nMPyC - A Python library for solving optimal control problems via MPC

Release 1.0.0

Jonas Schießl and Lisa Krügel

Oct 13, 2022

CONTENTS

nMPyC is a Python library for solving optimal control problems via model predictive control (MPC).

nMPyC can be understood as a blackbox method. The user can only enter the desired optimal control problem without having much knowledge of the theory of model predictive control or its implementation in Python. Nevertheless, for an advanced user, there is the possibility to adjust all parameters.

This library supports a variety of discretization methods and optimizers including CasADi and SciPy solvers.

In summary, nMPyC

- solves nonlinear finite horizon optimal control problems
- solves nonlinear optimal control problems with model predicitve control (MPC)
- uses algorithmic differentation via CasADi
- can chose between different discretization methods
- can chose between different solvers for nonlinear optimization (depending on the problem)
- supports time-varying optimal control problems
- supports the special structure of linear-quadratic optimal control problems
- supports discounted optimal control problems
- can save and load the simulation results

The **nMPyC** software is Python based and works therefore on any OS with a Python distribution (for more precise requiremnents see the *Installation* section). **nMPyC** has been developed by Jonas Schießl and Lisa Krügel under the supervision of Prof. Lars Grüne at the Chair of Applied Mathematic of University of Bayreuth. **nMPyC** is a further devolpement in Python of the Matlab code that was implemented for the NMPC Book from Lars Grüne and Jürgen Pannek [GruneP17].

CHAPTER

ONE

CONTENTS

1.1 Installation

1.1.1 Requirements

The nMPyC package is dependent on the following libraries:

- CasADi
- NumPy
- matplotlib
- SciPy
- dill
- osqp

1.1.2 Installation using PIP

The easiest way to install the nMPyC package is to use PIP. To do so, you just need to run the command line

```
pip install nmpyc
```

The main advantage of this method is that the package is automatically added to the Python default path and all dependencies are installed.

Additionally you can update the package by running

```
pip install nmpyc --upgrade
```

1.1.3 Installation by Source

To install the Python package by source, the source code from GitHub has to be downloaded. This can be done via Git using the command

```
git clone https://github.com/nMPyC/nmpyc
```

Now the toolbox can be used by importing the package according to its storage path in the Python code by adding it to the Python default path. To realize the letter case you can navigate to the location of the package and use

```
pip install .
```

This command will automatically add the package to the Python default path and install the required Python packages and their dependencies.

1.2 Getting Started

1.2.1 Import nMPyC

After the successfull installation of the nMPyC package, nMPyC has to be imported to our code. This can be done as shown in the following code snippet

```
# Add nMPyC to path (if necessary)
import sys
sys.path.append('../../path-to-nmpyc')

# Import nmpyc
import nmpyc
```

Note that the first two lines can be omitted if nMPyC has already been added to the Python default path as described in the *Installation* section. In this case the command `import nmpyc` is sufficient to import the nMPyC library.

Note: Please use the *nmpyc.nmpyc_array* functions and the *nmpyc.nmpyc_array.array* class for the calculations in the code to ensure error-free functionality of the program. Further informations about this issue can be found in *API References* and in the *FAQ* section.

1.2.2 Creating the System Dynamics

To define the system dynamics of the optimal control problem, we have to create a *nmpyc.system* object. We can define the possibly time-dependent and nonlinear system dynamics using a function of the form

```
def f(t,x,u):
    y = nmpyc.array(nx)
    ...
    return y
```

If this function is created, the system can be initialized by calling

```
system = nmpyc.system(f,nx,nu,system_type)
```

Where *nx* is the dimension of the state, *nu* is the dimension of the control variable, and *system_type* is a string indicating whether the system is continuous (*continuous*) or discrete (*discrete*).

Furthermore, the parameters *sampling_rate* (sampling rate), *t0* (initial time) and *method* can optionally be adjusted during the initialization of the system. The value of *method* determines the used integration method for the discretization of the differential equation in the continuous case. By default the CasADi integrator *cvodes* is used.

Further options of the used integration method can be defined by the command

```
system.set_integratorOptions(dict())
```

For more informations (also about the parameters and their standard values) see the API-References *nmpyc.system.system*.

1.2.3 Creating the Objective

To define the objective, we need to create – similar to the system dynamics – a *nmpyc.objective* object. To do so, we first define the stage cost

```
def l(t,x,u):
    ...
    return y
```

and add, optionally, a terminal cost of the form

```
def F(t,x):
    ...
    return y
```

Now we can initialize the objective by calling

```
objective = nmpyc.objective(l, F)
# Or alternatively without terminal costs
objective = nmpyc.objective(l)
```

For more informations see the API-References *nmpyc.objective.objective*.

1.2.4 Creating the Constraints

The optimal control problem can be extended with other constraints besides the necessary system dynamics. For this reason, we must first create an empty *nmpyc.constraints* object using the command

```
system = nmpyc.constraints()
```

We can now add the desired constraints to this object step by step. These constraints can be created in different ways. First, we can add box constraints in the form of bounds.

```
constraints.add_bound('lower', 'control', lbu) # lower bound for control
constraints.add_bound('upper', 'control', ubu) # upper bound for control
```

Here *lbu* or *lbx* is an *nmpyc.nmpyc_array.array* of dimension *(1,nu)* or *(nu,1)*. To add bounds for the state or terminal state, replace *control* with *state* or *terminal* in the above code and adjust the dimension of the array accordingly.

In addition to box constraints, general inequality and equality constraints can also be inserted.

```
# Equality constraint h(t,x,u) = 0
def h(t,x,u):
   y = mpc.array(len_constr)
   ...
   return y
constraints.add_constr('eq', h)

# Inequality constraint g(t,x,u) >= 0
def g(t,x,u):
```

(continues on next page)

(continued from previous page)

```
    y = mpc.array(len_constr)
    ...
    return y
constraints.add_constr('ineq', g)
```

Terminal constraints of the form $H(t,x) = 0$ or $G(t,x) \geq 0$ can also be added.

```
constraints.add_constr('terminal_eq', H)
constraints.add_constr('terminal_ineq', G)
```

Moreover it is possible to add linear equality and inequality constraints. For this purpose see *nmpyc.constraints.constraints.add_constr()*. For further general informations see the API-References *nmpyc.constraints.constraints*.

1.2.5 Running Simulations

After initializing all necessary objects, we can run simulations for our problem. We first create a *mpc.model* object and combine the different parts of the optimal control problem by calling

```
model = nmpyc.model(objective, system, constraints)
```

The *nmpyc.constraints* object is optional and can be omitted for a problem without constraints. Modyfying the default settings of the optimization, can be done with the help of the commands

```
model.opti.set_options(dict())
model.opti.set_solverOptions(dict())
```

For more informations about this methods see *nmpyc.model.model.opti*.

To start an open loop simulation, we execute the command

```
u_ol, x_ol = model.solve_ocp(x0,N,discount)
```

and for a closed loop simulation

```
res = model.mpc(x0,N,K,discount)
```

Here *x0* is a *nmpyc.nmpyc_array.array* which defines the initial value, *N* is the MPC horizon and the parameter *K* defines the number of MPC iterations. The parameter *discount* is optional and defines the discount factor (the default is *1*).

The result of the simulation can now be shown in the console by calling

```
print(res)
```

and as a visual output by calling

```
res.plot()
```

By default, the states and controls are displayed in two subplots. By passing a string as the first parameter (=*args*), the plot can be customized. For example, by calling

```
res.plot('state')
```

only the states are plotted. Other keywords are *control* for the control, *cost* for the stage costs, and *phase* to make a phase portrait of two states or controls. Furthermore, the plots displayed in this way can be additionally adjusted by further prameters, see *nmpyc.result.result.plot()*.

Further, the model and the simulation results can be saved for later use with the functions

```
model.save('path')
res.save('path')
```

These saved files can then be loaded with the help of

```
model = nmpyc.model.load('path')
res = nmpyc.result.load('path')
```

1.2.6 Advanced topics

The procedure described above is only an excerpt of the possibilities of the nMPyC Python library. For example, it is also possible to create autonomous systems and use the linear quadratic structure of a problem. For further informations see the *Examples* and *Templates* section. And for the implementation of linear system dynamics and quadratic costs, see also *nmpyc.system.system.LQP()* and *nmpyc.objective.objective.LQP()*.

1.3 Basics of model predictive control

Model predictive control (MPC) is an optimized-based method for obtaining an approximately optimal feedback control for an optimal control problem on an infinite or finite time horizon. The basic idea of MPC is to predict the future behavior of the controlled system over a finite time horizon and compute an optimal control input that, while ensuring satisfaction of given system constraints, minimizes the objective function. In each sampling instant a finite horizon open-loop optimal control problem is solved to calculate the control input. More precisley, this control input is used to define the feedback which is applied to the system until the next sampling instant, at whicht the horizon is shifted and the procedure is repeated again.

1.3.1 Optimal control problems

In order to describe the functionality of MPC we consider optimal control problems. To this end, we consider possibly nonlinear difference equations of the form

$$\begin{aligned} x(k+1, x_0) &= f(x(k, x_0), u(k)), \quad k = 0, \dots, N-1, \\ x(0) &= x_0 \end{aligned}$$

with $N \in \mathbb{N}$ or discretized differential equations.

Further, we impose nonempty state and input constraint sets $\mathbb{X} \subseteq \mathbb{R}^n$ and $\mathbb{U} \subseteq \mathbb{R}^m$, respectively, as well as a nonempty terminal constraint set $\mathbb{X}_0 \subseteq \mathbb{R}^n$.

Now we use optimal control to determine $u(0), \dots, u(N-1)$. For this reason, we fix a stage cost $\ell : \mathbb{X} \times \mathbb{U} \to \mathbb{R}$ which may be a very general function and a optional terminal cost $F : \mathbb{X} \to \mathbb{R}$. Regardless which cost function is used the objective function is defined by

$$J^N(x_0, u(\cdot)) := \sum_{k=0}^{N-1} \ell(x(k, x_0), u(k))$$

without terminal cost or by

$$J^N(x_0, u(\cdot)) := \sum_{k=0}^{N-1} \ell(x(k, x_0), u(k)) + F(x(N, x_0))$$

with terminal cost.

In summary, an optimal control problem without terminal conditions is given by

$$\begin{aligned} \min_{u(\cdot)\in\mathbb{U}} J^N(x_0, u(\cdot)) &= \sum_{k=0}^{N-1} \ell(x(k, x_0), u(k)) \\ \text{s.t.} \quad x(k+1, x_0) &= f(x(k, x_0), u(k)), \quad k = 0, \dots, N-1 \\ x(0) &= x_0 \\ x &\in \mathbb{X} \end{aligned} \tag{1.1}$$

and an optimal control problem with terminal conditions is given by

$$\begin{aligned} \min_{u(\cdot)\in\mathbb{U}} J^N(x_0, u(\cdot)) &= \sum_{k=0}^{N-1} \ell(x(k, x_0), u(k)) + F(x(N, x_0)) \\ \text{s.t.} \quad x(k+1, x_0) &= f(x(k, x_0), u(k)), \quad k = 0, \dots, N-1 \\ x(0) &= x_0 \\ x \in \mathbb{X}, &\quad x(N, x_0) \in \mathbb{X}_0 \end{aligned} \tag{1.2}$$

Additionally, with **nMPyC** it is possible to add constraints to the optimal control problem.

1.3.2 The basic MPC algorithm

Regardless of the type of the optimal control problem, the MPC algorithm is given by:

At each time instant $j = 0, 1, 2, \cdots$:

1. Measure the state $x(j) \in \mathbb{X}$ of the system.
2. Set $x_0 := x(j)$, solve the optimal control problem (with or without terminal conditions) and denote the obtained optimal control sequence by $u^\star(\cdot) \in \mathbb{U}^N(x_0)$.
3. Define the MPC-feedback value $\mu^N(x(j)) := u^\star(0) \in \mathbb{U}$ and use this control value in the next sampling period (apply the feedback to the system).

1.3.3 Notes and extensions

A special case of an optimal control problem is a linear-quadratic problem. There, the stage cost is defined as a quadratic function and the dynamics are linear. Thus, the linear-quadratic optimal control problem is given by

$$\begin{aligned} \min_{u(\cdot)\in\mathbb{U}} J^N(x_0, u(\cdot)) &= \sum_{k=0}^{N-1} \ell(x(k, x_0), u(k)) + F(x(N, x_0)) \\ &= \sum_{k=0}^{N-1} x(k, x_0)^T Q x(k, x_0) + u(k)^T R u(k) + 2x(k, x_0)^T N u(k) \\ &\quad + x(N, x_0)^T P x(N, x_0) \\ \text{s.t.} \quad x(k+1, x_0) &= Ax(k, x_0) + Bu(k), \quad k = 0, \dots, N-1 \\ x(0) &= x_0 \\ x \in \mathbb{X}, &\quad x(N, x_0) \in \mathbb{X}_0 \end{aligned} \tag{1.3}$$

where Q, R, N, P are weightening matrices and A, B the system matrices, each respectively of suitable dimension. Further, the constraints have to be also linear and of the form

$$Ex + Fu \geq h.$$

Note: **nMPyC** supports a time dependent formulation of optimal control problem. Hence, all functions, as f, ℓ, F, can depend on the time instance j.

