GRAMPC-D documentation

Version 1.0

Daniel Burk, Andreas Völz, Knut Graichen

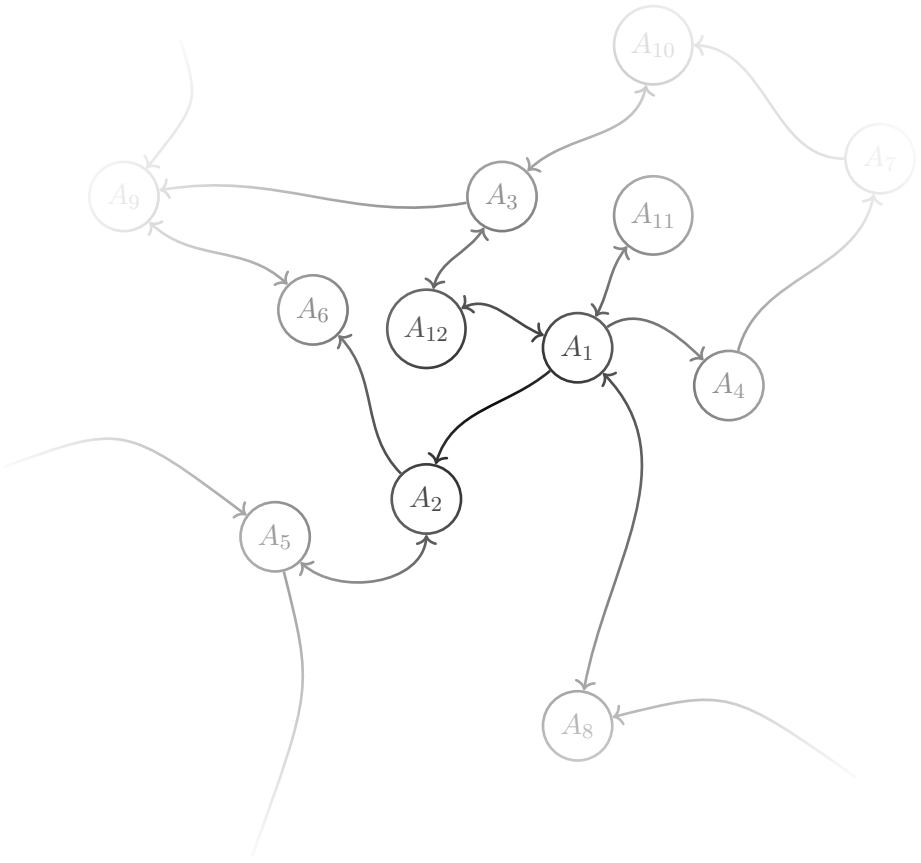
Chair of Automatic Control
Friedrich-Alexander-Universität Erlangen-Nürnberg

August 23, 2020

Contents

1 Introduction

This documentation explains how to use the DMPC-framework to solve up-scaled, nonlinear, time-continuous optimal control problems (OCP). It is capable of solving the global OCP using both a centralized and distributed controller. Regarding the distributed solution the ADMM algorithm is implemented that enables to fully decouple the global OCP into multiple local OCPs that can be solved in parallel. For the underlying minimization inside the ADMM algorithm as well as for the centralized solution the MPC-toolbox GRAMPC [2] is used that is tailored on embedded hardware. The software module Pybind11 [3] is used to create the Python-Interface. While this documentation focuses on the actual usage of the DMPC-framework, have a look into [1] for a more theoretical approach on the implemented algorithms.



2 Installation of GRAMPC-D

This section explains the installation of GRAMPC-D on Windows and Linux. In both cases, at first download the DMPC-framework from Github https://github.com/DanielBurk/GRAMPC-D and update all submodules.

```
git clone https://github.com/DanielBurk/GRAMPC-D
cd grampc-d
git submodule update --init
```

2.1 Building the software

To build GRAMPC-D using Linux is done by running the CMakeLists.txt and calling make afterwards.

```
cmake -DCMAKE_BUILD_TYPE=Release CMakeLists.txt
make
```

For building GRAMPC-D using Windows, the IDE *Visual Studio 19* is recommended. Therein, make sure to have installed both the C++-Buildtools and cmake. After starting Visual Studio, choose to open a folder and select the folder containing GRAMPC-D. Afterwards run the CMakeLists.txt by *Project→ Generate Cache for dmpc* and build the object files by *Build→Build All*.

2.2 Building the Python interface

To be able to use the Python interface, at first install Python 3 with the same architecture (32/64 bit) as your C++ compiler. Make sure to install the corresponding package manager *pip* as well and download the module *Matplotlib*. Note that only the usage of Python 3 is supported. To enable building the Python interface, open the main CMakeLists.txt of the DMPC-framework and change

```
set(PYTHON_AVAILABLE FALSE) -> set(PYTHON_AVAILABLE TRUE)
```

Afterwards repeat the steps in Section 2.1. The building procedure now generates a file inside the folder *bin* with a name familiar to *grampcd_interface.cp38-win_amd64.pyd*. It is composed by the name of the Python module (grampcd_interface), the version of Python (3.8), the used operating system (Windows) and the processor architecture (amd64). This is a standalone module of the whole framework that can be imported in Python by calling

```
import grampcd_interface
```

2.3 Running an example simulation

To check if everything went well, open the subfolder *main*. A set of example simulations are provided therein. Open the folder *coupled_watertanks* and run either the C++ or the Python script. The Python-script should generate the last plot of Section 5.3 while the C++-file should generate a set of .csv-files containing the corresponding trajectories.

