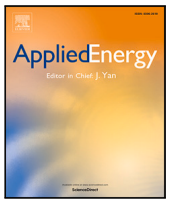
Applied Energy 385 (2025) 125496

Contents lists available at ScienceDirect

Applied Energy

journal homepage: www.elsevier.com/locate/apenergy



A modular Python framework for rapid development of advanced control algorithms for energy systems

Steffen Eser [a,*], Thomas Storek [a,d], Fabian Wüllhorst [a], Stefan Dähling [b], Jan Gall [c], Phillip Stoffel [a], Dirk Müller [a]

[a] RWTH Aachen University, E.ON Energy Research Center, Institute for Energy Efficient Buildings and Indoor Climate, Mathieustr. 10, Aachen, 52074, Germany
[b] RWTH Aachen University, E.ON Energy Research Center, Institute for Automation of Complex Power Systems, Mathieustr. 10, Aachen, 52074, Germany
[c] Robert Bosch GmbH, Corporate Sector Research and Advance Engineering, Robert-Bosch-Campus 1, Renningen, 71272, Germany
[d] Drees & Sommer SE, Habsburgerring 2, Cologne, 50674, Germany

ARTICLE INFO

Dataset link: https://github.com/RWTH-EBC/AgentLib

Keywords:
Multi agent system
Building energy management
Control
IoT

ABSTRACT

Due to the advance in energy engineering and necessary adaptations due to climate change, building energy systems are becoming increasingly complex, necessitating the development of advanced control strategies. However, there is often a gap between control algorithms developed in research and their practical adoption. To bridge this gap, we present *AgentLib* – a modular Python framework to aid the development, testing and deployment of advanced control systems for energy applications. *AgentLib* allows researchers and engineers to gradually scale up controller complexity, supporting the full development lifecycle from simulation and testing to distributed real-time implementation. The framework and its plugins provide a set of extensible modules for common agent functions like optimization, simulation and communication. Control engineers can leverage familiar tools for mathematical optimization and machine learning in Python. *AgentLib* is agnostic to specific communication protocols, allowing flexible interfacing with diverse energy systems and external services. To demonstrate the framework's capabilities, we present a case study on developing a distributed model predictive controller from concept to real-world experiment. We showcase how *AgentLib* enables a true parallel implementation of cooperative agents and supports gradual transition from development to deployment. By analyzing the system's performance, we highlight the real-world impacts of communication overhead on distributed control. The framework's capability to bridge the gap between theoretical research and practical applications marks a significant step forward in deploying sophisticated control strategies within the building energy management sector, and possibly other domains.

1. Introduction

Reducing the primary energy demand and CO_2 emissions are two major goals of today's energy policies to limit global warming. The coupling of the electrical and the thermal energy sector and the integration of renewable energy sources leads to increasingly decentralized, complex energy systems. For example, in modern non-residential buildings, heating and cooling power may be provided by two our more independent systems, e.g. fan coils for each room and thermally activated building systems [1]. Classical control alternatives such as rule based control and PID controllers reach limitations in these systems [2–4]. Improving on these classic control schemes is possible, but requires coordination between different components of the building automation system. For example, variable air volume (VAV) systems can be linked with the supply ventilator to coordinate its speed and power consumption, and room automation systems can implement checks to coordinate their setpoints. One major field of research is model predictive control (MPC) [5], with authors reporting energy savings of about 20% up to 35% while reducing the operating costs up to 73%, when compared to classical control strategies [2–4,6].

Still, usage of advanced control strategies (ACS) in buildings in the industry is quite rare and labor-intensive. To drive practical adoption and accelerate development times, virtual testing of models, control methods, and communication infrastructure is crucial. A workflow for a staged development concept, integrating and testing different aspects of ACS step by step, has been presented by Storek et al. [7].

* Corresponding author.
E-mail address: steffen.eser@eonerc.rwth-aachen.de (S. Eser).
URL: https://www.ebc.eonerc.rwth-aachen.de/cms/~dmzz/E-ON-ERC-EBC/ (S. Eser).

https://doi.org/10.1016/j.apenergy.2025.125496
Received 3 July 2024; Received in revised form 16 January 2025; Accepted 5 February 2025
Available online 16 February 2025

In this work, we present *AgentLib*, a framework with the goal of accelerating the integration of ACS, adopting this staged development concept. Our framework allows developers to devise, test and commission ACS, considering all crucial components that would be required for a commercial product. For researchers, it helps them consider practical hurdles in their work. For the industry, it provides a way to quickly develop prototypes before committing to development of production software on real hardware.

The remainder of this paper is structured as follows: In Section 1.1, we review existing work in the field of ACS for buildings with a focus on MPC, and derive the need for a new framework. We summarize the requirements on such a framework and explicitly state our contribution in Section 1.2. In Section 2, we introduce the framework, including the guiding principles, the architecture, and an overview of ready-to-use components. In Section 3, we present a demonstration of the framework, where we apply an alternating direction method of multipliers (ADMM)-based distributed MPC to a test bed for ventilation control. Then, Section 4 shows how to reuse the developed code for a different application (temperature control). Section 5 discusses how the framework helps solve real world problems and states its limitations. Section 6 provides concluding remarks.

1.1. Related work

The following section covers recent publications in the field of ACS for buildings, tools used for implementation of these methods and difficulties in practical adoption.

1.1.1. Advanced control for buildings

Looking at current research, we want to highlight the diversity in terms of controlled system, and the applied methods and tools used for implementation and communication.

Zanetti et al. [8] compared MPC with different classes of the optimal control problem (quadratic, nonlinear, mixed-integer nonlinear) and different numerical solvers. They test their MPC in a simulation against a Modelica-based residential building from the virtual test bed BOPTEST [9], which is interfaced through a REST-API. They determined a suitable optimization model in MATLAB and implemented the optimal control problem (OCP) in Python using pyomo.

In an experimental study, Kim et al. [10] implemented an MPC for a multi chiller system supplying a university campus. They implement the MPC as a mixed-integer linear program. The MPC received numerous inputs from different sources, like weather forecasts from a public web-API and load predictions from artificial neural network (ANN) based models developed in tensorflow. The MPC was deployed on a server, communicating with the buildings energy management system over the Simple Object Access Protocol (SOAP), utilizing a Python client.

Huber et al. [11] conducted experimental MPC on a residential building, comparing MPC with linear regression and random forest models against MPC with resistance-capacitance models. They accessed forecasts and the buildings sensor readings through an SQL database with a REST API, and connected with the buildings' actuators through an OPC UA client.

Lefebure et al. [12] examined distributed MPC based on dual decomposition for the same building. They implemented their MPC agents in MATLAB using gurobi, using Python to facilitate serial execution of the agents.

Stoffel et al. [13] compared different machine learning models in an experimental study on data-driven MPC. They implemented linear regression and gaussian process regression models with scikit-learn, and ANNs with keras. They access the buildings' sensors and actuators through an API with a commercial cloud platform.

Lin and Adetola [14] used distributed optimization based on ADMM to derive flexibility of the power demand for a simulated multi-zone building with up to 320 zones. Their algorithm is implemented in MATLAB.

In another study on distributed optimal control with ADMM, Li et al. [15,16] consider a multi zone system actuated with VAV units. They implemented their controller in MATLAB, with system simulation performed using TRNSYS.

As shown, the research on building MPC ranges from simulative to experimental, from centralized to distributed, or from physics-based modeling to machine learning. However, all these studies have in common that they need to interface with the simulation or real building in some way, and require access to input data, like weather forecasts. Data driven approaches might require separate steps for data handling, and user driven approaches will require a user interface. While interfacing other components might be trivial for a centralized MPC simulation, the overhead of these tasks increases for decentralized controllers, or experiments, where sensor data and forecasts for weather and energy prices might come from different sources. In addition, we could not find a study on distributed optimization applied to building control, where a parallelizable algorithm was actually implemented in a distributed way. We provide such an example in Section 3 of this work.

1.1.2. Difficulties in practical adoption of advanced controllers for buildings

Despite the recognized importance and research efforts in the field of MPC or other ACS in academia, their practical implementation is quite rare [17]. Ceccolini and Sangi [17] note the lack of awareness of ACS' benefits, the uncertainty in their adaption and the lack of demonstration projects. Schmidt and Åhlund [18] note that existing research on predictive building control usually does not address the challenge of building automation and control system (BACS) integration, and usually relies on simulations for validation. This is understandable, as the task of fully integrating an ACS is complex and time-consuming, as shown in the study by Blum et al. [19]. They implemented an MPC for a real office building, keeping track of the implementation effort and giving an overview of required tasks. In the preparation phase, system analysis, data collection and possibly sensor installation have to be performed. The modeling process includes the thermal envelope and HVAC systems, as well as routines for load and weather forecast. They use MPCPy [20] for controller software development. Finally, deployment consists of setting up the connection to the building management system, functional testing, commissioning, maintenance and performance evaluation. They report an estimated workload of 239 person days for the setup of the MPC, not including development of MPCPy. Blum et al. also gave some concrete examples of practical hurdles they encountered. Challenges included sensor inaccuracies, unexpected communication losses, and the need to incorporate input data from heterogeneous sources, requiring separate handling for all of them. They therefore deduct, that two additional crucial steps towards the real-world application of ACSs are the development of sophisticated system integration patterns, and comprehensive testing of the automation functions during the implementation.

Additionally, Ceccolini and Sangi [17] and Lucia et al. [21] also see the absence of standardized and systematic testing methods that do not require excessive reimplementation or reconfiguration as an additional significant hurdle towards practical adoption of ACS. Finally, inclusion of the underlying communication infrastructure is often neglected when ACS are compared in research [7].

1.1.3. Existing tools for development of advanced control methods in buildings

So far, we established that while practical applications of ACS in building energy systems (BESs) are not widely adopted and labor-intensive, research on the topic is plentiful. To implement ACS, there exist a number of commonly used tools that encompass the fields of detailed simulation and modeling, as well as control oriented modeling, and libraries for optimization and machine learning. Table 1 shows a list of commonly used tools in the space of building simulation, optimization and machine learning.

Table 1
Selection of commonly used tools in research for advanced building control.

Detailed simulation	Optimization modeling	Machine Learning
Buildings (Modelica) [22]	CasADi [23]	keras [24]
AixLib (Modelica) [25]	pyomo [26]	PyTorch [27]
Energyplus [28]	MATLAB	scikit-learn [29]
Spawn of Energyplus [30]	jump [31]	MATLAB
TRNSYS		

The first column contains tools for detailed modeling and simulation. These include open-source Modelica libraries like Buildings [22] and AixLib [25], which excel for modeling of physics and classical controllers. EnergyPlus [28] is an open-source tool especially suitable for detailed simulation of the building envelope. There are also commercial solutions that are commonly used for modeling, like MATLAB SIMULINK, and TRNSYS. One development improving the modularity of different simulation software is the introduction of the functional mockup interface (FMI), which allows standalone export and co-simulation of models. For example, Spawn of EnergyPlus [30] includes an FMU of an EnergyPlus model to allow co-simulation with Modelica.

The second column lists tools for modeling of optimal control problems, and interfacing with numerical solvers. In most of the studies that describe the implementation of their controllers, high level programming languages like MATLAB or Python are used to build optimization models or machine learning models. Open-source optimization tools that interface with these programming languages include pyomo [26] for mathematical modeling of potentially nonlinear systems and interfacing many different numerical solvers in Python, or JUMP [31], which is available for Julia. CasADi [23] is an open-source framework that allows formulation of nonlinear mathematical expressions and interfacing different solvers, providing low level control over OCPs. CasADi requires the user to formulate their own optimization problems. As a result, different libraries have emerged that build on top of CasADi to provide such functionality, including integrated MPC tools like do-mpc [32], mpcpy [20] or DDMPC [13]. Finally, MATLAB provides many tools for optimal control and modeling, including plugins like the model predictive control toolbox.

For machine learning applications, Python seems to be the leading programming language, offering a variety of machine learning libraries like scikit-learn [29], keras [24] or PyTorch [27], and a rich ecosystem of supporting data management libraries. While Python is dominant in machine learning, MATLAB also provides access to plugins like the Deep Learning Toolbox or the Statistics and Machine Learning Toolbox, and the Julia ecosystem offers tools like SciML.

As it can be seen, there are tools readily available for performing detailed building simulations, and developing and testing model based controllers or machine learning based applications on them. The Python programming language in particular offers a rich set of tools spanning all topics relevant to development of ACS. However, the effort to move the developed application to a real building requires significant overhead and repeated reimplementation of interfaces.

1.1.4. Connecting to the building

The reimplementation effort required to move to a real system is greatly augmented by the number of different data sources and protocols for building automation. This is also mentioned in the study by Blum et al. [19] where one source of the large implementation effort was that data from different systems (electrical, HVAC, weather forecast, weather live data etc.) came from different sources, requiring dedicated setup for each including authentication and account management. Reviews on building automation systems are given in [33,34], where different protocols and industry standards are covered. These include technologies on the field level like KNX, BACnet, LONworks, ZigBee or Modbus, and technologies on the management level like OPC UA or again, BACnet. These communication protocols can be implemented using different physical specifications for field buses or local networks, like Ethernet or WIFI. Additionally, there might be a BACS or an IoT middleware through which the physical systems can be interfaced. To be viable for use in a variety of real systems, a control application should be able to interface with a multitude of protocols and communication standards without reimplementing the application itself.

