PARODIS: One MPC framework to control them all. Almost.

Thomas Schmitt[1], Jens Engel[1,2], Matthias Hoffmann[1,3], Tobias Rodemann[2]

Abstract— **We introduce the MATLAB framework PARODIS, the Pareto optimal Model Predictive Control framework for distributed Systems. It is a general-purpose, flexible and easy-to-use framework for discrete state space models. Special features are the support of distributed (hierarchical) systems, scenario-based optimization and built-in methods for determination of the Pareto front and selection of a solution. It uses the popular MATLAB framework YALMIP for the symbolic formulation of optimization problems and models.**

I. INTRODUCTION

Model Predictive Control (MPC) has seen rapid developments over the last decade, both in theory and applications. To validate new MPC algorithms in extensive simulation studies, researchers face the challenge of implementing both control algorithms and system models. Therefore, they need a framework which is i) general enough to cover their problem class and common control approaches, ii) flexible enough to also allow implementation of new specialized algorithms and iii) easy enough in use to maintain a reasonable time investment.

However, current MPC frameworks have a different use case, are very specific or lack important features to satisfy these requirements. Multiple frameworks have been developed for real-time control, e. g. ACADO [1] or GRAMPC [2]. Usually based on C/C++ and designed to run on embedded hardware, they are very strong in controlling highly dynamical nonlinear systems, even with sampling times in the (sub)millisecond range. While they are very apt to perform this task, they do not generalize well and are not suitable for prototyping more complicated algorithms such as economic or distributed MPC.

Other frameworks are specialized in a specific field of application, e. g. robotics [3] or, commonly, building control [4], [5]. In this case, they usually employ more sophisticated MPC algorithms and might also feature co-simulation with simulation tools such as Modelica or EnergyPlus. However, their use is limited to the respective application. They do not support a more general form of system classes and are not flexible enough for implementation of new control algorithms.

The official MATLAB MPC toolbox [6] is powerful and can be used for more than just tracking MPC, but the implementation complexity rises significantly with deviation from this standard case, e. g. distributed systems. With the popular MATLAB toolbox *MPT3* [7], regulation or tracking MPC for linear, (piecewise) affine and mixed logical dynamic systems can be set up extremely easily. However, despite some impressive features like automatic conversion to explicit MPC, it does not support economic MPC in general and is not designed for implementation of more complex algorithms, i. e. not flexible. *MPCTools* [8] is an Octave interface to the popular optimization framework *CasADi* [9] with means to simplify the definition of nonlinear MPC systems. While there are no significant explicit limitations due to the use of CasADi, it lacks general features such as direct support for parametrization with external data or distributed systems. Another framework that makes use of CasADi is *do-mpc* [10], based on Python. It is more capable than MPCTools, as it allows for full parameterization of the system dynamics. It focuses on nonlinear time-continuous systems, but does support discrete system formulations. It has a very modular system structure and offers various features. Overall, flexibility and ease of use are given. However, there are no pre-defined interfaces for distributed systems and usage of large data sources.

Here, we present an alternative MATLAB framework, called PARODIS – **Par**eto **o**ptimal MPC framework for **di**stributed **s**ystems. It provides generality to a great extent while maintaining flexibility and easy usage and distinguishes it from other MPC frameworks by i) supporting distributed (hierarchical) systems, ii) providing convenient interfaces to use large-scale datasets for predicted and actual disturbances, iii) respecting scenarios for uncertain disturbances and parameters in a customizable fashion and iv) as its main innovation, the integrated support for multiple objectives by automated generation of the Pareto front and selection of solutions. Note that the Pareto optimization functionality is an optional feature. The complete functionality and its concept, showing its wide generality while maintaining flexibility, is explored in detail in Section II. The basic usage is explained in Section III and an exemplary case study is given in Section IV. We end with a conclusion in Section V.

II. FUNCTIONALITY AND CONCEPT

A. Functionality

PARODIS allows fully parameterized (and thus time-variant) descriptions of discrete state space systems in the form

$$x(k+1) = f(x(k), u(k), d(k), k), \qquad (1)$$

[1]Control Methods & Robotics Lab, Technical University of Darmstadt, Darmstadt, Germany. E-mail: thomas.schmitt@rmr.tu-darmstadt.de
Thomas Schmitt acknowledges the financial support from the Honda Research Institute Europe.
[2]Honda Research Institute Europe GmbH, Offenbach, Germany. E-mail: {jens.engel, tobias.rodemann}@honda-ri.de
[3]Systems Modeling and Simulation, Systems Engineering, Saarland University, Germany. E-mail: matthias.hoffmann@uni-saarland.de. Work was done while author was at the Technical University of Darmstadt.