Note: **nMPyC** supports also discounted optimal control problems. In the discrete case the objective is defined as

$$J^N(x_0, u(\cdot)) := \sum_{k=0}^{N-1} \beta^k \ell(x(k, x_0), u(k))$$

with $\beta \in (0, 1)$ the discount factor.

1.3.4 Further reading

For further reading and more theoretical insights we kindly refer to [GruneP17]

1.4 API Reference

system	A class used to define the system dynamics of an optimal control problem.

1.4.1 system

class system(*f*, *nx*, *nu*, *system_type='discrete'*, *sampling_rate=1.0*, *t0=0.0*, *method='cvodes'*)

A class used to define the system dynamics of an optimal control problem.

The dynamics can be discrete or continuous. A discrete system is defined by a difference equation

$$x(t_{k+1}) = f(t_k, x(t_k), u(t_k))$$

and a continous system is defined by the ordinary differential equation

$$\dot{x}(t_k) = f(t_k, x(t_k), u(t_k))).$$

In the letter case the differential equation will be discretized by a choosen integration method.

Parameters

- **f** (*callable*) – Function defining the right hand side of the system dynamics of the form $f(t, x, u)$ or $f(x, u)$ in the *autonomous* case. See also *f*.
- **nx** (*int*) – Dimension of the state. Must be a positive integer. See also *nx*.
- **nu** (*int*) – Dimension of the control. Must be a positive integer. See also *nu*.

- **system_type** (*str, optional*) – String defining if the given system dynamics are discrete or continuous. The default is 'discrete'.
- **sampling_rate** (*float, optional*) – Sampling rate defining at which time instances the dynamics are evaluated. The default is 1.
- **t0** (*float, optional*) – Initial time for the optimal control problem. The default is 0. See also *t0*.
- **method** (*str, optional*) – String defining which integration method should be used to discretize the system dynamics. The default is 'cvodes'. For further informations about the provided integrators see *method*.

Attributes

system.autonomous	If True, the system is time-invariant.
system.f	Right hand side $f(t, x, u)$ of the system dynamics.
system.h	Sampling time h of the system.
system.method	Integration method for discretization of the dynamics.
system.nu	Dimension of the control.
system.nx	Dimension of the state.
system.system_type	String defining whether the dynamics are discrete or continuous.
system.t0	Initial time of the optimal control problem.
system.type	Indicating whether the system dynamics are linear.

autonomous

Class property.

system.**autonomous**

If True, the system is time-invariant.

The right hand side of the dynamics $f(t, x, u)$ are not explicitly dependent on the time variable t. In this case $f(t, x, u) = f(x, u)$ holds.

Type
bool

f

Class property.

system.**f**

Right hand side $f(t, x, u)$ of the system dynamics.

The return value of this attribute depends on how the system is initialized. If it is initialized as a linear system by *LQP()* a list containing the arrays defining the system dynamics are returned. If the system is initalized by a possible nonlinear callable function this function is returned. Note, that even if *autonomous* is True the returned funtion depends on the time and always has the form $f(t, x, u)$.

Type
callable or list of *array*

h

Class property.

`system.h`

Sampling time h of the system.

This attribute defines at which time instances the dynamics are evaluated. This means the time t_k is given by the equation

$$t_k = t_0 + kh.$$

In addition, the control values are assumed to be constant during a sampling instance and can only be change at the times t_k.

Type
float

method

Class property.

`system.method`

Integration method for discretization of the dynamics.

The following integrators are currently supported:

- from CasADi: *cvodes*, *idas*, *collocation*, *oldcollocation* and *rk*
- from SciPy: *RK45*, *RK23*, *DOP853*, *Radau*, *BDF* and *LSODA*
- from nMPyC: *rk4*, *euler* und *heun* (fixed step integration methods)

Type
str

nu

Class property.

`system.nu`

Dimension of the control.

The value of $u(t)$ at a given time t_k is a element of $\mathbb{R}^{nu}$. In the linear case this value equals with the dimension of the columns of the control matrix $B \in \mathbb{R}^{nx \times nu}$.

Type
int

nx

Class property.

`system.`**`nx`**

Dimension of the state.

The value of $x(t)$ at a given time t_k is a element of $\mathbb{R}^{nx}$. In the linear case this value equals with the dimension of the system matrix $A \in \mathbb{R}^{nx \times nx}$.

Type
int

system_type

Class property.

`system.`**`system_type`**

String defining whether the dynamics are discrete or continuous.

A discrete system is defined by a difference equation

$$x(t_{k+1}) = f(t_k, x(t_k), u(t_k))$$

and a continous system is defined by the ordinary differential equation

$$\dot{x}(t_k) = f(t_k, x(t_k), u(t_k))).$$

Type
str

t0

Class property.

`system.`**`t0`**

Initial time of the optimal control problem.

The initial state x_0 is measured at time t_0. The state $x(t)$ is evaluated at the time instances $t_0 + kh$ during the MPC loop where h is the `sampling_rate`.

Type
float

type

Class property.

`system.`**`type`**

Indicating whether the system dynamics are linear.

If *LQP*, the system dynamics are linear. The right hand side of the system dynamics is given by

$$f(x, u) = Ax + Bu.$$

It also implies that the system is *autonomous*.

If the system dynamics are not initialized as linear with the *LQP()* method this attribute has the value *NLP*.

Type
str

Methods

system.LQP	Initialize the system with linear dynamics.
system.load	Loads a nMPyC system object from a file.
system.save	Saving the system to a given file with dill.
system.set_integratorOptions	Set options for the integration method.
system.system	Evaluate right hand side $f(t, x, u)$ of the dynamics.
system.system_discrete	Evaluate discretized right hand side of the system dynamics.

LQP

Class method.

LQP(*A*, *B*, *nx*, *nu*, *system_type='discrete'*, *sampling_rate=1.0*, *t0=0.0*, *method='euler'*)

Initialize the system with linear dynamics.

In this case the right hand side of the dynamics has the form :

$$f(x, u) = Ax + Bu$$

which is always *autonomous*. If not a fixed step method is choosen for integration the optimizer can not use the linear structure of the problem during the optimization process.

Parameters

- **A** (*array*) – Matrix definig the linear state input on the right hand side of the dynamics.
- **B** (*array*) – Matrix definig the linear state input on the right hand side of the dynamics.
- **nx** (*int*) – Dimension of the state. Must be a positive integer. See also *nx*.
- **nu** (*int*) – Dimension of the control. Must be a positive integer. See also *nu*.
- **system_type** (*str, optional*) – String defining whether the given system dynamics are discrete or continuous. The default is 'discrete'.
- **sampling_rate** (*float, optional*) – Sampling rate defining at which time instances the dynamics are evaluated. The default is 1.
- **t0** (*float, optional*) – Initial time for the optimal control problem. The default is 0. See also *t0*.
- **method** (*str, optional*) – String defining which integration method should be used to discretize the system dynamics. The default is 'euler'. For further informations about the provided integrators see *method*.

Returns
lqp – nMPyC-system class object suitable to define a linear quadratic optimal control problem..

Return type
system

load

Class method.

load(*path*)

Loads a nMPyC system object from a file.

The specified path must lead to a file that was previously saved with *save()*.

Parameters
path (*str*) – String defining the path to the file containing the nMPyC system object.

For example

```
>>> system.load('system.pickle')
```

will load the system previously saved with *save()*.

save

Class method.

save(*self*, *path*)

Saving the system to a given file with dill.

The path can be absolut or relative and the ending of the file is arbitrary.

Parameters
path (*str*) – String defining the path to the desired file.

For example

```
>>> system.save('system.pickle')
```

will create a file *system.pickle* containing the nMPyC system object.

set_integratorOptions

Class method.

set_integratorOptions(*self*, *options*)

Set options for the integration method.

Parameters
options (*dict*) – Dictionary containing the keywords of the required options and their values.

The available options are depending on the choosen *method* of integration. For the nMPyC integrators the only available option is *number_of_finit_elements* which must be an int greater than zero and defines how many discretation steps are computed during one sampling period defined by the sampling rate. The available options for the CasADi integrators can be found at Sourceforge and for the SciPy integrators at the Scipy documentation.

system

Class method.

system(*self*, *t*, *x*, *u*)

Evaluate right hand side $f(t, x, u)$ of the dynamics.

Parameters

- **t** (*float*) – Time instant at which the system dynamics are evaluated.
- **x** (*array*) – State value at which the system dynamics are evaluated.
- **u** (*array*) – Control value at which the system dynamics are evaluated.

Returns

Value of the possible not discrete right hand side of the dynamics evaluated at the given inputs.

Return type

array

system_discrete

Class method.

system_discrete(*self*, *t*, *x*, *u*)

Evaluate discretized right hand side of the system dynamics.

Parameters

- **t** (*float*) – Time instant at which the system dynamics are evaluated.
- **x** (*array*) – State value at which the system dynamics are evaluated.
- **u** (*array*) – Control value at which the system dynamics are evaluated.

Returns

Value of the discretized right hand side of the dynamics evaluated at the given inputs.

Return type

array

objective	A class used to define the objective of the optimal control problem.

1.4.2 objective

class objective(*stagecost*, *terminalcost=None*)

A class used to define the objective of the optimal control problem.

The objective depends on the stage cost and optional on terminal cost and has the form

$$J(t, x, u, N) := \sum_{k=0}^{N-1} \ell(t_k, x(t_k), u(t_k)) + F(t_N, x(t_N)).$$

The values of the times t_k are defined by initializing the *nmpyc.system.system*. For the slightly different form of the objective in the discounted case see *discount*.

Parameters

- **stagecost** (*callable*) – A function defining the stage cost of the optimal control problem. Has to be of the form $\ell(t, x, u)$ or $\ell(x, u)$ in the *autonomous* case. See also *stagecost*.
- **terminalcost** (*callable, optional*) – A function defining the terminal cost of the optimal control problem. Has to be of the form $F(t, x)$ or $F(x)$ in the autonomous case. If None, no terminal cost is added. The default is None. See also *terminalcost*.

Attributes

objective.autonomous	If True, the objective is autonomous.
objective.discount	The discount factor of the objective.
objective.stagecost	Stage cost $\ell(t, x, u)$.
objective.terminalcost	Terminal cost $F(t, x)$.
objective.type	Indicating whether the objective is quadratic or nonlinear.

autonomous

Class property.

objective.**autonomous**

If True, the objective is autonomous.

The stage cost and terminal cost of the objective $J(t, x, u, N)$ are not explicitly dependend on the time variable t. In this case $J(t, x, u, N) = J(x, u, N)$ holds.

Type
bool

discount

Class property.

objective.**discount**

The discount factor of the objective.

For a discount factor $\delta \in (0, 1]$ the discounted objective function is given by

$$J(t, x, u, N) = \sum_{k=0}^{N-1} \delta^k \ell(t_k, x(t_k), u(t_k)) + F(t_N, x(t_N)).$$

By default the discount factor is equal to 1 $\delta = 1$. Then, we name the problem undiscounted. The discount factor for the OCP of the MPC simulation can be set when the *nmpyc.model.model.mpc()* method is called.

Type
float

stagecost

Class property.

`objective.stagecost`

Stage cost $\ell(t, x, u)$.

The return value of this attribute depends on how the objective is initialized. If it is initialized as a quadratic objective by `LQP()` a list containing the arrays defining the stage cost are returned. If the obejctive is initalized by possibly nonlinear callable functions the function defining the stage cost is returned. Note, that even if `autonomous` is True the returned function depends on the time t and always has the form $\ell(t, x, u)$.

Type
callable or list of *array*

terminalcost

Class property.

`objective.terminalcost`

Terminal cost $F(t, x)$.

The return value of this attribute depends on how the objective is initialized. If it is initialized as a quadratic objective by `LQP()` the array defining the terminal cost is returned. If the obejctive is initalized by possibly nonlinear callable functions the function defining the terminal cost is returned. Note, that even if `autonomous` is True the returned function depends on the time t and always has the form $\ell(t, x, u)$.

Type
callable or *array*

type

Class property.

`objective.type`

Indicating whether the objective is quadratic or nonlinear.

If *LQP*, the objective is quadratic. Then the stage cost is given by

$$\ell(x, u) = x^T Q x + u^T R u + 2x^T N x.$$

and the terminal cost is given by

$$F(x) = x^T P x.$$

It also implies that the system is `autonomous`.

If the objective is not initialized as quadratic function with the `LQP()` method this attribute holds the value *NLP*.

Type
str

Methods

objective.J	Evaluate objective function of the OCP.
objective.LQP	Initialize a quadratic objective.
objective.add_termianlcost	Add terminal cost to the objective.
objective.endcosts	Evaluate termninal cost of the objective.
objective.load	Loads a nMPyC objective object from a file.
objective.save	Saving the objective to a given file with dill.
objective.stagecosts	Evaluate stage cost of the objective.

J

Class method.