3 System class

The class of systems considered in the DMPC-framework is described by

$$\min_{\boldsymbol{u}_i, i\in\mathcal{V}} \sum_{i\in\mathcal{V}} J_i(\boldsymbol{u}_i; \boldsymbol{x}_{i,0}) \tag{3.1a}$$

$$\text{s.t.} \quad \dot{\boldsymbol{x}}_i = \boldsymbol{f}_i(\boldsymbol{x}_i, \boldsymbol{u}_i, \tau) + \sum_{j\in \mathrm{N}_i^{\leftarrow}} \boldsymbol{f}_{ij}(\boldsymbol{x}_i, \boldsymbol{u}_i, \boldsymbol{x}_j, \boldsymbol{u}_j, \tau), \quad i \in \mathcal{V} \tag{3.1b}$$

$$\boldsymbol{x}_i(0) = \boldsymbol{x}_{i,0}, \quad i \in \mathcal{V} \tag{3.1c}$$

$$\boldsymbol{0} = \boldsymbol{g}_i(\boldsymbol{x}_i, \boldsymbol{u}_i, \tau), \quad i \in \mathcal{V} \tag{3.1d}$$

$$\boldsymbol{0} = \boldsymbol{g}_{ij}(\boldsymbol{x}_i, \boldsymbol{u}_i, \boldsymbol{x}_j, \boldsymbol{u}_j, \tau), \quad j \in \mathrm{N}_i^{\leftarrow},\ i \in \mathcal{V} \tag{3.1e}$$

$$\boldsymbol{0} \geq \boldsymbol{h}_i(\boldsymbol{x}_i, \boldsymbol{u}_i, \tau), \quad i \in \mathcal{V} \tag{3.1f}$$

$$\boldsymbol{0} \geq \boldsymbol{h}_{ij}(\boldsymbol{x}_i, \boldsymbol{u}_i, \boldsymbol{x}_j, \boldsymbol{u}_j, \tau), \quad j \in \mathrm{N}_i^{\leftarrow},\ i \in \mathcal{V} \tag{3.1g}$$

$$\boldsymbol{u}_i \in [\boldsymbol{u}_{i,\min},\ \boldsymbol{u}_{i,\max}], \quad i \in \mathcal{V} \tag{3.1h}$$

with

$$J_i(\boldsymbol{u}_i; \boldsymbol{x}_{i,0}) = V_i(\boldsymbol{x}_i(T), T) + \int_0^T l_i(\boldsymbol{x}_i, \boldsymbol{u}_i, \tau)\ \mathrm{d}\tau, \tag{3.2}$$

states $\boldsymbol{x}_i(\tau) \in \mathbb{R}^{n_{x,i}}$ and controls $\boldsymbol{u}_i(\tau) \in \mathbb{R}^{n_{u,i}}$. The cost functional of each agent consists of the terminal cost V_i and the cost function l_i, while the overall cost functional is given by the sum of all local cost functionals. The dynamics are given in neighbor affine form. This means, that they are composed of the function $\boldsymbol{f}_i$ and a sum of functions $\boldsymbol{f}_{ij}$. The function $\boldsymbol{f}_i$ only depend on states and controls of the agent while the functions $\boldsymbol{f}_{ij}$ depend on state and controls of the agent and one neighbor. The constraints are formulated analogously. The constraints $\boldsymbol{h}_i$ and $\boldsymbol{g}_i$ depend on states and controls of the agent while the constraints $\boldsymbol{h}_{ij}$ and $\boldsymbol{g}_{ij}$ depend on states and controls of the agent and one neighbor.

4 How to implement a new simulation

Implementing a new simulation example is done in two steps. At first, the new problem description has to be implemented using the problem description from GRAMPC. In the second step the actual simulation is set up.

4.1 Implementation of the problem description

The description of distributed system is based on coupled agents. Each agent is described by an *agent model* while the coupling between agents is described using *coupling models*. Both are meant to be saved inside the folder *model_description*. Example models are already implemented there. Inside the folder, at first create the .cpp and .hpp for the agent and coupling model. It is recommended to simply copy one of the provided example models and only adapt the files. Both the agent and coupling model have to be uniquely defined by choosing an unique class name. Inside the agent model, implement the number of controls, states and constraints as well as the box constraints on the controls in the constructor. The following notation is used for the function names: dfdu_vec represents $(\partial_{\boldsymbol{u}} \boldsymbol{f})^{\mathrm{T}} \boldsymbol{\lambda}$, hence the partial derivative of the function $\boldsymbol{f} \in \mathbb{R}^{n_x}$ with respect to the controls $\boldsymbol{u} \in \mathbb{R}^{n_u}$, transposed and multiplied with a vector of Lagrangian multipliers $\boldsymbol{\lambda} \in \mathbb{R}^{n_x}$. See the documentation of GRAMPC for more detailed information. The following functions have to be implemented obligatory:

- ffct:
  Describes the dynamics of the agent.
- dfdx_vec:
  The partial derivative of the function $\boldsymbol{f}_i$ with respect to the states, multiplied with the Lagrangian multipliers.
- dfdu_vec:
  The partial derivative of the function $\boldsymbol{f}_i$ with respect to the controls, multiplied with the Lagrangian multipliers.
- lfct:
  Cost function of the agent.
- dldx:
  Partial derivative of the cost function with respect to the states.
- dldu:
  Partial derivative of the cost function with respect to the controls.
- Vfct:
  Terminal cost of the agent.
- dVdx:
  Partial derivative of the terminal cost with respect to the states.

Optional functions can be defined to describe the constraints. Therefore, see the documentation of GRAMPC or have a look into the examples. Note that you can use the provided *model parameters* to generalize the description of your models and the *cost parameters* to define the variables inside the cost functional. These can be set later from the simulation. After implementing a coupling model, at first provide inside the constructor the number of states of both agents i and j that are meant to be coupled as well as the number of their controls. Furthermore, the number of coupling constraints has to be given. The following functions have to be defined

- ffct:
  This function describes the dynamics of the coupling.
- dfdxi_vec:
  The partial derivative of the function $\boldsymbol{f}_{ij}$ with respect to the states of agent i, multiplied with the Lagrangian multipliers.
- dfdui_vec:
  The partial derivative of the function $\boldsymbol{f}_{ij}$ with respect to the controls of agent i, multiplied with the Lagrangian multipliers.
- dfdxj_vec:
  The partial derivative of the function $\boldsymbol{f}_{ij}$ with respect to the states of neighbor j, multiplied with the Lagrangian multipliers.
- dfduj_vec:
  The partial derivative of the function $\boldsymbol{f}_{ij}$ with respect to the controls of neighbor j, multiplied with the Lagrangian multipliers.

while additional functions describing constraints on the coupling are optionally. After implementing the agent and coupling models, they have to be integrated into the framework. To get the framework to know the source files, they have to be added into the CMakeLists.txt inside the folder model_description by adding the path to the .cpp-files. After the source-files are known in the framework, the model factory has to be able to generate corresponding objects. Therefore, add the models into the general_model_factory.cpp inside the folder model_description. From this point on, the implemented models can be used in each simulation example.

4.2 Implementation of the simulation

The implementation of a simulation can be done from either the C++- or the Python-interface. Both are identical regarding the functionality and usage, while the syntax only differs due to the different languages. Files for the simulation are meant to be saved inside the folder *main*. Therein, example simulations are provided. For a quick start it is recommended to simply copy the folder of a provided simulation example and adapt it.

4.2.1 Using the C++-interface

If the C++-interface is used to implement a simulation in GRAMPC-D, at first create a new .cpp-file. As the framework has to know the file, create a file with name CMakeLists.txt and create a new exectuable with unique name by calling

```
add_executable(new_example new_example.cpp)
target_link_libraries(new_example dmpc)
```

with your filename instead of *new_example*. Inside the .cpp make sure to include the .hpp of the interface

```
#include "dmpc/interface/dmpc_interface.hpp"
```

After creating an object of the interface

```
DmpcInterfacePtr interface(new DmpcInterface());
```

the communication interface has to be initialized. If the central communication interface is meant to be used, simply call

```
interface->initialize_central_communicationInterface();
```

To initialize the local communication interface for a coordinator, call

```
interface->initialize_local_communicationInterface_as_coordinator(port);
```

and to initialize the local communication interface for an agent, at first implement the TCP-address of the coordinator and call

```
auto comm_info_coordinator = interface->communicationInfo();
comm_info_coordinator.ip_ = ip;
comm_info_coordinator.port_ = port;
interface->initialize_local_communicationInterface_as_agent(comm_info_coordinator);
```

To set the optimization parameters, create an optimization info and parameterize it

```
auto optimization_info = interface->optimizationInfo();
optimization_info.COMMON_Nhor_ = 21;
optimization_info.COMMON_Thor_ = 1;
optimization_info.COMMON_dt_ = 0.1;
optimization_info.GRAMPC_MaxGradIter_ = 10;
optimization_info.GRAMPC_MaxMultIter_ = 1;
optimization_info.ADMM_maxIterations_ = 5;
```

before registering it at the interface

```
interface->set_optimizationInfo(optimization_info);
```

After initializing a communication interface the agents may be registered. This is done by creating an agent info and implementing corresponding values.

```
auto agent = interface->agentInfo();
agent.model_name_ = "water_tank_agentModel";
agent.id_ = agent_id;
agent.model_parameters_ = { A, 1, 0 };
agent.cost_parameters_ = { 0, 0, R };
```

The specific model is chosen by implementing its name. Here it is the *water_tank_agentModel*. The agent can be registered by calling

```
interface->register_agent(agent, xinit, uinit);
interface->set_desiredAgentState(agent_id, xdes, udes);
```

Registering a coupling between two agents works analogously. At first, the model is generated and parameterized

```
auto coupling_info = interface->couplingInfo();
coupling_info.model_name_ = "water_tank_couplingModel";
coupling_info.model_parameters_ = { A, a };
coupling_info.agent_id_ = agent_id;
coupling_info.neighbor_id_ = neighbor_id;
```

and afterwards registered

```
interface->register_coupling(coupling_info);
```

Running a simulation using MPC is done by calling

```
interface->run_MPC(t0, T_sim);
```

while a distributed controller is used after calling

```
interface->run_DMPC(t0, Tsim);
```

Note that the same problem description can be used for both a centralized and a distributed controller.

