# Change the color for the two inputs:
for line_i in mpc_graphics.pred_lines['_u', 'phi_m_1_set']: line_i.set_color('#1f77b4')
for line_i in mpc_graphics.pred_lines['_u', 'phi_m_2_set']: line_i.set_color('#ff7f0e')

# Make all predictions transparent:
for line_i in mpc_graphics.pred_lines.full: line_i.set_alpha(0.2)
```

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

```
[50]: # Get line objects (note sum of lists creates a concatenated list)
lines = sim_graphics.result_lines['_x', 'phi_1']+sim_graphics.result_lines['_x', 'phi_2']
↪'+sim_graphics.result_lines['_x', 'phi_3']

ax[0].legend(lines, '123', title='disc')

# also set legend for second subplot:
lines = sim_graphics.result_lines['_u', 'phi_m_1_set']+sim_graphics.result_lines['_u',
↪'phi_m_2_set']
ax[1].legend(lines, '12', title='motor')

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

4.1.5.5 Running the control loop

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

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

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

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

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

```
[52]: %%capture
for i in range(20):
    u0 = mpc.make_step(x0)
    x0 = simulator.make_step(u0)
```

We can now plot the previously shown prediction from time $t = 0$, as well as the closed-loop trajectory from the simulator:

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

(continues on next page)

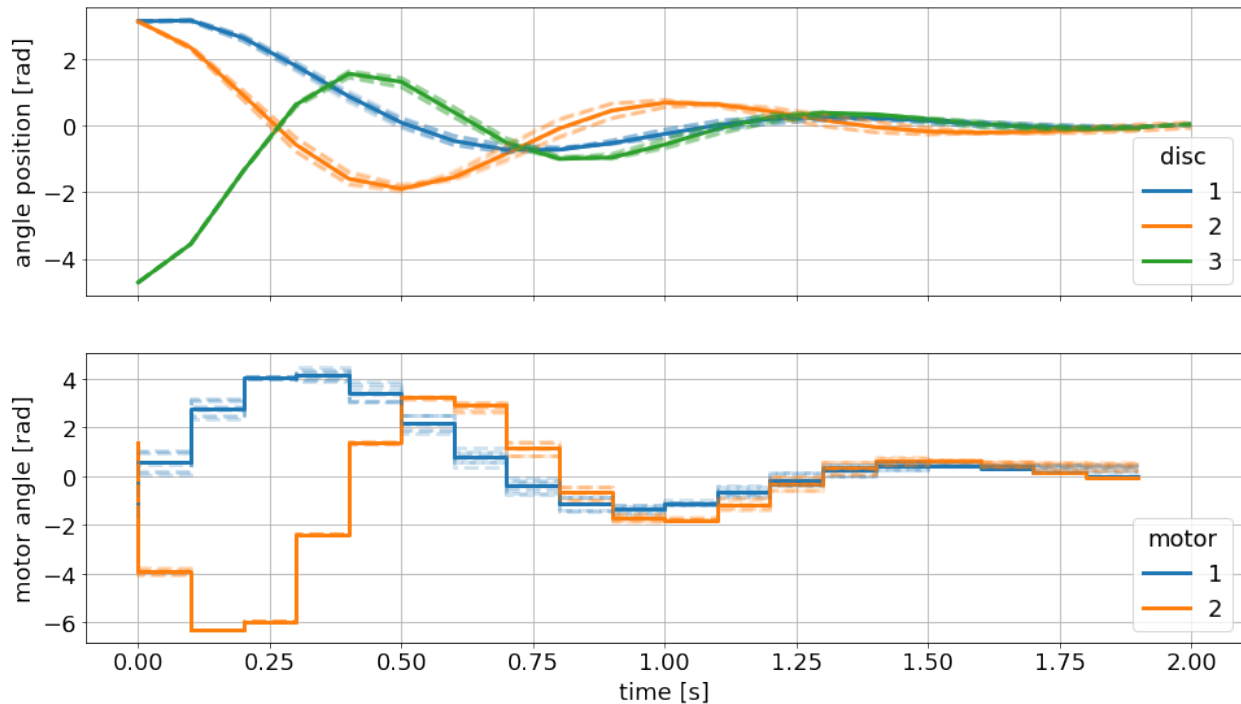
(continued from previous page)

```

sim_graphics.plot_results()
sim_graphics.reset_axes()
fig

```

[53]:



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

4.1.6 Data processing

4.1.6.1 Saving and retrieving results

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

```
[55]: from do_mpc.data import save_results, load_results
```

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

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

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

We investigate the content of the newly created folder:

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

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

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

The pickled results can be loaded manually by writing:

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

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

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

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

4.1.6.2 Working with data objects

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

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

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

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

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

For vector-valued states we can even query:

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

Of course, we could also query inputs etc.

Furthermore, we can easily retrieve the predicted trajectories with the `prediction` method. The syntax is slightly different: The first argument is a tuple that mimics the indices shown above. The second index is the time instance. With the following call we obtain the prediction of `phi_1` at time 0:

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

phi_1_pred.shape
```

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

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

4.1.6.3 Animating results

Animating MPC results, to compare prediction and closed-loop trajectories, allows for a very meaningful investigation of the obtained results.

do-mpc significantly facilitates this process while working hand in hand with Matplotlib for full customizability. Obtaining publication ready animations is as easy as writing the following short blocks of code:

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

def update(t_ind):
    sim_graphics.plot_results(t_ind)
    mpc_graphics.plot_predictions(t_ind)
    mpc_graphics.reset_axes()
```

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

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

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

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

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

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

4.2 Getting started: MHE

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

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

```
[1]: import numpy as np
from casadi import *

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

# Import do_mpc package:
import do_mpc
```

4.2.1 Creating the model

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

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

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

$$\dot{x}(t) = f(x(t), u(t), z(t), p(t), p_{tv}(t)) + w(t), \quad (4.21)$$

$$y(t) = h(x(t), u(t), z(t), p(t), p_{tv}(t)) + v(t), \quad (4.22)$$

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

4.2.1.1 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))
```

4.2.1.2 Model measurements

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

Note that some MHE implementations consider inputs separately.

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

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

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

4.2.1.3 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
```

4.2.1.4 Right-hand-side equation

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

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

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

dphi_next = vertcat(
    -c[0]/Theta_1*(phi[0]-phi_m[0])-c[1]/Theta_1*(phi[0]-phi[1])-d[0]/Theta_1*dphi[0],
    -c[1]/Theta_2*(phi[1]-phi[0])-c[2]/Theta_2*(phi[1]-phi[2])-d[1]/Theta_2*dphi[1],
    -c[2]/Theta_3*(phi[2]-phi[1])-c[3]/Theta_3*(phi[2]-phi_m[1])-d[2]/Theta_3*dphi[2],
)

model.set_rhs('dphi', dphi_next, process_noise = False)

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

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

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

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

4.2.2 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'])
```

4.2.2.1 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)
```

4.2.2.2 Objective function

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

$$\begin{aligned} \min_{\substack{\mathbf{x}_{0:N+1}, \mathbf{u}_{0:N}, p, \\ \mathbf{w}_{0:N}, \mathbf{v}_{0:N}}} \quad & \frac{1}{2} \|x_0 - \tilde{x}_0\|_{P_x}^2 + \frac{1}{2} \|p - \tilde{p}\|_{P_p}^2 + \sum_{k=0}^{N-1} \left(\frac{1}{2} \|v_k\|_{P_{v,k}}^2 + \frac{1}{2} \|w_k\|_{P_{w,k}}^2 \right), \\ \text{s.t.} \quad & \left. \begin{aligned} x_{k+1} &= f(x_k, u_k, z_k, p, p_{\text{tv},k}) + w_k, \\ y_k &= h(x_k, u_k, z_k, p, p_{\text{tv},k}) + v_k, \\ g(x_k, u_k, z_k, p_k, p_{\text{tv},k}) &\leq 0 \end{aligned} \right\} k = 0, \dots, N \end{aligned}$$

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

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

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

mhe.set_default_objective(P_x, P_v, P_p)
```


4.2.2.3 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)
```

4.2.2.4 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
```

4.2.2.5 Setup

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

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

4.2.3 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)
```

4.2.3.1 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)
```

4.2.3.2 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)
```

4.2.3.3 Setup

Finally, we call:

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

4.2.4 Creating the loop

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

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

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

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

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

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

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

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

4.2.4.1 Setting up the Graphic

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

We start by importing matplotlib:

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

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

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

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

```
[28]: %%capture
      # We just want to create the plot and not show it right now. This "inline magic"
      ↪ suppresses the output.
      fig, ax = plt.subplots(3, sharex=True, figsize=(16,9))
      fig.align_ylabels()

      # We create another figure to plot the parameters:
      fig_p, ax_p = plt.subplots(1, figsize=(16,4))
```

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

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

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

(continues on next page)

(continued from previous page)

```

ax[1].set_prop_cycle(None)

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

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

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

```

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

```

[30]: sim_graphics.result_lines
[30]: <do_mpc.tools.structure.Structure at 0x7fa7ba0e84f0>

```

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

```

[31]: # First element for state phi:
      sim_graphics.result_lines['_x', 'phi', 0]
[31]: [<matplotlib.lines.Line2D at 0x7fa7bab22340>]

```

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

```

[32]: for line_i in sim_graphics.result_lines.full:
      line_i.set_alpha(0.4)
      line_i.set_linewidth(6)

```

We furthermore use this property to create a legend:

```

[33]: ax[0].legend(sim_graphics.result_lines['_x', 'phi'], '123', title='Sim.', loc='center_
      ↪right')
      ax[1].legend(mhe_graphics.result_lines['_x', 'phi'], '123', title='MHE', loc='center_
      ↪right')
[33]: <matplotlib.legend.Legend at 0x7fa7bab34f70>

```

and another legend for the parameter plot:

```

[34]: ax_p.legend(sim_graphics.result_lines['_p', 'Theta_1']+mhe_graphics.result_lines['_p',
      ↪'Theta_1'], ['True', 'Estim.'])
[34]: <matplotlib.legend.Legend at 0x7fa7bab34d30>

```

4.2.4.2 Running the loop

We investigate the closed-loop MHE performance by alternating a simulation step ($y_0 = \text{simulator.make_step}(u_0)$) and an estimation step ($x_0 = \text{mhe.make_step}(y_0)$). Since we are lacking the controller which would close the loop ($u_0 = \text{mpc.make_step}(x_0)$), we define a random control input function:

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

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

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

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

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

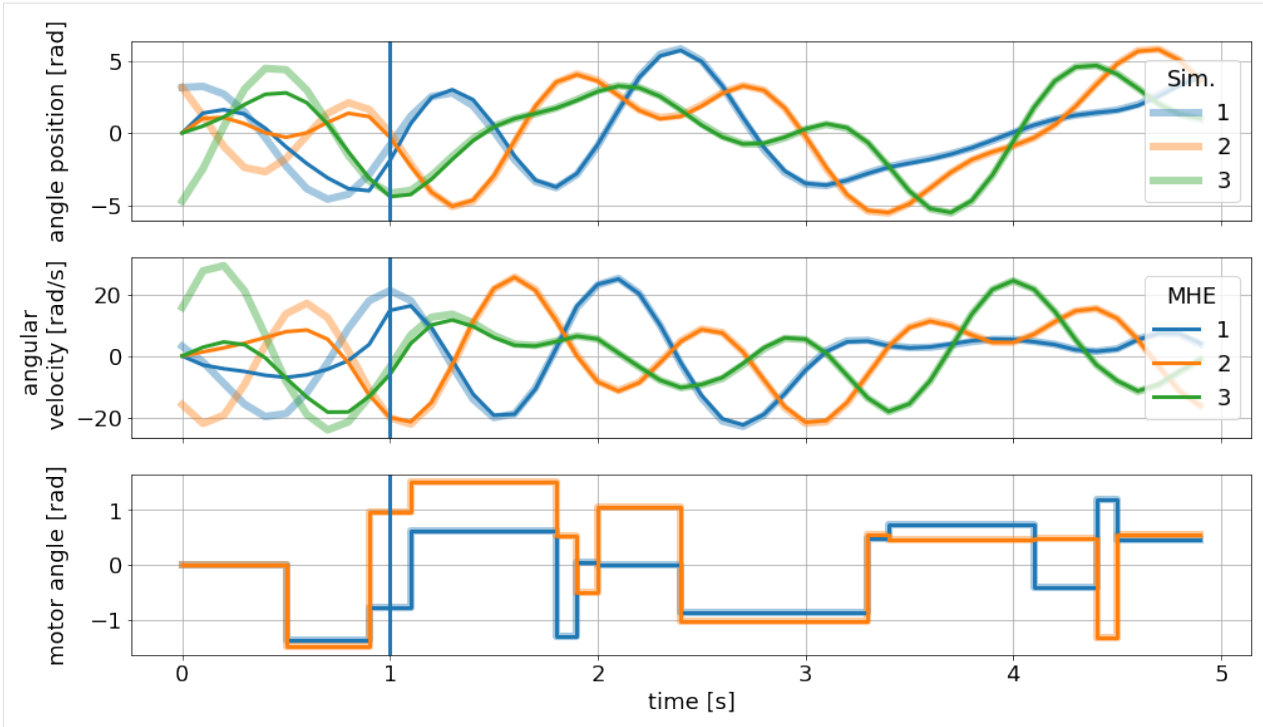
We can visualize the resulting trajectory with the pre-defined graphic:

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

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

# Show the figure:
fig
```

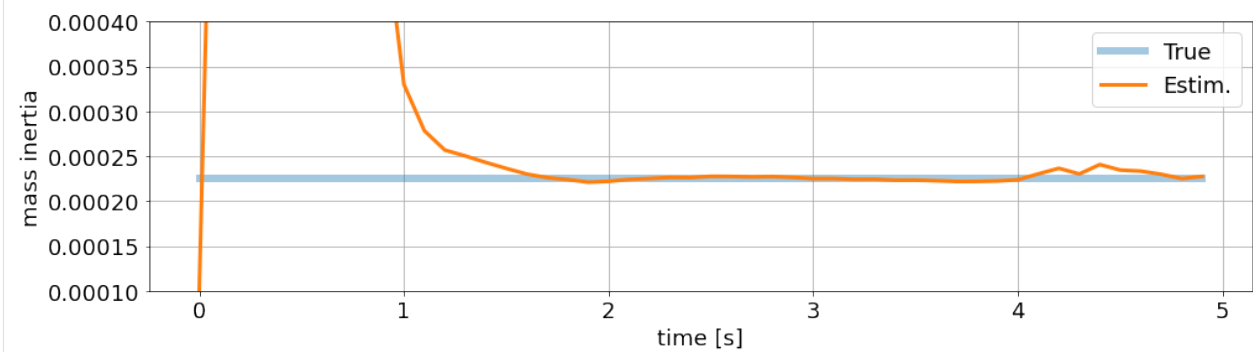
[37]:



Parameter estimation:

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

[38]:



4.2.5 MHE Advantages

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

We simply reconfigure the bounds:

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

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

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

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

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

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

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

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

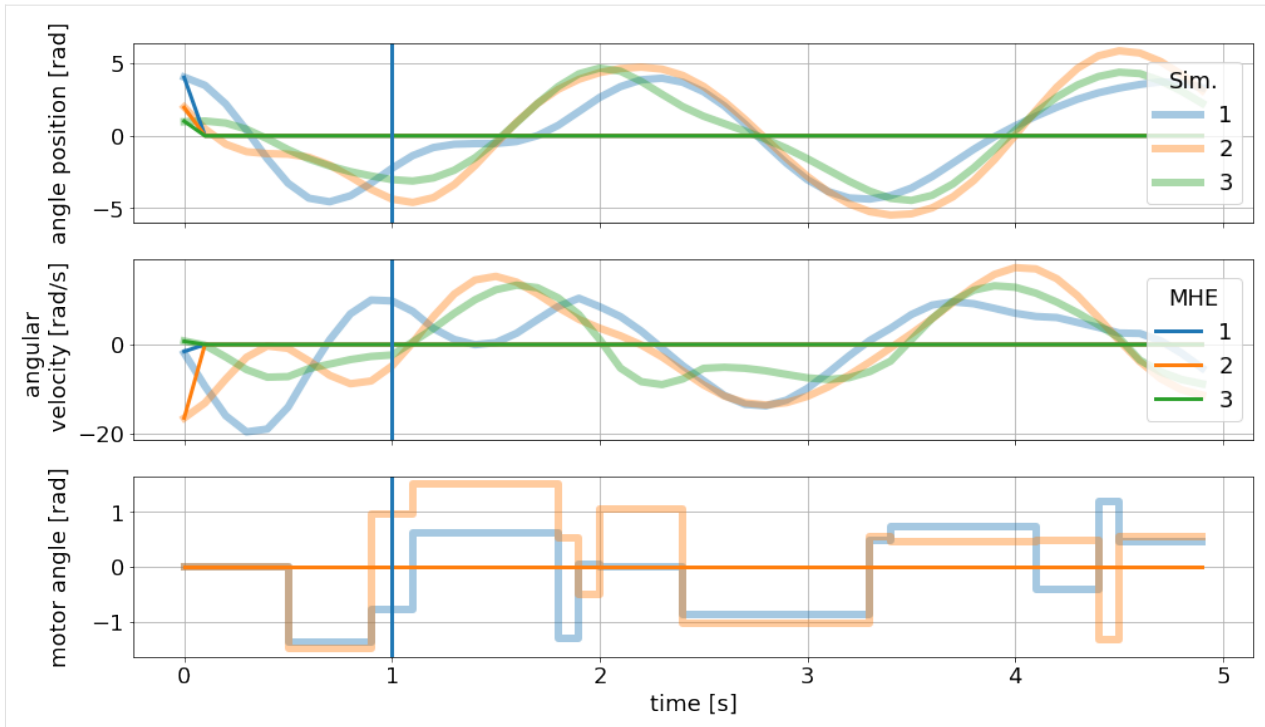
These results now look quite terrible:

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

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

      # Show the figure:
      fig
```

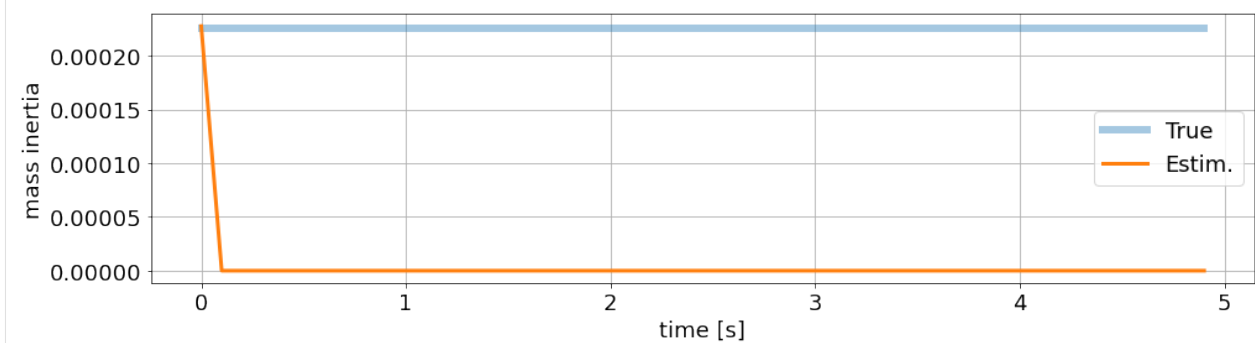
[43]:



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

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

[44]:



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

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

4.3 Orthogonal collocation on finite elements

A **dynamic system model** is at the core of all model predictive control (MPC) and moving horizon estimation (MHE) formulations. This model allows to predict and optimize the future behavior of the system (MPC) or establishes the relationship between past measurements and estimated states (MHE).

When working with **do-mpc** an essential question is whether a **discrete** or **continuous** model is supplied. The discrete time formulation:

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

$$0 = g(x_k, u_k, z_k, p_{tv,k}, p),$$

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

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

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

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

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

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

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

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

4.3.1 Lagrange polynomials for ODEs

To simplify things, we now consider the following ODE:

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

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

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

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

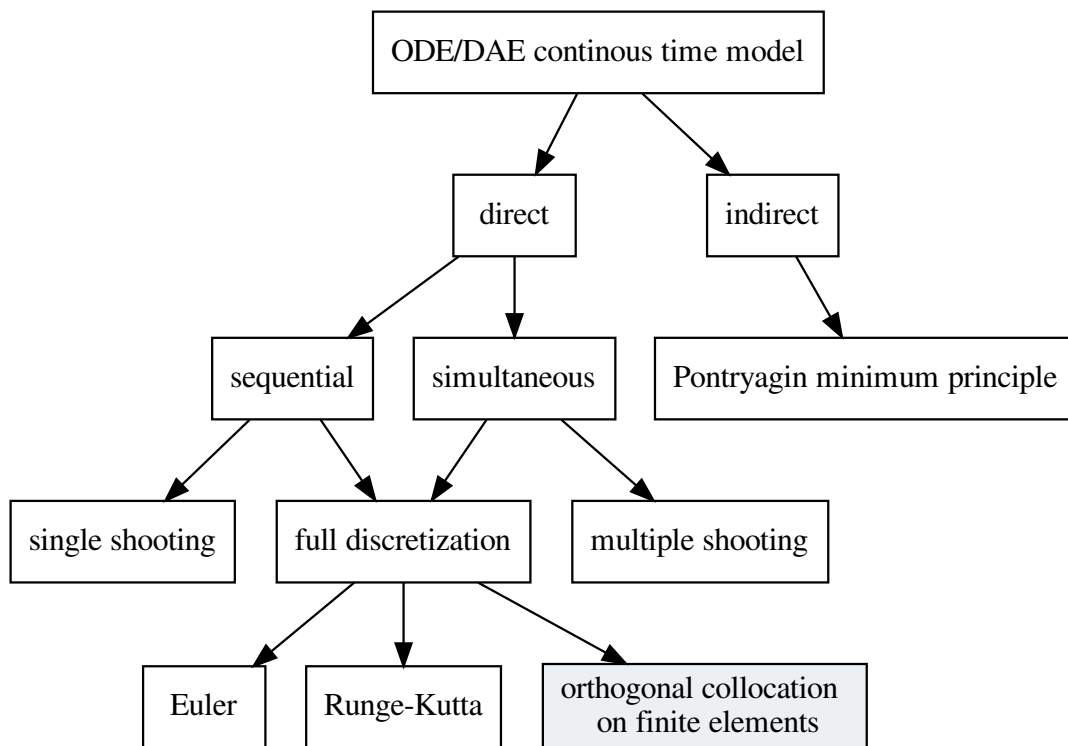


Fig. 1: Approaching an ODE/DAE continuous model for MPC or MHE.

The interpolation is based on $j = 0, \dots, K$ interpolation points $(t_j, x_{i,j})$ in the interval $[t_i, t_{i+1}]$. We are using the **Lagrange interpolation polynomial**:

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

$$\text{where: } L_j(\tau) = \prod_{\substack{k=0 \\ k \neq j}}^K \frac{(\tau - \tau_k)}{(\tau_j - \tau_k)}, \quad \tau = \frac{t - t_i}{\Delta t_i}, \quad \Delta t_i = t_{i+1} - t_i.$$

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

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

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

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

4.3.2 Deriving the integration equations

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

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

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

Because we assumed a polynomial structure of $x_i^K(t)$ the time derivative can be conveniently expressed as:

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

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

4.3.2.1 Collocation constraints

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

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

4.3.2.2 Continuity constraints

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

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

$$x_{i+1,0} = x_i^f$$

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

We thus evaluate the interpolation polynomial again and obtain:

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

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

4.3.2.3 Solving the ODE problem

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

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

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

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

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

4.3.2.4 Collocation with orthogonal polynomials

Finally we need to discuss how to choose the collocation points τ_j , $j = 0, \dots, K$. Only for fixed values of the collocation points the collocation constraints become mere algebraic equations.

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

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

$$x(t_i) = x(t_{i-1}) + \int_{t_{i-1}}^{t_i} f(x(t))dt,$$

which is solved numerically with the quadrature formula:

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

$$t_{i,j} = t_{i-1} + \tau_j \Delta t$$

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

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

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

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

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

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

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

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

4.3.3 Bibliography

4.4 Basics of model predictive control

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

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

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

- **constrained formulation:** Explicitly consider physical, safety or operational system constraints

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

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

4.4.1 System model

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

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

and for the discrete-time case by

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

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

4.4.2 Model predictive control problem

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

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

The OCP is then given by:

$$\begin{aligned}
 & \min_{\mathbf{x}_{0:N+1}, \mathbf{u}_{0:N}, \mathbf{z}_{0:N}} \\
 & m(x_{N+1}) + \sum_{k=0}^N l(x_k, z_k, u_k, p_k, p_{\text{tv},k}) \\
 & \text{subject to:} \\
 & x_0 = \hat{x}_0, \\
 & x_{k+1} = f(x_k, u_k, p_k, p_{\text{tv},k}), \\
 & \forall k = 0, \dots, N, \\
 & g(x_k, u_k, p_k, p_{\text{tv},k}) \leq 0 \\
 & \forall k = 0, \dots, N, \\
 & x_{\text{lb}} \leq x_k \leq x_{\text{ub}}, \\
 & \forall k = 0, \dots, N, \\
 & u_{\text{lb}} \leq u_k \leq u_{\text{ub}}, \\
 & \forall k = 0, \dots, N, \\
 & z_{\text{lb}} \leq z_k \leq z_{\text{ub}}, \\
 & \forall k = 0, \dots, N, \\
 & g_{\text{terminal}}(x_{N+1}) \leq 0,
 \end{aligned}$$

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

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

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

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

4.4.3 Robust multi-stage NMPC

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

4.4.3.1 General description

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

where one scenario is one path from the root node on the left side to one leaf node on the right, e.g. the state evolution for the first scenario S_4 would be $x_0 \rightarrow x_1^2 \rightarrow x_2^4 \rightarrow \dots \rightarrow x_5^4$. At every instant, the MPC problem at the root node x_0 is solved while explicitly taking into account the uncertain future evolution and the existence of future decisions, which can exploit the information gained throughout the evolution progress along the branches. Through this design, feedback information is considered in the open-loop optimization problem, which reduces the conservativeness of the multi-stage approach. Considering feedback information also means, that decisions u branching from the same node need to be identical, because they are based on the same information, e.g. $u_1^4 = u_1^5 = u_1^6$.

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

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

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

4.4.3.2 Robust horizon

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

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

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

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

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

4.4.3.3 Mathematical formulation

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

$$\begin{aligned}
 & \min_{x_k^j, u_k^j, z_k^j \forall (j,k) \in I} \\
 & \sum_{j=1}^{N_s} \omega_j J_j(\mathbf{x}_{0:N+1}^j, \mathbf{u}_{0:N}^j, \mathbf{z}_{0:N}^j) \\
 & \text{subject to:} \\
 & x_0 = \hat{x}_0 \\
 & x_{k+1}^j = f(x_k^{p(j)}, u_k^j, z_k^{p(j)}, p_k^{r(j)}, p_{\text{tv},k}) \\
 & \forall (j, k) \in I \\
 & u_k^i = u_k^j \text{ if } x_k^{p(i)} = x_k^{p(j)}, \\
 & \forall (i, k), (j, k) \in I \\
 & g(x_k^{p(j)}, u_k^j, z_k^{p(j)}, p_k^{r(j)}, p_{\text{tv},k}) \leq 0 \\
 & \forall (j, k) \in I \\
 & x_{\text{lb}} \leq x_k^j \leq x_{\text{ub}} \\
 & \forall (j, k) \in I \\
 & u_{\text{lb}} \leq u_k^j \leq u_{\text{ub}} \\
 & \forall (j, k) \in I \\
 & z_{\text{lb}} \leq z_k^j \leq z_{\text{ub}} \\
 & \forall (j, k) \in I \\
 & g_{\text{terminal}}(x_N^j, z_N^j) \leq 0 \\
 & \forall (j, N) \in I,
 \end{aligned}$$

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

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

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

For linear systems if $p_{\min} \leq p \leq p_{\max}$, considering the extreme values of the uncertainties in the scenario tree guarantees constraint satisfaction, even if the uncertainties are continuous and time-varying. This design of the scenario tree for nonlinear systems does not guarantee constraint satisfaction for all $p \in [p_{\min}, p_{\max}]$. However, also for nonlinear systems

the worst-case scenarios are often at the boundaries of the uncertainty intervals $[p_{\min}, p_{\max}]$. In practice, considering only the extreme values for nonlinear systems provides good results.

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

4.5 Basics of moving horizon estimation

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

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

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

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

4.5.1 System model

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

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

and for the discrete-time case by

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

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

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

4.5.2 Moving horizon estimation problem

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

4.5.2.1 Concept

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

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

This concept is visualized in the figure below.

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

4.5.2.2 Mathematical formulation

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

$$\begin{aligned} \min_{\mathbf{x}_{0:N+1}, \mathbf{u}_{0:N}, p, \mathbf{w}_{0:N}, \mathbf{v}_{0:N}} \quad & \frac{1}{2} \|x_0 - \tilde{x}_0\|_{P_x}^2 + \frac{1}{2} \|p - \tilde{p}\|_{P_p}^2 + \sum_{k=0}^{N-1} \left(\frac{1}{2} \|v_k\|_{P_{v,k}}^2 + \frac{1}{2} \|w_k\|_{P_{w,k}}^2 \right), \\ \text{s.t.} \quad & \left. \begin{aligned} x_{k+1} &= f(x_k, u_k, z_k, p, p_{\text{tv},k}) + w_k, \\ y_k &= h(x_k, u_k, z_k, p, p_{\text{tv},k}) + v_k, \\ g(x_k, u_k, z_k, p, p_{\text{tv},k}) &\leq 0 \end{aligned} \right\} k = 0, \dots, N \end{aligned}$$

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

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

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

4.6 License

GNU LESSER GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

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

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

0. Additional Definitions.

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

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

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

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

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

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

1. Exception to Section 3 of the GNU GPL.

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

2. Conveying Modified Versions.

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

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

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.

4.7 Installation

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

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

4.7.1 Requirements

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

- numpy
- CasADi
- matplotlib

4.7.2 Option 1: PIP

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

1. Install **do-mpc**:

```
pip install do-mpc
```

Tested on Windows and Linux (Ubuntu 19.04).

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

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

2. Get example documents:

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

4.7.3 Option 2: Clone from Github

More experienced users are advised to clone or fork the most recent version of **do-mpc** from [GitHub](#):

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

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

4.7.4 HSL linear solver for IPOPT

The standard configuration of **do-mpc** is based on [IPOPT](#) to solve the nonlinear constrained optimization problems that arise with the MPC and MHE formulation. The computational bottleneck of this method is repeatedly solving a large-scale linear systems for which IPOPT is offering 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.

4.7.4.1 Option 1: Pre-compiled binaries

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

4.7.4.1.1 Linux

(Tested on Ubuntu 19.10)

1. Obtain the [HSL](#) shared library. Choose the personal licence.
2. Unpack the archive and copy its content to a destination of your choice. (e.g. `/home/username/Documents/coinhsl/`)
3. Rename `libcoinhsl.so` to `libhsl.so`. CasADi is searching for the shared libraries under a depreciated name.
4. Locate your `.bashrc` file on your home directory (e.g. `/home/username/.bashrc`)
5. Add the previously created directory to your `LD_LIBRARY_PATH`, by adding the following line to your `.bashrc`

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

6. Install `libgfortran` with Anaconda:

```
conda install -c anaconda libgfortran
```

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

```
Tried to obtain MA27 from shared library "libhsl.so", but the following error 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.

4.7.4.2 Option 2: Compile from source

Please see the comprehensive guide on the [CasADi Github Wiki](#).

4.8 Credit

The developers of **do-mpc** own credit to [CasADi](#) and [Ipopt](#) which run at the core of our MPC and MHE implementation.

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

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

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

4.9 FAQ

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

4.9.1 Time-varying parameters

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

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

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

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

4.9.1.1 Implementation

Time-varying parameters are always introduced in the **do-mpc** `do_mpc.model.Model` with the `do_mpc.model.Model.set_variable` method. For example:

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

# Introduce state temperature:
temperature = model.set_variable(var_type='_x', var_name='temperature')
# Introduce tvp: Set-point for the temperature
temperature_set_point = model.set_variable(var_type='_tvp', var_name='temperature_set_
↪point')
# Introduce tvp: External temperature (disturbance)
temperature_external = model.set_variable(var_type='_tvp', var_name='temperature_external
↪')
```

(continues on next page)

(continued from previous page)

...

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

```
model.set_rhs('temperature', alpha*(temperature_external-temperature))
```

4.9.1.1.1 MPC configuration

Furthermore, we want to use the introduced set-point in a quadratic MPC cost function. To do this, we initiate an `do_mpc.controller.MPC` object with the configured model:

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

mpc.set_param(n_horizon = 20, t_step = 60)
```

And then use the attributes `do_mpc.model.Model.x` and `do_mpc.model.Model.tvp` to formulate a quadratic tracking cost.

```
lterm = (model.x['temperature']-model.tvp['temperature_set_point'])**2
mterm = lterm
mpc.set_objective(lterm=lterm, mterm=mterm)
```

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

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

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

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

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

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

The **tvf-function** returns numerical values for the currently valid sequences and passes them to the optimizer. Because the **tvf-function** is user-defined, the approach allows for the greatest flexibility.

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

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

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

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

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

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

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

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

The configuration of the MPC controller is thus completed.

4.9.1.1.2 MHE configuration

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

4.9.1.1.3 Simulator configuration

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

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

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

```
# Get simulator instance. The model contains _tvp.  
simulator = do_mpc.simulator.Simulator(model)  
# Set some required parameters  
simulator.set_param(t_step = 60)  
  
# Get the template  
tv_template = simulator.get_tvp_template()  
  
# Define the function (indexing is much simpler ...)  
def tvp_fun(t_now):  
    tvp_template['temperature_external'] = ...
```

(continues on next page)

(continued from previous page)

```

    return tvp_template

# Set the tvp_fun:
simulator.set_tvp_fun(tvp_fun)

```

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

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

4.9.2 Feasibility issues

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

4.9.2.1 Is the initial state feasible?

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

4.9.2.2 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.

4.9.2.3 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](#)

4.9.3 Silence IPOPT

IPOPT is the default solver for the [do_mpc.controller.MPC](#) controller and [do_mpc.estimator.MHE](#) estimator. While we generally **recommend to have a look at the solver output**, to check for feasibility issues, it may be useful to silence IPOPT in some cases.

This can be achieved conveniently over the [do_mpc.controller.MPCSettings](#) and [do_mpc.estimator.MHESettings](#) which are stored as the attribute `settings`, e.g.

```
# for the MPC

mpc.settings.supress_ipopt_output()

# or for the MHE

mhe.settings.supress_ipopt_output()
```

4.10 do_mpc

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

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

do_mpc.controller	Controller for dynamic systems.
do_mpc.data	Storage and handling of data.
do_mpc.differentiator	Tools for NLP differentiation.
do_mpc.estimator	State estimation for dynamic systems.
do_mpc.graphics	Visualization tools for do-mpc.
do_mpc.model	Dynamic modelling with do-mpc.
do_mpc.opcua	A OPC UA wrapper for do-mpc.
do_mpc.optimizer	Shared tools for optimization-based estimation (MHE) and control (MPC).
do_mpc.sampling	Sampling tools for data generation.
do_mpc.simulator	Simulate continous-time ODE/DAE or discrete-time dynamic systems.
do_mpc.sysid	Tools for machine learning and system identification.
do_mpc.tools	Various auxiliary tools for do-mpc.

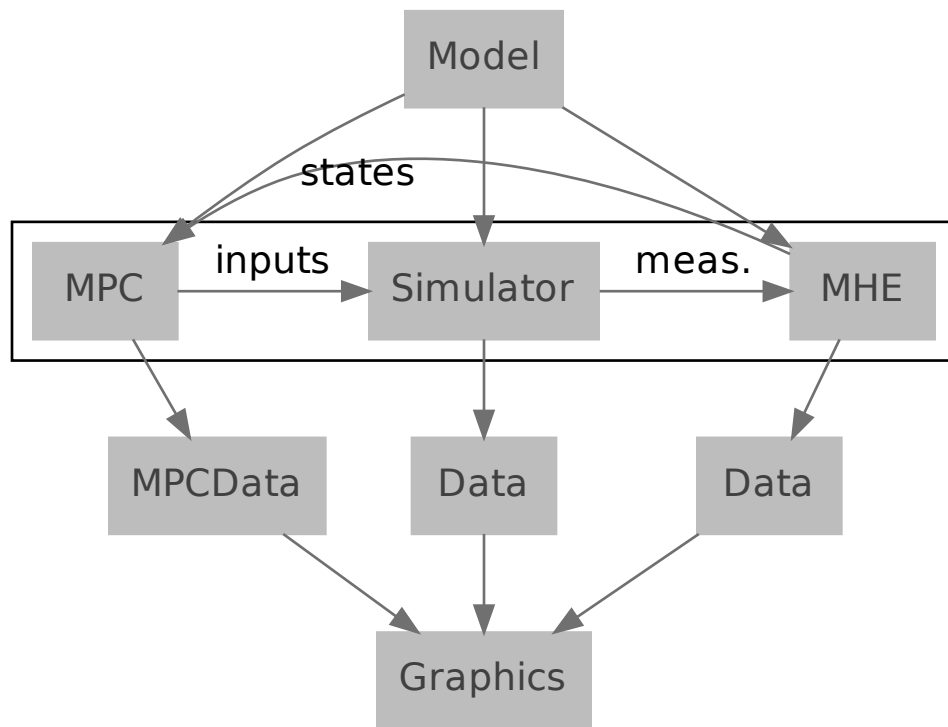


Fig. 3: **do-mpc** control loop and interconnection of classes.

4.10.1 controller

Controller for dynamic systems.

Classes

<i>LQR</i>	Linear Quadratic Regulator.
<i>LQRSettings</i>	Settings for <i>do_mpc.controller.LQR</i> .
<i>MPC</i>	Model predictive controller.
<i>MPCSettings</i>	Settings for <i>do_mpc.controller.MPC</i> .

4.10.1.1 LQR

class `LQR(model)`

Bases: [*IteratedVariables*](#)

Linear Quadratic Regulator.

New in version >v4.5.1: New interface to settings. The class has an attribute `settings` which is an instance of [*LQRSettings*](#) (please see this documentation for a list of available settings). Settings are now chosen as:

```
lqr.settings.t_step = .5
```

Previously, settings were passed to [*set_param\(\)*](#). This method is still available and wraps the new interface. The new method has important advantages:

1. The `lqr.settings` attribute can be printed to see the current configuration.
2. Context help is available in most IDEs (e.g. VS Code) to see the available settings, the type and a description.

Use this class to configure and run the LQR controller according to the previously configured [*do_mpc.model.LinearModel*](#) instance.

Two types of LQR can be designed:

1. **Finite Horizon** LQR by choosing, e.g. `n_horizon = 20`.
2. **Infinite Horizon** LQR by choosing `n_horizon = None`.

The value for `n_horizon` is set using [*set_param\(\)*](#).

Configuration and setup:

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

1. Configure the LQR controller with [*LQRSettings*](#) class. The LQR instance has the attribute `settings` which is an instance of [*LQRSettings*](#).
2. Set the objective of the control problem with [*set_objective\(\)*](#)
3. To finalize the class configuration call [*setup\(\)*](#).

The [*LQR*](#) can be used in **two different modes**:

1. **Standard** mode:
 - Set set-point with [*set_setpoint\(\)*](#) (default is 0).
 - Set Q and R values with [*set_objective\(\)*](#).

2. Input Rate Penalization mode:

- Setpoint can also be set using `set_setpoint()` (default is 0).
- Reformulate objective with `set_rterm()` to penalize the input rate by setting the value delR.
- Set Q and R values with `set_objective()`.

Note: The function `set_rterm()` mode is not recommended to use if the model is converted from an DAE to an ODE system. Because the converted model is already in the rated input formulation.

Note: During runtime call `make_step()` with the current state x to obtain the optimal control input u . During runtime call `set_setpoint()` with the set points of input u_{ss} and states x_{ss} in order to update the respective set points.

Parameters

model (`LinearModel`) – Linear model

4.10.1.1.1 Methods

discrete_gain

discrete_gain(*self*, A, B)

Computes discrete gain.

This method computes either the finite horizon discrete gain or infinite horizon discrete gain. The gain is computed by the solution of discrete-time algebraic Ricatti equation.

For finite horizon LQR, the problem formulation is as follows:

$$\begin{aligned}\pi(N) &= P_f \\ K(k) &= -(B'\pi(k+1)B)^{-1}B'\pi(k+1)A \\ \pi(k) &= Q + A'\pi(k+1)A - A'\pi(k+1)B(B'\pi(k+1)B + R)^{-1}B'\pi(k+1)A\end{aligned}$$

For infinite horizon LQR, the problem formulation is as follows:

$$\begin{aligned}K &= -(B'PB + P)^{-1}B'PA \\ P &= Q + A'PA - A'PB(R + B'PB)^{-1}B'PA\end{aligned}$$

For example:

```
K = lqr.discrete_gain(A,B)
```

Parameters

- **A** (ndarray) – State matrix - constant matrix with no variables
- **B** (ndarray) – Input matrix - constant matrix with no variables

Returns

ndarray – Gain matrix K

make_step**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.

Parameters

x0 (ndarray) – Current state of the system.

Returns

ndarray – u0 - current input of the system

reset_history**reset_history**(self)

Reset the history of the LQR.

Return type

None

set_objective**set_objective**(self, Q, R, P=None)

Sets the cost matrix for the Optimal Control Problem.

This method sets the inputs, states and algebraic states cost matrices for the given problem.

Since the controller can be operated in two modes. The objective function differs from each other and is as follows

Finite Horizon:

For **set-point tracking** mode:

$$J = \frac{1}{2} \sum_{k=0}^{N-1} (x_k - x_{ss})^T Q (x_k - x_{ss}) + (u_k - u_{ss})^T R (u_k - u_{ss}) \\ + (x_N - x_{ss})^T P (x_N - x_{ss})$$

For **Input Rate Penalization** mode:

$$J = \frac{1}{2} \sum_{k=0}^{N-1} (\tilde{x}_k - \tilde{x}_{ss})^T \tilde{Q} (\tilde{x}_k - \tilde{x}_{ss}) + \Delta u_k^T \Delta R \Delta u_k + (\tilde{x}_N - \tilde{x}_{ss})^T P (\tilde{x}_N - \tilde{x}_{ss})$$

Infinite Horizon:

For **set-point tracking** mode:

$$J = \frac{1}{2} \sum_{k=0}^{\inf} (x_k - x_{ss})^T Q (x_k - x_{ss}) + (u_k - u_{ss})^T R (u_k - u_{ss})$$

For **Input Rate Penalization** mode:

$$J = \frac{1}{2} \sum_{k=0}^{\inf} (\tilde{x}_k - \tilde{x}_{ss})^T \tilde{Q} (\tilde{x}_k - \tilde{x}_{ss}) + \Delta u_k^T \Delta R \Delta u_k$$

where $\tilde{x} = [x, u]^T$.

Note: For the problem to be solved in `inputRatePenalization` mode, Q, R and `delR` should be set. `delR` is set using `set_rterm()`. P term is set according to the need of the problem.

For example:

```
# Values used are to show how to use this function.
# For ODE models
lqr.set_objective(Q = np.identity(2), R = np.identity(2), P = np.identity(2))
```

Warning: Q, R, P is chosen as matrix of zeros since it is not passed explicitly. If P is not given explicitly, then Q is chosen as P for calculating finite discrete gain

Raises

- **exception** – Q matrix must be of type class `numpy.ndarray`
- **exception** – R matrix must be of type class `numpy.ndarray`
- **exception** – P matrix must be of type class `numpy.ndarray`

Parameters

- **Q** (`ndarray`) – State cost matrix
- **R** (`ndarray`) – Input cost matrix
- **P** (`Optional[ndarray]`) – Terminal cost matrix (optional)

Return type

None

set_param

set_param(*self*, ***kwargs*)

Set the parameters of the LQR class. Parameters must be passed as pairs of valid keywords and respective argument.

Two different kinds of LQR can be designed. In order to design a finite horizon LQR, `n_horizon` and to design an infinite horizon LQR, `n_horizon` should be set to `None` (default value). :rtype: None

Deprecated since version >v4.5.1: This function will be deprecated in the future

Warning: This method will be depreciated in a future version. Please set parameters via `do_mpc.controller.LQRSettings`.

Note: A comprehensive list of all available parameters can be found in `do_mpc.controller.LQRSettings`.

For example:

```
lqr.settings.n_horizon = 20
```

The old interface, as shown in the example below, can still be accessed until further notice.

For example:

```
lqr.set_param(n_horizon = 20)
```

Note: The only required parameters are `n_horizon`. All other parameters are optional. `set_param()` can be called multiple times. Previously passed arguments are overwritten by successive calls.

set_rterm

set_rterm(*self*, *delR*)

Modifies the model such that rated input acts as the input.

Warning: Calling `set_rterm()` modifies the objective function as well as the state and input matrix.

Warning: It is not advisable to execute LQR in the `inputRatePenalization` mode if the model is converted from DAE to ODE system. Because the converted model itself is in `inputRatePenalization` mode.

The input rate penalization formulation is given as:

$$x(k+1) = \tilde{A}x(k) + \tilde{B}\Delta u(k)$$
$$\text{where } \tilde{A} = \begin{bmatrix} A & B \\ 0 & I \end{bmatrix}, \quad \tilde{B} = \begin{bmatrix} B \\ I \end{bmatrix}$$

We introduce new states of this system as $\tilde{x} = [x, u]$ where x and u are the original states and input of the system. After reformulating the system with `set_rterm()`, the discrete gain is calculated using `discrete_gain()`.

As the system state matrix and input matrix are altered, cost matrices are also modified accordingly:

$$\tilde{Q} = \begin{bmatrix} Q & 0 \\ 0 & R \end{bmatrix}, \quad \tilde{R} = \Delta R$$

Parameters

delR (ndarray) – Rated input cost matrix - constant matrix with no variables

Return type

None

set_setpoint

set_setpoint(*self*, *xss=None*, *uss=None*)

Sets setpoints for states and inputs.

This method can be used to set setpoints for either states or inputs or for both (states and inputs) at each time step. It can be called inside simulation loop to change the set point dynamically.

Note: If setpoints is not specifically mentioned it will be set to zero (default).

For example:

```
# For ODE models
lqr.set_setpoint(xss = np.array([[10],[15]]),uss = np.array([[2],[3]]))
```

Parameters

- **xss** (Optional[ndarray]) – set point for states of the system(optional)
- **uss** (Optional[ndarray]) – set point for inputs of the system(optional)

Return type

None

setup

setup(*self*)

Prepares LQR for execution. This method initializes and ensures that all the parameters that are necessary to design the lqr are available.

Raises

exception – mode must be standard, inputRatePenalization, None. you have {string value}

Return type

None

4.10.1.1.2 Attributes

t0

LQR.**t0**

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

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

u0

LQR.u0

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

Example:

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

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

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

Useful CasADi symbolic structure methods:

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

x0

LQR.x0

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

Example:

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

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

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

Useful CasADi symbolic structure methods:

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

z0**LQR.z0**

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

Example:

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

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

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

Useful CasADi symbolic structure methods:

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

4.10.1.2 LQRSettings

class LQRSettings(*t_step=None, n_horizon=None*)

Bases: ControllerSettings

Settings for *do_mpc.controller.LQR*.

The *do_mpc.controller.LQR* automatically creates an instance of type *LQRSettings* and adds it to its class attributes.

Example to change settings:

```
lqr.settings.n_horizon = 20
```

Note: Settings cannot be updated after calling *do_mpc.controller.LQR.setup()*.

Parameters

- **t_step** (Optional[float]) –
- **n_horizon** (Optional[int]) –

4.10.1.2.1 Methods

`check_for_mandatory_settings`

`check_for_mandatory_settings(self)`

Method to assert the necessary settings required to design `do_mpc.controller`

4.10.1.2.2 Attributes

`n_horizon`

`LQRSettings.n_horizon: int = None`

Prediction horizon of the optimal control problem. Defaults to `None`, which represents an infinite horizon.

`t_step`

`LQRSettings.t_step: float = None`

Timestep of the controller

4.10.1.3 MPC

`class MPC(model, settings=None)`

Bases: `Optimizer`, `IteratedVariables`

Model predictive controller.

New in version >v4.5.1: New interface to settings. The class has an attribute `settings` which is an instance of `MPCSettings` (please see this documentation for a list of available settings). Settings are now chosen as:

```
mpc.settings.n_horizon = 20
```

Previously, settings were passed to `set_param()`. This method is still available and wraps the new interface. The new method has important advantages:

1. The `mpc.settings` attribute can be printed to see the current configuration.
2. Context help is available in most IDEs (e.g. VS Code) to see the available settings, the type and a description.
3. The `MPCSettings` class has convenient methods, such as `MPCSettings.supress_ipopt_output()` to silence the solver.

For general information on model predictive control, please read our [background article](#).

The MPC controller extends the `do_mpc.optimizer.Optimizer` base class (which is also used for the `do_mpc.estimator.MHE` estimator).

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

Configuration and setup:

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

1. Configure the MPC controller with `MPCSettings`. The MPC instance has the attribute `settings` which is an instance of `MPCSettings`.
2. Set the objective of the control problem with `set_objective()` and `set_rterm()`
3. Set upper and lower bounds with `bounds` (optional).
4. Set further (non-linear) constraints with `set_nl_cons()` (optional).
5. Use the low-level API (`get_p_template()` and `set_p_fun()`) or high level API (`set_uncertainty_values()`) to create scenarios for robust MPC (optional).
6. Use `get_tvp_template()` and `set_tvp_fun()` to create a method to obtain new time-varying parameters at each iteration.
7. To finalize the class configuration there are two routes. The default approach is to call `setup()`. For deep customization use the combination of `prepare_nlp()` and `create_nlp()`. See graph below for an illustration of the process.

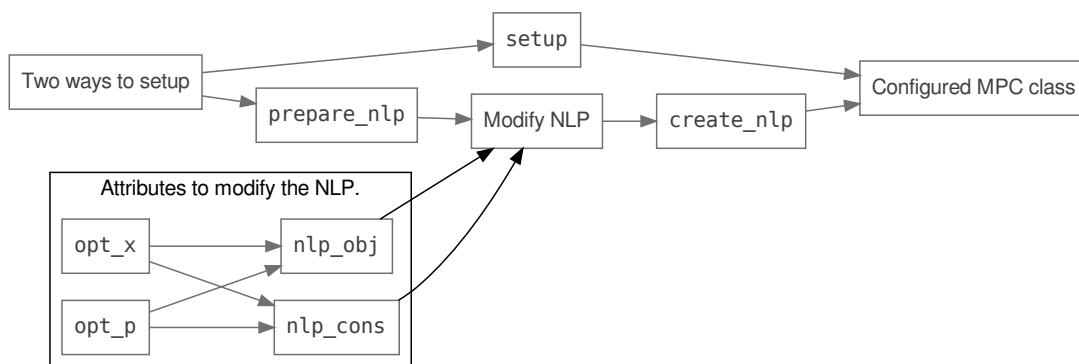


Fig. 4: Route to setting up the MPC class.

Parameters

- **model** (Union[`Model`, `LinearModel`]) – Model
- **settings** (Optional[`MPCSettings`]) – Settings for the MPC controller. See `MPCSettings` for details.

Warning: Before running the controller, make sure to supply a valid initial guess for all optimized variables (states, algebraic states and inputs). Simply set the initial values of `x0`, `z0` and `u0` and then call `set_initial_guess()`.

To take full control over the initial guess, modify the values of `opt_x_num`.

During runtime call `make_step()` with the current state `x` to obtain the optimal control input `u`.

4.10.1.3.1 Methods

compile_nlp

compile_nlp(*self*, *overwrite=False*, *cname='nlp.c'*, *libname='nlp.so'*, *compiler_command=None*)

Compile the NLP. This may accelerate the optimization. As compilation is time consuming, the default option is to NOT overwrite (*overwrite=False*) an existing compilation. If an existing compilation with the name *libname* is found, it is used. **This can be dangerous, if the NLP has changed** (user tweaked the cost function, the model etc.).

Warning: This feature is experimental and currently only supported on Linux and MacOS.

What happens here?

1. The NLP is written to a C-file (*cname*)
2. The C-File (*cname*) is compiled. The custom compiler uses:

```
gcc -fPIC -shared -O1 {cname} -o {libname}
```

3. The compiled library is linked to the NLP. This overwrites the original NLP. Options from the previous NLP (e.g. linear solver) are kept.