J(*self*, *t*, *x*, *u*, *N*)

Evaluate objective function of the OCP.

The objective function is assembled from the stage cost $\ell(t, x, u)$ and optional terminal cost $F(t, x)$ and has the form

$$J(t, x, u, N) = \sum_{k=0}^{N-1} \delta^k \ell(t_k, x(t_k), u(t_k)) + F(t_N, x(t_N)).$$

Where $\delta \in (0, 1]$ is a possible discount factor, see *discount*.

Parameters

- **t** (*array*) – Times instant at which the stage costs and terminal cost are evaluated.
- **x** (*array*) – State trajectory at which the stage cost and terminal cost are evaluated.
- **u** (*array*) – Control sequence at which the stage cost is evaluated.
- **N** (*int*) – Maximum index up to which the stage cost are summed. During the MPC iteration this index is equivalent to the MPC horizon.

Returns

J – Value of the objective function at the given input parameters.

Return type

array

LQP

Class method.

LQP(*Q*, *R*, *N=None*, *P=None*)

Initialize a quadratic objective.

In this case the stage cost of the objective has the form

$$\ell(x, u) = x^T Q x + u^T R u + 2 x^T Q u$$

and the optional terminal cost is defined as

$$F(x, u) = x^T P x.$$

In this case the objective is always *autonomous*.

Parameters

- **Q** (*array*) – Matrix defining the cost of the state of the form $x^T Q x$.
- **R** (*array*) – Matrix defining the cost of the control of the form $u^T R u$.
- **N** (*array, optional*) – Possible Matrix defining the mixed cost term of the form $2x^T N u$. The default is None.
- **P** (*array, optional*) – Posible Matrix defining the terminal cost of the form $x^T P x$. The default is None.

Returns

QP – nMPyC-objective class object suitable to define a linear quadratic optimal control problem.

Return type

objective

add_termianlcost

Class method.

add_termianlcost(*self*, *terminalcost*)

Add terminal cost to the objective.

The terminal cost must be a callable function of the form $F(t, x)$ or $F(x)$ in the autonomous case. If terminal cost already exists they will be over written.

Parameters

terminalcost (*callable*) – A function defining the terminal cost of the optimal control problem. Has to be of the form $F(t, x)$ or $F(x)$ in the autonomous case.

endcosts

Class method.

endcosts(*self*, *t*, *x*)

Evaluate termninal cost of the objective.

Parameters

- **t** (*float*) – Time instant at which the terminal cost is evaluated.
- **x** (*array*) – Current state at which the terminal cost is evaluated.

Returns

Terminal cost evaluated at the given input values.

Return type

array

load

Class method.

load(*path*)

Loads a nMPyC objective object from a file.

The specified path must lead to a file that was previously saved with *save()*.

Parameters
path (*str*) – String defining the path to the file containing the nMPyC objective object.

For example

```
>>> objective.load('objective.pickle')
```

will load the objective previously saved with *save()*.

save

Class method.

save(*self*, *path*)

Saving the objective to a given file with dill.

The path can be absolut or relative and the ending of the file is arbitrary.

Parameters
path (*str*) – String defining the path to the desired file.

For example

```
>>> objective.save('objective.pickle')
```

will create a file *objective.pickle* containing the nMPyC objective object.

stagecosts

Class method.

stagecosts(*self*, *t*, *x*, *u*)

Evaluate stage cost of the objective.

Parameters

- **t** (*float*) – Time instant at which the stage cost is evaluated.
- **x** (*array*) – Current state at which the stage cost is evaluated.
- **u** (*array*) – Current control at which the stage cost is evaluated.

Returns
Stage cost evaluated at the given input values.

Return type
array

constraints	Class used to define the constraints of the optimnal control problem.

1.4.3 constraints

class constraints

Class used to define the constraints of the optimnal control problem.

Support for nonlinear, linear and box constraints are implemented and provided.

To define the constraints, first, an empty object have to be created. Then the individual constraints can be added with the help of the methods *add_bound()* and *add_constr()*.

Attributes

constraints.linear_constr	Collection of all linear constraints.
constraints.lower_bndend	Lower bound $l_x \in \mathbb{R}^{nx}$ of the terminal state.
constraints.lower_bndu	Lower bound $l_u \in \mathbb{R}^{nu}$ for the control.
constraints.lower_bndx	Lower bound $l_x \in \mathbb{R}^{nx}$ of the state.
constraints.nonlinear_constr	Collection of all nonlinear constraints.
constraints.type	Indicating whether all constraints are linear.
constraints.upper_bndend	Upper bound $u_x \in \mathbb{R}^{nx}$ of the terminal state.
constraints.upper_bndu	Upper bound $u_u \in \mathbb{R}^{nu}$ of the control.
constraints.upper_bndx	Upper bound $u_x \in \mathbb{R}^{nx}$ of the state.

linear_constr

Class property.

constraints.**linear_constr**

Collection of all linear constraints.

This dictionary has the following form:

```
>>> linear_constr = {'eq': [..], 'ineq': [..],
>>>                  'terminal_eq': [..], 'terminal_ineq': [..]}
```

The arrays that define the constraints are saved as lists which are contained in the dictionary. For example

```
>>> linear_constr['eq'][0]
```

returns a list with the arrays H, F and h defining the first equality constraint

$$Hx + Fu = h.$$

Type
dict

lower_bndend

Class property.

`constraints.lower_bndend`

Lower bound $l_x \in \mathbb{R}^{nx}$ of the terminal state.

For the terminal state $x(t_N)$ the inequality

$$x_i(t_N) \geq l_{x_i} \quad \text{for } i = 1, \ldots, nx$$

holds as a constraint.

Type
array

lower_bndu

Class property.

`constraints.lower_bndu`

Lower bound $l_u \in \mathbb{R}^{nu}$ for the control.

For all controls $u(t_k)$ the inequalities

$$u_i(t_k) \geq l_{u_i} \quad \text{for } i = 1, \ldots, nu$$

hold as a constraint.

Type
array

lower_bndx

Class property.

`constraints.lower_bndx`

Lower bound $l_x \in \mathbb{R}^{nx}$ of the state.

For all states $x(t_k)$ the inequalities

$$x_i(t_k) \geq l_{x_i} \quad \text{for } i = 1, \ldots, nx$$

hold as a constraint.

Type
array

nonlinear_constr

Class property.

`constraints.nonlinear_constr`

Collection of all nonlinear constraints.

This dictionary has the following form:

```
>>> nonlinear_constr = {'eq': [..], 'ineq': [..],
>>>                     'terminal_eq': [..], 'terminal_ineq': [..]}
```

In the lists contained in the dictionary the functions defining the constraints are saved. For example

```
>>> nonlinear_constr['eq'][0]
```

returns the function $h(t, x, u)$ defining the first equality constraint

$$h(t, x, u) = 0.$$

Type
dict

type

Class property.

`constraints.`**type**

Indicating whether all constraints are linear.

If *LQP*, all constraints are linear. Then all constraints are of the form

$$Ex + Fu \leq h$$

If at least one constraint is initialized as a nonlinear constraint this attribute has the value *NLP*.

Type
str

upper_bndend

Class property.

`constraints.`**upper_bndend**

Upper bound $u_x \in \mathbb{R}^{nx}$ of the terminal state.

For the terminal state $x(t_N)$ the inequalities

$$x_i(t_N) \leq u_{x_i} \quad \text{for } i = 1, \ldots, nx$$

hold as a constraint.

Type
array

upper_bndu

Class property.

`constraints.`**upper_bndu**

Upper bound $u_u \in \mathbb{R}^{nu}$ of the control.

For all controls $u(t_k)$ the inequalities

$$u_i(t_k) \leq u_{u_i} \quad \text{for } i = 1, \ldots, nu$$

hold as a constraint.

Type
array

upper_bndx

Class property.

`constraints.`**upper_bndx**

Upper bound $u_x \in \mathbb{R}^{nx}$ of the state.

For all states $x(t_k)$ the inequalities

$$x_i(t_k) \leq u_{x_i} \quad \text{for } i = 1, \ldots, nx$$

hold as a constraint.

Type
array

Methods

constraints.add_bound	Add bounds as linear constraints to the OCP.
constraints.add_constr	Add linear or nonlinear constraints to the OCP.
constraints.add_terminalconstr	Add linear or nonlinear terminal constraints to the OCP.
constraints.load	Loads a nMPyC constraints object from a file.
constraints.save	Saving the constraints to a given file with dill.

add_bound

Class method.

add_bound(*self*, *bnd_type*, *variable*, *bound*)

Add bounds as linear constraints to the OCP.

Note while adding the bound it is not checked if the bounds have the correct shape. This will be verified later during the optimization progress.

Parameters

- **bnd_type** (*str*) – String defining whether the bound is a lower or upper bound.
- **variable** (*str*) – String defining on which variable the bound should be applied. Possible values are *state*, *control* and *terminal*.
- **bound** (*array*) – Array containing the values of the bound.

For example

```
>>> constraints.add_bound('lower', 'state', lbx)
```

will add *lbx* as *lower_bndx* while

```
>>> constraints.add_bound('upper', 'terminal', ub_end)
```

will add *ub_end* as *upper_bndend*.

add_constr

Class method.

add_constr(*self*, *cons_type*, **args*)

Add linear or nonlinear constraints to the OCP.

Nonlinear inequality constraints are of the form

$$g(t, x, u) \geq 0 \quad \text{or} \quad g(x, u) \geq 0.$$

Nonlinear equality constraints are of the form

$$h(t, x, u) = 0 \quad \text{or} \quad h(x, u) = 0.$$

Linear inequality constraints are of the form

$$Ex + Fu \geq h.$$

Linear equality constraints are of the form

$$Ex + Fu = h.$$

For the form of terminal constrains see *add_terminalconstr()*.

Parameters

- **cons_type** (*str*) – String that defines the type of the constraints. Possible values are *eq*, *ineq*, *terminal_eq* and *terminal_ineq*.
- ***args** (*callable or arrays*) – Function defining the (nonlinear) constraints or arrays defining the linear constraints. In the letter case the order of arguments are E, F, h and if h is undefined this array is set to zero.

For example

```
>>> constraints.add_constr('ineq', E, F, h)
```

will add a linear inequality constraint to *linear_constr* while

```
>>> constraints.add_constr('terminal_eq',h_end)
```

will add a nonlinear equality terminal constraint to *nonlinear_constr*.

add_terminalconstr

Class method.

add_terminalconstr(*self*, *cons_type*, **args*)

Add linear or nonlinear terminal constraints to the OCP.

Nonlinear terminal inequality constraints are of the form

$$g(t, x) \geq 0 \quad \text{or} \quad g(x) \geq 0.$$

Nonlinear terminal equality constraints are of the form

$$h(t, x) = 0 \quad \text{or} \quad h(x) = 0.$$

Linear terminal inequality constraints are of the form

$$Ex \geq h.$$

Linear terminal equality constraints are of the form

$$Ex = h.$$

Parameters

- **cons_type** (*str*) – String that defines the type of the terminal constraints. Possible values are *eq* or *ineq*.
- ***args** (*callable or arrays*) – Function defining the (nonlinear) terminal constraints or arrays defining the linear constraints. In the letter case the order of arguments are E, h and if h is undefined this array is set to zero.

For example

```
>>> constraints.add_terminalconstr('ineq', E, F, h)
```

will add a linear inequality terminal constraint to *linear_constr* while

```
>>> constraints.add_constr('eq',h)
```

will add a nonlinear equality terminal constraint to *nonlinear_constr*.

load

Class method.

load(*path*)

Loads a nMPyC constraints object from a file.

The specified path must lead to a file that was previously saved with *save()*.

Parameters

path (*str*) – String defining the path to the file containing the nMPyC constraints object.

For example

```
>>> constraints.load('constraints.pickle')
```

will load the constraints previously saved with *save()*.

save

Class method.

save(*self*, *path*)

Saving the constraints to a given file with dill.

The path can be absolut or relative and the ending of the file is arbitrary.

Parameters

path (*str*) – String defining the path to the desired file.

For example

```
>>> constraints.save('constraints.pickle')
```

will create a file *constraints.pickle* containing the nMPyC constraints object.

model	Class that contains all the components of the optimal control problem.

1.4.4 model

class model(*objective*, *system*, *constraints=None*)

Class that contains all the components of the optimal control problem.

Can be used to perform open and closed loop simulations.

Parameters

- **objective** (*objective*) – nMPyC-objective defining the objective of the optimal control problem.
- **system** (*system*) – nMPyC-system defining the system dynamics of the optimal control problem.
- **constraints** (*constraints optional*) – nMPyC-constraints defining the constraints of the optimal control problem. If constraints is None the problem is unconstrained. The default is None.

Attributes

model.N	Prediction horizon of the MPC loop.
model.constraints	Constraints of the optimal control problem.
model.objective	Objective of the optimal control problem.
model.opti	Optimizer for the optimal control problem.
model.system	System dynamics of the optimal control problem.

N

Class property.

model.**N**

Prediction horizon of the MPC loop.

Type
int

constraints

Class property.