4.2.2 Using the Python-interface

The usage of the python interface follows the same logic and nearly the same syntax as using the C++-interface. At first, import the corresponding module

```
import grampcd_interface
```

and create an interface

```
interface = grampcd_interface.interface()
```

Set the optimization parameters by creating and parameterizing an optimization info

```
optimization_info = grampcd_interface.OptimizationInfo()
optimization_info.COMMON_Nhor_ = 21
optimization_info.COMMON_Thor_ = 1
optimization_info.COMMON_dt_ = 0.1
optimization_info.GRAMPC_MaxGradIter_ = 10
optimization_info.GRAMPC_MaxMultIter_ = 1
optimization_info.ADMM_maxIterations_ = 5
interface.set_optimizationInfo(optimization_info)
```

Now the communication interface has to be initialized by either choosing a centralized communication interface

```
interface.initialize_central_communicationInterface()
```

or a distributed communication interface. Thereby, the interface for the coordinator is initialized by calling

```
interface.initialize_local_communicationInterface_as_coordinator(port)
```

and the one for the agents by first generating the required TCP-address of the coordinator

```
communicationInfo = grampcd_interface.CommunicationInfo()
communicationInfo.ip_ = ip
communicationInfo.port_ = port
```

and subsequent initializing it

```
interface.initialize_local_communicationInterface_as_agent(communicationInfo)
```

An agent is described by creating and parameterizing an agent info

```
agent = grampcd_interface.AgentInfo()
agent.id_ = agent_id
agent.model_name_ = "water_tank_agentModel"
agent.model_parameters_ = [A, 1, 0]
agent.cost_parameters_ = [0, 0, R]
```

and registering it at the interface

```
interface.register_agent(agent, xinit, uinit)
interface.set_desiredAgentState(agent_id, xdes, udes)
```

Couplings between agents are registered analogously by creating and parameterizing a coupling info

```
coupling_info = grampcd_interface.CouplingInfo()
coupling_info.model_name_ = 'water_tank_couplingModel'
coupling_info.model_parameters_ = [A, a]
coupling_info.agent_id_ = agent_id
coupling_info.neighbor_id_ = neighbor_id
```

and registering it

```
interface.register_coupling(coupling_info)
```

The simulation is run by either calling

```
interface.run_MPC(t0, Tsim)
```

or

```
interface.run_DMPC(t0, Tsim)
```

5 How to approach a numerical solution

5.1 Quickstart

After implementing a distributed optimal control problem it is recommended to follow these steps.

1. Solve the global OCP using a centralized controller to find an adequate set for the basic parameters such as the length of the horizon, the number of discretization points and potentially some GRAMPC-specific parameters.

2. Set the flag ADMM_debugCost_ in the optimization info to 1 and the simulation time shorter than the time step to only simulate a single time step. Now tune the parameters regarding the ADMM algorithm to achieve a required convergence behavior. Therefore, evaluate the trajectory *debugCost_* inside each solution that contains the cost in each ADMM iteration. Note that the basic set of parameters found in Step 1 are only a first guess and not necessarily optimal parameters for the local problems as well. The convergence behavior can be improved using Gauss-Seidel iterations by setting the option ADMM_innerIterations_ to a value greater 1 or by using neighbor approximation.

3. Find a value for ADMM_ConvergenceTolerance_ that stops the ADMM algorithm from iterating at a desired point of convergence.

4. Reset ADMM_debugCost_, increase the simulation time and check the simulation results. If they're not satisfying, restart from Step 2.

5.2 Which parameters can be used for tuning?

In this section the main parameters for tuning are explained. For each parameter an actual value is given to start with. Handle these numbers with care, as an actual good value depends strongly on the implemented system.

5.2.1 Parameters named as COMMON

COMMON_Thor_
Choose a horizon that is large enough to capture the whole dynamics of your system. For example, if the resulting cost is falling at first and then rising again even though the algorithm converges, increase the horizon. If there are oscillations in the cost with a period longer than your horizon, increase the horizon to a value longer than the period of the oscillation. On the same hand be careful with long horizons, as they may require a large number of discretization steps.
Starting parameter: 1

COMMON_dt_
Choose a time step that is small enough to control your system but large enough to be able to solve the OCPs in real time.
Starting parameter: 0.01

COMMON_Nhor_
This value defines the number of points for the discretization of the horizon. There is a minimum value of discretization points for integrating the ODEs with a sufficient small error. Find this point and leave the value there. There is no reason to increase it further.
Starting parameter: 21

5.2.2 Parameters named as ADMM

Penalty parameters
As the penalty adaption is working better than a user-defined value for most examples, it is recommended to keep the default values.