```
self.S = nlpsol('solver_compiled', 'ipopt', f'{libname}', self.nlpsol_opts)
```

Parameters

- **overwrite** (bool) – If True, the existing compiled NLP will be overwritten.
- **cname** (str) – Name of the C file that will be exported.
- **libname** (str) – Name of the shared library that will be created after compilation.
- **compiler_command** (Optional[str]) – Command to use for compiling. If None, the default compiler command will be used. Please make sure to use matching strings for *libname* when supplying your custom compiler command.

Return type

None

create_nlp

create_nlp(*self*)

Create the optimization problem. Typically, this method is called internally from [setup\(\)](#).

Users should only call this method if they intend to modify the objective with [nlp_obj](#), the constraints with [nlp_cons](#), [nlp_cons_lb](#) and [nlp_cons_ub](#).

To finish the setup process, users MUST call [create_nlp\(\)](#) afterwards.

Note: Do NOT call [setup\(\)](#) if you intend to go the manual route with [prepare_nlp\(\)](#) and [create_nlp\(\)](#).

Note: Only AFTER calling `prepare_nlp()` the previously mentioned attributes `nlp_obj`, `nlp_cons`, `nlp_cons_lb`, `nlp_cons_ub` become available.

Returns

None – None

get_p_template

`get_p_template(self, n_combinations)`

Obtain output template for `set_p_fun()`.

Low level API method to set user defined scenarios for robust multi-stage MPC by defining an arbitrary number of combinations for the parameters defined in the model. For more details on robust multi-stage MPC please read our [background article](#)

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

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

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

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

Example:

```
# in model definition:
alpha = model.set_variable(var_type='_p', var_name='alpha')
beta = model.set_variable(var_type='_p', var_name='beta')

...
# in MPC configuration:
n_combinations = 3
p_template = MPC.get_p_template(n_combinations)
p_template['_p',0] = np.array([1,1])
p_template['_p',1] = np.array([0.9, 1.1])
p_template['_p',2] = np.array([1.1, 0.9])

def p_fun(t_now):
    return p_template

MPC.set_p_fun(p_fun)
```

Note the nominal case is now: `alpha = 1`, `beta = 1` which is determined by the order in the arrays above (first element is nominal).

Parameters

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

Return type

None

get_tvp_template**get_tvp_template(self)**

Obtain output template for `set_tvp_fun()`.

The method returns a structured object with `n_horizon+1` elements, and a set of time-varying parameters (as defined in `do_mpc.model.Model`) for each of these instances. The structure is initialized with all zeros. Use this object to define values of the time-varying parameters.

This structure (with numerical values) should be used as the output of the `tvp_fun` function which is set to the class with `set_tvp_fun()`. Use the combination of `get_tvp_template()` and `set_tvp_fun()`.

Example:

```
# 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

Union[SXStruct, MXStruct] – Casadi SX or MX structure

make_step**make_step(self, x0)**

Main method of the class during runtime. This method is called at each timestep and returns the control input for the current initial state `x0`.

The method prepares the MHE by setting the current parameters, calls `solve()` and updates the `do_mpc.data.Data` object.

Parameters

`x0` (Union[ndarray, DM]) – Current state of the system.

Returns

ndarray – `u0`

prepare_nlp

prepare_nlp(self)

Prepare the optimization problem. Typically, this method is called internally from `setup()`.

Users should only call this method if they intend to modify the objective with `nlp_obj`, the constraints with `nlp_cons`, `nlp_cons_lb` and `nlp_cons_ub`.

To finish the setup process, users MUST call `create_nlp()` afterwards.

Note: Do NOT call `setup()` if you intend to go the manual route with `prepare_nlp()` and `create_nlp()`.

Note: Only AFTER calling `prepare_nlp()` the previously mentioned attributes `nlp_obj`, `nlp_cons`, `nlp_cons_lb`, `nlp_cons_ub` become available.

Returns

None – None

reset_history

reset_history(self)

Reset the history of the optimizer. All data from the `do_mpc.data.Data` instance is removed.

Return type

None

set_initial_guess

set_initial_guess(self)

Initial guess for optimization variables. Uses the current class attributes `x0`, `z0` and `u0` to create the initial guess. The initial guess is simply the initial values for all $k = 0, \dots, N$ instances of x_k , u_k and z_k . :rtype: None

Warning: If no initial values for `x0`, `z0` and `u0` were supplied during setup, these default to zero.

Note: The initial guess is fully customizable by directly setting values on the class attribute: `opt_x_num`.

set_nl_cons

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** (str) – Arbitrary name for the given expression. Names are used for key word indexing.
- **expr** (Union[SX, MX]) – CasADi SX or MX function depending on `_x`, `_u`, `_z`, `_tvp`, `_p`.
- **ub** (float) – Upper bound
- **soft_constraint** (bool) – Flag to enable soft constraint
- **penalty_term_cons** (int) – Penalty term constant
- **maximum_violation** (float) – Maximum violation

Raises

- **assertion** – `expr_name` must be str
- **assertion** – `expr` must be a casadi SX or MX type

Returns

Union[SX, MX] – Returns the newly created expression. Expression can be used e.g. for the RHS.

set_objective

set_objective(*self*, *mterm=None*, *lterm=None*)

Sets the objective of the optimal control problem (OCP). We introduce the following cost function:

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

which is applied to the discrete-time model **AND** the discretized continuous-time model. For discretization we use [orthogonal collocation on finite elements](#). The cost function is evaluated only on the first collocation point of each interval.

`set_objective()` is used to set the $l(x_k, z_k, u_k, p_k, p_{tv,k})$ (**lterm**) and $m(x_{N+1})$ (**mterm**), where N is the prediction horizon. Please see `set_rterm()` for the penalization of the control inputs.

Parameters

- **lterm** (Union[SX, MX, None]) – Stage cost - **scalar** symbolic expression with respect to `_x`, `_u`, `_z`, `_tvp`, `_p`
- **mterm** (Union[SX, MX, None]) – Terminal cost - **scalar** symbolic expression with respect to `_x` and `_p`

Raises

- **assertion** – mterm must have `shape=(1,1)` (scalar expression)
- **assertion** – lterm must have `shape=(1,1)` (scalar expression)

Return type

None

set_p_fun**set_p_fun**(self, p_fun)

Set function which returns parameters. The `p_fun` is called at each optimization step to get the current values of the (uncertain) parameters.

This is the low-level API method to set user defined scenarios for robust multi-stage MPC by defining an arbitrary number of combinations for the parameters defined in the model. For more details on robust multi-stage MPC please read our [background article](#).

The method takes as input a function, which MUST return a structured object, based on the defined parameters and the number of combinations. The defined function has time as a single input.

Obtain this structured object first, by calling `get_p_template()`.

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

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

Example:

```
# in model definition:
alpha = model.set_variable(var_type='_p', var_name='alpha')
beta = model.set_variable(var_type='_p', var_name='beta')

...
# in MPC configuration:
n_combinations = 3
p_template = MPC.get_p_template(n_combinations)
p_template['_p',0] = np.array([1,1])
p_template['_p',1] = np.array([0.9, 1.1])
p_template['_p',2] = np.array([1.1, 0.9])

def p_fun(t_now):
    return p_template

MPC.set_p_fun(p_fun)
```

Note the nominal case is now: `alpha = 1`, `beta = 1` which is determined by the order in the arrays above (first element is nominal).

Parameters

p_fun (Callable[[float], Union[SXStruct, MXStruct]]) – Function which returns a structure with numerical values. Must be the same structure as obtained from [get_p_template\(\)](#). Function must have a single input (time).

Return type

None

set_param

set_param(self, **kwargs)

Set the parameters of the MPC class. Parameters must be passed as pairs of valid keywords and respective argument. :rtype: None

Deprecated since version >v4.5.1: This function will be deprecated in the future

Note: A comprehensive list of all available parameters can be found in [do_mpc.controller.MPCSettings](#)

For example:

```
mpc.settings.n_horizon = 20
```

The old interface, as shown in the example below, can still be accessed until further notice.

```
mpc.set_param(n_horizon = 20)
```

Note: The only required parameters are `n_horizon` and `t_step`. All other parameters are optional.

Note: We highly suggest to change the linear solver for IPOPT from *mumps* to *MA27*. Any available linear solver can be set using [do_mpc.controller.MPCSettings.set_linear_solver\(\)](#). For more details, please check the [do_mpc.controller.MPCSettings](#).

Note: The output of IPOPT can be suppressed [do_mpc.controller.MPCSettings.supress_ipopt_output\(\)](#). For more details, please check the [do_mpc.controller.MPCSettings](#).

set_rterm

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: :rtype: None

$$\Delta u_{k,i} = u_{k,i} - u_{k-1,i}$$

and add:

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

the weighted squared cost to the MPC objective function.

Example:

```
# in model definition:
Q_heat = model.set_variable(var_type='_u', var_name='Q_heat')
F_flow = model.set_variable(var_type='_u', var_name='F_flow')

...
# in MPC configuration:
MPC.set_rterm(Q_heat = 10)
MPC.set_rterm(F_flow = 10)
# or alternatively:
MPC.set_rterm(Q_heat = 10, F_flow = 10)
```

In the above example we set $r_{Q_{\text{heat}}} = 10$ and $r_{F_{\text{flow}}} = 10$.

Note: For $k = 0$ we obtain u_{-1} from the previous solution.

set_tvp_fun

set_tvp_fun(self, tvp_fun)

Set function which returns time-varying parameters.

The tvp_fun is called at each optimization step to get the current prediction of the time-varying parameters. The supplied function must be callable with the current time as the only input. Furthermore, the function must return a CasADi structured object which is based on the horizon and on the model definition. The structure can be obtained with [get_tvp_template\(\)](#).

Example:

```
# in model definition:
alpha = model.set_variable(var_type='_tvp', var_name='alpha')
beta = model.set_variable(var_type='_tvp', var_name='beta')

...
# in optimizer configuration:
tvp_temp_1 = optimizer.get_tvp_template()
tvp_temp_1['_tvp', :] = np.array([1,1])

tvp_temp_2 = optimizer.get_tvp_template()
tvp_temp_2['_tvp', :] = np.array([0,0])

def tvp_fun(t_now):
    if t_now < 10:
        return tvp_temp_1
    else:
        tvp_temp_2

optimizer.set_tvp_fun(tvp_fun)
```

Note: The method [set_tvp_fun\(\)](#) must be called prior to setup IF time-varying parameters are defined in

the model. It is not required to call the method if no time-varying parameters are defined.

Parameters

ttp_fun (Callable[[float], Union[SXStruct, MXStruct]]) – Function that returns the predicted ttp values at each timestep. Must have single input (float) and return a structure3.DMStruct (obtained with [get_ttp_template\(\)](#)).

Return type

None

set_uncertainty_values

set_uncertainty_values(*self*, ***kwargs*)

Define scenarios for the uncertain parameters. High-level API method to conveniently set all possible scenarios for multistage MPC. For more details on robust multi-stage MPC please read our [background article](#) .

Pass a number of keyword arguments, where each keyword refers to a user defined parameter name from the model definition. The value for each parameter must be an array (or list), with an arbitrary number of possible values for this parameter. The first element is the nominal case.

Example:

```
# in model definition:
alpha = model.set_variable(var_type='_p', var_name='alpha')
beta = model.set_variable(var_type='_p', var_name='beta')
gamma = model.set_variable(var_type='_p', var_name='gamma')
...
# in MPC configuration:
alpha_var = np.array([1., 0.9, 1.1])
beta_var = np.array([1., 1.05])
MPC.set_uncertainty_values(
    alpha = alpha_var,
    beta = beta_var
)
```

Note: Parameters that are not important for the MPC controller (e.g. MHE tuning matrices) can be ignored with the new interface (see `gamma` in the example above).

Note the nominal case is now: `alpha = 1, beta = 1` which is determined by the order in the arrays above (first element is nominal).

Parameters

kwargs – Arbitrary number of keyword arguments.

Return type

None

setup

setup(self)

Setup the MPC class. Internally, this method will create the MPC optimization problem under consideration of the supplied dynamic model and the given MPC class instance configuration.

The `setup()` method can be called again after changing the configuration (e.g. adapting bounds) and will simply overwrite the previous optimization problem. :rtype: None

Note: After this call, the `solve()` and `make_step()` method is applicable.

Warning: The `setup()` method may take a while depending on the size of your MPC problem. Note that especially for robust multi-stage MPC with a long robust horizon and many possible combinations of the uncertain parameters very large problems will arise.

For more details on robust multi-stage MPC please read our [background article](#) .

solve

solve(self)

Solves the optimization problem.

The current problem is defined by the parameters in the `opt_p_num` CasADi structured Data.

Typically, `opt_p_num` is prepared for the current iteration in the `make_step()` method. It is, however, valid and possible to directly set parameters in `opt_p_num` before calling `solve()`.

The method updates the `opt_p_num` and `opt_x_num` attributes of the class. By resetting `opt_x_num` to the current solution, the method implicitly enables **warmstarting the optimizer** for the next iteration, since this vector is always used as the initial guess. :rtype: None

Warning: The method is part of the public API but it is generally not advised to use it. Instead we recommend to call `make_step()` at each iterations, which acts as a wrapper for `solve()`.

Raises

assertion – Optimizer was not setup yet.

4.10.1.3.2 Attributes

bounds

MPC.bounds

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

lb_opt_x

MPC.lb_opt_x

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

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

Note: The attribute automatically considers the scaling variables when setting the bounds. See `scaling` for more details.

Note: Modifications must be done after calling `prepare_nlp()` or `setup()` respectively.

nlp_cons

MPC.nlp_cons

Query and modify (symbolically) the NLP constraints. Use the variables in `opt_x` and `opt_p`.

Prior to calling `create_nlp()` this attribute returns a list of symbolic constraints. After calling `create_nlp()` this attribute returns the concatenation of this list and the attribute cannot be altered anymore.

It is advised to append to the current list of `nlp_cons`:

```
mpc.prepare_nlp()

# Create new constraint: Input at timestep 0 and 1 must be identical.
extra_cons = mpc.opt_x['_u', 0, 0] - mpc.opt_x['_u', 1, 0]
mpc.nlp_cons.append(
    extra_cons
)
```

(continues on next page)

(continued from previous page)

```
# Create appropriate upper and lower bound (here they are both 0 to create an
→equality constraint)
mpc.nlp_cons_lb.append(np.zeros(extra_cons.shape))
mpc.nlp_cons_ub.append(np.zeros(extra_cons.shape))

mpc.create_nlp()
```

See the documentation of *opt_x* and *opt_p* on how to query these attributes.

Warning: This is a VERY low level feature and should be used with extreme caution. It is easy to break the code.

Be especially careful NOT to accidentally overwrite the default objective.

Note: Modifications must be done after calling *prepare_nlp()* and before calling *create_nlp()*

nlp_cons_lb

MPC.nlp_cons_lb

Query and modify the lower bounds of the *nlp_cons*.

Prior to calling *create_nlp()* this attribute returns a list of lower bounds matching the list of constraints obtained with *nlp_cons*. After calling *create_nlp()* this attribute returns the concatenation of this list.

Values for lower (and upper) bounds MUST be added when adding new constraints to *nlp_cons*.

Warning: This is a VERY low level feature and should be used with extreme caution. It is easy to break the code.

Note: Modifications must be done after calling *prepare_nlp()*

nlp_cons_ub

MPC.nlp_cons_ub

Query and modify the upper bounds of the *nlp_cons*.

Prior to calling *create_nlp()* this attribute returns a list of upper bounds matching the list of constraints obtained with *nlp_cons*. After calling *create_nlp()* this attribute returns the concatenation of this list.

Values for upper (and lower) bounds MUST be added when adding new constraints to *nlp_cons*.

Warning: This is a VERY low level feature and should be used with extreme caution. It is easy to break the code.

Note: Modifications must be done after calling `prepare_nlp()`

nlp_obj

MPC.nlp_obj

Query and modify (symbolically) the NLP objective function. Use the variables in `opt_x` and `opt_p`.

It is advised to add to the current objective, e.g.:

```
mpc.prepare_nlp()
# Modify the objective
mpc.nlp_obj += sum1(vercat(*mpc.opt_x['_x', -1, 0])**2)
# Finish creating the NLP
mpc.create_nlp()
```

See the documentation of `opt_x` and `opt_p` on how to query these attributes.

Warning: This is a VERY low level feature and should be used with extreme caution. It is easy to break the code.

Be especially careful NOT to accidentally overwrite the default objective.

Note: Modifications must be done after calling `prepare_nlp()` and before calling `create_nlp()`

opt_p

MPC.opt_p

Full structure of (symbolic) MPC parameters.

The attribute is a CasADi numeric structure with nested power indices. It can be indexed as follows:

```
# initial state:
opt_p['_x0', _x_name]
# uncertain scenario parameters
opt_p['_p', scenario, _p_name]
# time-varying parameters:
opt_p['_tvp', time_step, _tvp_name]
# input at time k-1:
opt_p['_u_prev', time_step, scenario]
```

The names refer to those given in the `do_mpc.model.Model` configuration. Further indices are possible, if the variables are itself vectors or matrices.

Warning: Do not tweak or overwrite this attribute unless you known what you are doing.

Note: The attribute is populated when calling `setup()` or `prepare_nlp()`

opt_p_num

MPC.opt_p_num

Full MPC parameter vector.

This attribute is used when calling the MPC solver to pass all required parameters, including

- initial state
- uncertain scenario parameters
- time-varying parameters
- previous input sequence

do-mpc handles setting these parameters automatically in the `make_step()` method. However, you can set these values manually and directly call `solve()`.

The attribute is a CasADi numeric structure with nested power indices. It can be indexed as follows:

```
# initial state:
opt_p_num['_x0', _x_name]
# uncertain scenario parameters
opt_p_num['_p', scenario, _p_name]
# time-varying parameters:
opt_p_num['_tvp', time_step, _tvp_name]
# input at time k-1:
opt_p_num['_u_prev', time_step, scenario]
```

The names refer to those given in the `do_mpc.model.Model` configuration. Further indices are possible, if the variables are itself vectors or matrices.

<p>Warning: Do not tweak or overwrite this attribute unless you know what you are doing.</p>

Note: The attribute is populated when calling `setup()`

opt_x

MPC.opt_x

Full structure of (symbolic) MPC optimization variables.

The attribute is a CasADi symbolic structure with nested power indices. It can be indexed as follows:

```
# dynamic states:
opt_x['_x', time_step, scenario, collocation_point, _x_name]
# algebraic states:
opt_x['_z', time_step, scenario, collocation_point, _z_name]
# inputs:
```

(continues on next page)

(continued from previous page)

```
opt_x['_u', time_step, scenario, _u_name]
# slack variables for soft constraints:
opt_x['_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 alter the objective function or constraints of the NLP.

How to query?

Querying the structure is more complicated than it seems at first look because of the scenario-tree used for robust MPC. To obtain all collocation points for the finite element at time-step k and scenario b use:

```
horzcat(*[mpc.opt_x['_x',k,b,-1]]+mpc.opt_x['_x',k+1,b,:-1])
```

Due to the multi-stage formulation at any given time k we can have multiple future scenarios. However, there is only exactly one scenario that lead to the current node in the tree. Thus the collocation points associated to the finite element k lie in the past.

The concept is illustrated in the figure below:

Note: The attribute `opt_x` carries the scaled values of all variables.

Note: The attribute is populated when calling `setup()` or `prepare_nlp()`

opt_x_num

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**.

How to query?

Querying the structure is more complicated than it seems at first look because of the scenario-tree used for robust MPC. To obtain all collocation points for the finite element at time-step k and scenario b use:

```
horzcat(*[mpc.opt_x_num['_x',k,b,-1]]+mpc.opt_x_num['_x',k+1,b,:-1])
```

Due to the multi-stage formulation at any given time k we can have multiple future scenarios. However, there is only exactly one scenario that lead to the current node in the tree. Thus the collocation points associated to the finite element k lie in the past.

The concept is illustrated in the figure below:

Note: The attribute `opt_x_num` carries the scaled values of all variables. See `opt_x_num_unscaled` for the unscaled values (these are not used as the initial guess).

Warning: Do not tweak or overwrite this attribute unless you known what you are doing.

Note: The attribute is populated when calling `setup()`

scaling

MPC.scaling

Query and set scaling of the optimization variables. The `Optimizer.scaling()` method is an indexed property, meaning getting and setting this property requires an index and calls this function. The power index (elements are seperated by comas) must contain atleast the following elements:

order	index name	valid options
1	variable type	<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.Model</code> .

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

Example:

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

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

Scaling factors a affect the MHE / MPC optimization problem. The optimization variables are scaled variables:

$$\bar{\phi} = \frac{\phi}{a_{\phi}} \quad \forall \phi \in [x, u, z, p_{\text{est}}]$$

Scaled variables are used to formulate the bounds $\bar{\phi}_{lb} \leq \bar{\phi}_{ub}$ and for the evaluation of the ODE. For the objective function and the nonlinear constraints the unscaled variables are used. The algebraic equations are also not scaled.

Note: Scaling the optimization problem is suggested when states and / or inputs take on values which differ by orders of magnitude.

t0

MPC.t0

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

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

terminal_bounds

MPC.terminal_bounds

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

order	index name	valid options
1	bound type	lower and upper
2	variable name	Names defined in <code>do_mpc.model.Model</code> .

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

Example:

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

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

u0

MPC.u0

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

Example:

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

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

(continues on next page)

(continued from previous page)

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

Useful CasADi symbolic structure methods:

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

ub_opt_x

MPC.ub_opt_x

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

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

Note: The attribute automatically considers the scaling variables when setting the bounds. See `scaling` for more details.

Note: Modifications must be done after calling `prepare_nlp()` or `setup()` respectively.

x0

MPC.x0

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

Example:

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

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

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

Useful CasADi symbolic structure methods:

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

`z0`

`MPC.z0`

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

Example:

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

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

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

Useful CasADi symbolic structure methods:

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

4.10.1.4 MPCSettings

```
class MPCSettings(t_step=None, n_horizon=None, n_robust=0, open_loop=False, use_terminal_bounds=False,
                  state_discretization='collocation', collocation_type='radau', collocation_deg=2,
                  collocation_ni=1, nl_cons_check_colloc_points=False, nl_cons_single_slack=False,
                  cons_check_colloc_points=True, store_full_solution=False, store_lagr_multiplier=True,
                  store_solver_stats=<factory>, nlpsol_opts=<factory>)
```

Bases: `ControllerSettings`

Settings for `do_mpc.controller.MPC`. The `do_mpc.controller.MPC` automatically creates an instance of type `MPCSettings` and adds it to its class attributes.

Example to change settings:

```
mpc.settings.n_horizon = 20
```

Note: Settings cannot be updated after calling `do_mpc.controller.MPC.setup()`.

Parameters

- **t_step** (Optional[float]) –
- **n_horizon** (Optional[int]) –
- **n_robust** (int) –
- **open_loop** (bool) –
- **use_terminal_bounds** (bool) –
- **state_discretization** (str) –
- **collocation_type** (str) –
- **collocation_deg** (int) –
- **collocation_ni** (int) –
- **nl_cons_check_colloc_points** (bool) –
- **nl_cons_single_slack** (bool) –
- **cons_check_colloc_points** (bool) –
- **store_full_solution** (bool) –
- **store_lagr_multiplier** (bool) –
- **store_solver_stats** (List[str]) –
- **nlpsol_opts** (Dict) –

4.10.1.4.1 Methods

check_for_mandatory_settings

check_for_mandatory_settings(*self*)

Method to assert the necessary settings required to design *do_mpc.controller.MPC*

set_linear_solver

set_linear_solver(*self*, *solver_name*='MA27')

Method to set the linear solver to MA27.

This method enables to set the linear solver to MA27. This change in many cases will drastically boost the speed of do-mpc.

Example:

```
mpc.settings.set_linear_solver(solver_name = "MA27")
```

Parameters

solver_name (str) – Specify the linear solver name

supress_ipopt_output

supress_ipopt_output(*self*)

Method to supress the ipopt solver output.

This method set the revelvant settings in the ipopt solver in order to supress the output log.

4.10.1.4.2 Attributes

collocation_deg

MPCSettings.collocation_deg: int = 2

Choose the collocation degree for continuous models with collocation as state discretization.

collocation_ni

MPCSettings.collocation_ni: int = 1

For orthogonal collocation choose the number of finite elements for the states within a time-step (and during constant control input).

Can be used to avoid high-order polynomials.

collocation_type

MPCSettings.collocation_type: str = 'radau'

Choose the collocation type for continuous models with collocation as state discretization.

Note: Currently only 'radau' is available.

cons_check_colloc_points

MPCSettings.cons_check_colloc_points: bool = True

For orthogonal collocation choose whether the linear bounds set with *do_mpc.controller.MPC.bounds* are evaluated once per finite Element or for each collocation point.

n_horizon

MPCSettings.n_horizon: int = None

Prediction horizon of the optimal control problem.

Parameter must be set by user

n_robust

`MPCSettings.n_robust: int = 0`

Robust horizon for robust scenario-tree MPC.

Note: Optimization problem grows exponentially with `n_robust`.

nl_cons_check_colloc_points

`MPCSettings.nl_cons_check_colloc_points: bool = False`

For orthogonal collocation choose whether the nonlinear bounds set with `do_mpc.controller.MPC.set_nl_cons()` are evaluated once per finite Element or for each collocation point.

nl_cons_single_slack

`MPCSettings.nl_cons_single_slack: bool = False`

If `True`, soft-constraints set with `do_mpc.controller.MPC.set_nl_cons()` introduce only a single slack variable for the entire horizon.

open_loop

`MPCSettings.open_loop: bool = False`

Setting for scenario-tree MPC.

Note: 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.

state_discretization

`MPCSettings.state_discretization: str = 'collocation'`

Choose the state discretization for continuous models.

Note: Currently only 'collocation' is available. Defaults to 'collocation'. Has no effect if model is created in discrete type.

store_full_solution

`MPCSettings.store_full_solution: bool = False`

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.

store_lagr_multiplier

`MPCSettings.store_lagr_multiplier: bool = True`

Choose whether to store the lagrange multipliers of the optimization problem.

Note: Increases the required storage.

t_step

`MPCSettings.t_step: float = None`

Timestep of the controller

use_terminal_bounds

`MPCSettings.use_terminal_bounds: bool = False`

Choose if terminal bounds for the states are used.

Set terminal bounds with `do_mpc.controller.MPC.terminal_bounds`.

store_solver_stats

`MPCSettings.store_solver_stats: List[str]`

Choose which solver statistics to store.

Must be a list of valid statistics. This attribute is an object of type list.

nlpsol_opts

`MPCSettings.nlpsol_opts: Dict`

Dictionary with options for the CasADi solver call `nlpsol` with plugin `ipopt`.

All options are listed [here](#).

4.10.2 data

Storage and handling of data.

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.

4.10.2.1 load_results

Class method.

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.

Returns

Dict – Returns the results stored in .pkl file.

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

4.10.2.2 save_results

Class method.

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** (str) – Name of the result file, defaults to 'result'.
- **result_path** (str) – Result path, defaults to './results/'.
- **overwrite** (bool) – Option to overwrite existing results, defaults to False. Index will be appended if file already exists.

Raises

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

Return type

None

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

Classes

<i>Data</i>	do-mpc data container.
<i>MPCData</i>	do-mpc data container for the <i>do_mpc.controller.MPC</i> instance.

4.10.2.3 Data**class Data(model)**

Bases: object

do-mpc data container. An instance of this class is created for the active **do-mpc** classes, e.g. *do_mpc.simulator.Simulator*, *do_mpc.estimator.MHE*.

The class is initialized with an instance of the *do_mpc.model.Model* which contains all information about variables (e.g. states, inputs etc.).

The *Data* class has a public API but is mostly used by other **do-mpc** classes, e.g. updated in the `.make_step` calls.

Parameters

model (Union[*Model*, *LinearModel*]) – model object from the *do_mpc.model*

__getitem__(ind)

Query data fields. This method can be used to obtain the stored results in the *Data* instance.

The full list of available fields can be inspected with:

```
print(data.data_fields)
```

The dict also denotes the dimension of each field.

The method allows for power indexing the results for the fields `_x`, `_u`, `_z`, `_tvp`, `_p`, `_aux`, `_y` where further indices refer to the configured variables in the *do_mpc.model.Model* and *do_mpc.model.LinearModel* instance.

Example:


```

# Assume the following model was used (excerpt):
model = do_mpc.model.Model('continuous')

model.set_variable('_x', 'Temperature', shape=(5,1)) # Vector
model.set_variable('_p', 'disturbance', shape=(3,3)) # Matrix
model.set_variable('_u', 'heating')                  # scalar

...

# the model was used (among others) for the MPC controller
mpc = do_mpc.controller.MPC(model)

...

# Query the mpc.data instance:
mpc.data['_x']                # Return all states
mpc.data['_x', 'Temperature'] # Return the 5 temp states
mpc.data['_x', 'Temperature', :2] # Return the first 2 temp. states
mpc.data['_p', 'disturbance', 0, 2] # Matrix allows for further indices

# Other fields can also be queried, e.g.:
mpc.data['_time']              # current time
mpc.data['t_wall_total']       # optimizer runtime
# These do not allow further indices.

```

Parameters

ind (Tuple) – Power index to query the prediction of a specific variable.

Returns

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

4.10.2.3.1 Methods**export****export(self)**

The export method returns a dictionary of the stored data.

Returns

dict – Dictionary of the currently stored data.

init_storage**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.

Return type

None

set_meta

set_meta(*self*, ***kwargs*)

Set meta data for the current instance of the data object.

Return type

None

update

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 (Dict[ndarray, DM]) – Arbitrary number of key word arguments for data fields that should be updated.

Raises

assertion – Keyword must be in existing `data_fields`.

Return type

None

4.10.2.4 MPCData

class MPCData(*model*)

Bases: [Data](#)

do-mpc data container for the [do_mpc.controller.MPC](#) instance. This method inherits from [Data](#) and extends it to query the MPC predictions.

Warning: For robust multi-stage MPC, the [MPCData](#) class stores by default only the nominal values of the uncertain parameters.

Parameters

model (Union[[Model](#), [LinearModel](#)]) – model from *do_mpc.model*

__getitem__(ind)

Query data fields. This method can be used to obtain the stored results in the [Data](#) instance.

The full list of available fields can be inspected with:

```
print(data.data_fields)
```

The dict also denotes the dimension of each field.

The method allows for power indexing the results for the fields `_x`, `_u`, `_z`, `_tvp`, `_p`, `_aux`, `_y` where further indices refer to the configured variables in the *do_mpc.model.Model* and *do_mpc.model.LinearModel* instance.

Example:

```
# Assume the following model was used (excerpt):
model = do_mpc.model.Model('continuous')

model.set_variable('_x', 'Temperature', shape=(5,1)) # Vector
model.set_variable('_p', 'disturbance', shape=(3,3)) # Matrix
model.set_variable('_u', 'heating')                  # scalar

...

# the model was used (among others) for the MPC controller
mpc = do_mpc.controller.MPC(model)

...

# Query the mpc.data instance:
mpc.data['_x']                # Return all states
mpc.data['_x', 'Temperature'] # Return the 5 temp states
mpc.data['_x', 'Temperature', :2] # Return the first 2 temp. states
mpc.data['_p', 'disturbance', 0, 2] # Matrix allows for further indices

# Other fields can also be queried, e.g.:
mpc.data['_time']              # current time
mpc.data['t_wall_total']       # optimizer runtime
# These do not allow further indices.
```

Parameters

ind (Tuple) – Power index to query the prediction of a specific variable.

Returns

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

4.10.2.4.1 Methods

export

export(*self*)

The export method returns a dictionary of the stored data.

Returns

dict – Dictionary of the currently stored data.

init_storage

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.

Return type

None

prediction

prediction(*self*, *ind*, *t_ind=-1*)

Query the MPC trajectories. Use this method to obtain specific MPC trajectories from the data object.

Warning: This method requires that the optimal solution is stored in the `do_mpc.data.MPCData` instance. Storing the optimal solution must be activated with `do_mpc.controller.MPC.set_param()`.

Querying predicted trajectories requires the use of power indices, which is passed as tuple e.g.:

```
data.prediction((var_type, var_name, i), t_ind)
```

where

- `var_type` refers to `_x`, `_u`, `_z`, `_tvp`, `_p`, `_aux`
- `var_name` refers to the user-defined names in the `do_mpc.model.Model`
- Use `i` to index vector valued variables.

The method returns a multidimensional `numpy.ndarray`. The dimensions refer to:

```
arr = data.prediction(('x', 'x_1'))
arr.shape
>> (n_size, n_horizon, n_scenario)
```

with:

- `n_size` denoting the number of elements in `x_1`, where `n_size = 1` is a scalar variable.
- `n_horizon` is the MPC horizon defined with `do_mpc.controller.MPC.set_param()`
- `n_scenario` refers to the number of uncertain scenarios (for robust MPC).

Additional to the power index tuple, a time index (`t_ind`) can be passed to access the prediction for a certain time.

Parameters

- **ind** (tuple) – Power index to query the prediction of a specific variable.
- **t_ind** (float) – Time index

Returns

ndarray – Predicted trajectories for the queries variable.

set_meta

set_meta(*self*, ***kwargs*)

Set meta data for the current instance of the data object.

Return type

None

update

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 (Dict[ndarray, DM]) – Arbitrary number of key word arguments for data fields that should be updated.

Raises

assertion – Keyword must be in existing `data_fields`.

Return type

None

4.10.3 differentiator

Tools for NLP differentiation.

Warning: This is an experimental feature. The API might change in the future.

Classes

<i>DoMPCDifferentiator</i>	Nonlinear program (NLP) Differentiator for <code>do_mpc</code> objects.
<i>NLPDifferentiator</i>	Base class for nonlinear program (NLP) differentiator.

4.10.3.1 DoMPCDifferentiator

class `DoMPCDifferentiator`(*optimizer*, ***kwargs*)

Bases: *NLPDifferentiator*

Nonlinear program (NLP) Differentiator for `do_mpc` objects. Can be used with `do_mpc.controller.MPC` and `do_mpc.estimator.MHE` objects. The class inherits the *NLPDifferentiator* class and overwrites the `differentiate()` method.

Example:

1. Setup a `do_mpc` optimizer object (e.g. `do_mpc.controller.MPC` or `do_mpc.estimator.MHE`).

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

2. Initialize the differentiator with the `do_mpc` optimizer object.

```
nlp_diff = DoMPCDifferentiator(mpc)
```

3. Configure the differentiator settings with the `settings` attribute.

```
nlp_diff.settings.check_LICQ = False
```

4. Solve the NLP of the original `do_mpc` optimizer object.

```
mpc.make_step(x0)
```

5. Call the `differentiate()` method of the differentiator object to compute the parametric sensitivities. The current parameters and optimal solution are read from the `do_mpc` optimizer object.

```
dx_dp_num, dlam_dp_num = nlp_diff.differentiate()
```

6. Typically, we are interested in specific segments of the parametric sensitivities. These can be retrieved by powerindexing the `sens_num` attribute.

```
du0dx0 = nlp_diff.sens_num['dxdp', indexf['_u', 0, 0], indexf['_x', 0, 0]]
```

This last step returns the parametric sensitivity of the first input with respect to the initial state.

Parameters

optimizer (*Optimizer*) – `do_mpc` class that inherits the `Optimizer` class, that is, a `do_mpc.controller.MPC` or `do_mpc.estimator.MHE` object.

4.10.3.1.1 Methods

differentiate

`differentiate(self)`

Main method of the class. Computes the parametric sensitivities of the underlying NLP of the MPC or MHE. Should be called after solving the underlying NLP. The current optimal solution and the corresponding parameters are read from the `do_mpc` object.

4.10.3.1.2 Attributes

`sens_num`

`DoMPCDifferentiator.sens_num`

The sensitivity structure of the NLP. This can be queried as follows:

```
from casadi.tools import indexf

du0dx0 = nlp_diff.sens_num['dxdp', indexf['_u', 0, 0], indexf['_x0']]
```

The powerindices passed to `indexf` are derived from the attributes:

- `do_mpc.controller.MPC.opt_x`
- `do_mpc.controller.MPC.opt_p`

settings

`DoMPCDifferentiator.settings`

Settings of the NLP differentiator. This is an annotated dataclass that can also be printed for convenience. See `do_mpc.differentiator.helper.NLPDifferentiatorSettings` for more information.

Example:

```
nlp_diff = NLPDifferentiator(nlp, nlp_bounds)
nlp_diff.settings.check_licq = False
```

Note: Settings can also be passed as keyword arguments to the constructor of `NLPDifferentiator`.

status

DoMPCDifferentiator.status

Status of the NLP differentiator. This is an annotated dataclass that can also be printed for convenience. See [do_mpc.differentiator.helper.NLPDifferentiatorStatus](#) for more information.

4.10.3.2 NLPDifferentiator

class NLPDifferentiator(nlp, nlp_bounds, **kwargs)

Bases: object

Base class for nonlinear program (NLP) differentiator. This class can be used independently from do-mpc to differentiate a given NLP.

Note: This is an experimental feature. The API might change in the future.

Example:

1. Consider an NLP created with CasADi, including

- optimization variables x
- optimization parameters p
- objective function f
- constraints g

```
import casadi as ca

x = ca.SX.sym('x', 2)
p = ca.SX.sym('p', 1)
f = (1-x[0])**2 + 0.2*(x[1]-x[0]**2)**2
cons_inner = (x[0] + 0.5)**2+x[1]**2

g = ca.vertcat(
    p**2/4 - cons_inner,
    cons_inner - p**2
)

ca_solver = ca.nlpsol('solver', 'ipopt', nlp)
```

2. Create dictionaries for the NLP and the NLP bounds:

```
nlp = {'x':x, 'p':p, 'f':cost, 'g':cons}
nlp_bounds = {
    'lbx': np.array([0, -ca.inf]).reshape(-1,1),
    'ubx': np.array([ca.inf, ca.inf]).reshape(-1,1),
    'lbg': np.array([-ca.inf, -ca.inf]).reshape(-1,1),
    'ubg': np.array([0, 0]).reshape(-1,1)
}
```

3. Initialize the NLP differentiator with the NLP and the NLP bounds.


```
nlp_diff = NLPDifferentiator(nlp, nlp_bounds)
```

4. Configure the differentiator settings with the `settings` attribute.

```
nlp_diff.settings.check_LICQ = False
```

5. Solve the parametric NLP for the parameters `p0`, e.g. with the CasADi solver `ca_solver`. Pass the same bounds that were used previously.

```
p0 = np.array([1.])
r = solver(p=p0, **nlp_bounds)
```

6. Calculate the parametric NLP sensitivity matrices with `differentiate()` considering the solution `r` and the corresponding parameters `p0`.

```
dxdp, dlamdp = nlp_diff.get_sensitivity_matrices(r, p0)
```

Parameters

- **nlp** (Dict) – Dictionary with keys `x`, `p`, `f`, `g`.
- **nlp_bounds** (Dict) – Dictionary with keys `lbx`, `ubx`, `lbg`, `ubg`.

4.10.3.2.1 Methods

`differentiate`

differentiate(*self*, *nlp_sol*, *p_num*)

Main method of the class. Call this method to obtain the parametric sensitivities. The sensitivity matrix `dx_dp` is of shape `(n_x, n_p)`.

Note: Please read the documentation of the class `NLPDifferentiator` for more information.

Parameters

- **nlp_sol** (dict) – Dictionary containing the optimal solution of the NLP.
- **p_num** (DM) – Numerical value of the parameters of the NLP.

Returns

Tuple[DM, DM] – Parametric sensitivities of the decision variables and lagrange multipliers.

4.10.3.2.2 Attributes

settings

NLPDifferenziator.settings

Settings of the NLP differentiator. This is an annotated dataclass that can also be printed for convenience. See [do_mpc.differentiator.helper.NLPDifferenziatorSettings](#) for more information.

Example:

```
nlp_diff = NLPDifferenziator(nlp, nlp_bounds)
nlp_diff.settings.check_licq = False
```

Note: Settings can also be passed as keyword arguments to the constructor of [NLPDifferenziator](#).

status

NLPDifferenziator.status

Status of the NLP differentiator. This is an annotated dataclass that can also be printed for convenience. See [do_mpc.differentiator.helper.NLPDifferenziatorStatus](#) for more information.

do_mpc.differentiator.helper	Helper functions for the NLPDifferenziator.
--	---

4.10.3.3 helper

Helper functions for the NLPDifferenziator.

Classes

NLPDifferenziatorSettings	Settings for NLPDifferenziator.
NLPDifferenziatorStatus	Status of the NLPDifferenziator.

4.10.3.3.1 NLPDifferenziatorSettings

```
class NLPDifferenziatorSettings(lin_solver=<factory>, check_LICQ=True, check_SC=True,  
                               track_residuals=True, check_rank=False, lstsq_fallback=False,  
                               active_set_tol=1e-06, set_lam_zero=False)
```

Bases: object

Settings for NLPDifferenziator.

Parameters

- **lin_solver** (str) –
- **check_LICQ** (bool) –
- **check_SC** (bool) –

- **track_residuals** (bool) –
- **check_rank** (bool) –
- **lstsq_fallback** (bool) –
- **active_set_tol** (float) –
- **set_lam_zero** (bool) –

Methods

Attributes

active_set_tol

`NLPDifferenziatorSettings.active_set_tol: float = 1e-06`

Tolerance for the active set constraints.

check_LICQ

`NLPDifferenziatorSettings.check_LICQ: bool = True`

Check if the constraints are linearly independent. The result of this check is stored in *NLPDifferenziatorStatus*.

Warning: This feature is computationally demanding and should only be used for debugging purposes.

check_SC

`NLPDifferenziatorSettings.check_SC: bool = True`

Check if strict complementarity holds. The result of this check is stored in *NLPDifferenziatorStatus*.

check_rank

`NLPDifferenziatorSettings.check_rank: bool = False`

Check if the KKT system has full rank. The result of this check is stored in *NLPDifferenziatorStatus*.

Warning: This feature is computationally demanding and should only be used for debugging purposes.

lstsq_fallback

NLPDifferenziatorSettings.**lstsq_fallback**: bool = False

Fallback to least squares if the linear solver fails.

set_lam_zero

NLPDifferenziatorSettings.**set_lam_zero**: bool = False

Set the Lagrangian multipliers to exactly zero if they are below the tolerance.

track_residuals

NLPDifferenziatorSettings.**track_residuals**: bool = True

Compute the residuals of the KKT system.

lin_solver

NLPDifferenziatorSettings.**lin_solver**: str

Choose the linear solver for the KKT system. Can be 'casadi', 'scipy' or 'lstsq' (least squares).

4.10.3.3.2 NLPDifferenziatorStatus

class NLPDifferenziatorStatus(*LICQ=None, SC=None, residuals=None, lse_solved=False, full_rank=None, sym_KKT=False, reduced_nlp=False*)

Bases: object

Status of the NLPDifferenziator.

Parameters

- **LICQ** (Optional[bool]) –
- **SC** (Optional[bool]) –
- **residuals** (Optional[ndarray]) –
- **lse_solved** (bool) –
- **full_rank** (Optional[bool]) –
- **sym_KKT** (bool) –
- **reduced_nlp** (bool) –

Methods

Attributes

LICQ

NLPDifferentiatorStatus.LICQ: Optional[bool] = None

Linear independence constraint qualification. Status is only updated if `check_LICQ` is set to `True`. The value is `None` if condition is not checked.

SC

NLPDifferentiatorStatus.SC: Optional[bool] = None

Strict complementarity. Status is only updated if `check_SC` is set to `True`. The value is `None` if condition is not checked.

full_rank

NLPDifferentiatorStatus.full_rank: Optional[bool] = None

Status of the rank of the KKT matrix. `True` if the matrix has full rank. Status is only updated if `check_rank` is set to `True`. The value is `None` if condition is not checked.

lse_solved

NLPDifferentiatorStatus.lse_solved: bool = False

Status of the linear system of equations. `True` if the system is solved successfully. The value is `None` if condition is not checked.

reduced_nlp

NLPDifferentiatorStatus.reduced_nlp: bool = False

Status of preparing the reduced NLP. `True` if the NLP has been prepared.

residuals

NLPDifferentiatorStatus.residuals: Optional[ndarray] = None

Residuals of the KKT system. Status is only updated if `track_residuals` is set to `True`. The value is `None` if condition is not checked.

sym_KKT

NLPDifferentiatorStatus.**sym_KKT**: bool = False

Status of preparing the symbolic KKT matrix. True if the matrix has been prepared.

4.10.4 estimator

State estimation for dynamic systems.

Classes

<i>EKF</i>	Extended Kalman Filter.
<i>Estimator</i>	The Estimator base class.
<i>MHE</i>	Moving horizon estimator.
<i>MHESettings</i>	Settings for <i>do_mpc.estimator.MHE</i> .
<i>StateFeedback</i>	Simple state-feedback "estimator".

4.10.4.1 EKF

class **EKF**(*model*)

Bases: [*Estimator*](#)

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.

4.10.4.1.1 Methods

make_step

make_step(*self*, *y0*)

Main method during runtime. Pass the most recent measurement and retrieve the estimated state.

reset_history

reset_history(*self*)

Reset the history of the estimator

Return type

None

4.10.4.1.2 Attributes

t0

EKF.t0

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

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

u0

EKF.u0

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

Example:

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

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

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

Useful CasADi symbolic structure methods:

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

x0

EKF.x0

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

Example:

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

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

(continues on next page)

(continued from previous page)

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

Useful CasADi symbolic structure methods:

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

z0

EKF.z0

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

Example:

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

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

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

Useful CasADi symbolic structure methods:

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

4.10.4.2 Estimator

class `Estimator(model)`

Bases: `IteratedVariables`

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`.

Parameters

model (Union[*Model*, *LinearModel*]) – model from class *do_mpc.model*

4.10.4.2.1 Methods**reset_history**

reset_history(*self*)

Reset the history of the estimator

Return type

None

4.10.4.2.2 Attributes**t0**

Estimator.**t0**

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

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

u0

Estimator.**u0**

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

Example:

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

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

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

Useful CasADi symbolic structure methods:

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

x0

Estimator.x0

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

Example:

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

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

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

Useful CasADi symbolic structure methods:

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

z0

Estimator.z0

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

Example:

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

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

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

Useful CasADi symbolic structure methods:

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

4.10.4.3 MHE

class `MHE`(*model*, *p_est_list*=[])

Bases: `Optimizer`, `Estimator`

Moving horizon estimator.

New in version >v4.5.1: New interface to settings. The class has an attribute `settings` which is an instance of `MHESettings` (please see this documentation for a list of available settings). Settings are now chosen as:

```
mhe.settings.n_horizon = 20
```

Previously, settings were passed to `set_param()`. This method is still available and wraps the new interface. The new method has important advantages:

1. The `mhe.settings` attribute can be printed to see the current configuration.
2. Context help is available in most IDEs (e.g. VS CODE) to see the available settings, the type and a description.
3. The `MHESettings` class has convenient methods, such as `MHESettings.supress_ipopt_output()` to silence the solver.

For general information on moving horizon estimation, please read our [background article](#).

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

The class is initiated by passing a list of the **parameters that should be estimated**. This must be a subset (or all) of the parameters defined in `do_mpc.model.Model`. This allows to define parameters in the model that influence the model externally (e.g. weather predictions), and those that are internal e.g. system parameters and can be estimated. Passing an empty list (default) value, means that no parameters are estimated.

Note: Parameters are influencing the model equation at all timesteps but are constant over the entire horizon. Parameters could also be introduced as states without dynamic but this would increase the total number of optimization variables.

Configuration and setup:

Configuring and setting up the MHE involves the following steps:

1. Configure the MHE controller with `MHESettings`. The MHE instance has the attribute `settings` which is an instance of `MHESettings`.
2. Set the objective of the control problem with `set_default_objective()` or use the low-level interface `set_objective()`.
5. Set upper and lower bounds.
6. Optionally, set further (non-linear) constraints with `set_nl_cons()`.
7. Use `get_p_template()` and `set_p_fun()` to set the function for the (not estimated) parameters.
8. Use `get_tvp_template()` and `set_tvp_fun()` to create a method to obtain new time-varying parameters at each iteration.
9. To finalize the class configuration there are two routes. The default approach is to call `setup()`. For deep customization use the combination of `prepare_nlp()` and `create_nlp()`. See graph below for an illustration of the process.

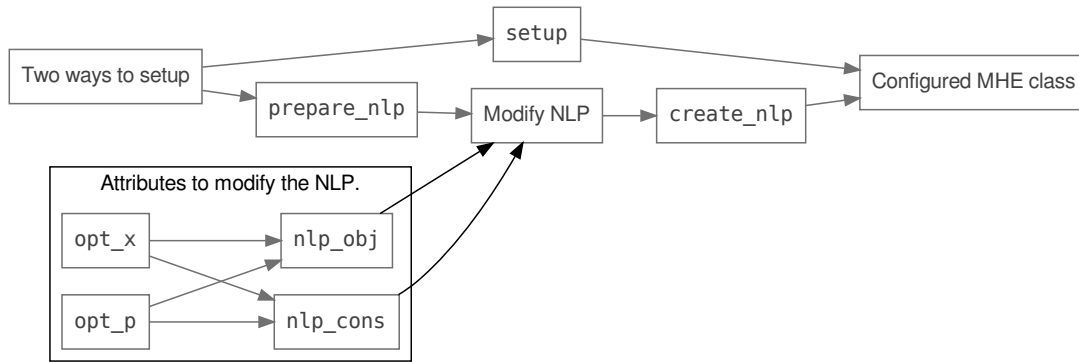


Fig. 5: Route to setting up the MHE class.

Warning: Before running the estimator, make sure to supply a valid initial guess for all estimated variables (states, algebraic states, inputs and parameters). Simply set the initial values of `x0`, `z0`, `u0` and `p_est0` and then call `set_initial_guess()`. To take full control over the initial guess, modify the values of `opt_x_num`.

During runtime use `make_step()` with the most recent measurement to obtain the estimated states.

Parameters

- **model** (Union[`Model`, `LinearModel`]) – A configured and setup `do_mpc.model`
- **p_est_list** (list) – List with names of parameters (`_p`) defined in `model`

4.10.4.3.1 Methods

`compile_nlp`

compile_nlp(*self*, *overwrite=False*, *cname='nlp.c'*, *libname='nlp.so'*, *compiler_command=None*)

Compile the NLP. This may accelerate the optimization. As compilation is time consuming, the default option is to NOT overwrite (`overwrite=False`) an existing compilation. If an existing compilation with the name `libname` is found, it is used. **This can be dangerous, if the NLP has changed** (user tweaked the cost function, the model etc.).

Warning: This feature is experimental and currently only supported on Linux and MacOS.

What happens here?

1. The NLP is written to a C-file (`cname`)
2. The C-File (`cname`) is compiled. The custom compiler uses:

```
gcc -fPIC -shared -O1 {cname} -o {libname}
```

3. The compiled library is linked to the NLP. This overwrites the original NLP. Options from the previous NLP (e.g. linear solver) are kept.