`model.constraints`

Constraints of the optimal control problem.

Type
constraints

objective

Class property.

`model.objective`

Objective of the optimal control problem.

Type
objective

opti

Class property.

`model.opti`

Optimizer for the optimal control problem.

This property can be used to set different optimization options. A distinction is made between basic settings of the optimizer and solver-specific settings.

The basic settings can be adjusted by calling

```
>>> model.opti.set_options({..})
```

The dictionary that is passed can contain the following entries

Parameter	Description	Default value
solver	String defining which solver is for optimization. Currently supported solvers are • ipotpt • sqpmethod • ospq • SLSQP • trust-constr For auto a suitable solver depending on other options and parameters is selected.	auto
full_discretization	If True, the method of full discretiziation is used for optimization. Otherwise the the system dynamics is resolved in the objective function.	True
tol	The toleranz of the solver. If the solver distinguishes between relative and absolute tolerances, both are set to this value.	1e-06
maxiter	Maximal number of iterations during the optimization progress.	5000
verbose	If True, the verbose option of the selected solver ist activated.	False
initial_guess	Initial guess for the optimization variable u. Must be an array of shape (nx,N). If the initial guess has not the right shape or is None it will be set to nmpyc.ones((nu,N))*0.1 by default.	None

The auto option of the solver selection follows the rule

1. If the optimal control problem is recognized as a LQP and a fixed step discretization of the system is given, osqp is selected.
2. If a condition of 1. is violated and not a SciPy discretization method is choosen, ipopt is selected.
3. Otherwise SLSQP is selected.

The solver-specific settings can be custamized by calling

```
>>> model.opt.set_solverOptions({..})
```

Valid parameters which the passed dictionary can contain are depending on the selected solver. For a list of these settings take a look at

- Sourceforge for the CasADi solvers
- SciPy Documentation for the SciPy solvers
- OSQP Website for the osqp solver

Type
opti

system

Class property.

`model.system`

System dynamics of the optimal control problem.

Type
system

Methods

model.load	Loads a nMPyC model object from a file.
model.mpc	Solves the optimal control problem via model predictive control.
model.save	Saving the model to a given file with dill.
model.solve_ocp	Solves the finit horizon optimal control problem.

load

Class method.

load(*path*)

Loads a nMPyC model object from a file.

The specified path must lead to a file that was previously saved with *save()*.

Parameters
path (*str*) – String defining the path to the file containing the nMPyC model object.

For example

```
>>> model.load('model.pickle')
```

will load the model previously saved with *save()*.

mpc

Class method.

mpc(*self*, *x0*, *N*, *K*, *discount=None*)

Solves the optimal control problem via model predictive control.

Parameters

- **x0** (*array*) – Initial state of the optimal control problem.
- **N** (*int*) – MPC horizon.
- **K** (*int*) – Number of MPC itertaions.
- **discount** (*float, optional*) – Discountfactor of the objective. The default is None.

Returns

res – nMPyC result object containing the optimiaztion results of the closed and open loop simulations.

Return type

result

save

Class method.

save(*self*, *path*)

Saving the model to a given file with dill.

The path can be absolut or relative and the ending of the file is arbitrary.

Parameters

path (*str*) – String defining the path to the desired file.

For example

```
>>> model.save('objective.pickle')
```

will create a file *model.pickle* containing the nMPyC model object.

solve_ocp

Class method.

solve_ocp(*self*, *x0*, *N*, *discount=None*)

Solves the finit horizon optimal control problem.

Parameters

- **x0** (*array*) – Initial value of the optimal control problem.
- **N** (*int*) – Prediction horizon of the control problem.
- **discount** (*float, optional*) – Discountfactor of the objective. The default is None.

Returns

- **u_ol** (*array*) – Optimal control sequence.
- **x_ol** (*array*) – Optimal trajectory corresponding to the optimal control sequence.

result	Class used to store the simulation results of the MPC simulation.

1.4.5 result

class result(*x0*, *t0*, *h*, *N*, *K*)

Class used to store the simulation results of the MPC simulation.

To obtain the individual components of the simulation, such as closed loop and open loop results, the individual attributes need to be called.

Also the result object contains information about errors and other solver statistics, which can be used for further investigation of the simulation progress.

Additionally, this class provides a way to visualize the results in a suitable way with the *plot()* method.

Parameters

- **x0** (array) – Initial state.
- **t0** (*float*) – Initial time.
- **h** (*float*) – Sampling rate.
- **N** (*int*) – MPC horizon.
- **K** (*int*) – Number of MPC Interations.

Attributes

result.N	MPC horizon
result.ellapsed_time	Total ellapsed time for the closed loop simulation.
result.ellapsed_time_per_itertaion	List containing the ellapsed time of every single itertaion of the closed loop simulation.
result.error	Error message with which the solver failed.
result.l_cl	Stage costs evaluated at the closed loop trajectory and feedback.
result.l_ol	List containing the stage costs evaluated at the open loop trajectories and controls of all MPC itertaions.
result.sampling_rate	Sampling rate.
result.solver	Name of the choosen optimization method.
result.succes	True if the solver converged sucessfully in all MPC itertaions, false if the MPC loop abort prematurely.
result.sucessfull_itertaions	Number of succesfull MPC iterations.
result.t0	Initial time
result.t_cl	Time sequence at which the closed loop states and controls are evaluated.
result.t_ol	List containing the time sequences at which the closed loop states and controls are evaluated in the open loop simulations.
result.u_cl	Closed loop feedback.
result.u_ol	List containing the open loop optimal control values of all MPC itertaions.
result.x0	Initial state.
result.x_cl	Closed loop trajectory.
result.x_ol	List containing the open loop trajectories of all MPC itertaions.

N

Class property.

`result.N`

MPC horizon

Type
int

ellapsed_time

Class property.

`result.ellapsed_time`

Total ellapsed time for the closed loop simulation.

Type
float

ellapsed_time_per_itertaion

Class property.

`result.ellapsed_time_per_itertaion`

List containing the ellapsed time of every single itertaion of the closed loop simulation.

Type
list of float

error

Class property.

`result.error`

Error message with which the solver failed. If success is True error is None.

Type
str

l_cl

Class property.

`result.l_cl`

Stage costs evaluated at the closed loop trajectory and feedback.

Type
numpy.ndarray

l_ol

Class property.

`result.l_ol`

List containing the stage costs evaluated at the open loop trajectories and controls of all MPC itertaions.

Type
list of numpy.ndarrays

sampling_rate

Class property.

`result.sampling_rate`

Sampling rate.

Type
float

solver

Class property.

`result.solver`

Name of the choosen optimization method.

Type
str

succes

Class property.

`result.succes`

True if the solver converged sucessfully in all MPC itertaions, false if the MPC loop abort prematurely.

Type
bool

sucessfull_itertaions

Class property.

`result.sucessfull_itertaions`

Number of succesfull MPC iterations.

Type
int

t0

Class property.

`result.t0`

Initial time

Type
float

t_cl

Class property.

`result.t_cl`

Time sequence at which the closed loop states and controls are evaluated.

Type
numpy.ndarray

t_ol

Class property.

`result.t_ol`

List containing the time sequences at which the closed loop states and controls are evaluated in the open loop simulations.

Type
list of numpy.ndarrays

u_cl

Class property.

`result.u_cl`

Closed loop feedback.

Type
numpy.ndarray

u_ol

Class property.

`result.u_ol`

List containing the open loop optimal control values of all MPC itertaions.

Type
list of numpy.ndarrays

x0

Class property.

`result.x0`

Initial state.

Type
numpy.ndarray

x_cl

Class property.

`result.x_cl`

Closed loop trajectory.

Type
numpy.ndarray

x_ol

Class property.

`result.x_ol`

List containing the open loop trajectories of all MPC itertaions.

Type
list of numpy.ndarrays

Methods

result.load	Loads a nMPyC result object from a file.
result.plot	Plot the results of the MPC simulation.
result.save	Saving the result to a given file with dill.
result.show_errors	Shows the errors that occurred during the simualtion.

load

Class method.

`load`(*path*)

Loads a nMPyC result object from a file.

The specified path must lead to a file that was previously saved with *save()*.

Parameters
path (*str*) – String defining the path to the file containing the nMPyC result object.

For example

```
>>> result.load('result.pickle')
```

will load the result previously saved with *save()*.

plot

Class method.

`plot`(*self*, **args*, ***kwargs*)

Plot the results of the MPC simulation.

If no argument is passed, by default the closed loop states and controls are plotted seperated in two subplots.

If only a specific component of the solution should be plotted, this can be customized by using a string as the first argument. Valid arguments for this are

- *state* for only plotting the closed loop state trajectories
- *control* for only plotting the closed loop control values
- *cost* for plotting the stage costs evaluated at the closed loop results
- *phase* for plotting a phase portrait of two components of the solution.

Further adjustments can be made with the help of the following keyword arguments.

Argument	Description	Default value
xk	List specifying which components of the state are plotted.	*[1,...,nx]*
uk	List specifying which components of the control are plotted.	*[1,...,nu]*
show_ol	If True, the open loop simulation rersults are also plotted additionaly to the closed loop results.	True
iters	List indicating from which iteration on the open loop results should be plotted. Will be ignored if show_ol is False.	*[1,...,K+1]*
usetex	If True, the captions are displayed in TEX style.	True
grid	If True, a grid is displayed in the background of the plot.	True
show_legend	If True, a legend will be displayed inside the plot.	True
phase1	Phase 1 of the phase portrait plot. Has the form *x_k* or *u_k*, where *k* determines the respective component. Will be ignored if args!='phase'.	None
phase2	Phase 2 of the phase portrait plot. Has the form *x_k* or *u_k*, where *k* determines the respective component. Will be ignored if args!='phase'.	None
dpi	resolution of the figure, see	100
figsize	The size of the figure. Has the form [width, height] in inches.	[8., 6.]
linewidth	Set the line width in points.	2.
fontsize	The font size of the annotations. If the value is numeric the size will be the absolute font size in points.	14.

save

Class method.

save(*self*, *path*)

Saving the result to a given file with dill.

The path can be absolut or relative and the ending of the file is arbitrary.

Parameters
path (*str*) – String defining the path to the desired file.

For example

```
>>> result.save('result.pickle')
```

will create a file *result.pickle* containing the nMPyC result object.

show_errors

Class method.

show_errors(*self*)

Shows the errors that occurred during the simualtion.

For example, if the solver ipopt was selected and the defined optimal control problem is infeasible, this method will print out the message

An error occured during itertaion 1 of 100:
Error in Opti::solve [OptiNode] at …/casadi/core/optistack.cpp:159:
…/casadi/core/optistack_internal.cpp:999: Assertion "return_success(accept_limit)" failed:
Solver failed. You may use opti.debug.value to investigate the latest values of variables. return_status is 'Infeasible_Problem_Detected'

For more informations about the error messages take a look at the documentation of the respective solver.

If the simulation was completly succesfull, the message

No error occured during the MPC-Loop

will be printed out.

nmpyc_array	Module for array definition and computation.

1.4.6 nmpyc_array

Module for array definition and computation.

This module provides an array class and associated functions for corresponding matrix calculations.

The goal of this class and the individual functions is to enable compatibility of calculations with both casadi and numpy objects without changing the syntax of the program. This enables the user to program as easily as possible and at the same time to switch between symbolic and numeric calculation.

Classes

array	Class used to save arrays with symbolic or numeric values.

array

class array(*dim=0*)

Class used to save arrays with symbolic or numeric values.

The symbolic entries are provided by CasADi and will be transformed automatically to numeric values of numpy type if it is posiible.

Parameters

dim (*int, tuple, cas.MX, cas.SX, cas.DM, list or numpy.ndarray, optional*) – Dimension of which an empty array is created or object from which the entries and dimension are copied. The default is 0.

Attributes

array.A	Array containing all entries.
array.T	Transposed array.
array.dim	Dimension of the array.
array.symbolic	True if array has symbolic entries, False otherwise.

A

Class property.

array.A

Array containing all entries.

Type

casadi.MX or numpy.array

T

Class property.

`array.T`

Transposed array.

Type
array

dim

Class property.

`array.dim`

Dimension of the array.

Type
tuple

symbolic

Class property.

`array.symbolic`

True if array has symbolic entries, False otherwise.

Type
bool

Methods

array.fill	Fill all entries with one value.
array.flatten	Flat array to one dimension.
array.transpose	Transpose array.

fill

Class method.

`fill`(*self*, *a*)

Fill all entries with one value.

Parameters
a (*int or float*) – Value that all entries should take.

flatten

Class method.

`flatten`(*self*)

Flat array to one dimension.

Returns
y – Flatten array with dimension (1,n).

Return type
array

transpose

Class method.

`transpose`(*self*)

Transpose array.

Returns
y – Transposed array.