1.1.5. Agent frameworks for modular development

The integration of ACS encompasses a multitude of tasks and requires the coordination of many singular components, programmed with different tools and interfacing with different communication protocols or cloud systems. A framework aiming to encapsulate all these requirements needs to be modular and be executable in a distributed manner (multiple devices or cloud). Multi-agent system (MAS) is a computing paradigm that has been around since the 1990s [35] that is already used in many different domains, including modeling complex systems, smart grids, and computer networks [36]. Developing ACS as a modular multi-agent system (MAS) could reduce reimplementation of components, and improve scalability and fault tolerance. Dorri et al. [36] provide a comprehensive overview of MAS, outlining their characteristics, the methods used for communication between agents, and the challenges in implementing MAS. In the following, we discuss three frameworks for multi-agent systems, JADE, SPADE and cloneMAP.

JADE (Java Agent DEvelopment framework), is the most well-known platform for MAS [37]. It was originally released in 1998, with its first publication stemming from the year 2000 [38]. JADE, and derived from it WADE, account for a large part of the projects surveyed in [36,39,40]. JADE is a feature-rich framework with a long development history, supporting the implementation of agents in JAVA in a distributed environment, supporting programmers with graphical user interfaces. While JADE is still actively used and developed, there are still hurdles that limit the use of JADE in the control of energy systems [37]. Perhaps most importantly, the usage of JADE is restricted to the JAVA programming language.

SPADE (Smart Python Agent Development Environment) aims to provide an agent development environment that is accessible for Python developers [41]. SPADE implements peer-to-peer communication between agents based on the XMPP protocol and handles concurrent execution of agent behaviors through Python's asyncio module. However, there are still hurdles for its use in developing control applications. While XMPP offers several features that generally favor MAS, locking into this architecture forces developers into a complex communication protocol. XMPP requires hosting an own server, hindering easy prototyping and debugging at an early stage of development. Additionally, asyncio is a more advanced Python ecosystem, increasing the barrier of entry.

cloneMAP (cloud-native Multi-Agent Platform) [35] is a multi-agent-platform that focuses on cloud deployment and containerization to make MAS scalable and fault-tolerant. While it is written in Go, it also supports implementation of agents in Python.

While the frameworks above are already available to run an agent system, they do not support control engineers during the development and research process, requiring the introduction of the full complexity of the controller and the distributed deployment at once.

1.2. Contribution

Considering the existing research on ACS for BES, we recognize a wide variety of control methods in research, and a rich ecosystem of tools in the fields of simulation, optimization and machine learning. Current research dealing with the integration of ACS in real systems faces significant implementation overhead interfacing with real systems and other services. A modular approach allows developers to reuse core components while interfacing with increasingly diverse BES,

external services or data processing utilities. Existing frameworks for MAS and Internet of Things (IoT)-middleware burden researchers and developers with high barriers of entry, and do not encourage a gradual development process. Based on the established challenges, a framework facilitating the extensible development of ACS for BES should fulfill the following requirements:

1. Support a high level programming language that supports the use of modern modeling and optimization tools. We identify Python to be suitable.
2. Support both simulation and real application.
3. Have a low barrier of entry. In particular, the user should be able to start out with a simple Python script, which is similar in complexity to something they would write without the framework.
4. Allow to increase complexity and scope of the application gradually, while retaining as much code as possible from the prototype stage.
5. Allow functionality to be reusable and configurable across different use cases.
6. Not restrict the user to any specific communication protocols and be freely extensible.

The contribution of this paper is as follows:

1. We present *AgentLib*,[1] a Python framework for implementation, development and testing of software agents for advanced control of energy systemsadvanced control systems]. Along with *AgentLib* itself, we also release two plugins, providing advanced and ready-to-use modules for MPC[2] and IoT-connection with FIWARE.[3]
2. We demonstrate an application of *AgentLib* in a cloud-based experiment implementing a distributed MPC with true parallel execution of the agents. We show how to gradually increase complexity of the system with *AgentLib*, exposing how different real world effects influence the controller behavior. In a second use-case, we demonstrate how to reuse modules for different applications.
3. We discuss how *AgentLib* can be used to deal with common issues in control of energy systems, including integration of heterogeneous data sources, reliability and scalability.

2. AgentLib

In this section, we present the core functions of *AgentLib*. First, we will consider the different requirements that are placed on our framework, depending on which stage we are in the development lifecycle. Then we will go over the core structure of the agents, and list the functional modules that define the agent's behaviors.

2.1. Identifying complexity dimensions for integration of advanced controllers

As per our requirements (see Section 1.2), the framework should support development of ACS as modular agent based systems from concept phase to implementation on real hardware. To achieve this, it has to both support quick run and debug cycles in the early phases, while also running on real distributed systems. Fig. 1 shows four complexity domains that arise when developing and commissioning an agent-based control system.

Time domain. When developing an advanced control application, the engineer usually performs simulations to debug and assess the control algorithm. In these simulations, the execution of control steps occurs instantly with simulation steps directly in sequence. This allows the engineer to ignore other factors such as communication delay and saves development time. However, when moving towards a real system, this omitted complexity has to be reintroduced. We therefore distinguish between the time domains *instant* and *real-time*. We also allow for *scaled real-time*, speeding up debug and test cycles for real-time applications.

Execution environment. The basic premise of an MAS is, that agents run autonomously. This implies agents should run in their own process, or even on a different hardware entirely. However, concurrent programming is notoriously difficult and cumbersome to debug, especially if the application engineer does not have a strong background in programming. When initially implementing a control algorithm, running the program sequentially in a single thread guarantees deterministic execution and allows for easy debugging. Thus, we consider execution in a *single process* and in a *distributed* environment. Note that distributed execution here is defined by running on different processes, and does not require immediate introduction of different machines or cloud architecture, although that is possible.

Communication scope. When communicating over a network, messages that are exchanged between agents have to be serialized into a format that is supported by the chosen networking protocol, for example into a JSON string. Additionally, there is a time-delay between sender and receiver. When developing on a single computing process, this overhead is not needed, as agents have access to the same address space in memory. As such, we distinguish between a *local* communication scope and communication through a *network*.

Controlled system. Before deploying a new control method, development and testing is often performed on simulations and test beds with varying degree of realism. We broadly consider the cases simulation and real system, but note that simulations can come with varying degrees of detail.

With our proposed software framework *AgentLib*, we aim to accompany developers from initial testing to real application, while requiring minimal code and configuration changes when transitioning between complexity layers. To accomplish this, we provide modular function blocks that can be embedded in different environments that control the order and speed of task execution. For execution in the cloud, instead of developing a novel middleware to execute the MAS, we developed an integration into cloneMAP, enabling a containerized execution.

2.2. Core structure

A common issue with creating agents is, that many side tasks regarding data management and communication have to be coded together within the agents' core task. To deal with this, the *AgentLib* is modular not only by nature of being a multi-agent framework, but also within the agents themselves, as functionality is coded into an agents' modules. Core functionality of an agent can be reused independently of other tasks or the chosen communication protocol. For example, a simple controller agent could consist of a PID-module and a module implementing MQTT communication with the actor. Swapping the communicator or adding functionality to an agent can be done by altering the agent's configured modules. In this way, the *AgentLib* not only allows users to extend the available selection of communication protocols, but also delivers on the promise of easily increasing complexity from concept to real implementation.

To build modular agents, we chose the structure shown in Fig. 2. It consists of the DataBroker, the Environment, and an array of functional modules. These modules communicate with each other through *AgentVariables*.

[1] https://github.com/RWTH-EBC/AgentLib.
[2] https://github.com/RWTH-EBC/AgentLib-MPC.
[3] https://github.com/RWTH-EBC/AgentLib-FIWARE.

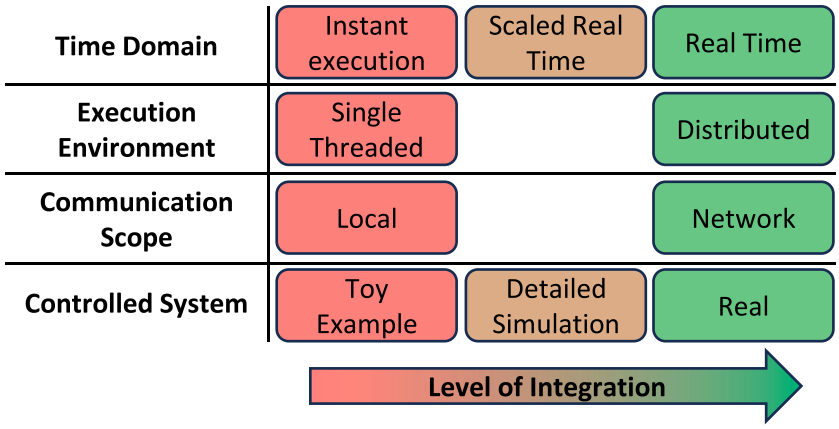


Fig. 1. Illustration of the complexity domains arising during controller development and integration.

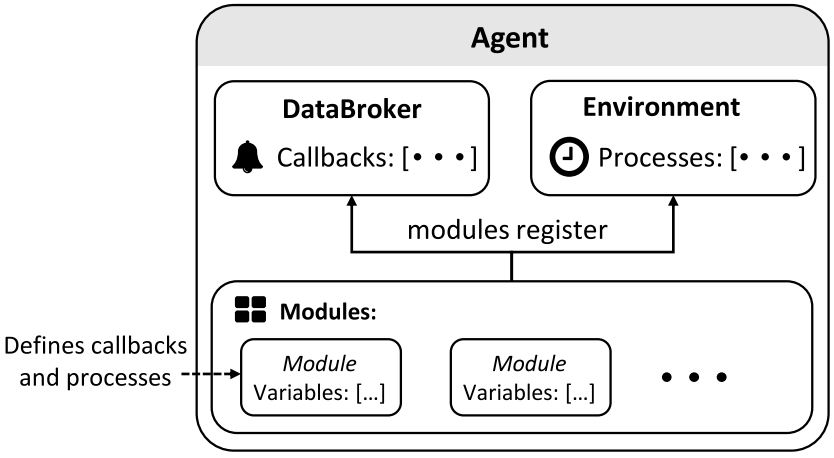


Fig. 2. Structure of a modular agent.

Modules. An agent may have an arbitrary number of modules that define the function of the agent. Users may define behavior and functions of custom modules by inheriting from the provided `BaseModule` class. To communicate with other modules in an agent, a module keeps track of so-called *AgentVariables*. It can set the value of an *AgentVariable* to share it with other modules of the same agent, and register callbacks to be notified about the change in other modules' variables (see *DataBroker*). It can also register processes with the *Environment*, defining recurring tasks. The configuration of an agent's modules is given by a JSON file or a Python dictionary, specifying their parameters and mapping the variables that are shared across agents. Typically, an agent has at least one communicator module that handles message exchange with other agents or real devices. The *AgentLib* provides a selection of different communicators and other functional modules, as covered in Section 2.3. Additionally, modules can be provided through plugins (see Section 2.4).

AgentVariables. The *AgentVariable* is the central communication object throughout the *AgentLib*. It consists of the following main fields, along with some metadata that is omitted here.

- *name*: The local name of the variable.
- *alias*: The public name of the variable.
- *value*: The value of the variable. Can be of any JSON-serializable type.
- *timestamp*: The timestamp indicating when the value was last updated.
- *source*: Which agent the variable was sent from.
- *shared*: Flag indicating whether the variable is shared with other agents.

An important concept is the distinction between *name* and *alias*. The *alias* is the public name of the variable. *AgentVariables* can be recognized by other modules as relevant communication objects, if the *alias* matches. For module-internal use, the *name* can be independent of the alias, which is often relevant for the implementation of a module. With the *source* field, modules can specify from which agents they want to receive a variable. The *shared* flag indicates whether a variable should be sent to other agents, or if it is private for agent-internal communication between modules.

Environment. The *Environment* handles the execution and synchronization of recurring tasks within an agent. It is built on SimPy,[4] and allows for execution of multiple concurrent tasks within a single computing thread. Modules register so-called processes within the *Environment* that perform a regular task and pass a timeout-event to the *Environment*, informing it when to call the process next. The *Environment* enables setting the execution speed of an agent to real-time, instant, or scaled real-time. In single-process execution mode, all agents share the same environment, whereas in distributed execution mode, each agent has its own environment.

DataBroker. In contrast to the *Environment*, the *DataBroker* handles event-based tasks within an agent, including communication between

[4] https://simpy.readthedocs.io/en/latest/.

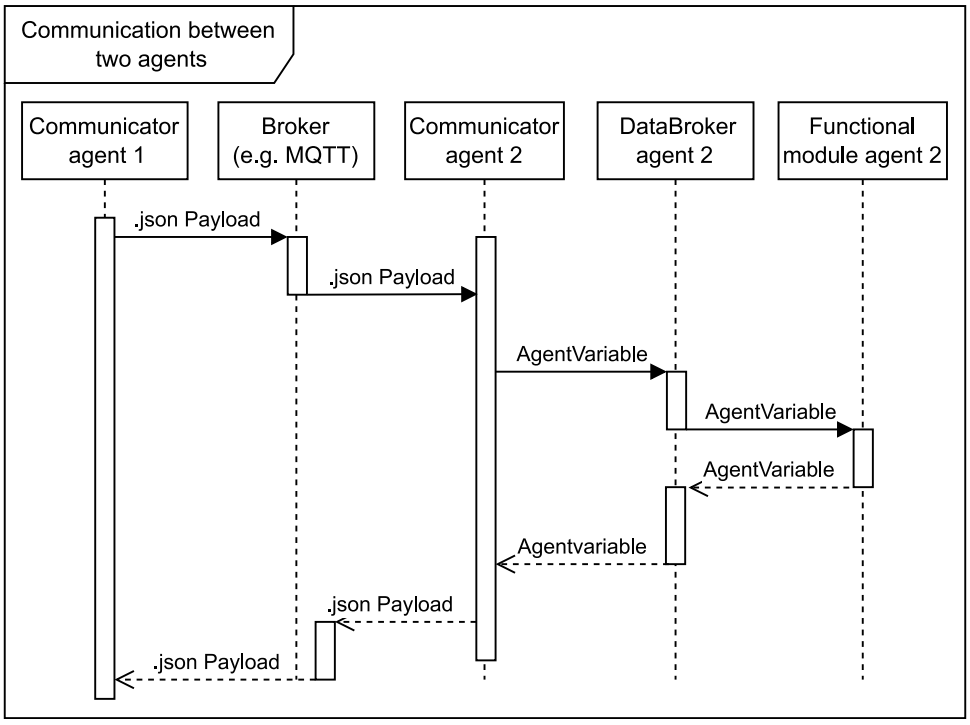


Fig. 3. Visualization of message transport through the different layers of two agents, with MQTT communicators as example.

different modules of the same agent. The functional modules of the agent register *Callbacks* with the *DataBroker* that are triggered whenever a module – either through a process or a different callback – sets a new value on one of its variables.