```
self.S = nlpsol('solver_compiled', 'ipopt', f'{libname}', self.nlpsol_opts)
```

Parameters

- **overwrite** (bool) – If True, the existing compiled NLP will be overwritten.
- **cname** (str) – Name of the C file that will be exported.
- **libname** (str) – Name of the shared library that will be created after compilation.
- **compiler_command** (Optional[str]) – Command to use for compiling. If None, the default compiler command will be used. Please make sure to use matching strings for **libname** when supplying your custom compiler command.

Return type

None

create_nlp

create_nlp(self)

Create the optimization problem. Typically, this method is called internally from [setup\(\)](#).

Users should only call this method if they intend to modify the objective with [nlp_obj](#), the constraints with [nlp_cons](#), [nlp_cons_lb](#) and [nlp_cons_ub](#).

To finish the setup process, users MUST call [create_nlp\(\)](#) afterwards.

Note: Do NOT call [setup\(\)](#) if you intend to go the manual route with [prepare_nlp\(\)](#) and [create_nlp\(\)](#).

Note: Only AFTER calling [prepare_nlp\(\)](#) the previously mentioned attributes [nlp_obj](#), [nlp_cons](#), [nlp_cons_lb](#), [nlp_cons_ub](#) become available.

Returns

None – None

get_p_template

get_p_template(self)

Obtain output template for [set_p_fun\(\)](#). This is used to set the (not estimated) parameters. Use this structure as the return of a user defined parameter function (p_fun) that is called at each MHE step. Pass this function to the MHE by calling [set_p_fun\(\)](#).

Note: The combination of [get_p_template\(\)](#) and [set_p_fun\(\)](#) is identical to the [do_mpc.simulator.Simulator](#) methods, if the MHE is not estimating any parameters.

Returns

Union[SXStruct, MXStruct] – p_template

get_tvp_template

get_tvp_template(self)

Obtain output template for `set_tvp_fun()`.

The method returns a structured object with `n_horizon+1` elements, and a set of time-varying parameters (as defined in `do_mpc.model.Model`) for each of these instances. The structure is initialized with all zeros. Use this object to define values of the time-varying parameters.

This structure (with numerical values) should be used as the output of the `tvp_fun` function which is set to the class with `set_tvp_fun()`. Use the combination of `get_tvp_template()` and `set_tvp_fun()`.

Example:

```
# 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

Union[SXStruct, MXStruct] – Casadi SX or MX structure

get_y_template

get_y_template(self)

Obtain output template for `set_y_fun()`.

Use this structure as the return of a user defined parameter function (`y_fun`) that is called at each MHE step. Pass this function to the MHE by calling `set_y_fun()`.

The structure carries a set of measurements **for each time step of the horizon** and can be accessed as follows:

```
y_template['y_meas', k, 'meas_name']
# Slicing is possible, e.g.:
y_template['y_meas', :, 'meas_name']
```

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

Note: The structure is ordered, such that $k=0$ is the “oldest measurement” and $k=N_horizon$ 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

Union[SXStruct, MXStruct] – `y_template`

make_step

make_step(*self*, *y0*)

Main method of the class during runtime. This method is called at each timestep and returns the current state estimate for the current measurement `y0`.

The method prepares the MHE by setting the current parameters, calls `solve()` and updates the `do_mpc.data.Data` object.

Warning: Moving horizon estimation will only work reliably once a full sequence of measurements corresponding to the set horizon is available.

Parameters

`y0` (ndarray) – Current measurement.

Returns

ndarray – `x0`, estimated state of the system.

prepare_nlp

prepare_nlp(self)

Prepare the optimization problem. Typically, this method is called internally from `setup()`.

Users should only call this method if they intend to modify the objective with `nlp_obj`, the constraints with `nlp_cons`, `nlp_cons_lb` and `nlp_cons_ub`.

To finish the setup process, users MUST call `create_nlp()` afterwards.

Note: Do NOT call `setup()` if you intend to go the manual route with `prepare_nlp()` and `create_nlp()`.

Note: Only AFTER calling `prepare_nlp()` the previously mentioned attributes `nlp_obj`, `nlp_cons`, `nlp_cons_lb`, `nlp_cons_ub` become available.

Returns

None – None

reset_history

reset_history(self)

Reset the history of the optimizer. All data from the `do_mpc.data.Data` instance is removed.

Return type

None

set_default_objective

set_default_objective(self, P_x, P_v=None, P_p=None, P_w=None)

Configure the suggested default MHE formulation.

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

$$\begin{aligned} \min_{\mathbf{x}_{0:N+1}, \mathbf{u}_{0:N}, \mathbf{p}, \mathbf{w}_{0:N}, \mathbf{v}_{0:N}} \quad & m(x_0, \tilde{x}_0, p, \tilde{p}) + \sum_{k=0}^{N-1} l(v_k, w_k, p, p_{\text{tv},k}), \\ \text{s.t.} \quad & \left. \begin{aligned} x_{k+1} &= f(x_k, u_k, z_k, p, p_{\text{tv},k}) + w_k, \\ y_k &= h(x_k, u_k, z_k, p, p_{\text{tv},k}) + v_k, \\ g(x_k, u_k, z_k, p_k, p_{\text{tv},k}) &\leq 0 \end{aligned} \right\} k = 0, \dots, N \end{aligned}$$

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

Pass the weighting matrices P_x , P_p and P_v and P_w . The matrices must be of appropriate dimension and array-like.

Note: It is possible to pass parameters or time-varying parameters defined in the `do_mpc.model.Model` as weighting. You'll probably choose time-varying parameters (`_tvp`) for P_v and P_w and parameters (`_p`) for

P_x and P_p . Use `set_p_fun()` and `set_tvp_fun()` to configure how these values are determined at each time step.

General remarks:

- In the case that no parameters are estimated, the weighting matrix P_p is not required.
- In the case that the `do_mpc.model.Model` is configured without process-noise (see `do_mpc.model.Model.set_rhs()`) the parameter P_w is not required.
- In the case that the `do_mpc.model.Model` is configured without measurement-noise (see `do_mpc.model.Model.set_meas()`) the parameter P_v is not required.

The respective terms are not present in the MHE formulation in that case.

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

Parameters

- **P_x** (Union[ndarray, SX, MX]) – Tuning matrix P_x of dimension $n \times n$ ($x \in \mathbb{R}^n$)
- **P_v** (Union[ndarray, SX, MX, None]) – Tuning matrix P_v of dimension $m \times m$ ($v \in \mathbb{R}^m$)
- **P_p** (Union[ndarray, SX, MX, None]) – Tuning matrix P_p of dimension $l \times l$ ($p_{\text{est}} \in \mathbb{R}^l$)
- **P_w** (Union[ndarray, SX, MX, None]) – Tuning matrix P_w of dimension $k \times k$ ($w \in \mathbb{R}^k$)

Return type

None

set_initial_guess

set_initial_guess(self)

Initial guess for optimization variables. Uses the current class attributes `x0`, `z0` and `u0`, `p_est0` to create an initial guess for the MHE. The initial guess is simply the initial values for all $k = 0, \dots, N$ instances of x_k , u_k and z_k , $p_{\text{est},k}$. :rtype: None

Warning: If no initial values for `x0`, `z0` and `u0` were supplied during setup, these default to zero.

Note: The initial guess is fully customizable by directly setting values on the class attribute: `opt_x_num`.

set_nl_cons

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** (str) – Arbitrary name for the given expression. Names are used for key word indexing.
- **expr** (Union[SX, MX]) – CasADi SX or MX function depending on `_x`, `_u`, `_z`, `_tvp`, `_p`.
- **ub** (float) – Upper bound
- **soft_constraint** (bool) – Flag to enable soft constraint
- **penalty_term_cons** (int) – Penalty term constant
- **maximum_violation** (float) – Maximum violation

Raises

- **assertion** – `expr_name` must be str
- **assertion** – `expr` must be a casadi SX or MX type

Returns

Union[SX, MX] – Returns the newly created expression. Expression can be used e.g. for the RHS.

set_objective

set_objective(*self*, *stage_cost*, *arrival_cost*)

Set the stage cost $l(\cdot)$ and arrival cost $m(\cdot)$ function for the MHE problem:

$$\begin{aligned} \min_{\substack{\mathbf{x}_{0:N+1}, \mathbf{u}_{0:N}, p, \\ \mathbf{w}_{0:N}, \mathbf{v}_{0:N}}} \quad & m(x_0, \tilde{x}_0, p, \tilde{p}) + \sum_{k=0}^{N-1} l(v_k, w_k, p, p_{tv,k}), \\ \text{s.t.} \quad & \left. \begin{aligned} x_{k+1} &= f(x_k, u_k, z_k, p, p_{tv,k}) + w_k, \\ y_k &= h(x_k, u_k, z_k, p, p_{tv,k}) + v_k, \\ g(x_k, u_k, z_k, p_k, p_{tv,k}) &\leq 0 \end{aligned} \right\} k = 0, \dots, N \end{aligned}$$

Use the class attributes:

- `mhe._w` as w_k

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

To formulate the objective function and pass the stage cost and arrival cost independently.

Note: The retrieved attributes are symbolic structures, which can be queried with the given variable names, e.g.:

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

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

Example:

```
# Get variables:
v = mhe._v.cat

stage_cost = v.T@np.diag(np.array([1,1,1,20,20]))@v

x_0 = mhe._x
x_prev = mhe._x_prev
p_0 = mhe._p_est
p_prev = mhe._p_est_prev

dx = x_0.cat - x_prev.cat
dp = p_0.cat - p_prev.cat

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

mhe.set_objective(stage_cost, arrival_cost)
```

Note: Use `set_default_objective()` as a high-level wrapper for this method, if you want to use the default MHE objective function.

Parameters

- **stage_cost** (Union[SX, MX]) – Stage cost that is added to the MHE objective at each age.
- **arrival_cost** (Union[SX, MX]) – Arrival cost that is added to the MHE objective at the initial state.

Return type

None

set_p_fun

set_p_fun(self, p_fun)

Set function which returns parameters.. The `p_fun` is called at each MHE time step and returns the (fixed) parameters. The function must return a numerical CasADi structure, which can be retrieved with `get_p_template()`.

Parameters

`p_fun` (Callable[[float], Union[SXStruct, MXStruct]]) – Parameter function.

Return type

None

set_param

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. :rtype: None

Deprecated since version >v4.5.1: This function will be deprecated in the future

Note: A comprehensive list of all available parameters can be found in `do_mpc.estimator.MHESettings`.

For example:

```
mhe.settings.n_horizon = 20
```

The old interface, as shown in the example below, can still be accessed until further notice.

For example:

```
mhe.set_param(n_horizon = 20)
```

Note: The only required parameters are `n_horizon` and `t_step`. All other parameters are optional.

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**. Any available linear solver can be set using `do_mpc.estimator.MHESettings.set_linear_solver()`. For more details, please check the `do_mpc.estimator.MHESettings`.

Note: The output of IPOPT can be suppressed `do_mpc.estimator.MHESettings.suppress_ipopt_output()`. For more details, please check the `do_mpc.estimator.MHESettings`.

set_tvp_fun

set_tvp_fun(self, tvp_fun)

Set function which returns time-varying parameters.

The `tvp_fun` is called at each optimization step to get the current prediction of the time-varying parameters. The supplied function must be callable with the current time as the only input. Furthermore, the function must return a CasADi structured object which is based on the horizon and on the model definition. The structure can be obtained with `get_tvp_template()`.

Example:

```
# in model definition:
alpha = model.set_variable(var_type='_tvp', var_name='alpha')
beta = model.set_variable(var_type='_tvp', var_name='beta')

...
# in optimizer configuration:
tvp_temp_1 = optimizer.get_tvp_template()
tvp_temp_1['_tvp', :] = np.array([1,1])

tvp_temp_2 = optimizer.get_tvp_template()
tvp_temp_2['_tvp', :] = np.array([0,0])

def tvp_fun(t_now):
    if t_now<10:
        return tvp_temp_1
    else:
        tvp_temp_2

optimizer.set_tvp_fun(tvp_fun)
```

Note: The method `set_tvp_fun()`. must be called prior to setup IF time-varying parameters are defined in the model. It is not required to call the method if no time-varying parameters are defined.

Parameters

tvp_fun (Callable[[float], Union[SXStruct, MXStruct]]) – 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()`).

Return type

None

set_y_fun

set_y_fun(self, y_fun)

Set the measurement function. The function must return a CasADi structure which can be obtained from [get_y_template\(\)](#). See the respective doc string for details.

Parameters

y_fun (Callable[[float], Union[SXStruct, MXStruct]]) – measurement function.

Return type

None

setup

setup(self)

The setup method finalizes the MHE creation. The optimization problem is created based on the configuration of the module. :rtype: None

Note: After this call, the [solve\(\)](#) and [make_step\(\)](#) method is applicable.

solve

solve(self)

Solves the optimization problem.

The current problem is defined by the parameters in the [opt_p_num](#) CasADi structured Data.

Typically, [opt_p_num](#) is prepared for the current iteration in the [make_step\(\)](#) method. It is, however, valid and possible to directly set parameters in [opt_p_num](#) before calling [solve\(\)](#).

The method updates the [opt_p_num](#) and [opt_x_num](#) attributes of the class. By resetting [opt_x_num](#) to the current solution, the method implicitly enables **warmstarting the optimizer** for the next iteration, since this vector is always used as the initial guess. :rtype: None

Warning: The method is part of the public API but it is generally not advised to use it. Instead we recommend to call [make_step\(\)](#) at each iterations, which acts as a wrapper for [solve\(\)](#).

Raises

AssertionError – Optimizer was not setup yet.

4.10.4.3.2 Attributes

bounds

MHE.bounds

Query and set bounds of the optimization variables. The [bounds\(\)](#) method is an indexed property, meaning getting and setting this property requires an index and calls this function. The power index (elements are separated by commas) must contain 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.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']
```

lb_opt_x

MHE.lb_opt_x

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

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

Note: The attribute automatically considers the scaling variables when setting the bounds. See `scaling` for more details.

Note: Modifications must be done after calling `prepare_nlp()` or `setup()` respectively.

nlp_cons

MHE.nlp_cons

Query and modify (symbolically) the NLP constraints. Use the variables in `opt_x` and `opt_p`.

Prior to calling `create_nlp()` this attribute returns a list of symbolic constraints. After calling `create_nlp()` this attribute returns the concatenation of this list and the attribute cannot be altered anymore.

It is advised to append to the current list of `nlp_cons`:

```
mpc.prepare_nlp()

# Create new constraint: Input at timestep 0 and 1 must be identical.
extra_cons = mpc.opt_x['_u', 0, 0] - mpc.opt_x['_u', 1, 0]
mpc.nlp_cons.append(
    extra_cons
)
```

(continues on next page)

(continued from previous page)

```
# Create appropriate upper and lower bound (here they are both 0 to create an
→equality constraint)
mpc.nlp_cons_lb.append(np.zeros(extra_cons.shape))
mpc.nlp_cons_ub.append(np.zeros(extra_cons.shape))

mpc.create_nlp()
```

See the documentation of *opt_x* and *opt_p* on how to query these attributes.

Warning: This is a VERY low level feature and should be used with extreme caution. It is easy to break the code.

Be especially careful NOT to accidentally overwrite the default objective.

Note: Modifications must be done after calling *prepare_nlp()* and before calling *create_nlp()*

nlp_cons_lb

MHE.nlp_cons_lb

Query and modify the lower bounds of the *nlp_cons*.

Prior to calling *create_nlp()* this attribute returns a list of lower bounds matching the list of constraints obtained with *nlp_cons*. After calling *create_nlp()* this attribute returns the concatenation of this list.

Values for lower (and upper) bounds MUST be added when adding new constraints to *nlp_cons*.

Warning: This is a VERY low level feature and should be used with extreme caution. It is easy to break the code.

Note: Modifications must be done after calling *prepare_nlp()*

nlp_cons_ub

MHE.nlp_cons_ub

Query and modify the upper bounds of the *nlp_cons*.

Prior to calling *create_nlp()* this attribute returns a list of upper bounds matching the list of constraints obtained with *nlp_cons*. After calling *create_nlp()* this attribute returns the concatenation of this list.

Values for upper (and lower) bounds MUST be added when adding new constraints to *nlp_cons*.

Warning: This is a VERY low level feature and should be used with extreme caution. It is easy to break the code.

Note: Modifications must be done after calling `prepare_nlp()`

nlp_obj

MHE.nlp_obj

Query and modify (symbolically) the NLP objective function. Use the variables in `opt_x` and `opt_p`.

It is advised to add to the current objective, e.g.:

```
mpc.prepare_nlp()
# Modify the objective
mpc.nlp_obj += sum1(vercat(*mpc.opt_x['_x', -1, 0])**2)
# Finish creating the NLP
mpc.create_nlp()
```

See the documentation of `opt_x` and `opt_p` on how to query these attributes.

Warning: This is a VERY low level feature and should be used with extreme caution. It is easy to break the code.

Be especially careful NOT to accidentally overwrite the default objective.

Note: Modifications must be done after calling `prepare_nlp()` and before calling `create_nlp()`

opt_p

MHE.opt_p

Full structure of (symbolic) MHE parameters.

The attribute can be used to alter the objective function or constraints of the NLP.

The attribute is a CasADi numeric structure with nested power indices. It can be indexed as follows:

```
# previously estimated state:
opt_p['_x_prev', _x_name]
# previously estimated parameters:
opt_p['_p_est_prev', _x_name]
# known parameters
opt_p['_p_set', _p_name]
# time-varying parameters:
opt_p['_tvp', time_step, _tvp_name]
# sequence of measurements:
opt_p['_y_meas', time_step, _y_name]
```

The names refer to those given in the `do_mpc.model.Model` configuration. Further indices are possible, if the variables are itself vectors or matrices.

Warning: Do not tweak or overwrite this attribute unless you know what you are doing.

Note: The attribute is populated when calling `setup()` or `create_nlp()`.

opt_p_num

MHE.opt_p_num

Full MHE parameter vector.

This attribute is used when calling the solver to pass all required parameters, including

- previously estimated state(s)
- previously estimated parameter(s)
- known parameters
- sequence of time-varying parameters
- sequence of measurements parameters

do-mpc handles setting these parameters automatically in the `make_step()` method. However, you can set these values manually and directly call `solve()`.

The attribute is a CasADi numeric structure with nested power indices. It can be indexed as follows:

```
# previously estimated state:
opt_p_num['_x_prev', _x_name]
# previously estimated parameters:
opt_p_num['_p_est_prev', _x_name]
# known parameters
opt_p_num['_p_set', _p_name]
# time-varying parameters:
opt_p_num['_tvp', time_step, _tvp_name]
# sequence of measurements:
opt_p_num['_y_meas', time_step, _y_name]
```

The names refer to those given in the `do_mpc.model.Model` configuration. Further indices are possible, if the variables are itself vectors or matrices.

Warning: Do not tweak or overwrite this attribute unless you know what you are doing.

Note: The attribute is populated when calling `setup()`

opt_x

MHE.opt_x

Full structure of the (symbolic) MHE optimization variables.

The attribute is a CasADi numeric structure with nested power indices. It can be indexed as follows:

```
# dynamic states:
opt_x['_x', time_step, collocation_point, _x_name]
# algebraic states:
opt_x['_z', time_step, collocation_point, _z_name]
# inputs:
opt_x['_u', time_step, _u_name]
# estimated parameters:
opt_x_Num['_p_est', _p_names]
# slack variables for soft constraints:
opt_x['_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 alter the objective function or constraints of the NLP.

Note: The attribute `opt_x` carries the scaled values of all variables.

Warning: Do not tweak or overwrite this attribute unless you know what you are doing.

Note: The attribute is populated when calling `setup()` or `prepare_nlp()`

opt_x_num

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 `setup()`

p_est0

MHE.p_est0

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

Example:

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

# Initiate MHE with list of estimated parameters:
mhe = do_mpc.estimator.MHE(model, ['temperature'])

# Get or set current value of variable:
mhe.p_est0['temperature', 0] # 0th element of variable
mhe.p_est0['temperature']    # all elements of variable
mhe.p_est0['temperature', 0:2] # 0th and 1st element
```

Usefull CasADi symbolic structure methods:

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

scaling

MHE.scaling

Query and set scaling of the optimization variables. The `Optimizer.scaling()` method is an indexed property, meaning getting and setting this property requires an index and calls this function. The power index (elements are separated by comas) 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.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']
```

Scaling factors a affect the MHE / MPC optimization problem. The optimization variables are scaled variables:

$$\bar{\phi} = \frac{\phi}{a_{\phi}} \quad \forall \phi \in [x, u, z, p_{\text{est}}]$$

Scaled variables are used to formulate the bounds $\bar{\phi}_{lb} \leq \bar{\phi}_{ub}$ and for the evaluation of the ODE. For the objective function and the nonlinear constraints the unscaled variables are used. The algebraic equations are also not scaled.

Note: Scaling the optimization problem is suggested when states and / or inputs take on values which differ by orders of magnitude.

t0

MHE.t0

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

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

u0

MHE.u0

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

Example:

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

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

(continues on next page)

(continued from previous page)

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

Useful CasADi symbolic structure methods:

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

ub_opt_x

MHE.ub_opt_x

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

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

Note: The attribute automatically considers the scaling variables when setting the bounds. See `scaling` for more details.

Note: Modifications must be done after calling `prepare_nlp()` or `setup()` respectively.

x0

MHE.x0

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

Example:

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

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

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

Useful CasADi symbolic structure methods:

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

`z0`

`MHE.z0`

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

Example:

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

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

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

Useful CasADi symbolic structure methods:

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

4.10.4.4 MHESettings

```
class MHESettings(n_horizon=None, t_step=None, meas_from_data=False, state_discretization='collocation',
                  collocation_type='radau', collocation_deg=2, collocation_ni=1,
                  nl_cons_check_colloc_points=False, nl_cons_single_slack=False,
                  cons_check_colloc_points=True, store_full_solution=False, store_lagr_multiplier=True,
                  store_solver_stats=<factory>, nlpsol_opts=<factory>)
```

Bases: `EstimatorSettings`

Settings for `do_mpc.estimator.MHE`.

The `do_mpc.estimator.MHE` automatically creates an instance of type `MHESettings` and adds it to its class attributes.

Example to change settings:

```
mhe.settings.n_horizon = 20
```

Note: Settings cannot be updated after calling `do_mpc.estimator.MHE.setup()`.

Parameters

- **n_horizon** (Optional[int]) –
- **t_step** (Optional[float]) –
- **meas_from_data** (bool) –
- **state_discretization** (str) –
- **collocation_type** (str) –
- **collocation_deg** (int) –
- **collocation_ni** (int) –
- **nl_cons_check_colloc_points** (bool) –
- **nl_cons_single_slack** (bool) –
- **cons_check_colloc_points** (bool) –
- **store_full_solution** (bool) –
- **store_lagr_multiplier** (bool) –
- **store_solver_stats** (List[str]) –
- **nlpsol_opts** (Dict) –

4.10.4.4.1 Methods**check_for_mandatory_settings****check_for_mandatory_settings**(*self*)

Method to assert the necessary settings required to design *do_mpc.estimator.MHE*

set_linear_solver**set_linear_solver**(*self*, *solver_name*='MA27')

Method to set the linear solver to MA27.

This method enables to set the linear solver to MA27. This change in many cases will drastically boost the speed of do-mpc.

Example:

```
mhe.settings.set_linear_solver(solver_name = "MA27")
```

Parameters

- **solver_name** (str) – Specify the solver name.

supress_ipopt_output

supress_ipopt_output(*self*)

Method to suppress the ipopt solver output.

This method set the relevant settings in the ipopt solver in order to suppress the output log.

4.10.4.4.2 Attributes

collocation_deg

MHESettings.collocation_deg: int = 2

Choose the collocation degree for continuous models with collocation as state discretization.

collocation_ni

MHESettings.collocation_ni: int = 1

For orthogonal collocation, choose the number of finite elements for the states within a time-step (and during constant control input).

Note: Can be used to avoid high-order polynomials.

collocation_type

MHESettings.collocation_type: str = 'radau'

Choose the collocation type for continuous models with collocation as state discretization.

Note: Currently only 'radau' collocation type is available.

cons_check_colloc_points

MHESettings.cons_check_colloc_points: bool = True

For orthogonal collocation choose whether the linear bounds set with *do_mpc.estimator.MHE.bounds* are evaluated once per finite Element or for each collocation point.

meas_from_data

MHESettings.meas_from_data: bool = False

Default option to retrieve past measurements for the MHE optimization problem.

The *do_mpc.estimator.MHE.set_y_fun()* is called during setup.

n_horizon

MHESettings.n_horizon: int = None

Prediction horizon of the optimal control problem.

Parameter must be set by user.

nl_cons_check_colloc_points

MHESettings.nl_cons_check_colloc_points: bool = False

For orthogonal collocation choose whether the bounds set with `do_mpc.estimated.MHE.set_nl_cons()` are evaluated once per finite Element or for each collocation point.

nl_cons_single_slack

MHESettings.nl_cons_single_slack: bool = False

If True, soft-constraints set with `do_mpc.estimated.MHE.set_nl_cons()` introduce only a single slack variable for the entire horizon.

state_discretization

MHESettings.state_discretization: str = 'collocation'

Choose the state discretization for continuous models.

Currently only 'collocation' is available.

Note: This attribute has no effect if model is created in discrete type.

store_full_solution

MHESettings.store_full_solution: bool = False

Choose whether to store the full solution of the optimization problem.

This is required for animating the predictions in post processing.

Warning: It drastically increases the required storage.

store_lagr_multiplier

MHESettings.store_lagr_multiplier: bool = True

Choose whether to store the lagrange multipliers of the optimization problem.

Warning: Increases the required storage.

t_step

MHESettings.t_step: float = None

Timestep of the estimator.

store_solver_stats

MHESettings.store_solver_stats: List[str]

Choose which solver statistics to store.

Must be a list of valid statistics.

nlpsol_opts

MHESettings.nlpsol_opts: Dict

Dictionary with options for the CasADi solver call nlpsol with plugin ipopt.

All options are listed [here](#).

4.10.4.5 StateFeedback

class StateFeedback(model)

Bases: *Estimator*

Simple state-feedback “estimator”. The main method *StateFeedback.make_step()* simply returns the input. Why do you even bother to use this class?

4.10.4.5.1 Methods

make_step

make_step(self, y0)

Returns the measurement.

Parameters

y0 (ndarray) – measurment

Returns

ndarray – Return the measurement y0.

reset_history

reset_history(self)

Reset the history of the estimator

Return type

None

4.10.4.5.2 Attributes

t0

StateFeedback.t0

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

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

u0

StateFeedback.u0

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

Example:

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

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

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

Useful CasADi symbolic structure methods:

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

x0

StateFeedback.x0

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

Example:

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

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

(continues on next page)

(continued from previous page)

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

Useful CasADi symbolic structure methods:

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

z0

StateFeedback.z0

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

Example:

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

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

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

Useful CasADi symbolic structure methods:

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

4.10.5 graphics

Visualization tools for do-mpc.

Functions

<code>default_plot</code>	Pass a <code>do_mpc.data.Data</code> object and create a default do-mpc plot.
<code>animate</code>	Animation helper function.

4.10.5.1 default_plot

Class method.

default_plot(*data*, *states_list*=None, *dae_states_list*=None, *inputs_list*=None, *aux_list*=None, ****kwargs**)

Pass a `do_mpc.data.Data` object and create a default **do-mpc** plot. By default all states, inputs and auxiliary expressions are plotted on individual axes. Pass lists of states, inputs and aux names (string) to plot only a subset of these trajectories.

Returns a figure, axis and configured `Graphics` object.

Parameters

- **data** – **do-mpc** data instance.
- **states_list** (Optional[list]) – List of strings containing a subset of state names defined in `py:class:do_mpc.model.Model`. These states are plotted.
- **dae_states_list** (Optional[list]) – List of strings containing a subset of dae states (`_z`) names defined in `py:class:do_mpc.model.Model`. These states are plotted.
- **inputs_list** (Optional[list]) – List of strings containing a subset of input names defined in `py:class:do_mpc.model.Model`. These inputs are plotted.
- **aux_list** (Optional[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

Tuple[figure, axes, `Graphics`] – Matplotlib fig and ax and configured `Graphics` object.

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

4.10.5.2 animate

Class method.

animate(*graphics*, *fig*, *n_steps*=None, *export_path*='./', *export_name*='animation', *overwrite*=False, *format*='gif', *fps*=5, *writer*=None)

Animation helper function.

Call this function with a configured `Graphics` instance and the respective figure. This function will export an animation with the results from the `do_mpc.data.Data` object.

Either specify format and fps or supply a configured writer (e.g. `ImageMagickWriter` for gifs).

Parameters

- **graphics** (`Graphics`) – Configured `Graphics` instance.
- **fig** (figure) – Matplotlib Figure.

- **n_steps** (Optional[int]) – number of time steps for the animation.
- **export_path** (str) – Path where to export the animation. Directory will be created if it doesn't exist.
- **export_name** (str) – Name of the resulting animation (gif/mp4) file.
- **overwrite** (bool) – Check if export_name already exists in the supplied directory and overwrite or alter export_name.
- **format** (str) – Choose between gif or mp4.
- **fps** (int) – Frames per second for the resulting animation.
- **writer** (Union[FFMpegWriter, ImageMagickWriter, None]) – If supplied, the fps and format argument are discarded. Use this to configure your own writer.

Return type

None

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

Classes*Graphics*Graphics module to present the results of **do-mpc**.**4.10.5.3 Graphics****class Graphics**(data)

Bases: object

Graphics module to present the results of **do-mpc**. The module is independent of all other modules and can be used optionally. The module can also be used with pickled result files in post-processing for flexible and custom graphics.

The graphics module is based on Matplotlib and allows for fully customizable, publication ready graphics and animations.

The Graphics module is initialized with an *do_mpc.data.Data* or *do_mpc.data.MPCData* module and will showcase this data.

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

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

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

# Create figure with arbitrary Matplotlib method
fig, ax = plt.subplots(5, sharex=True)
# Configure plot (pass the previously obtained ax objects):
graphics.add_line(var_type='_x', var_name='C_a', axis=ax[0])
graphics.add_line(var_type='_x', var_name='C_b', axis=ax[0])
graphics.add_line(var_type='_x', var_name='T_R', axis=ax[1])
graphics.add_line(var_type='_x', var_name='T_K', axis=ax[1])
graphics.add_line(var_type='_aux', var_name='T_dif', axis=ax[2])
```

(continues on next page)

(continued from previous page)

```

graphics.add_line(var_type='_u', var_name='Q_dot', axis=ax[3])
graphics.add_line(var_type='_u', var_name='F', axis=ax[4])
# Optional configuration of the plot(s) with matplotlib:
ax[0].set_ylabel('c [mol/l]')
ax[1].set_ylabel('Temperature [K]')
ax[2].set_ylabel('\Delta T [K]')
ax[3].set_ylabel('Q_heat [kW]')
ax[4].set_ylabel('Flow [l/h]')

fig.align_ylabels()

```

After initializing the `Graphics` module, the `Graphics.add_line()` method is used to define which results are to be plotted on which existing axes object. The method created (empty) line objects for each plotted variable. The graphic is updated with the most recent data with `Graphics.plot_results()`. Furthermore, the module contains the `Graphics.plot_predictions()` method which is applicable only for `do_mpc.data.MPCData`, and can be used to show the predicted trajectories.

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

Animations can be setup with the following loop:

```

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

    graphics.plot_results()
    graphics.plot_predictions()
    graphics.reset_axes()
    plt.show()
    plt.pause(0.01)

```

Parameters

data (Union[`Data`, `MPCData`]) – Data object from the **do-mpc** modules (simulator, estimator, controller)

4.10.5.3.1 Methods

add_line

add_line(*self*, *var_type*, *var_name*, *axis*, ***pltkwargs*)

`add_line` is called during setting up the `Graphics` class. This is typically the last step of configuring **do-mpc**. Each call of `Graphics.add_line()` adds a line to the passed axis according to the variable type (`_x`, `_u`, `_z`, `_tvp`, `_p`, `_aux`) and its name (as defined in the `do_mpc.model.Model`). Furthermore, all valid matplotlib `.plot` arguments can be passed as optional keyword arguments, e.g.: `linewidth`, `color`, `alpha`.

Note: Lines can also be configured after adding them with this method. Use the `result_lines()` and

`pred_lines()` attributes for this purpose.

Parameters

- **var_type** (str) – Variable type to be plotted. Valid arguments are `_x`, `_u`, `_z`, `_tvp`, `_p`, `_aux`.
- **var_name** (str) – Variable name. Must reference the names defined in the model for the given variable type.
- **axis** (Axes) – Axis object on which to plot the line(s).
- **pltkwargs** – 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.

Return type

None

clear

`clear(self, lines=None)`

Clears all data from lines.

Parameters

lines (Optional[list]) –

Return type

None

plot_predictions

`plot_predictions(self, t_ind=-1)`

Plots the predicted trajectories for the plot configuration. The predicted trajectories are part of the optimal solution at each timestep and are **optionally** stored in the `do_mpc.data.MPCData` object.

Warning: This method requires that the optimal solution is stored in the `do_mpc.data.MPCData` instance. Storing the optimal solution must be activated with `do_mpc.controller.MPC.set_param()`.

The `plot_predictions` method can only be called with data from the `do_mpc.controller.MPC` object and raises an error if called with data from other objects. Use the `t_ind` parameter to plot the prediction for the given time instance. This can be used in post-processing for animations.

Parameters

t_ind (int) – Plot predictions at this time index.

Raises

- **assertion** – Can only call `plot_predictions` with data object from do-mpc optimizer

- **Exception** – Cannot plot predictions if full solution is not stored or supplied when calling the method
- **assertion** – `t_ind` argument must be a int
- **assertion** – `t_ind` argument must not exceed the length of the results

Return type

None

plot_results**plot_results**(*self*, *t_ind=-1*)

Plots the results stored in the data object. Use the `t_ind` parameter to plot only until the given time index. This can be used in post-processing for animations.

Parameters

t_ind (int) – Plot results up until this time index.

Raises

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

Return type

None

reset_axes**reset_axes**(*self*)

Relimits and scales all axes. This method calls

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

on all axes instances in the class.

Return type

None

reset_prop_cycle**reset_prop_cycle**(*self*)

Resets the property cycle for all axes which were passed with `Graphics.add_line()`. The matplotlib color cycler is restarted.

Return type

None

4.10.5.3.2 Attributes

pred_lines

Graphics.pred_lines

Structure that holds the prediction line objects. Query this structure with power indices. The power indices must have the following order:

```
pred_lines[var_type, var_name, i, k]
```

where

- `var_type` refers to `_x`, `_u`, `_z`, `_tvp`, `_p`, `_aux`
- `var_name` refers to the user-defined names in the `do_mpc.model.Model`
- Use `i` to index vector valued variables (choose 0 for scalars).
- Use `k` to select the `k`-th scenario (for robust MPC). Note the `k=0` is the nominal case.

Note that (e.g.) `pred_lines['_x']` will return all lines for all states and `pred_lines.full` can be used to retrieve all line objects.

This property can be used to query and configure specific lines in the current graphic.

Example:

```
# Update properties for all lines:
for line_i in graphics.pred_lines.full:
    line_i.set_linewidth(2)
    line_i.set_alpha(0.5)
```

An extensive list of all line properties can be found [here](#).

Parameters

powerind (Optional[tuple]) – Tuple of indices (power indices) to obtain the desired line objects

Returns

List of line objects.

result_lines

Graphics.result_lines

Structure that holds the result line objects. Query this structure with power indices. The power indices must have the following order:

```
result_lines[var_type, var_name, i]
```

where

- `var_type` refers to `_x`, `_u`, `_z`, `_tvp`, `_p`, `_aux`
- `var_name` refers to the user-defined names in the `do_mpc.model.Model`
- Index `i` is applicable if the selecte variable is vector valued.

Note that (e.g.) `result_lines['_x']` will return all lines for all states and `result_lines.full` can be used to retrieve all line objects.

This property can be used to query and configure specific lines in the current graphic.

Example:

```
# Update properties for all lines:
for line_i in graphics.result_lines.full:
    line_i.set_linewidth(2)
    line_i.set_alpha(0.5)
```

An extensive list of all line properties can be found [here](#).

Parameters

powerind (Optional[tuple]) – Tuple of indices (power indices) to obtain the desired line objects

Returns

List of line objects.

4.10.6 model

Dynamic modelling with do-mpc. The basis for all other classes.

Functions

<i>dae2odeconversion</i>	Converts index-1 DAE system to ODE system.
<i>linearize</i>	Linearize the non-linear <i>Model</i> to obtain a <i>LinearModel</i> .

4.10.6.1 dae2odeconversion

Class method.

dae2odeconversion(model)

Converts index-1 DAE system to ODE system.

This method utilizes the differentiation method of converting index-1 DAE systems to ODE systems. This method cannot handle higher index DAE systems. The DAE system is as follows:

$$\begin{aligned}\dot{x} &= f(x, u, z) \\ 0 &= g(x, u, z)\end{aligned}$$

where x is the states, u is the input and z is the algebraic states of the system. Differentiation method is as follows:

$$\dot{z} = -\frac{\partial g^{-1}}{\partial z} \frac{\partial g}{\partial x} f - \frac{\partial g^{-1}}{\partial z} \frac{\partial g}{\partial u} \dot{u}$$

Therefore the converted ODE system looks like:

$$\begin{pmatrix} \dot{x} \\ \dot{u} \\ \dot{z} \end{pmatrix} = \begin{pmatrix} f(x, u, z) \\ q \\ g(x, u, z) \end{pmatrix}$$

where \dot{x} , \dot{u} , \dot{z} are the states of the model and q is the input to the model. Similarly, it can be extended to discrete time systems. The dae to ode converted model assumes that converted algebraic states and states measurements are available.

Parameters

model (*Model*) – Index-1 DAE model

Returns

Model – Converted ODE Model

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

4.10.6.2 linearize

Class method.

linearize(*model*, *xss=None*, *uss=None*, *tv0=None*, *p0=None*)

Linearize the non-linear *Model* to obtain a *LinearModel*. The linearized model is required, e.g. for the *do_mpc.controller.LQR* controller.

This method uses the taylor expansion series to linearize non-linear model to linear model at the specified set points. Linearized model retains the same variable names for all states, inputs with respect to the original model. The non-linear model equation this method can solve is as follows:

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

The above model is linearized around steady state set point x_{ss} and steady state input u_{ss}

$$\begin{aligned}\frac{\partial f}{\partial x}|_{x_{ss}} &= 0 \\ \frac{\partial f}{\partial u}|_{u_{ss}} &= 0\end{aligned}$$

The linearized model is as follows:

$$\Delta\dot{x} = A\Delta x + B\Delta u$$

Similarly, it can be extended to discrete time systems. Since the linearized model has only rate of change input and state. The names are appended with 'del' to differentiate from the original model. This can be seen in the above model definition. Therefore, the solution of the lqr will be \mathbf{u} and its corresponding \mathbf{x} . In order to fetch $\Delta\mathbf{u}$ and $\Delta\mathbf{x}$, setpoints has to be subtracted from the solution of lqr.

Parameters

- **model** (*Model*) – dynamic systems model
- **xss** (Optional[ndarray]) – Steady state state
- **uss** (Optional[ndarray]) – Steady state input
- **tv0** (Optional[ndarray]) – value for tvp variable
- **p0** (Optional[ndarray]) – value for parameter variable

Returns

LinearModel – Linearized Model

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

Classes

<i>IteratedVariables</i>	Class to initiate properties and attributes for iterated variables.
<i>LinearModel</i>	The do-mpc LinearModel class.
<i>Model</i>	The do-mpc model class.

4.10.6.3 IteratedVariables

class IteratedVariables

Bases: object

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

Warning: This base class can not be used independently.

4.10.6.3.1 Methods

4.10.6.3.2 Attributes

t0

IteratedVariables.t0

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

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

u0

IteratedVariables.u0

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

Example:

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

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

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

Useful CasADi symbolic structure methods:

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

x0

IteratedVariables.x0

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

Example:

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

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

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

Useful CasADi symbolic structure methods:

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

z0

IteratedVariables.z0

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

Example:

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

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

# Get or set current value of variable:
mpc.z0['temperature', 0] # 0th element of variable
```

(continues on next page)

(continued from previous page)

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

Useful CasADi symbolic structure methods:

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

4.10.6.4 LinearModel

class LinearModel(*model_type=None*, *symvar_type='SX'*)

Bases: `Model`

The **do-mpc** LinearModel class. This class is inherited from **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`, `do_mpc.controller.LQR` and `do_mpc.estimator.Estimator`. This class can be used to define the linear time invariant models in both continuous and discrete time. The `LinearModel` class is created with setting the `model_type` (continuous or discrete).

A continuous linear model consists of an underlying ordinary differential equation (ODE)

$$\begin{aligned}\dot{x}(t) &= Ax(t) + Bu(t), \\ y &= Cx(t) + Du(t)\end{aligned}$$

whereas a discrete linear model consists of a difference equation.

$$\begin{aligned}x_{k+1} &= Ax_k + Bu_k, \\ y_k &= Cx_k + Du_k\end{aligned}$$

The **do-mpc** linear model can be initiated with SX variable type.

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

Configuration and setup: Configuring and setting up the `LinearModel` involves the following steps:

Model can be setup in two different ways. The first method is as follows:

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

The second method is as follows:

1. Use `set_variable()` to introduce new variables to the linear model.
2. Optionally introduce “auxiliary” expressions as functions of the previously defined variables with `set_expression()`. The expressions can be used for monitoring or be reused as constraints, the cost function etc.
3. Call `setup()` and pass the system dynamics matrices as arguments instead of setting up the right hand side equations and measurement equations to finalize the `LinearModel`. No further changes are possible afterwards.

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

Parameters

- **model_type** (Optional[str]) – Set if the model is discrete or continuous.
- **symvar_type** (str) – Set if the model is configured with CasADi SX variables (default).

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']
```

4.10.6.4.1 Methods

discretize

discretize(self, t_step=0, conv_method='zoh')

Converts continuous time to discrete time system.

This method utilizes the existing function in scipy library called `cont2discrete` to convert continuous time to discrete time system. This method allows the user to specify the type of discretization. For more details about the function [click here](#).

where A_{discrete} and B_{discrete} are the discrete state matrix and input matrix respectively and t_{sample} is the sampling time.

Warning: sampling time is zero when not specified or not required

Parameters

- **t_step** (Union[float, int]) – Sampling time (default - 0)
- **conv_method** (str) – Method of discretization - Five different methods can be applied. (default - 'zoh')

Returns

LinearModel – Discretized linear model

get_linear_system_matrices

get_linear_system_matrices(self, xss=None, uss=None, z=None, tvp=None, p=None)

Returns the matrix quadruple (A, B, C, D) of the linearized system around the operating point ($x_{ss}, u_{ss}, z, tvp, p, w, v$). All arguments are optional in which case the matrices might still be symbolic. If the matrices are not symbolic, they are returned as numpy arrays.

Parameters

- **xss** (Optional[ndarray]) – Steady state state
- **uss** (Optional[ndarray]) – Steady state input
- **z** (Optional[ndarray]) – Steady state algebraic states
- **tvp** (Optional[ndarray]) – time varying parameters set point
- **p** (Optional[ndarray]) – parameters set point

Returns

Union[Tuple[SX, SX, SX, SX], Tuple[ndarray, ndarray, ndarray, ndarray]] – State matrix, Input matrix, Output matrix, Feedforward matrix

get_steady_state

get_steady_state(self, xss=None, uss=None)

Calculates steady states for the given input or states.

This method calculates steady states of a discrete system for the given steady state input and vice versa. The mathematical formulation can be described as:

$$x_{ss} = (I - A)^{-1} B u_{ss}$$

or

$$u_{ss} = B^{-1} (I - A) x_{ss}$$

Parameters

- **xss** (Optional[ndarray]) – Steady state State values
- **uss** (Optional[ndarray]) – Steady state Input values

Returns

ndarray – Steady state state or Steady state input

set_alg

set_alg(*self*, *expr_name*, *expr*, **args*, ***kwargs*)

Warning: This method is not supported for linear models.

set_expression

set_expression(*self*, *expr_name*, *expr*)

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

Example:

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

```
Introduce two scalar states:
x_1 = model.set_variable('_x', 'x_1')
x_2 = model.set_variable('_x', 'x_2')

# Introduce expression:
model.set_expression('x1x2', x_1*x_2)
```

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

Parameters

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

Raises

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

Returns

Union[SX, MX] – Returns the newly created expression. Expression can be used e.g. for the RHS.

set_meas

set_meas(*self*, *name*, *meas*)

Checks if the measurement function is linear and calls `Model.set_meas()`.

Parameters

- **name** (str) – Arbitrary name for the given expression. Names are used for key word indexing.
- **meas** (SX) – CasADi SX function depending on `_x`, `_u`, `_tvp`, `_p`.

Return type

None

set_rhs**set_rhs**(self, name, rhs)Checks if the right-hand-side function is linear and calls `Model.set_rhs()`.**Parameters**

- **name** (str) – Reference to previously introduced state names (with `LinearModel.set_variable()`)
- **rhs** (SX) – CasADi SX function depending on `_x`, `_u`, `_tvp`, `_p`.

Return type

None

set_variable**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** (str) – Declare the type of the variable.
- **var_name** (str) – Set a user-defined name for the parameter. The names are reused throughout `do_mpc`.
- **shape** (Union[int, Tuple]) – Shape of the current variable (optional), defaults to 1.

The following types of **var_type** are valid (long or short name is possible):

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

Raises

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

Returns

Union[SX, MX] – Returns the newly created symbolic variable.

setup

setup(*self*, *A=None*, *B=None*, *C=None*, *D=None*)

Setup method must be called to finalize the modelling process. All required model variables must be declared. The right hand side expression for `_x` can be set with `set_rhs()` or can be set by passing the state matrix and input matrix in `setup()`.

Sets default measurement function (state feedback) if `set_meas()` was not called or output matrix, feedforward matrix are not passed in `setup()`.

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

Raises

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

Parameters

- **A** (Optional[ndarray]) – State matrix (optional)
- **B** (Optional[ndarray]) – Input matrix (optional)
- **C** (Optional[ndarray]) – Output matrix (optional)
- **D** (Optional[ndarray]) – Feedforward matrix (optional)

Return type

None

4.10.6.4.2 Attributes**aux**

`LinearModel.aux`

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

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

Example:

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

Useful CasADi symbolic structure methods:

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

Raises

assertion – Cannot set aux directly Use set_expression instead.

p

LinearModel.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
```

Useful CasADi symbolic structure methods:

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

Raises

assertion – Cannot set model variables directly Use set_variable instead.

sys_A

LinearModel.sys_A

State matrix. This property provides the state matrix in the numerical array format. Accessible only after model is setup.

sys_B

LinearModel.sys_B

Input matrix. This property provides the input matrix in the numerical array format. Accessible only after model is setup.

sys_C

LinearModel.sys_C

Output matrix. This property provides the output matrix in the numerical array format. Accessible only after model is setup.

sys_D

LinearModel.sys_D

Feedforward matrix. This property provides the feedforward matrix in the numerical array format. Accessible only after model is setup.

tvp

LinearModel.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
```

Useful CasADi symbolic structure methods:

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

Raises

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

u**LinearModel.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
```

Useful CasADi symbolic structure methods:

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

Raises

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

v**LinearModel.v**

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

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

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

Useful CasADi symbolic structure methods:

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

Raises

assertion – Cannot set v directly

w**LinearModel.w**

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

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

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

Useful CasADi symbolic structure methods:

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

Raises

assertion – Cannot set w directly

x**LinearModel.x**

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

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

Example:

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

Useful CasADi symbolic structure methods:

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

Raises

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

y

LinearModel.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
```

Useful CasADi symbolic structure methods:

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

Raises

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

z

LinearModel.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
```

Useful CasADi symbolic structure methods:

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

Raises

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

4.10.6.5 Model

class Model(*model_type=None*, *symvar_type='SX'*)

Bases: object

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):

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

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

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

The **do-mpc** model can be initiated with either **SX** or **MX** variable type. We refer to the CasADi documentation on the difference of these two types.

Note: **SX** vs. **MX** in a nutshell: In general use **SX** variables (default). If your model consists of scalar operations **SX** variables will be beneficial. Your implementation will most likely only benefit from **MX** variables if you use large(r)-scale matrix-vector multiplications.

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

Configuration and setup:

Configuring and setting up the *Model* involves the following steps:

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

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

Parameters

- **model_type** (Optional[str]) – Set if the model is discrete or continuous.
- **symvar_type** (str) – Set if the model is configured with CasADi SX or MX variables.

Raises

- **assertion** – model_type must be string
- **assertion** – model_type must be either discrete or continuous

__getitem__(ind)

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

```
# 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']
```

4.10.6.5.1 Methods

get_linear_system_matrices

get_linear_system_matrices(self, xss=None, uss=None, z=None, tvp=None, p=None)

Returns the matrix quadrupel (A, B, C, D) of the linearized system around the operating point ($xss, uss, z, tvp, p, w, v$). All arguments are optional in which case the matrices might still be symbolic. If the matrices are not symbolic, they are returned as numpy arrays.

Parameters

- **xss** (Optional[ndarray]) – Steady state state
- **uss** (Optional[ndarray]) – Steady state input
- **z** (Optional[ndarray]) – Steady state algebraic states
- **tvp** (Optional[ndarray]) – time varying parameters set point
- **p** (Optional[ndarray]) – parameters set point

Returns

Union[Tuple[SX, SX, SX, SX], Tuple[ndarray, ndarray, ndarray, ndarray]] – State matrix, Input matrix, Output matrix, Feedforward matrix

set_alg

set_alg(self, expr_name, expr)

Introduce new algebraic equation to model.