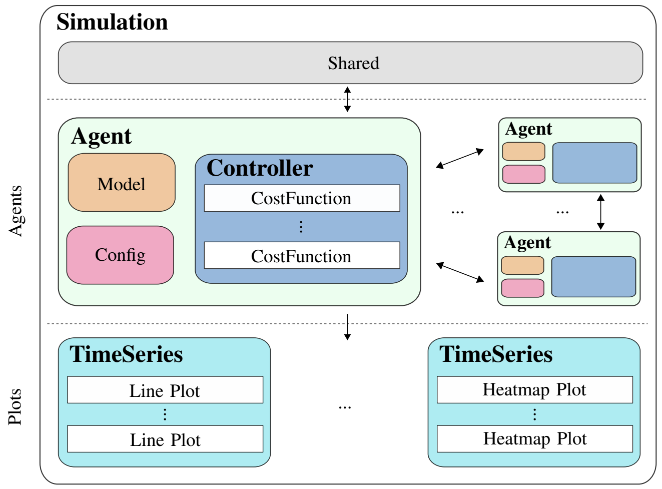

Fig. 1: Structure of a simulation setup in PARODIS. Boxes with a bold title represent an instance of the respective class. An agent fully defines a system and consists of a model, a `Controller` instance and some optional configuration. Every `TimeSeries` instance creates one figure with an arbitrary number of subplots and may contain elements of any agent. In the simulation's *shared* struct, agents can store information of any kind. It is cleared after every iteration.

where x, u and d are the state, input and disturbance vectors, respectively. Multiple systems, in PARODIS called *agents*, can be defined and controlled in a distributed (or hierarchical) fashion. Model predictive controllers are defined for every agent by default, whereby the cost functions can be defined arbitrarily, i. e. as in economic MPC. However, controllers could easily be exchanged. There are two specialties of PARODIS. First, it distinguishes not only between predicted and real disturbances d, but it also supports the consideration of *multiple scenarios* in the prediction of disturbances and parameters, thus advocating it for the use of scenario-based stochastic MPC. As sources, either function handles or `csv`-files can be defined, thus allowing for easy import of large data sets for long-term simulations. In contrast to do-mpc [10], the scenarios can be chosen freely (not only as a full scenario-tree of all possible combinations of distinct values). Furthermore, the incorporation of scenarios in the optimization problem is customizable. Second, it ships with algorithms for multi-objective optimization, i. e. both methods for determining the Pareto front for multiple objectives and methods for automatic selection of a solution, which is repeated at every time step. PARODIS utilizes YALMIP [11] and its symbolic variables to formulate the optimization problem, which makes the large selection of solvers supported by YALMIP accessible.[1]

B. Concept

PARODIS is object-oriented. All necessary information is wrapped in an instance of the `Simulation`-class, which may contain multiple instances of the `Agent`- and `TimeSeries`-classes (figures), as illustrated in Figure 1.

A simplified flow chart diagram is shown in Figure 2. In

[1]See `https://yalmip.github.io/allsolvers/` for an overview.