2.3. Available modules

In the following, we introduce the existing modules for the *AgentLib* and how to extend them. At the moment of writing, the following modules exist in the *AgentLib*:

2.3.1. Communicators

All communicators that are included with the core library follow the *publish–subscribe* pattern. Fig. 3 illustrates how communication is handled between two agents. The communicator module of *agent 1* sends a JSON message to the broker registered with the communicator, e.g., an MQTT-broker. The subscribed communicator in *agent 2* receives that message, and forwards it to the agent-internal *DataBroker* as an *AgentVariable*. The functional modules of *agent 2* that may be interested in that particular variable have their callbacks triggered, and can process the message. If there is an answer, the message is passed through the other way around. If the communication protocol should be changed, the communicator modules in *agent 1* and *agent 2* need to be swapped, but no changes to the functional modules are necessary. The following communicators are available with the *AgentLib*:

- *local*: Directly sends the message to other agents that are run in the same computing process. No protocol is needed, as agents share the same address space in RAM.
- *multiprocessing*: Uses Python's builtin multiprocessing module to send messages to agents across computing processes.
- *mqtt*: Sends messages through an MQTT broker to all other agents that are subscribed to the agent.
- *cloneMAP*: Sends messages through MQTT to other agents hosted via cloneMAP.

For the initial release, MQTT was chosen as the supported network protocol, as it is an established communication protocol in IoT systems, requires a low learning curve and offers a flexible payload structure. Furthermore, it relies on TCP/IP transport protocol, enabling encryption via TLS. If other communication protocols are required, they can be implemented by users of the framework by extending existing base modules. Although standards for inter-agent communication like Agent Communicate Language (ACL) defined by the Foundation of Intelligent Physical Agents (FIPA) exist, *AgentLib* defines its own payload protocol based on JSON. In comparison to other formats, JSON is a human-readable, commonly used data format and efficient to parse and validate in Python.

2.3.2. Simulator

During implementation and testing of an energy service, a substitute for the real system is required. As this is such a common application, *AgentLib* provides a module that is dedicated to running simulations. It continuously executes simulation steps of a model while updating exogenous inputs, sending the results in regular intervals. The simulator accepts FMUs [42] as a model, as well as linear state space models. By enabling the simulation of FMUs, it is possible to simulate models created with Modelica, MATLAB Simulink or EnergyPlus, to name a few. In the *AgentLib_MPC* plugin, an additional model type based on CasADi [23] is introduced, with which users can define arbitrary differential–algebraic systems to simulate.

2.3.3. Controllers

AgentLib enables complex control patterns but also includes basic conventional controllers. These include an on–off hysteresis controller and a PID-controller.

2.4. Module customization and extension of the library

The currently available modules of the *AgentLib* are tailored towards smart energy services and control algorithms for energy systems and buildings. However, users can easily add custom modules to perform

any task that can be written in Python within the *AgentLib* framework. By providing a path to the source code in the agent's JSON configuration, the custom code can be executed. The custom module is configured through the central configuration of the agent. Additionally, users may write plugin packages that define modules that can be used in the same way as the core modules, as long as the plugin is installed in the Python environment. Together with *AgentLib* itself, we publish two plugins, one for MPC and one for FIWARE integration, that we will use for our demonstration.

2.4.1. AgentLib_FIWARE: IoT modules for integration with FIWARE

FIWARE is an open source framework for accelerating the development of smart IoT solutions [43]. Previous work has shown how FIWARE can be used for the implementation of smart control services [44]. Communicators that connect with FIWARE are included in the plugin *AgentLib_FIWARE*. They make use of the FiLiP[5] package that predefines clients for various FIWARE components. At the time of writing, different types of communicators are available:

- IOTA-Agent: Typically, field devices connect FIWARE over IoT-Agents (IOTA). In *AgentLib*, simply pair the module `iotamqtt.device` with a module that provides measurements and relays actuations (e.g. a simulator or a module connecting to the physical system) to send measurements and receive actuator commands. The control agents can communicate with the device using the module `iotamqtt.context_broker`.
- ContextBroker: The Orion context broker of FIWARE is the central component of the FIWARE stack that provides update, query, registration and subscription functionality. The data itself is stored in an underlying MongoDB database. Several modules exists (`context_broker.*`) to communicate in a scheduled or callback-based manner with single attributes or entities.
- Time-series databases: To access time-series data for, e.g., model training, communicators to QuantumLeap `time_series.quantum_leap` and InfluxDB `time_series.influx` exist.

2.4.2. AgentLib_MPC: Modules for nonlinear and distributed MPC

The MPC plugin provides JSON configurable MPC agents that rely on gray-box or black-box models inheriting from the *AgentLib_MPC*s CasadiModel class. CasADi [23] is also the backend that is used to define and solve the optimization problems of the MPC. The nonlinear optimal control problems can be discretized by direct collocation or multiple shooting and support C–Code generation thanks to CasADi, allowing for fast computations. The plugin also provides modules for distributed model predictive control (DMPC) based on the ADMM-algorithm [45]. Adding a DMPC agent to a network can be done by changing a small number of configuration fields from the decentralized MPC, allowing the developer the confirm the model and MPC in a standalone setting first.

2.5. Comparison to existing multi agent frameworks

Popular multi agent frameworks like JADE and SPADE require developers to define new agent classes for each agent type, with behaviors directly coded into these classes. In these frameworks, communication between behaviors requires direct access to the agents' attributes or dedicated message protocols, and changing communication methods requires modifying the agent's code. While *AgentLib*'s modules also use inheritance patterns, the key distinction lies in how agents are composed: Rather than coding new agent classes, *AgentLib* agents are assembled from independent modules through configuration files. As shown in Fig. 3, this separation allows core functionality like communication to be handled by swappable modules, independent of the agent's main tasks. The *DataBroker*'s structured variable system enables the seamless communication between modules. The configuration-driven architecture enables rapid prototyping and simplified deployment. For example, transitioning from simulation to hardware deployment often requires only updating the communication configuration while keeping control configuration intact. This separation of concerns allows domain experts to focus on their control strategies. The accessibility of *AgentLib* is complemented by ready-to-use modules for common tasks such as simulation of FMU-models, standard controllers and MPC.

3. Example application: From concept to cloud-connected test bed in four stages

To demonstrate the capabilities of our framework, we provide an example how we implemented an advanced control scheme for room ventilation, gradually increasing the complexity from the simulation stage to a test bed. Let us consider the scenario, where we want to implement a novel control algorithm on an IoT platform and run it against a real test bed. Presumably, we might not be familiar with the algorithm. Considering the algorithm development together with the challenges a real system brings would seem like a daunting task. *AgentLib* can help to break down this task into smaller steps. In this section, we demonstrate an exemplary development process in four stages.

1. Local simulation
2. Distributed simulation
3. Connection with IoT infrastructure and cloud execution
4. Experiment on test bed

The remainder of this chapter is organized as follows. First, the control problem and solution algorithm are explained. Then, we will discuss the implementation and results of each stage in detail.

3.1. Example application

We consider a resource sharing problem in room ventilation, where the CO_2 concentration of four rooms has to be controlled. The rooms need to keep their CO_2 concentrations below 800 ppm. The total fresh air volume flow that can be provided to the rooms is limited. A DMPC for the system must find the optimal distribution of the limited air flow, so that the comfort violations remains minimal and are evenly distributed. With a DMPC, a potentially large number of rooms can coordinate the air flow distribution, avoiding a large central control problem. The same problem has already been examined in simulations by Li and Wang [16]. Our goal is to run the distributed controller against a real system and accommodate for the full complexity that comes with an experiment.

3.1.1. Test bench

The physical system is the test bed shown in Figs. 4 and 5. It consists of a central ventilator and four isolated cabinets that each encompass an air volume of about $2\,m^3$. Each cabinet can have an individual supply air volume flow controlled by a VAV. Additionally, the cabinets can receive CO_2 injections individually to emulate the presence of humans. The CO_2 valve is controlled through pulse width modulation and fitted to emulate the occupancy of a $72\,m^3$ reference room. The local control of the test bed is handled by a Beckhoff™ PLC (Programmable Logic Controller). We interface with the PLC through a custom communicator module (see Section 2.4), which uses the pyads[6] package.

[5] https://github.com/RWTH-EBC/FiLiP.

[6] https://github.com/stlehmann/pyads.

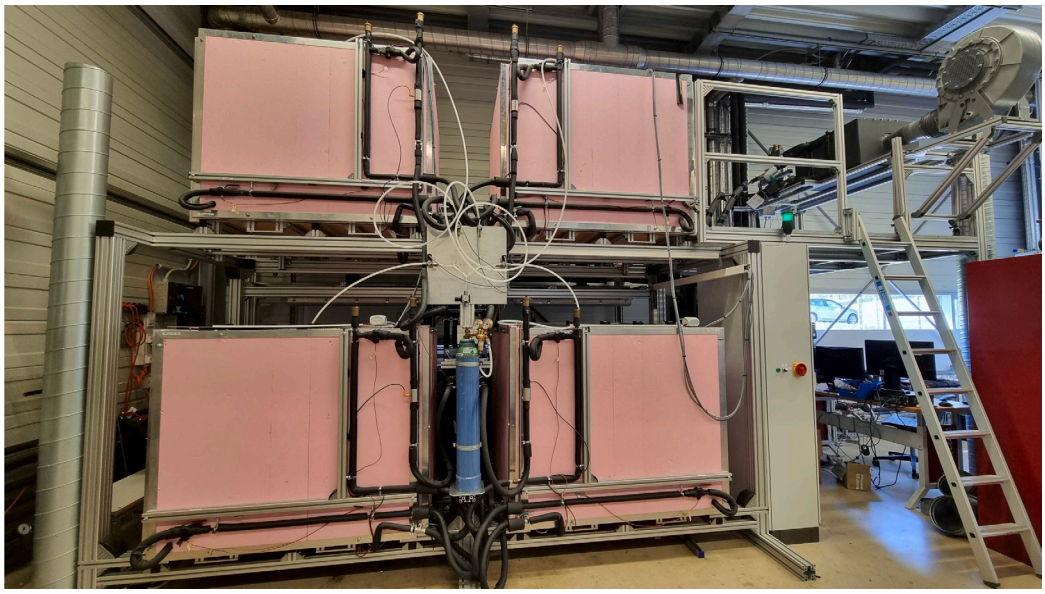


Fig. 4. Test bed for the multi-agent ventilation problem.

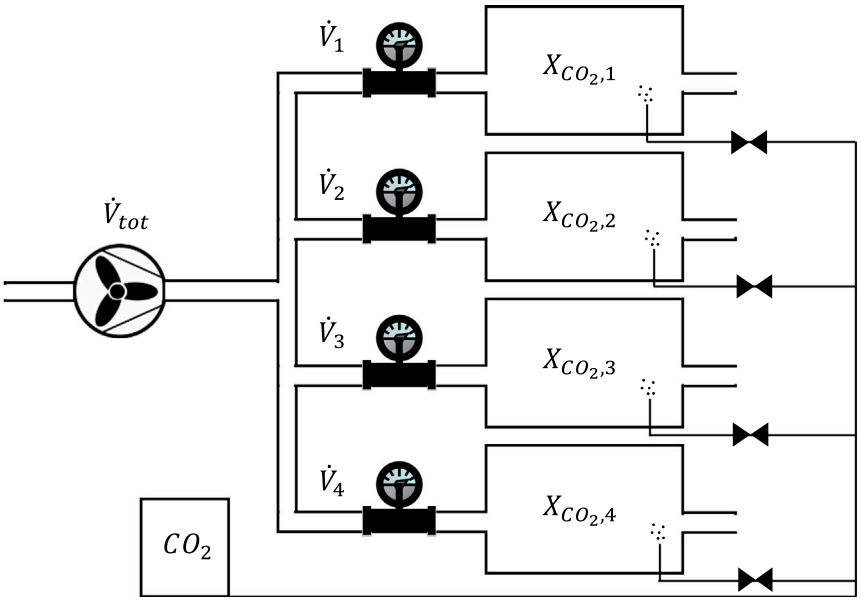


Fig. 5. Schematic of the test bed.