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

$$0 = g(x(t), u(t), z(t), p(t), p_{tv}(t))$$

or for a discrete model:

$$0 = g(x_k, u_k, z_k, p_k, p_{tv,k})$$

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

Parameters

- **expr_name** (str) – Name of the introduced expression
- **expr** (Union[SX, MX]) – CasADi SX or MX function depending on `_x`, `_u`, `_z`, `_tvp`, `_p`.

Return type

None

set_expression

set_expression(self, expr_name, expr)

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

Example:

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

```
Introduce two scalar states:
x_1 = model.set_variable('_x', 'x_1')
x_2 = model.set_variable('_x', 'x_2')

# Introduce expression:
model.set_expression('x1x2', x_1*x_2)
```

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

Parameters

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

Raises

- **assertion** – `expr_name` must be str

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

Returns

Union[SX, MX] – Returns the newly created expression. Expression can be used e.g. for the RHS.

set_meas

set_meas(self, meas_name, expr, meas_noise=True)

Introduce new measurable output to the model class.

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

or in case of discrete dynamics:

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

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

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

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

Note: It is often suggested to deactivate measurement noise for “measured” inputs (`_u`). These can typically be seen as certain variables.

Example:

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

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

Parameters

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

Raises

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

Returns

Union[SX, MX] – Returns the newly created measurement expression.

set_rhs

set_rhs(*self*, *var_name*, *expr*, *process_noise=False*)

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

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

or the update equation in case of discrete dynamics:

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

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

Example:

```
tank_level = model.set_variable('states', 'tank_level')
tank_temp = model.set_variable('states', 'tank_temp')

tank_level_next = 0.5*tank_level
tank_temp_next = ...

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

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

Parameters

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

Raises

- **assertion** – `var_name` must be str
- **assertion** – `expr` must be a casadi SX or MX type
- **assertion** – `var_name` must refer to the previously defined states
- **assertion** – Cannot call after `:py:func`setup``.

Return type

None

set_variable

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** (str) – Declare the type of the variable.
- **var_name** (str) – Set a user-defined name for the parameter. The names are reused throughout do_mpc.
- **shape** (Union[int, Tuple]) – Shape of the current variable (optional), defaults to 1.

The following types of **var_type** are valid (long or short name is possible):

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

Raises

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

Returns

Union[SX, MX] – Returns the newly created symbolic variable.

setup

setup(*self*)

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

Sets default measurement function (state feedback) if `set_meas()` was not called. :rtype: None

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

Raises

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

4.10.6.5.2 Attributes

aux

Model.aux

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

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

Example:

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

Useful CasADi symbolic structure methods:

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

Raises

assertion – Cannot set aux directly Use `set_expression` instead.

p

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
```

Useful CasADi symbolic structure methods:

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

Raises

assertion – Cannot set model variables directly Use set_variable instead.

tvp

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
```

Useful CasADi symbolic structure methods:

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

Raises

assertion – Cannot set model variables directly Use set_variable instead.

u**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
```

Useful CasADi symbolic structure methods:

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

Raises

assertion – Cannot set model variables directly Use set_variable instead.

v**Model.v**

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

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

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

Useful CasADi symbolic structure methods:

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

Raises

assertion – Cannot set v directly

w**Model.w**

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

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

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

Useful CasADi symbolic structure methods:

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

Raises

assertion – Cannot set w directly

x**Model.x**

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

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

Example:

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

Useful CasADi symbolic structure methods:

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

Raises

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

y

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
```

Useful CasADi symbolic structure methods:

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

Raises

assertion – Cannot set model variables directly Use set_meas instead.

z

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
```

Useful CasADi symbolic structure methods:

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

Raises

assertion – Cannot set model variables directly Use set_variable instead.

4.10.7 opcua

A OPC UA wrapper for do-mpc.

Classes

<i>ClientOpts</i>	Client Options.
<i>Namespace</i>	An OPC UA Namespace draft.
<i>NamespaceEntry</i>	An OPC UA node ID Namespace Entry.
<i>RTBase</i>	Real Time Base.
<i>RTClient</i>	Real Time Client.
<i>RTServer</i>	Real Time Server.
<i>ServerOpts</i>	Server Options.

4.10.7.1 ClientOpts

class `ClientOpts`(*name*, *address*, *port*, *timeunit*=1)

Bases: object

Client Options. A helper class to correctly define client options. Used for the setup of `do_mpc.opcua.RTClient`.

Parameters

- **name** (str) – Name of the client.
- **address** (str) – IP address of the target server.
- **port** (int) – Used port number of the target server.
- **timeunit** (int) – Time unit factor to convert the time unit used by the dynamic system into seconds. The default value is 1 for seconds. Use 60 for minutes, 3600 for hours, and so on.

4.10.7.1.1 Methods

4.10.7.1.2 Attributes

timeunit

`ClientOpts.timeunit: int = 1`

name

`ClientOpts.name: str`

IP address of the target server.

address

`ClientOpts.address: str`

port

`ClientOpts.port: int`

4.10.7.2 Namespace

class `Namespace(namespace_name, entry_list, _namespace_index=None)`

Bases: `object`

An OPC UA Namespace draft. A helper class to create node IDs for the setup of an OPC UA namespace. Used to setup `do_mpc.opcua.RTBase` and `do_mpc.opcua.RTClient`.

Parameters

- **namespace_name** (`str`) – Namespace name.
- **entry_list** (`List[NamespaceEntry]`) – A list of node IDs.
- **_namespace_index** (`Optional[int]`) – The index of an OPC UA namespace.

4.10.7.2.1 Methods**4.10.7.2.2 Attributes****namespace_name**

`Namespace.namespace_name: str`

entry_list

`Namespace.entry_list: List[NamespaceEntry]`

4.10.7.3 NamespaceEntry

class `NamespaceEntry(objectnode, variable)`

Bases: `object`

An OPC UA node ID Namespace Entry. A helper class to create an OPC UA node ID for `do_mpc.opcua.Namespace`.

Parameters

- **objectnode** (`str`) – Object node name .
- **variable** (`str`) – Variable name.

4.10.7.3.1 Methods

`get_node_id`

`get_node_id(self, namespace_index)`

Creates a node ID containing the namespace index as well as the variable name.

Parameters

namespace_index (int) – A OPC UA namespace index.

Returns

str – An OPC UA node ID string containing the namespace index and the variable name.

4.10.7.3.2 Attributes

`objectnode`

NamespaceEntry.**objectnode**: str

`variable`

NamespaceEntry.**variable**: str

4.10.7.4 RTBase

class `RTBase(do_mpc_object, clientOpts, namespace=None)`

Bases: object

Real Time Base. The RTBase class extends do-mpc with an OPC UA interface.

Note: The `do_mpc.estimator.MHE` class is currently not supported.

Use this class to configure an OPC UA client for a previously initiated do-mpc class e.g.. `do_mpc.controller.MPC` or `do_mpc.simulator.Simulator`.

Configuration and setup:

Configuring and setting up the RTBase class involves the following steps:

1. Use `do_mpc.opcua.ClientOpts` dataclass to specify client name as well as IP adress and port number of the target server.
2. Initiate the RTBase class with a do-mpc object and the dataclass `do_mpc.opcua.ClientOpts`.
3. Use `set_write_tags()` and `set_read_tags()` to take over the namespace tags (node IDs) from another instance of RTBase (optional).

Note: Use `set_write_tags()` and `set_read_tags()` only after registering all clients with the `do_mpc.opcua.RTServer.namespace_from_client()` method.

4. Use `connect()` to connect the client to the OPC UA server.

4. Use `write_to_tags()` to write initial values to the OPC UA server.
5. Use `async_step_start()` to run the do-mpc method `do_mpc.controller.MPC.make_step()`.

Parameters

- **do_mpc_object** – An instance of a do-mpc class.
- **clientOpts** (*ClientOpts*) – Client Options
- **namespace** (Optional[*Namespace*]) – Namespace containing OPC UA node IDs

4.10.7.4.1 Methods

`async_run`

`async_run(self)`

This method is called inside of `async_step_start()`. It calls `make_step()` and `async_step_start()`.

Return type

None

`async_step_start`

`async_step_start(self)`

This method calls the `async_run()` method in a frequency given by the do-mpc classes `t_step` value.

Return type

None

`async_step_stop`

`async_step_stop(self)`

Stops `async_step_start()` from running.

Return type

None

`connect`

`connect(self)`

Connects client to the server.

Return type

None

disconnect

disconnect(*self*)

Disconnects client from the server.

Return type

None

get_default_namespace

get_default_namespace(*self*, *namespace_name*)

Sets default namespace using [namespace_from_model\(\)](#).

Parameters

namespace_name (str) – Name given to the generated namespace

Return type

None

make_step

make_step(*self*)

Calls the do-mpc make_step method e.g.. [do_mpc.controller.MPC.make_step\(\)](#). The input for make_step is taken from node IDs specified in [read_from_tags\(\)](#). The output of make_step is written to the node IDs specified in [write_to_tags\(\)](#).

Return type

None

namespace_from_model

namespace_from_model(*self*, *model*, *model_name*)

Create a OPC UA namespace from the provided model.

Parameters

- **model** ([Model](#)) – A do-mpc model.
- **model_name** (str) – Name given to the generated namespace.

Returns

[Namespace](#) – Namespace generated from the OPC UA model.

read_from_tags

read_from_tags(*self*)

Read from the node IDs specified in [read_from_tags\(\)](#).

Returns

ndarray – Values stored on the OPC UA server.

set_read_tags

set_read_tags(*self*, *tagin*)

Set tags (node IDs) to read from. The provided tags must match the node IDs registered on the taget server.

Parameters

tagin (List[str]) – A list of node IDs from which the client reads.

Return type

None

set_write_tags

set_write_tags(*self*, *tagout*)

Set tags (node IDs) to write to. The provided tags must match the node IDs registered on the taget server.

Parameters

tagout (List[str]) – A list of node IDs to which the client writes.

Return type

None

write_to_tags

write_to_tags(*self*, *data*)

Write to the node IDs specified in [write_to_tags\(\)](#)

Parameters

data (ndarray) – data which is written to server.

Return type

None

4.10.7.5 RTClient

class RTClient(*opts*, *namespace*)

Bases: object

Real Time Client. The RTClient class extends do-mpc by an easy to setup OPC UA client.

Note: The RTClient class main purpose is to setup an OPC UA client inside the [do_mpc.opcua.RTBase](#) class.

Configuration and setup:

Configuring and setting up the RTClient client involves the following steps:

1. Use [do_mpc.opcua.ClientOpts](#) dataclass to specify client name as well as IP adress and port for the target server.
2. Use the [do_mpc.opcua.Namespace](#) dataclass to setup the namespace stored in the RTClient instance.
3. Initiate the RTClient instance with instances of [do_mpc.opcua.ClientOpts](#) and [do_mpc.opcua.Namespace](#).
4. Connect the RTClient to the taget server with [connect\(\)](#)

Note: Remember to disconnect the RTClient class afterwards with `disconnect()`

Parameters

- **opts** (*ClientOpts*) – Client options.
- **namespace** (*Namespace*) – Namespace draft stored in RTClient.

4.10.7.5.1 Methods

add_namespace_url

add_namespace_url(*self*, *url*)

This method is used to add an OPC UA namespace index to the stored namespace.

Parameters

url (int) – The OPC UA namespace index.

Return type

None

connect

connect(*self*)

Connects the client to the target server.

Return type

None

disconnect

disconnect(*self*)

Disconnects the client from the target server.

Return type

None

readData

readData(*self*, *tag*)

Reads a variable from the target server.

Parameters

tag (str) – The node ID of the target variable on the OPC UA server.

Returns

float – The value stored on the target server.

writeData

writeData(*self*, *tag*, *dataVal*)

Overwrites a variable on the target server.

Parameters

- **tag** (str) – The node ID of the target variable on the OPC UA server.
- **dataVal** (list) – The value written to the specified node ID

Return type

None

4.10.7.6 RTServer

class RTServer(*opts*)

Bases: object

Real Time Server. The RTServer class extends do-mpc with an easy to setup opcua server.

Configuration and setup:

Configuring and setting up the RTServer client involves the following steps:

1. Use `do_mpc.opcua.ServerOpts` dataclass to specify server name as well as IP adress and port for the server.
2. Initiate the RTServer class with the ServerOpts dataclass.
3. Use the `namespace_from_client()` to automatically generate a namespace from a `do_mpc.opcua.RTBase` instance (optional).
4. Start the OPC UA server by calling `start()`

Note: Remember to properly stop the server afterwards using the `stop()` method.

Parameters

opts (`ServerOpts`) – Server options.

4.10.7.6.1 Methods

add_variable_to_node

add_variable_to_node(*self*, *namespace_entry*, *namespace_url*)

Adds a variable to a registered node on the OPC UA server.

Parameters

- **namespace_entry** (`NamespaceEntry`) – A OPCUA node ID. Contains the variable as well as the target node name.
- **namespace_url** (int) – The namespace index identifying the namespace on the OPC UA server

Return type

None

namespace_from_client

namespace_from_client(*self*, *client*)

Takes an instance of `do_mpc.opcua.RTBase` as input and registers an OPC UA namespace for the namespace stored in the RTBase class.

Parameters

client (*RTClient*) – A client with a stored namespace.

Return type

None

start

start(*self*)

Starts the OPC UA server.

Return type

None

stop

stop(*self*)

Stops the OPC UA server

Return type

None

4.10.7.7 ServerOpts

class ServerOpts(*name*, *address*, *port*)

Bases: object

Server Options. A helper class to correctly define server options. Used for the setup of `do_mpc.opcua.RTServer`

Parameters

- **name** (str) – Name of the server.
- **address** (str) – IP address of the server.
- **port** (int) – Used port number.

4.10.7.7.1 Methods

4.10.7.7.2 Attributes

name

ServerOpts.name: str

address

ServerOpts.address: str

port

ServerOpts.port: int

4.10.8 optimizer

Shared tools for optimization-based estimation (MHE) and control (MPC).

Classes

Optimizer

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

4.10.8.1 Optimizer

class Optimizer

Bases: object

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. The methods and properties are inherited to *do_mpc.estimator.MHE* and *do_mpc.controller.MPC*.

4.10.8.1.1 Methods

compile_nlp

compile_nlp(self, overwrite=False, cname='nlp.c', libname='nlp.so', compiler_command=None)

Compile the NLP. This may accelerate the optimization. As compilation is time consuming, the default option is to NOT overwrite (overwrite=False) an existing compilation. If an existing compilation with the name libname is found, it is used. **This can be dangerous, if the NLP has changed** (user tweaked the cost function, the model etc.).

Warning: This feature is experimental and currently only supported on Linux and MacOS.

What happens here?

1. The NLP is written to a C-file (cname)
2. The C-File (cname) is compiled. The custom compiler uses:

```
gcc -fPIC -shared -O1 {cname} -o {libname}
```

3. The compiled library is linked to the NLP. This overwrites the original NLP. Options from the previous NLP (e.g. linear solver) are kept.

```
self.S = nlpsol('solver_compiled', 'ipopt', f'{libname}', self.nlpsol_opts)
```

Parameters

- **overwrite** (bool) – If True, the existing compiled NLP will be overwritten.
- **cname** (str) – Name of the C file that will be exported.
- **libname** (str) – Name of the shared library that will be created after compilation.
- **compiler_command** (Optional[str]) – Command to use for compiling. If None, the default compiler command will be used. Please make sure to use matching strings for **libname** when supplying your custom compiler command.

Return type

None

create_nlp

create_nlp(self)

Create the optimization problem. Typically, this method is called internally from `setup()`.

Users should only call this method if they intend to modify the objective with `nlp_obj`, the constraints with `nlp_cons`, `nlp_cons_lb` and `nlp_cons_ub`.

To finish the setup process, users MUST call `create_nlp()` afterwards.

Note: Do NOT call `setup()` if you intend to go the manual route with `prepare_nlp()` and `create_nlp()`.

Note: Only AFTER calling `prepare_nlp()` the previously mentioned attributes `nlp_obj`, `nlp_cons`, `nlp_cons_lb`, `nlp_cons_ub` become available.

Returns

None – None

get_tvp_template

get_tvp_template(self)

Obtain output template for `set_tvp_fun()`.

The method returns a structured object with `n_horizon+1` elements, and a set of time-varying parameters (as defined in `do_mpc.model.Model`) for each of these instances. The structure is initialized with all zeros. Use this object to define values of the time-varying parameters.

This structure (with numerical values) should be used as the output of the `tvp_fun` function which is set to the class with `set_tvp_fun()`. Use the combination of `get_tvp_template()` and `set_tvp_fun()`.

Example:

```

# 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

Union[SXStruct, MXStruct] – Casadi SX or MX structure

prepare_nlp**prepare_nlp(self)**

Prepare the optimization problem. Typically, this method is called internally from `setup()`.

Users should only call this method if they intend to modify the objective with `nlp_obj`, the constraints with `nlp_cons`, `nlp_cons_lb` and `nlp_cons_ub`.

To finish the setup process, users MUST call `create_nlp()` afterwards.

Note: Do NOT call `setup()` if you intend to go the manual route with `prepare_nlp()` and `create_nlp()`.

Note: Only AFTER calling `prepare_nlp()` the previously mentionned attributes `nlp_obj`, `nlp_cons`, `nlp_cons_lb`, `nlp_cons_ub` become available.

Returns

None – None

reset_history

reset_history(*self*)

Reset the history of the optimizer. All data from the `do_mpc.data.Data` instance is removed.

Return type

None

set_nl_cons

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** (str) – Arbitrary name for the given expression. Names are used for key word indexing.
- **expr** (Union[SX, MX]) – CasADi SX or MX function depending on `_x`, `_u`, `_z`, `_tvp`, `_p`.
- **ub** (float) – Upper bound
- **soft_constraint** (bool) – Flag to enable soft constraint
- **penalty_term_cons** (int) – Penalty term constant
- **maximum_violation** (float) – Maximum violation

Raises

- **assertion** – `expr_name` must be str
- **assertion** – `expr` must be a casadi SX or MX type

Returns

Union[SX, MX] – Returns the newly created expression. Expression can be used e.g. for the RHS.

set_tvp_fun

set_tvp_fun(self, tvp_fun)

Set function which returns time-varying parameters.

The `tvp_fun` is called at each optimization step to get the current prediction of the time-varying parameters. The supplied function must be callable with the current time as the only input. Furthermore, the function must return a CasADi structured object which is based on the horizon and on the model definition. The structure can be obtained with `get_tvp_template()`.

Example:

```
# in model definition:
alpha = model.set_variable(var_type='_tvp', var_name='alpha')
beta = model.set_variable(var_type='_tvp', var_name='beta')

...
# in optimizer configuration:
tvp_temp_1 = optimizer.get_tvp_template()
tvp_temp_1['_tvp', :] = np.array([1,1])

tvp_temp_2 = optimizer.get_tvp_template()
tvp_temp_2['_tvp', :] = np.array([0,0])

def tvp_fun(t_now):
    if t_now<10:
        return tvp_temp_1
    else:
        tvp_temp_2

optimizer.set_tvp_fun(tvp_fun)
```

Note: The method `set_tvp_fun()`. must be called prior to setup IF time-varying parameters are defined in the model. It is not required to call the method if no time-varying parameters are defined.

Parameters

tvp_fun (Callable[[float], Union[SXStruct, MXStruct]]) – 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()`).

Return type

None

solve

`solve(self)`

Solves the optimization problem.

The current problem is defined by the parameters in the `opt_p_num` CasADi structured Data.

Typically, `opt_p_num` is prepared for the current iteration in the `make_step()` method. It is, however, valid and possible to directly set parameters in `opt_p_num` before calling `solve()`.

The method updates the `opt_p_num` and `opt_x_num` attributes of the class. By resetting `opt_x_num` to the current solution, the method implicitly enables **warmstarting the optimizer** for the next iteration, since this vector is always used as the initial guess. `:rtype:` None

Warning: The method is part of the public API but it is generally not advised to use it. Instead we recommend to call `make_step()` at each iterations, which acts as a wrapper for `solve()`.

Raises

AssertionError – Optimizer was not setup yet.

4.10.8.1.2 Attributes

bounds

Optimizer.bounds

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

lb_opt_x

Optimizer.lb_opt_x

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

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

Note: The attribute automatically considers the scaling variables when setting the bounds. See *scaling* for more details.

Note: Modifications must be done after calling *prepare_nlp()* or `setup()` respectively.

nlp_cons

Optimizer.nlp_cons

Query and modify (symbolically) the NLP constraints. Use the variables in `opt_x` and `opt_p`.

Prior to calling *create_nlp()* this attribute returns a list of symbolic constraints. After calling *create_nlp()* this attribute returns the concatenation of this list and the attribute cannot be altered anymore.

It is advised to append to the current list of *nlp_cons*:

```
mpc.prepare_nlp()

# Create new constraint: Input at timestep 0 and 1 must be identical.
extra_cons = mpc.opt_x['_u', 0, 0]-mpc.opt_x['_u', 1, 0]
mpc.nlp_cons.append(
    extra_cons
)

# Create appropriate upper and lower bound (here they are both 0 to create an
↪equality constraint)
mpc.nlp_cons_lb.append(np.zeros(extra_cons.shape))
mpc.nlp_cons_ub.append(np.zeros(extra_cons.shape))

mpc.create_nlp()
```

See the documentation of `opt_x` and `opt_p` on how to query these attributes.

Warning: This is a VERY low level feature and should be used with extreme caution. It is easy to break the code.

Be especially careful NOT to accidentally overwrite the default objective.

Note: Modifications must be done after calling *prepare_nlp()* and before calling *create_nlp()*

nlp_cons_lb

Optimizer.nlp_cons_lb

Query and modify the lower bounds of the *nlp_cons*.

Prior to calling *create_nlp()* this attribute returns a list of lower bounds matching the list of constraints obtained with *nlp_cons*. After calling *create_nlp()* this attribute returns the concatenation of this list.

Values for lower (and upper) bounds MUST be added when adding new constraints to *nlp_cons*.

Warning: This is a VERY low level feature and should be used with extreme caution. It is easy to break the code.

Note: Modifications must be done after calling *prepare_nlp()*

nlp_cons_ub

Optimizer.nlp_cons_ub

Query and modify the upper bounds of the *nlp_cons*.

Prior to calling *create_nlp()* this attribute returns a list of upper bounds matching the list of constraints obtained with *nlp_cons*. After calling *create_nlp()* this attribute returns the concatenation of this list.

Values for upper (and lower) bounds MUST be added when adding new constraints to *nlp_cons*.

Warning: This is a VERY low level feature and should be used with extreme caution. It is easy to break the code.

Note: Modifications must be done after calling *prepare_nlp()*

nlp_obj

Optimizer.nlp_obj

Query and modify (symbolically) the NLP objective function. Use the variables in *opt_x* and *opt_p*.

It is advised to add to the current objective, e.g.:

```
mpc.prepare_nlp()
# Modify the objective
mpc.nlp_obj += sum1(vercat(*mpc.opt_x['_x', -1, 0])**2)
# Finish creating the NLP
mpc.create_nlp()
```

See the documentation of *opt_x* and *opt_p* on how to query these attributes.

Warning: This is a VERY low level feature and should be used with extreme caution. It is easy to break the code.

Be especially careful NOT to accidentally overwrite the default objective.

Note: Modifications must be done after calling `prepare_nlp()` and before calling `create_nlp()`

scaling

Optimizer.scaling

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

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.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']
```

Scaling factors a affect the MHE / MPC optimization problem. The optimization variables are scaled variables:

$$\bar{\phi} = \frac{\phi}{a_{\phi}} \quad \forall \phi \in [x, u, z, p_{\text{est}}]$$

Scaled variables are used to formulate the bounds $\bar{\phi}_{lb} \leq \bar{\phi}_{ub}$ and for the evaluation of the ODE. For the objective function and the nonlinear constraints the unscaled variables are used. The algebraic equations are also not scaled.

Note: Scaling the optimization problem is suggested when states and / or inputs take on values which differ by orders of magnitude.

ub_opt_x

Optimizer.ub_opt_x

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

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

Note: The attribute automatically considers the scaling variables when setting the bounds. See *scaling* for more details.

Note: Modifications must be done after calling *prepare_nlp()* or *setup()* respectively.

4.10.9 sampling

Sampling tools for data generation.

For a quick introduction of the **do-mpc** sampling tools we are providing this video tutorial:

Classes

<i>DataHandler</i>	Post-processing data created from a sampling plan.
<i>Sampler</i>	Generate samples based on a sampling plan.
<i>SamplingPlanner</i>	A class for generating sampling plans.

4.10.9.1 DataHandler

class `DataHandler`(*sampling_plan*, ****kwargs**)

Bases: `object`

Post-processing data created from a sampling plan. Data (individual samples) were created with *do_mpc.sampling.Sampler*. The list of all samples originates from *do_mpc.sampling.SamplingPlanner* and is used to initiate this class (*sampling_plan*).

The class can be created with optional keyword arguments which are passed to *set_param()*.

Configuration and retrieving processed data:

1. Initiate the object with the *sampling_plan* originating from *do_mpc.sampling.SamplingPlanner*.
2. Set parameters with *set_param()*. Most importantly, the directory in which the individual samples are located should be passed with *data_dir* argument.
3. (Optional) set one (or multiple) post-processing functions. These functions are applied to each loaded sample and can, e.g., extract or compile important information.
4. Load and return samples either by indexing with the *__getitem__()* method or by filtering with *filter()*.

Example:


```

sp = do_mpc.sampling.SamplingPlanner()

# Plan with two variables alpha and beta:
sp.set_sampling_var('alpha', np.random.randn)
sp.set_sampling_var('beta', lambda: np.random.randint(0,5))

plan = sp.gen_sampling_plan(n_samples=10)

sampler = do_mpc.sampling.Sampler(plan)

# Sampler computes the product of two variables alpha and beta
# that were created in the SamplingPlanner:

def sample_function(alpha, beta):
    return alpha*beta

sampler.set_sample_function(sample_function)

sampler.sample_data()

# Create DataHandler object with same plan:
dh = do_mpc.sampling.DataHandler(plan)

# Assume you want to compute the square of the result of each sample
dh.set_post_processing('square', lambda res: res**2)

# As well as the value itself:
dh.set_post_processing('default', lambda res: res)

# Query all post-processed results with:
dh[:]

```

__getitem__(*ind*)

Index results from the *DataHandler*. Pass an index or a slice operator.

4.10.9.1.1 Methods

filter

filter(*self*, *input_filter=None*, *output_filter=None*)

Filter data from the *DataHandler*. Filters can be applied to inputs or to results that were obtained with the post-processing functions. Filtering returns only a subset from the created samples based on arbitrary conditions.

Example:

```

sp = do_mpc.sampling.SamplingPlanner()

# SamplingPlanner with two variables alpha and beta:
sp.set_sampling_var('alpha', np.random.randn)
sp.set_sampling_var('beta', lambda: np.random.randint(0,5))
plan = sp.gen_sampling_plan()

```

(continues on next page)

(continued from previous page)

```

...

dh = do_mpc.sampling.DataHandler(plan)
dh.set_post_processing('square', lambda res: res**2)

# Return all samples with alpha < 0 and beta > 2
dh.filter(input_filter = lambda alpha, beta: alpha < 0 and beta > 2)
# Return all samples for which the computed value square < 5
dh.filter(output_filter = lambda square: square < 5)

```

Parameters

- **input_filter** (Union[FunctionType, BuiltinMethodType, None]) – Function to filter the data.
- **output_filter** (Union[FunctionType, BuiltinMethodType, None]) – Function to filter the data

Raises

- **assertion** – No post processing function is set
- **assertion** – filter_fun must be either Function of BuiltinFunction_or_Method

Returns

list – Returns the post processed samples that satisfy the filter

set_param

set_param(self, **kwargs)

Set the parameters of the DataHandler.

Parameters must be passed as pairs of valid keywords and respective argument. For example:

```
datahandler.set_param(overwrite = True)
```

Parameters

- **data_dir** (bool) – Directory where the data can be found (as defined in the [do_mpc.sampling.Sampler](#)).
- **sample_name** (str) – Naming scheme for samples (as defined in the [do_mpc.sampling.Sampler](#)).
- **save_format** (str) – Choose either pickle or mat (as defined in the [do_mpc.sampling.Sampler](#)).

Return type

None

set_post_processing

set_post_processing(self, name, post_processing_function)

Set a post processing function. The post processing function is applied to all loaded samples, e.g. with `__getitem__()` or `filter()`. Users can set an arbitrary amount of post processing functions by repeatedly calling this method.

The `post_processing_function` can have two possible signatures:

1. `post_processing_function(case_definition, sample_result)`
2. `post_processing_function(sample_result)`

Where `case_definition` is a dict of all variables introduced in the `do_mpc.sampling.SamplingPlanner` and `sample_results` is the result obtained from the function introduced with `do_mpc.sampling.Sampler.set_sample_function`.

Note: Setting a post processing function with an already existing name will overwrite the previously set post processing function.

Example:

```
sp = do_mpc.sampling.SamplingPlanner()

# Plan with two variables alpha and beta:
sp.set_sampling_var('alpha', np.random.randn)
sp.set_sampling_var('beta', lambda: np.random.randint(0,5))

plan = sp.gen_sampling_plan(n_samples=10)

sampler = do_mpc.sampling.Sampler(plan)

# Sampler computes the product of two variables alpha and beta
# that were created in the SamplingPlanner:

def sample_function(alpha, beta):
    return alpha*beta

sampler.set_sample_function(sample_function)

sampler.sample_data()

# Create DataHandler object with same plan:
dh = do_mpc.sampling.DataHandler(plan)

# Assume you want to compute the square of the result of each sample
dh.set_post_processing('square', lambda res: res**2)

# As well as the value itself:
dh.set_post_processing('default', lambda res: res)

# Query all post-processed results with:
dh[:]
```

Parameters

- **name** (str) – Name of the output of the post-processing operation
- **post_processing_function** (Union[FunctionType, BuiltinMethodType]) – The post processing function to be evaluated

Raises

- **assertion** – name must be string
- **assertion** – post_processing_function must be either Function of BuiltinFunction

Return type

None

4.10.9.1.2 Attributes

data_dir**DataHandler.data_dir**

Set the directory where the results are stored.

4.10.9.2 Sampler

class Sampler(*sampling_plan*, ***kwargs*)

Bases: object

Generate samples based on a sampling plan. Initiate the class by passing a *do_mpc.sampling.SamplingPlanner* (*sampling_plan*) object. The class can be configured to create samples based on the defined cases in the *sampling_plan*.

The class can be created with optional keyword arguments which are passed to *set_param()*.

Configuration and sampling:

1. (Optional) use *set_param()* to configure the class. Use *data_dir* to choose the save location for the samples.
2. Set the sample generating function with *set_sample_function()*. This function is executed for each of the samples in the *sampling_plan*.
3. Use *sample_data()* to generate all samples defined in the *sampling_plan*. A new file is written for each sample.
4. **Or:** Create an individual sample result with *sample_idx()*, where an index (int) referring to the *sampling_plan* determines the sampled case.

Note: By default, the *Sampler* will only create samples that do not already exist in the chosen *data_dir*.

Example:

```
sp = do_mpc.sampling.SamplingPlanner()

# Plan with two variables alpha and beta:
sp.set_sampling_var('alpha', np.random.randn)
```

(continues on next page)

(continued from previous page)

```

sp.set_sampling_var('beta', lambda: np.random.randint(0,5))

plan = sp.gen_sampling_plan(n_samples=10)

sampler = do_mpc.sampling.Sampler(plan)

# Sampler computes the product of two variables alpha and beta
# that were created in the SamplingPlanner:

def sample_function(alpha, beta):
    return alpha*beta

sampler.set_sample_function(sample_function)

sampler.sample_data()

```

Parameters**sampling_plan** (list) –**4.10.9.2.1 Methods****sample_data****sample_data(self)**

Sample data after having configured the Sampler. No user input is required and the method will iterate through all the items defined in the `sampling_plan` (obtained with `do_mpc.sampling.SamplingPlanner`). :rtype: None

Note: Depending on your `sample_function` (set with `set_sample_function()`) and the total number of samples, executing this method may take some time.

Note: If `sampler.set_param(overwrite = False)` (default) data will only be sampled for instances that do not yet exist.

sample_idx**sample_idx(self, idx)**

Sample case based on the index of the sample.

Parameters**idx** (int) – Index of the `sampling_plan` for which the sample should be created.**Raises**

- **assertion** – Index must be between 0 and `n_samples`.
- **assertion** – `sample_function` must be set prior to sampling data.

Return type

None

set_param**set_param**(*self*, ***kwargs*)Configure the `do_mpc.sampling.Sampler` class.

Parameters must be passed as pairs of valid keywords and respective argument. For example:

```
sampler.set_param(overwrite = True)
```

Parameters

- **overwrite** (*bool*) – Should previously created results be overwritten. Default is False
- **sample_name** (*str*) – Naming scheme for samples.
- **save_format** (*str*) – Choose either pickle or mat.
- **print_progress** (*bool*) – Print progress-bar to terminal. Default is True.

Return type

None

set_sample_function**set_sample_function**(*self*, *sample_function*)Set sample generating function. The sampling function produces a sample result for each sample definition in the `sampling_plan` and is called in the method `sample_data()`.It is important that the sample function only uses keyword arguments **with the same name as previously defined** in the `sampling_plan`.**Example:**

```
sp = do_mpc.sampling.SamplingPlanner()

sp.set_sampling_var('alpha', np.random.randn)
sp.set_sampling_var('beta', lambda: np.random.randint(0,5))

sampler = do_mpc.sampling.Sampler(plan)

def sample_function(alpha, beta):
    return alpha*beta

sampler.set_sample_function(sample_function)
```

Parameters

sample_function (Callable[[Union[FunctionType, BuiltinMethodType], Union[FunctionType, BuiltinMethodType]], Union[FunctionType, BuiltinMethodType]]) – Function to create each sample of the sampling plan.

Return type

None

4.10.9.2.2 Attributes

data_dir

`Sampler.data_dir`

Set the save directory for the results. If the directory does not exist yet, it is created. If the directory is nested all (non-existing) parent folders are also created.

Example:

```
sampler = do_mpc.sampling.Sampler()
sampler.data_dir = './samples/experiment_1/'
```

This will set the directory to the indicated path. If the path does not exist, all folders are created.

4.10.9.3 SamplingPlanner

`class SamplingPlanner(**kwargs)`

Bases: `object`

A class for generating sampling plans. These sampling plans will be executed by `do_mpc.sampling.Sampler` to generate data.

The class can be created with optional keyword arguments which are passed to `set_param()`.

Configuration and sampling plan generation:

1. Set variables which should be sampled with `set_sampling_var()`.
2. (Optional) Set further options of the SamplingPlanner with `set_param()`
3. Generate the sampling plan with `gen_sampling_plan()`.
4. And / or: Add specific sampling case with `add_sampling_case()`.
5. Export the plan with all sampling cases with `export()`

4.10.9.3.1 Methods

add_sampling_case

`add_sampling_case(self, **kwargs)`

Manually add sampling case with user-defined values. Create a sampling case by choosing values for the previously introduced sampling variables (with `set_sampling_var()`).

Method takes arbitrary (keyword, argument) pairs, where the keywords must refer to previously introduced sampling variables. `add_sampling_case()` will automatically augment the sampling case with values for variables that are not passed as arguments. This only works if these variables were created with the argument `fun_var_pdf`.

Example:

```
sp = do_mpc.sampling.SamplingPlanner()

# Plan with two variables alpha and beta:
sp.set_sampling_var('alpha', np.random.randn)
```

(continues on next page)

(continued from previous page)

```
sp.set_sampling_var('beta', lambda: np.random.randint(0,5))

# Create two new sampling cases, missing variable is auto-generated:
sp.add_sampling_case(alpha=1)
sp.add_sampling_case(beta= 0)
```

Returns

list – Returns the newly created sampling plan.

export**export**(self, sampling_plan_name)

Export SamplingPlan in pickle format. Pass `sampling_plan_name` without any path. File extension can be added (but will be stripped automatically). Change the path with `data_dir`.

Parameters

sampling_plan_name (str) – Name of the exported sampling plan file.

Raises

assertion – `sampling_plan_name` must be string.

Return type

None

gen_sampling_plan**gen_sampling_plan**(self, n_samples)

Generate the sampling plan. The generated plan contains `n_samples` samples based on the defined variables and the corresponding evaluation functions.

Parameters

n_samples (int) – The number of generated samples

Raises

assertion – `n_samples` must be int

Returns

list – Returns the newly created sampling plan.

product**product**(self, **kwargs)

Cartesian product of input variables. This method is inspired by `itertools.product`.

Must pass a list for each `sampling_var` that should be considered. Not all `sampling_vars` must be referenced. Sampling vars that are excluded, will generate a value according to their assigned `fun_var_pdf` (see `set_sampling_var()`).

Parameters

kwargs (dict) – Keyword arguments of the form `var_name=var_values`.

Returns

list – Returns the newly created sampling plan.

set_param

set_param(*self*, ***kwargs*)

Set the parameters of the SamplingPlanner class. Parameters must be passed as pairs of valid keywords and respective argument. For example:

```
sp.set_param(overwrite = True)
```

It is also possible and convenient to pass a dictionary with multiple parameters simultaneously as shown in the following example:

```
setup_dict = {
    'overwrite': True,
    'save_format': pickle,
}
sp.set_param(**setup_dict)
```

This makes use of thy python “unpack” operator. See [more details here](#).

Note: `set_param()` can be called multiple times. Previously passed arguments are overwritten by successive calls.

The following parameters are available:

Parameters

- **overwrite** (*bool*) – Overwrites existing samplingplan under the same name, if set to True.
- **id_precision** (*str*) – Padding for IDs of created samples. Defaults to 3. This means sample 20 will be denoted as 020.

Return type

None

set_sampling_var

set_sampling_var(*self*, *name*, *fun_var_pdf*=None)

Introduce new sampling variables to the SamplingPlanner. Define variable name. Optionally add a function to generate values for the sampled variable (e.g. following some distribution). The parameter `fun_var_pdf` defaults to None.

Note: If no value-generating function is passed (for any of the introduced variables), all sampling cases must be created manually with `add_sampling_case()`.

Note: Value generating function `fun_var_pdf` must not require inputs.

Example:

```
sp = do_mpc.sampling.SamplingPlanner()
```

(continues on next page)

(continued from previous page)

```
# Plan with two variables alpha and beta:
sp.set_sampling_var('alpha', np.random.randn)
sp.set_sampling_var('beta', lambda: np.random.randint(0,5))
```

In the example we have passed a `BuiltinFunction` for the introduced variable `alpha`. We use the function that created values from the random normal distribution with zero mean and unity covariance. For the variable `beta` we created a new lambda function that draws random integers from 0 to 5.

Parameters

- **name** (str) – Name of the sampled variable
- **fun_var_pdf** (Optional[Callable[[], Union[float, int]]]) – Declare the value-generating function of the sampled variable

Raises

- **assertion** – name must be string
- **assertion** – fun_var_pdf must be Function or BuiltinFunction

Return type

None

4.10.9.3.2 Attributes

data_dir

SamplingPlanner.data_dir

Set the save directory for the `samplingplan`. If the directory does not exist yet, it is created. If the directory is nested all (non-existing) parent folders are also created.

Example:

```
sp = do_mpc.sampling.SamplingPlanner()
sp.data_dir = './samples/experiment_1/'
```

This will set the directory to the indicated path. If the path does not exist, all folders are created.

4.10.10 simulator

Simulate continuous-time ODE/DAE or discrete-time dynamic systems.

Classes

<i>Simulator</i>	A class for simulating systems.
<i>SimulatorSettings</i>	Settings for <i>Simulator</i> .
<i>ContinuousSimulatorSettings</i>	Settings for <i>Simulator</i> for continuous-time systems.

4.10.10.1 Simulator

class Simulator(*model*)

Bases: *IteratedVariables*

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

New in version >v4.5.1: New interface to settings. The class has an attribute `settings` which is an instance of *SimulatorSettings* or *ContinuousSimulatorSettings* (please see this documentation for a list of available settings). Settings are now chosen as:

```
simulator.settings.t_step = 0.5
```

Previously, settings were passed to `set_param()`. This method is still available and wraps the new interface. The new method has important advantages:

1. The `simulator.settings` attribute can be printed to see the current configuration.
2. Context help is available in most IDEs (e.g. VS Code) to see the available settings, the type and a description.

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. Configure the simulator with *SimulatorSettings* or *ContinuousSimulatorSettings*. The simulator instance has the attribute `settings` which is an instance of *SimulatorSettings* or *ContinuousSimulatorSettings*.
2. Set parameter function with `get_p_template()` and `set_p_fun()`.
3. Set time-varying parameter function with `get_tvp_template()` and `set_tvp_fun()`.
4. Setup simulator with `setup()`.

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

Parameters

model (*Model*) – A configured and setup `do_mpc.model.Model`

4.10.10.1.1 Methods

get_p_template

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

Union[SXStruct, MXStruct] – numerical CasADi structure

get_tvp_template

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

Union[SXStruct, MXStruct] – numerical CasADi structure

init_algebraic_variables

init_algebraic_variables(*self*)

Initializes the algebraic variables. Solve the algebraic equations for the initial values of `x0`, `u0`, `p0`, `tvp0`. Sets the results to `z0` and returns them.

Note: The method internally calls `set_initial_guess()` to set the initial guess for the algebraic variables.

The initialization is computed by solving the algebraic model equations under consideration of the initial guess supplied in `z0`.

Example:

```
simulator = do_mpc.simulator.Simulator(model)

# Set initial value for the state:
simulator.x0 = np.array([0.1, 0.1]).reshape(-1,1)

# Obtain initial guess for the algebraic variables:
z0 = simulator.init_algebraic_variables()

# Initial guess is stored in simulator.z0 and simulator.set_initial_guess() was
# ↪ called internally.
```

Returns

ndarray – Initial guess for the algebraic variables.

make_step

make_step(*self*, *u0=None*, *v0=None*, *w0=None*)

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

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

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

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

Parameters

- **u0** (Optional[ndarray]) – Current input to the system. Optional parameter for autonomous systems.
- **v0** (Optional[ndarray]) – Additive measurement noise
- **w0** (Optional[ndarray]) – Additive process noise

Returns

ndarray – y_next

reset_history**reset_history(self)**

Reset the history of the simulator.

Return type

None

set_initial_guess**set_initial_guess(self)**

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

The simulator uses “warmstarting” to solve the continuous/discrete DAE system by using the previously computed algebraic states as an initial guess. Thus, this method is typically only invoked once. :rtype: None

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

set_p_fun**set_p_fun(self, p_fun)**

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

Example:

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

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

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

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

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

The parameter function can look something like this:

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

def p_fun(t_now):
    return p_template

simulator.set_p_fun(p_fun)
```

which results in constant parameters.

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

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

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

Parameters

p_fun (Callable[[float], Union[SXStruct, MXStruct]]) – A function which gives the values of the parameters

Raises

assert – p must have the right structure

Return type

None

set_param

set_param(self, **kwargs)

Warning: This method will be depreciated in a future version. Settings are available via the `settings` attribute which is an instance of `ContinuousSimulatorSettings` or `SimulatorSettings`.

Note: A comprehensive list of all available parameters can be found in `ContinuousSimulatorSettings` or `SimulatorSettings`.

For example:

```
simulator.settings.t_step = 0.5
```

The old interface, as shown in the example below, can still be accessed until further notice.

```
simulator.set_param(t_step=0.5)
```

Note: The only required parameters are `t_step`. All other parameters are optional.

Return type

None

set_tvp_fun

set_tvp_fun(*self*, *tvp_fun*)

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

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

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

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

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

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

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

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

simulator.set_tvp_fun(tvp_fun)
```

which results in constant parameters.

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

Parameters

tvp_fun (Callable[[float], Union[SXStruct, MXStruct]]) – 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.

Return type

None

setup

`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

Return type

None

simulate

`simulate(self)`

Call the CasADi simulator.

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

Numerical values for `sim_x_num` and `sim_p_num` need to be provided beforehand in order to simulate the system for one time step:

- states `sim_c_num['_x']`
- algebraic states `sim_z_num['_z']`
- inputs `sim_p_num['_u']`
- parameter `sim_p_num['_p']`
- time-varying parameters `sim_p_num['_tvp']`

The function returns the new state of the system.

Returns

ndarray – `x_new`

4.10.10.1.2 Attributes

`t0`

Simulator.`t0`

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

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

u0**Simulator.u0**

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

Example:

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

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

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

Useful CasADi symbolic structure methods:

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

x0**Simulator.x0**

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

Example:

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

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

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

Useful CasADi symbolic structure methods:

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

z0

Simulator.z0

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

Example:

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

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

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

Useful CasADi symbolic structure methods:

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

4.10.10.2 SimulatorSettings

class SimulatorSettings(*t_step=None*)

Bases: object

Settings for *Simulator*. An instance of this class is automatically generated as the attribute *settings* when creating the *Simulator*.

Example:

```
simulator = do_mpc.simulator.Simulator(model)
simulator.settings.t_step = 0.5
```

Parameters

t_step (Optional[float]) –

4.10.10.2.1 Methods

check_for_mandatory_settings

check_for_mandatory_settings(*self*)

Method to assert the necessary settings required to design *do_mpc.controller*

4.10.10.2.2 Attributes

t_step

`SimulatorSettings.t_step: float = None`

Timestep of the Simulator

4.10.10.3 ContinousSimulatorSettings

class `ContinousSimulatorSettings(t_step=None)`

Bases: `SimulatorSettings`

Settings for `Simulator` for continous-time systems.

An instance of this class is automatically generated as the attribute settings when creating the `Simulator`.

Example:

```
simulator = do_mpc.simulator.Simulator(model)
simulator.settings.t_step = 0.5
```

Parameters

t_step (Optional[float]) –

4.10.10.3.1 Methods

check_for_mandatory_settings

check_for_mandatory_settings(*self*)

Method to assert the necessary settings required to design `do_mpc.controller`

4.10.10.3.2 Attributes

abstol

`ContinousSimulatorSettings.abstol: float = 1e-10`

Absolute tolerance for the integrator

integration_tool

`ContinousSimulatorSettings.integration_tool: str = 'cvodes'`

Integration tool to be used. Options are 'cvodes' and 'idas'

reltol`ContinuousSimulatorSettings.reltol: float = 1e-10`

Relative tolerance for the integrator

t_step`ContinuousSimulatorSettings.t_step: float = None`

Timestep of the Simulator

4.10.11 sysid

Tools for machine learning and system identification.

Warning: The *ONNXConversion* class is experimental.**Classes**

<i>ONNXConversion</i>	Transform <i>ONNX</i> model.
<i>ONNXOperations</i>	CasADi operations, which are available in the <i>ONNXConversion</i> class.

4.10.11.1 ONNXConversion

class *ONNXConversion*(*model*, *model_name=None*)

Bases: object

Transform *ONNX* model. The transformation returns a CasADi expression of the model and can be used e.g. in the *do_mpc.model.Model* class.**Warning:** The feature is experimental and currently only has a limited number of supported operations. All supported operations can be found in the *ONNXOperations* class.

Other known limitations are listed at the end of this page.

How to use:

1. Create an ONNX model in your favorite framework (e.g. TensorFlow, PyTorch, Keras, ONNX).
2. Initiate the *ONNXConversion* class with the ONNX model as input.
3. Obtain information about model inputs and outputs by printing the class instance.
4. Call the *ONNXConversion.convert()* method, passing with keyword arguments the external inputs of the model. The inputs are propagated through the model and all node expressions are created.
5. Query the class instance with the respective layer or node name to obtain the CasADi expression of the respective layer or node.

Example:

We start with a simple Tensorflow (with Keras) model:

```
model_input = keras.Input(shape=(3), name='input')
hidden_layer = keras.layers.Dense(5, activation='relu', name='hidden')(model_input)
output_layer = keras.layers.Dense(1, activation='linear', name='output')(hidden_
↪layer)

keras_model = keras.Model(inputs=model_input, outputs=output_layer)
```

We then proceed to export the model in the ONNX format, using the `tf2onnx` package:

```
model_input_signature = [
    tf.TensorSpec(np.array((1, 3)), name='input'),
]
output_path = os.path.join('models', 'model.onnx')

onnx_model, _ = tf2onnx.convert.from_keras(keras_model,
    output_path=output_path,
    input_signature=model_input_signature
)
```

We can now use the ONNX model (either directly or loaded from disc) to initialize the `ONNXConversion` class:

```
casadi_converter = do_mpc.sysid.ONNXConversion(onnx_model)
```

Obtain information about the model inputs and outputs by calling `print(casadi_converter)`, yielding, in this example:

```
ONNX2Casadi model 'casadi_model'
-----
Call 'convert' by supplying the inputs with respective name and shape below.
Input shape of 'input' is (1, 3)
-----
Query the instance with the following keywords to obtain the CasADi expression of ↪
↪the respective layer or graph operation node:
- 'input'
- 'model_4/hidden/MatMul:0'
- 'model_4/hidden/Relu:0'
- 'output'
```

Call the `ONNXConversion.convert()` method, considering the name and shape of the inputs:

```
# Inputs can be numpy arrays
casadi_converter.convert(input=np.ones((1,3)))

# or CasADi expressions
x = casadi.SX.sym('x',1,3)
casadi_converter.convert(input=x)
```

Query the instance with the respective layer or node name to obtain the CasADi expression of the respective layer or node:

```
print(casadi_converter['output'])
```

Parameters

- **model** (ModelProto) – An ONNX model.
- **model_name** (Optional[str]) – Name of the model

__getitem__(key)

Enables the output of the CasADi expression of a specific layer or graph operation node.

To learn about possible keywords, it is recommended to print the instance of the class:

```
print(converter)
```

Parameters

key (str) – Name of the layer of the ONNX graph.

4.10.11.1 Methods

convert

convert(self, verbose=False, **kwargs)

Evaluate ONNX model with inputs of type `casadi.SX`, `casadi.MX`, `casadi.DM` or `numpy.ndarray`.

The keyword arguments of this method refer to the names of the inputs of the model. If these names are unknown, print the instance of the class to obtain the names.

Convert does not return anything. The converted model is stored in the instance of the class. To obtain the results of the conversion at an arbitrary internal layer, query the instance with the respective layer name. Layer names can be obtained by printing the instance of the class.

Parameters

- **verbose** – If True, prints the conversion progress.
- ****kwargs** – Keyword arguments of the method refer to the names of the inputs of the model. The values of the keyword arguments are the inputs of the model and can be of type `casadi.SX`, `casadi.MX`, `casadi.DM` or `numpy.ndarray`.

Return type

None

4.10.11.2 ONNXOperations

class ONNXOperations

Bases: `object`

CasADi operations, which are available in the [ONNXConversion](#) class. See [ONNX documentation](#) for a full list of operations.

Note: This class is not intended to be used directly. It is used by the [ONNXConversion](#) class. The purpose of this class is to provide a list of all available operations in the [ONNXConversion](#) class.

4.10.11.2.1 Methods

Add

Add(*self*, *args, attribute=None)

Addition of two or more tensors. See [ONNX documentation](#) for more details.