Return type
array

Functions

abs	Calculates the absolute value of a number or array.
arccos	Calculates the arcuscosinus of a given number or array
arccosh	Calculates the arcuscosinus hypernolicus of a given number or array
arcsin	Calculates the arcussinus of a given number or array
arcsinh	Calculates the arcussinus hyperbolicus of a given number or array
arctan	Calculates the arcustangens of a given number or array
arctanh	Calculates the arcustangens hyperbolicus of a given number or array
concatenate	Join a sequence of arrays along an existing axis.
convert	Convert a numpy-, casadi- or nMPyC-array to another of these intances.
cos	Calculates the cosinus of a given number or array
cosh	Calculates the cosinus hyperbolicus of a given number or array
diag	Creates an diagonal matrix from a given vector.
exp	Calculates the exponential of a given number or array
eye	Creates an array defining the idendity.
log	Calculates the natural logarithm of a given number or array
matrix_power	Raises a square matrix to the n-th power.
max	Return the maximal value of the arguments
min	Returns the minimal value of the arguments
norm	Returns the norm of a vector or matrix.
ones	Creates an array with only entries equal to one.
power	Calculates the (elementwise) n-th power of a number or array.
reshape	Reshape an array to a new size.
sin	Calculates the sinus of a given number or array
sinh	Calculates the sinus hyperbolicus of a given number or array
sqrt	Calculates the square root of a given number or array
tan	Calculates the tangens of a given number or array
tanh	Calculates the tangens hyperblicus of a given number or array
zeros	Creates an array with only zero entries.

abs

abs(*x*)

Calculates the absolute value of a number or array.

arccos

arccos(*x*)

Calculates the arcuscosinus of a given number or array

arccosh

arccosh(*x*)

Calculates the arcuscosinus hypernolicus of a given number or array

arcsin

arcsin(*x*)

Calculates the arcussinus of a given number or array

arcsinh

arcsinh(*x*)

Calculates the arcussinus hyperbolicus of a given number or array

arctan

arctan(*x*)

Calculates the arcustangens of a given number or array

arctanh

arctanh(*x*)

Calculates the arcustangens hyperbolicus of a given number or array

concatenate

concatenate(*arrays*, *axis=0*)

Join a sequence of arrays along an existing axis.

Parameters

- **arrays** (*tuple of casadi.MX, casadi.SX, casadi.DM or numpy.ndarrays*) – Sequence of arrays which will be concatenated. The arrays must have the same shape, except in the dimension corresponding to axis.
- **axis** (*int, optional*) – The axis along which the arrays will be joined. The default is 0.

Returns
The concatenated array.

Return type
array

convert

convert(*a*, *dtype='auto'*)

Convert a numpy-, casadi- or nMPyC-array to another of these intances.

Parameters

- **a** (*array, cas.MX, cas.SX, cas.DM or numpy.ndarray*) – Array which should be converted.
- **dtype** (*str, optional*) – Name of the class to which the array will be converted. The default is 'auto'.

Returns
The converted object.

Return type
numpy.ndarray, cas.MX, cas.SX, cas.DM

cos

cos(*x*)

Calculates the cosinus of a given number or array

cosh

cosh(*x*)

Calculates the cosinus hyperbolicus of a given number or array

diag

diag(*x*)

Creates an diagonal matrix from a given vector.

Parameters
x (*array, numpy.ndarray, cas.MX, cas.SX or cas.DX, list*) – Vector containing the diagonal entries of the matrix.

Returns
Diagonal matrix with the desired diagonal elements.

Return type
array

exp

exp(*x*)

Calculates the exponential of a given number or array

eye

eye(*dim*)

Creates an array defining the idendity.

Parameters
dim (*int*) – Dimnension of the idendity matrix.

Raises

- **ValueError** – If the given dimension is not supported.
- **TypeError** – If an input parameter has not the right type.

Returns
y – Idendity matrix as an instance of array.

Return type
array

log

log(*x*)

Calculates the natural logarithm of a given number or array

matrix_power

matrix_power(*x*, *n*)

Raises a square matrix to the n-th power.

Parameters

- **x** (*int, float, numpy.ndarray, cas.MS, cas.SX or cas.DM*) – Number or array of which the n-th power should be computed.
- **n** (*int or float*) – Number defining the exponent.

max

max(**args*)

Return the maximal value of the arguments

min

min(**args*)

Returns the minimal value of the arguments

norm

norm(*x*, *order=None*)

Returns the norm of a vector or matrix.

Parameters

- **x** (*array, numpy.ndarray, cas.MX, cas.SX or cas.DM*) – Vector or matrix of which the norm should be calculated.
- **order** (*number or str, optional*) – String defining the type of the norm. Posiible values are 1, 2, 'fro' or 'inf'. The default is None.

ones

ones(*dim*)

Creates an array with only entries equal to one.

Parameters
dim (*int or tuple*) – Dimension of the array.

Raises

- **ValueError** – If the given dimension is not supported.
- **TypeError** – If the given dimension has not the right type.

Returns
y – An array of the given dimension with only entries equal to one.

Return type
array

power

power(*x*, *n*)

Calculates the (elementwise) n-th power of a number or array.

Parameters

- **x** (*int, float, numpy.ndarray, cas.MS, cas.SX or cas.DM*) – Number or array of which the n-th power should be computed.
- **n** (*int or float*) – Number defining the exponent.

reshape

reshape(*a*, *new_size*)

Reshape an array to a new size.

Parameters

- **a** (array) – .
- **new_size** (*tuple*) – New shape.

Returns

An array instance with the new shape.

Return type

array

sin

sin(*x*)

Calculates the sinus of a given number or array

sinh

sinh(*x*)

Calculates the sinus hyperbolicus of a given number or array

sqrt

sqrt(*x*)

Calculates the square root of a given number or array

tan

tan(*x*)

Calculates the tangens of a given number or array

tanh

tanh(*x*)

Calculates the tangens hyperblicus of a given number or array

zeros

zeros(*dim*)

Creates an array with only zero entries.

Parameters
dim (*int or tuple*) – Dimension of the array.

Raises

- **ValueError** – If the given dimension is not supported.
- **TypeError** – If the given dimension has not the right type.

Returns
y – An array of the given dimension with only zero entries.

Return type
array

Attributes

inf	Constant to define infinity.
pi	Constant to define the number pi.

inf

inf = inf

Constant to define infinity.

Type
float

pi

pi = 3.141592653589793

Constant to define the number pi.

Type
float

1.5 Examples

In addition to the information from the *API References*, the following examples are intended to provide guidance for implementing your own problems.

To show the different possibilities of the nMPyC package, we illustrate them with different examples.

Therefore, the chemical reactor is a nonlinear autonomous problem, the inverted pendulum is a linear quadratic problem, the heat pump is a nonlinear time-varying problem and the 2d investment problem is a discounted problem.

1.5.1 Chemical Reactor

We consider a single first-order, irreversible chemical reaction in an isothermal CSTR

$$A \to B.$$

The material balances and the system data are provided in [DAR11] and is given by the nonlinear model

$$c_A(k+1) = c_A(k) + h\left(\frac{Q(k)}{V}(c_f^A - c_A(k)) - k_r c_A(k)\right)$$
$$c_B(k+1) = c_B(k) + h\left(\frac{Q(k)}{V}(c_f^B - c_B(k)) + k_r c_B(k)\right),$$

in which $c_A \geq 0$ and $c_B \geq 0$ are the molar concentrations of A and B respectively, and $Q \leq 20$ (L/min) is the flow through the reactor. The constants and their meanings are given in table below.

Reactor constants			
		value	unit
feed concentration of A	c_f^A	1	mol/L
feed concentration of B	c_f^B	0	mol/L
volume of the reactor	V_R	10	L
rate constant	k_r	1.2	L/(mol min)
equilibrium	(c_e^A, c_e^B, Q_e)	$(\frac{1}{2}, \frac{1}{2}, 12)$	
initial value	(c_0^A, c_0^B)	$(0.4, 0.2)$	

To initialize the system dynamics a function that implements $f(x,u)$, where $x = (c_A, c_B)^T$ and $u = Q$ has to be defined.

```
V = 10.
cf_A = 1.
cf_B = 0.
k_r = 1.2

def f(x,u):
    y = nmpyc.array(2)
    y[0] = x[0] + 0.5*((u[0]/V) *(cf_A - x[0]) - k_r*x[0])
    y[1] = x[1] + 0.5*((u[0]/V) *(cf_B - x[1]) + k_r*x[1])
    return y
```

After that, the nMPyC system object can be set by calling

```
system = nmpyc.system(f, 2, 1, system_type='discrete')
```

In the next step, the objective is defined by using the stage cost given by

$$\ell(c_A(k), c_B(k), Q(k)) = \frac{1}{2}|c_A(k) - \frac{1}{2}|^2 + \frac{1}{2}|c_B(k) - \frac{1}{2}|^2 + \frac{1}{2}|Q(k) - 12|^2$$

Since we do not need terminal cost, we can initialize the objective directly using the following implementation.

```
def l(x,u):
    return 0.5 * (x[0]-0.5)**2 + 0.5 * (x[1]-0.5)**2 + 0.5 * (u[0]-12)**2

objective = nmpyc.objective(l)
```

In terms of the constraints we assume that

$$\begin{aligned} 0 &\leq x_1(k) \\ &< \infty \\ \text{for } k &= 0, \dots, N \\ 0 &\leq x_2(k) \\ &< \infty \\ \text{for } k &= 0, \dots, N \\ 0 &\leq u(k) \\ &\leq 20 \\ \text{for } k &= 0, \dots, N-1. \end{aligned}$$

This can be realized in the code as follows:

```
constraints = nmpyc.constraints()
lbx = nmpyc.zeros(2)
ubu = nmpyc.ones(1)*20
lbu = nmpyc.zeros(1)
constraints.add_bound('lower','state', lbx)
constraints.add_bound('lower','control', lbu)
constraints.add_bound('upper','control', ubu)
```

Moreover, we consider the equilibrium (c_e^A, c_e^B, Q_e) as th terminal condition for our optimal control problem, which is implemented as

```
xeq = nmpyc.array([0.5,0.5])
def he(x):
    return x - xeq
constraints.add_constr('terminal_eq', he)
```

After all components of the optimal control problem have been implemented, we can now combine them into a model and start the MPC loop. For this Purpose, we define

$$x(0) = (0.4, 0.2)^T$$

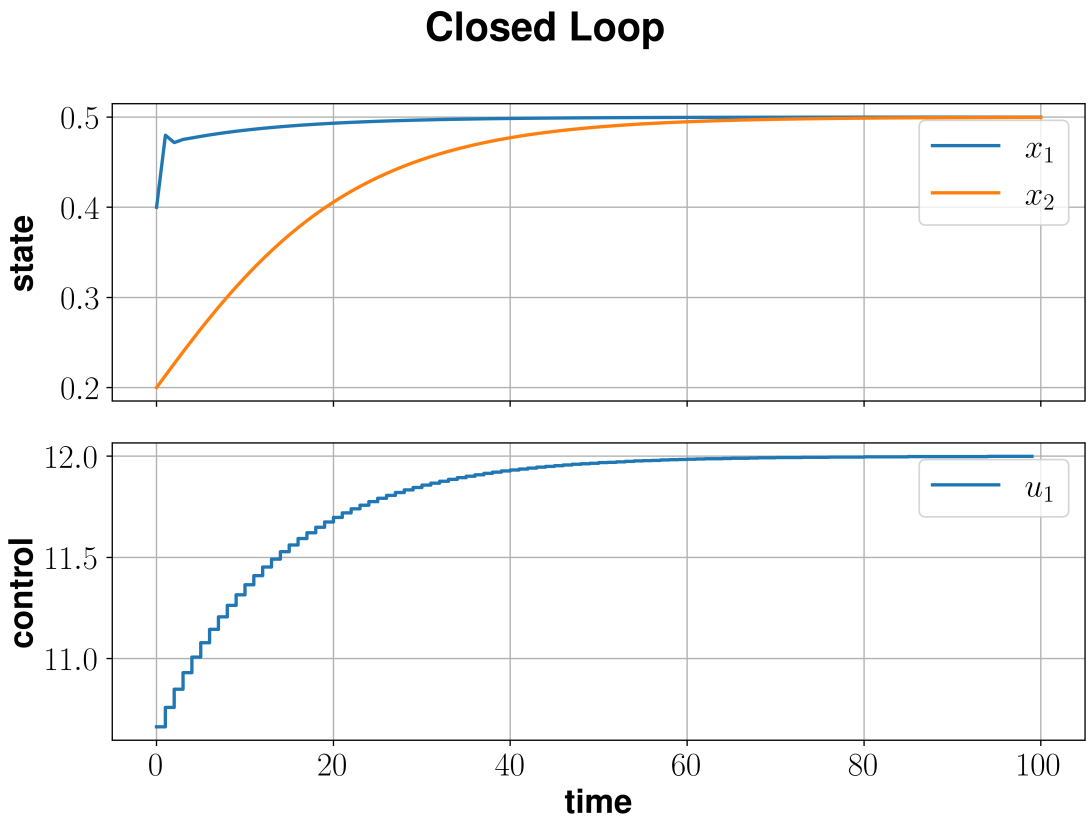
and set $N = 15$, $K = 100$.

```
model = nmpyc.model(objective,system,constraints)
x0 = nmpyc.array([0.4,0.2])
res = model.mpc(x0,15,100)
```

Following the simulation we can visualize the results by calling

```
res.plot()
```

which generates the plot bellow.