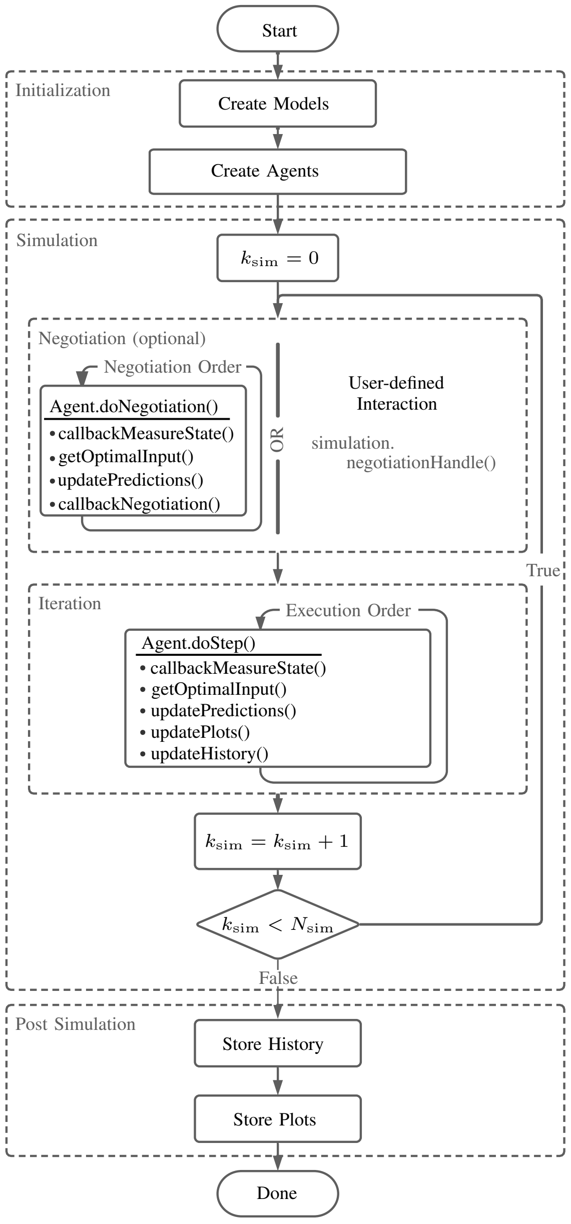

Fig. 2: PARODIS starts with the initialization of all agents. In the actual simulation, every step of the for-loop consists of an optional negotiation between the agents and the actual iteration, where every agent executes its step. The results are stored at the end.

distributed or hierarchical MPC approaches, agents usually interact with each other and *negotiate* what actions they should take in the current time step. In PARODIS, this interaction is called the *negotiation* and is separate from the agents' actual step. There are two options for this negotiation: The first one is for the agents to execute their *negotiation step* in a fixed order. Here, their optimal control problem is solved as usual and a user-defined callback is called afterwards. Using this callback, agents can exchange information freely. The second option is a user-defined interaction using `simulation.negotiationHandle`: This is a user-defined function, where any arbitrary algorithm can be implemented. In this function, all agents, controllers and simulation data can be accessed. After the negotiation, the actual simulation step is executed.

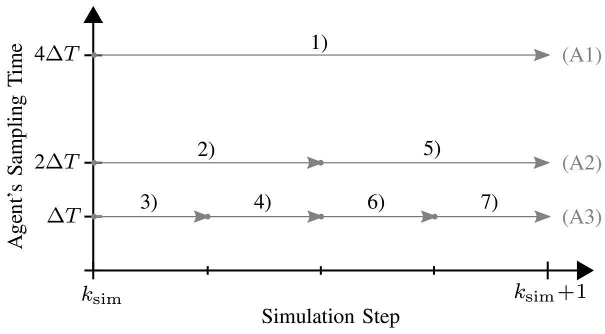

Fig. 3: Exemplary iteration for a simulation with 3 agents and sampling times $4\Delta T$ (A1), $2\Delta T$ (A2), and ΔT (A3). Slower agents step first. Thus, the execution order would be `[A1, A2, A3, A3, A2, A3, A3]`.

Note that multiple optional callback-functions exist and enable the user to adjust the standard simulation flow, e. g. `agent.callbackMeasureState()` to update an agent's state before optimization.

PARODIS supports different agent sampling times, as long as the first steps of the agents' horizons are multiples of each other. Thus, an execution order is determined at the beginning. Figure 3 illustrates this. Within the iteration of a simulation step, every agent's `doStep()`-function is called following this execution order. Thereby, all disturbances and parameters are set. Then, the optimal trajectory is determined, plots are updated and the first input is applied.

In PARODIS, the `Controller`-class handles the formulation and solving of the optimization problem. There are currently three different controller classes available in PARODIS, the `SymbolicController`, the `ExplicitController` and the `ParetoController`. All controllers are easily interchangeable, as they all provide the same interface.

Both the symbolic and explicit controller formulate a standard optimization problem, where the objective function is the (weighted) sum of the configured cost functions. The difference between the two controllers is the representation of the optimization problem: The symbolic controller pre-compiles a fully parameterized optimization problem using YALMIP's `optimizer` feature, while the explicit controller re-builds the problem at each time step and replaces all parameters with explicit values. The `SymbolicController` is to be considered the default controller, as it is very efficient for smaller problems, while the `ExplicitController` is suited for larger problems and debugging purposes as it scales much better and is more transparent in its model representation.