Concat

Concat(*self*, *args, attribute=None)

Elu

Elu(*self*, x, attribute=None)

Gemm

Gemm(*self*, *args, attribute=None)

General Matrix Multiplication. See [ONNX documentation](#) for more details.

MatMul

MatMul(*self*, *args, attribute=None)

Mul

Mul(*self*, *args, attribute=None)

Relu

Relu(*self*, x, attribute=None)

Reshape

Reshape(*self*, *args, attribute=None)

Shape

Shape(*self*, *args, attribute=None)

Sigmoid

Sigmoid(*self*, x, attribute=None)

Slice

Slice(*self*, *args, attribute=None)

Squeeze

Squeeze(*self*, *args, attribute=None)

Sub

Sub(*self*, *args, attribute=None)

Sum

Sum(*self*, *args, attribute=None)

Tanh

Tanh(*self*, x, attribute=None)

Unsqueeze

Unsqueeze(*self*, *args, attribute=None)

4.10.12 tools

Various auxiliary tools for do-mpc.

Functions

load_pickle

printProgressBar

Print a progress bar to the console.

save_pickle

4.10.12.1 load_pickle

Class method.

load_pickle(*path_to_file*)

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

4.10.12.2 printProgressBar

Class method.

printProgressBar(*iteration, total, prefix="", suffix="", decimals=1, length=100, fill="", printEnd='\r'*)

Print a progress bar to the console.

Parameters

- **iteration** (int) – Current iteration
- **total** (int) – Total iterations
- **prefix** (str) – Prefix string
- **suffix** (str) – Suffix string
- **decimals** (int) – Positive number of decimals in percent complete
- **length** (int) – Character length of bar
- **fill** (str) – Bar fill character
- **printEnd** (str) – End character

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

4.10.12.3 save_pickle

Class method.

save_pickle(*filename, data*)

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

Classes

<i>IndexedProperty</i>	Based on the python implementation of the regular property() decorator.
<i>Structure</i>	Simple structure class that can hold any type of data.
<i>Timer</i>	

4.10.12.4 IndexedProperty

class IndexedProperty(*fget=None, fset=None, doc=None*)

Bases: object

Based on the python implementation of the regular property() decorator. See for example: <https://docs.python.org/3/howto/descriptor.html>

The main tweak is `__get__`, where the above mentioned implementation directly calls the `fget` function. We instead return the class instance itself, where the parent class is now added to the class dict. Since the call is followed by brackets, immediately the `__getitem__` or `__setitem__` methods are invoked. These methods are lacking the parent class but it now exists in the scope of the property instance. We can therefore call `fget` or `fset` with the parent class.

4.10.12.4.1 Methods

getter

getter(*self, fget*)

setter

setter(*self, fset*)

4.10.12.5 Structure

class Structure

Bases: object

Simple structure class that can hold any type of data. Structure is constructed when calling `__setitem__` and can grow in complexity.

Example:

```
s = Structure()
s['_x', 'C_a'] = {'C_a_0': [1,2,3], 'C_a_1': [2,3,4]}
s['_x', 'C_b'] = 'C_b'
s['_u', 'C_a'] = 'C_a'
```

investigate the indices with `s.powerindex`. This yield the following:

```
[('_x', 'C_a', 'C_a_0', 0),
 ('_x', 'C_a', 'C_a_0', 1),
 ('_x', 'C_a', 'C_a_0', 2),
 ('_x', 'C_a', 'C_a_1', 0),
 ('_x', 'C_a', 'C_a_1', 1),
 ('_x', 'C_a', 'C_a_1', 2),
 ('_x', 'C_b'),
 ('_u', 'C_a'),
 ('_x', 'C_a', 'C_a_0', 0),
 ('_x', 'C_a', 'C_a_0', 1),
 ('_x', 'C_a', 'C_a_0', 2),
 ('_x', 'C_a', 'C_a_1'),
 ('_x', 'C_b'),
 ('_u', 'C_a')]
```

Query the structure as follows:

```
s['_x', 'C_a']
>> [1, 2, 3, 2, 3, 4]

s['_x', 'C_b']
>> [C_b]
```

Slicing is supported:

```
s['_x', 'C_a', :, 1:]
>> [[[2], [3]], [[3], [4]]]
```

and introduces nested lists for each slice element.

4.10.12.5.1 Methods

4.10.12.5.2 Attributes

full

Structure.full

Return all elements of the structure. Elements are returned in an unnested list.

get_index

Structure.get_index

Get regular indices ([0,1,2, ... N]) for the queried elements. This call mimics the `__getitem__` method but returns the indices of the queried elements instead of their values.

This is an IndexedProperty and can thus be queried as shown below:

Example:

```
# Sample structure:
s = Structure()
```

(continues on next page)

(continued from previous page)

```
s['_x', 'C_a'] = {'C_a_0': [1,2,3], 'C_a_1': [2,3,4]}
s['_x', 'C_b'] = 'C_b'
s['_u', 'C_a'] = 'C_a'

# Get indices:
s.get_index['_x', 'C_a']
s.get_index['_x', 'C_a', :, 1:]
```

The same nested list structure is obtained when using slices.

4.10.12.6 Timer

```
class Timer(name='timer', unit='ms')
```

Bases: object

4.10.12.6.1 Methods

hist

```
hist(self, *args, **kwargs)
```

info

```
info(self)
```

tic

```
tic(self)
```

toc

```
toc(self)
```

4.11 Release notes

This content is autogenerated from our Github [release notes](#).

4.11.1 v4.5.1

Please see the changes for v.4.5.0. This is a minor update to fix the installation routine of do-mpc.

4.11.2 v4.5.0

4.11.2.1 Major changes

4.11.2.1.1 Linear control

- Newly implemented class `LinearModel`. The linear model can be created by:
 - linearizing a regular (nonlinear) `Model` instance
 - passing system matrices (A, B, C, D)
 - creating (linear) equations in `LinearModel` with the well known syntax used in `Model`.
- Discretize a (continuous-time) `LinearModel`.
- Setup the new discrete-time linear quadratic regulator (LQR) controller class

4.11.2.1.2 Data-based system identification with neural networks

- Use neural networks as system models in do-mpc
 - `ONNXConversion` can convert a previously trained and stored neural network to a CasADi graph
 - ONNX files can be exported from most major neural network frameworks (e.g. tensorflow, pytorch, matlab, ...)
 - Some limitations to the neural network operations apply.

4.11.2.2 Minor changes

4.11.2.2.1 Compile NLP

- The MHE, MPC classes can now export and compile the NLP.
- Compiled NLPs can be loaded and solved. This may result in faster optimization times.

4.11.2.2.2 Improved sampling tools for data generation

- Optional parameters in `__init__` of `Sampler`, `SamplingPlanner`, `DataHandler` that are passed to `set_param`. This allows for a more concise setup.
- `SamplingPlanner` method `product` to create cartesian product (grid) of input variables to create test cases.

4.11.2.2.3 Simulator

- `Simulator.make_step()` can now be called without control input for autonomous systems.

4.11.2.3 Bug fixes

- Example files now import do-mpc relative path with `os.path.join` to yield a OS agnostic implementation.
- MHE class can now be created without estimating parameters.
- Solves visualization bug described in #340

4.11.2.4 Backend

- Significant code refactoring
 - Many modules (e.g. `controller.py`) were getting to large
 - Individual files (e.g. `_mpc.py`) in subfolder `do_mpc/controller/` for large classes
 - User-facing classes (e.g. `MPC`) imported in `do_mpc/controller/__init__.py`.
 - * Imported such that `do_mpc.controller.MPC` still yields the `MPC` class
 - * No changes in front-end for users
 - Recursive documentation with Sphinx / Readthedocs is fully automated now. Important considerations:
 - * Private files (marked as e.g. `_mpc.py`) are not documented
 - * Imported modules are documented (e.g. the imported `MPC` class).
 - * Importing with `from casadi import *` could result in documentation of CasADi package in **do-mpc**. This is avoided by:
 - Explicitly excluding certain packages (e.g. `casadi`) to be documented.
 - Marking functions or classes as private with `_Name`.
 - Using the `__all__ = [...]` variable to mark in certain files the list of elements that should be documented.

4.11.3 v4.4.0

4.11.3.1 Major changes

- MHE/MPC bounds on optimization variables can now be changed after calling `mhe.setup()` and `mpc.setup()` respectively (fixes #289). The simplest way to set bounds is the `mhe.bounds` and `mpc.bounds` property ([docs](#))
- More granular control over the bounds is now possible, e.g. choosing different values for each time-step of the horizon or for different collocation points (if that makes sense). For this purpose two new properties `lb_opt_x` and `ub_opt_x` are now documented and accessible to the user. These properties are indexed similarly to the property `opt_x`. Importantly, setting new values on these structure automatically considers the scaling factors.
- The do-mpc model can now be pickled. Pickling is restricted, however and requires (error messages are thrown otherwise):
 - the model class must be setup

- the model must use SX symbolic variables
- Enhanced warmstarting: The solver is now supplied with a guess for the dual variables

4.11.3.2 Minor changes

- Bug fix: `MPCData.prediction` was previously unable to query algebraic states `_z`.
- Fixed #283: Algebraic states can now be plotted with the graphics package

4.11.4 v4.3.5

4.11.4.1 Minor fixes

- Setup for release in v4.3.4 was incomplete.

4.11.5 v4.3.4

4.11.5.1 Minor fixes

- `model.aux` can now be queried before calling `model.setup()`
- Some typos in documentation

4.11.6 v4.3.3

4.11.6.1 Major changes

- `DataHandler` class can now create `post_processing_function` considering inputs from the case-definition as created in the `SamplingPlanner`.

4.11.6.2 Minor changes

- All cost terms that are continuously appended to are now initialised with value $DM(0)$. If nothing is appended (i.e. the term is not active), this avoids unclear error messages. This should fix #214 and #86

4.11.6.3 Documentation

Minor fixes.

4.11.6.4 Example files

Please download the example files for release v4.3.3 [here](#).

4.11.7 v4.3.2.

4.11.7.1 Major fixes

- Solved #215
- Hoping to solve #233

4.11.8 v4.3.1

Fixed an issue with release 4.3.0 where sampling tools were not included on pypi.

4.11.9 v4.3.0

4.11.9.1 Major changes

4.11.9.1.1 do-mpc sampling tools

With this release we are integrated a major new feature in do-mpc which was originally started at the [do-mpc developer conference 2021](#). To learn more about the new feature we have prepared a [video tutorial](#).

4.11.9.2 Minor changes

- Fixed an issue with saving computation time in MHE/MPC in data object.
- New example: Kite systems

4.11.9.3 Example files

Please download the example files for release v4.3.0 [here](#).

4.11.10 v4.2.5

4.11.10.1 Major changes

Full customization of the MPC or MHE optimization problem is now possible. Instead of using `MPC.setup()` to finalize the MPC optimization problem, an alternative two step process is now possible:

- `MPC.prepare_nlp()`
- `MPC.create_nlp()`

In between these two calls, users can add custom constraints and terms to the cost function using state, input etc. variables from different time-steps, collocation points or scenarios. A typical example would be to constrain changes of inputs or two enforce a cyclic behavior over the course of the horizon.

The new feature is fully documented and we suggest to have a look at the API reference for the MPC or MHE object.

4.11.10.2 Backend

4.11.10.2.1 Model

Internal functions in `do_mpc.model.Model` class have now properly named inputs and outputs. These inputs/outputs were previously automatically named `i0`, `i1`, They are now name e.g. `_x`, `_u`, `_z`

Here is an example (from the backend):

```
self._rhs_fun = Function('rhs_fun',
                        [_x, _u, _z, _tvp, _p, _w], [self._rhs],
↳   ### variables
                        ["_x", "_u", "_z", "_tvp", "_p", "_w"], ["_rhs"])  ##
↳ # names
```

This may help for debugging because we now have that:

```
model = do_mpc.model.Model("continuous")
....
model.setup()
print(model._rhs_fun)
```

Returns

```
Function(rhs_fun:(_x[6],_u,_z[3],_tvp,_p[2],_w[0])->(_rhs[6]) SXFunction)
```

4.11.11 v4.2.0

4.11.11.1 Major changes

4.11.11.1.1 MX support

This addresses #34. The `do-mpc` model class can now be created with the `symvar_type` argument, defining whether the model is using CasADi SX or MX optimization variables.

```
model = do_mpc.model.Model('continuous', 'MX')
```

all classes (MPC, MHE, Simulator ...) created from a MX model will also use MX variables. From a users-perspective the change has no significant influence on the experience. It does, however, allow for significantly faster matrix vector operations, which is main motivation to use the MX support.

The new feature resulted in some major changes to the backend. This is because CasADi does not allow (e.g.):

```
x = MX.sym('x')
struct_symMX([
    entry('x', sym=x)
])
```

on which the model configuration heavily relied on.

Most importantly:

- The Model class attributes `_x`, `_u` etc. are now dicts prior to calling `Model.setup`.
- Calling `model['x']` still works prior to calling `Model.setup` but works differently internally

- a new method `_convert2struct` converts dicts (e.g. of all the states) to symbolic structures (used in `Model.setup`). The only problem: These structs hold variables with the same name but which are different.
- a new method `_substitute_struct_vars` is introduced and substitutes the variables in the dicts in all expressions (e.g. `_rhs` with the new variables from the symbolic structs.
- the MHE also required some major internal changes. The problem is that we split the parameters (`_p`) for the MHE into estimated and set parameters. Splitting symbolic variables with the MX type is problematic.

4.11.11.2 Minor changes

- Solved #149 : Option to only have a single slack variable (for each soft-constraint) over the entire horizon

4.11.11.3 Bug fixes

- Resolves #89. Discrete-time model now inherits its properties to MHE/MPC etc.

4.11.12 v4.1.1

4.11.12.1 Major changes

4.11.12.1.1 Adapted time-varying parameters for MPC object

Time-varying parameters (tvp) are now defined for $k=0, \dots, N+1$ as opposed to $k=0, \dots, N$ in the previous version. The main consequence is that the `mterm` for `mpc.set_objective` can now include the tvp in its expression. This is beneficial e.g. for set-point tracking.

4.11.12.2 Documentation

Time-varying parameters are also described in greater detail now in [this](#) article.

4.11.13 do-mpc v4.1.0

4.11.13.1 Major changes

4.11.13.1.1 DAE support

This addresses the long overdue #3 (and closes #36). DAE works for both discrete time and continuous time formulations.

- DAE's are introduced in the model with the `set_alg` method.
- Algebraic states are introduced with the `set_variable` method and have the `var_type='_z'`.
- The model checks that for each newly introduced algebraic state there must be one new algebraic equation. Otherwise the problem is under-determined.
- Algebraic states can be scaled and bounded in both MHE and MPC similar to states, inputs etc. The algebraic equations itself are not automatically scaled. This is different for the ODE which is scaled with the scaling factor for its respective state.

Continuous time (orthogonal collocation)

When using DAEs with continuous time models the DAE equation is added as an additional constraint at each collocation point (both for MHE/MPC).

The simulator must use the `idasintegration` tool (or some other tool supporting DAEs). The default tool `cvodes` does not support DAE equations.

Discrete time

When using DAEs with continuous time models the DAE equation is added as an additional constraint at each time-step (both for MHE/MPC).

The simulator cannot simply evaluate the discrete time equation to obtain the next state as it is an expression of the unknown algebraic states. Thus we first solve the algebraic equation with the current state, input etc (using `nlpsol`) and then evaluate the discrete time equation with the obtained algebraic states.

4.11.13.1.2 Constraints with MPC / MHE with orthogonal collocation

Added a flag that can be set during MPC / MHE setup to choose whether **inequality constraints** are evaluated for each collocation point or only on the beginning of the finite Element. The flag is set during setup of the MPC / MHE with the `set_param` method:

```
mpc = do_mpc.controller.MPC(model)
setup_mpc = {
    'n_horizon': 20,
    't_step': 0.005,
    'nl_cons_check_colloc_points': True,
}
mpc.set_param(**setup_mpc)
```

Currently defaults to False.

Added a flag that can be set during MPC / MHE setup to choose whether **bounds** (lower and upper) are evaluated for each collocation point or only on the beginning of the finite Element. The flag is set during setup of the MPC / MHE with the `set_param` method:

```
mpc = do_mpc.controller.MPC(model)
setup_mpc = {
    'n_horizon': 20,
    't_step': 0.005,
    'cons_check_colloc_points': True,
}
mpc.set_param(**setup_mpc)
```

Currently defaults to True.

4.11.13.1.3 Terminal bounds for MPC

This fixes #35 .

- The MPC controller now supports terminal bounds for the states which can be different to the generic state constraints set with the `bounds` attribute.
- Set terminal bounds with the new `terminal_bounds` attribute.
- If no terminal bounds are explicitly set, they default to the regular state bounds (this means that previously working examples won't have to add terminal bounds to obtain the same results).
- If this behavior is undesired (e.g. terminal state should be unbounded even though all other states are bounded) set the parameter `use_terminal_bounds=False` during MPC setup.

4.11.13.2 Minor changes

- `MPC.set_objective`: The `mterm` (terminal cost) now allows parameters (`_p`) in the formulation.
- `Simulator.set_initial_guess`: Introduced this method to set the initial guess for the algebraic variables. The guess is based on the class attributes `z0` which is inline with how the estimator and controller work.
- `Simulator.make_step`: No longer takes the initial value/guess for `x0` and `z0` as arguments. The initial state `x0` can be changed via its class attribute whereas the initial guess for `z0` is changed as described above.
- Addressed #71 : The initial state is no longer constrained through upper and lower bounds.
- Addressed #65 and removed depreciated methods from all modules.

4.11.13.3 Documentation

- New non-linear example on the front page (double inverted pendulum with obstacle avoidance). This addresses #70.
- Fixed documentation of `MPC.opt_x_num`. This fixes #72

4.11.13.4 Example files

Please download the example files for release do-mpc v4.1.0 [here](#).

4.11.14 do-mpc v4.0.0

We are finally out of beta with **do-mpc** v4.0.0. Thanks to everyone who has contributed, for the feedback and all the interest. This release includes some important changes and bugfixes and also significantly extends our homepage [do-mpc](#).

We hope you will like the new features and content. Development will now continue with work on version 4.1.0 (and potentially some in between versions with minor features). Stay tuned on our [Github](#) page and feel free to open issues or join the discussion!

4.11.14.1 Major changes

4.11.14.1.1 New properties for Simulator, Estimator and MPC

Inheriting from the new class `IteratedVariables` these classes now obtain the attributes `_x0`, `_u0`, `_z0` (and `_p_est0`). Users can access these attributes with the properties with `x0`, `u0`, `z0` (and `p_est0`), which are listed in the documentation and have sanity checks etc. when setting them. This fixes e.g. #55. These new properties are used for two things:

Set initial values

For the simulator the initial state is self explanatory and a very important attribute. For the MHE and MPC class the attributes are used when calling the important `set_initial_guess` method, which does exactly that: Set the initial guess of the optimization problem.

Obtain the current values of the iterated variables

This is very useful for conditional MPC loops: E.g. stop the controller and simulation when a certain state has reached a certain value.

4.11.14.1.2 Measurement noise

Currently, the `do_mpc.model.Model.set_rhs` method allows to set an additive process noise. This is used for the MHE optimization problem. In a similar fashion, the `do_mpc.model.Model.set_meas` method now allows to set an **additive measurement noise**.

In the MHE the measurement noise is introduced as a new optimization variable and the measurement equation is added as an additional constraint. The full optimization problem now looks like this:

$$\begin{aligned} \min_{\substack{\mathbf{x}_{0:N}, \mathbf{u}_{0:N-1}, p, \\ \mathbf{w}_{0:N-1}, \mathbf{v}_{0:N-1}}} \quad & \frac{1}{2} \|\mathbf{x}_0 - \tilde{\mathbf{x}}_0\|_{P_x}^2 + \frac{1}{2} \|p - \tilde{p}\|_{P_p}^2 + \sum_{k=0}^{N-1} \left(\frac{1}{2} \|v_k\|_{P_{v,k}}^2 + \frac{1}{2} \|w_k\|_{P_{w,k}}^2 \right), \\ \text{s. t.} \quad & \left. \begin{aligned} \mathbf{x}_{k+1} &= f(\mathbf{x}_k, \mathbf{u}_k, \mathbf{z}_k, p, p_{\text{tv},k}) + \mathbf{w}_k, \\ y_k &= h(\mathbf{x}_k, \mathbf{u}_k, \mathbf{z}_k, p, p_{\text{tv},k}) + v_k, \\ g(\mathbf{x}_k, \mathbf{u}_k, \mathbf{z}_k, p, p_{\text{tv},k}) &\leq 0 \end{aligned} \right\} k = 0, \dots, N-1 \end{aligned}$$

This change makes it possible for the user to decide, which measurements are enforced and which can be perturbed. A typical example would be to ensure that input “measurements” are completely trusted.

4.11.14.1.3 Simulator with disturbances

The newly introduced measurement noise and the existing process noise are now used within the simulator. With each call of `Simulator.make_step` values can be passed to obtain an imperfectly simulated and measured system..

4.11.14.2 Documentation

- Release notes are now included in the documentation. They are autogenerated from the Github release notes which can be accessed via Rest API.
- The release notes are appended with a section that includes a download link for the example files that were written for the respective versions.
- Installation instructions now refer to these download links. This solves #62 .
- Added new section **Example gallery**, explaining the supplied examples in **do-mpc** in Jupyter Notebooks (rendered on readthedocs)
- Added new section **Background** with various articles explaining the mathematics behind **do-mpc**.
- Parameter `collocation_ni` in MPC/MHE is now explained more clearly.

4.11.14.3 Minor changes

- Renamed `model.setup_model()` -> `model.setup()` in all examples. This addresses #38
- `opt_p_num` and `opt_x_num` for MHE/MPC are now instance properties instead of class attributes. They still appear in the documentation and can be used as before. Having them as class attributes can lead to problems when multiple classes are live during the same session.

4.11.14.4 Example files

Please download the example files for release do-mpc v4.0.0 [here](#).

4.11.15 do-mpc v4.0.0-beta3

4.11.15.1 Major changes

4.11.15.1.1 Data

- New `__getitem__` method to conveniently retrieve values from Data object (details [here](#))
- New `MPCData` class (which inherits from `Data`). This adds the `prediction` method, which can be used to query the optimal trajectories. Details [here](#).

Both methods were previously (in a slightly different form) in the `Graphics` module. They are still used in this class but can also be convenient under different circumstances.

4.11.15.1.2 Graphics

The Graphics module is now initialized with a specific Data instance (e.g. `mpc.data`). Each Data class has their own Graphics class (if it is supposed to be displayed). Compared to the previous implementation, we now initialize all lines that are supposed to be plotted (and store them in `pred_lines` and `result_lines`). During runtime, the data on these lines is getting updated.

- Added new structure class in `do_mpc.tools`. Used for tracking the new Graphics properties: `pred_lines` and `result_lines`.
- The properties `pred_lines` and `result_lines` can be used to retrieve line instances with power indices. Line instances can be easily configured (linestyle, alpha, color etc.)

4.11.15.1.3 Process noise

Process noise can be added to rhs of Model class: [link](#)

This is solving issue #53 .

This change was necessary to allow for the more natural MHE formulation where the process noise is penalised in the cost function. The user can define for each state (vector) individually if this is intended or not.

As a consequence of this change I had to introduce the new variable `w` throughout **do-mpc**. For the MPC and simulator module this is without effect.

The main difference is [here](#)

Remark: The change also allows to estimate parameters that change over time (e.g. environmental influences). Our regular estimated parameters are constant over the entire MHE horizon, which is not always valid. To estimate varying parameters, they should be defined as states with unknown dynamics. Concretely, their RHS is zero (for ODEs) and they have a high process noise.

4.11.15.1.4 Symbolic variables for MHE weighting matrices

As originally intended, it is now possible to have symbolic matrices as MHE tuning factors. The result of this change can be seen in the `rotating_oscillating_masses` example.

The symbolic variables are defined in the **do-mpc** Model where typically, you want to have `P_x` and `P_p` as parameters and `P_y` and `P_w` as time-varying parameters. Example of their [definition](#).

and [here](#) they are used.

The purpose of using symbolic weighting is of course to update them at each iteration. Since they are parameters and time-varying parameters respectively, this is done with the `set_p_fun` and `set_tvp_fun` method of the MHE: [link](#)

Note that in the example above, we don't actually need varying weighting matrices and the returned values are in fact constant. This can be seen as a proof of concept.

This change had some other implications. Most notably, having additional parameters interferes with the multi-stage robust MPC module. Where we previously had to pass a number of scenarios for each defined parameter. Since parameters for the MHE are irrelevant for MPC the API for the call `set_uncertainty_values` has changed: [link](#)

The new API is fully backwards compatible. However, it is much more intuitive now. The function is called with keyword arguments, where each keyword refers to one uncertain parameter (note that we can ignore the parameters that are irrelevant). In practice this looks something like [this](#)

4.11.15.2 Example files

Please download the example files for release do-mpc v4.0.0-beta3 [here](#).

4.11.16 do-mpc v4.0.0-beta2

Error in release. Immediately replaced with beta3.

4.11.17 do-mpc v4.0.0-beta1

4.11.17.1 Major changes

- We are now explicitly pointing out attributes of the `Model` such as states, inputs, etc. These should be used to obtain these attributes and replace the previous `get_variables` method which is now depreciated. The `Model` also supports a `__get_variable__` call now to conveniently select items.
- `setup_model` is replaced by `setup` to be more consistent with other setup methods. The old method is still available and shows a depreciation warning.
- The MHE now supports the `set_default_objective` method.

4.11.17.2 Bug fixes

- The MHE formulation had an error in the `make_step` method. We used the wrong time step from the previous solution to compute the arrival cost.

4.11.17.3 Other changes

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

4.11.17.4 Example files

Please download the example files for release do-mpc v4.0.0-beta1 [here](#).

4.11.18 do-mpc v4.0.0-beta

do-mpc has undergone a massive overhaul and comes with a completely new interface, new features and a comprehensive documentation.

Please note that previously written code is not compatible with **do-mpc** 4.0.0. If you want to continue working with older code please use version 3.0.0.

This is the beta release of version 4.0.0. We expect minor modifications and bug fixes in the near future.

Please see our documentation on our new project homepage www.do-mpc.com for a full list of features.

4.11.18.1 Example files

Please download the example files for release do-mpc v4.0.0-beta [here](#).

4.11.19 do-mpc v3.0.0

4.11.19.1 Main modifications

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

4.11.20 do-mpc v2.0.0

Compatible with CasADi 3.0.0

4.11.21 do-mpc version 1.0.0

4.12 Batch Bioreactor

In this Jupyter Notebook we illustrate the example **batch_reactor**.

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

The example consists of the three modules **template_model.py**, which describes the system model, **template_mpc.py**, which defines the settings for the control and **template_simulator.py**, which sets the parameters for the simulator. The modules are used in **main.py** for the closed-loop execution of the controller.

In the following the different parts are presented. But first, we start by importing basic modules and **do-mpc**.

```
[1]: import numpy as np
import sys
from casadi import *

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

# Import do_mpc package:
import do_mpc
```

4.12.1 Model

In the following we will present the configuration, setup and connection between these blocks, starting with the model. The considered model of the batch bioreactor is continuous and has 4 states and 1 control input, which are depicted below:

The model is initiated by:

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

4.12.1.1 States and control inputs

The four states are concentration of the biomass X_s , the concentration of the substrate S_s , the concentration of the product P_s and the volume V_s :

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

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

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

4.12.1.2 ODE and parameters

The system model is described by the ordinary differential equation:

$$\dot{X}_s = \mu(S_s)X_s - \frac{u_{\text{inp}}}{V_s}X_s, \quad (4.23)$$

$$\dot{S}_s = -\frac{\mu(S_s)X_s}{Y_x} - \frac{vX_s}{Y_p} + \frac{u_{\text{inp}}}{V_s}(S_{\text{in}} - S_s), \quad (4.24)$$

$$\dot{P}_s = vX_s - \frac{u_{\text{inp}}}{V_s}P_s, \quad (4.25)$$

$$\dot{V}_s = u_{\text{inp}}, \quad (4.26)$$

$$(4.27)$$

where:

$$\mu(S_s) = \frac{\mu_m S_s}{K_m + S_s + (S_s^2/K_i)}, \quad (4.28)$$

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

```
[5]: # Certain parameters
mu_m = 0.02
K_m = 0.05
```

(continues on next page)

(continued from previous page)

```

K_i    = 5.0
v_par  = 0.004
Y_p    = 1.2

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

```

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

```

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

# Differential equations
model.set_rhs('X_s', mu_S*X_s - inp/V_s*X_s)
model.set_rhs('S_s', -mu_S*X_s/Y_x - v_par*X_s/Y_p + inp/V_s*(S_in-S_s))
model.set_rhs('P_s', v_par*X_s - inp/V_s*P_s)
model.set_rhs('V_s', inp)

```

Finally, the model setup is completed:

```

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

```

4.12.2 Controller

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

```

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

```

We choose the prediction horizon `n_horizon`, set the robust horizon `n_robust` to 3. The time step `t_step` is set to one second and parameters of the applied discretization scheme orthogonal collocation are as seen below:

```

[9]: setup_mpc = {
    'n_horizon': 20,
    'n_robust': 1,
    'open_loop': 0,
    't_step': 1.0,
    'state_discretization': 'collocation',
    'collocation_type': 'radau',
    'collocation_deg': 2,
    'collocation_ni': 2,
    'store_full_solution': True,
    # Use MA27 linear solver in ipopt for faster calculations:
    #nlpsol_opts: {'ipopt.linear_solver': 'MA27'}
}

mpc.set_param(**setup_mpc)

```

4.12.2.1 Objective

The batch bioreactor is used to produce penicillin. Hence, the objective of the controller is to maximize the concentration of the product P_s . Additionally, we add a penalty on input changes, to obtain a smooth control performance.

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

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

4.12.2.2 Constraints

In the next step, the constraints of the control problem are set. In this case, there are only upper and lower bounds for each state and the input:

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

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

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

4.12.2.3 Uncertain values

The explicit values of the two uncertain parameters Y_x and S_{in} , which are considered in the scenario tree, are given by:

```
[12]: Y_x_values = np.array([0.5, 0.4, 0.3])
      S_in_values = np.array([200.0, 220.0, 180.0])

      mpc.set_uncertainty_values(Y_x = Y_x_values, S_in = S_in_values)
```

This means with $n_{\text{robust}}=1$, that 9 different scenarios are considered. The setup of the MPC controller is concluded by:

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

4.12.3 Estimator

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

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

4.12.4 Simulator

To create a simulator in order to run the MPC in a closed-loop, we create an instance of the **do-mpc** simulator which is based on the same model:

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

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

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

simulator.set_param(**params_simulator)
```

4.12.4.1 Realizations of uncertain parameters

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

```
[17]: p_num = simulator.get_p_template()
```

We define a function which is called in each simulation step, which gives the current realization of the uncertain parameters, with respect to defined inputs (in this case `t_now`):

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

# function definition
def p_fun(t_now):
    return p_num

# Set the user-defined function above as the function for the realization of the
# uncertain parameters
simulator.set_p_fun(p_fun)
```

By defining `p_fun` as above, the function will always return the same values. To finish the configuration of the simulator, call:

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

4.12.5 Closed-loop simulation

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

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

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

4.12.5.1 Prepare visualization

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

```
[21]: import matplotlib.pyplot as plt
plt.ion()
from matplotlib import rcParams
rcParams['text.usetex'] = True
rcParams['text.latex.preamble'] = [r'\usepackage{amsmath}', r'\usepackage{siunitx}]
rcParams['axes.grid'] = True
rcParams['lines.linewidth'] = 2.0
rcParams['axes.labelsize'] = 'xx-large'
rcParams['xtick.labelsize'] = 'xx-large'
rcParams['ytick.labelsize'] = 'xx-large'
```

We use the plotting capabilities, which are included in **do-mpc**. The `mpc_graphics` contain information like the current estimated state and the predicted trajectory of the states and inputs based on the solution of the control problem. The `sim_graphics` contain the information about the simulated evaluation of the system.

```
[22]: mpc_graphics = do_mpc.graphics.Graphics(mpc.data)
sim_graphics = do_mpc.graphics.Graphics(simulator.data)
```

A figure containing the 4 states and the control input are created:

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

for g in [sim_graphics, mpc_graphics]:
    # Plot the state on axis 1 to 4:
    g.add_line(var_type='_x', var_name='X_s', axis=ax[0], color='#1f77b4')
    g.add_line(var_type='_x', var_name='S_s', axis=ax[1], color='#1f77b4')
    g.add_line(var_type='_x', var_name='P_s', axis=ax[2], color='#1f77b4')
    g.add_line(var_type='_x', var_name='V_s', axis=ax[3], color='#1f77b4')
```

(continues on next page)

(continued from previous page)

```

# Plot the control input on axis 5:
g.add_line(var_type='_u', var_name='inp', axis=ax[4], color='#1f77b4')

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

```

4.12.5.2 Run closed-loop

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

```

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

```

4.12.5.3 Results

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

```

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

# The function describing the gif:
def update(t_ind):
    sim_graphics.plot_results(t_ind)
    mpc_graphics.plot_predictions(t_ind)
    mpc_graphics.reset_axes()

if False:
    anim = FuncAnimation(fig, update, frames=n_steps, repeat=False)
    gif_writer = ImageMagickWriter(fps=10)
    anim.save('anim_batch_reactor_final.gif', writer=gif_writer)

```

The result is shown below, where solid lines are the recorded trajectories and dashed lines are the predictions of the scenarios:

4.13 Continuous stirred tank reactor (CSTR)

In this Jupyter Notebook we illustrate the example CSTR.

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

The example consists of the three modules **template_model.py**, which describes the system model, **template_mpc.py**, which defines the settings for the control and **template_simulator.py**, which sets the parameters for the simulator. The modules are used in **main.py** for the closed-loop execution of the controller. The file **post_processing.py** is used for the visualization of the closed-loop control run. One exemplary result will be presented at the end of this tutorial as a gif.

In the following the different parts are presented. But first, we start by importing basic modules and **do-mpc**.

```
[1]: import numpy as np
import sys
from casadi import *

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

# Import do_mpc package:
import do_mpc

import matplotlib.pyplot as plt
```

4.13.1 Model

In the following we will present the configuration, setup and connection between these blocks, starting with the model. The considered model of the CSTR is continuous and has 4 states and 2 control inputs. The model is initiated by:

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

4.13.1.1 States and control inputs

The four states are concentration of reactant A (C_A), the concentration of reactant B (C_B), the temperature inside the reactor (T_R) and the temperature of the cooling jacket (T_K):

```
[3]: # States struct (optimization variables):
C_a = model.set_variable(var_type='_x', var_name='C_a', shape=(1,1))
C_b = model.set_variable(var_type='_x', var_name='C_b', shape=(1,1))
T_R = model.set_variable(var_type='_x', var_name='T_R', shape=(1,1))
T_K = model.set_variable(var_type='_x', var_name='T_K', shape=(1,1))
```

The control inputs are the feed F and the heat flow \dot{Q} :


```
[4]: # Input struct (optimization variables):
F = model.set_variable(var_type='_u', var_name='F')
Q_dot = model.set_variable(var_type='_u', var_name='Q_dot')
```

4.13.1.2 ODE and parameters

The system model is described by the ordinary differential equation:

$$\dot{C}_A = F \cdot (C_{A,0} - C_A) - k_1 \cdot C_A - k_3 \cdot C_A^2, \quad (4.29)$$

$$\dot{C}_B = -F \cdot C_B + k_1 \cdot C_A - k_2 \cdot C_B, \quad (4.30)$$

$$\dot{T}_R = \frac{k_1 \cdot C_A \cdot H_{R,ab} + k_2 \cdot C_B \cdot H_{R,bc} + k_3 \cdot C_A^2 \cdot H_{R,ad} - \rho \cdot c_p}{\rho \cdot c_p \cdot V_R} \quad (4.31)$$

$$+ F \cdot (T_{in} - T_R) + \frac{K_w \cdot A_R \cdot (T_K - T_R)}{\rho \cdot c_p \cdot V_R}, \quad (4.32)$$

$$\dot{T}_K = \frac{\dot{Q} + k_w \cdot A_R \cdot T_{dif}}{m_k \cdot C_{p,k}}, \quad (4.33)$$

where

$$k_1 = \beta \cdot k_{0,ab} \cdot \exp\left(\frac{-E_{A,ab}}{T_R + 273.15}\right), \quad (4.34)$$

$$k_2 = k_{0,bc} \cdot \exp\left(\frac{-E_{A,bc}}{T_R + 273.15}\right), \quad (4.35)$$

$$k_3 = k_{0,ad} \cdot \exp\left(\frac{-\alpha \cdot E_{A,ad}}{T_R + 273.15}\right). \quad (4.36)$$

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

```
[5]: # Certain parameters
K0_ab = 1.287e12 # K0 [h^-1]
K0_bc = 1.287e12 # K0 [h^-1]
K0_ad = 9.043e9 # K0 [l/mol.h]
R_gas = 8.3144621e-3 # Universal gas constant
E_A_ab = 9758.3*1.00 #* R_gas# [kj/mol]
E_A_bc = 9758.3*1.00 #* R_gas# [kj/mol]
E_A_ad = 8560.0*1.0 #* R_gas# [kj/mol]
H_R_ab = 4.2 # [kj/mol A]
H_R_bc = -11.0 # [kj/mol B] Exothermic
H_R_ad = -41.85 # [kj/mol A] Exothermic
Rou = 0.9342 # Density [kg/l]
Cp = 3.01 # Specific Heat capacity [kj/Kg.K]
Cp_k = 2.0 # Coolant heat capacity [kj/kg.k]
A_R = 0.215 # Area of reactor wall [m^2]
V_R = 10.01 #0.01 # Volume of reactor [l]
m_k = 5.0 # Coolant mass[kg]
T_in = 130.0 # Temp of inflow [Celsius]
K_w = 4032.0 # [kj/h.m^2.K]
C_A0 = (5.7+4.5)/2.0*1.0 # Concentration of A in input Upper bound 5.7 lower bound 4.5
    ↪ [mol/l]

# Uncertain parameters:
```

(continues on next page)

(continued from previous page)

```
alpha = model.set_variable(var_type='_p', var_name='alpha')
beta = model.set_variable(var_type='_p', var_name='beta')
```

In the next step, we formulate the k_i -s:

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

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

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

With the help of the k_i -s and T_{dif} we can define the ODEs:

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

Finally, the model setup is completed:

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

4.13.2 Controller

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

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

We choose the prediction horizon n_{horizon} , set the robust horizon n_{robust} to 1. The time step t_{step} is set to one second and parameters of the applied discretization scheme orthogonal collocation are as seen below:

```
[11]: setup_mpc = {
    'n_horizon': 20,
    'n_robust': 1,
    'open_loop': 0,
    't_step': 0.005,
    'state_discretization': 'collocation',
    'collocation_type': 'radau',
    'collocation_deg': 2,
    'collocation_ni': 2,
    'store_full_solution': True,
    # Use MA27 linear solver in ipopt for faster calculations:
    #nlpsol_opts': {'ipopt.linear_solver': 'MA27'}
}
```

(continues on next page)

(continued from previous page)

```
mpc.set_param(**setup_mpc)
```

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

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

4.13.2.1 Objective

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

```
[13]: _x = model.x
      mterm = (_x['C_b'] - 0.6)**2 # terminal cost
      lterm = (_x['C_b'] - 0.6)**2 # stage cost

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

      mpc.set_rterm(F=0.1, Q_dot = 1e-3) # input penalty
```

4.13.2.2 Constraints

In the next step, the constraints of the control problem are set. In this case, there are only upper and lower bounds for each state and the input:

```
[14]: # lower bounds of the states
      mpc.bounds['lower', '_x', 'C_a'] = 0.1
      mpc.bounds['lower', '_x', 'C_b'] = 0.1
      mpc.bounds['lower', '_x', 'T_R'] = 50
      mpc.bounds['lower', '_x', 'T_K'] = 50

      # upper bounds of the states
      mpc.bounds['upper', '_x', 'C_a'] = 2
      mpc.bounds['upper', '_x', 'C_b'] = 2
      mpc.bounds['upper', '_x', 'T_K'] = 140

      # lower bounds of the inputs
      mpc.bounds['lower', '_u', 'F'] = 5
      mpc.bounds['lower', '_u', 'Q_dot'] = -8500

      # upper bounds of the inputs
      mpc.bounds['upper', '_u', 'F'] = 100
      mpc.bounds['upper', '_u', 'Q_dot'] = 0.0
```

If a constraint is not critical, it is possible to implement it as a **soft** constraint. This means, that a small violation of the constraint does not render the optimization infeasible. Instead, a penalty term is added to the objective. **Soft** constraints can always be applied, if small violations can be accepted and it might even be necessary to apply MPC in a safe way

(by avoiding avoiding numerical instabilities). In this case, we define the upper bounds of the reactor temperature as a **soft** constraint by using `mpc.set_nl_cons()`.

```
[15]: mpc.set_nl_cons('T_R', _x['T_R'], ub=140, soft_constraint=True, penalty_term_cons=1e2)
[15]: SX((T_R-eps_T_R))
```

4.13.2.3 Uncertain values

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

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

      mpc.set_uncertainty_values(alpha = alpha_var, beta = beta_var)
```

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

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

4.13.3 Estimator

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

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

4.13.4 Simulator

To create a simulator in order to run the MPC in a closed-loop, we create an instance of the **do-mpc** simulator which is based on the same model:

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

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

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

      simulator.set_param(**params_simulator)
```

4.13.4.1 Realizations of uncertain parameters

For the simulation, it is necessary to define the numerical realizations of the uncertain parameters in `p_num` and the time-varying parameters in `tv_p_num`. First, we get the structure of the uncertain and time-varying parameters:

```
[21]: p_num = simulator.get_p_template()
      tv_p_num = simulator.get_tv_p_template()
```

We define two functions which are called in each simulation step, which return the current realizations of the parameters, with respect to defined inputs (in this case `t_now`):

```
[22]: # function for time-varying parameters
      def tv_p_fun(t_now):
          return tv_p_num

      # uncertain parameters
      p_num['alpha'] = 1
      p_num['beta'] = 1
      def p_fun(t_now):
          return p_num
```

These two custom functions are used in the simulation via:

```
[23]: simulator.set_tv_p_fun(tv_p_fun)
      simulator.set_p_fun(p_fun)
```

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

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

4.13.5 Closed-loop simulation

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

```
[25]: # Set the initial state of mpc, simulator and estimator:
      C_a_0 = 0.8 # This is the initial concentration inside the tank [mol/l]
      C_b_0 = 0.5 # This is the controlled variable [mol/l]
      T_R_0 = 134.14 #[C]
      T_K_0 = 130.0 #[C]
      x0 = np.array([C_a_0, C_b_0, T_R_0, T_K_0]).reshape(-1,1)

      mpc.x0 = x0
      simulator.x0 = x0
      estimator.x0 = x0

      mpc.set_initial_guess()
```

Now, we simulate the closed-loop for 50 steps (and suppress the output of the cell with the magic command `%%capture`):

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

4.13.6 Animating the results

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

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

We quickly configure Matplotlib.

```
[28]: from matplotlib import rcParams
rcParams['axes.grid'] = True
rcParams['font.size'] = 18
```

We then create a figure, configure which lines to plot on which axis and add labels.

```
[29]: %%capture
fig, ax = plt.subplots(5, sharex=True, figsize=(16,12))
# Configure plot:
mpc_graphics.add_line(var_type='_x', var_name='C_a', axis=ax[0])
mpc_graphics.add_line(var_type='_x', var_name='C_b', axis=ax[0])
mpc_graphics.add_line(var_type='_x', var_name='T_R', axis=ax[1])
mpc_graphics.add_line(var_type='_x', var_name='T_K', axis=ax[1])
mpc_graphics.add_line(var_type='_aux', var_name='T_dif', axis=ax[2])
mpc_graphics.add_line(var_type='_u', var_name='Q_dot', axis=ax[3])
mpc_graphics.add_line(var_type='_u', var_name='F', axis=ax[4])
ax[0].set_ylabel('c [mol/l]')
ax[1].set_ylabel('T [K]')
ax[2].set_ylabel('$\Delta$ T [K]')
ax[3].set_ylabel('Q [kW]')
ax[4].set_ylabel('Flow [l/h]')
ax[4].set_xlabel('time [h]')
```

Some “cosmetic” modifications are easily achieved with the structure `pred_lines` and `result_lines`.

```
[30]: # Update properties for all prediction lines:
for line_i in mpc_graphics.pred_lines.full:
    line_i.set_linewidth(2)
# Highlight nominal case:
for line_i in np.sum(mpc_graphics.pred_lines['_x', :, :, 0]):
    line_i.set_linewidth(5)
for line_i in np.sum(mpc_graphics.pred_lines['_u', :, :, 0]):
    line_i.set_linewidth(5)
for line_i in np.sum(mpc_graphics.pred_lines['_aux', :, :, 0]):
    line_i.set_linewidth(5)

# Add labels
label_lines = mpc_graphics.result_lines['_x', 'C_a'] + mpc_graphics.result_lines['_x', 'C_b'
```

(continues on next page)

(continued from previous page)

```

↪']
ax[0].legend(label_lines, ['C_a', 'C_b'])
label_lines = mpc_graphics.result_lines['_x', 'T_R']+mpc_graphics.result_lines['_x', 'T_K']
↪']
ax[1].legend(label_lines, ['T_R', 'T_K'])

fig.align_ylabels()

```

After importing the necessary package:

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

We obtain the animation with:

```
[32]: def update(t_ind):
    print('Writing frame: {}'.format(t_ind), end='\r')
    mpc_graphics.plot_results(t_ind=t_ind)
    mpc_graphics.plot_predictions(t_ind=t_ind)
    mpc_graphics.reset_axes()
    lines = mpc_graphics.result_lines.full
    return lines

n_steps = mpc.data['_time'].shape[0]

anim = FuncAnimation(fig, update, frames=n_steps, blit=True)

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

Writing frame: 49.
```

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

4.14 Industrial polymerization reactor

In this Jupyter Notebook we illustrate the example `industrial_poly`.

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

The example consists of the three modules `template_model.py`, which describes the system model, `template_mpc.py`, which defines the settings for the control and `template_simulator.py`, which sets the parameters for the simulator. The modules are used in `main.py` for the closed-loop execution of the controller.

In the following the different parts are presented. But first, we start by importing basic modules and `do-mpc`.