1.5.2 Inverted Pendulum

We consider the mechanical model of an inverted rigid pendulum mounted on a carriage, see [Grune21], [GruneP17].

By means of physical laws an "exact" differential equation model can be derived. However, since in our case we like to obtain a linear quadratic problem, we linearize the differential equation at the origin. Thus, we obtain the system dynamics defined by

$$\dot{x}(t) = \begin{pmatrix} 0 & 1 & 0 & 0 \\ g & -k & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} x(t) + \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} u(t).$$

Here, the state vector $x \in \mathbb{R}^4$ consists of 4 components. x_1 corresponds to the angle ψ of the pendulum, which increases counterclockwise, where $x_1 = 0$ corresponds to the upright pendulum. x_2 is the angular velocity, x_3 the position of the carriage and x_4 its velocity. The control u is the acceleration of the carriage. The constant $k = 0.1$ describes the friction of the pendulum and the constant $g \approx 9.81 m/s^2$ is the acceleration due to gravity.

Since the system dynamics are linear, we can initialize them using the LQP method.

```
g = 9.81
k = 0.1
A = nmpyc.array([[0, 1, 0, 0],
                 [g, -k, 0, 0],
                 [0, 0, 0, 1],
                 [0, 0, 0, 0]])
B = nmpyc.array([0, 1, 0, 1])
```

(continues on next page)

(continued from previous page)

```
system = nmpyc.system.LQP(A, B, 4, 1, 'continuous',
                          sampling_rate=0.1, method='rk4')
```

Note that we have to use one of the fixed step methods as *euler*, *heun* or *rk4* as integration method if we like to exploit the linear quadratic structure of the problem in the optimization.

In the next step, we have to define the objective of the optimal control problem. In doing so, we assume the stage cost

$$\ell(x, u) = 2x^T x + 4u^T u.$$

Since we assume no terminal cost, we can implement the objective as shown in the following code snippet.

```
Q = 2*nmpyc.eye(4)
R = 4*nmpyc.eye(1)
objective = nmpyc.objective.LQP(Q, R)
```

Again, we use the LQP method to exploit the linear quadratic structure of the problem later.

In terms of the constraints we assume the state constraints

$$-9 \leq x_i(t) \leq 5$$

for $i = 1, \ldots, 4$ and the control constraint

$$-20 \leq u(t) \leq 6$$

This can be realized in the code as

```
constraints = nmpyc.constraints()
lbx = nmpyc.zeros(4)*(-9)
ubx = nmpyc.ones(4)*5
constraints.add_bound('lower','state', lbx)
constraints.add_bound('upper','state', ubx)
constraints.add_bound('lower', 'control', nmpyc.array([-20]))
constraints.add_bound('upper', 'control', nmpyc.array([6]))
```

After all components of the optimal control problem have been implemented, we can now combine them into a model and start the MPC loop. For this Purpose, we define the inital value

$$x(0) = (1, 1, 1, 1)^T$$

and set $N = 20$, $K = 100$.

```
model = nmpyc.model(objective,system,constraints)
x0 = nmpyc.array([1, 1, 1, 1])
res = model.mpc(x0,20,100)
```

Since the problem is linear-quadratic, the program automatically takes advantage of this fact and uses the appropriate solver osqp. To change this and use for example the SciPy solver SLSQP, we can use the *set_options* method before calling model.mpc().

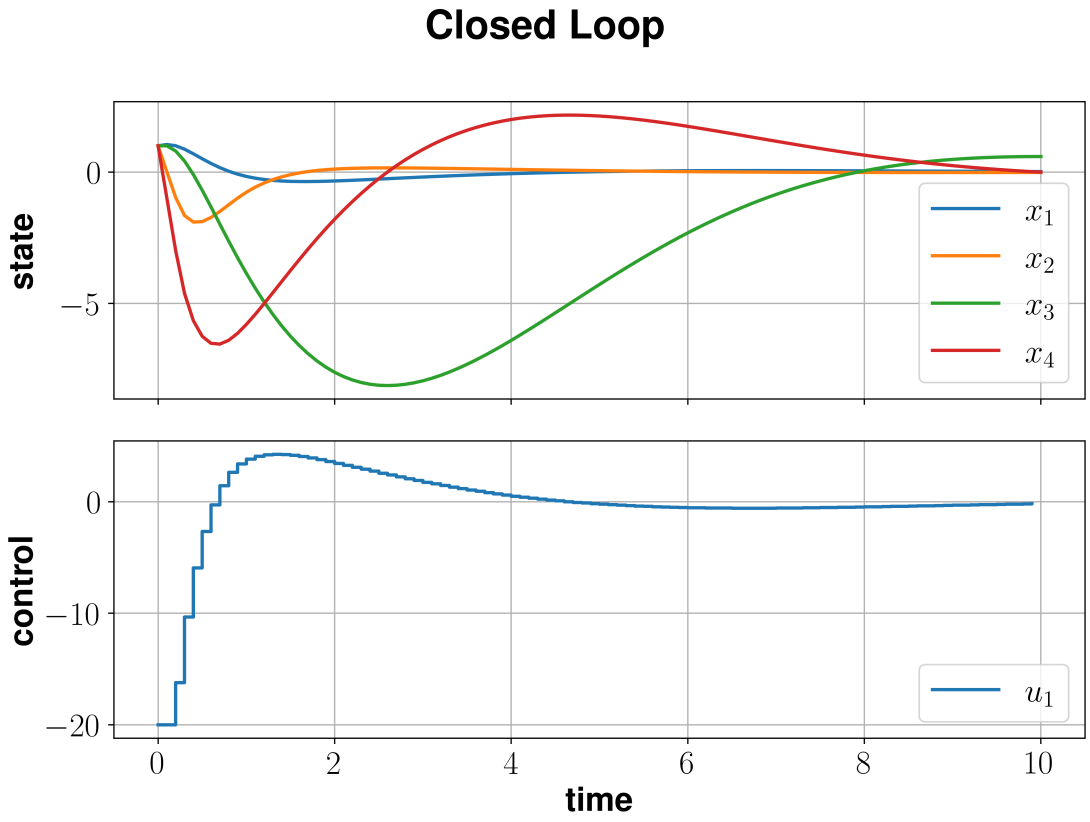
```
model.opti.set_options(dict(solver='SLSQP'))
```

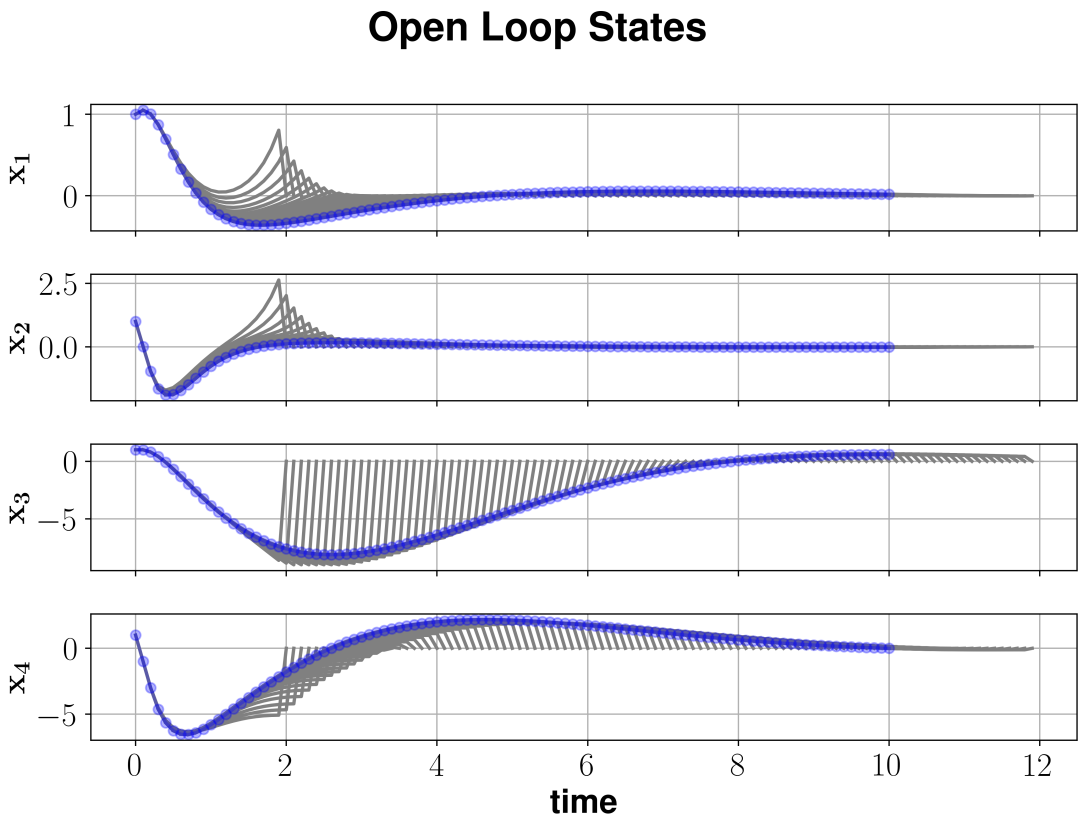
Note that changing the optimizer usually does not have any advantage and is therefore not necessarily recommended. At this point we only like to demomnstrate the use of this function.

Following the simulation we can visualize the open and closed loop results by calling

```
res.plot() # plot closed loop results
res.plot('state', show_ol=True) # plot open loop states
```

which generates the plots bellow.





1.5.3 Heat Pump

This example describes a home heating system that involves the optimal control of a small heat pump coupled to a floor heating system. The corresponding dynamic model is introduced in [LHDI10] and is given by

$$\dot{x_1} = \frac{-k_{WR}}{\rho_W c_W V_H} x_1 + \frac{k_{WR}}{\rho_W c_W V_H} x_2 + \frac{1}{\rho_W c_W V_H} u \tag{1.4}$$

$$\dot{x_2} = \frac{k_{WR}}{k_G \tau_G} x_1 - \frac{k_{WR} + k_G}{k_G \tau_G} x_2 + \frac{1}{\tau_G} T_{\text{amb}}, \tag{1.5}$$

where x_1 denotes the temperature of the water returning from the heating, x_2 denotes the room temperature and u is the heat supplied from the heat pump to the floor. Further, the ambient temperature

$$T_{\text{amb}}(t) = 2.5 + 7.5 \sin\left(\frac{2\pi t}{t_f} - \frac{\pi}{2}\right)$$

describes a sinusoidal disturbance from the outside temperature where $t_f = 24$. The remaining constants are summarized in the table below.

Reactor constants			
		value	unit
density of the water	ρ_W	997	kg/m^3
specific heat capacity of water	c_W	4.1851	J/kgK
volume of the water	V_H	7.4	m^3
thermal conductivity between water and the room	k_{WR}	510	W/K
thermal conductivity between the room and the environment	k_G	125	W/K
thermal time constant of the room	τ_G	260	s

First, we have to implement the outside temperature in the code to define our system dynamics.

```
t_f = 24
def T_amb(t):
 return 2.5 + 7.5*nmpyc.sin((2*nmpyc.pi*t)/t_f - (nmpyc.pi/2))
```

After that, we can define the right hand side of the system by

```
rho_W = 997
c_W = 4.1851
V_H = 7.4
k_WR = 510
k_G = 125
thau_G = 260
def f(t,x,u):
   y = nmpyc.array(2)
   y[0] = (-k_WR/(rho_W*c_W*V_H)*x[0]
             + k_WR/(rho_W*c_W*V_H)*x[1]
             + 1/(rho_W*c_W*V_H)*u[0])
   y[1] = (k_WR/(k_G*thau_G)*x[0]
             - (k_WR + k_G)/(k_G*thau_G)*x[1]
             + (1/thau_G)*T_amb(t))
   return y
```

And finally initialize the system by

```
system = nmpyc.system(f, 2, 1, 'continuous', sampling_rate=0.5, method='euler')
```

In the heating system the conflict between energy and thermal comfort arises. Thus, the stage cost reads

$$\ell(x,u) = \frac{u}{P_{\max}} + (x_2 - T_{\text{ref}})^2,$$

where $P_{\max} = 15000(W)$ is the maximal power of the heating pump and $T_{\text{ref}} = 22°C$ is the desired temperature of the room. The reference temperature T_{ref} can be selected differently – depending on the individual thermal comfort.