C. Pareto Optimization

Pareto optimization refers to the optimization of multiple (competing) objectives. Usually, the set of Pareto-optimal (or non-dominated) solutions is derived. A solution is non-dominated, if there exists no other solution which is better in objective i without being worse in at least one other objective j [12]. This set is also called *Pareto front*, from which a compromise is then selected. The `ParetoController` provides such functionality. In the same way as any other controller, it derives an optimal input to be applied to the system. It does so in three steps. First, an *extreme point function* calculates the extreme points, i. e. the Pareto optimal points minimizing single objectives. Second, a *front determination scheme* systematically samples points on the Pareto front, using YALMIP optimizer objects for increased efficiency. Finally, a Pareto optimal solution with its corresponding trajectories is selected. While *metric functions* select a solution automatically, the implemented interactivity tool enables the user to instead select a point manually (for $n_{\text{obj}} = 2$ or 3 objectives). For each category, PARODIS provides a set of predefined functions, which can be selected within the `ParetoController`'s configuration. Due to the modular implementation, users can alternatively define their own methods. Table I summarizes the functions currently implemented.

III. Usage

In this section, we give a brief explanation of how problems are implemented in PARODIS. For a detailed explanation, the user is referred to the documentation.

A. Installation

Installing and setting up PARODIS is very straightforward. Simply download the latest release from the GitHub repository,[2] unpack the archive and add the directory to the MATLAB path. As PARODIS is built on YALMIP, it has to be installed separately in the same way. PARODIS itself does not depend on any MATLAB toolboxes.

B. Creating Agents

The agent is the core of PARODIS. It represents a system and fully determines its optimal control problem. To create an agent, a model and a controller need to be defined first. For the model, the user has to implement a function which receives the sampling time T_s and returns the corresponding ordinary difference equations, the number of states n_x, inputs n_u, and disturbances n_d.

```
function [ode, n_x, n_u, n_d]= model_fun(T_s)
  ode = @(x, u, d)( ...
    [ T_s*( x(1)^2 + u(1) + d(1) ); ...
      T_s*( sin( x(2) ) + u(2) + d(2) ) ] );
  n_x = 2;  n_u = 2;  n_d = 2;
end
```

The model itself is then created using the `createModel()` function with additional information of the time horizon vector `T_hor` and the number of scenarios be considered.

```
numScen = 1;
T_hor = 30*ones(N_pred, 1);
model = createModel(model_fun,T_hor,numScen);
```

For the controller, one of the three available types presented in Section II-B can be used, e. g.

```
controller = SymbolicController(numScen);
```

[2]https://github.com/teamparodis/parodis

TABLE I: List of extreme point functions, front determination schemes and metric functions.

Method Type	Name	Description
Extreme Point Method	Lexicographic Approach	Optimization in lexicographic order, i. e. minimizing a single objective J_i with optimal values for $J_{j\neq i}$ as constraints [12].
	Miniscule Weight Approx.	Approximation of the extreme points with weights $w_i = 1$ and $w_{j\neq i} = 10^{-5}$.
	Normalized Miniscule Weight Approximation	Miniscule Weight Approximation with prior normalization of the objectives to a range of 0 to 1 for a better conditioning of the optimization problem.
Front Determination Scheme	Normal Boundary Intersection	The boundary plane, i. e. the hyperplane connecting all extreme points, is evenly sampled. These plane points are then projected onto the Pareto front in the boundary planes' normal vector direction [12].
	Focus Point Boundary Intersection	*Self-developed.* Similar scalarization method like NBI, but with different plane sampling and projection direction.
	Adaptive Weight Determination Scheme	Geometric interpretation of the weighted sum method. From n_{obj} parents, weights are determined for which the new solution lies between the parents [13].
Metric Function	Closest-to-Utopia-point	Compromise solution: Minimal (normalized) Euclidean distance to the Utopia Point.
	Angle to extreme points	Compromise solution: Minimal n_{obj}-dimensional angle to the extreme points.
	Angle to neighbors	Knee point metric: Minimal n_{obj}-dimensional angle to n_{obj} neighbor points.
	Radius of curvature	Knee point metric (*self-developed*): Minimal (approximated) radius of curvature.