3.1.2. Distributed model predictive controller

We consider an MAS consisting of an agent for each room, and a supply air agent checking the air volume flow limit. The agents solve the consensus problem

$$\begin{aligned} \min \quad & f_C + \sum_i f_i \\ s.t.: \quad & \dot{V}_i^C - \dot{V}_i = 0 \quad \forall i \in \{1,2,3,4\}, \end{aligned} \tag{1}$$

where f_C is the objective of the central ventilator, f_i are the objectives of the rooms, and the superscript C corresponds to the central ventilator. The consensus constraint on the individual air volume flows $\dot{V}_i$ ensures that the supply air agent knows the airflow of all rooms. The central ventilator enforces the constraint

$$0 = \dot{V}_{\text{tot}} - \sum_i \dot{V}_i, \tag{2}$$

which defines the total volume flow $\dot{V}_{tot}$ as the sum of expected volume flows per room. The total volume flow $\dot{V}_{tot}$ is limited, while also affecting the local objective of the supply air agent

$$f_C = q_V \dot{V}_{tot}, \tag{3}$$

where q_V is a weight penalizing higher volume flow rates. The room agents i have the objective of keeping the CO_2 concentration below a maximum comfort bound. This objective is implemented through a soft constraint

$$X^i_{\text{CO}_2} - s_i \leq X^{max}_{\text{CO}_2}, \tag{4}$$

with the slack variable s_i, which is penalized in the local objective of the room agent

$$f_i = q_{s,i} s_i^2 \tag{5}$$

with the weighting parameter $q_{s,i}$. The evolution of the CO_2-concentration in a room i is modeled by the differential equation

$$\frac{d X_{\text{CO}_2,i}}{dt} = \frac{n_i \cdot \dot{m}''_{\text{CO}_2} \cdot \frac{M_{\text{air}}}{M_{\text{CO}_2}} - \rho_{\text{air}} \cdot \dot{V}_i \cdot (X_{\text{CO}_2,i} - X_{\text{CO}_2,in})}{V_i \cdot \rho_{air}}, \tag{6}$$

where X_{CO_2} is the CO_2 concentration, n_i is the number of occupants in room i, $\dot{m}''_{CO_2}$ is the mass flow of CO_2 emissions per person, M_{air} and M_{CO_2} are the molar masses of the air and CO_2 respectively, and $X_{CO_2,in}$ is the CO_2 concentration in the supply air. V_i is the volume of room i and ρ_{air} is the density of air. The actuation variable of the rooms is the ventilation air volume flow $\dot{V}_i$.

The distributed optimization problem (1) is solved with the ADMM algorithm. ADMM is a well-researched technique for distributed optimization, which we will cover briefly. For a thorough explanation of ADMM, refer to [45]. For the ADMM algorithm, the Augmented Lagrangian of problem (1) is required and reads:

$$L(\lambda, \dot{V}_i, \dot{V}_i^C) = f_C + \sum_i f_i + \sum_i \lambda(\dot{V}_i - \dot{V}_i^C) + \sum_i \frac{\rho}{2}\|\dot{V}_i - \dot{V}_i^C\|_2, \tag{7}$$

with multipliers λ and the penalty parameter ρ. From this Augmented Lagrangian, an iterative procedure can be derived to find a local optimum of the original consensus problem (1). In the following, we will state the optimization problems that are solved by the local agents, and state the iterative procedure to solve the consensus problem, as it is used within this work.

The subproblem solved for room i at each iteration of the ADMM algorithm is described by the following equations (the index i has been omitted for readability):

$$J_{\text{room}}\left(\dot{V}(\cdot), X_{CO_2}(\cdot), s(\cdot)\right) = \tag{8a}$$
$$\min_{\dot{V}(\cdot), X_{CO_2}(\cdot), s(\cdot)} \sum_{k=0}^{N-1} \left(w(s^k)^2 + \lambda^k \dot{V}^k + \frac{\rho}{2}(\dot{V}^k - \dot{V}^{avg,k})^2\right)$$
$$s.t.: \quad X_{CO_2}^{k+1} = X_{CO_2}^k + \int_{t_k}^{t_{k+1}} \frac{dX_{CO_2,i}}{dt}\, dt \tag{8b}$$
$$0 \le \dot{V}^k \le \dot{V}^{max} \tag{8c}$$
$$X_{CO_2}^k - s \le X_{CO_2}^{max} \tag{8d}$$
$$\forall k \in [0, \dots, N-1]. \tag{8e}$$

The notation $\dot{V}(\cdot)$ denotes all entries of this variable over the prediction horizon N, where the exact dimension depends on the transcription of the problem. The average volume flow $\dot{V}^{avg,k}$ is the average between the ventilator's and the rooms' value for the air volume flow, which is determined in the ADMM coordination step (see Eq. (12)). The subproblem which is solved by the central supply air agent reads

$$J_{\text{ventilator}}\left(\dot{V}_i(\cdot), \dot{V}_{tot}(\cdot)\right) = \tag{9a}$$
$$\min_{\dot{V}_i(\cdot), \dot{V}_{tot}(\cdot)} \sum_{k=0}^{N-1} \left(q_m \dot{V}_{tot}^k + \sum_{i=1}^{M} \lambda_i \dot{V}_i + \frac{\rho}{2}(\dot{V}_i - \dot{V}_i^{avg})^2\right)$$
$$s.t.: \quad 0 = \sum_i \dot{V}_i^k - \dot{V}_{\text{tot}}^k \tag{9b}$$
$$\forall k \in [0, \dots, N-1]. \tag{9c}$$

The ADMM update steps are [45]

$$\dot{V}_i(\cdot)^{v+1} \leftarrow J_{\text{room},i}\left(\lambda_i^v, (\dot{V}_i^{avg})^v\right) \tag{10}$$
$$\dot{V}_i^C(\cdot)^{v+1} \leftarrow J_{\text{ventilator}}\left(\lambda_i^{C,v}, (\dot{V}_i^{avg})^v\right) \tag{11}$$
$$\dot{V}_i^{avg}(\cdot)^{v+1} = \frac{\dot{V}_i(\cdot)^{v+1} + \dot{V}_i^C(\cdot)^{v+1}}{2} \tag{12}$$
$$\lambda_i^{v+1} = \lambda_i^v + \rho\left(\dot{V}_i(\cdot)^{v+1} - \dot{V}_i^{avg}(\cdot)^{v+1}\right) \tag{13}$$
$$\lambda_i^{C,v+1} = \lambda_i^{C,v} + \rho\left(\dot{V}_i^C(\cdot)^{v+1} - \dot{V}_i^{avg}(\cdot)^{v+1}\right) \tag{14}$$

Here, v is the iteration number. The algorithm is initialized with 0 for all multipliers λ. The initial guess for the volume flows $\dot{V}$ is initialized as the average of each individual agents' initial guess, which is obtained to be in the center of the upper and lower boundaries for the volume flow. We terminate the algorithm after a fixed number of iterations. For subsequent control steps, the algorithm is warm-started with the previous solution. Note that we describe the ADMM algorithm with little generics, preserving the physical meaning of the variables in our problem. Our framework does however provide a fully generic implementation of the ADMM algorithm for consensus and exchange problems, allowing the solution of arbitrary distributed MPC problems.

3.2. Implementation in four stages

The following section guides the reader through the implementation of the distributed controller on the test bed. We implement four versions of the control task, each introducing new complexities that arise when controlling real systems. We consolidate our development process into four stages for the sake of brevity, but note that many combinations of the drivers of complexity introduced in Section 2.1 can be varied independently, resulting in different development stages. The development process is therefore flexible with regard to the needs of the developer.

3.3. Stage 1: Local simulation

The first stage is the local simulation stage. This stage introduces the modeling of our system and the numerical solution methods for the controller. As process model for our MPC, we create a *CasadiModel* with the *Agentlib_MPC* plugin, where we implement the differential equation (6). Additionally, we define the constraint (4) and the objective function (5) of the room. We also define a model for the agent supervising the ventilator, consisting of the constraint (2) and the objective (3). The `admm_coordinated`-module allows agents to participate in a coordinated DMPC-network, where the coordinator agent is fitted with an `admm_coordinator`-module. We configure four agents with an ADMM module using the room model, one agent with an ADMM module using the ventilator model, and a coordinator for ADMM. To close the control loop, we also need a system to run the distributed controller against. For that, we configure an agent with four simulator modules from the *AgentLib*'s core, and assign them the same CasadiModel that the DMPC uses. Therefore, there is no model-plant mismatch in stage 1. Fig. 6 shows how the agents are configured in each stage. In stage 1 all agents are fitted with a local communicator to allow for instant execution of the simulation. The MAS is executed in a single-process environment with instant execution, i.e. no artificial time delay.

Simulation results of stage 1

We simulate a scenario of 15 min where the goal is to reach a steady-state, tuning the parameters for MPC and ADMM in the process. We end up with a prediction horizon of 7 min with a step size of 60 s, discretized with direct collocation over Legendre polynomials of second order. We set the penalty parameter $\rho = 10^{-6}$ and configure a fixed number of ADMM iterations of 30. Each of the four rooms have a different occupancy, with room 1 having the lowest, and room 4 having the highest occupancy, resulting in varying CO_2-emissions per room. Fig. 7 shows the evolution of the CO_2 concentrations and the associated air volume flows in stage 1. We start the simulation with a CO_2 concentration of 400 ppm in each room. As the distributed MPC agents are aware of the occupancy in their room, they immediately request adequate air supply from the ventilator agent, to avoid violating the comfort bound of 800 ppm. There are still comfort violations even though the total volume flow matches the maximum volume flow, confirming that there is in fact a ventilation shortage and therefore a need to coordinate distribution of total volume flow. We can confirm that the agent with the strongest disturbance (room 4) is granted the largest share of the total ventilation power. However, we can also note that the steady-state CO_2 concentrations are not precisely equal, indicating that the ADMM algorithm did not converge to optimality. As were are using ADMM on a nonlinear problem, convergence to lower accuracy is expected, and we accept this result.

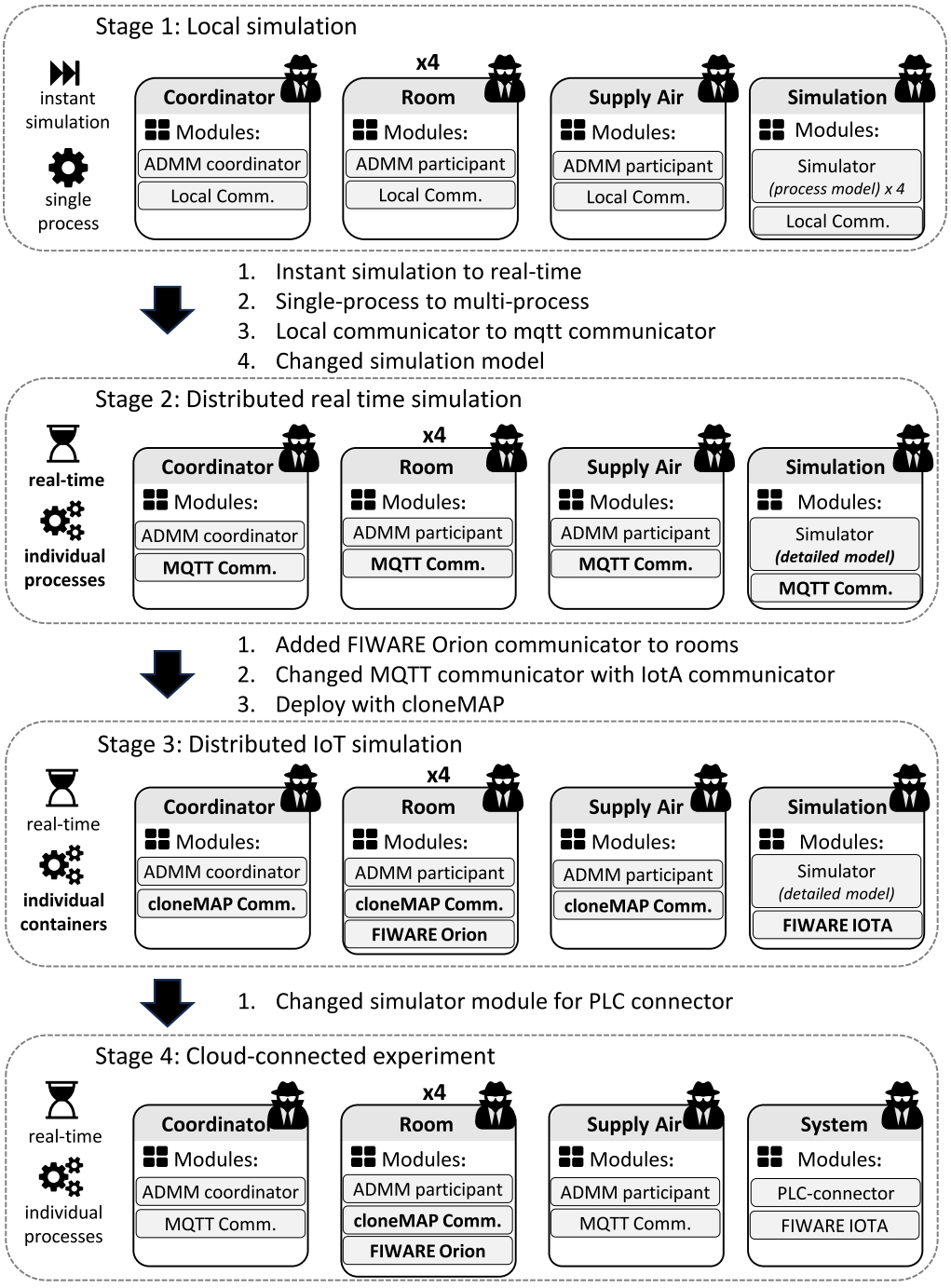


Fig. 6. Agent configurations over each stage of the development process. Changed configurations are shown in bold.