According to this we can initialize our objective by

```
P_max = 15000
T_ref = 22
def l(x,u):
   return (u[0]/P_max) + (x[1]-T_ref)**2
```

and implement the control constraint

$$0 \leq u(t) \leq P_{max}$$

as

```
constraints = nmpyc.constraints()
constraints.add_bound('lower', 'control', nmpyc.array([0]))
constraints.add_bound('upper', 'control', nmpyc.array([P_max]))
```

After all components of the optimal control problem have been implemented, we can now combine them into a model and start the MPC loop. For this purpose, we define

$$x(0) = (22, 19.5)^T$$

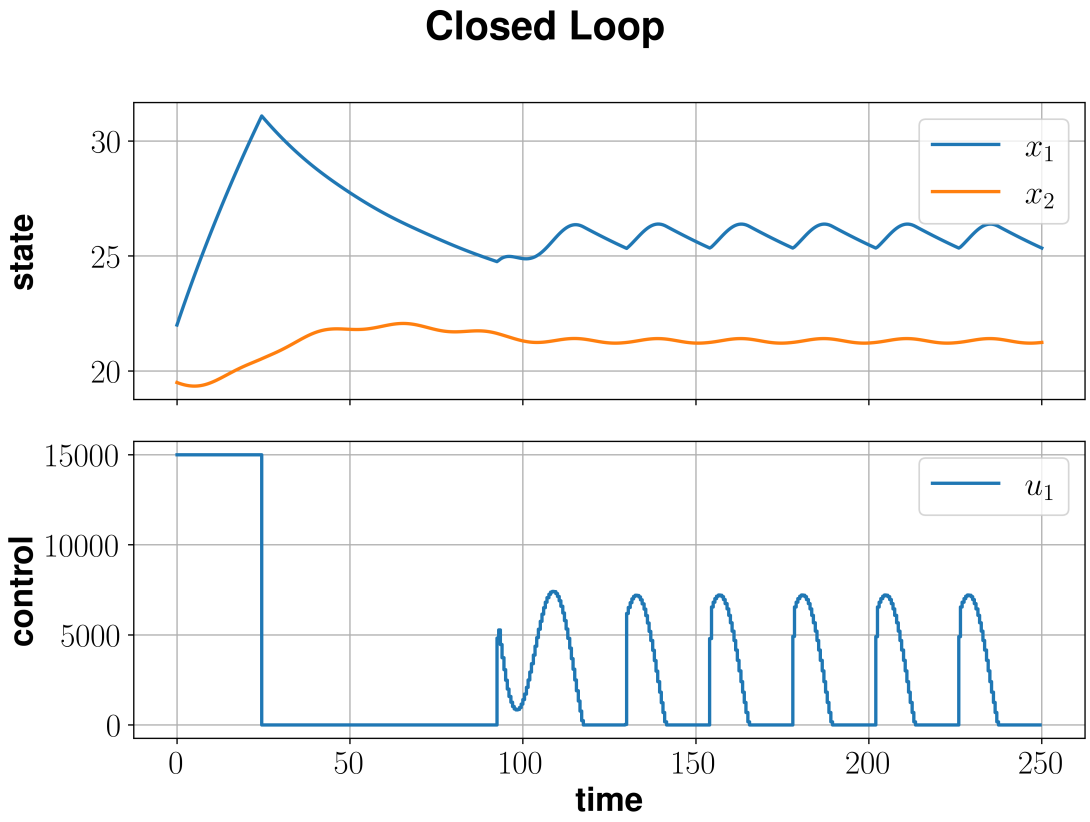
and set $N = 30$ and $K = 500$.

```
model = nmpyc.model(objective,system,constraints)
x0 = nmpyc.array([22., 19.5])
res = model.nmpyc(x0,N,K)
```

Following the simulation we can visualize the results by calling

```
res.plot()
```

which generates the plot bellow.



1.5.4 2d Investment Problem

To examplify a discounted problem we consider a 2d variant of an investment problem, originally introduced in [HKHF03] and furhter explored in [GruneSS15].

The system dynamics for this problem are given by

$$\dot{x}_1(t) = x_2(t) - \sigma x_1(t)$$
$$\dot{x}_2(t) = u(t)$$

where we set $\sigma = 0.25$ for our calculations.

To implement the dynamics we have to initialize a function that implements the right hand side of the dynamics.

```
sigma = 0.25
def f(x,u):
    y = nmpyc.array(2)
```

(continues on next page)

(continued from previous page)

```
    y[0] = x[1]-sigma *x[0]
    y[1] = u
    return y
```

After that, the nMPyC system object can be set by calling

```
system = nmpyc.system(f, 2, 1, 'continuous', sampling_rate=0.2, method='heun')
```

To model the payoff of the investment problem we assume th stage cost

$$\ell(x,u) = -(R(x_1) - c(x_2) - v(u))$$

where $R(x_1) = k_1\sqrt{x_1} - x_1/(1+k_2x_1^4)$ is a revenue function of the firm with a convex segment due to increasing returns. $c(x_2) = c_1x_2 + c_2x_2^2/2$ denotes adjustment costs of investment and $v(u) = \alpha u^2/2$ represents adjustment costs of the change of investment. The convex segment in the payoff function just mentioned is likely to generate two domains of attraction. Additionally we choose $k_1 = 2$, $k_2 = 0.0117$, $c_1 = 0.75$, $c_2 = 2.5$ and $\alpha = 12$ for our computations.

With the nMPyC package the implemnetiation of the objective corresponding to this costs can be done as follws.

```
def l(x,u):
    R = k1*x[0]**(1/2)-x[0]/(1+k2*x[0]**4)
    c = c1*x[1]+(c2*x[1]**2)/2
    v = (alpha*u[0]**2)/2
    return -(R - c - v)

objective = nmpyc.objective(l)
```

Since this problem is unconstrained we can now initialize our model by

```
model = nmpyc.model(objective,system)
```

For our simulation we assume set the discount factor to

$$\beta = e^{-\delta h}$$

where $h = 0.2$ is our samplimng rate and $\delta = 0.04$ is the continuous discount rate.

It can now be shown that this problem has two domains of attraction, one at roughly $x^* = (0.5, 0.2)$ and the other roughly at $x^* = (4.2, 1.1)$. Now we choose ifferent initial values from both domains of attraction to test, if we can replicate the two domains of attraction for a finite decision horizon by using nonlinear model predictive control. For this purpose we set the MPC horizon $N = 50$ and the number of MPC iterations to $K = 500$.

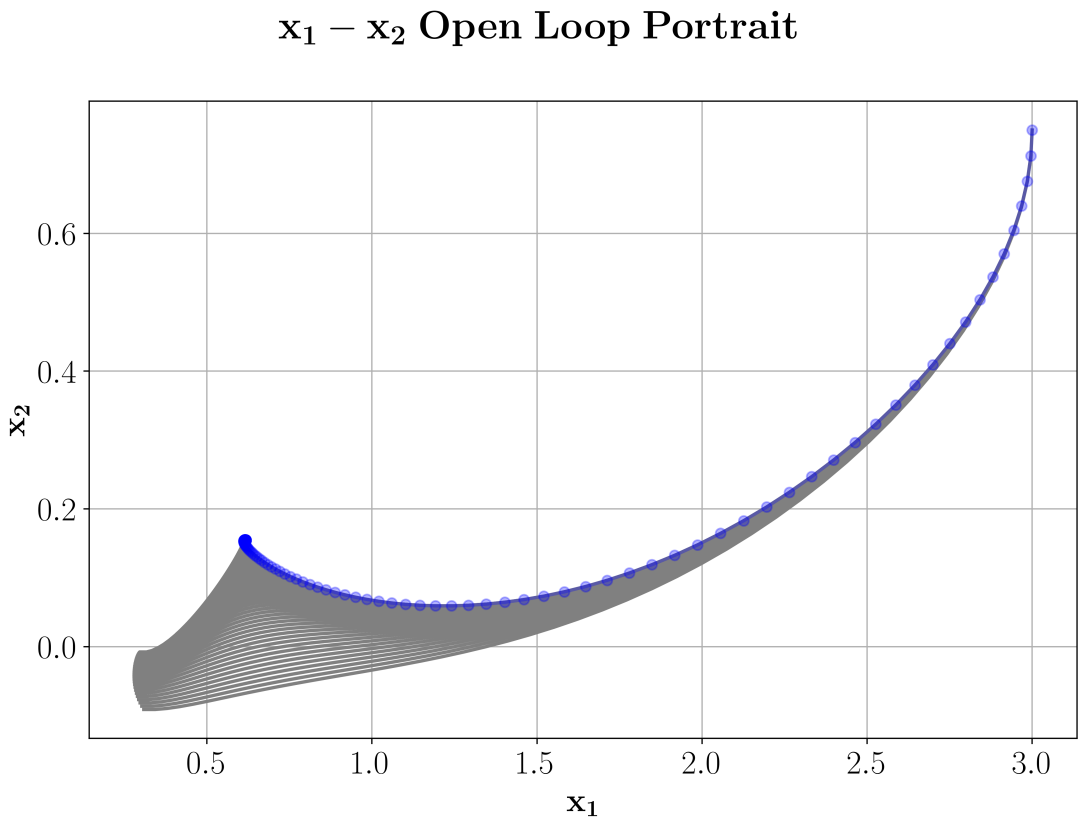
This leads to the following code for running the closed loop simulation for the discounted problem.

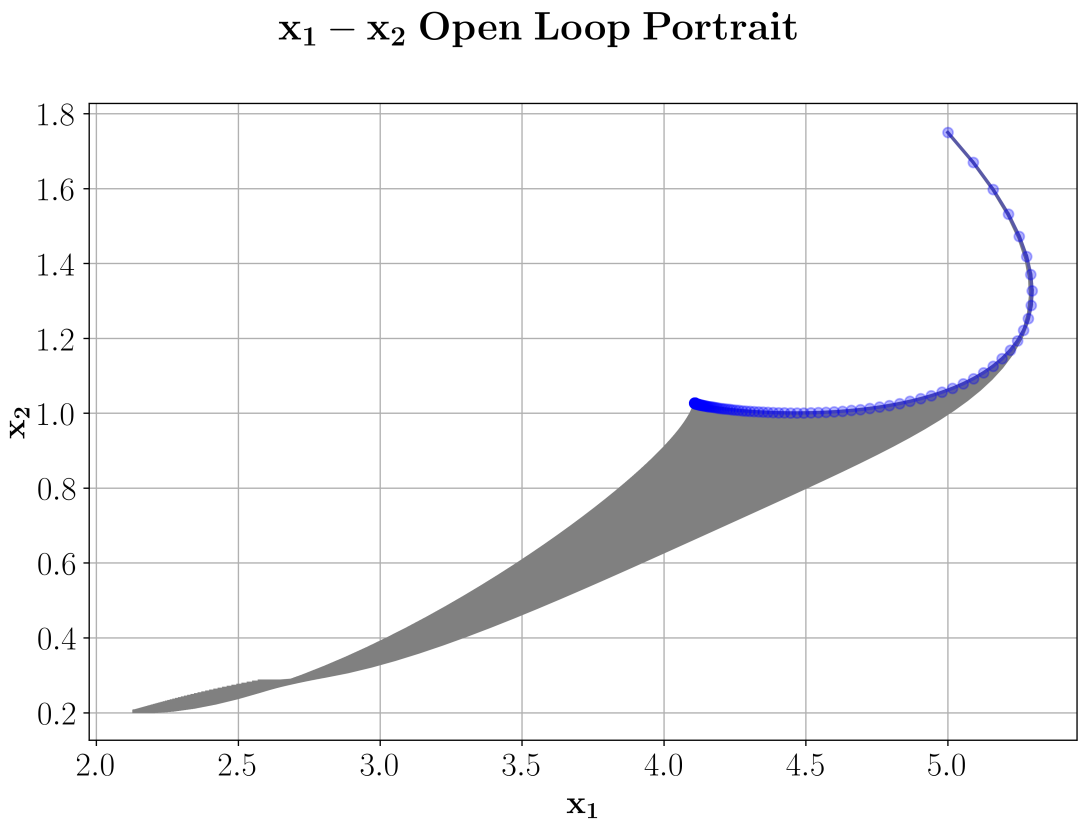
```
discount = nmpyc.exp(-0.04*0.2)
N = 50
K = 500

x0 = nmpyc.array([3.0,0.75])
res1 = nmpyc.mpc(x0,N,K,discount)

x0 = nmpyc.array([5.0,1.75])
res2 = nmpyc.mpc(x0,N,K,discount)
```

Looking at the phase portraits of the two simulations, we can confirm that we really converge against the two different equilibria with the closed loop trajectory. The phase portraits of our simulations can be plotted with the nMPyC package by calling

```
res1.plot('phase', phase1='x_1', phase2='x_2', show_ol=True)
res2.plot('phase', phase1='x_1', phase2='x_2', show_ol=True)
```

The option `show_ol=True` will also plot the pahase portraits of the open loop simulations of each iteration, which leads the output below.





1.6 Templates

In addition to the examples, we also provide templates to facilitate the implementation.

To take advantage of the different structures of the problems, we have implemented templates for the following problem types.