Herein lies one of the strengths of PARODIS: switching to Pareto optimization is as easy as swapping the controller.

```
controller = ParetoController(numScen);
```

With the model and controller defined, the optimization problem can be configured by adding parameters, which can either be scenario-dependent or not (`scenDep`).

```
controller.addParam(...
    name, [rows cols], source, scenDep);
```

The source can be a function handle, a csv-file or static values. Parameters can be used both in the constraints and the cost functions. Adding constraints to the optimization problem is just as simple. Box constraints, constraints on differences $\Delta x = x(k+1) - x(k)$ or Δu, as well as free YALMIP constraint expressions are supported.

```
controller.addBoxConstraint('x', lb1, ub1);
controller.addDeltaConstraint('du', lb2, ub2);
controller.addConstraint(expression)
```

In PARODIS, cost functions are user-defined classes, which define a cost expression and may define additional slack variables and constraints. This is very useful for, e. g., reformulations or constraint relaxations.

```
classdef myCostFcn < CostFunction
    [slacks] = getSlacks( ... )
    [constraints] = getConstraints( ... )
    [expr] = buildExpression( ... )
    [horizon] = evaluateHorizon( ... )
end
```

They have to be assigned to the controller, too.

```
controller.addCostFunction( ...
    'costsName1', myCostFcn1);
```

For possible disturbances $d(k)$ on the system, sources for both predictions and the real values can be defined. Again, these can be either function handles, csv-files or static values.

```
controller.predDisturbanceSource=@pred_d;
controller.realDisturbanceSource='real_d.csv';
```

Finally, the agent instance can be created with its initial state.

```
agent1=Agent(name,model,controller,T_hor,x0);
```

Another feature of PARODIS are so-called eval-functions, with which arbitrary values can be calculated during the simulation. These are user-defined functions that can evaluate the simulation during runtime. They are added to an agent, since they may evaluate both its status (predictions at the current time step) and history (past trajectories).

```
agent1.addEvalFunction(name,fcnHandle,scenDep)
```

C. Plotting

Plotting is handled by the `TimeSeries` class. Every instance represents a figure and can have multiple rows and columns of subplots.

```
fig1 = TimeSeries(name, rows, cols)
```

To each subplot, either line plots or heatmap plots can be added, which may display information from different agents.

```
fig1.addLine(agent, var, index, labels, ...);
fig1.addHeatmap(...);
```

The first argument defines from which agent data should be displayed. Using `var`, the user can plot any states, inputs, disturbances, eval- or cost functions. All plots can be configured to be plotted live during the simulation and/or at the end. This can be defined either individually for each figure, or globally for all figures.

D. Running a Simulation

With agents and plots defined, creating and running a simulation is very straightforward. Note that agents and figures have to be added to the simulation instance.

```
agent1 = createAgent(...);
fig1 = TimeSeries(...);
...
simulationTime = 24*60;
sim = Simulation(name, simulationTime);
sim.config.livePlot = true;
sim.addAgent(agent1);
sim.addPlot(fig1);
sim.runSimulation();
```

IV. Exemplary Hierarchical System

To illustrate the most important features of PARODIS, we present the model of an energy management system of an office building, which is controlled using a hierarchical MPC approach.

A. Higher Level

The main model, in the following called the *higher level* (HL), contains the energy E of a stationary battery and the building's overall temperature ϑ_b as states. The system's inputs are the electrical power obtained from the power grid P_grid, the electrical power P_chp from the building's combined heat and power plant (CHP), the thermal power from a gas heating unit $\dot{Q}_\mathrm{rad}$ and the thermal cooling power from a heating, ventilation and air conditioning system $\dot{Q}_\mathrm{cool}$. As uncontrollable disturbances acting on the system, the electrical power from a photo-voltaic plant P_ren, the electrical power demand P_dem and the air temperature ϑ_air are considered. With these, the system's dynamic is given by

$$E(k+1) = E(k) + T_\mathrm{s} \cdot \left(P_\mathrm{grid}(k) + P_\mathrm{chp}(k) + \frac{\dot{Q}_\mathrm{cool}(k)}{\varepsilon_\mathrm{c}} \right) \\ \ldots + T_\mathrm{s} \cdot \left(P_\mathrm{ren}(k) + P_\mathrm{dem}(k) \right), \tag{2}$$

$$\vartheta_\mathrm{b}(k+1) = -\,\mathrm{e}^{\frac{-H_\mathrm{air,b}}{C_\mathrm{th,b}}} \vartheta_\mathrm{b}(k) + \ldots \\ \mu \cdot \left(\frac{P_\mathrm{chp}(k)}{c_\mathrm{CHP}} + \dot{Q}_\mathrm{rad}(k) + \dot{Q}_\mathrm{cool}(k) + H_\mathrm{air,b} \cdot \vartheta_\mathrm{air}(k) \right) \tag{3}$$

with $\mu = \frac{1-\mathrm{e}^{-\frac{H_\mathrm{air}}{C_\mathrm{th}} T_\mathrm{s}}}{H_\mathrm{air}}$. T_s is the sampling time, $C_\mathrm{th,b}$ the building's total thermal capacity, $H_\mathrm{air,b}$ the total heat transfer coefficient, ε_c the energy efficiency ratio of the cooling machine and c_CHP the CHP's current constant. For the higher level, Pareto optimization is applied with the monetary costs and the quadratic temperature deviation from 21 °C as two objectives. For a detailed insight into both modeling and the Pareto approach, the reader is referred to [14], [15].