Execution time

In the following section, we give an overview over the computation times and simulation overhead. The simulations were performed on a laptop with an Intel i7-1355U CPU running Python 3.11. We used CasADi's C–Code generation to reduce time spent during the optimization, exposing overheads of the framework as much as possible. We analyzed the time spent on different tasks with the Python profiler py-spy.[7] Running the first stage takes about 17 s, which includes 15 MPC steps, each with 30 ADMM iterations across 5 agents, resulting in the solution of 2250 nonlinear optimization problems (NLPs). Table 2 gives the share of different operations of the total execution time. The largest share with 81% is attributed to the solution of the NLPs occurring in each agent. Other notable tasks include results saving, which includes saving the complete prediction of optimization variables, inputs and parameters of each of the 2250 NLPs to a CSV file. About 6.5% of the runtime can be categorized as other, including tasks of the coordinator agent, and overhead with regard to communication between agents and modules.

3.4. Stage 2: Distributed simulation

In the distributed simulation stage, the agents are not run in a single process, but in a separate Python process per agent. This necessitates the use of a network communicator, which exchanges JSON

[7] https://github.com/benfred/py-spy.

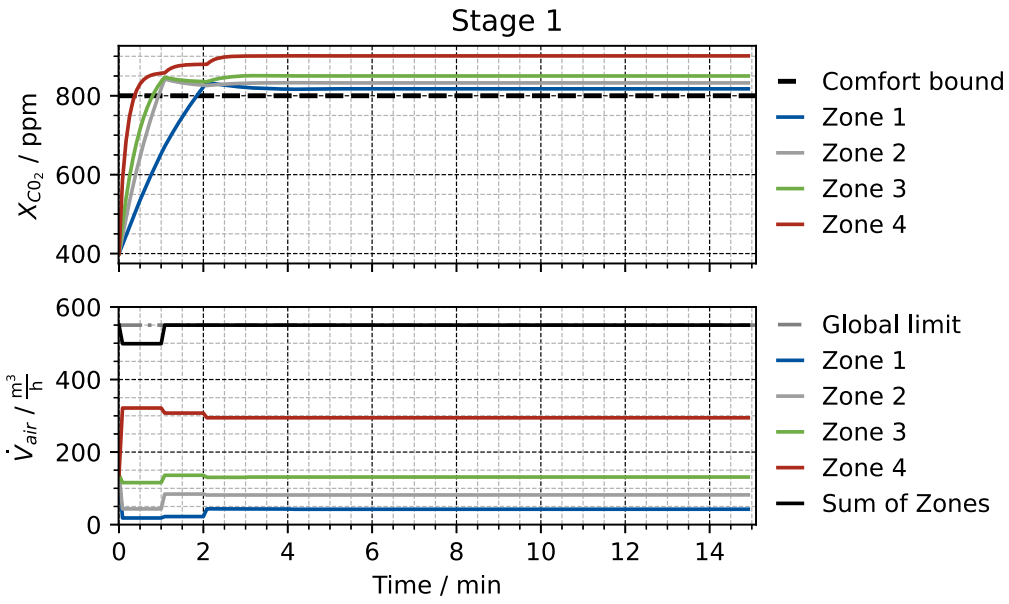


Fig. 7. Simulated CO_2 concentration and volume flow per room in stage 1.

Table 2
Distribution of execution times per task in stage 1 determined with py-spy. The total execution time was 17 s.

Task	Percentage
NLP solving	81.0%
Saving MPC results	6.9%
MPC input preprocessing	3.3%
Optimization setup & C-code compilation	1.3%
Simulation	1.0%
Other	6.5%

serialized messages between the agents. In this case, we choose the MQTT communicator, although we could have used the Multiprocessing Communicator, eliminating the need for an MQTT broker. The execution time of the simulation is switched from instant to real time to accommodate for the network connection. With this execution setup, we now consider effects like latency due to network connection and latency in agent initialization, requiring synchronization. In this step, we also exchange the simulation model, which before was the same as the controller model, for a more sophisticated model that is written in Modelica using the AixLib [25] and exported as an FMU [42]. We include this step here for the sake of brevity, but generally advise to keep the move to a distributed architecture separate from a model change. The configuration of the MAS that is executed in stage 2 is shown in the second row of Fig. 6, with changes highlighted in bold. Note that the user does not have to rewrite any major code. Instead, the shown changes are simple option changes in the configuration. If this simulation were written without help of a framework, such large changes would be impossible without diving into the execution code.

Results of stage 2

Fig. 8 shows the evolution of the CO_2-concentration and the air volume flow for stage 2. Compared with stage 1, it is noticeable that there are large comfort violations at the beginning of the simulation. This is explained with startup times and the registration process caused by the distributed execution with Python's multiprocessing. While the agents register with the coordinator, they miss the first control cycle and the air volume flow for each room remains at its default value. However, when the registration is completed, the agents correctly allocate the limited supply air, quickly returning the rooms' CO_2 concentrations to an even distribution. We can also see that the local volume flow setpoint of room 4 is not exactly met, which can be explained by model-plant-mismatch between the CasadiModel and the Modelica model. In the Modelica model, the actual volume flows are determined by a ventilator and VAVs which are unable to provide the expected total volume flow.

Execution and communication times

In this stage, we ran the simulation on the same laptop as before (Intel i7-1355U CPU), but as the setup is distributed, each agent runs on a separate computing process. As the agents run in parallel and the ADMM implementation is synchronous, the optimization times are determined by the slowest agent and network latency. The MQTT broker is hosted on a virtual machine located in the same network. We examined the execution times for a control step based on timers within the admm-coordinator module and the solution stats of the local NLPs. A single control step with 30 ADMM iterations includes 32 round-trip messages sent between the coordinator and each local agent, and the solution of 30 NLPs per agent. Execution of a control step took 2.4 s on average measured by the coordinator. The slowest agent spent on average 0.7 s for the solution of all 30 NLPs. Based on the profiling results from py-spy, MPC data-handling and CPU-bound overhead of the MQTT client account for no more than 0.2 s each per local agent. This leaves 1.5 s that are unaccounted for by actions of the agent that are lost on network latency and the MQTT broker. Considering the number of messages per control step, the one-way communication delay here is on average 23 ms. We can see that, even though the MQTT broker was located within the same network, communication makes up for about two thirds of the algorithm execution time in this case.

3.5. Stage 3: Connection with IoT infrastructure

For the third stage, our goal is to execute the agents on cloud infrastructure, and integrate an IoT middleware for data management. We use cloneMAP to deploy our agents in containers, and FIWARE as an IoT middleware. The purpose of this stage is to demonstrate how scalable infrastructure can be incorporated in the testing phase. For cloneMAP, the mqtt-communication module of each Agent is changed to a cloneMAP module, which allows for communication with the cloneMAP middleware and other Agents in cloneMAP through MQTT. The JSON-Configs of the Agents are posted to the REST-API of cloneMAP.

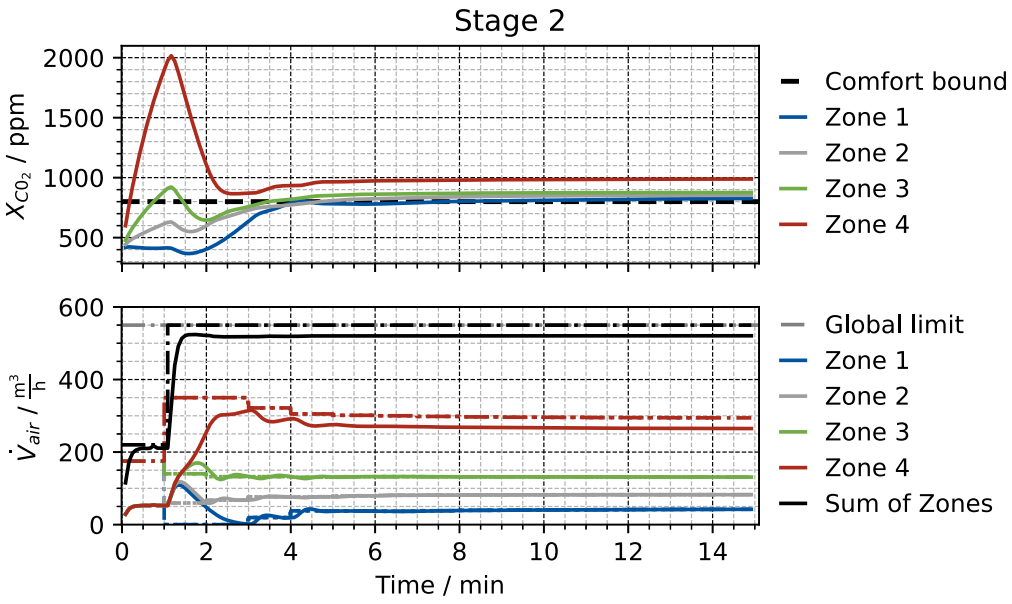


Fig. 8. Results stage 2 and 3. The continuous lines show actual measurements from the Modelica model, while the dash-dotted lines show the set points received from the MPC.

In the previous stages, we did not discriminate between messages that are sent between control agents, and messages that are exchanged with the Simulator, i.e. sensor readings and actuator set points. Now, we consider the case, where the sensor and actuators of our controlled system are managed through FIWARE, but we still simulate the system. We introduce two additional communicator modules to communicate sensor and actuator values with FIWARE, while the ADMM communication messages are handled over the cloneMAP MQTT communicator. The simulator is configured with an IOTA-communicator module, which communicates with the FIWARE IOTA devices related to its inputs and outputs. The control agents are configured with a context broker communicator, which synchronizes the changes of values in the agent with the values in the ORION context broker. Through utility functions of the *AgentLib_FIWARE*-plugin, we can automatically generate the required entities and devices in FIWARE based on their agent configuration, allowing for easy setup of the FIWARE emulation. Hence, this stage emulates the commissioning of an IoT-based system using FIWARE. The configuration for stage 3 is also shown in Fig. 6.

The results of stage 3 are almost identical to the results from stage 2 (Fig. 8), indicating that the communication overhead of FIWARE does not significantly impact the effectiveness of the control strategy.

3.6. Stage 4: Implementation on test bed

In the final stage, we switch the simulation model for the test bed. We change the simulator module with an `ads_module`, which is a custom module that handles communication with Beckhoff PLCs. The PLC has an internal control that can be overwritten by our control algorithm, reads the sensor signals and sends the actuator signals. The agent with the ADS module is run on the machine that is connected to the PLC. The other agents are run on a different machine. As stage 3 already showed the capability of containerized execution, we opted for multiple processes in this stage for easier debugging. The final configuration of the MAS is shown in Fig. 6.

Results of stage 4

For the final stage, we adjusted some parameters to accommodate for the real system. The maximum air volume flow is set to 100 $\frac{m^3}{h}$ and the occupancy values are changed, but their relative magnitude is kept as before. Fig. 9 shows results for a 15 min experiment in the final stage. Room 4, which has the highest load, violates the comfort bound, but not by an unreasonable amount, just as in the previous stages. Rooms 2 and 3 also have small comfort violations. We would like to restate that these violations are due to the design of the experiment, as only an insufficient available airflow results in meaningful coordination between the agents. As the test bed allows for larger volume flows, the limit of the total volume flow is purely virtual. Therefore, we have some violations of the total volume flow at the beginning of the experiment. However, those are due to the inability of the room 2 VAV to precisely hit the set point. Looking at the execution and communication times, we see little difference between execution and communication time compared to stage 2, however the system has a larger control delay, in the sense that the local VAVs take some time to provide the requested volume flow.

4. Reusing the code for new applications: Multi-room supply temperature consensus

Having demonstrated the framework's effectiveness in the ventilation scenario, we now showcase its reusability and scalability by applying it to a different building control application: temperature regulation. The task is to control the temperature of n rooms while negotiating a supply temperature, as shown in Fig. 10. This problem can be solved as a nonlinear DMPC, where room agents set their radiator valve position $u_{\text{valve,i}}$ as local controls, the heating agent sets the reference supply temperature T_{supply}^{ref} and the supply temperature T_{supply} is the consensus variable. We reuse the existing ADMM-modules, adjusting only the configured variables and model class.