1.6.1 Time-variant Problem

```
# Import nMPyc package
import nmpyc

# Define system parameters
nx = .. # dimension of state
nu = .. # dimension of control
system_type = .. # system type: continuous or discrete
sampling_rate = 1. # sampling rate h (optional)
t0 = 0. # initial time (optional)
method = 'cvodes' # integrator (optinal)

# Define MPC parameters
N = .. # MPC horizon
K = .. # MPC itertaions
x0 = .. # initial value
```

(continues on next page)

(continued from previous page)

```
discount = 1. # dicount factor (optional)

# Define right hand side of the system dynamics
def f(t, x, u):
   y = nmpyc.array(nx)
   ..
   return y

# Initialize system dynamics
system = nmpyc.system(f, nx, nu, system_type, sampling_rate, t0, method)

# Define stage cost
def l(t, x, u):
   return ..

# Define terminal cost (optional)
def F(t, x):
   return ..

# Initialize objective
objective = nmpyc.objective(l, F)

# Define constraints
constraints = nmpyc.constraints()

# Add bounds (optional)
lbx = .. # lower bound for states
constraints.add_bound('lower', 'state', lbx)
ubx = .. # upper bound for states
constraints.add_bound('upper', 'state', ubx)
lbu = .. # lower bound for control
constraints.add_bound('lower', 'control', lbu)
ubu = .. # upper bound for control
constraints.add_bound('upper', 'control', ubu)
lbend = .. # lower bound for terminal state
constraints.add_bound('lower', 'terminal', lbend)
ubend = .. # upper bound for terminal state
constraints.add_bound('upper', 'terminal', ubend)

# Add equality constraints (h(t,x,u)=0, optional)
len_eqconstr = .. # number of equality constraints
def h(t, x, u):
   c_eq = nmpyc.array(len_eqconstr)
   ..
   return c_eq

constraints.add_constr('eq', h)

# Add inequality constraints (g(t,x,u)>=0, optional)
len_ineqconstr = .. # number of inequality constraints
def g(t, x, u):
   c_ineq = nmpyc.array(len_ineqconstr)
```

(continues on next page)

(continued from previous page)

```
    ..
    return c_ineq

constraints.add_constr('ineq', g)

# Add terminal equality constraints (H(t,x)=0, optional)
len_terminaleq = .. # number of terminal equality constraints
def H(t, x):
    cend_eq = nmpyc.array(len_terminaleq)
    ..
    return cend_eq

constraints.add_constr('terminal_eq', H)

# Add terminal equality constraints (G(t,x)>=0, optional)
len_terminalineq = .. # number of terminal equality constraints
def G(t, x):
    cend_ineq = nmpyc.array(len_terminalineq)
    ..
    return cend_ineq

constraints.add_constr('terminal_ineq', G)

# Initialize model
model = nmpyc.model(objective, system, constraints)

# Start MPC loop
res = model.mpc(x0, N, K, discount)

# Plot results
res.plot()
```

1.6.2 Autonomous Problem

```
# Import nMPyc package
import nmpyc

# Define system parameters
nx = .. # dimension of state
nu = .. # dimension of control
system_type = .. # system type: contiunous or discrete
sampling_rate = 1. # sampling rate h (optional)
method = 'cvodes' # integrator (optinal)

# Define MPC parameters
N = .. # MPC horizon
K = .. # MPC itertaions
x0 = .. # initial value
discount = 1. # dicount factor (optional)
```

(continues on next page)

(continued from previous page)

```
# Define right hand side of the system dynamics
def f(x, u):
    y = nmpyc.array(nx)
    ..
    return y

# Initialize system dynamics
system = nmpyc.system(f, nx, nu, system_type, sampling_rate, method=method)

# Define stage cost
def l(x, u):
    return ..

# Define terminal cost (optional)
def F(x):
    return ..

# Initialize objective
objective = nmpyc.objective(l, F)

# Define constraints
constraints = nmpyc.constraints()

# Add bounds (optional)
lbx = .. # lower bound for states
constraints.add_bound('lower', 'state', lbx)
ubx = .. # upper bound for states
constraints.add_bound('upper', 'state', ubx)
lbu = .. # lower bound for control
constraints.add_bound('lower', 'control', lbu)
ubu = .. # upper bound for control
constraints.add_bound('upper', 'control', ubu)
lbend = .. # lower bound for terminal state
constraints.add_bound('lower', 'terminal', lbend)
ubend = .. # upper bound for terminal state
constraints.add_bound('upper', 'terminal', ubend)

# Add equality constraints (h(x,u)=0, optional)
len_eqconstr = .. # number of equality constraints
def h(x, u):
    c_eq = nmpyc.array(len_eqconstr)
    ..
    return c_eq

constraints.add_constr('eq', h)

# Add inequality constraints (g(x,u)>=0, optional)
len_ineqconstr = .. # number of inequality constraints
def g(x, u):
    c_ineq = nmpyc.array(len_ineqconstr)
    ..
    return c_ineq
```

(continues on next page)

(continued from previous page)

```
constraints.add_constr('ineq', g)

# Add terminal equality constraints (H(x)=0, optional)
len_terminaleq = .. # number of terminal equality constraints
def H(x):
   cend_eq = nmpyc.array(len_terminaleq)
   ..
   return cend_eq

constraints.add_constr('terminal_eq', H)

# Add terminal equality constraints (G(x)>=0, optional)
len_terminalineq = .. # number of terminal equality constraints
def G(x):
   cend_ineq = nmpyc.array(len_terminalineq)
   ..
   return cend_ineq

constraints.add_constr('terminal_ineq', G)

# Initialize model
model = nmpyc.model(objective, system, constraints)

# Start MPC loop
res = model.mpc(x0, N, K, discount)

# Plot results
res.plot()
```

1.6.3 Linear Quadratic Problem

```
# Import nMPyc package
import nmpyc

# Define system parameters
nx = .. # dimension of state
nu = .. # dimension of control
system_type = .. # system type: continuous or discrete
sampling_rate = 1. # sampling rate h (optional)
method = 'cvodes' # integrator (optinal)

# Define MPC parameters
N = .. # MPC horizon
K = .. # MPC itertaions
x0 = .. # initial value
discount = 1. # dicount factor (optional)

# Define linear right hand side of the system dynamics f(x,u) = Ax + Bu
A = ..
```

(continues on next page)

(continued from previous page)

```
B = ..

# Initialize system dynamics
system = nmpyc.system.LQP(A, B, nx, nu, system_type, sampling_rate, method=method)

# Define quadratic stage cost l(x,u) = x^TQx + u^TRu + 2*x^TNx
Q = ..
R = ..
N = nmpyc.zeros((nx,nu)) # optional

# Define terminal cost x^TPx
P = nmpyc.zeros((nx,nx)) # optional

# Initialize objective
objective = nmpyc.objective.LQP(Q, R, N, P)

# Define constraints
constraints = nmpyc.constraints()

# Add bounds (optional)
lbx = .. # lower bound for states
constraints.add_bound('lower', 'state', lbx)
ubx = .. # upper bound for states
constraints.add_bound('upper', 'state', ubx)
lbu = .. # lower bound for control
constraints.add_bound('lower', 'control', lbu)
ubu = .. # upper bound for control
constraints.add_bound('upper', 'control', ubu)
lbend = .. # lower bound for terminal state
constraints.add_bound('lower', 'terminal', lbend)
ubend = .. # upper bound for terminal state
constraints.add_bound('upper', 'terminal', ubend)

# Add equality constraints (Ex + Fu = b, optional)
E_eq = ..
F_eq = ..
b_eq = ..
constraints.add_constr('eq', E_eq, F_eq, b_eq)

# Add equality constraints (Ex + Fu >= b, optional)
E_ineq = ..
F_ineq = ..
b_ineq = ..
constraints.add_constr('ineq', E_ineq, F_ineq, b_ineq)

# Add terminal equality constraints (Hx = 0, optional)
H_eq = ..
constraints.add_constr('terminal_eq', H_eq)

# Add terminal equality constraints (Hx >= 0, optional)
H_ineq = ..
constraints.add_constr('terminal_ineq', H_ineq)
```

(continues on next page)

(continued from previous page)

```
# Initialize model
model = nmpyc.model(objective, system, constraints)

# Start MPC loop
res = model.mpc(x0, N, K, discount)

# Plot results
res.plot()
```

Note: Any problem, whether nonlinear, linear, autonomous, or time-varying, can be initialized as a nonlinear time-varying optimal control problem. Therefore, you can always fall back on such an implementation. However, if you know the structure of your problem and this is to be exploited by the program in order to possibly speed up the simulation, it is necessary to initialize the problem as such.

1.7 FAQ

1.7.1 Why use the *nmpyc_array* module?

The idea of the *nmpyc.nmpyc_array* module is to provide a simple syntax for the input, which is similar to the one of NumPy. At the same time we ensure a switching between symbolic calculation with CasADi and completely numeric calculations.

A completely numerical calculation is advantageous, for example, if non-differentiable functions have to be evaluated at critical points, e.g. the norm at the origin. Here the algorithmic differentiation of CasADi can lead to problems.

Therefore this module is built in a way that the array class can automatically switch between CasADi and NumPy objects. In addition, the individual functions are built in such a way that they recognize the type of the input and call the appropriate function from NumPy or CasADi accordingly.

1.7.2 What to do if a function is not defined in the *nmpyc_array* module?

We have tried to implement the most common functions. Nevertheless, it can happen that a certain function that you need is missing.

If this is the case, there is the possibility to implement an own overload of this function. A good orientation for this is the already programmed functions in the *nmpyc.nmpyc_array* module.

Another possibility in such cases is to use the call `x.A` to access the CasADi or NumPy array in which the entries of `x` are stored. Afterwards the appropriate necessary computations can be accomplished with the help of NumPy or CasADi functions. Note, however, that in this way if applicable no smooth change between numeric and symbolic calculation is possible.

1.7.3 Which solver should be used?

In the most cases, the automatic selection of the solver by the program is recommended. In this way, if possible, the linear quadratic structure of a problem is exploited or at least algorithmic differentiation is still exploited to perform an advantageous optimization.

However, as already mentioned, this algorithmic differentiation can also lead to problems in some cases. For example, if a non-differentiable function must be evaluated at critical points, e.g. the norm at the origin. In such cases, a numerical calculation should be used for the optimization and a SciPy solver, such as SLSQP, should be selected.

1.7.4 Which discretization method should be used?

In our numerical simulations we have experienced that mostly a fixed step integration method like euler is sufficient to guarantee the necessary accuracy during the simulation. The advantage of these methods is that with them the largest speed up among the available integrators can be achieved.

However, if it is necessary to achieve higher integration accuracy by an adaptive integration method, one of the CasADi integrators, e.g. cvodes, should always be chosen if possible.

The SciPy integrators should only be considered as a kind of backup in case the other methods fail, since they lead to an above-average lag of time during the simulation in our implementation.

1.7.5 What to do if I a LaTeX Error occurs while plotting?

In our experience such errors occur mainly on MacOS if Spyder is used for programming, which in turn is opened via the Anaconda Navigator. In this case it is sufficient to open spyder directly and not to take the detour via the Anaconda Navigator to solve the problem.

However, if this procedure does not solve the problem or the problem has another cause, it is also possible to disable the LaTeX labeling of the plots by setting the option `usetex=False`. For more details see *nmpyc.result.result.plot()*.

1.8 How to Cite

If you use **nMPyC** for published work please cite it as

```
@misc{nmpyc,
       author = {Jonas Schie{\ss}l and Lisa Kr{\"u}gel},
       title ={{nMPyC} - A Python library for solving optimal control problems via MPC},
       howpublished = {\url{http://nmpyc.readthedocs.io/}},
       year = {2022}
}
```

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

For any specific algorithm, also consider citing the original author's paper.

1.9 References

BIBLIOGRAPHY

[DAR11] Moritz Diehl, Rishi Amrit, and James B. Rawlings. A lyapunov function for economic optimizing model predictive control. *IEEE Transactions on Automatic Control*, 56(3):703–707, mar 2011. doi:10.1109/TAC.2010.2101291.

[Grune21] Lars Grüne. Mathematical control theory. 2021. Lecture Notes. URL: https://num.math.uni-bayreuth.de/de/team/lars-gruene/skripten/kontrolltheorie/kt_2021_en.pdf.

[GruneP17] Lars Grüne and Jürgen Pannek. *Nonlinear Model Predictive Control : Theory and Algorithms. 2nd Edition*. Communications and Control Engineering. Springer, Cham, Switzerland, 2017. URL: https://eref.uni-bayreuth.de/35127/.

[GruneSS15] Lars Grüne, Willi Semmler, and Marleen Stieler. Using nonlinear model predictive control for dynamic decision problems in economics. *Journal of Economic Dynamics and Control*, 60:112–133, 2015. URL: https://eref.uni-bayreuth.de/20841/.

[HKHF03] Josef L. Haunschmied, Peter M. Kort, Richard F. Hartl, and Gustav Feichtinger. A DNS-curve in a two-state capital accumulation model: a numerical analysis. *Journal of Economic Dynamics and Control*, 27(4):701–716, feb 2003. doi:10.1016/S0165-1889(01)00070-7.

[LHDI10] Filip Logist, Boris Houska, Moritz Diehl, and Jan Van Impe. Fast pareto set generation for nonlinear optimal control problems with multiple objectives. *Structural and Multidisciplinary Optimization*, 42(4):591–603, may 2010. doi:10.1007/s00158-010-0506-x.

PYTHON MODULE INDEX

n

INDEX

O

P

R

S

T

U

X

Z