B. Lower Level

As a second (sub)model, in the following called the *lower level* (LL), the building's thermal energy system is split up into 9 different temperature zones (neglecting the electrical energy system). In the time-continuous case, the temperature $\vartheta_{\mathrm{b},i}$ of zone i is described by

$$\dot{\vartheta}_{\mathrm{b},i}(t) = -\frac{H_{\mathrm{air},i}}{C_{\mathrm{th},i}} \left(\vartheta_{\mathrm{b},i}(t) - \vartheta_\mathrm{air}(t) \right) \ldots \tag{4}$$
$$-\sum_{j \neq i} \frac{\beta_{ij}}{C_{\mathrm{th},i}} \left(\vartheta_{\mathrm{b},i}(t) - \vartheta_{\mathrm{b},j}(t) \right) + \frac{1}{C_{\mathrm{th},i}} \left(\dot{Q}_{\mathrm{heat},i}(t) + \dot{Q}_{\mathrm{cool},i}(t) \right).$$

where $C_{\mathrm{th},i}$ is the thermal capacity from zone i, $H_{\mathrm{air},i}$ is the heat transfer coeffienct between zone i and the outside air, β_{ij} is the heat transfer by coefficient between zones i and j, and $\dot{Q}_{\mathrm{heat},i}$ and $\dot{Q}_{\mathrm{cool},i}$ are the respective heating and cooling powers allocated to zone i. For brevity, we omit stating the (discretized) state space model used for implementation. Note that $\sum_{i=1}^{9} C_{\mathrm{th},i} = C_\mathrm{th,b}$ and $\sum_{i=1}^{9} H_{\mathrm{air},i} = H_\mathrm{air,b}$.

The lower level's task is to distribute the heating and cooling power produced on the higher level between the 9 temperature zones. Thus, its only objective (and cost function) is to minimize the temperature deviation of all zones from the desired 21 °C,

$$l_\mathrm{LL}(k) = \sum_{i=1}^{9} C_{\mathrm{th},i} \left(\vartheta_\mathrm{b,i}(k) - 21 \right)^2. \tag{5}$$

Note that the deviation for every zone is scaled by its capacity to prevent the preferred regulation of smaller zones.

C. Communication

For a simple hierarchy, the negotiation loop is not necessary. The higher level executes first. At the beginning, it updates ϑ_b with an `agent.callbackMeasurement` function, where it reads the states $\vartheta_{\mathrm{b},i}$ of the lower level and updates its ϑ_b.

```
states_LL = ...
  simulation.agents.LL.history.x(:,end);
agent.history.x(2,end) = w_th * states_LL;
```

`w_th` is a weighting vector with elements $w_{\mathrm{th},i} = \frac{C_{\mathrm{th},i}}{C_\mathrm{th,b}}$. Afterwards, the step is performed as usual.

The lower level has to use the total produced heating and cooling powers as constraints for its own input variables, i. e.

$$\sum_{i=1}^{9} \dot{Q}_{\mathrm{heat},i}(k) = \dot{Q}_\mathrm{rad}(k) + \frac{P_\mathrm{chp}(k)}{c_\mathrm{CHP}}, \forall k \in [0, N_\mathrm{pred}-1], \tag{6}$$

$$\sum_{i=1}^{9} \dot{Q}_{\mathrm{cool},i}(k) = \dot{Q}_\mathrm{cool}(k), \qquad \forall k \in [0, N_\mathrm{pred}-1]. \tag{7}$$

This is realized by adding 2 parameterized constraints to the controller in its definition before simulation:

```
LLController.addConstraint( ...
  (sum(LLModel.u(1:9,:),1)==QheatFromHL{1})
);
LLController.addConstraint( ...
  (sum(LLmodel.u(10:18,:),1)==QcoolFromHL{1})
);
```