ADMM_maxIterations_
If the ADMM algorithm is converging, increasing the number of ADMM iterations improves the simulation result. If you need more than 100 iterations for a satisfying solution, the issue is somewhere else for most systems so it is not recommended to increase the number of ADMM iterations further.
Starting parameter: 20

ADMM_innerIterations_
The inner iterations can be seen as a Gaus-Seidel iteration and can improve the convergence behavior significantly for some examples. Use them carefully, as they increase the computation effort significantly.
Starting parameter: 1

ADMM_ConvergenceTolerance_
Use the convergence criterion to stop the ADMM algorithm early if it reached a desired grade of convergence. Find an adequate value for the tolerance by trial-and-error.
Starting parameter: 0.01

5.2.3 Parameters named as GRAMPC

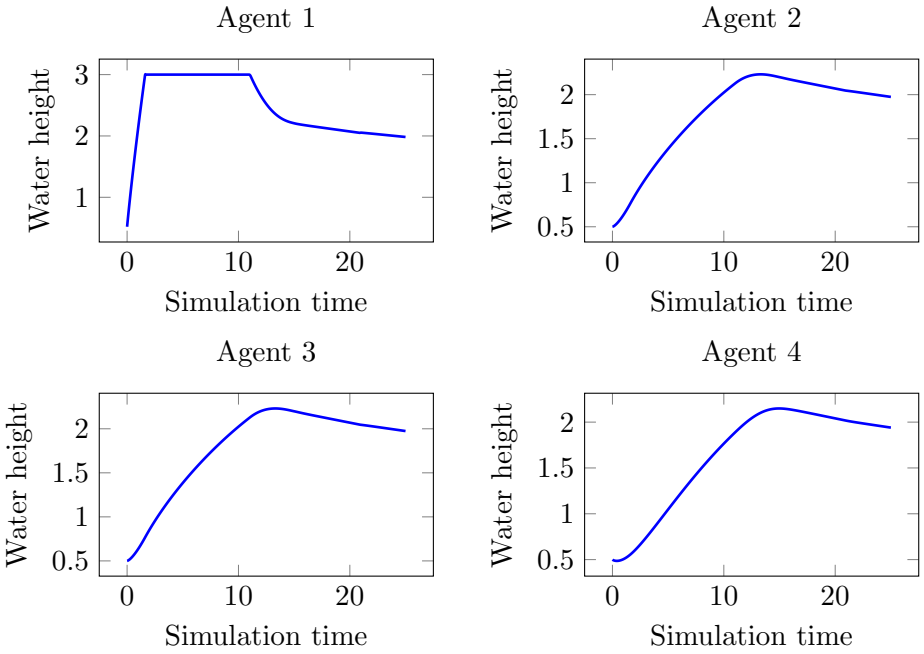
The most important parameters of GRAMPC are shortly explained in the following. Please have a look into its documentation for detailed information.

GRAMPC_MaxGradIter_
This value defines the number of gradient iterations executed in the gradient algorithm. More iterations lead to a more precise local solution that can decrease the number of required ADMM iterations which is often faster than executing a low number of gradient iterations and a high number of ADMM iterations. However, executing more than 50 gradient iterations does not lead a significant improvement in precision anymore. If you would need more, often the problem is somewhere else.
Starting parameter: 15

GRAMPC_MaxMultIter_
This value defines the number of multipliers iterations in the gradient algorithm. Be careful with a large number as the computation effort of GRAMPC is given approximately by multiplier iterations times gradient iterations.
Starting parameter: 3

5.3 Example

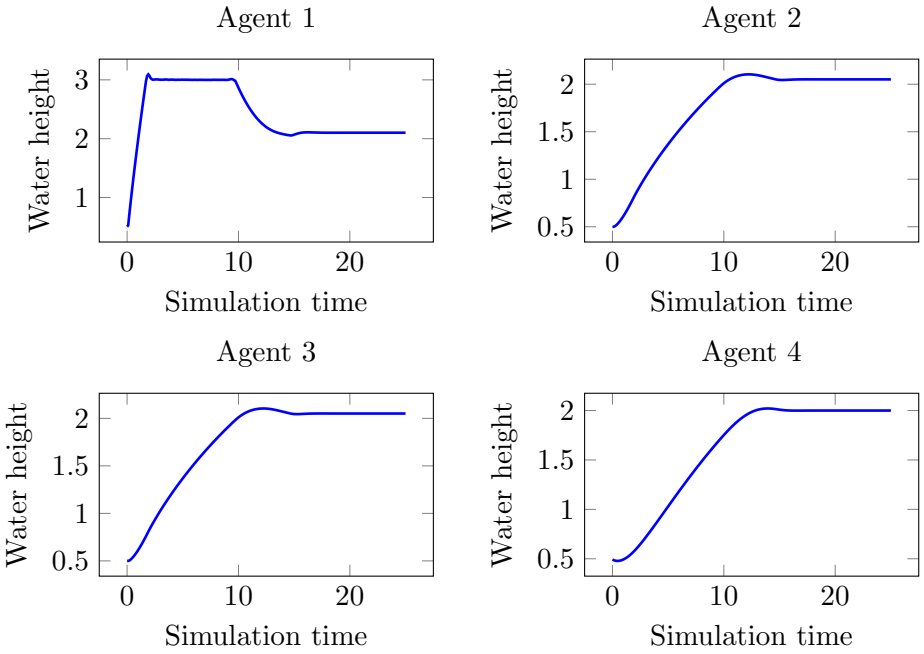
In this section an example is setup using the tuning routing from above for the provided simulation example *coupled_watertanks*. The task is to reach a height of 2 m for the fourth water tank by not exceeding 3 m anywhere.