4.1. Modeling

To model the thermal dynamics of the heating system, we employ a two-capacity RC model for each room (Fig. 11). This model captures the heat transfer between the air, walls, and radiator. The radiator's heat transfer rate, $\dot{Q}_{\text{rad}}$, is modeled using a modified NTU (Number of Transfer Units) method, which, while typically used for heat exchangers, effectively captures the diminishing returns relationship between valve opening and heating power in our radiator model:

$$\dot{Q}_{\text{rad}} = \epsilon \cdot \dot{m} \cdot c_{\text{water}} \cdot (T_{\text{supply}} - T_{\text{air}}) \tag{15}$$

where the effectiveness ϵ is

$$\epsilon = 1 - e^{-NTU} \tag{16}$$

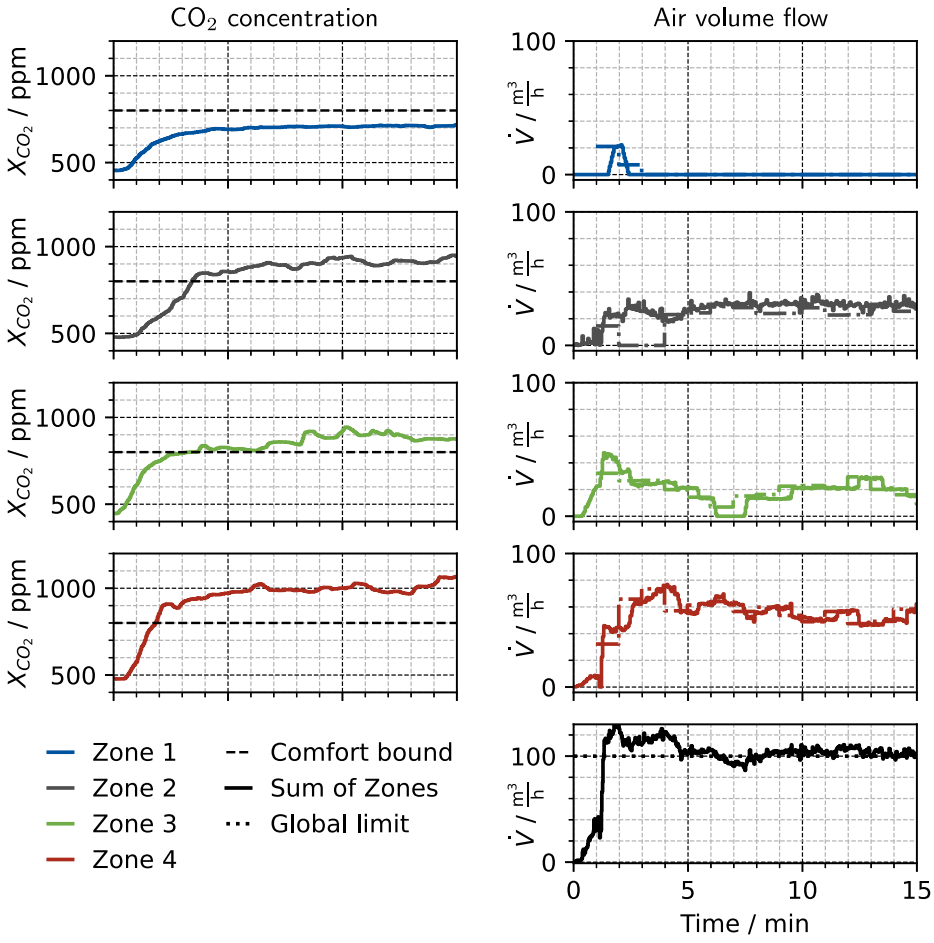


Fig. 9. Experimental results of the distributed controller on the test bed. Dash-dotted lines display the setpoints determined by the distributed controller and continuous lines show the measurements.

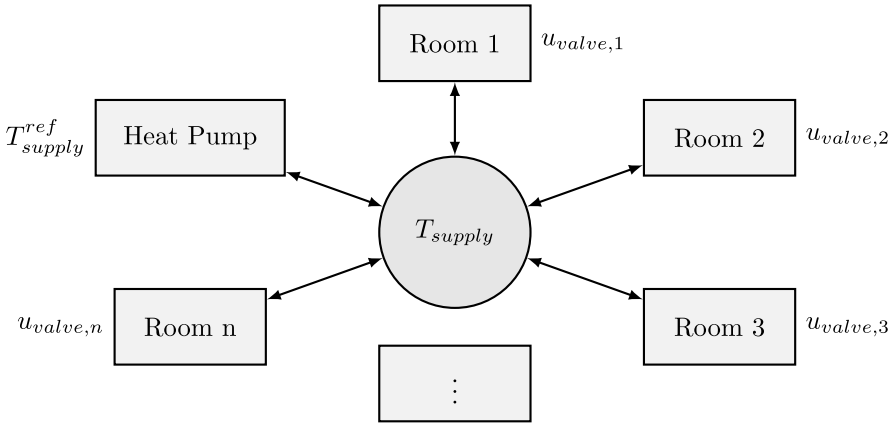


Fig. 10. Schematic of the multi-room heating control problem.

with

$$NTU = \frac{\alpha_{\text{rad}}}{\dot{m} \cdot c_{\text{water}}}, \tag{17}$$

where α_{rad} represents the overall heat transfer coefficient of the radiator, c_{water} is the specific heat capacity of water, T_{supply} is the supply water temperature and T_{air} is the room air temperature. The mass flow rate of water through the radiator, $\dot{m}$, is controlled by the radiator valve position, u_{valve} (ranging from 0 to 1), and is given by

$$\dot{m} = (u_{\text{valve}} + \varepsilon) \cdot \dot{m}_{\text{max}}, \tag{18}$$

where $\dot{m}_{\text{max}}$ is the maximum mass flow rate of water through the radiator and $\varepsilon > 0$ is a small positive constant for numerical stability, preventing division by zero when the valve is fully closed.

Heat transfer between other nodes i and j (e.g., air and walls) is modeled using a standard thermal resistance approach

$$\dot{Q}_{ij} = \alpha_{ij} \cdot (T_i - T_j), \tag{19}$$

where $\dot{Q}_{ij}$ is the heat flow rate between nodes i and j, α_{ij} is the heat transfer coefficient between nodes i and j, and T_i and T_j are the temperatures of nodes i and j. The thermal capacities and heat transfer

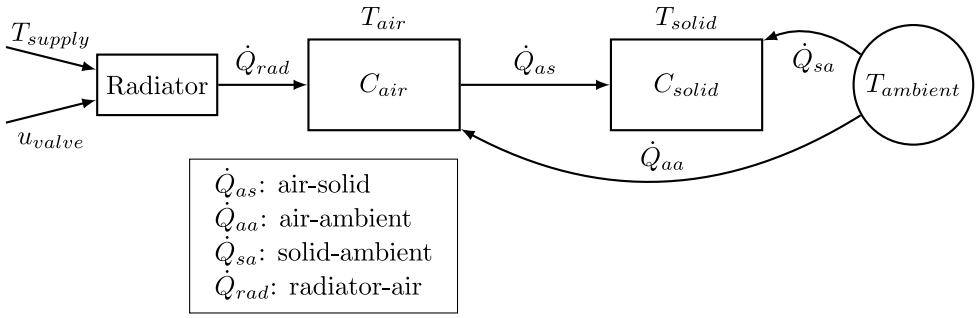


Fig. 11. RC-model of the room heating system showing thermal capacities and heat flows.

coefficients of the rooms were randomly generated based on assumed floor areas sizes between 15 m^2 and 30 m^2. The heating supply agent employs a stationary model where the reference supply temperature T^{ref}_{supply} is effective immediately.

We give the heating supply agent a linear cost term proportional to the supply temperature T_{supply}, encouraging operation at low T_{supply}, which is efficient for e.g. heat pumps without adding numerical complexity. The room agents have a linear objective term with regard to the valve opening u_{valve}, and a quadratic penalty for violation of comfort bounds, ensuring the occupants comfort while encouraging energy efficient behavior. To improve ADMM convergence, the supply temperature, T_{supply}, is scaled as follows:

$$\tilde{T}_{\text{supply}} = \frac{T_{\text{supply}}}{40} \tag{20}$$

This scaling factor of 40 was chosen to normalize the coupling variable to be on the order of 1. Because the coupling variable appears in the agents' objectives both as a linear term and within a quadratic penalty term, this normalization helps balance the contributions of these terms and improves the convergence speed of the ADMM algorithm. It also simplifies the tuning of other weights in the local objective functions relative to the ADMM penalty parameter. The implementation of this model is shown in the Appendix in Fig. 17

4.2. Implementation and results

We simulated the MAS of 50 rooms for a week. The MAS configuration can be found in Fig. 12, consisting of 103 agents. We use the same model for the simulation and the MPC. As the internal wall temperature T_{wall} is not directly measurable in a real-world scenario, we employ a Moving Horizon Estimator (MHE) approach to estimate its value. The *AgentLib_MPC* plugin provides an MHE module, which we configure to estimate T_{wall} based on past inputs and measurements over a receding horizon. The MHE module utilizes the same process model as the MPC. A custom comfort module generates time-varying comfort boundaries based on the time of day and day of the week. A weather agent has been added that informs all rooms about the weather forecast, replaying historical data from a file. The subscriptions in the room agents need to be updated with the ID of the weather agent to automatically listen to weather updates, however no code changes are necessary for the integration. As can be seen, data from various sources or preprocessing steps can be added seamlessly to existing code by adding or swapping modules and agents.

Fig. 13 shows the evolution of the room temperatures, comfort violations, valve positions and the agreed supply temperature. Since the global goal is to minimize the supply temperature T_{supply}, the intuitive solution for this problem is that the room with the highest heat losses should open its valve fully, requesting just the required supply temperature. In the valve plot of Fig. 13 the envelope shows there is always a room with valve opening 1, indicating the DMPC successfully finds this solution, while dynamically changing T_{supply} at 8 am and 6 pm, quickly adhering to the setpoint changes. The average comfort violation over a week was 0.8 Kh, with the worst room exhibiting 22.3 Kh of discomfort, caused by a constant offset of around 0.13 K. In the 50-room heating scenario, each agent required an average of 23 ms per DMPC iteration, with 20 iterations performed per time step, resulting in a computation time of about 33 s per control step including all overhead and all agents, even without parallelization.

4.3. Implementation effort

Implementing the multi-room heating control use case leveraged the existing modules and required no adaptation of the core codebase. The primary development effort focused on the following tasks:

- **Thermal Model Development:** The model itself represents a relatively standard approach. Implementing it merely requires inheriting from a Python class and writing down the equations using CasADi (see Fig. 17). In a real-world implementation, this would also include tuning the model with real data.
- **Comfort and Weather Modules:** Two new modules were implemented for this use case: a comfort module and a weather module. This is a quick task and required inheriting from the *BaseModule* and implementing the core logic. As the source of these data will usually be unique, it makes sense to have a custom implementation. The implementation of the weather module is given in Fig. 16.
- **Configuration Scripting:** To facilitate the creation and management of configurations for a large number of rooms (50 in this study), a dedicated configuration script was developed. This script automates the process of generating agent configurations, assigning unique agent IDs, managing communicator subscriptions, and populating parameter values based on the random room generation process. The generated configurations also serve as documentation of the generated parameterization.
- **Tuning:** We selected suitable weights for the local objectives and the ADMM penalty. This was accomplished by doing a few short simulations with a lower number of agents.

Moving forward, one can proceed similarly to the stages described in the ventilation use case, swapping the simple models for sophisticated ones, moving to parallel real-time execution, swapping the historic weather forecast for a live one and finally, controlling a real building. Building up to this, one might add modules emulating sensor failures on the one hand, and watchdogs or data validation on the other hand, testing and improving the system's resilience. As the MPC becomes more sophisticated, models can be swapped and modules can be added to handle the required data, for example connecting to an API for live electricity prices, listening to user setpoints or updating model parameters through online-learning. As the flow of variables is handled through the configuration files, it can be rerouted without touching existing control implementations. For example, input data to the MPC module can be validated by a pre-processing module, and post-processors can catch invalid controls. Alternatively, one might want to test a different controller, reusing the weather and simulation agents for a fair comparison.

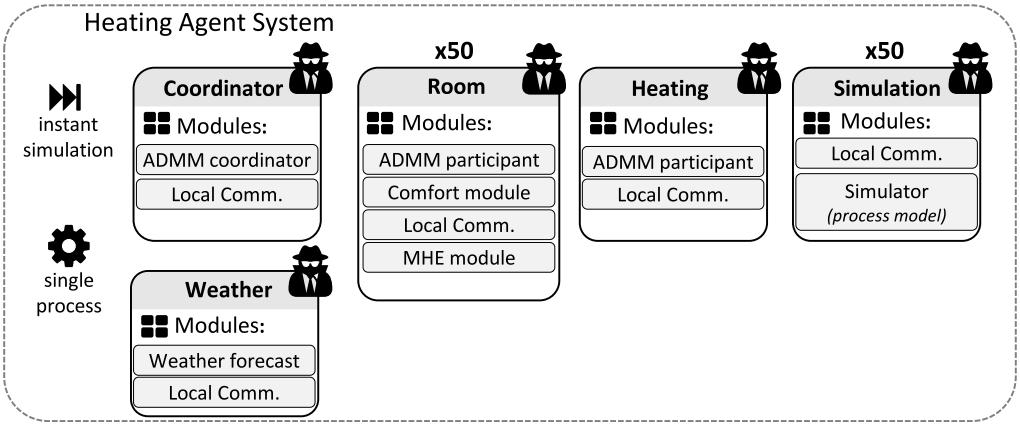


Fig. 12. Agent setup for the multi-room heating control problem.