```
[29]: import numpy as np
import matplotlib.pyplot as plt
import sys
```

(continues on next page)

(continued from previous page)

```

from casadi import *

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

# Import do_mpc package:
import do_mpc

```

4.14.1 Model

In the following we will present the configuration, setup and connection between these blocks, starting with the model. The considered model of the industrial reactor is continuous and has 10 states and 3 control inputs. The model is initiated by:

```

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

```

4.14.1.1 System description

The system consists of a reactor into which monomer is fed. The monomer turns into a polymer via a very exothermic chemical reaction. The reactor is equipped with a jacket and with an External Heat Exchanger (EHE) that can both be used to control the temperature inside the reactor. A schematic representation of the system is presented below:

The process is modeled by a set of 8 ordinary differential equations (ODEs):

$$\dot{m}_W = \dot{m}_F \omega_{W,F} \quad (4.37)$$

$$\dot{m}_A = \dot{m}_F \omega_{A,F} - k_{R1} m_{A,R} - k_{R2} m_{AWT} m_A / m_{ges}, \quad (4.38)$$

$$\dot{m}_P = k_{R1} m_{A,R} + p_1 k_{R2} m_{AWT} m_A / m_{ges}, \quad (4.39)$$

$$\begin{aligned} \dot{T}_R = 1/(c_{p,R} m_{ges}) [\dot{m}_F c_{p,F} (T_F - T_R) + \Delta H_R k_{R1} m_{A,R} - k_K A (T_R - T_S) \\ - \dot{m}_{AWT} c_{p,R} (T_R - T_{EK})], \end{aligned} \quad (4.40)$$

$$\dot{T}_S = 1/(c_{p,S} m_S) [k_K A (T_R - T_S) - k_K A (T_S - T_M)],$$

$$\dot{T}_M = 1/(c_{p,W} m_{M,KW}) [\dot{m}_{M,KW} c_{p,W} (T_M^{IN} - T_M) + k_K A (T_S - T_M)] + k_K A (T_S - T_M), \quad (4.41)$$

$$\begin{aligned} \dot{T}_{EK} = 1/(c_{p,R} m_{AWT}) [\dot{m}_{AWT} c_{p,W} (T_R - T_{EK}) - \alpha (T_{EK} - T_{AWT}) \\ + k_{R2} m_A m_{AWT} \Delta H_R / m_{ges}], \end{aligned} \quad (4.43)$$

$$\dot{T}_{AWT} = [\dot{m}_{AWT,KW} c_{p,W} (T_{AWT}^{IN} - T_{AWT}) - \alpha (T_{AWT} - T_{EK})] / (c_{p,W} m_{AWT,KW}), \quad (4.44)$$

where

$$U = m_P / (m_A + m_P), \quad (4.45)$$

$$m_{\text{ges}} = m_W + m_A + m_P, \quad (4.46)$$

$$k_{R1} = k_0 e^{\frac{-E_a}{R(T_R + 273.15)}} (k_{U1} (1 - U) + k_{U2} U), \quad (4.47)$$

$$k_{R2} = k_0 e^{\frac{-E_a}{R(T_{EK} + 273.15)}} (k_{U1} (1 - U) + k_{U2} U), \quad (4.48)$$

$$k_K = (m_W k_{WS} + m_A k_{AS} + m_P k_{PS}) / m_{\text{ges}}, \quad (4.49)$$

$$m_{A,R} = m_A - m_A m_{AWT} / m_{\text{ges}}. \quad (4.50)$$

The model includes mass balances for the water, monomer and product hold-ups (m_W , m_A , m_P) and energy balances for the reactor (T_R), the vessel (T_S), the jacket (T_M), the mixture in the external heat exchanger (T_{EK}) and the coolant leaving the external heat exchanger (T_{AWT}). The variable U denotes the polymer-monomer ratio in the reactor, m_{ges} represents the total mass, k_{R1} is the reaction rate inside the reactor and k_{R2} is the reaction rate in the external heat exchanger. The total heat transfer coefficient of the mixture inside the reactor is denoted as k_K and $m_{A,R}$ represents the current amount of monomer inside the reactor.

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

An overview of the parameters are listed below:

Parameter	Description	Value	Units
R	Gas constant	8.314	$\text{kJ kmol}^{-1} \text{K}^{-1}$
$c_{p,W}$	Specific heat capacity of the coolant	4.2	$\text{kJ kg}^{-1} \text{K}^{-1}$
$c_{p,S}$	Specific heat capacity of the steel	0.47	$\text{kJ kg}^{-1} \text{K}^{-1}$
$c_{p,F}$	Specific heat capacity of the feed	3	$\text{kJ kg}^{-1} \text{K}^{-1}$
$c_{p,R}$	Specific heat capacity of the reactor contents	5.0	$\text{kJ kg}^{-1} \text{K}^{-1}$
k_{WS}	Heat transfer coeff. water-steel	4800	$\text{W m}^{-2} \text{K}^{-1}$
T_F	Feed temperature	25	$^{\circ}\text{C}$
A	Heat exchange surface of the jacket	65	m^2
$m_{M,KW}$	Mass of coolant in the jacket	5000	kg
m_S	Mass of reactor steel	39,000	kg
m_{AWT}	Mass of the product in the EHE	200	kg
$m_{AWT,KW}$	Mass of the coolant in the EHE	1000	kg
$\dot{m}_{M,KW}$	Coolant flow of the jacket	300,000	kg h^{-1}
$\dot{m}_{AWT,KW}$	Coolant flow of the EHE	100,000	kg h^{-1}
\dot{m}_{AWT}	Product flow to the EHE	20,000	kg h^{-1}
E_a	Activation energy	8500	kJ kmol^{-1}
ΔH_R	Specific reaction enthalpy	950	kJ kg^{-1}
k_0	Specific reaction rate	7	–
k_{U2}	Reaction parameter 1	32	–
k_{U1}	Reaction parameter 2	4	–
$w_{W,F}$	Mass fraction of water in the feed	0.333	–
$w_{A,F}$	Mass fraction of A in the feed	0.667	–
k_{AS}	Heat transfer coeff. monomer-steel	1000	$\text{W m}^{-2} \text{K}^{-1}$
k_{PS}	Heat transfer coeff. product-steel	100	$\text{W m}^{-2} \text{K}^{-1}$
α	Experimental coefficient	3,600,000	s^{-1}

4.14.1.2 Implementation

First, we set the certain parameters:

```
[3]: # Certain parameters
R      = 8.314           #gas constant
T_F    = 25 + 273.15    #feed temperature
E_a    = 8500.0          #activation energy
delH_R = 950.0*1.00      #sp reaction enthalpy
A_tank = 65.0           #area heat exchanger surface jacket 65

k_0    = 7.0*1.00       #sp reaction rate
k_U2   = 32.0           #reaction parameter 1
```

(continues on next page)

(continued from previous page)

```

k_U1      = 4.0          #reaction parameter 2
w_WF      = .333        #mass fraction water in feed
w_AF      = .667        #mass fraction of A in feed

m_M_KW     = 5000.0      #mass of coolant in jacket
fm_M_KW    = 300000.0    #coolant flow in jacket 300000;
m_AWT_KW   = 1000.0      #mass of coolant in EHE
fm_AWT_KW  = 100000.0    #coolant flow in EHE
m_AWT      = 200.0       #mass of product in EHE
fm_AWT     = 20000.0     #product flow in EHE
m_S        = 39000.0     #mass of reactor steel

c_pW       = 4.2         #sp heat cap coolant
c_pS       = .47         #sp heat cap steel
c_pF       = 3.0         #sp heat cap feed
c_pR       = 5.0         #sp heat cap reactor contents

k_WS       = 17280.0     #heat transfer coeff water-steel
k_AS       = 3600.0      #heat transfer coeff monomer-steel
k_PS       = 360.0       #heat transfer coeff product-steel

alfa       = 5*20e4*3.6

p_1        = 1.0

```

and afterwards the uncertain parameters:

```

[4]: # Uncertain parameters:
delH_R = model.set_variable('_p', 'delH_R')
k_0 = model.set_variable('_p', 'k_0')

```

The 10 states of the control problem stem from the 8 ODEs, `accum_monom` models the amount that has been fed to the reactor via $\dot{m}_F^{\text{acc}} = \dot{m}_F$ and T_{adiab} ($T_{\text{adiab}} = \frac{\Delta H_R}{c_{p,R}} \frac{m_A}{m_{\text{ges}}} + T_R$, hence $\dot{T}_{\text{adiab}} = \frac{\Delta H_R}{m_{\text{ges}} c_{p,R}} \dot{m}_A - (\dot{m}_W + \dot{m}_A + \dot{m}_P) \left(\frac{m_A \Delta H_R}{m_{\text{ges}}^2 c_{p,R}} \right) + \dot{T}_R$) is a virtual variable that is important for safety aspects, as we will explain later. All states are created in **do-mpc** via:

```

[5]: # States struct (optimization variables):
m_W = model.set_variable('_x', 'm_W')
m_A = model.set_variable('_x', 'm_A')
m_P = model.set_variable('_x', 'm_P')
T_R = model.set_variable('_x', 'T_R')
T_S = model.set_variable('_x', 'T_S')
Tout_M = model.set_variable('_x', 'Tout_M')
T_EK = model.set_variable('_x', 'T_EK')
Tout_AWT = model.set_variable('_x', 'Tout_AWT')
accum_monom = model.set_variable('_x', 'accum_monom')
T_adiab = model.set_variable('_x', 'T_adiab')

```

and the control inputs via:

```

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

```

(continues on next page)

(continued from previous page)

```
T_in_M = model.set_variable('_u', 'T_in_M')
T_in_EK = model.set_variable('_u', 'T_in_EK')
```

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

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

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

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

dot_T_R = 1./(c_pR * m_ges) * ((m_dot_f * c_pF * (T_F - T_R)) - (k_K * A_tank * (T_R - T_
↳ S)) - (fm_AWT * c_pR * (T_R - T_EK)) + (delH_R * k_R1 * (m_A - ((m_A*m_AWT)/(m_W+m_A+m_
↳ P)))))
model.set_rhs('T_R', dot_T_R)
model.set_rhs('T_S', 1./(c_pS * m_S) * ((k_K * A_tank * (T_R - T_S)) - (k_K * A_tank *
↳ (T_S - Tout_M))))
model.set_rhs('Tout_M', 1./(c_pW * m_M_KW) * ((fm_M_KW * c_pW * (T_in_M - Tout_M)) + (k_
↳ K * A_tank * (T_S - Tout_M)))
model.set_rhs('T_EK', 1./(c_pR * m_AWT) * ((fm_AWT * c_pR * (T_R - T_EK)) - (alfa * (T_
↳ EK - Tout_AWT)) + (p_1 * k_R2 * (m_A/m_ges) * m_AWT * delH_R)))
model.set_rhs('Tout_AWT', 1./(c_pW * m_AWT_KW) * ((fm_AWT_KW * c_pW * (T_in_EK - Tout_
↳ AWT)) - (alfa * (Tout_AWT - T_EK))))
model.set_rhs('accum_monom', m_dot_f)
model.set_rhs('T_adiab', delH_R/(m_ges*c_pR)*dot_m_A-(dot_m_A+dot_m_W+dot_m_P)*(m_A*delH_
↳ R/(m_ges*m_ges*c_pR))+dot_T_R)
```

Finally, the model setup is completed:

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

4.14.2 Controller

Next, the model predictive controller is configured (in `template_mpc.py`). First, one member of the `mpc` class is generated with the prediction model defined above:

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

Real processes are also subject to important safety constraints that are incorporated to account for possible failures of the equipment. In this case, the maximum temperature that the reactor would reach in the case of a cooling failure is constrained to be below 109°C. The temperature that the reactor would achieve in the case of a complete cooling failure is T_{adiab} , hence it needs to stay beneath 109°C.

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

```
[11]: setup_mpc = {
    'n_horizon': 20,
    'n_robust': 1,
    'open_loop': 0,
    't_step': 50.0/3600.0,
    'state_discretization': 'collocation',
    'collocation_type': 'radau',
    'collocation_deg': 2,
    'collocation_ni': 2,
    'store_full_solution': True,
    # Use MA27 linear solver in ipopt for faster calculations:
    #nlpsol_opts': {'ipopt.linear_solver': 'MA27'}
}

mpc.set_param(**setup_mpc)
```

4.14.2.1 Objective

The goal of the economic NMPC controller is to produce 20680 kg of m_P as fast as possible. Additionally, we add a penalty on input changes for all three control inputs, to obtain a smooth control performance.

```
[12]: _x = model.x
mterm = - _x['m_P'] # terminal cost
lterm = - _x['m_P'] # stage cost

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

mpc.set_rterm(m_dot_f=0.002, T_in_M=0.004, T_in_EK=0.002) # penalty on control input,
↪ changes
```

4.14.2.2 Constraints

The temperature at which the polymerization reaction takes place strongly influences the properties of the resulting polymer. For this reason, the temperature of the reactor should be maintained in a range of $\pm 2.0^\circ\text{C}$ around the desired reaction temperature $T_{\text{set}} = 90^\circ\text{C}$ in order to ensure that the produced polymer has the required properties.

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

State	Init. cond.	Min.	Max.	Unit
m_W	10,000	0	inf.	kg
m_A	853	0	inf.	kg
m_P	26.5	0	inf.	kg
T_R	90.0	$T_{\text{set}} - 2.0$	$T_{\text{set}} + 2.0$	$^\circ\text{C}$
T_S	90.0	0	100	$^\circ\text{C}$
T_M	90.0	0	100	$^\circ\text{C}$
T_{EK}	35.0	0	100	$^\circ\text{C}$
T_{AWT}	35.0	0	100	$^\circ\text{C}$
T_{adiab}	104.897	0	109	$^\circ\text{C}$
m_F^{acc}	0	0	30,000	kg

and set via:

```
[13]: # auxiliary term
temp_range = 2.0

# lower bound states
mpc.bounds['lower', '_x', 'm_W'] = 0.0
mpc.bounds['lower', '_x', 'm_A'] = 0.0
mpc.bounds['lower', '_x', 'm_P'] = 26.0

mpc.bounds['lower', '_x', 'T_R'] = 363.15 - temp_range
mpc.bounds['lower', '_x', 'T_S'] = 298.0
mpc.bounds['lower', '_x', 'Tout_M'] = 298.0
mpc.bounds['lower', '_x', 'T_EK'] = 288.0
mpc.bounds['lower', '_x', 'Tout_AWT'] = 288.0
mpc.bounds['lower', '_x', 'accum_monom'] = 0.0

# upper bound states
mpc.bounds['upper', '_x', 'T_S'] = 400.0
mpc.bounds['upper', '_x', 'Tout_M'] = 400.0
mpc.bounds['upper', '_x', 'T_EK'] = 400.0
mpc.bounds['upper', '_x', 'Tout_AWT'] = 400.0
mpc.bounds['upper', '_x', 'accum_monom'] = 30000.0
mpc.bounds['upper', '_x', 'T_adiab'] = 382.15
```

The upper bound of the reactor temperature is set via a soft-constraint:

```
[ ]: mpc.set_nl_cons('T_R_UB', _x['T_R'], ub=363.15+temp_range, soft_constraint=True, penalty_
    ↳ term_cons=1e4)
```

The bounds of the inputs are summarized below:

Control	Min.	Max.	Unit
\dot{m}_F	0	30,000	kg h^{-1}
T_M^{IN}	60	100	$^{\circ}\text{C}$
$T_{\text{AWT}}^{\text{IN}}$	60	100	$^{\circ}\text{C}$

and set via:

```
[14]: # lower bound inputs
mpc.bounds['lower', '_u', 'm_dot_f'] = 0.0
mpc.bounds['lower', '_u', 'T_in_M'] = 333.15
mpc.bounds['lower', '_u', 'T_in_EK'] = 333.15

# upper bound inputs
mpc.bounds['upper', '_u', 'm_dot_f'] = 3.0e4
mpc.bounds['upper', '_u', 'T_in_M'] = 373.15
mpc.bounds['upper', '_u', 'T_in_EK'] = 373.15
```

4.14.2.3 Scaling

Because the magnitudes of the states and inputs are very different, the performance of the optimizer can be enhanced by properly scaling the states and inputs:

```
[15]: # states
mpc.scaling['_x', 'm_W'] = 10
mpc.scaling['_x', 'm_A'] = 10
mpc.scaling['_x', 'm_P'] = 10
mpc.scaling['_x', 'accum_monom'] = 10

# control inputs
mpc.scaling['_u', 'm_dot_f'] = 100
```

4.14.2.4 Uncertain values

In a real system, usually the model parameters cannot be determined exactly, what represents an important source of uncertainty. In this work, we consider that two of the most critical parameters of the model are not precisely known and vary with respect to their nominal value. In particular, we assume that the specific reaction enthalpy ΔH_R and the specific reaction rate k_0 are constant but uncertain, having values that can vary $\pm 30\%$ with respect to their nominal values

```
[16]: delH_R_var = np.array([950.0, 950.0 * 1.30, 950.0 * 0.70])
k_0_var = np.array([7.0 * 1.00, 7.0 * 1.30, 7.0 * 0.70])

mpc.set_uncertainty_values(delH_R = delH_R_var, k_0 = k_0_var)
```

This means with $n_{\text{robust}}=1$, that 9 different scenarios are considered. The setup of the MPC controller is concluded by:

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

4.14.3 Estimator

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

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

4.14.4 Simulator

To create a simulator in order to run the MPC in a closed-loop, we create an instance of the **do-mpc** simulator which is based on the same model:

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

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

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

simulator.set_param(**params_simulator)
```

4.14.4.1 Realizations of uncertain parameters

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

```
[21]: p_num = simulator.get_p_template()
      tvp_num = simulator.get_tvp_template()
```

We define a function which is called in each simulation step, which returns the current realizations of the parameters with respect to defined inputs (in this case `t_now`):

```
[22]: # uncertain parameters
      p_num['delH_R'] = 950 * np.random.uniform(0.75, 1.25)
      p_num['k_0'] = 7 * np.random.uniform(0.75*1.25)
      def p_fun(t_now):
          return p_num
      simulator.set_p_fun(p_fun)
```

By defining `p_fun` as above, the function will return a constant value for both uncertain parameters within a range of $\pm 25\%$ of the nominal value. To finish the configuration of the simulator, call:

```
[23]: simulator.setup()
```

4.14.5 Closed-loop simulation

For the simulation of the MPC configured for the CSTR, we inspect the file **main.py**. We define the initial state of the system and set it for all parts of the closed-loop configuration:

```
[24]: # Set the initial state of the controller and simulator:
# assume nominal values of uncertain parameters as initial guess
delH_R_real = 950.0
c_pR = 5.0

# x0 is a property of the simulator - we obtain it and set values.
x0 = simulator.x0

x0['m_W'] = 10000.0
x0['m_A'] = 853.0
x0['m_P'] = 26.5

x0['T_R'] = 90.0 + 273.15
x0['T_S'] = 90.0 + 273.15
x0['Tout_M'] = 90.0 + 273.15
x0['T_EK'] = 35.0 + 273.15
x0['Tout_AWT'] = 35.0 + 273.15
x0['accum_monom'] = 300.0
x0['T_adiab'] = x0['m_A']*delH_R_real/((x0['m_W'] + x0['m_A'] + x0['m_P']) * c_pR) + x0[
↪ 'T_R']

mpc.x0 = x0
simulator.x0 = x0
estimator.x0 = x0

mpc.set_initial_guess()
```

Now, we simulate the closed-loop for 100 steps (and suppress the output of the cell with the magic command %%capture):

```
[25]: %%capture
for k in range(100):
    u0 = mpc.make_step(x0)
    y_next = simulator.make_step(u0)
    x0 = estimator.make_step(y_next)
```

4.14.6 Animating the results

To animate the results, we first configure the **do-mpc** graphics object, which is initiated with the respective data object:

```
[48]: mpc_graphics = do_mpc.graphics.Graphics(mpc.data)
```

We quickly configure Matplotlib.

```
[49]: from matplotlib import rcParams
rcParams['axes.grid'] = True
rcParams['font.size'] = 18
```


We then create a figure, configure which lines to plot on which axis and add labels.

```
[50]: %%capture
fig, ax = plt.subplots(5, sharex=True, figsize=(16,12))
plt.ion()
# Configure plot:
mpc_graphics.add_line(var_type='_x', var_name='T_R', axis=ax[0])
mpc_graphics.add_line(var_type='_x', var_name='accum_monom', axis=ax[1])
mpc_graphics.add_line(var_type='_u', var_name='m_dot_f', axis=ax[2])
mpc_graphics.add_line(var_type='_u', var_name='T_in_M', axis=ax[3])
mpc_graphics.add_line(var_type='_u', var_name='T_in_EK', axis=ax[4])

ax[0].set_ylabel('T_R [K]')
ax[1].set_ylabel('acc. monom')
ax[2].set_ylabel('m_dot_f')
ax[3].set_ylabel('T_in_M [K]')
ax[4].set_ylabel('T_in_EK [K]')
ax[4].set_xlabel('time')

fig.align_ylabels()
```

After importing the necessary package:

```
[43]: from matplotlib.animation import FuncAnimation, ImageMagickWriter
```

We obtain the animation with:

```
[51]: def update(t_ind):
    print('Writing frame: {}'.format(t_ind), end='\r')
    mpc_graphics.plot_results(t_ind=t_ind)
    mpc_graphics.plot_predictions(t_ind=t_ind)
    mpc_graphics.reset_axes()
    lines = mpc_graphics.result_lines.full
    return lines

n_steps = mpc.data['_time'].shape[0]

anim = FuncAnimation(fig, update, frames=n_steps, blit=True)

gif_writer = ImageMagickWriter(fps=5)
anim.save('anim_poly_batch.gif', writer=gif_writer)

Writing frame: 99.
```

We are displaying recorded values as solid lines and predicted trajectories as dashed lines. Multiple dashed lines exist for different realizations of the uncertain scenarios.

The most interesting behavior here can be seen in the state T_R , which has the upper bound:

```
[38]: mpc.bounds['upper', '_x', 'T_R']
```

```
[38]: DM(375.15)
```

Due to robust control, we are approaching this value but hold a certain distance as some possible trajectories predict a temperature increase. As the reaction finishes we can safely increase the temperature because a rapid temperature change due to uncertainty is impossible.

4.15 Oscillating masses

In this Jupyter Notebook we illustrate the example `oscillating_masses_discrete`.

Open an interactive online Jupyter Notebook with this content on Binder:

The example consists of the three modules `template_model.py`, which describes the system model, `template_mpc.py`, which defines the settings for the control and `template_simulator.py`, which sets the parameters for the simulator. The modules are used in `main.py` for the closed-loop execution of the controller. One exemplary result will be presented at the end of this tutorial as a gif.

In the following the different parts are presented. But first, we start by importing basic modules and `do-mpc`.

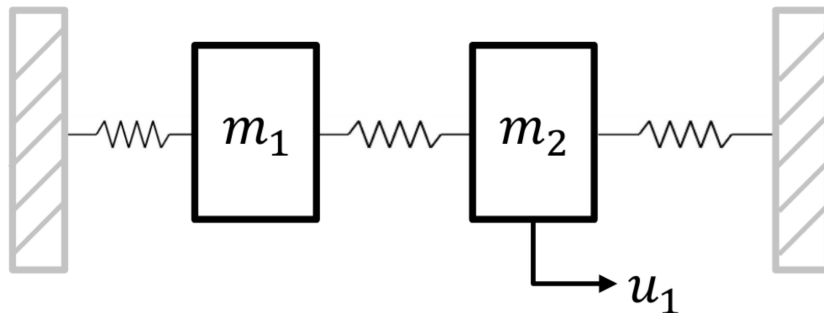
```
[1]: import numpy as np
import sys
from casadi import *

# Add do_mpc to path. This is not necessary if it was installed via pip
import os
rel_do_mpc_path = os.path.join('.', '..', '..')
sys.path.append(rel_do_mpc_path)

# Import do_mpc package:
import do_mpc
```

4.15.1 Model

In the following we will present the configuration, setup and connection between these blocks, starting with the `model`. The considered model are two horizontally oscillating masses interconnected via a spring where each one is connected via a spring to a wall, as shown below:



The states of each mass are its position s_i and velocity v_i , $i = 1, 2$. A force u_1 can be applied to the right mass. The

via first-order hold and a sampling time of 0.5 seconds discretized model $x_{k+1} = Ax_k + Bu_k$ is given by:

$$A = \begin{bmatrix} 0.763 & 0.460 & 0.115 & 0.020 \\ 0.899 & 0.763 & 0.420 & 0.115 \\ 0.115 & 0.020 & 0.763 & 0.460 \\ 0.420 & 0.115 & 0.899 & 0.763 \end{bmatrix}, \quad B = \begin{bmatrix} 0.014 \\ 0.063 \\ 0.221 \\ 0.367 \end{bmatrix},$$

where $x = [s_1, v_1, s_2, v_2]^T$ and $u = [u_1]$.

The discrete model is initiated by:

```
[2]: model_type = 'discrete' # either 'discrete' or 'continuous'
model = do_mpc.model.Model(model_type)
```

4.15.1.1 States and control inputs

The states and the inputs are directly created as vectors:

```
[3]: _x = model.set_variable(var_type='_x', var_name='x', shape=(4,1))
_u = model.set_variable(var_type='_u', var_name='u', shape=(1,1))
```

Afterwards the discrete-time LTI model is added:

```
[4]: A = np.array([[ 0.763,  0.460,  0.115,  0.020],
                  [-0.899,  0.763,  0.420,  0.115],
                  [ 0.115,  0.020,  0.763,  0.460],
                  [ 0.420,  0.115, -0.899,  0.763]])

B = np.array([[0.014],
              [0.063],
              [0.221],
              [0.367]])

x_next = A@_x + B@_u

model.set_rhs('x', x_next)
```

Additionally, we will define an expression, which represents the stage and terminal cost of our control problem. This term will be later used as the cost in the MPC formulation and can be used to directly plot the trajectory of the cost of each state.

```
[5]: model.set_expression(expr_name='cost', expr=sum1(_x**2))
[5]: SX(((sq(x_0)+sq(x_1))+sq(x_2))+sq(x_3)))
```

The model setup is completed via:

```
[6]: # Build the model
model.setup()
```

4.15.2 Controller

Next, the model predictive controller is configured. First, one member of the mpc class is generated with the prediction model defined above:

```
[7]: mpc = do_mpc.controller.MPC(model)
```

We choose the prediction horizon `n_horizon` to 7 and set the robust horizon `n_robust` to zero, because no uncertainties are present. The time step `t_step` is set to 0.5 seconds (like the discretization time step)). There is no need to apply a discretization scheme, because the system is discrete:

```
[8]: setup_mpc = {
    'n_robust': 0,
    'n_horizon': 7,
    't_step': 0.5,
    'state_discretization': 'discrete',
    'store_full_solution': True,
    # Use MA27 linear solver in ipopt for faster calculations:
    '#nlpsol_opts': {'ipopt.linear_solver': 'MA27'}
}

mpc.set_param(**setup_mpc)
```

4.15.2.1 Objective

The goal of the controller is to bring the system to the origin, hence we apply a quadratic cost with weight one to every state and penalty on input changes for a smooth operation. This is here done by using the the cost expression defined in the model:

```
[9]: mterm = model.aux['cost'] # terminal cost
lterm = model.aux['cost'] # terminal cost
      # stage cost

mpc.set_objective(mterm=mterm, lterm=lterm)

mpc.set_rterm(u=1e-4) # input penalty
```

4.15.2.2 Constraints

In the next step, the constraints of the control problem are set. In this case, there are only upper and lower bounds for each state and the input. The displacement has to fulfill $-4\text{m} \leq s_i \leq 4\text{m}$, the velocity $-10\text{ms}^{-1} \leq v_i \leq 10\text{ms}^{-1}$ and the force cannot exceed $-0.5\text{N} \leq u_1 \leq 0.5\text{N}$:

```
[10]: max_x = np.array([[4.0], [10.0], [4.0], [10.0]])

# lower bounds of the states
mpc.bounds['lower', '_x', 'x'] = -max_x

# upper bounds of the states
mpc.bounds['upper', '_x', 'x'] = max_x

# lower bounds of the input
```

(continues on next page)

(continued from previous page)

```
mpc.bounds['lower','_u','u'] = -0.5

# upper bounds of the input
mpc.bounds['upper','_u','u'] = 0.5
```

The setup of the MPC controller is concluded by:

```
[11]: mpc.setup()
```

4.15.3 Estimator

We assume, that all states can be directly measured (state-feedback):

```
[12]: estimator = do_mpc.estimator.StateFeedback(model)
```

4.15.4 Simulator

To create a simulator in order to run the MPC in a closed-loop, we create an instance of the **do-mpc** simulator which is based on the same model:

```
[13]: simulator = do_mpc.simulator.Simulator(model)
```

Because the model is discrete, we do not need to specify options for the integration necessary for simulating the system. We only set the time step `t_step` which is identical to the one used for the optimization and finish the setup of the simulator:

```
[14]: simulator.set_param(t_step = 0.1)
simulator.setup()
```

4.15.5 Closed-loop simulation

For the simulation of the MPC configured for the oscillating masses, we inspect the file **main.py**. We set the initial state of the system (randomly seeded) and set it for all parts of the closed-loop configuration:

```
[15]: # Seed
np.random.seed(99)

# Initial state
e = np.ones([model.n_x,1])
x0 = np.random.uniform(-3*e,3*e) # Values between -3 and +3 for all states
mpc.x0 = x0
simulator.x0 = x0
estimator.x0 = x0

# Use initial state to set the initial guess.
mpc.set_initial_guess()
```

Now, we simulate the closed-loop for 50 steps (and suppress the output of the cell with the magic command `%%capture`):

```
[16]: %%capture
for k in range(50):
    u0 = mpc.make_step(x0)
    y_next = simulator.make_step(u0)
    x0 = estimator.make_step(y_next)
```

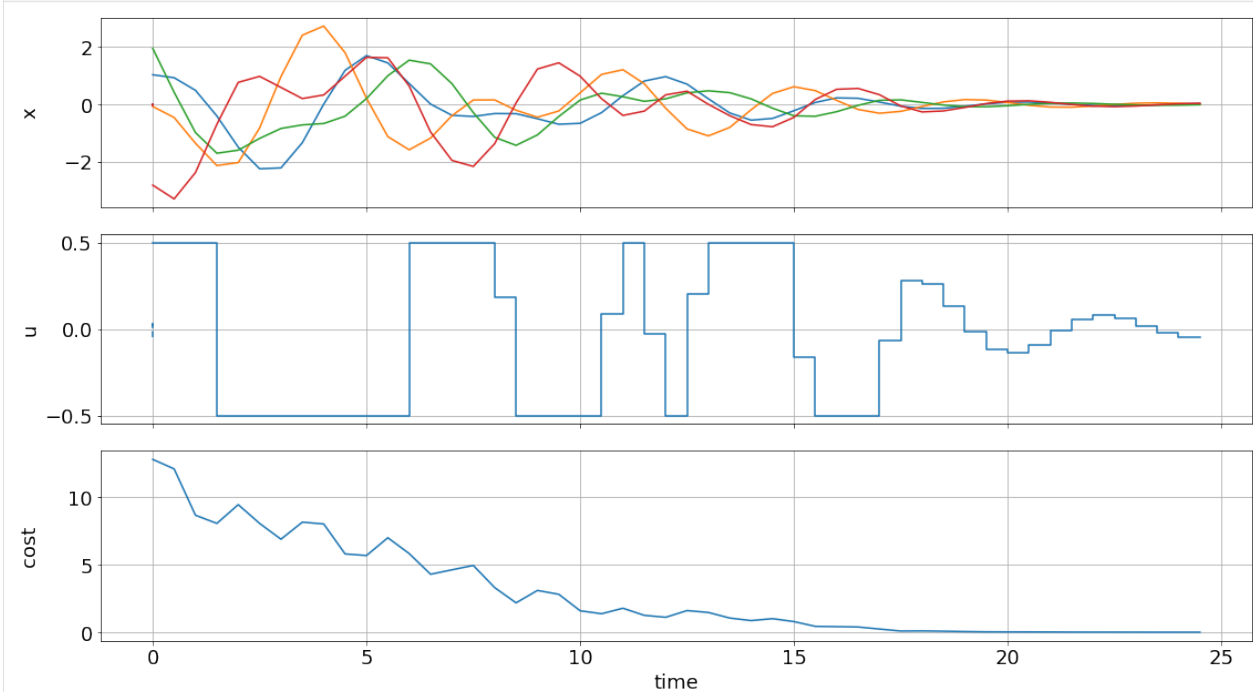
4.15.6 Displaying the results

After some slight configuration of matplotlib:

```
[17]: from matplotlib import rcParams
rcParams['axes.grid'] = True
rcParams['font.size'] = 18
```

We use the convenient `default_plot` function of the `graphics` module, to obtain the graphic below.

```
[18]: import matplotlib.pyplot as plt
fig, ax, graphics = do_mpc.graphics.default_plot(mpc.data, figsize=(16,9))
graphics.plot_results()
graphics.reset_axes()
plt.show()
```



We can see that the control objective was successfully fulfilled and that bounds, e.g. for the control inputs are satisfied.

4.16 Double inverted pendulum

In this Jupyter Notebook we illustrate the example **DIP**. This example illustrates how to use **DAE** models in **do-mpc**.

Open an interactive online Jupyter Notebook with this content on Binder:

The example consists of the three modules **template_model.py**, which describes the system model, **template_mpc.py**, which defines the settings for the control and **template_simulator.py**, which sets the parameters for the simulator. The modules are used in **main.py** for the closed-loop execution of the controller.

We start by importing basic modules and **do-mpc**.

```
[1]: import numpy as np
import sys
from casadi import *

# Add do_mpc to path. This is not necessary if it was installed via pip
import os
rel_do_mpc_path = os.path.join('.', '..', '..')
sys.path.append(rel_do_mpc_path)

# Import do_mpc package:
import do_mpc
```

4.16.1 Model

In the following we will present the configuration, setup and connection between these blocks, starting with the model.

In this example we consider the double pendulum on a cart as depicted below:

The system is described in terms of its horizontal position x and the two angles θ , where $\theta_1 = \theta_2 = 0$ denotes the upright position.

We will formulate a continuous dynamic model for this system and start by initiating a **do-mpc** Model instance:

```
[2]: model_type = 'continuous' # either 'discrete' or 'continuous'
model = do_mpc.model.Model(model_type)
```

4.16.1.1 Parameters

The model is configured with the following (certain) parameters:

```
[3]: m0 = 0.6 # kg, mass of the cart
m1 = 0.2 # kg, mass of the first rod
m2 = 0.2 # kg, mass of the second rod
L1 = 0.5 # m, length of the first rod
L2 = 0.5 # m, length of the second rod

g = 9.80665 # m/s^2, Gravity
```

We furthermore introduce the following derived parameters to conveniently formulate the model equations below.

```
[4]: l1 = L1/2 # m,
      l2 = L2/2 # m,
      J1 = (m1 * l1**2) / 3 # Inertia
      J2 = (m2 * l2**2) / 3 # Inertia

      h1 = m0 + m1 + m2
      h2 = m1*l1 + m2*L1
      h3 = m2*l2
      h4 = m1*l1**2 + m2*L1**2 + J1
      h5 = m2*l2*L1
      h6 = m2*l2**2 + J2
      h7 = (m1*l1 + m2*L1) * g
      h8 = m2*l2*g
```

4.16.1.2 Euler-Lagrangian equations

The dynamics of the double pendulum can be derived from the Euler-Lagrangian equations, which yield:

$$h_1 \ddot{x} + h_2 \ddot{\theta}_1 \cos(\theta_1) + h_3 \ddot{\theta}_2 \cos(\theta_2) = (h_2 \dot{\theta}_1^2 \sin(\theta_1) + h_3 \dot{\theta}_2^2 \sin(\theta_2) + u) \quad (4.51)$$

$$h_2 \cos(\theta_1) \ddot{x} + h_4 \ddot{\theta}_1 + h_5 \cos(\theta_1 - \theta_2) \ddot{\theta}_2 = (h_7 \sin(\theta_1) - h_5 \dot{\theta}_2^2 \sin(\theta_1 - \theta_2)) \quad (4.52)$$

$$h_3 \cos(\theta_2) \ddot{x} + h_5 \cos(\theta_1 - \theta_2) \ddot{\theta}_1 + h_6 \ddot{\theta}_2 = (h_5 \dot{\theta}_1^2 \sin(\theta_1 - \theta_2) + h_8 \sin(\theta_2)) \quad (4.53)$$

we introduce the states

$$x = [x, \theta_1, \theta_2, \dot{x}, \dot{\theta}_1, \dot{\theta}_2]^T$$

and input:

$$u = f$$

which is the horizontal force applied to the cart.

```
[5]: pos = model.set_variable('_x', 'pos')
      theta = model.set_variable('_x', 'theta', (2,1))
      dpos = model.set_variable('_x', 'dpos')
      dtheta = model.set_variable('_x', 'dtheta', (2,1))

      u = model.set_variable('_u', 'force')
```

At this point we have two options. The typical approach would be to rewrite the system as:

$$M(x)\dot{x} = A(x)x + Bu$$

where it can be shown that

$$\det(M) > 0, \forall x$$

such that we can obtain the ODE:

$$\dot{x} = M(x)^{-1}(A(x)x + Bu)$$

do-mpc fully supports this option but it requires some nasty reformulations of the above equations and yields a very complex expression for the ODE.

Instead we suggest ...

4.16.1.3 Differential algebraic equation (DAE)

We introduce new algebraic states

$$z = [\ddot{x}, \ddot{\theta}_1, \ddot{\theta}_2]^T$$

```
[6]: ddpos = model.set_variable('_z', 'ddpos')
      ddtheta = model.set_variable('_z', 'ddtheta', (2,1))
```

which makes it very convenient to formulate the ODE in terms of x, u, z :

$$\dot{x} = [\dot{x}, \dot{\theta}_1, \dot{\theta}_2, \ddot{x}, \ddot{\theta}_1, \ddot{\theta}_2]^T$$

```
[7]: model.set_rhs('pos', dpos)
      model.set_rhs('theta', dtheta)
      model.set_rhs('dpos', ddpos)
      model.set_rhs('dtheta', ddtheta)
```

The only remaining challenge is to implement the relationship between x, u and z , in the form of:

$$g(x, u, z) = 0$$

with **do-mpc** this is easily achieved:

```
[8]: euler_lagrange = vertcat(
    # 1
    h1*ddpos+h2*ddtheta[0]*cos(theta[0])+h3*ddtheta[1]*cos(theta[1])
    - (h2*dtheta[0]**2*sin(theta[0]) + h3*dtheta[1]**2*sin(theta[1]) + u),
    # 2
    h2*cos(theta[0])*ddpos + h4*ddtheta[0] + h5*cos(theta[0]-theta[1])*ddtheta[1]
    - (h7*sin(theta[0]) - h5*dtheta[1]**2*sin(theta[0]-theta[1])),
    # 3
    h3*cos(theta[1])*ddpos + h5*cos(theta[0]-theta[1])*ddtheta[0] + h6*ddtheta[1]
    - (h5*dtheta[0]**2*sin(theta[0]-theta[1]) + h8*sin(theta[1]))
)

model.set_alg('euler_lagrange', euler_lagrange)
```

with just a few lines of code we have defined the dynamics of the double pendulum!

4.16.1.4 Energy equations

The next step is to introduce new “auxiliary” expressions to **do-mpc** for the kinetic and potential energy of the system. This is required in this example, because we will need these expressions for the formulation of the MPC controller.

Introducing these expressions has the additional advantage that **do-mpc** saves and exports the calculated values, which is great for monitoring and debugging.

For the kinetic energy, we have:

$$\begin{aligned}
 E_{\text{kin, cart}} &= \frac{1}{2} m \dot{x}^2 \\
 E_{\text{kin, p1}} &= \frac{1}{2} m_1 ((\dot{x} + l_1 \dot{\theta}_1 \cos(\theta_1))^2 + (l_1 \dot{\theta}_1 \sin(\theta_1))^2) + \frac{1}{2} J_1 \dot{\theta}_1^2 \\
 E_{\text{kin, p2}} &= \frac{1}{2} m_2 ((\dot{x} + L_1 \dot{\theta}_1 \cos(\theta_1) + l_2 \dot{\theta}_2 \cos(\theta_2))^2 \\
 &\quad + (L_1 \dot{\theta}_1 \sin(\theta_1) + l_2 \dot{\theta}_2 \sin(\theta_2))^2) + \frac{1}{2} J_2 \dot{\theta}_2^2
 \end{aligned}$$

and for the potential energy:

$$E_{\text{pot}} = m_1 g l_1 \cos(\theta_1) + m_2 g (L_1 \cos(\theta_1) + l_2 \cos(\theta_2))$$

It only remains to formulate the expressions and set them to the model:

```
[9]: E_kin_cart = 1 / 2 * m0 * dpos**2
E_kin_p1 = 1 / 2 * m1 * (
    (dpos + l1 * dtheta[0] * cos(theta[0]))**2 +
    (l1 * dtheta[0] * sin(theta[0]))**2) + 1 / 2 * J1 * dtheta[0]**2
E_kin_p2 = 1 / 2 * m2 * (
    (dpos + L1 * dtheta[0] * cos(theta[0]) + l2 * dtheta[1] * cos(theta[1]))**2 +
    (L1 * dtheta[0] * sin(theta[0]) + l2 * dtheta[1] * sin(theta[1]))**
    2) + 1 / 2 * J2 * dtheta[1]**2

E_kin = E_kin_cart + E_kin_p1 + E_kin_p2

E_pot = m1 * g * l1 * cos(
    theta[0]) + m2 * g * (L1 * cos(theta[0]) +
    l2 * cos(theta[1]))

model.set_expression('E_kin', E_kin)
model.set_expression('E_pot', E_pot)

[9]: SX(((0.490333*cos(theta_0))+(1.96133*((0.5*cos(theta_0))+(0.25*cos(theta_1))))))
```

Finally, the model setup is completed:

```
[10]: # Build the model
model.setup()
```

4.16.2 Controller

Next, the controller is configured. First, an instance of the MPC class is generated with the prediction model defined above:

```
[11]: mpc = do_mpc.controller.MPC(model)
```

We choose the prediction horizon `n_horizon=100`, set the robust horizon `n_robust = 0`. The time step `t_step` is set to 0.04s and parameters of the applied discretization scheme orthogonal collocation are as seen below:

```
[12]: setup_mpc = {
    'n_horizon': 100,
```

(continues on next page)

(continued from previous page)

```

'n_robust': 0,
'open_loop': 0,
't_step': 0.04,
'state_discretization': 'collocation',
'collocation_type': 'radau',
'collocation_deg': 3,
'collocation_ni': 1,
'store_full_solution': True,
# Use MA27 linear solver in ipopt for faster calculations:
'nlpsol_opts': {'ipopt.linear_solver': 'mumps'}
}
mpc.set_param(**setup_mpc)

```

4.16.2.1 Objective

The control objective is to erect the double pendulum and to stabilize it in the up-up position. It is not straight-forward to formulate an objective which yields this result. Classical set-point tracking, e.g. with the set-point:

$$\theta_s = [0, 0, 0]$$

and a quadratic cost:

$$J = \sum_{k=0}^N (\theta - \theta_s)^2$$

is **known to work very poorly**. Clearly, the problem results from the fact that $\theta_s = 2\pi n$, $n \in \mathbb{Z}$ is also a valid solution.

Instead we will use an energy-based formulation for the objective. If we think of energy in terms of potential and kinetic energy it is clear that we want to maximize the potential energy (up-up position) and minimize the kinetic energy (stabilization).

Since we have already introduced the expressions for the potential and kinetic energy in the model, we can now simply reuse these expressions for the fomulation of the objective function, as shown below:

```

[13]: mterm = model.aux['E_kin'] - model.aux['E_pot'] # terminal cost
      lterm = model.aux['E_kin'] - model.aux['E_pot'] # stage cost

mpc.set_objective(mterm=mterm, lterm=lterm)
# Input force is implicitly restricted through the objective.
mpc.set_rterm(force=0.1)

```

4.16.2.2 Constraints

In the next step, the constraints of the control problem are set. In this case, there is only an upper and lower bounds for the input.

```

[14]: mpc.bounds['lower', '_u', 'force'] = -4
      mpc.bounds['upper', '_u', 'force'] = 4

```

We can now setup the controller.

```
[15]: mpc.setup()
```

4.16.3 Estimator

We assume, that all states can be directly measured (state-feedback):

```
[16]: estimator = do_mpc.estimator.StateFeedback(model)
```

4.16.4 Simulator

To create a simulator in order to run the MPC in a closed-loop, we create an instance of the **do-mpc** simulator which is based on the same model:

```
[29]: simulator = do_mpc.simulator.Simulator(model)
```

For the simulation, we use the time step `t_step` as for the optimizer:

```
[30]: params_simulator = {  
    # Note: ccode doesn't support DAE systems.  
    'integration_tool': 'idas',  
    'abstol': 1e-8,  
    'reltol': 1e-8,  
    't_step': 0.04  
}  
  
simulator.set_param(**params_simulator)
```

```
[31]: simulator.setup()
```

4.16.5 Closed-loop simulation

For the simulation of the MPC configured for the DIP, we inspect the file **main.py**. We define the initial state of the system and set for all parts of the closed-loop configuration:

```
[32]: simulator.x0['theta'] = 0.99*np.pi  
  
x0 = simulator.x0.cat.full()  
  
mpc.x0 = x0  
estimator.x0 = x0  
  
mpc.set_initial_guess()
```

Note that `mpc.set_initial_guess()` is very important in this example. Also note that we didn't set the initial state at exactly π which results in unfavorable numerical issues (it will work however).

4.16.5.1 Prepare visualization

For the visualization of the control performance, we first import matplotlib and change some basic settings:

```
[33]: import matplotlib.pyplot as plt
plt.ion()
from matplotlib import rcParams
rcParams['text.usetex'] = False
rcParams['axes.grid'] = True
rcParams['lines.linewidth'] = 2.0
rcParams['axes.labelsize'] = 'xx-large'
rcParams['xtick.labelsize'] = 'xx-large'
rcParams['ytick.labelsize'] = 'xx-large'
```

We use the plotting capabilities, which are included in **do-mpc**. The `mpc_graphics` contain information like the current estimated state and the predicted trajectory of the states and inputs based on the solution of the control problem. The `sim_graphics` contain the information about the simulated evaluation of the system.

```
[34]: mpc_graphics = do_mpc.graphics.Graphics(mpc.data)
```

For the example of the DIP we create a new function which takes as input the states (at a given time k) and returns the x and y positions of the two bars (the arms of the pendulum).

```
[35]: def pendulum_bars(x):
    x = x.flatten()
    # Get the x,y coordinates of the two bars for the given state x.
    line_1_x = np.array([
        x[0],
        x[0]+L1*np.sin(x[1])
    ])

    line_1_y = np.array([
        0,
        L1*np.cos(x[1])
    ])

    line_2_x = np.array([
        line_1_x[1],
        line_1_x[1] + L2*np.sin(x[2])
    ])

    line_2_y = np.array([
        line_1_y[1],
        line_1_y[1] + L2*np.cos(x[2])
    ])

    line_1 = np.stack((line_1_x, line_1_y))
    line_2 = np.stack((line_2_x, line_2_y))

    return line_1, line_2
```

We then setup a graphic:

```
[36]: %%capture

fig = plt.figure(figsize=(16,9))

ax1 = plt.subplot2grid((4, 2), (0, 0), rowspan=4)
ax2 = plt.subplot2grid((4, 2), (0, 1))
ax3 = plt.subplot2grid((4, 2), (1, 1))
ax4 = plt.subplot2grid((4, 2), (2, 1))
ax5 = plt.subplot2grid((4, 2), (3, 1))

ax2.set_ylabel('$E_{kin}$ [J]')
ax3.set_ylabel('$E_{pot}$ [J]')
ax4.set_ylabel('Angle [rad]')
ax5.set_ylabel('Input force [N]')

# Axis on the right.
for ax in [ax2, ax3, ax4, ax5]:
    ax.yaxis.set_label_position("right")
    ax.yaxis.tick_right()
    if ax != ax5:
        ax.xaxis.set_ticklabels([])

ax5.set_xlabel('time [s]')

mpc_graphics.add_line(var_type='_aux', var_name='E_kin', axis=ax2)
mpc_graphics.add_line(var_type='_aux', var_name='E_pot', axis=ax3)
mpc_graphics.add_line(var_type='_x', var_name='theta', axis=ax4)
mpc_graphics.add_line(var_type='_u', var_name='force', axis=ax5)

ax1.axhline(0,color='black')

bar1 = ax1.plot([],[], '-o', linewidth=5, markersize=10)
bar2 = ax1.plot([],[], '-o', linewidth=5, markersize=10)

ax1.set_xlim(-1.8,1.8)
ax1.set_ylim(-1.2,1.2)
ax1.set_axis_off()

fig.align_ylabels()
fig.tight_layout()
```

4.16.5.2 Run open-loop

Before we test the closed loop case, lets plot one open-loop prediction to check how the resulting graphic looks like.