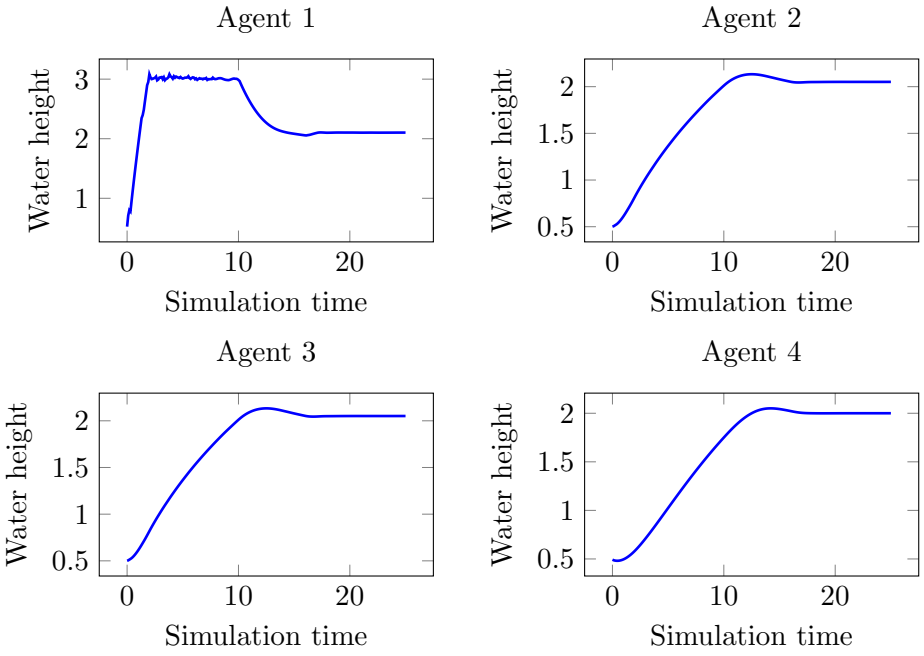
It can be seen that using a central controller with standard parametrization the fourth tank does overshoot the 2 m and drops below it afterwards. The average computation time is 2.05 ms. After changing the parameters

```
optimization_info.COMMON_Thor_ = 5
optimization_info.COMMON_dt_ = 0.1
optimization_info.GRAMPC_MaxGradIter_ = 7
optimization_info.GRAMPC_MaxMultIter_ = 1
```

the fourth agent reaches the desired water height while the computation time is reduced to 0.38 ms.



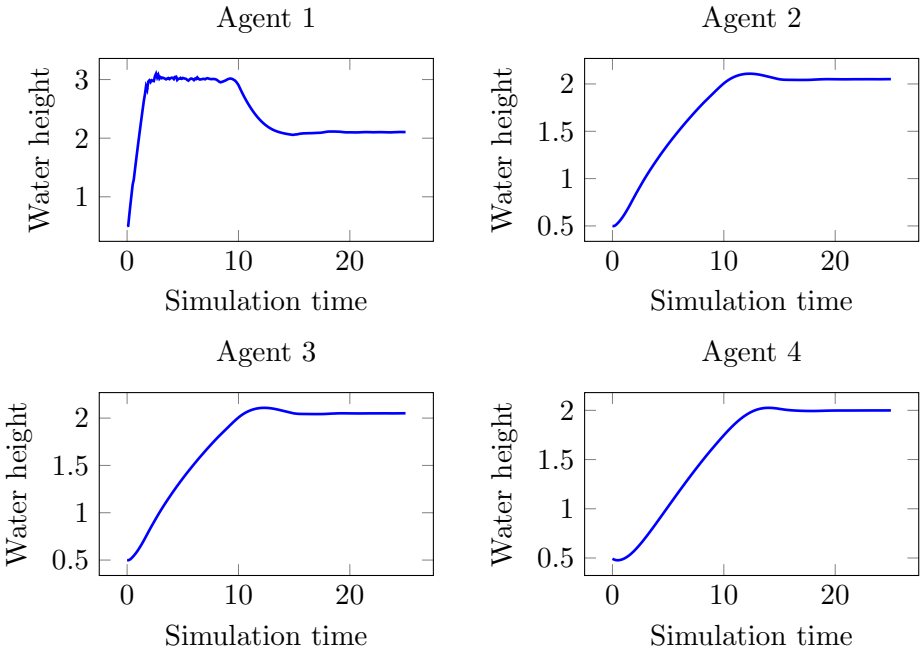
Using standard parameterization for the distributed controller, the algorithm is directly converging with a computation time of 2.16 ms in average and 8.25 ms at maximum per agent.