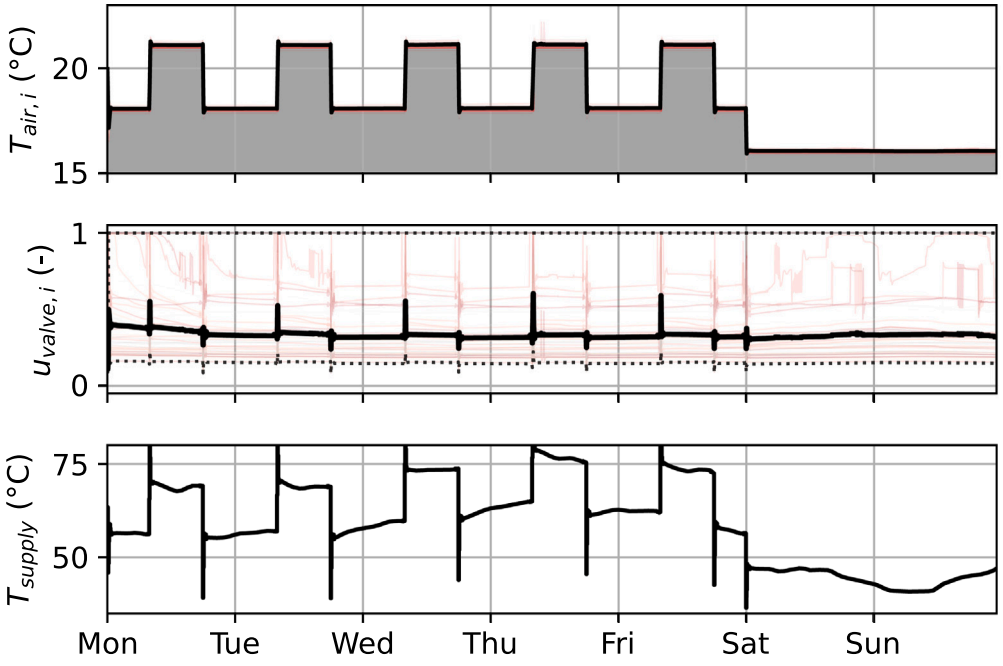


Fig. 13. Simulation results for multi-room study. Black lines show the average of all rooms. The discomfort area is shaded in gray in the top plot. In the plot for the valve position u_{valve} the black solid line is the average of all rooms, while the dotted lines show the minimum and maximum at each point in time.

```python
import logging
from agentlib.utils.multi_agent_system
    import LocalMASAgency
from create_configs_stage_1 import
    make_configs

def main(until=180, log_level=logging.INFO):
    logging.basicConfig(level=log_level)
    env_config = {"rt": False}
    mas = LocalMASAgency(
        # returns list of json configs
        agent_configs=make_configs(),
        env=env_config,
    )
    mas.run(until=until)

if __name__ == "__main__":
    main(until=3600, log_level=logging.INFO)
```

Listing 1: Stage 4. Local Execution

```python
import logging
from agentlib.utils.multi_agent_system
    import MultiProcessingMAS
from create_configs_stage_4 import
    make_configs

def main(until=180, log_level=logging.INFO):
    env_config = {"rt": True}
    mas = MultiProcessingMAS(
        # returns list of json configs
        agent_configs=make_configs(),
        env=env_config,
        log_level=log_level,
    )
    mas.run(until=until)

if __name__ == "__main__":
    main(until=3600, log_level=logging.INFO)
```

Listing 2: Stage 4. Distributed Execution

Fig. 14. Comparison of the main python file for different testing stages.

```
"id": "Simulator-Agent",
"modules": [
{
  "type": "simulator",
  "model": {
    "type": "fmu",
    "path": "models/hil_binary.fmu"
  },

  "t_sample": 5,
  "outputs": [
    {
      "name": "roomBus[1].CO2Con_out",
      "description": "Room 1 CO2 concentration",
      "alias": "urn:fiware:Room1/CO2"
    },
    {
      "name": "roomBus[2].CO2Con_out",
      "description": "Room 2 CO2 concentration",
      "alias": "urn:fiware:Room2/CO2"
    },
    ...
  ],
  "inputs": [
    {
      "name": "conCenFan_oveAct_u",
      "description": "Actuator signal central fan",
      "alias": "urn:fiware:CentralAHU/VDot"
    },
    {
      "name": "disRoo1Occ_oveAct_u",
      "description": "Number of occupants in zone",
      "alias": "urn:fiware:Room1/Persons"
    },
    ...
  ]
},
{
  "type": "mqtt",
  "url": "mqtt://xxx.xx.xxx.x:1883",
  "subscriptions": [
    "room_1",
    ...
}
]
```

Listing 3: JSON config for the simulator agent from stage 2

```
"id": "PLC-AGENT",
"modules": [
{
  "type": {
    "file": "../ads_module.py",
    "class_name": "ADS"
  },
  "plc_ip": "xxx.xx.xxx.xx.x.x",
  "plc_port": "851",
  "t_sample": 3,
  "read_values": [
    {
      "name": "Zones.Zone[1].CO2_concentration",
      "description": "Room 1 CO2 concentration",
      "alias": "urn:fiware:Room1/CO2"
    },
    {
      "name": "Zones.Zone[2].CO2_concentration",
      "description": "Room 2 CO2 concentration",
      "alias": "urn:fiware:Room2/CO2"
    },
    ...
  ],
  "write_values": [
    {
      "name": "SupplyAir.target_volume_flow",
      "description": "Actuator signal central fan",
      "alias": "urn:fiware:CentralAHU/VDot"
    },
    {
      "name": "Zones.Zone[1].number_of_occupants",
      "description": "Number of occupants in zone",
      "alias": "urn:fiware:Room1/Persons"
    },
    ...
  ]
},
{
  "type": "fiware_iota_client",
  "devices": [
    {
      "entity_name": "urn:fiware:CentralAHU",
      "attributes": [
        {
          "name": "VDot",
          "type": "Number",
          ...
}
]
```

Listing 4: JSON config for the PLC Agent from stage 4

Fig. 15. Comparison of agent configurations for different testing stages.

```python
import agentlib
import pandas as pd

class WeatherForecastConfig(agentlib.BaseModuleConfig):
    data_file: str
    ambient_temperature: agentlib.AgentVariable = agentlib.AgentVariable(
        name="t_amb", type="pd.Series"
    )
    t_sample: float = 600  # seconds
    forecast_length: float = 7200  # seconds
    shared_variable_fields: list[str] = ["ambient_temperature"]

class WeatherForecast(agentlib.BaseModule):
    config: WeatherForecastConfig

    def __init__(self, config: dict, agent: agentlib.Agent):
        super().__init__(config=config, agent=agent)
        self.data = pd.read_csv(self.config.data_file)

    def process(self):  # abstract method to implement regular tasks
        while True:
            t_amb = self.get_forecast()
            self.set("t_amb", t_amb)  # sends variable to DataBroker
            yield self.env.timeout(self.config.t_sample)  # returns control back to environment

    def get_forecast(self):  # helper method
        current_time = self.env.time
        end_time = current_time + self.config.forecast_length
        mask = (self.data.index >= current_time) & (self.data.index <= end_time)
        return self.data[mask]

    def register_callbacks(self):  # abstract method to implement asynchronous callbacks
        pass  # this module does not react, it only sends proactively
```

Fig. 16. Implementation of the weather forecast module.

```python
import casadi as ca
from agentlib_mpc.models.casadi_model import *

def compute_q_rad(valve_setpoint, UA_rad, c_water, m_max, T_supply, T_air):
    m = (valve_setpoint + 1e-4) * (m_max)  # add small leakage mass flow for stability
    C = m * c_water
    NTU = UA_rad / C
    effectiveness = 1 - ca.exp(-NTU)
    Q_rad = effectiveness * C * (T_supply - T_air)
    return Q_rad

class RoomModelConfig(CasadiModelConfig):
    inputs: List[CasadiInput] = [
        CasadiInput(name="valve_setpoint", value=0.5),
        ...,  # rest of inputs
    ]
    states: List[CasadiState] = [
        CasadiState(name="T_air", value=20, unit="K"),
        ...,  # rest of states
    ]
    parameters: List[CasadiParameter] = [
        CasadiParameter(name="C_air", value=5000, description="Heat capacity of air"),
        ...,  # rest of parameters
    ]
    outputs: List[CasadiOutput] = [CasadiOutput(name="T_supply_scaled")]

class RoomModel(CasadiModel):
    config: RoomModelConfig

    def setup_system(self):
        Q_as = self.UA_as * (self.T_solid - self.T_air)
        Q_aa = self.UA_aa * (self.T_ambient - self.T_air)
        Q_sa = self.UA_sa * (self.T_ambient - self.T_solid)
        Q_rad = compute_q_rad(
            self.valve_setpoint,
            self.UA_rad,
            self.c_water,
            self.m_max,
            self.T_supply * self.T_scale,
            self.T_air,
        )

        self.T_air.ode = (Q_as + Q_aa + Q_rad) / self.C_air
        self.T_solid.ode = (-Q_as + Q_sa) / self.C_solid
        self.T_supply_scaled.alg = self.T_supply / self.T_scale

        # Soft constraints
        self.constraints = [(self.T_lower, self.T_air + self.T_slack_lower, float("inf"))]

        # Cost function
        objective = self.valve_setpoint * self.r_valve + self.w_slack * self.T_slack_lower**2
        return objective
```

Fig. 17. Implementation of the heating model.

5. Discussion and limitations

The *AgentLib* framework aims to streamline the development and deployment of ACS for energy systems, offering a modular and extensible architecture. While the framework offers significant benefits in terms of flexibility and code reusability, certain limitations exist. This section discusses these aspects, comparing *AgentLib* with existing approaches and highlighting its contributions to the field of energy systems control.

5.1. Advantages of the framework

AgentLib addresses several key challenges in developing and deploying advanced control systems for energy applications, offering practical solutions to issues highlighted in the literature. As discussed by Blum et al. [19], running ACS such as MPC in the real world requires handling multiple sources of data, testing and tuning, and implementing watchdogs to trigger fallback controls. The framework's modularity, combined with its staged development process (Fig. 6), enables a flexible and efficient workflow to deal with these tasks. This is clearly demonstrated by the two distinct use cases presented: the ventilation control and the multi-room heating scenario. In both cases, the core control algorithms and agent structures were readily adapted and reused, significantly reducing development time and effort. Test runs and tuning can be performed locally, before introducing network communication, cloud-connection and real world control. This adaptability is further exemplified by the seamless integration of diverse functionalities within individual agents. For instance, the heating scenario incorporates MHE modules for preprocessing sensor data and a module to create constraint trajectories based on occupancy schedules while listening to a dedicated weather agent. Furthermore, *AgentLib* facilitates the integration of diverse data sources and communication protocols. This is evident in the transition from simulated environments to real-world hardware, where the same agent logic can be used with different communicator modules, swapping a simulated PLC connection for a real one without altering the core control code (see comparison in Fig. 15).

5.2. Scalability

Beyond ease of development and modularity, *AgentLib* provides a foundation for building robust and scalable control systems capable of handling the complexities of real-world energy applications. A key aspect of this is the framework's ability to manage a large number of interacting agents efficiently. The 50-room heating example demonstrates this scalability, showcasing the deployment of a distributed control system with numerous agents, each operating with its own parameters and constraints. Crucially, this was achieved using a consistent agent template and configuration-driven approach, minimizing the development overhead associated with managing a large-scale system. The performance analysis (see Table 2) further reinforces this point, demonstrating the low overhead introduced by the framework for essential tasks like inter-agent communication and data logging, even in local testing environments. This scalability is particularly relevant in the context of energy systems comprising a vast array of interconnected components. Building control systems, smart grids, and district heating networks are prime examples where the ability to manage and efficiently coordinate numerous interacting agents is essential.

5.3. Addressing the unique challenges of energy systems

Energy systems are often highly customized, decentralized, and heterogeneous, integrating diverse components like renewable generation, storage, and flexible loads. This complexity is further compounded by the need to incorporate data from various sources, such as weather forecasts, market prices, and sensor networks. As shown in this study, *AgentLib* is especially suited for applications within BES. That said, it might also be suitable for other applications in energy management.

Consider a number of distributed energy resources (DER) within a virtual power plant, aggregating their capacity and providing grid services. Each DER acts as an independent agent, optimizing its own operation while communicating with a central market agent that aggregates the available power. The local constraints and data sources of each DER can be individual, while potential bidding or optimization strategies – iterative or non-iterative – can be implemented through a common module. While controlled studies on this topic exist [46,47], moving on to live experiments is intimidating, given the real impact on consumers and markets and the need to convince the owners of such facilities to participate.

In another case, consider the coordination of optimal electric vehicle charging, including factors like grid capacity, electricity prices, and individual vehicle needs. Agents representing each vehicle would have access to local constraints, such as current maximum charging speed based on temperature and state of charge or user preference like a required state of charge by a certain time limit. The central charging agent would access current grid conditions and electricity prices, optimally scheduling the available charging power among connected vehicles. Existing studies on this topic perform simulations [48,49], but moving to an experiment will require rigorous pre-testing and confidence in the models and algorithms.

AgentLib would be ideal for such studies, allowing flexible communication with both other agents and the local system through different protocols, while encouraging the extensive pretesting required before a real experiment on such facilities. While *AgentLib* offers advantages for distributed systems common in energy, it might not be the best choice for all applications. For instance, in robotics, real-time control loops often require millisecond-level response times, which might be difficult to achieve with the overhead of a Python-based framework. Similarly, large-scale chemical processes often involve control systems tailored to the specific plant design with high demands on performance and reliability. In these cases, the development cost of a bespoke solution can be justified by the scale of the operation and the criticality of performance.

5.4. Limitations

While *AgentLib* significantly simplifies the development of distributed control systems, some limitations remain. Users still need to familiarize themselves with the framework's core concepts, such as aliases and the callback-based communication of the *DataBroker*. Especially the generation of configuration files contains the main complexity when using *AgentLib*. Debugging in a distributed environment also presents inherent challenges, although *AgentLib* provides tools like single-process execution and granular logging to aid in this process.

Furthermore, while the framework promotes modularity and often allows new functionalities to be added by simply integrating new modules, there are cases where modifications to existing modules might be necessary. For instance, integrating a new module that requires data not currently provided by other modules necessitates updating the relevant sending modules. Similarly, while rerouting or bypassing existing data flows can often be achieved through new modules (e.g., using a watchdog to override MPC outputs), directly influencing the internal behavior of a module (e.g., enabling/disabling the MPC algorithm itself) might require modifications to the module's code. However, the clearly defined interfaces and modular structure of *AgentLib* generally minimize the scope of such modifications, making them relatively straightforward to implement.

Finally, the framework primarily focuses on the software aspects of the control system. Addressing hardware-specific issues, such as actuator availability or communication network reliability, requires external mechanisms. Similarly, model building and parameter estimation are not directly addressed by the framework, although it provides a structured environment for integrating these tasks.

To overcome these limitations, developments in two areas are critical. First, a growing library of maintained modules including templates for specific data sources or protocols, or data-driven modeling will help to build more complex applications in less time. Second, to manage complex MAS, the development of utilities and interfaces for generating, visualizing and validating a number of interconnected configurations is needed.