The parameters' sources are defined as function handles and the values are automatically updated at the beginning of every step, e. g.:

```
QcoolFromHL = ...
 {agents.HL.previousStatus.uPred(4,:)};
```

D. Results

For both systems a prediction horizon of 24 h is used. The first 8 h are split in 15 min steps, the second 8 h in 30 min steps and the last 8 h in 1 h steps, i. e. a total of $N_\mathrm{pred} = 56$ steps. On the higher level, the resulting optimization problem (including, e. g., epigraph reformulations [14]) consists of 338 decision variables and 1066 constraints. The lower level results in 1521 decision variables and 3649 constraints. With GUROBI as the solver, the simulation of 2 days with realistic data from April 2020 (and Pareto optimization on the higher level) takes 247 s on an Intel i7-8550U notebook CPU. Without Pareto optimization, it reduces to 58 s.

Figure 4 shows the resulting trajectories for the higher level, Figure 5 the Pareto fronts for its two objectives (monetary and comfort costs). Figure 6 shows the temperature and corresponding heating power progressions of the lower level agent's zones.

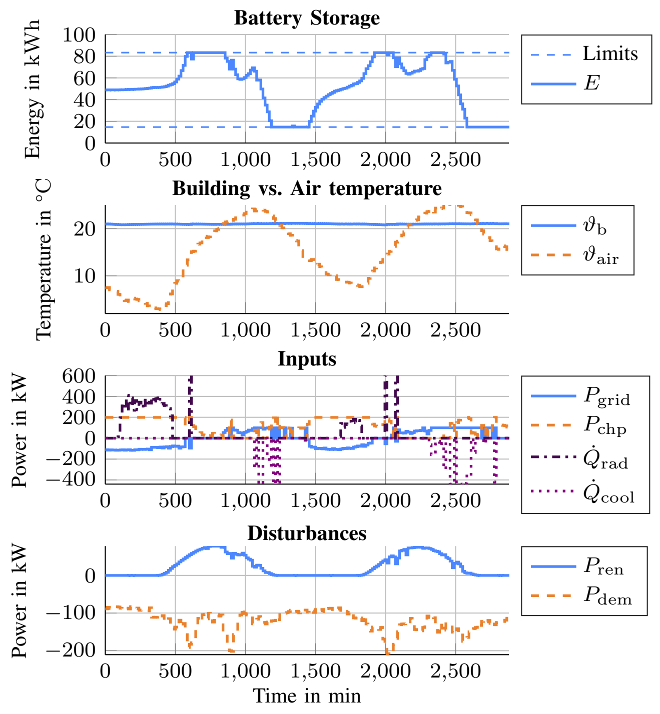

Fig. 4: Trajectories of the higher level agent. ϑ_b is kept closely at the desired 21 °C. For $P_{grid} > 100$ kW, peak costs would apply and are thus avoided. The MPC successfully keeps P_{grid} within this limit by utilizing the stationary battery.

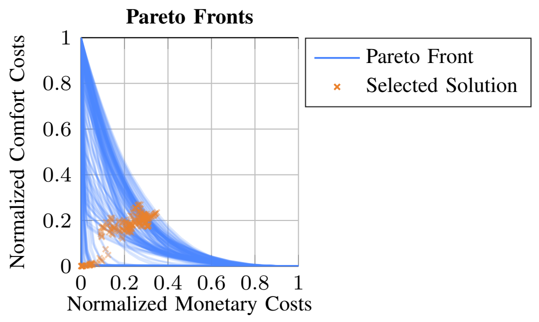

Fig. 5: Generated Pareto Fronts of the higher level agent for every time step in normalized space, i. e. the Utopia point is at the origin and the extreme points are at (0,1) and (1,0). 'AWDS' has been used as front determination scheme, 'CUP' as metric. On average, one Pareto front consists of 22.92 points.

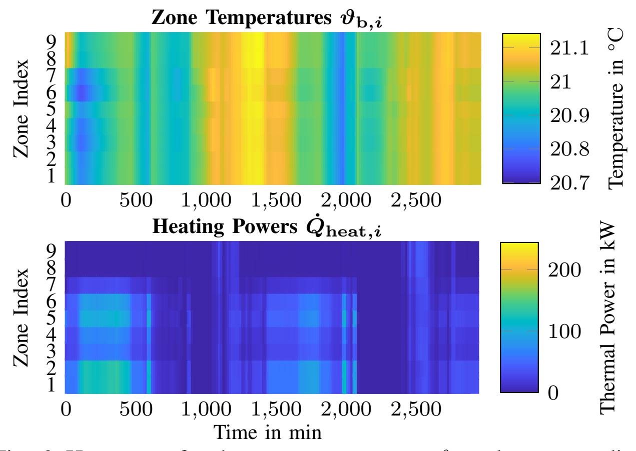

Fig. 6: Heat maps for the zone temperatures ϑ_i and corresponding heating powers $\dot{Q}_{heat,i}$ of the lower level agent. Differences between zones occur from different heat transfer coefficients $H_{air,i}$ and β_{ij}.