```
[25]: 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
```

(continues on next page)

(continued from previous page)

This is Ipopt version 3.12.3, running with linear solver mumps.

NOTE: Other linear solvers might be more efficient (see Ipopt documentation).

Number of nonzeros in equality constraint Jacobian...: 19406
 Number of nonzeros in inequality constraint Jacobian.: 0
 Number of nonzeros in Lagrangian Hessian...: 6814

Total number of variables...: 4330
 variables with only lower bounds: 0
 variables with lower and upper bounds: 100
 variables with only upper bounds: 0

Total number of equality constraints...: 4206

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	1.9799658e+002	4.62e-002	1.00e-003	-1.0	0.00e+000	-	0.00e+000	0.00e+000	0
1	1.9809206e+002	7.33e-005	3.25e-004	-1.0	1.53e+000	-4.0	1.00e+000	1.00e+000h	1
2	1.9808344e+002	8.59e-005	1.22e-004	-2.5	4.56e-001	-3.6	1.00e+000	1.00e+000h	1
3	1.9753270e+002	2.78e+000	5.70e-001	-3.8	2.32e+001	-4.1	6.69e-001	1.00e+000f	1
4	1.9586898e+002	2.82e-001	3.12e-001	-3.8	1.14e+001	-3.6	3.78e-001	1.00e+000h	1
5	1.9529674e+002	1.38e+000	1.17e+000	-3.8	5.85e+001	-4.1	4.40e-001	4.20e-001H	1
6	1.9266815e+002	2.02e-001	3.44e-001	-3.8	9.02e+000	-3.7	1.63e-002	1.00e+000h	1
7	1.9230231e+002	1.47e-001	2.58e-001	-3.8	2.10e+001	-4.2	9.32e-001	2.44e-001h	1
8	1.9168746e+002	3.42e-002	2.53e-002	-3.8	6.96e+000	-3.7	1.00e+000	1.00e+000h	1
9	1.9143721e+002	3.64e-002	1.08e-001	-3.8	2.48e+001	-4.2	1.00e+000	1.82e-001h	1
10	1.9079370e+002	3.46e-001	7.82e-001	-3.8	4.32e+002	-4.7	4.15e-001	6.65e-002f	1
11	1.8908306e+002	4.01e-001	4.09e-001	-3.8	4.52e+001	-4.3	1.00e+000	8.14e-001h	1
12	1.8756861e+002	2.58e-001	1.53e-001	-3.8	1.60e+001	-3.8	9.86e-001	1.00e+000h	1
13	1.8672963e+002	1.14e-001	1.17e-001	-3.8	9.14e+000	-3.4	1.00e+000	7.85e-001h	1
14	1.8599444e+002	2.54e-001	2.29e-001	-3.8	6.29e+001	-3.9	1.00e+000	1.66e-001f	1
15	1.8497680e+002	2.03e-001	2.37e-001	-3.8	1.23e+001	-3.5	4.49e-001	1.00e+000h	1
16	1.8431902e+002	1.32e+000	8.76e-001	-3.8	2.62e+002	-3.9	4.75e-001	1.14e-001f	1
17	1.8419241e+002	1.26e+000	8.40e-001	-3.8	1.56e+002	-4.4	4.12e-001	4.26e-002h	1
18	1.8355642e+002	1.89e-001	3.51e-001	-3.8	1.42e+001	-4.0	4.32e-001	1.00e+000h	1
19	1.8340528e+002	8.42e-002	1.34e-001	-3.8	1.80e+001	-3.6	3.85e-001	6.91e-001h	1
20	1.8278290e+002	1.71e-001	8.75e-002	-3.8	1.52e+001	-4.0	5.64e-001	1.00e+000h	1
21	1.8137877e+002	1.19e+000	1.93e-001	-3.8	4.27e+001	-4.5	7.99e-001	1.00e+000h	1
22	1.8054394e+002	9.66e-001	2.76e-001	-3.8	2.48e+002	-5.0	1.00e+000	1.90e-001h	1
23	1.7935141e+002	6.67e-001	6.46e-001	-3.8	7.50e+001	-4.6	1.00e+000	1.00e+000h	1
24	1.7911409e+002	2.49e-002	6.09e-002	-3.8	2.17e+001	-4.1	8.34e-001	1.00e+000h	1
25	1.7901217e+002	2.49e-002	7.69e-002	-3.8	1.01e+002	-5.1	1.00e+000	7.60e-002h	1
26	1.7841964e+002	3.14e-001	2.48e-001	-3.8	2.56e+002	-5.6	1.00e+000	2.07e-001f	1
27	1.7779311e+002	4.29e-001	2.39e-001	-3.8	2.35e+002	-5.1	6.42e-001	1.87e-001h	1
28	1.7641493e+002	3.35e+000	5.22e+000	-3.8	2.10e+002	-4.7	1.00e+000	8.47e-001h	1
29	1.7645866e+002	2.41e+000	3.83e+000	-3.8	5.03e+001	-4.3	3.56e-003	3.01e-001h	1

(continues on next page)

(continued from previous page)

iter	objective	inf_pr	inf_du	lg(mu)	d	lg(rg)	alpha_du	alpha_pr	ls
30	1.7678347e+002	4.10e-001	2.53e+000	-3.8	5.37e+001	-3.9	6.38e-001	1.00e+000h	1
31	1.7626872e+002	1.41e-001	1.57e-001	-3.8	3.71e+001	-4.3	1.00e+000	1.00e+000h	1
32	1.7595595e+002	1.68e-001	1.65e-001	-3.8	1.13e+002	-4.8	4.39e-001	2.67e-001h	1
33	1.7542093e+002	1.98e-001	9.70e-002	-3.8	5.13e+001	-4.4	1.00e+000	1.00e+000h	1
34	1.7511185e+002	6.43e-002	1.12e-001	-3.8	3.50e+001	-4.0	1.00e+000	1.00e+000h	1
35	1.7415559e+002	6.50e-001	1.10e+000	-3.8	1.31e+002	-4.4	8.92e-001	7.73e-001f	1
36	1.7380288e+002	3.13e-002	1.50e-001	-3.8	3.09e+001	-4.0	7.01e-001	1.00e+000h	1
37	1.7304848e+002	9.75e-002	3.54e-001	-3.8	5.16e+001	-4.5	8.25e-001	1.00e+000h	1
38	1.7285605e+002	9.13e-002	3.20e-001	-3.8	1.54e+002	-5.0	2.54e-001	8.40e-002h	1
39	1.7252950e+002	9.73e-002	3.00e-001	-3.8	3.42e+002	-5.4	2.53e-001	7.95e-002h	1
iter	objective	inf_pr	inf_du	lg(mu)	d	lg(rg)	alpha_du	alpha_pr	ls
40	1.7200551e+002	1.49e-001	2.55e-001	-3.8	1.47e+002	-5.0	3.78e-001	3.66e-001h	1
41	1.7157999e+002	1.24e-001	3.12e-001	-3.8	5.67e+001	-4.6	1.00e+000	1.00e+000h	1
42	1.7119524e+002	3.75e-001	1.99e-001	-3.8	2.21e+002	-5.1	5.87e-001	3.09e-001h	1
43	1.7097272e+002	3.44e-001	1.89e-001	-3.8	1.65e+002	-4.6	1.00e+000	3.35e-001h	1
44	1.7078865e+002	8.53e-002	2.85e-001	-3.8	5.44e+001	-4.2	1.00e+000	1.00e+000h	1
45	1.7024868e+002	4.88e-001	1.27e+000	-3.8	2.24e+002	-4.7	3.93e-001	4.54e-001h	1
46	1.7001929e+002	1.26e-001	1.96e-001	-3.8	6.59e+001	-4.3	1.00e+000	7.89e-001h	1
47	1.6968674e+002	1.12e-001	4.08e-001	-3.8	6.88e+001	-4.8	1.00e+000	1.00e+000h	1
48	1.6906303e+002	4.92e-001	1.04e+000	-3.8	1.39e+002	-5.2	1.00e+000	1.00e+000h	1
49	1.6886981e+002	1.19e-001	1.54e-001	-3.8	7.22e+001	-4.8	1.00e+000	1.00e+000h	1
iter	objective	inf_pr	inf_du	lg(mu)	d	lg(rg)	alpha_du	alpha_pr	ls
50	1.6869616e+002	1.07e+000	8.69e-001	-3.8	4.15e+002	-5.3	1.69e-001	1.82e-001h	1
51	1.6829813e+002	6.56e-001	5.55e-001	-3.8	1.59e+002	-4.9	1.56e-002	3.75e-001h	1
52	1.6813212e+002	1.78e-001	4.90e-001	-3.8	6.84e+001	-4.4	1.00e+000	1.00e+000h	1
53	1.6790310e+002	2.83e-001	8.01e-001	-3.8	1.70e+003	-4.9	1.88e-002	2.72e-002h	1
54	1.6725101e+002	9.59e-001	8.27e-001	-3.8	9.57e+001	-4.5	7.48e-001	1.00e+000h	1
55	1.6704769e+002	1.91e-001	1.78e-001	-3.8	7.40e+001	-5.0	1.00e+000	1.00e+000h	1
56	1.6657356e+002	9.58e-001	5.43e-001	-3.8	1.29e+002	-5.4	1.00e+000	6.39e-001h	1
57	1.6606471e+002	6.17e-001	4.89e-001	-3.8	9.69e+001	-5.0	1.00e+000	1.00e+000h	1
58	1.6464272e+002	5.07e+000	7.50e+000	-3.8	1.62e+003	-5.5	4.56e-002	1.58e-001f	1
59	1.6626159e+002	1.68e+000	2.52e+000	-3.8	9.05e+001	-5.1	1.00e+000	8.45e-001h	1
iter	objective	inf_pr	inf_du	lg(mu)	d	lg(rg)	alpha_du	alpha_pr	ls
60	1.6458662e+002	1.33e+000	3.83e+000	-3.8	2.07e+002	-4.6	5.73e-001	6.34e-001h	1
61	1.6393555e+002	5.06e-001	1.56e+000	-3.8	1.98e+002	-5.1	1.00e+000	1.00e+000h	1
62	1.6335159e+002	1.02e+000	1.14e+000	-3.8	2.38e+002	-5.6	1.00e+000	6.25e-001h	1
63	1.6158405e+002	6.86e+000	8.40e+000	-3.8	2.86e+002	-5.2	1.81e-001	1.00e+000h	1
64	1.6215958e+002	5.45e+000	6.73e+000	-3.8	9.45e+001	-4.7	1.00e+000	2.15e-001h	1
65	1.6273466e+002	3.79e+000	4.83e+000	-3.8	9.35e+001	-4.3	1.00e+000	3.27e-001h	1
66	1.6183355e+002	1.80e+000	2.33e+000	-3.8	1.71e+002	-4.8	7.42e-002	5.37e-001h	1
67	1.6091649e+002	6.21e-001	1.95e+000	-3.8	1.90e+002	-5.3	2.54e-001	1.00e+000h	1
68	1.5984939e+002	8.56e+000	5.36e+000	-3.8	5.89e+002	-5.7	3.10e-001	6.94e-001h	1
69	1.5879235e+002	5.00e+000	2.61e+000	-3.8	3.62e+002	-5.3	3.31e-001	3.85e-001h	1
iter	objective	inf_pr	inf_du	lg(mu)	d	lg(rg)	alpha_du	alpha_pr	ls
70	1.5878528e+002	1.98e+000	9.23e-001	-3.8	1.22e+002	-4.9	4.57e-001	6.28e-001h	1
71	1.5737834e+002	1.80e+000	1.53e+000	-3.8	2.68e+002	-5.4	2.19e-001	6.64e-001h	1
72	1.5708649e+002	1.64e+000	1.44e+000	-3.8	5.33e+002	-4.9	1.50e-002	8.87e-002h	1
73	1.5683289e+002	5.99e-001	1.52e+000	-3.8	1.38e+002	-4.5	2.34e-001	1.00e+000f	1
74	1.5669989e+002	3.72e-001	8.79e-001	-3.8	4.34e+001	-4.1	7.25e-001	3.93e-001h	1
75	1.5604987e+002	5.37e-001	6.23e-001	-3.8	8.31e+001	-4.6	6.88e-001	1.00e+000h	1
76	1.5588161e+002	4.59e-001	5.45e-001	-3.8	1.32e+002	-5.0	2.47e-001	1.82e-001h	1

(continues on next page)

(continued from previous page)

```

77 1.5563047e+002 4.74e-001 5.73e-001 -3.8 4.92e+002 -5.5 1.16e-001 7.24e-002h 1
78 1.5539016e+002 4.58e-001 5.75e-001 -3.8 3.16e+002 -5.1 6.28e-002 1.13e-001h 1
79 1.5521279e+002 3.46e-001 4.57e-001 -3.8 9.52e+001 -4.7 1.00e+000 2.73e-001h 1
iter   objective   inf_pr   inf_du lg(mu)  ||d|| lg(rg) alpha_du alpha_pr ls
80 1.5505242e+002 8.15e-002 1.32e-001 -3.8 3.74e+001 -4.2 1.00e+000 1.00e+000h 1
81 1.5493811e+002 1.01e-001 1.31e-001 -3.8 2.12e+002 -4.7 2.12e-001 7.92e-002h 1
82 1.5465089e+002 1.74e-001 2.32e-001 -3.8 5.54e+001 -4.3 1.00e+000 7.47e-001h 1
83 1.5456570e+002 1.77e-001 2.33e-001 -3.8 2.83e+002 -4.8 1.04e-001 3.27e-002h 1
84 1.5440037e+002 1.44e-001 1.66e-001 -3.8 4.80e+001 -4.3 4.72e-001 3.36e-001h 1
85 1.5407611e+002 2.49e-001 1.65e-001 -3.8 1.44e+002 -4.8 2.12e-001 2.59e-001h 1
86 1.5371105e+002 4.04e-001 3.14e-001 -3.8 3.83e+002 -5.3 1.31e-001 1.26e-001h 1
87 1.5361938e+002 3.69e-001 2.90e-001 -3.8 1.17e+002 -4.9 5.48e-001 1.03e-001h 1
88 1.5316240e+002 5.61e-001 5.09e-001 -3.8 9.94e+002 -5.3 7.88e-002 6.80e-002f 1
89 1.5309233e+002 1.53e-001 1.12e-001 -3.8 2.79e+001 -4.0 7.90e-001 7.26e-001h 1
iter   objective   inf_pr   inf_du lg(mu)  ||d|| lg(rg) alpha_du alpha_pr ls
90 1.5299821e+002 1.21e-001 8.12e-002 -3.8 5.49e+001 -4.5 1.00e+000 2.44e-001h 1
91 1.5272112e+002 2.39e-001 2.69e-001 -3.8 2.84e+002 -5.0 4.16e-001 1.41e-001h 1
92 1.5248948e+002 3.02e-001 3.58e-001 -3.8 1.23e+003 -5.0 1.24e-001 2.23e-002h 1
93 1.5208923e+002 2.92e-001 1.91e-001 -3.8 7.93e+001 -4.6 1.00e+000 5.35e-001h 1
94 1.5165593e+002 4.43e-001 3.89e-001 -3.8 1.99e+002 -5.1 3.68e-001 2.49e-001h 1
95 1.5154571e+002 3.47e-001 3.06e-001 -3.8 5.88e+001 -4.6 1.00e+000 2.29e-001h 1
96 1.5097015e+002 6.25e-001 5.01e-001 -3.8 2.31e+002 -5.1 4.66e-001 3.02e-001h 1
97 1.5076115e+002 5.87e-001 4.83e-001 -3.8 1.89e+002 -4.7 4.59e-001 1.31e-001h 1
98 1.5059995e+002 1.75e-001 2.03e-001 -3.8 3.56e+001 -4.3 1.00e+000 7.24e-001h 1
99 1.5026190e+002 1.89e-001 1.84e-001 -3.8 7.00e+001 -4.8 8.53e-001 4.10e-001h 1
iter   objective   inf_pr   inf_du lg(mu)  ||d|| lg(rg) alpha_du alpha_pr ls
100 1.4889151e+002 4.19e+000 2.64e+000 -3.8 1.02e+003 -5.2 1.64e-001 2.34e-001f 1
101 1.4886649e+002 4.06e+000 2.57e+000 -3.8 1.05e+002 -4.8 1.00e+000 2.96e-002h 1
102 1.4893088e+002 2.49e+000 1.77e+000 -3.8 8.55e+001 -4.4 1.00e+000 4.10e-001h 1
103 1.4854574e+002 1.52e+000 1.09e+000 -3.8 9.81e+001 -4.9 7.78e-001 3.86e-001h 1
104 1.4839437e+002 7.61e-001 5.56e-001 -3.8 4.35e+001 -4.4 1.00e+000 5.00e-001h 1
105 1.4812349e+002 6.49e-001 4.73e-001 -3.8 1.79e+002 -4.9 7.38e-001 1.50e-001h 1
106 1.4779392e+002 5.70e-001 4.12e-001 -3.8 2.16e+002 -5.0 1.63e-001 1.26e-001h 1
107 1.4739548e+002 2.05e-001 2.93e-001 -3.8 5.17e+001 -4.5 1.00e+000 8.06e-001h 1
108 1.4721843e+002 2.01e-001 2.78e-001 -3.8 1.22e+002 -5.0 3.97e-001 1.27e-001h 1
109 1.4708638e+002 2.11e-001 2.89e-001 -3.8 7.01e+002 -5.5 1.16e-001 1.71e-002h 1
iter   objective   inf_pr   inf_du lg(mu)  ||d|| lg(rg) alpha_du alpha_pr ls
110 1.4683471e+002 2.09e-001 3.33e-001 -3.8 1.24e+002 -5.1 1.00e+000 1.73e-001h 1
111 1.4661437e+002 1.32e-001 1.83e-001 -3.8 4.93e+001 -4.6 1.00e+000 4.66e-001h 1
112 1.4625878e+002 3.44e-001 6.18e-001 -3.8 5.97e+002 -5.1 1.10e-001 7.02e-002h 1
113 1.4605316e+002 3.23e-001 5.71e-001 -3.8 1.34e+002 -4.7 5.12e-001 1.25e-001h 1
114 1.4587093e+002 6.48e-002 1.26e-001 -3.8 2.55e+001 -4.3 1.00e+000 8.23e-001h 1
115 1.4568491e+002 6.21e-002 1.21e-001 -3.8 4.04e+001 -4.7 1.00e+000 3.96e-001h 1
116 1.4545966e+002 9.02e-002 1.44e-001 -3.8 1.03e+002 -5.2 3.53e-001 2.06e-001h 1
117 1.4507278e+002 2.02e-001 4.40e-001 -3.8 3.25e+002 -5.7 1.72e-001 1.33e-001h 1
118 1.4481436e+002 2.14e-001 4.96e-001 -3.8 1.66e+002 -5.3 1.87e-001 1.33e-001h 1
119 1.4453267e+002 2.14e-001 4.22e-001 -3.8 8.65e+001 -4.8 1.00e+000 3.70e-001h 1
iter   objective   inf_pr   inf_du lg(mu)  ||d|| lg(rg) alpha_du alpha_pr ls
120 1.4448491e+002 1.79e-001 3.62e-001 -3.8 3.21e+001 -4.4 1.00e+000 1.71e-001h 1
121 1.4413536e+002 3.32e-001 1.27e+000 -3.8 1.55e+002 -4.9 7.11e-001 2.79e-001h 1
122 1.4400210e+002 1.99e-001 7.27e-001 -3.8 3.19e+001 -4.5 1.00e+000 4.03e-001h 1
123 1.4387560e+002 1.80e-001 6.44e-001 -3.8 8.81e+001 -4.9 7.87e-001 1.16e-001h 1

```

(continues on next page)

(continued from previous page)

124	1.4359493e+002	1.76e-001	5.64e-001	-3.8	3.24e+002	-5.4	1.15e-001	9.49e-002h	1
125	1.4282377e+002	1.25e+000	1.47e+000	-3.8	1.20e+002	-5.0	1.00e+000	1.00e+000h	1
126	1.4240341e+002	4.57e-001	6.60e-001	-3.8	7.68e+001	-4.6	5.65e-001	6.41e-001h	1
127	1.4135154e+002	2.36e+000	1.06e+000	-3.8	3.83e+002	-5.0	3.24e-001	4.13e-001h	1
128	1.4123980e+002	1.94e+000	8.81e-001	-3.8	6.70e+001	-4.6	6.77e-001	1.77e-001h	1
129	1.4119646e+002	1.92e+000	8.71e-001	-3.8	2.44e+002	-5.1	3.24e-001	1.17e-002h	1
iter	objective	inf_pr	inf_du	lg(mu)	d	lg(rg)	alpha_du	alpha_pr	ls
130	1.4108190e+002	1.50e+000	6.79e-001	-3.8	7.45e+001	-4.7	1.00e+000	2.18e-001h	1
131	1.4028488e+002	8.63e-001	1.55e+000	-3.8	1.95e+002	-5.1	4.36e-001	5.52e-001h	1
132	1.3989449e+002	2.64e-001	8.00e-001	-3.8	5.86e+001	-4.7	3.48e-001	6.92e-001h	1
133	1.3967474e+002	2.71e-001	8.57e-001	-3.8	2.11e+002	-5.2	3.55e-001	1.08e-001h	1
134	1.3950765e+002	2.77e-001	8.95e-001	-3.8	4.32e+002	-5.2	5.27e-002	3.64e-002h	1
135	1.3937031e+002	2.85e-001	9.27e-001	-3.8	5.39e+003	-5.3	9.99e-003	2.47e-003f	1
136	1.3914186e+002	3.29e-001	1.03e+000	-3.8	7.31e+001	-4.9	1.00e+000	7.18e-001h	1
137	1.3899514e+002	8.55e-002	2.01e-001	-3.8	2.77e+001	-4.4	1.00e+000	1.00e+000h	1
138	1.3894414e+002	8.28e-002	1.98e-001	-3.8	7.17e+001	-4.9	2.73e-001	6.98e-002h	1
139	1.3885984e+002	9.19e-002	3.08e-001	-3.8	3.16e+002	-5.4	2.05e-001	2.81e-002h	1
iter	objective	inf_pr	inf_du	lg(mu)	d	lg(rg)	alpha_du	alpha_pr	ls
140	1.3830738e+002	7.17e-001	1.02e+000	-3.8	1.00e+003	-5.9	1.44e-001	1.05e-001f	1
141	1.3787017e+002	1.21e-001	1.82e-001	-3.8	5.50e+001	-4.5	1.00e+000	1.00e+000h	1
142	1.3770434e+002	1.57e-001	1.51e-001	-3.8	1.26e+002	-5.0	1.43e-001	2.50e-001h	1
143	1.3675757e+002	1.01e+002	2.68e+001	-3.8	1.13e+003	-5.5	1.59e-001	1.00e+000f	1
144	1.3625684e+002	8.57e+001	2.27e+001	-3.8	4.12e+002	-5.1	1.14e-002	1.56e-001h	1
145	1.3622971e+002	8.45e+001	2.24e+001	-3.8	2.91e+002	-5.6	4.90e-001	1.40e-002h	1
146	1.3598151e+002	4.31e+001	1.26e+001	-3.8	3.25e+002	-5.1	2.16e-001	5.22e-001h	1
147	1.3583076e+002	2.97e+001	8.95e+000	-3.8	3.83e+002	-5.6	3.95e-001	3.30e-001h	1
148	1.3193233e+002	3.22e+001	1.41e+001	-3.8	9.31e+002	-6.1	3.09e-003	1.00e+000h	1
149	1.2948992e+002	4.62e+001	9.21e+000	-3.8	1.94e+003	-6.6	5.40e-002	5.00e-001h	2
iter	objective	inf_pr	inf_du	lg(mu)	d	lg(rg)	alpha_du	alpha_pr	ls
150	1.2730554e+002	1.20e+001	1.33e+001	-3.8	1.21e+003	-6.1	1.67e-001	5.00e-001h	2
151	1.2850471e+002	1.51e+000	1.24e+000	-3.8	2.14e+002	-5.7	1.86e-001	1.00e+000h	1
152	1.2686679e+002	1.35e+001	6.38e+000	-3.8	4.49e+002	-5.3	1.06e-001	1.00e+000h	1
153	1.2717669e+002	2.09e+000	2.02e+000	-3.8	1.93e+002	-5.8	1.60e-001	9.23e-001h	1
154	1.2594944e+002	1.84e+000	9.12e-001	-3.8	6.81e+002	-6.2	1.14e-001	3.18e-001h	1
155	1.2272685e+002	9.62e+001	1.44e+001	-3.8	3.13e+003	-6.7	5.25e-002	3.42e-001f	2
156	1.2255668e+002	7.99e+001	1.24e+001	-3.8	3.41e+003	-7.2	6.26e-002	1.38e-001h	1
157	1.2235994e+002	5.58e+001	8.59e+000	-3.8	5.00e+002	-5.0	2.00e-001	3.11e-001h	1
158	1.2241173e+002	4.54e+001	6.92e+000	-3.8	3.95e+002	-4.5	1.19e-001	1.91e-001h	1
159	1.2241971e+002	3.86e+001	5.87e+000	-3.8	4.02e+002	-5.0	2.35e-001	1.53e-001h	1
iter	objective	inf_pr	inf_du	lg(mu)	d	lg(rg)	alpha_du	alpha_pr	ls
160	1.1987220e+002	3.73e+001	1.12e+001	-3.8	1.77e+003	-5.5	7.81e-002	4.16e-001h	1
161	1.2012337e+002	3.11e+001	9.28e+000	-3.8	2.91e+002	-6.0	1.96e-001	1.70e-001h	1
162	1.2012923e+002	2.29e+001	6.95e+000	-3.8	4.44e+002	-6.4	8.16e-002	2.70e-001h	1
163	1.1925205e+002	9.49e+000	4.42e+000	-3.8	5.85e+002	-6.9	2.30e-001	5.12e-001h	1
164	1.1843605e+002	8.11e+000	4.02e+000	-3.8	2.00e+003	-5.6	1.05e-002	9.63e-002h	1
165	1.1828840e+002	7.20e+000	3.59e+000	-3.8	3.13e+002	-5.2	2.41e-001	1.11e-001h	1
166	1.1687591e+002	4.23e+000	2.58e+000	-3.8	8.21e+002	-5.6	6.72e-002	3.66e-001h	1
167	1.1695091e+002	3.65e+000	2.23e+000	-3.8	1.55e+002	-5.2	7.70e-001	1.37e-001h	1
168	1.1403130e+002	1.16e+002	3.35e+001	-3.8	1.19e+003	-5.7	6.03e-002	1.00e+000h	1
169	1.1396433e+002	1.02e+002	2.96e+001	-3.8	5.11e+002	-5.3	3.76e-001	1.21e-001h	1
iter	objective	inf_pr	inf_du	lg(mu)	d	lg(rg)	alpha_du	alpha_pr	ls
170	1.1441372e+002	7.54e+001	2.24e+001	-3.8	4.44e+002	-4.8	2.41e-001	2.68e-001h	1

(continues on next page)

(continued from previous page)

```

171 1.1458239e+002 7.13e+001 2.12e+001 -3.8 3.55e+002 -5.3 1.68e-001 5.43e-002h 1
172 1.1469604e+002 6.88e+001 2.05e+001 -3.8 3.49e+002 -4.9 1.00e+000 3.55e-002h 1
173 1.1545675e+002 5.43e+001 1.62e+001 -3.8 3.66e+002 -5.4 1.49e-001 2.16e-001h 1
174 1.1545941e+002 4.66e+001 1.40e+001 -3.8 5.94e+002 -5.8 1.97e-001 1.45e-001h 1
175 1.1251362e+002 3.14e+001 2.07e+001 -3.8 1.46e+003 -6.3 4.09e-003 5.00e-001h 2
176 1.0909732e+002 5.24e+001 2.17e+001 -3.8 1.12e+005 -5.9 6.44e-004 9.01e-003f 2
177 1.0972329e+002 4.39e+001 1.82e+001 -3.8 9.19e+002 -6.4 4.38e-001 1.63e-001h 1
178 1.0795690e+002 2.82e+001 1.82e+001 -3.8 1.83e+003 -5.9 3.80e-004 1.08e-001H 1
179 1.0772544e+002 1.76e+001 1.27e+001 -3.8 2.57e+002 -6.4 1.15e-002 3.40e-001h 1
iter   objective   inf_pr   inf_du lg(mu)  ||d|| lg(rg) alpha_du alpha_pr ls
180 1.0645927e+002 1.12e+001 1.56e+001 -3.8 1.49e+003 -6.9 6.03e-002 3.56e-001h 1
181 1.0665505e+002 9.96e+000 1.40e+001 -3.8 7.46e+002 -4.7 1.39e-001 9.82e-002h 1
182 1.0693419e+002 9.31e+000 1.31e+001 -3.8 2.69e+002 -4.2 1.70e-001 6.48e-002h 1
183 1.0714176e+002 8.25e+000 1.16e+001 -3.8 5.27e+002 -4.7 1.58e-001 1.09e-001h 1
184 1.0721437e+002 7.36e+000 1.03e+001 -3.8 3.12e+002 -5.2 1.81e-001 1.08e-001h 1
185 1.0717031e+002 5.61e+000 7.40e+000 -3.8 3.71e+002 -5.7 2.81e-001 2.47e-001h 1
186 1.0735766e+002 3.75e+000 4.35e+000 -3.8 2.52e+002 -6.2 1.28e-001 3.71e-001h 1
187 1.0757405e+002 1.20e+000 1.26e+000 -3.8 2.31e+002 -6.6 2.60e-001 6.51e-001h 1
188 1.0736537e+002 9.85e-001 9.75e-001 -3.8 3.01e+002 -5.3 2.22e-001 2.26e-001h 1
189 1.0724804e+002 7.30e-001 6.01e-001 -3.8 1.92e+002 -4.9 3.92e-001 3.98e-001h 1
iter   objective   inf_pr   inf_du lg(mu)  ||d|| lg(rg) alpha_du alpha_pr ls
190 1.0725223e+002 5.78e-001 4.76e-001 -3.8 6.48e+001 -4.4 6.14e-001 2.08e-001h 1
191 1.0646175e+002 4.95e+000 3.32e+000 -3.8 2.29e+002 -4.9 2.10e-001 8.23e-001h 1
192 1.0636671e+002 4.35e+000 2.92e+000 -3.8 2.25e+002 -5.4 7.82e-001 1.19e-001h 1
193 1.0642119e+002 3.46e+000 2.36e+000 -3.8 9.29e+001 -5.0 1.00e+000 2.06e-001h 1
194 1.0599766e+002 2.14e+000 1.32e+000 -3.8 2.99e+002 -5.5 4.28e-001 3.50e-001h 1
195 1.0590057e+002 6.50e-001 3.91e-001 -3.8 1.11e+002 -5.0 1.00e+000 6.50e-001h 1
196 1.0587632e+002 2.05e-001 2.85e-001 -3.8 6.32e+001 -4.6 9.49e-001 1.00e+000h 1
197 1.0530230e+002 2.17e+000 2.34e+000 -3.8 9.23e+002 -5.1 4.48e-002 1.59e-001f 1
198 1.0521833e+002 1.49e+000 1.61e+000 -3.8 6.20e+001 -4.7 6.11e-001 3.11e-001h 1
199 1.0508252e+002 1.35e+000 1.43e+000 -3.8 1.89e+002 -5.1 7.04e-002 1.07e-001h 1
iter   objective   inf_pr   inf_du lg(mu)  ||d|| lg(rg) alpha_du alpha_pr ls
200 1.0469443e+002 1.32e+000 1.15e+000 -3.8 4.66e+002 -5.6 1.15e-001 1.34e-001h 1
201 1.0439211e+002 1.33e+000 1.02e+000 -3.8 9.86e+002 -6.1 1.01e-001 4.85e-002h 1
202 1.0435698e+002 9.98e-001 7.73e-001 -3.8 5.72e+001 -4.8 1.75e-001 2.46e-001h 1
203 1.0436089e+002 5.06e-001 4.06e-001 -3.8 1.84e+001 -4.3 1.00e+000 4.95e-001h 1
204 1.0435698e+002 4.47e-003 2.40e-002 -3.8 9.03e+000 -3.9 8.17e-001 1.00e+000h 1
205 1.0431862e+002 2.52e-002 9.41e-002 -3.8 6.28e+001 -4.4 1.65e-001 1.90e-001h 1
206 1.0418205e+002 2.21e-002 1.15e-001 -3.8 1.37e+001 -4.0 1.12e-001 4.36e-001h 1
207 1.0395873e+002 9.32e-003 1.28e-001 -3.8 6.14e+000 -3.5 2.80e-001 8.52e-001f 1
208 1.0369948e+002 3.04e-002 2.73e-001 -3.8 4.13e+002 -4.0 6.36e-003 1.91e-002f 1
209 1.0369484e+002 3.04e-002 1.77e-001 -3.8 7.62e+002 -4.5 1.06e-002 9.30e-005h 1
iter   objective   inf_pr   inf_du lg(mu)  ||d|| lg(rg) alpha_du alpha_pr ls
210 1.0339571e+002 3.11e-002 2.00e-001 -3.8 3.44e+001 -4.1 1.09e-001 1.22e-001f 1
211 1.0317165e+002 2.85e-002 1.88e-001 -3.8 6.84e+001 -4.5 1.85e-002 8.92e-002h 1
212 1.0309052e+002 2.67e-002 1.76e-001 -3.8 2.18e+001 -4.1 9.94e-002 6.59e-002h 1
213 1.0288887e+002 2.72e-002 1.55e-001 -3.8 8.52e+001 -4.6 6.12e-002 9.10e-002h 1
214 1.0285420e+002 2.61e-002 1.49e-001 -3.8 1.87e+001 -4.2 1.86e-001 4.04e-002h 1
215 1.0274139e+002 2.50e-002 1.33e-001 -3.8 7.05e+001 -4.6 1.81e-001 8.06e-002h 1
216 1.0251850e+002 8.17e-002 3.09e-001 -3.8 1.06e+003 -5.1 4.35e-002 2.27e-002f 1
217 1.0227868e+002 7.78e-002 2.76e-001 -3.8 5.95e+001 -4.7 9.01e-002 1.96e-001h 1
218 1.0201351e+002 1.13e-001 3.18e-001 -3.8 2.20e+002 -5.2 7.94e-001 1.43e-001h 1

```

(continues on next page)

(continued from previous page)

219	1.0191683e+002	9.28e-002	2.56e-001	-3.8	5.83e+001	-4.7	3.19e-001	1.85e-001h	1
iter	objective	inf_pr	inf_du	lg(mu)	d	lg(rg)	alpha_du	alpha_pr	ls
220	1.0145399e+002	7.30e-001	4.70e-001	-3.8	2.79e+002	-5.2	5.67e-001	2.68e-001h	1
221	1.0063872e+002	2.16e+000	3.07e+000	-3.8	2.50e+002	-4.8	1.00e+000	7.02e-001h	1
222	1.0044644e+002	5.45e-002	2.13e-001	-3.8	6.24e+001	-4.4	1.00e+000	9.99e-001h	1
223	1.0031700e+002	5.19e-002	1.56e-001	-3.8	6.24e+001	-4.8	1.00e+000	3.28e-001h	1
224	9.9718832e+001	5.81e-001	9.56e-001	-3.8	1.17e+002	-5.3	1.00e+000	8.73e-001f	1
225	9.9108535e+001	1.34e+000	1.31e+000	-3.8	3.83e+002	-5.8	3.36e-001	2.94e-001h	1
226	9.8093059e+001	4.22e+000	2.81e+000	-3.8	3.09e+002	-5.4	4.56e-001	6.12e-001h	1
227	9.8315991e+001	2.78e+000	1.82e+000	-3.8	8.62e+001	-4.0	1.00e+000	3.52e-001h	1
228	9.8022717e+001	3.60e-001	3.72e-001	-3.8	7.60e+001	-4.5	5.34e-001	1.00e+000h	1
229	9.7355332e+001	1.16e+000	1.01e+000	-3.8	1.39e+002	-5.0	2.48e-001	7.02e-001h	1
iter	objective	inf_pr	inf_du	lg(mu)	d	lg(rg)	alpha_du	alpha_pr	ls
230	9.6087508e+001	3.68e+000	3.53e+000	-3.8	2.27e+002	-5.5	6.75e-001	1.00e+000h	1
231	9.6034280e+001	3.44e+000	3.30e+000	-3.8	2.32e+002	-5.0	1.02e-001	6.41e-002h	1
232	9.6190352e+001	1.27e+000	1.17e+000	-3.8	9.33e+001	-4.6	1.00e+000	6.19e-001h	1
233	9.5830048e+001	9.12e-001	1.43e+000	-3.8	1.87e+002	-5.1	2.71e-001	5.98e-001h	1
234	9.5240415e+001	1.28e+000	1.83e+000	-3.8	1.86e+002	-4.7	5.90e-001	4.91e-001h	1
235	9.4612201e+001	1.33e+000	1.15e+000	-3.8	3.51e+002	-5.1	1.64e-001	2.37e-001h	1
236	9.3452047e+001	2.76e+000	3.30e+000	-3.8	4.02e+002	-5.6	2.60e-001	4.60e-001h	1
237	9.2596866e+001	3.10e+000	2.09e+000	-3.8	1.12e+003	-5.2	1.57e-002	1.11e-001h	1
238	9.1618590e+001	2.88e+000	3.07e+000	-3.8	1.31e+003	-4.8	9.43e-002	9.90e-002h	1
239	9.1466051e+001	2.59e+000	2.73e+000	-3.8	3.42e+002	-5.2	2.99e-001	9.62e-002h	1
iter	objective	inf_pr	inf_du	lg(mu)	d	lg(rg)	alpha_du	alpha_pr	ls
240	9.0795018e+001	2.28e+000	1.95e+000	-3.8	1.04e+003	-5.7	2.70e-001	9.61e-002h	1
241	9.0825122e+001	2.06e+000	1.74e+000	-3.8	1.95e+002	-5.3	1.00e+000	9.43e-002h	1
242	8.9756097e+001	3.36e+000	2.19e+000	-3.8	2.68e+003	-5.8	4.66e-002	8.14e-002h	1
243	9.0270793e+001	2.39e+000	1.53e+000	-3.8	8.18e+001	-4.4	7.66e-001	2.97e-001h	1
244	9.0096434e+001	2.18e+000	1.50e+000	-3.8	3.32e+002	-4.9	3.68e-001	8.43e-002h	1
245	9.1013646e+001	8.15e-002	3.05e-001	-3.8	5.71e+001	-4.5	1.00e+000	9.51e-001h	1
246	9.0564951e+001	4.99e-001	1.02e+000	-3.8	9.32e+001	-5.0	8.13e-001	7.68e-001h	1
247	9.0340773e+001	4.40e-001	9.91e-001	-3.8	1.66e+002	-5.5	2.26e-001	1.99e-001h	1
248	9.0275084e+001	4.24e-001	9.62e-001	-3.8	2.36e+002	-5.9	2.78e-001	4.32e-002h	1
249	8.9963644e+001	4.81e-001	1.14e+000	-3.8	3.39e+002	-6.4	1.43e-001	1.54e-001h	1
iter	objective	inf_pr	inf_du	lg(mu)	d	lg(rg)	alpha_du	alpha_pr	ls
250	8.9476331e+001	1.21e+000	1.45e+000	-3.8	1.24e+003	-6.9	9.72e-002	6.88e-002h	1
251	8.9407240e+001	5.51e-001	1.58e+000	-3.8	9.15e+001	-4.7	1.00e+000	9.99e-001h	1
252	8.9378692e+001	9.17e-002	1.53e-001	-3.8	2.69e+001	-4.2	1.00e+000	9.06e-001h	1
253	8.9074534e+001	3.32e-001	4.21e-001	-3.8	5.33e+001	-4.7	1.00e+000	8.30e-001h	1
254	8.8956757e+001	3.17e-001	4.23e-001	-3.8	1.53e+002	-5.2	8.86e-001	9.31e-002h	1
255	8.7941468e+001	1.70e+000	5.09e+000	-3.8	2.73e+002	-5.7	3.62e-001	5.49e-001h	1
256	8.7620316e+001	1.38e+000	3.62e+000	-3.8	1.78e+002	-5.2	6.63e-001	2.41e-001h	1
257	8.7670392e+001	8.85e-001	2.17e+000	-3.8	8.62e+001	-4.8	1.00e+000	3.60e-001h	1
258	8.8061495e+001	3.25e-002	1.95e-001	-3.8	3.08e+001	-4.4	1.00e+000	1.00e+000h	1
259	8.7997534e+001	5.03e-002	1.70e-001	-3.8	6.68e+001	-4.9	2.49e-001	2.65e-001h	1
iter	objective	inf_pr	inf_du	lg(mu)	d	lg(rg)	alpha_du	alpha_pr	ls
260	8.7889911e+001	1.48e-001	5.11e-001	-3.8	2.83e+003	-5.3	1.45e-002	1.07e-002f	1
261	8.7850877e+001	1.51e-001	7.86e-001	-3.8	2.78e+002	-4.9	6.42e-001	2.44e-002h	1
262	8.7681871e+001	1.56e-001	3.11e-001	-3.8	3.57e+001	-4.5	1.00e+000	7.64e-001h	1
263	8.7428995e+001	2.80e-001	5.35e-001	-3.8	8.32e+001	-5.0	6.90e-001	4.93e-001h	1
264	8.7291842e+001	6.23e-002	1.89e-001	-3.8	2.81e+001	-4.5	1.00e+000	1.00e+000h	1
265	8.7284489e+001	1.04e-002	2.83e-002	-3.8	1.03e+001	-4.1	1.00e+000	1.00e+000h	1

(continues on next page)

(continued from previous page)

```

266 8.7260376e+001 1.83e-003 5.26e-003 -3.8 4.00e+000 -3.7 1.00e+000 1.00e+000h 1
267 8.7194400e+001 1.09e-001 2.23e-001 -3.8 1.98e+001 -4.2 1.00e+000 8.50e-001h 1
268 8.6981128e+001 3.61e-002 4.92e-002 -3.8 7.84e+000 -3.7 1.00e+000 1.00e+000h 1
269 8.6799113e+001 8.82e-002 2.08e-001 -3.8 2.07e+001 -4.2 1.00e+000 1.00e+000h 1
iter   objective   inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
270 8.6689873e+001 5.36e-002 1.72e-001 -3.8 2.90e+001 -4.7 1.00e+000 4.52e-001h 1
271 8.6550756e+001 1.09e-001 2.84e-001 -3.8 5.70e+001 -5.2 1.00e+000 4.06e-001h 1
272 8.6472602e+001 1.29e-001 3.38e-001 -3.8 2.59e+002 -5.6 2.18e-001 5.32e-002h 1
273 8.6348190e+001 2.05e-001 5.95e-001 -3.8 1.94e+002 -5.2 4.10e-001 1.44e-001h 1
274 8.6151228e+001 2.21e-001 8.33e-001 -3.8 7.90e+001 -4.8 8.82e-001 4.08e-001h 1
275 8.5934213e+001 8.36e-002 5.60e-001 -3.8 3.06e+001 -4.4 1.00e+000 1.00e+000h 1
276 8.5700421e+001 1.67e-001 9.23e-001 -3.8 1.03e+002 -4.8 1.19e-001 2.49e-001h 1
277 8.5101673e+001 5.97e-001 1.93e+000 -3.8 3.60e+002 -5.3 5.85e-002 1.73e-001f 1
278 8.4745154e+001 5.31e-001 1.61e+000 -3.8 1.72e+002 -5.8 4.91e-001 1.67e-001h 1
279 8.4294451e+001 5.09e-001 2.10e+000 -3.8 2.92e+002 -6.3 1.85e-001 1.60e-001h 1
iter   objective   inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
280 8.3895819e+001 6.33e-001 2.27e+000 -3.8 3.31e+002 -5.8 3.99e-001 1.51e-001h 1
281 8.3902083e+001 6.29e-001 2.26e+000 -3.8 2.38e+001 -4.5 1.00e+000 7.68e-003h 1
282 8.3870136e+001 4.72e-001 1.49e+000 -3.8 1.13e+002 -5.0 1.00e+000 3.22e-001h 1
283 8.4183246e+001 2.43e-001 7.66e-001 -3.8 3.69e+001 -4.6 1.00e+000 4.81e-001h 1
284 8.4608001e+001 3.44e-002 1.11e-001 -3.8 9.09e+000 -4.1 1.00e+000 8.51e-001h 1
285 8.4615601e+001 3.01e-002 2.86e-001 -3.8 2.41e+001 -4.6 1.00e+000 7.30e-001h 1
286 8.4588308e+001 9.39e-003 9.90e-002 -3.8 1.02e+001 -4.2 1.00e+000 1.00e+000h 1
287 8.4457666e+001 1.23e-001 1.27e+000 -3.8 4.57e+001 -4.7 1.00e+000 7.63e-001h 1
288 8.4330631e+001 1.26e-001 1.25e+000 -3.8 1.19e+002 -5.1 2.43e-001 1.07e-001h 1
289 8.4156090e+001 1.61e-001 1.46e+000 -3.8 5.44e+002 -5.6 3.12e-001 3.89e-002h 1
iter   objective   inf_pr   inf_du lg(mu)  ||d||  lg(rg) alpha_du alpha_pr  ls
290 8.4034261e+001 1.61e-001 1.39e+000 -3.8 1.97e+002 -6.1 1.25e-001 7.73e-002h 1
291 8.3848975e+001 1.77e-001 1.04e+000 -3.8 9.33e+001 -6.6 1.00e+000 3.81e-001h 1
292 8.3898784e+001 1.13e-001 8.75e-001 -3.8 3.73e+001 -7.1 1.00e+000 1.00e+000h 1
293 8.4074886e+001 3.43e-002 2.70e-001 -3.8 1.85e+001 -7.5 1.00e+000 1.00e+000h 1
294 8.4022386e+001 2.77e-003 1.59e-002 -3.8 5.56e+000 -8.0 1.00e+000 1.00e+000h 1
295 8.4025598e+001 7.08e-007 8.11e-006 -3.8 1.47e-001 -8.5 1.00e+000 1.00e+000h 1
296 8.4017779e+001 2.89e-005 2.29e-004 -5.7 5.26e-001 -9.0 9.97e-001 1.00e+000f 1
297 8.4017734e+001 4.90e-009 3.97e-008 -5.7 6.49e-003 -9.4 1.00e+000 1.00e+000h 1
298 8.4017636e+001 4.70e-009 3.72e-008 -8.6 6.69e-003 -9.9 1.00e+000 1.00e+000h 1
299 8.4017636e+001 1.43e-014 5.68e-014 -8.6 1.23e-006 -10.4 1.00e+000 1.00e+000h 1

```

Number of Iterations...: 299

```

                                (scaled)                (unscaled)
Objective...: 8.4017636352189939e+001  8.4017636352189939e+001
Dual infeasibility...: 5.6843418860808015e-014  5.6843418860808015e-014
Constraint violation...: 1.4311902357677653e-014  1.4311902357677653e-014
Complementarity...: 2.5059048518614286e-009  2.5059048518614286e-009
Overall NLP error...: 2.5059048518614286e-009  2.5059048518614286e-009

```

```

Number of objective function evaluations      = 315
Number of objective gradient evaluations      = 300
Number of equality constraint evaluations      = 315
Number of inequality constraint evaluations    = 0

```

(continues on next page)

(continued from previous page)

```

Number of equality constraint Jacobian evaluations = 300
Number of inequality constraint Jacobian evaluations = 0
Number of Lagrangian Hessian evaluations = 299
Total CPU secs in IPOPT (w/o function evaluations) = 13.914
Total CPU secs in NLP function evaluations = 2.054

```

EXIT: Optimal Solution Found.

S	:	t_proc	(avg)	t_wall	(avg)	n_eval
nlp_f		75.00ms	(238.10us)	75.11ms	(238.45us)	315
nlp_g		354.00ms	(1.12ms)	354.22ms	(1.12ms)	315
nlp_grad		0	(0)	0	(0)	1
nlp_grad_f		89.00ms	(295.68us)	91.04ms	(302.46us)	301
nlp_hess_l		816.00ms	(2.73ms)	813.18ms	(2.72ms)	299
nlp_jac_g		691.00ms	(2.30ms)	691.45ms	(2.30ms)	301
total		15.98 s	(15.98 s)	15.98 s	(15.98 s)	1

The first optimization will take rather long (4 seconds) but in the end we get:

EXIT: Optimal Solution Found.

which tells us that we found an optimal solution. Note that follow-up optimizations take around 100 ms due to warm-starting.

We can visualize the open-loop prediction with:

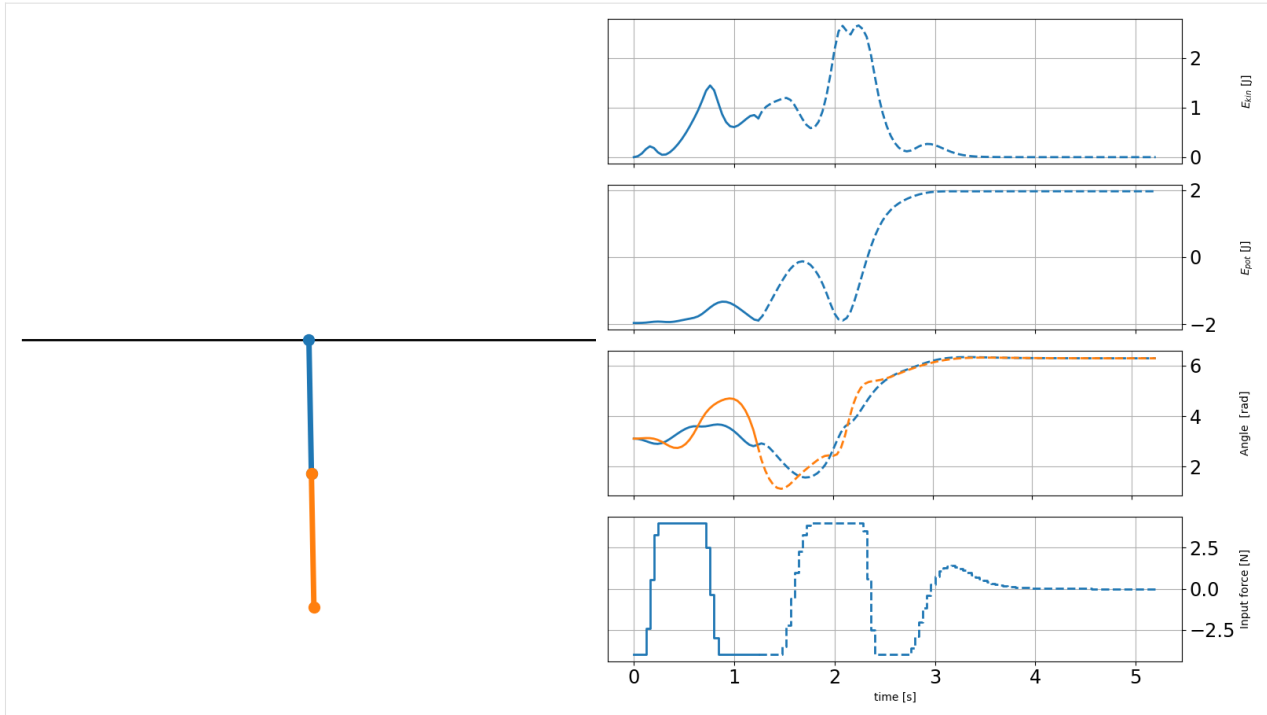
```

[37]: line1, line2 = pendulum_bars(x0)
      bar1[0].set_data(line1[0],line1[1])
      bar2[0].set_data(line2[0],line2[1])
      mpc_graphics.plot_predictions()
      mpc_graphics.reset_axes()

      fig

```

[37]:



The open-loop prediction looks perfectly fine! We see that within the horizon the potential energy settles on a plateau greater than zero, while the kinetic energy becomes zero. This indicates our desired up-up position. Both angles seem to reach 2π .

4.16.5.3 Run closed-loop

The closed-loop system is now simulated for 100 steps (and the output of the optimizer is suppressed):

```
[38]: %%capture
# Quickly reset the history of the MPC data object.
mpc.reset_history()

n_steps = 100
for k in range(n_steps):
    u0 = mpc.make_step(x0)
    y_next = simulator.make_step(u0)
    x0 = estimator.make_step(y_next)
```

4.16.5.4 Results

The next cell converts the results of the closed-loop MPC simulation into a gif (might take a few minutes):

```
[39]: from matplotlib.animation import FuncAnimation, FFMpegWriter, ImageMagickWriter

# The function describing the gif:
x_arr = mpc.data['_x']
def update(t_ind):
    line1, line2 = pendulum_bars(x_arr[t_ind])
```

(continues on next page)

(continued from previous page)

```

bar1[0].set_data(line1[0],line1[1])
bar2[0].set_data(line2[0],line2[1])
mpc_graphics.plot_results(t_ind)
mpc_graphics.plot_predictions(t_ind)
mpc_graphics.reset_axes()

anim = FuncAnimation(fig, update, frames=n_steps, repeat=False)
gif_writer = ImageMagickWriter(fps=20)
anim.save('anim_dip.gif', writer=gif_writer)

```

The result is shown below, where solid lines are the recorded trajectories and dashed lines are the predictions of the scenarios:

4.16.6 Controller with obstacle avoidance

To make the example even more interesting it is possible to add obstacles and include a set-point tracking task, where the pendulum must be erect at a desired location.

Please note that the animation below now shows the pendulum position (of the cart) as well as the desired setpoint instead of the angles.

The code to create this animation is included in the **do-mpc** example files and is just an extension of the example shown above.

The necessary changes include the detection of obstacle intersection and an adapted objective function that includes the set-point tracking for the position.

4.17 Efficient data generation and handling with do-mpc

This notebook was used in our video tutorial on [data generation and handling with do-mpc](#).

We start by importing basic modules and **do-mpc**.

```

[1]: import numpy as np
import sys
from casadi import *
import os
import time

# Add do_mpc to path. This is not necessary if it was installed via pip
import os
rel_do_mpc_path = os.path.join('..', '..', '..')
sys.path.append(rel_do_mpc_path)

# Import do_mpc package:
import do_mpc

import matplotlib.pyplot as plt

```

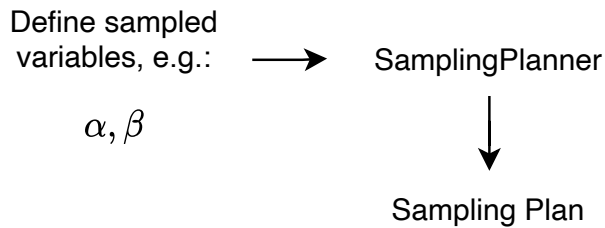
(continues on next page)

(continued from previous page)

```
import pandas as pd
```

4.17.1 Toy example

Step 1: Create the `sampling_plan` with the `SamplingPlanner`.



id	α	β	...
01	-1.221	4	...
02	0.034	0	...
\vdots	\vdots	\vdots	\vdots
n_{samples}	-0.689	3	...

The planner is initiated and we set some (optional) parameters.

```
[2]: sp = do_mpc.sampling.SamplingPlanner()
      sp.set_param(overwrite = True)
      # This generates the directory, if it does not exist already.
      sp.data_dir = './sampling_test/'
```

We then introduce new variables to the `SamplingPlanner` which will later jointly define a sampling case. Think of header rows in a table (see figure above).

These variables can themselves be sampled from a generating function or we add user defined cases one by one. If we want to sample variables to define the sampling case, we need to pass a sample generating function as shown below:

```
[3]: sp.set_sampling_var('alpha', np.random.randn)
      sp.set_sampling_var('beta', lambda: np.random.randint(0,5))
```

In this example we have two variables `alpha` and `beta`. We have:

$$\alpha \sim \mathcal{N}(\mu, \sigma)$$

and

$$\beta \sim \mathcal{U}([0, 5])$$

Having defined generating functions for **all of our variables**, we can now generate a sampling plan with an arbitrary amount of cases:

```
SamplingPlanner.gen_sampling_plan(n_samples)
```

```
[4]: plan = sp.gen_sampling_plan(n_samples=10)
```

We can inspect the plan conveniently by converting it to a pandas DataFrame. Natively, the plan is a list of dictionaries.