To increase the performance of the distributed controller by changing the parameters

```
optimization_info.ADMM_maxIterations_ = 10
optimization_info.ADMM_ConvergenceTolerance_ = 0.02.
```

leads to the computation time of 1 ms in average and 4.5 ms at maximum per agent while the desired water height is still reached.

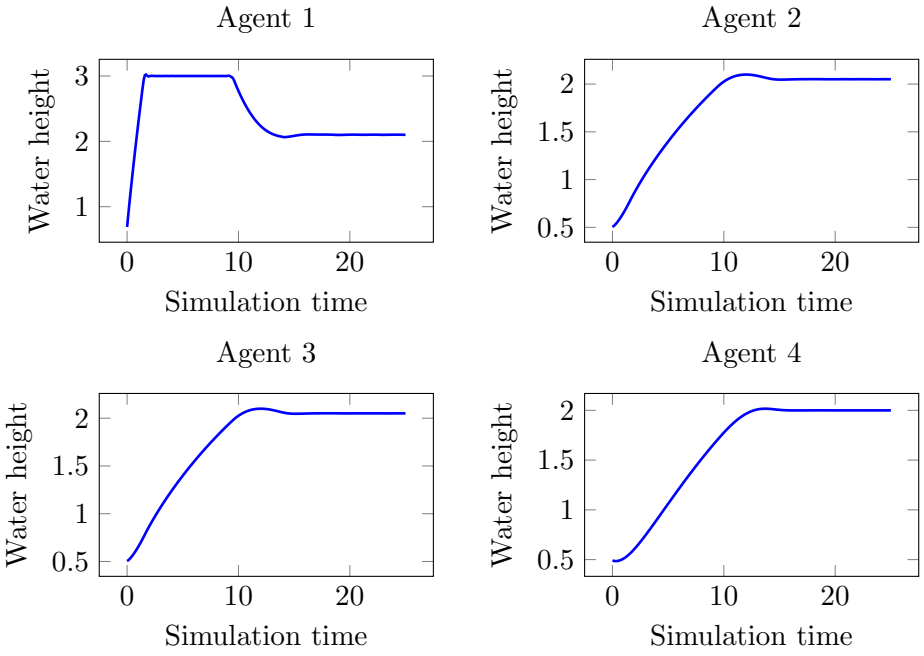


If the solution should be more precise, define

```
optimization_info.GRAMPC_MaxGradIter_ = 10
optimization_info.GRAMPC_MaxMultIter_ = 2
optimization_info.ADMM_maxIterations_ = 10
optimization_info.ADMM_ConvergenceTolerance_ = 0.02.
optimization_info.APPROX_ApproximateCost_ = 1
```

```
optimization_info.APPROX_ApproximateConstraints_ = 1
optimization_info.APPROX_ApproximateDynamics_ = 1.
```

Now the controller takes 2.42 ms in average and 22.75 ms at maximum per agent but the simulation result is more precise.



6 Parameters and Options

All parameters and options are presented in this chapter.

Interface

name	description	arguments
initialize_central_communicationInterface	Initialize the framework for communication inside a single processor.	
initialize_central_communicationInterface	Initialize the framework for communication inside a single processor using multi-threading.	Integer with number of threads
initialize_local_communicationInterface_as_agent	Initialize the framework for communication over a network. Use this for agents.	Communication info with address of the coordinator
initialize_local_communicationInterface_as_coordinator	Initialize the framework for communication over a network. Use this for the coordinator.	Integer as Port
register_agent	Register an agent.	Agent info
deregister_agent	Deregister an agent.	Agent info
set_desiredAgentState	Set a desired agent state for an agent.	Integer as agent id, vector of doubles for initial state, vector of doubles for initial controls
register_coupling	Register a coupling.	Coupling info.
deregister_coupling	Deregister a coupling.	Coupling info
run_MPC	Solve the global OCP using MPC in an endless mode, in which the system is controlled and simulated endlessly.	
run_MPC	Solve the global OCP using MPC for a specific simulation time.	Double with initial time, double with simulation time.
run_DMPC	Solve the global OCP using DMPC in an endless mode, in which the system is controlled and simulated endlessly.	
run_DMPC	Solve the global OCP using DMPC for a specific simulation time.	Double with initial time, double with simulation time.