V. Conclusion

We present the MATLAB MPC framework PARODIS of which we hope that it will support researchers in both implementation and testing of newly developed MPC algorithms as well as empirical verification of applications by data-based simulation studies. It is the first of its kind with built-in Pareto optimization methods. Even if permanent multi-objective optimization is not intended, they can be used as an automatic setup for finding appropriate weights for multiple objectives. The modular object-oriented structure of PARODIS allows for easy adaption for both new models and control algorithms. Thus, PARODIS is not limited to its present scope.

References

[1] B. Houska, H. J. Ferreau, and M. Diehl, "ACADO toolkit—an open-source framework for automatic control and dynamic optimization," *Optimal Control Applications and Methods*, vol. 32, no. 3, pp. 298–312, 2011.

[2] T. Englert, A. Völz, F. Mesmer, S. Rhein, and K. Graichen, "A software framework for embedded nonlinear model predictive control using a gradient-based augmented lagrangian approach (GRAMPC)," *Optimization and Engineering*, vol. 20, no. 3, pp. 769–809, 2019.

[3] M. Giftthaler, M. Neunert, M. Stäuble, and J. Buchli, "The control toolbox — An open-source C++ library for robotics, optimal and model predictive control," in *2018 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, 2018, pp. 123–129.

[4] D. Blum and M. Wetter, "MPCPy: An open-source software platform for model predictive control in buildings," in *In Proceedings of the 15th Conference of International Building Performance Simulation*, San Francisco, CA, 2017.

[5] M. Baranski, L. Meyer, J. Fütterer, and D. Müller, "Comparative study of neighbor communication approaches for distributed model predictive control in building energy systems," *Energy*, vol. 182, pp. 840 – 851, 2019.

[6] The MathWorks, Inc., *MATLAB and Model Predictive Control Toolbox Release 2020a*, Natick, Massachusetts, United States, 2020.

[7] M. Herceg, M. Kvasnica, C. Jones, and M. Morari, "Multi-parametric toolbox 3.0," in *Proc. of the European Control Conference*, Zürich, Switzerland, Jul. 2013, pp. 502–510, http://control.ee.ethz.ch/~mpt.

[8] M. Risbeck and J. Rawlings, "MPCTools: Nonlinear model predictive control tools for casadi (octave interface)," 2016.

[9] J. A. E. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl, "CasADi – A software framework for nonlinear optimization and optimal control," *Mathematical Programming Computation*, vol. 11, no. 1, pp. 1–36, 2019.

[10] S. Lucia, A. Tătulea-Codrean, C. Schoppmeyer, and S. Engell, "Rapid development of modular and sustainable nonlinear model predictive control solutions," *Control Engineering Practice*, vol. 60, pp. 51 – 62, 2017.

[11] J. Löfberg, "YALMIP : A toolbox for modeling and optimization in MATLAB," in *In Proceedings of the CACSD Conference*, Taipei, Taiwan, 2004.

[12] R. T. Marler and J. S. Arora, "Survey of multi-objective optimization methods for engineering," *Structural and Multidisciplinary Optimization*, vol. 26, no. 6, pp. 369–395, Apr. 2004.

[13] N. Ryu and S. Min, "Multiobjective optimization with an adaptive weight determination scheme using the concept of hyperplane," *International Journal for Numerical Methods in Engineering*, vol. 118, no. 6, pp. 303–319, 2019.

[14] T. Schmitt, T. Rodemann, and J. Adamy, "Multi-objective model predictive control for microgrids," *at - Automatisierungstechnik*, vol. 68, no. 8, pp. 687 – 702, 2020.

[15] T. Schmitt, J. Engel, T. Rodemann, and J. Adamy, "Application of pareto optimization in an economic model predictive controlled microgrid," in *2020 28th Mediterranean Conference on Control and Automation (MED)*. IEEE, 2020, pp. 868–874.