```
[5]: pd.DataFrame(plan)
```

```
[5]:
```

	alpha	beta	id
0	0.105326	0	000
1	0.784304	2	001
2	0.257489	1	002
3	1.552975	1	003
4	0.053229	3	004
5	1.041070	4	005
6	0.473513	0	006
7	0.917850	3	007
8	0.984259	0	008
9	0.715357	0	009

If we do not wish to automatically generate a sampling plan, we can also add sampling cases one by one with:

```
[6]: plan = sp.add_sampling_case(alpha=1, beta=-0.5)
      print(plan[-1])
```

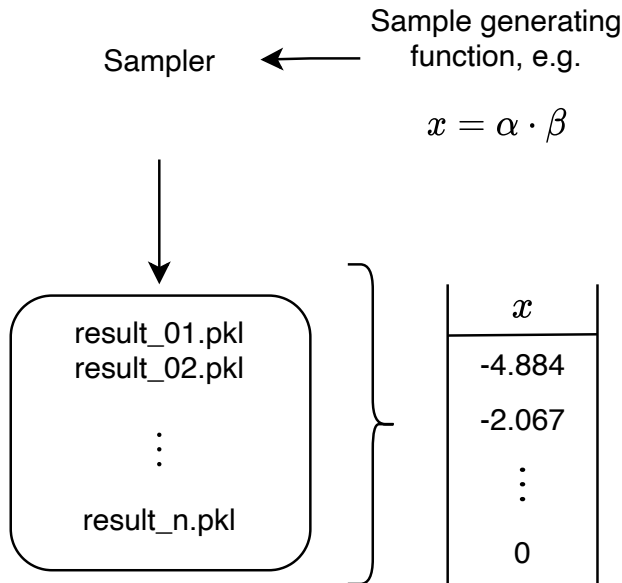
```
{'alpha': 1, 'beta': -0.5, 'id': '010'}
```

Typically, we finish the process of generating the sampling plan by saving it to the disc. This is simply done with:

```
sp.export(sampling_plan_name)
```

The save directory was already set with `sp.data_dir =`

Step 2: Create the Sampler object by providing the `sampling_plan`:



```
[7]: sampler = do_mpc.sampling.Sampler(plan)
sampler.set_param(overwrite = True)
```

Most important setting of the sampler is the `sample_function`. This function takes as arguments previously the defined `sampling_var` (from the configuration of the `SamplingPlanner`).

In this example, we create a dummy sampling generating function, where:

$$f(\alpha, \beta) = \alpha \cdot \beta$$

```
[8]: def sample_function(alpha, beta):
    time.sleep(0.1)
    return alpha*beta

sampler.set_sample_function(sample_function)
```

Before we sample, we want to set the directory for the created files and a name:

```
[9]: sampler.data_dir = './sampling_test/'
sampler.set_param(sample_name = 'dummy_sample')
```

Now we can actually create all the samples:

```
[10]: sampler.sample_data()

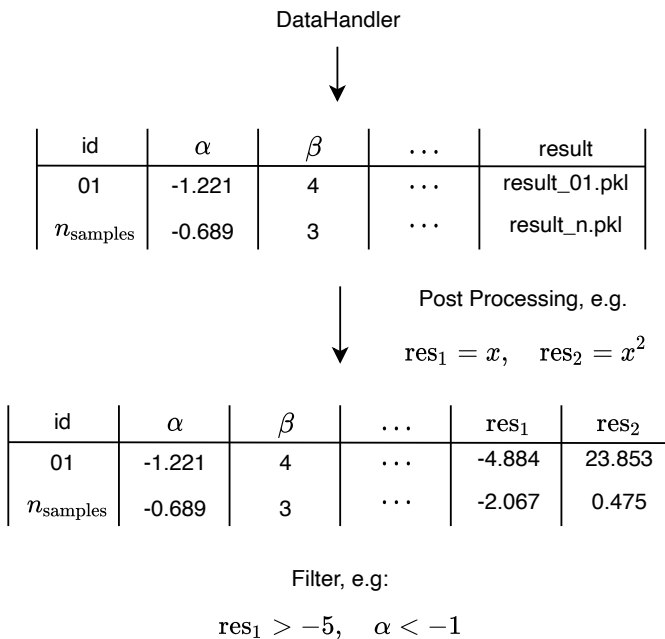
Progress: || 100.0% Complete
```

The sampler will now create the sampling results as a new file for each result and store them in a subfolder with the same name as the `sampling_plan`:

```
[11]: ls = os.listdir('./sampling_test/')
ls.sort()
ls
```

```
[11]: ['dummy_sample_000.pkl',
'dummy_sample_001.pkl',
'dummy_sample_002.pkl',
'dummy_sample_003.pkl',
'dummy_sample_004.pkl',
'dummy_sample_005.pkl',
'dummy_sample_006.pkl',
'dummy_sample_007.pkl',
'dummy_sample_008.pkl',
'dummy_sample_009.pkl',
'dummy_sample_010.pkl',
'dummy_sample_011.pkl',
'dummy_sample_012.pkl']
```

Step 3: Process data in the data handler class.



The first step is to initiate the class with the `sampling_plan`:

```
[12]: dh = do_mpc.sampling.DataHandler(plan)
```

We then need to point out where the data is stored and how the samples are called:

```
[13]: dh.data_dir = './sampling_test/'
dh.set_param(sample_name = 'dummy_sample')
```

Next, we define the post-processing functions. For this toy example we do some “dummy” post-processing and request to compute two results:

```
[14]: dh.set_post_processing('res_1', lambda x: x)
dh.set_post_processing('res_2', lambda x: x**2)
```

The interface of `DataHandler.set_post_processing` requires a name that we will see again later and a function that processes the output of the previously defined `sample_function`.

We can now obtain **obtained processed data** from the `DataHandler` in two ways. Note that we convert the returned list of dictionaries directly to a `DataFrame` for a better visualization.

1. Indexing:

```
[15]: pd.DataFrame(dh[:3])
```

```
[15]:
```

	alpha	beta	id	res_1	res_2
0	0.105326	0	000	0.000000	0.000000
1	0.784304	2	001	1.568608	2.460532
2	0.257489	1	002	0.257489	0.066301

Or we use a more complex filter with the `DataHandler.filter` method. This method requires either an input or an output filter in the form of a function.

Let's retrieve all samples, where $\alpha < 0$:

```
[16]: pd.DataFrame(dh.filter(input_filter = lambda alpha: alpha<0))
```

```
[16]: Empty DataFrame
Columns: []
Index: []
```

Or we can filter by outputs, e.g. with:

```
[17]: pd.DataFrame(dh.filter(output_filter = lambda res_2: res_2>10))
```

```
[17]:
```

	alpha	beta	id	res_1	res_2
0	1.04107	4	005	4.164281	17.341236

4.17.2 Sampling closed-loop trajectories

A more reasonable use-case in the scope of **do-mpc** is to sample closed-loop trajectories of a dynamical system with a (MPC) controller.

The approach is almost identical to our toy example above. The main difference lies in the `sample_function` that is passed to the `Sampler` and the `post_processing` in the `DataHandler`.

In the presented example, we will sample the oscillating mass system which is part of the `do-mpc` example library.

```
[18]: sys.path.append('../examples/oscillating_masses_discrete/')
from template_model import template_model
from template_mpc import template_mpc
from template_simulator import template_simulator
```

Step 1: Create the sampling plan with the `SamplingPlanner`

We want to generate various closed-loop trajectories of the system starting from random initial states, hence we design the `SamplingPlanner` as follows:

```
[19]: # Initialize sampling planner
sp = do_mpc.sampling.SamplingPlanner()
sp.set_param(overwrite=True)

# Sample random feasible initial states
def gen_initial_states():

    x0 = np.random.uniform(-3*np.ones((4,1)), 3*np.ones((4,1)))

    return x0

# Add sampling variable including the corresponding evaluation function
sp.set_sampling_var('X0', gen_initial_states)
```

This implementation is sufficient to generate the sampling plan:

```
[20]: plan = sp.gen_sampling_plan(n_samples=9)
```

Since we want to run the system in the closed-loop in our sample function, we need to load the corresponding configuration:

```
[21]: model = template_model()
mpc = template_mpc(model)
estimator = do_mpc.estimator.StateFeedback(model)
simulator = template_simulator(model)
```

We can now define the sampling function:

```
[22]: def run_closed_loop(X0):
    mpc.reset_history()
    simulator.reset_history()
    estimator.reset_history()

    # set initial values and guess
    x0 = X0
    mpc.x0 = x0
    simulator.x0 = x0
    estimator.x0 = x0

    mpc.set_initial_guess()

    # run the closed loop for 150 steps
    for k in range(100):
        u0 = mpc.make_step(x0)
        y_next = simulator.make_step(u0)
        x0 = estimator.make_step(y_next)

    # we return the complete data structure that we have obtained during the closed-loop.
    ↪run
    return simulator.data
```

Now we have all the ingredients to make our sampler:

```
[23]: %%capture
# Initialize sampler with generated plan
sampler = do_mpc.sampling.Sampler(plan)
# Set directory to store the results:
sampler.data_dir = './sampling_closed_loop/'
sampler.set_param(overwrite=True)

# Set the sampling function
sampler.set_sample_function(run_closed_loop)

# Generate the data
sampler.sample_data()
```

Step 3: Process data in the data handler class. The first step is to initiate the class with the `sampling_plan`:

```
[24]: # Initialize DataHandler
dh = do_mpc.sampling.DataHandler(plan)
dh.data_dir = './sampling_closed_loop/'
```

In this case, we are interested in the states and the inputs of all trajectories. We define the following post processing functions:

```
[25]: dh.set_post_processing('input', lambda data: data['_u', 'u'])
dh.set_post_processing('state', lambda data: data['_x', 'x'])
```

To retrieve all post-processed data from the datahandler we use slicing. The result is stored in `res`.

```
[26]: res = dh[:]
```

To inspect the sampled closed-loop trajectories, we create an array of plots where in each plot x_2 is plotted over x_1 . This shows the different behavior, based on the sampled initial conditions:

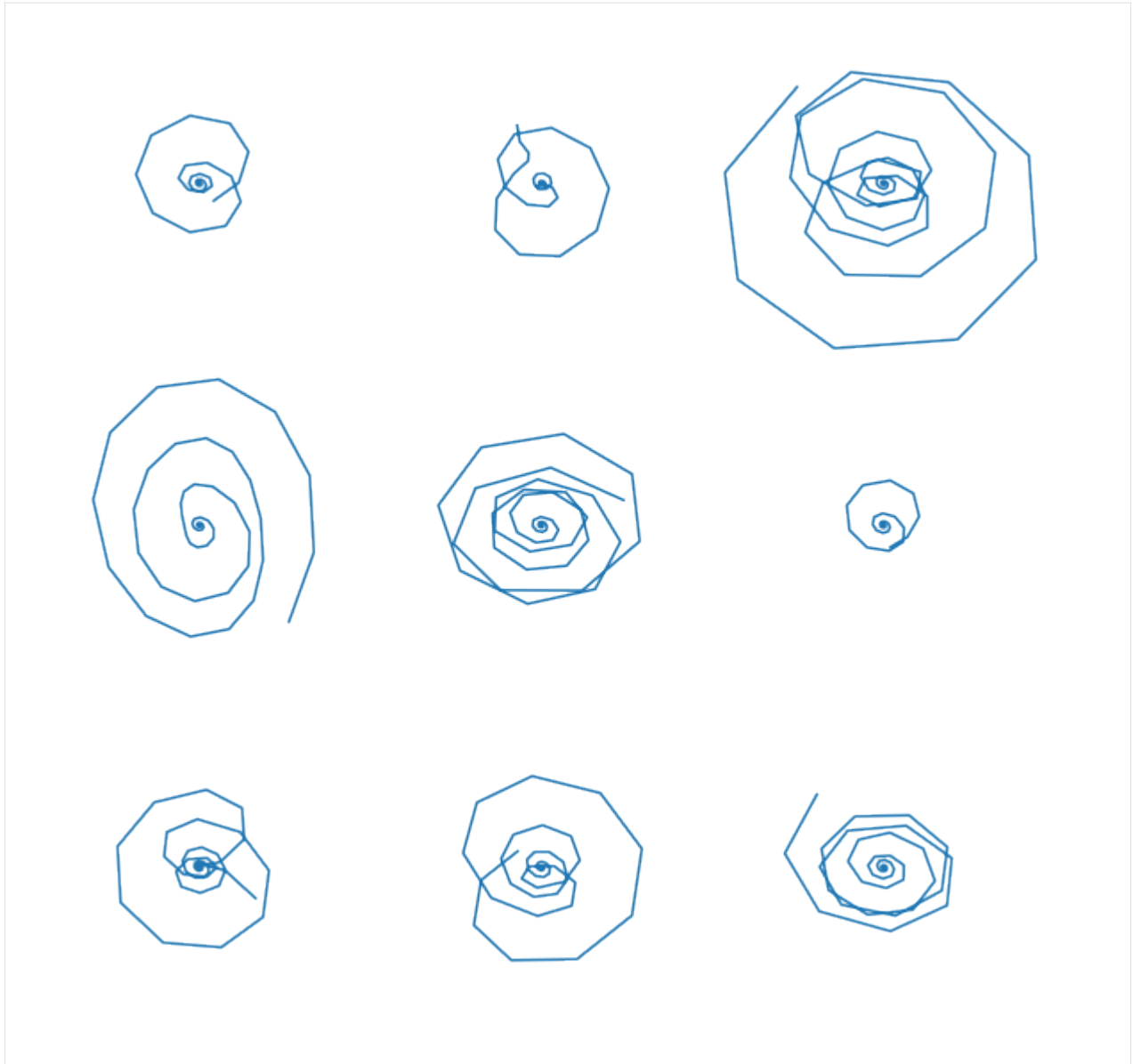
```
[27]: n_res = min(len(res), 80)

n_row = int(np.ceil(np.sqrt(n_res)))
n_col = int(np.ceil(n_res/n_row))

fig, ax = plt.subplots(n_row, n_col, sharex=True, sharey=True, figsize=(8,8))
for i, res_i in enumerate(res):
    ax[i//n_col, np.mod(i,n_col)].plot(res_i['state'][:,1], res_i['state'][:,0])

for i in range(ax.size):
    ax[i//n_col, np.mod(i,n_col)].axis('off')

fig.tight_layout(pad=0)
```



4.18 Continuous stirred tank reactor (CSTR) - LQR

In this Jupyter Notebook we illustrate the example CSTR. We design a Linear Quadratic Regulator(LQR) to regulate CSTR.

Open an interactive online Jupyter Notebook with this content on Binder:

The example consists of the three modules **template_model.py**, which describes the system model, **template_lqr.py**, which defines the settings for the control and **template_simulator.py**, which sets the parameters for the simulator. The modules are used in **main.py** for the closed-loop execution of the controller. The file **post_processing.py** is used for the visualization of the closed-loop control run.

In the following the different parts are presented. But first, we start by importing basic modules and **do-mpc**.


```
[1]: import numpy as np
import sys
from casadi import *
from casadi.tools import *
import matplotlib.pyplot as plt
import pdb

# Add do_mpc to path. This is not necessary if it was installed via pip
import os
rel_do_mpc_path = os.path.join('.', '..', '..')
sys.path.append(rel_do_mpc_path)

# Import do_mpc package:
import do_mpc
from do_mpc.tools import Timer
import pickle
import time
```

4.18.1 Model

In the following we will present the configuration, setup and connection between these blocks, starting with the model. The considered model of the CSTR is continuous and has 4 states and 2 control inputs. The model is initiated by:

```
[2]: model_type = 'continuous' # either 'discrete' or 'continuous'
model = do_mpc.model.Model(model_type)
```

4.18.1.1 States and control inputs

The four states are concentration of reactant A (C_A), the concentration of reactant B (C_B), the temperature inside the reactor (T_R) and the temperature of the cooling jacket (T_J):

```
[3]: # States struct (optimization variables):
C_a = model.set_variable(var_type='_x', var_name='C_a', shape=(1,1))
C_b = model.set_variable(var_type='_x', var_name='C_b', shape=(1,1))
T_R = model.set_variable(var_type='_x', var_name='T_R', shape=(1,1))
T_J = model.set_variable(var_type='_x', var_name='T_J', shape=(1,1))
```

The control inputs are the feed Fr and the heat removal by the jacket Q_J :

```
[4]: # Input struct (optimization variables):
Fr = model.set_variable(var_type='_u', var_name='Fr')
Q_J = model.set_variable(var_type='_u', var_name='Q_J')
```

4.18.1.2 ODE and parameters

The system model is described by the ordinary differential equation:

$$\dot{C}_A = \frac{Fr}{V} \cdot (C_{A_{in}} - C_A) - r_1, \quad (4.54)$$

$$\dot{C}_B = -\frac{Fr}{V} \cdot C_B + r_1 - r_2, \quad (4.55)$$

$$\dot{T}_R = \frac{Fr}{V} \cdot (T_{in} - T_R) - \frac{k \cdot A \cdot (T_R - T_J)}{\rho \cdot c_p \cdot V} + \frac{\Delta H_{R,1} \cdot (-r_1) + \Delta H_{R,2} \cdot (-r_2)}{\rho \cdot c_p}, \quad (4.56)$$

$$\dot{T}_J = \frac{-Q_J + k \cdot A \cdot (T_R - T_J)}{m_j \cdot C_{p,J}}, \quad (4.57)$$

$$(4.58)$$

where

$$r_1 = k_{0,1} \cdot \exp\left(\frac{-E_{R,1}}{T_R}\right) \cdot C_A \quad (4.59)$$

$$r_2 = k_{0,2} \cdot \exp\left(\frac{-E_{R,2}}{T_R}\right) \cdot C_B \quad (4.60)$$

$$(4.61)$$

```
[5]: # Certain parameters
K0_1 = 2.145e10      # [min^-1]
K0_2 = 2.145e10      # [min^-1]
E_R_1 = 9758.3       # [K]
E_R_2 = 9758.3       # [K]
delH_R_1 = -4200     # [kJ/kmol]
delH_R_2 = -11000    # [kJ/kmol]
T_in = 387.05        # [K]
rho = 934.2          # [kg/m^3]
cp = 3.01            # [kJ/m^3.K]
cp_J = 2             # [kJ/m^3.K]
m_j = 5              # [kg]
kA = 14.448          # [kJ/min.K]
C_ain = 5.1          # [kmol/m^3]
V = 0.01             # [m^3]
```

In the next step, we formulate the r_i -s:

```
[6]: # Auxiliary terms
r_1 = K0_1 * exp((-E_R_1)/((T_R))) * C_a
r_2 = K0_2 * exp((-E_R_2)/((T_R))) * C_b
```

With the help of the k_i -s and other available parameters we can define the ODEs:

```
[7]: # Differential equations
model.set_rhs('C_a', (Fr/V)*(C_ain-C_a)-r_1)
model.set_rhs('C_b', -(Fr/V)*C_b + r_1 - r_2)
model.set_rhs('T_R', (Fr/V)*(T_in-T_R)-(kA/(rho*cp*V))*(T_R-T_J)+(1/(rho*cp))*((delH_R_
  ↪ 1*(-r_1))+(delH_R_2*(-r_2))))
model.set_rhs('T_J', (1/(m_j*cp_J))*(-Q_J+kA*(T_R-T_J)))
```

Finally, the model setup is completed:

```
[8]: # Build the model
model.setup()
```

To design a LQR, we need a discrete Linear Time Invariant (LTI) system. In the following blocks of code, we will obtain such a model. Firstly, we will linearize a non-linear model around equilibrium point.

```
[9]: # Steady state values
F_ss = 0.002365 # [m^3/min]
Q_ss = 18.5583 # [kJ/min]

C_ass = 1.6329 # [kmol/m^3]
C_bss = 1.1101 # [kmolm^3]
T_Rss = 398.6581 # [K]
T_Jss = 397.3736 # [K]

uss = np.array([F_ss], [Q_ss])
xss = np.array([C_ass], [C_bss], [T_Rss], [T_Jss])

# Linearize the non-linear model
linearmodel = do_mpc.model.linearize(model, xss, uss)
```

Now we discretize the continuous LTI model with sampling time $t_{\text{step}} = 0.5$.

```
[10]: t_step = 0.5
model_dc = linearmodel.discretize(t_step, conv_method = 'zoh') # ['zoh', 'foh', 'bilinear',
↳ 'euler', 'backward_diff', 'impulse']

d:\Study_Materials\student_job\research_assistant\work_files\do_mpc_git\do-mpc\
↳ documentation\source\example_gallery\...\do_mpc\model\_linearmodel.py:296:
↳ UserWarning: sampling time is 0.5
warnings.warn('sampling time is {}'.format(t_step))
```

4.18.2 Controller

Now, we design Linear Quadratic Regulator for the above configured model. First, we create an instance of the class.

```
[11]: # Initialize the controller
lqr = do_mpc.controller.LQR(model_dc)
```

We choose the prediction horizon $n_{\text{horizon}} = 10$, the time step $t_{\text{step}} = 0.5$ s second.

```
[12]: # Initialize parameters
setup_lqr = {'t_step': t_step}
lqr.set_param(**setup_lqr)
```

4.18.2.1 Objective

The goal of CSTR is to drive the states to the desired set points.

Inputs:

Input	SetPoint
$F_{r,ref}$	$0.002365 \frac{m^3}{min}$
$Q_{J,ref}$	$18.5583 \frac{kJ}{min}$

States:

States	SetPoint
$C_{A,ref}$	$1.6329 \frac{kmol}{m^3}$
$C_{B,ref}$	$1.1101 \frac{kmol}{m^3}$
$T_{R,ref}$	$398.6581 K$
$T_{J,ref}$	$397.3736 K$

```
[13]: # Set objective
Q = 10*np.array([[1,0,0,0],[0,1,0,0],[0,0,0.01,0],[0,0,0,0.01]])
R = np.array([[1e-1,0],[0,1e-5]])

lqr.set_objective(Q=Q, R=R)
```

Now we run the LQR with the rated input. In order to do so, we set the cost matrix for the rated input as below:

```
[14]: Rdelu = np.array([[1e8,0],[0,1]])
lqr.set_rterm(delR = Rdelu)
```

Finally, LQR setup is completed

```
[15]: # set up lqr
lqr.setup()

d:\Study_Materials\student_job\research_assistant\work_files\do_mpc_git\do-mpc\
documentation\source\example_gallery\...\do_mpc\controller\_lqr.py:478:
UserWarning: discrete infinite horizon gain will be computed since prediction horizon
is set to default value 0
warnings.warn('discrete infinite horizon gain will be computed since prediction
horizon is set to default value 0')
```

4.18.3 Estimator

We assume, that all states can be directly measured (state-feedback):

```
[16]: estimator = do_mpc.estimator.StateFeedback(model)
```

4.18.4 Simulator

To create a simulator in order to run the LQR in a closed-loop, we create an instance of the **do-mpc** simulator which is based on the non-linear model:

```
[17]: simulator = do_mpc.simulator.Simulator(model)
```

For the simulation, we use the same time step `t_step` as for the optimizer:

```
[18]: params_simulator = {
    'integration_tool': 'cvodes',
    'abstol': 1e-10,
    'reltol': 1e-10,
    't_step': t_step
}

simulator.set_param(**params_simulator)
```

To finish the configuration of the simulator, call:

```
[19]: simulator.setup()
```

4.18.5 Closed-loop simulation

For the simulation of the LQR configured for the CSTR, we inspect the file **main.py**. We define the initial state of the system and set it for all parts of the closed-loop configuration. Furthermore, we set the desired destination for the states and input.

```
[20]: # Set the initial state of simulator:
C_a0 = 0
C_b0 = 0
T_R0 = 387.05
T_J0 = 387.05

x0 = np.array([C_a0, C_b0, T_R0, T_J0]).reshape(-1,1)

simulator.x0 = x0

lqr.set_setpoint(xss=xss,uss=uss)
```

Now, we simulate the closed-loop for 100 steps:

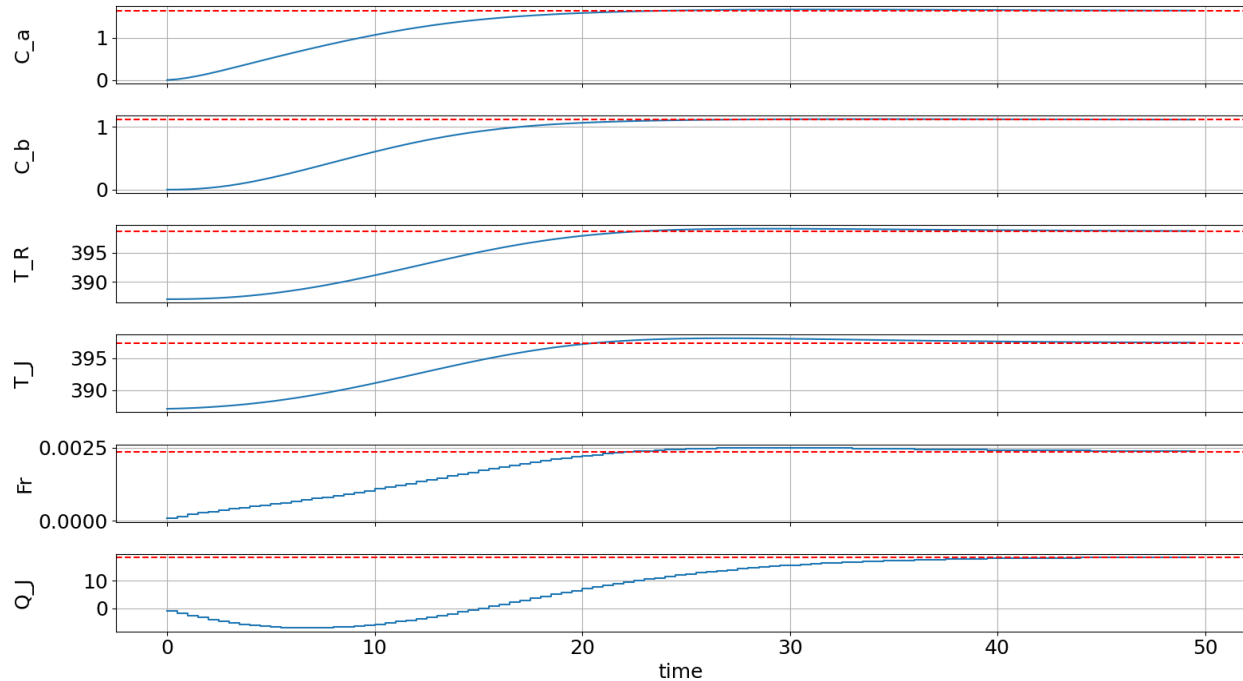
```
[21]: #Run LQR main loop:
sim_time = 100
for k in range(sim_time):
    u0 = lqr.make_step(x0)
    y_next = simulator.make_step(u0)
    x0 = y_next
```

4.18.6 Plotting

Now we plot the results obtained in the closed loop simulation.

```
[24]: from matplotlib import rcParams
rcParams['axes.grid'] = True
rcParams['font.size'] = 18
```

```
[25]: fig, ax, graphics = do_mpc.graphics.default_plot(simulator.data, figsize=(16,9))
graphics.plot_results()
graphics.reset_axes()
ax[0].axhline(y=C_ass,xmin=0,xmax=sim_time*t_step,color='r',linestyle='dashed')
ax[1].axhline(y=C_bss,xmin=0,xmax=sim_time*t_step,color='r',linestyle='dashed')
ax[2].axhline(y=T_Rss,xmin=0,xmax=sim_time*t_step,color='r',linestyle='dashed')
ax[3].axhline(y=T_Jss,xmin=0,xmax=sim_time*t_step,color='r',linestyle='dashed')
ax[4].axhline(y=F_ss,xmin=0,xmax=sim_time*t_step,color='r',linestyle='dashed')
ax[5].axhline(y=Q_ss,xmin=0,xmax=sim_time*t_step,color='r',linestyle='dashed')
plt.show()
```



INDICES AND TABLES

- `genindex`
- `search`

BIBLIOGRAPHY

[Biegler2010] L.T. Biegler. Nonlinear Programming: Concepts, Algorithms, and Applications to Chemical Processes. SIAM, 2010.

PYTHON MODULE INDEX

d

- `do_mpc`, [58](#)
- `do_mpc.controller`, [60](#)
- `do_mpc.data`, [93](#)
- `do_mpc.differentiator`, [100](#)
- `do_mpc.differentiator.helper`, [104](#)
- `do_mpc.estimator`, [108](#)
- `do_mpc.graphics`, [139](#)
- `do_mpc.model`, [146](#)
- `do_mpc.opcua`, [172](#)
- `do_mpc.optimizer`, [181](#)
- `do_mpc.sampling`, [190](#)
- `do_mpc.simulator`, [200](#)
- `do_mpc.sysid`, [210](#)
- `do_mpc.tools`, [214](#)

Symbols

__getitem__() (*Data method*), 94
 __getitem__() (*DataHandler method*), 191
 __getitem__() (*LinearModel method*), 151
 __getitem__() (*MPCData method*), 97
 __getitem__() (*Model method*), 162
 __getitem__() (*ONNXConversion method*), 212

A

abstol (*ContinuousSimulatorSettings attribute*), 209
 active_set_tol (*NLPDifferentialSettings attribute*), 105
 Add() (*in module do_mpc.sysid.ONNXOperations*), 213
 add_line() (*in module do_mpc.graphics.Graphics*), 142
 add_namespace_url() (*in module do_mpc.opcua.RTClient*), 178
 add_sampling_case() (*in module do_mpc.sampling.SamplingPlanner*), 197
 add_variable_to_node() (*in module do_mpc.opcua.RTServer*), 179
 address (*ClientOpts attribute*), 173
 address (*ServerOpts attribute*), 181
 animate() (*in module do_mpc.graphics*), 140
 async_run() (*in module do_mpc.opcua.RTBase*), 175
 async_step_start() (*in module do_mpc.opcua.RTBase*), 175
 async_step_stop() (*in module do_mpc.opcua.RTBase*), 175
 aux (*LinearModel attribute*), 155
 aux (*Model attribute*), 167

B

bounds (*MHE attribute*), 124
 bounds (*MPC attribute*), 79
 bounds (*Optimizer attribute*), 186

C

check_for_mandatory_settings() (*in module do_mpc.controller.LQRSettings*), 68
 check_for_mandatory_settings() (*in module do_mpc.controller.MPCSettings*), 89

check_for_mandatory_settings() (*in module do_mpc.estimator.MHESettings*), 134
 check_for_mandatory_settings() (*in module do_mpc.simulator.ContinuousSimulatorSettings*), 209
 check_for_mandatory_settings() (*in module do_mpc.simulator.SimulatorSettings*), 208
 check_LICQ (*NLPDifferentialSettings attribute*), 105
 check_rank (*NLPDifferentialSettings attribute*), 105
 check_SC (*NLPDifferentialSettings attribute*), 105
 clear() (*in module do_mpc.graphics.Graphics*), 143
 ClientOpts (*class in do_mpc.opcua*), 172
 collocation_deg (*MHESettings attribute*), 135
 collocation_deg (*MPCSettings attribute*), 90
 collocation_ni (*MHESettings attribute*), 135
 collocation_ni (*MPCSettings attribute*), 90
 collocation_type (*MHESettings attribute*), 135
 collocation_type (*MPCSettings attribute*), 90
 compile_nlp() (*in module do_mpc.controller.MPC*), 70
 compile_nlp() (*in module do_mpc.estimator.MHE*), 114
 compile_nlp() (*in module do_mpc.optimizer.Optimizer*), 181
 Concat() (*in module do_mpc.sysid.ONNXOperations*), 213
 connect() (*in module do_mpc.opcua.RTBase*), 175
 connect() (*in module do_mpc.opcua.RTClient*), 178
 cons_check_colloc_points (*MHESettings attribute*), 135
 cons_check_colloc_points (*MPCSettings attribute*), 90
 ContinuousSimulatorSettings (*class in do_mpc.simulator*), 209
 convert() (*in module do_mpc.sysid.ONNXConversion*), 212
 create_nlp() (*in module do_mpc.controller.MPC*), 70
 create_nlp() (*in module do_mpc.estimator.MHE*), 115
 create_nlp() (*in module do_mpc.optimizer.Optimizer*), 182

D

dae2odeconversion() (*in module do_mpc.model*), 146

Data (class in *do_mpc.data*), 94
 data_dir (DataHandler attribute), 194
 data_dir (Sampler attribute), 197
 data_dir (SamplingPlanner attribute), 200
 DataHandler (class in *do_mpc.sampling*), 190
 default_plot() (in module *do_mpc.graphics*), 140
 differentiate() (in module *do_mpc.differentiator.DoMPCDifferentiator*), 101
 differentiate() (in module *do_mpc.differentiator.NLPDifferentiator*), 103
 disconnect() (in module *do_mpc.opcua.RTBase*), 176
 disconnect() (in module *do_mpc.opcua.RTClient*), 178
 discrete_gain() (in module *do_mpc.controller.LQR*), 61
 discretize() (in module *do_mpc.model.LinearModel*), 151
 do_mpc
 module, 58
 do_mpc.controller
 module, 60
 do_mpc.data
 module, 93
 do_mpc.differentiator
 module, 100
 do_mpc.differentiator.helper
 module, 104
 do_mpc.estimator
 module, 108
 do_mpc.graphics
 module, 139
 do_mpc.model
 module, 146
 do_mpc.opcua
 module, 172
 do_mpc.optimizer
 module, 181
 do_mpc.sampling
 module, 190
 do_mpc.simulator
 module, 200
 do_mpc.sysid
 module, 210
 do_mpc.tools
 module, 214
 DoMPCDifferentiator (class in *do_mpc.differentiator*), 100

E

EKF (class in *do_mpc.estimator*), 108
 Elu() (in module *do_mpc.sysid.ONNXOperations*), 213
 entry_list (Namespace attribute), 173
 Estimator (class in *do_mpc.estimator*), 110

export() (in module *do_mpc.data.Data*), 95
 export() (in module *do_mpc.data.MPCData*), 98
 export() (in module *do_mpc.sampling.SamplingPlanner*), 198

F

filter() (in module *do_mpc.sampling.DataHandler*), 191
 full (Structure attribute), 217
 full_rank (NLPDifferentiatorStatus attribute), 107

G

Gemm() (in module *do_mpc.sysid.ONNXOperations*), 213
 gen_sampling_plan() (in module *do_mpc.sampling.SamplingPlanner*), 198
 get_default_namespace() (in module *do_mpc.opcua.RTBase*), 176
 get_index (Structure attribute), 217
 get_linear_system_matrices() (in module *do_mpc.model.LinearModel*), 152
 get_linear_system_matrices() (in module *do_mpc.model.Model*), 162
 get_node_id() (in module *do_mpc.opcua.NamespaceEntry*), 174
 get_p_template() (in module *do_mpc.controller.MPC*), 71
 get_p_template() (in module *do_mpc.estimator.MHE*), 115
 get_p_template() (in module *do_mpc.simulator.Simulator*), 201
 get_steady_state() (in module *do_mpc.model.LinearModel*), 152
 get_tvp_template() (in module *do_mpc.controller.MPC*), 72
 get_tvp_template() (in module *do_mpc.estimator.MHE*), 116
 get_tvp_template() (in module *do_mpc.optimizer.Optimizer*), 182
 get_tvp_template() (in module *do_mpc.simulator.Simulator*), 202
 get_y_template() (in module *do_mpc.estimator.MHE*), 116
 getter() (in module *do_mpc.tools.IndexedProperty*), 216

Graphics (class in *do_mpc.graphics*), 141

H

hist() (in module *do_mpc.tools.Timer*), 218

I

IndexedProperty (class in *do_mpc.tools*), 216
 info() (in module *do_mpc.tools.Timer*), 218
 init_algebraic_variables() (in module *do_mpc.simulator.Simulator*), 202

init_storage() (in module *do_mpc.data.Data*), 95
 init_storage() (in module *do_mpc.data.MPCData*), 98
 integration_tool (ContinuousSimulatorSettings attribute), 209
 IteratedVariables (class in *do_mpc.model*), 148

L

lb_opt_x (MHE attribute), 125
 lb_opt_x (MPC attribute), 80
 lb_opt_x (Optimizer attribute), 187
 LICQ (NLPDifferenziatorStatus attribute), 107
 lin_solver (NLPDifferenziatorSettings attribute), 106
 linearize() (in module *do_mpc.model*), 147
 LinearModel (class in *do_mpc.model*), 150
 load_pickle() (in module *do_mpc.tools*), 215
 load_results() (in module *do_mpc.data*), 93
 LQR (class in *do_mpc.controller*), 60
 LQRSettings (class in *do_mpc.controller*), 67
 lse_solved (NLPDifferenziatorStatus attribute), 107
 lstsq_fallback (NLPDifferenziatorSettings attribute), 106

M

make_step() (in module *do_mpc.controller.LQR*), 62
 make_step() (in module *do_mpc.controller.MPC*), 72
 make_step() (in module *do_mpc.estimator.EKF*), 108
 make_step() (in module *do_mpc.estimator.MHE*), 117
 make_step() (in module *do_mpc.estimator.StateFeedback*), 137
 make_step() (in module *do_mpc.opcua.RTBase*), 176
 make_step() (in module *do_mpc.simulator.Simulator*), 202
 MatMul() (in module *do_mpc.sysid.ONNXOperations*), 213
 meas_from_data (MHESettings attribute), 135
 MHE (class in *do_mpc.estimator*), 113
 MHESettings (class in *do_mpc.estimator*), 133
 Model (class in *do_mpc.model*), 161
 module
 do_mpc, 58
 do_mpc.controller, 60
 do_mpc.data, 93
 do_mpc.differentiator, 100
 do_mpc.differentiator.helper, 104
 do_mpc.estimator, 108
 do_mpc.graphics, 139
 do_mpc.model, 146
 do_mpc.opcua, 172
 do_mpc.optimizer, 181
 do_mpc.sampling, 190
 do_mpc.simulator, 200
 do_mpc.sysid, 210
 do_mpc.tools, 214

MPC (class in *do_mpc.controller*), 68
 MPCData (class in *do_mpc.data*), 96
 MPCSettings (class in *do_mpc.controller*), 88
 Mul() (in module *do_mpc.sysid.ONNXOperations*), 213

N

n_horizon (LQRSettings attribute), 68
 n_horizon (MHESettings attribute), 136
 n_horizon (MPCSettings attribute), 90
 n_robust (MPCSettings attribute), 91
 name (ClientOpts attribute), 172
 name (ServerOpts attribute), 180
 Namespace (class in *do_mpc.opcua*), 173
 namespace_from_client() (in module *do_mpc.opcua.RTServer*), 180
 namespace_from_model() (in module *do_mpc.opcua.RTBase*), 176
 namespace_name (Namespace attribute), 173
 NamespaceEntry (class in *do_mpc.opcua*), 173
 nl_cons_check_colloc_points (MHESettings attribute), 136
 nl_cons_check_colloc_points (MPCSettings attribute), 91
 nl_cons_single_slack (MHESettings attribute), 136
 nl_cons_single_slack (MPCSettings attribute), 91
 nlp_cons (MHE attribute), 125
 nlp_cons (MPC attribute), 80
 nlp_cons (Optimizer attribute), 187
 nlp_cons_lb (MHE attribute), 126
 nlp_cons_lb (MPC attribute), 81
 nlp_cons_lb (Optimizer attribute), 188
 nlp_cons_ub (MHE attribute), 126
 nlp_cons_ub (MPC attribute), 81
 nlp_cons_ub (Optimizer attribute), 188
 nlp_obj (MHE attribute), 127
 nlp_obj (MPC attribute), 82
 nlp_obj (Optimizer attribute), 188
 NLPDifferenziator (class in *do_mpc.differentiator*), 102
 NLPDifferenziatorSettings (class in *do_mpc.differentiator.helper*), 104
 NLPDifferenziatorStatus (class in *do_mpc.differentiator.helper*), 106
 nlpsol_opts (MHESettings attribute), 137
 nlpsol_opts (MPCSettings attribute), 92

O

objectnode (NamespaceEntry attribute), 174
 ONNXConversion (class in *do_mpc.sysid*), 210
 ONNXOperations (class in *do_mpc.sysid*), 212
 open_loop (MPCSettings attribute), 91
 opt_p (MHE attribute), 127
 opt_p (MPC attribute), 82
 opt_p_num (MHE attribute), 128

opt_p_num (MPC attribute), 83
 opt_x (MHE attribute), 129
 opt_x (MPC attribute), 83
 opt_x_num (MHE attribute), 129
 opt_x_num (MPC attribute), 84
 Optimizer (class in *do_mpc.optimizer*), 181

P

p (LinearModel attribute), 156
 p (Model attribute), 168
 p_est0 (MHE attribute), 130
 plot_predictions() (in module *do_mpc.graphics.Graphics*), 143
 plot_results() (in module *do_mpc.graphics.Graphics*), 144
 port (ClientOpts attribute), 173
 port (ServerOpts attribute), 181
 pred_lines (Graphics attribute), 145
 prediction() (in module *do_mpc.data.MPCData*), 98
 prepare_nlp() (in module *do_mpc.controller.MPC*), 73
 prepare_nlp() (in module *do_mpc.estimator.MHE*), 118
 prepare_nlp() (in module *do_mpc.optimizer.Optimizer*), 183
 printProgressBar() (in module *do_mpc.tools*), 215
 product() (in module *do_mpc.sampling.SamplingPlanner*), 198

R

read_from_tags() (in module *do_mpc.opcua.RTBase*), 176
 readData() (in module *do_mpc.opcua.RTClient*), 178
 reduced_nlp (NLPDifferentialiatorStatus attribute), 107
 reltol (ContinuousSimulatorSettings attribute), 210
 Relu() (in module *do_mpc.sysid.ONNXOperations*), 213
 reset_axes() (in module *do_mpc.graphics.Graphics*), 144
 reset_history() (in module *do_mpc.controller.LQR*), 62
 reset_history() (in module *do_mpc.controller.MPC*), 73
 reset_history() (in module *do_mpc.estimator.EKF*), 108
 reset_history() (in module *do_mpc.estimator.Estimator*), 111
 reset_history() (in module *do_mpc.estimator.MHE*), 118
 reset_history() (in module *do_mpc.estimator.StateFeedback*), 137
 reset_history() (in module *do_mpc.optimizer.Optimizer*), 184
 reset_history() (in module *do_mpc.simulator.Simulator*), 203

reset_prop_cycle() (in module *do_mpc.graphics.Graphics*), 144
 Reshape() (in module *do_mpc.sysid.ONNXOperations*), 213
 residuals (NLPDifferentialiatorStatus attribute), 107
 result_lines (Graphics attribute), 145
 RTBase (class in *do_mpc.opcua*), 174
 RTClient (class in *do_mpc.opcua*), 177
 RTServer (class in *do_mpc.opcua*), 179

S

sample_data() (in module *do_mpc.sampling.Sampler*), 195
 sample_idx() (in module *do_mpc.sampling.Sampler*), 195
 Sampler (class in *do_mpc.sampling*), 194
 SamplingPlanner (class in *do_mpc.sampling*), 197
 save_pickle() (in module *do_mpc.tools*), 215
 save_results() (in module *do_mpc.data*), 93
 SC (NLPDifferentialiatorStatus attribute), 107
 scaling (MHE attribute), 130
 scaling (MPC attribute), 85
 scaling (Optimizer attribute), 189
 sens_num (DoMPCDifferentialiator attribute), 101
 ServerOpts (class in *do_mpc.opcua*), 180
 set_alg() (in module *do_mpc.model.LinearModel*), 153
 set_alg() (in module *do_mpc.model.Model*), 163
 set_default_objective() (in module *do_mpc.estimator.MHE*), 118
 set_expression() (in module *do_mpc.model.LinearModel*), 153
 set_expression() (in module *do_mpc.model.Model*), 163
 set_initial_guess() (in module *do_mpc.controller.MPC*), 73
 set_initial_guess() (in module *do_mpc.estimator.MHE*), 119
 set_initial_guess() (in module *do_mpc.simulator.Simulator*), 203
 set_lam_zero (NLPDifferentialiatorSettings attribute), 106
 set_linear_solver() (in module *do_mpc.controller.MPCSettings*), 89
 set_linear_solver() (in module *do_mpc.estimator.MHESettings*), 134
 set_meas() (in module *do_mpc.model.LinearModel*), 153
 set_meas() (in module *do_mpc.model.Model*), 164
 set_meta() (in module *do_mpc.data.Data*), 96
 set_meta() (in module *do_mpc.data.MPCData*), 99
 set_nl_cons() (in module *do_mpc.controller.MPC*), 74
 set_nl_cons() (in module *do_mpc.estimator.MHE*), 120

set_nl_cons() (in module *do_mpc.optimizer.Optimizer*), 184
 set_objective() (in module *do_mpc.controller.LQR*), 62
 set_objective() (in module *do_mpc.controller.MPC*), 74
 set_objective() (in module *do_mpc.estimator.MHE*), 120
 set_p_fun() (in module *do_mpc.controller.MPC*), 75
 set_p_fun() (in module *do_mpc.estimator.MHE*), 122
 set_p_fun() (in module *do_mpc.simulator.Simulator*), 203
 set_param() (in module *do_mpc.controller.LQR*), 63
 set_param() (in module *do_mpc.controller.MPC*), 76
 set_param() (in module *do_mpc.estimator.MHE*), 122
 set_param() (in module *do_mpc.sampling.DataHandler*), 192
 set_param() (in module *do_mpc.sampling.Sampler*), 196
 set_param() (in module *do_mpc.sampling.SamplingPlanner*), 199
 set_param() (in module *do_mpc.simulator.Simulator*), 204
 set_post_processing() (in module *do_mpc.sampling.DataHandler*), 193
 set_read_tags() (in module *do_mpc.opcua.RTBase*), 177
 set_rhs() (in module *do_mpc.model.LinearModel*), 154
 set_rhs() (in module *do_mpc.model.Model*), 165
 set_rterm() (in module *do_mpc.controller.LQR*), 64
 set_rterm() (in module *do_mpc.controller.MPC*), 76
 set_sample_function() (in module *do_mpc.sampling.Sampler*), 196
 set_sampling_var() (in module *do_mpc.sampling.SamplingPlanner*), 199
 set_setpoint() (in module *do_mpc.controller.LQR*), 65
 set_tvp_fun() (in module *do_mpc.controller.MPC*), 77
 set_tvp_fun() (in module *do_mpc.estimator.MHE*), 123
 set_tvp_fun() (in module *do_mpc.optimizer.Optimizer*), 185
 set_tvp_fun() (in module *do_mpc.simulator.Simulator*), 205
 set_uncertainty_values() (in module *do_mpc.controller.MPC*), 78
 set_variable() (in module *do_mpc.model.LinearModel*), 154
 set_variable() (in module *do_mpc.model.Model*), 166
 set_write_tags() (in module *do_mpc.opcua.RTBase*), 177
 set_y_fun() (in module *do_mpc.estimator.MHE*), 124
 setter() (in module *do_mpc.tools.IndexedProperty*), 216
 settings (*DoMPCDifferentiator* attribute), 101
 settings (*NLPDifferentiator* attribute), 104
 setup() (in module *do_mpc.controller.LQR*), 65
 setup() (in module *do_mpc.controller.MPC*), 79
 setup() (in module *do_mpc.estimator.MHE*), 124
 setup() (in module *do_mpc.model.LinearModel*), 155
 setup() (in module *do_mpc.model.Model*), 167
 setup() (in module *do_mpc.simulator.Simulator*), 206
 Shape() (in module *do_mpc.sysid.ONNXOperations*), 214
 Sigmoid() (in module *do_mpc.sysid.ONNXOperations*), 214
 simulate() (in module *do_mpc.simulator.Simulator*), 206
 Simulator (class in *do_mpc.simulator*), 201
 SimulatorSettings (class in *do_mpc.simulator*), 208
 Slice() (in module *do_mpc.sysid.ONNXOperations*), 214
 solve() (in module *do_mpc.controller.MPC*), 79
 solve() (in module *do_mpc.estimator.MHE*), 124
 solve() (in module *do_mpc.optimizer.Optimizer*), 186
 Squeeze() (in module *do_mpc.sysid.ONNXOperations*), 214
 start() (in module *do_mpc.opcua.RTServer*), 180
 state_discretization (*MHESettings* attribute), 136
 state_discretization (*MPCSettings* attribute), 91
 StateFeedback (class in *do_mpc.estimator*), 137
 status (*DoMPCDifferentiator* attribute), 102
 status (*NLPDifferentiator* attribute), 104
 stop() (in module *do_mpc.opcua.RTServer*), 180
 store_full_solution (*MHESettings* attribute), 136
 store_full_solution (*MPCSettings* attribute), 92
 store_lagr_multiplier (*MHESettings* attribute), 136
 store_lagr_multiplier (*MPCSettings* attribute), 92
 store_solver_stats (*MHESettings* attribute), 137
 store_solver_stats (*MPCSettings* attribute), 92
 Structure (class in *do_mpc.tools*), 216
 Sub() (in module *do_mpc.sysid.ONNXOperations*), 214
 Sum() (in module *do_mpc.sysid.ONNXOperations*), 214
 supress_ipopt_output() (in module *do_mpc.controller.MPCSettings*), 90
 supress_ipopt_output() (in module *do_mpc.estimator.MHESettings*), 135
 sym_KKT (*NLPDifferentiatorStatus* attribute), 108
 sys_A (*LinearModel* attribute), 157
 sys_B (*LinearModel* attribute), 157
 sys_C (*LinearModel* attribute), 157
 sys_D (*LinearModel* attribute), 157
T
 t0 (*EKF* attribute), 109
 t0 (*Estimator* attribute), 111
 t0 (*IteratedVariables* attribute), 148
 t0 (*LQR* attribute), 65

`t0` (*MHE attribute*), 131
`t0` (*MPC attribute*), 86
`t0` (*Simulator attribute*), 206
`t0` (*StateFeedback attribute*), 138
`t_step` (*ContinuousSimulatorSettings attribute*), 210
`t_step` (*LQRSettings attribute*), 68
`t_step` (*MHESettings attribute*), 137
`t_step` (*MPCSettings attribute*), 92
`t_step` (*SimulatorSettings attribute*), 209
`Tanh()` (in module `do_mpc.sysid.ONNXOperations`), 214
`terminal_bounds` (*MPC attribute*), 86
`tic()` (in module `do_mpc.tools.Timer`), 218
`Timer` (class in `do_mpc.tools`), 218
`timeunit` (*ClientOpts attribute*), 172
`toc()` (in module `do_mpc.tools.Timer`), 218
`track_residuals` (*NLPDifferentiatorSettings attribute*), 106
`tvf` (*LinearModel attribute*), 157
`tvf` (*Model attribute*), 168

U

`u` (*LinearModel attribute*), 158
`u` (*Model attribute*), 169
`u0` (*EKF attribute*), 109
`u0` (*Estimator attribute*), 111
`u0` (*IteratedVariables attribute*), 148
`u0` (*LQR attribute*), 66
`u0` (*MHE attribute*), 131
`u0` (*MPC attribute*), 86
`u0` (*Simulator attribute*), 207
`u0` (*StateFeedback attribute*), 138
`ub_opt_x` (*MHE attribute*), 132
`ub_opt_x` (*MPC attribute*), 87
`ub_opt_x` (*Optimizer attribute*), 190
`Unsqueeze()` (in *module* `do_mpc.sysid.ONNXOperations`), 214
`update()` (in module `do_mpc.data.Data`), 96
`update()` (in module `do_mpc.data.MPCData`), 99
`use_terminal_bounds` (*MPCSettings attribute*), 92

V

`v` (*LinearModel attribute*), 158
`v` (*Model attribute*), 169
`variable` (*NamespaceEntry attribute*), 174

W

`w` (*LinearModel attribute*), 159
`w` (*Model attribute*), 170
`write_to_tags()` (in module `do_mpc.opcua.RTBase`), 177
`writeData()` (in module `do_mpc.opcua.RTClient`), 179

X

`x` (*LinearModel attribute*), 159

`x` (*Model attribute*), 170
`x0` (*EKF attribute*), 109
`x0` (*Estimator attribute*), 112
`x0` (*IteratedVariables attribute*), 149
`x0` (*LQR attribute*), 66
`x0` (*MHE attribute*), 132
`x0` (*MPC attribute*), 87
`x0` (*Simulator attribute*), 207
`x0` (*StateFeedback attribute*), 138

Y

`y` (*LinearModel attribute*), 160
`y` (*Model attribute*), 171

Z

`z` (*LinearModel attribute*), 160
`z` (*Model attribute*), 171
`z0` (*EKF attribute*), 110
`z0` (*Estimator attribute*), 112
`z0` (*IteratedVariables attribute*), 149
`z0` (*LQR attribute*), 67
`z0` (*MHE attribute*), 133
`z0` (*MPC attribute*), 88
`z0` (*Simulator attribute*), 208
`z0` (*StateFeedback attribute*), 139