set_optimizationInfo	Set the optimization info.	Optimization info
get_optimizationInfo	Get the current optimization info.	
wait_for_connections	Define that the coordinator should wait on a specific number of connected agents and couplings before proceeding.	Double for number of agents, double for number of couplings
send_flag_to_agents	Send an acknowledge flag to agents.	Integer with agent id
send_flag_to_agents	Send an acknowledge flag to agents.	Vector of integers with agent ids
send_flag_to_agents	Send an acknowledge flag to agents.	String containing 'all' for all agents
wait_blocking_s	Wait for some seconds before proceeding.	Integer with number of seconds.
waitFor_flag_ from_coordinator	Define that the agent should wait for a flag from the coordinator before proceeding.	
set_passive	Set the agent or coordinator passive	
get_solution	Get the solution of the optimization problem.	Int with agent id
get_solution	Get the solution of the optimization problem.	String that contains 'all' for all agents.
reset_solution	Reset the solution of an agent.	Int with agent id
reset_solution	Reset the solution of all agents.	String that contains 'all' for all agents.
print_solution_to_file	Print the solution of an agent into a .csv-file.	Int with agent id
print_solution_to_file	Print the solution of an agent into a .csv-file.	Int with agent id, string with prefix for filename
print_solution_to_file	Print the solution of all agents into a .csv-file.	String that contains 'all' for all agents
print_solution_to_file	Print the solution of all agents into a .csv-file.	String that contains 'all' for all agents, string with prefix for filename
simulate_realtime	Wait after each time step to simulate controlling a system in real time	Boolean
cap_stored_data	Cap the number of stored time steps.	Integer with number of time steps
set_print_base	Show messages of type basic.	Boolean

set_print_error	Show messages of type error.	Boolean
set_print_message	Show messages of type message.	Boolean
set_print_warning	Show messages of type warning.	Boolean

Agent state

name	description
i_	Index of the agent
t0_	Current time
t_	Vector with the discretized horizon
u_	Vector including the controls
v_	Vector including the external influence
x_	Vector including the states

Solution

name	description
agentState_	This agent state contains the simulation results.
predicted_agentState_	This agent state contains the predicted trajectories.
cost_	The simulated cost.
predicted_cost_	The predicted cost
debug_cost_	The cost in each ADMM iteration. Is only filled, if [...] is set.

Agent info

name	description
id_	Index of the agent
model_name_	Name of the used model
model_parameters_	Vector with model parameters
cost_parameters_	Vector with cost parameters

Coupling info

name	description
agent_id_	Index of the agent
neighbor_id_	Index of the neighbor
model_name_	Name of the used model
model_parameters_	Vector with model parameters

Optimization info

name	description	default
COMMON_Thor_	Length of horizon	1
COMMON_dt_	Time step	0.05
COMMON_Nhor_	Number of discretization points	21
COMMON_ShiftControl_	Shift the control at new iteration	true
COMMON_Integrator_	Used integration method	heun
GRAMPC_MaxGradIter_	See documentation of GRAMPC	
GRAMPC_MaxMultIter_	See documentation of GRAMPC	
GRAMPC_PenaltyMin_	See documentation of GRAMPC	
GRAMPC_PenaltyMax_	See documentation of GRAMPC	
GRAMPC_AugLagUpdateGradientRelTol_	See documentation of GRAMPC	
GRAMPC_Integrator_	See documentation of GRAMPC	
GRAMPC_LineSearchType_	See documentation of GRAMPC	
GRAMPC_PenaltyIncreaseFactor_	See documentation of GRAMPC	
GRAMPC_PenaltyDecreaseFactor_	See documentation of GRAMPC	
GRAMPC_LineSearchMax_	See documentation of GRAMPC	
GRAMPC_LineSearchMin_	See documentation of GRAMPC	
GRAMPC_ConvergenceCheck_	See documentation of GRAMPC	
GRAMPC_ConvergenceGradientRelTol_	See documentation of GRAMPC	
GRAMPC_ConstraintsAbsTol_	See documentation of GRAMPC	
GRAMPC_PenaltyIncreaseThreshold_	See documentation of GRAMPC	
GRAMPC_LineSearchInit_	See documentation of GRAMPC	
ADMM_maxIterations_	Number of ADMM iterations	20
ADMM_innerIterations_	Number of Gauß-Seidel iterations	1
ADMM_ConvergenceTolerance_	Convergence tolerance	0.02
ADMM_PenaltyIncreaseFactor_	Factor to increase penalty parameter	1.5
ADMM_PenaltyDecreaseFactor_	Factor to decrease penalty parameter	0.75
ADMM_PenaltyMin_	Minimum value for penalty parameters	1e-4
ADMM_PenaltyMax_	Maximum value for penalty parameters	1e4
ADMM_PenaltyInit_	Initial value for penalty parameters	1
ADMM_AdaptPenaltyParameter_	Adapt the penalty parameters	true
ADMM_DebugCost_	Safe cost in each ADMM iteration	false
APPROX_ApproximateCost_	Approximate neighbors cost functional	false
APPROX_ApproximateConstraints_	Approximate neighbors constraints	false
APPROX_ApproximateDynamics_	Approximate neighbors dynamics	false

Bibliography

[1] Daniel Burk, Andreas Völz, and Knut Graichen. A Modular Framework for Distributed Model Predictive Control of Nonlinear Time-Continuous Systems. *unpublished*.

[2] Tobias Englert, Andreas Völz, Felix Mesmer, Sönke Rhein, and Knut Graichen. A software framework for embedded nonlinear model predictive control using a gradient-based augmented Lagrangian approach (GRAMPC). *Optimization and Engineering*, 20(3):769–809, September 2019.

[3] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. pybind11 – Seamless operability between C++11 and Python, 2017. https://github.com/pybind/pybind11.