6. Conclusion and future perspectives

With *AgentLib* we present a modular Python library that accelerates the development and research of controllers and distributed systems for energy applications. By providing an open-source software complete with examples relevant to energy-engineers, we aim to encourage adoption of more ambitious control systems. In particular, *AgentLib* supports researchers to elevate their controllers from an early design stage towards prototypical integration. It provides a standardized definition of modules, creating agents that are modular within themselves, allowing for easy extension and modification of an agent's functionality.

AgentLib includes functionalities for common tasks out of the box, including simulation of FMUs and standard controllers like PID. Modules supporting performant nonlinear MPC based on CasADi are publicly available from the plugin *AgentLib_MPC*. The plugin also contains an implementation of the ADMM algorithm, allowing MPC agents to be executed either standalone or as part of a distributed MPC network. Communicators for connection to the open source IoT-platform FIWARE are released with the plugin *AgentLib_FIWARE*.

Future work will include the release of additional plugins, and the further development of existing ones, with a focus on energy applications. There are also ongoing efforts to develop utility features for the library, such as utilities to configure agents and a graph visualization for MAS. Another field of interest should be the automatic generation of configurations for real systems, possibly based on existing semantic data, or the application of agents as digital twins.

CRediT authorship contribution statement

Steffen Eser: Writing – original draft, Software, Methodology, Investigation, Conceptualization. **Thomas Storek:** Writing – original draft, Software, Methodology, Conceptualization. **Fabian Wüllhorst:** Writing – review & editing, Software, Methodology, Investigation. **Stefan Dähling:** Software. **Jan Gall:** Writing – review & editing, Software. **Phillip Stoffel:** Writing – review & editing, Supervision, Project administration. **Dirk Müller:** Writing – review & editing, Supervision, Project administration, Funding acquisition.

Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work the authors used Claude in order to improve the language of select sections. After using this service, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We gratefully acknowledge the financial support provided by the BMWK (Federal Ministry for Economic Affairs and Climate Action, Germany), promotional references 03ET1495A, 03EN1006A and 03EN1022B.

Appendix. Code examples

The following section provides some example code and configuration. The listings in Fig. 14 show one way to easily start a MAS, with differences between local simulation and distributed real time execution highlighted in the line numbers. Note that it is possible to start distributed agents on separate machines and have them communicate, the `MultiprocessingMAS` starting agents from a single script is merely a utility. The main notable differences between the two are the environment config, the used MAS-utility and the location where the log level is specified.

Of course the configuration of the agents also needs to be adjusted between stages, accounting for different communicators or simulators. Fig. 15 shows the comparison between the simulator config from stage 2, implementing a FMU simulation, with the PLC-connector that replaces it in stage 4. In this configuration, the `type` keyword specifies the location of the module implementation, which can be a string for standard modules (like `simulator`). The configuration can contain fields specific to that module, like an IP-address or the path to the FMU. The modules contain the same variables through which they communicate with the other agents. From the variable definition it can be seen that the names differ, as they need to match the local names within the PLC or the FMU respectively, while the alias (i.e. the public name) is consistent to match what other agents expect. The communicator config at the bottom changes as well, going from the `mqtt` module to the FIWARE communicator.

As an example of how to implement a module, Fig. 16 shows the implementation of the weather module from Section 4. The module consists of a config class and the module implementation. The config defines the *AgentVariable* that is used to communicate and static parameters like the sample time or the file location. In the module implementation, the parameters from the config can be accessed directly, while *AgentVariables* can be accessed through `get` and `set` methods, where the latter will automatically inform other modules (i.e. communicators) of the change. A module always contains the `process` and `register_callback` methods, and can contain any number of helper methods, for example `get_forecast` in this case. For a live weather module, the `get_forecast` might make calls to a weather API, using authentication info from the config, while the rest of the module structure remains unchanged.

Fig. 17 shows the implementation of the temperature controlled room based on Eqs. (15)–(20).

Data availability

The general framework and algorithms of this work are available as open-source code under https://github.com/RWTH-EBC/AgentLib Specific case studies or models are made available upon request.

References

[1] Killian M, Kozek M. Ten questions concerning model predictive control for energy efficient buildings. Build Environ 2016;105:403–12. http://dx.doi.org/10.1016/j.buildenv.2016.05.034.

[2] Afram A, Janabi-Sharifi F. Theory and applications of HVAC control systems – A review of model predictive control (MPC). Build Environ 2014;72:343–55. http://dx.doi.org/10.1016/j.buildenv.2013.11.016.

[3] Serale G, Fiorentini M, Capozzoli A, Bernardini D, Bemporad A. Model predictive control (MPC) for enhancing building and HVAC system energy efficiency: Problem formulation, applications and opportunities. Energies 2018;11(3):631. http://dx.doi.org/10.3390/en11030631.

[4] Killian M, Kozek M. Implementation of cooperative fuzzy model predictive control for an energy-efficient office building. Energy Build 2018;158:1404–16. http://dx.doi.org/10.1016/j.enbuild.2017.11.021.

[5] Drgoňa J, Arroyo J, Cupeiro Figueroa I, Blum D, Arendt K, Kim D, Ollé EP, Oravec J, Wetter M, Vrabie DL, Helsen L. All you need to know about model predictive control for buildings. Annu Rev Control 2020. http://dx.doi.org/10.1016/j.arcontrol.2020.09.001.

[6] Blum D, Arroyo J, Huang S, Drgoňa J, Jorissen F, Walnum HT, Chen Y, Benne K, Vrabie D, Wetter M, Helsen L. Building optimization testing framework (BOPTEST) for simulation-based benchmarking of control strategies in buildings. J Build Perform Simul 2021;14(5):586–610. http://dx.doi.org/10.1080/19401493.2021.1986574.

[7] Storek T, Wüllhorst F, Koßler S, Baranski M, Kümpel A, Müller D. A virtual test bed for evaluating advanced building automation algorithms. In: Proceedings of building simulation 2021: 17th conference of IBPSA, 1-3 Sept, Bruges. Bruges, Belgium; 2021.

[8] Zanetti E, Kim D, Blum D, Scoccia R, Aprile M. Performance comparison of quadratic, nonlinear, and mixed integer nonlinear MPC formulations and solvers on an air source heat pump hydronic floor heating system. J Build Perform Simul 2022;1–19. http://dx.doi.org/10.1080/19401493.2022.2120631.

[9] Blum D, Jorissen F, Huang S, Chen Y, Arroyo J, Benne K, Li Y, Gavan V, Rivalin L, Helsen L, Vrabie D, Wetter M, Sofos M. Prototyping The BOPTEST Framework For Simulation-Based Testing Of Advanced Control Strategies In Buildings. In: Building simulation 2019. Rome, Italy; 2019, p. 2737–44. http://dx.doi.org/10.26868/25222708.2019.211276.

[10] Kim D, Wang Z, Brugger J, Blum D, Wetter M, Hong T, Piette MA. Site demonstration and performance evaluation of MPC for a large chiller plant with TES for renewable energy integration and grid decarbonization. Appl Energy 2022;321:119343. http://dx.doi.org/10.1016/j.apenergy.2022.119343.

[11] Huber B, Bünning F, Decoussemaeker A, Heer P, Aboudonia A, Lygeros J. Benchmarking of data predictive control in a real-life apartment during heating season. J Phys: Conf Ser 2021;2042(1):012024. http://dx.doi.org/10.1088/1742-6596/2042/1/012024.

[12] Lefebure N, Khosravi M, Hudoba de Badyn M, Bünning F, Lygeros J, Jones C, Smith RS. Distributed model predictive control of buildings and energy hubs. Energy Build 2022;259:111806. http://dx.doi.org/10.1016/j.enbuild.2021.111806.

[13] Stoffel P, Berktold M, Müller D. Real-life data-driven model predictive control for building energy systems comparing different machine learning models. Energy Build 2024;305:113895. http://dx.doi.org/10.1016/j.enbuild.2024.113895.

[14] Lin F, Adetola V. Flexibility characterization of multi-zone buildings via distributed optimization. In: 2018 annual American control conference. ACC, Milwaukee, WI: IEEE; 2018, p. 5412–7. http://dx.doi.org/10.23919/ACC.2018.8431400.

[15] Li W, Wang S, Koo C. A real-time optimal control strategy for multi-zone VAV air-conditioning systems adopting a multi-agent based distributed optimization method. Appl Energy 2021;287:116605. http://dx.doi.org/10.1016/j.apenergy.2021.116605.

[16] Li W, Wang S. A multi-agent based distributed approach for optimal control of multi-zone ventilation systems considering indoor air quality and energy use. Appl Energy 2020;275:115371. http://dx.doi.org/10.1016/j.apenergy.2020.115371.

[17] Ceccolini C, Sangi R. Benchmarking approaches for assessing the performance of building control strategies: A review. Energies 2022;15(4):1270. http://dx.doi.org/10.3390/en15041270.

[18] Schmidt M, Åhlund C. Smart buildings as cyber-physical systems: Data-driven predictive control strategies for energy efficiency. Renew Sustain Energy Rev 2018;90:742–56. http://dx.doi.org/10.1016/j.rser.2018.04.013.

[19] Blum D, Wang Z, Weyandt C, Kim D, Wetter M, Hong T, Piette MA. Field demonstration and implementation analysis of model predictive control in an office HVAC system. Appl Energy 2022;318:119104. http://dx.doi.org/10.1016/j.apenergy.2022.119104.

[20] Blum D, Wetter M. MPCPy: An open-source software platform for model predictive control in buildings. United States; 2019.

[21] Lucia S, Tătulea-Codrean A, Schoppmeyer C, Engell S. Rapid development of modular and sustainable nonlinear model predictive control solutions. Control Eng Pract 2017;60:51–62. http://dx.doi.org/10.1016/j.conengprac.2016.12.009.

[22] Wetter M, Zuo W, Nouidui TS, Pang X. Modelica buildings library. J Build Perform Simul 2014;7(4):253–70. http://dx.doi.org/10.1080/19401493.2013.765506.

[23] Andersson JAE, Gillis J, Horn G, Rawlings JB, Diehl M. CasADi: A software framework for nonlinear optimization and optimal control. Math Program Comput 2019;11(1):1–36. http://dx.doi.org/10.1007/s12532-018-0139-4.

[24] Chollet F, et al. Keras. 2015, https://keras.io.

[25] Maier L, Jansen D, Wüllhorst F, Kremer M, Kümpel A, Blacha T, Müller D. AixLib: an open-source modelica library for compound building energy systems from component to district level with automated quality management. J Build Perform Simul 2023;1–24. http://dx.doi.org/10.1080/19401493.2023.2250521.

[26] Bynum ML, Hackebeil GA, Hart WE, Laird CD, Nicholson BL, Siirola JD, Watson J-P, Woodruff DL. Pyomo–optimization modeling in python. 3rd ed.. vol. 67, Springer Science & Business Media; 2021.

[27] Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L, Desmaison A, Kopf A, Yang E, DeVito Z, Raison M, Tejani A, Chilamkurthy S, Steiner B, Fang L, Bai J, Chintala S. PyTorch: An imperative style, high-performance deep learning library. In: Advances in neural information processing systems 32. Curran Associates, Inc.; 2019, p. 8024–35.
[28] EnergyPlus™, version 00. 2017, URL https://www.osti.gov//servlets/purl/1395882.
[29] Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E. Scikit-learn: Machine learning in python. J Mach Learn Res 2011;12:2825–30.
[30] Wetter M, Benne K, Tummescheit H, Winther C. Spawn: Coupling modelica buildings library and EnergyPlus to enable new energy system and control applications. J Build Perform Simul 2024;17(2):274–92. http://dx.doi.org/10.1080/19401493.2023.2266414.
[31] Lubin M, Dowson O, Dias Garcia J, Huchette J, Legat B, Vielma JP. JuMP 1.0: Recent improvements to a modeling language for mathematical optimization. Math Program Comput 2023. http://dx.doi.org/10.1007/s12532-023-00239-3.
[32] Fiedler F, Karg B, Lüken L, Brandner D, Heinlein M, Brabender F, Lucia S. do-mpc: Towards FAIR nonlinear and robust model predictive control. Control Eng Pract 2023;140:105676. http://dx.doi.org/10.1016/j.conengprac.2023.105676.
[33] Merz H, Hansemann T, Hübner C. Building automation. Signals and communication technology, Cham: Springer International Publishing; 2018, http://dx.doi.org/10.1007/978-3-319-73223-7.
[34] Domingues P, Carreira P, Vieira R, Kastner W. Building automation systems: Concepts and technology review. Comput Stand Interfaces 2016;45:1–12. http://dx.doi.org/10.1016/j.csi.2015.11.005.
[35] Dähling S, Razik L, Monti A. Enabling scalable and fault-tolerant multi-agent systems by utilizing cloud-native computing. Auton Agents Multi-agent Syst 2021;35(1):10. http://dx.doi.org/10.1007/s10458-020-09489-0.
[36] Dorri A, Kanhere SS, Jurdak R. Multi-agent systems: A survey. IEEE Access 2018;6:28573–93. http://dx.doi.org/10.1109/ACCESS.2018.2831228.
[37] Bergenti F, Caire G, Monica S, Poggi A. The first twenty years of agent-based software development with JADE. Auton Agents Multi-agent Syst 2020;34(2):36. http://dx.doi.org/10.1007/s10458-020-09460-z.
[38] Bellifemine F, Poggi A, Rimassa G. Developing multi-agent systems with a FIPA-compliant agent framework. Software: Pr Exp 2001;31(2):103–28. http://dx.doi.org/10.1002/1097-024X(200102)31:2<103::AID-SPE358>3.0.CO;2-O.
[39] Müller JP, Fischer K. Application impact of multi-agent systems and technologies: A survey. In: Shehory O, Sturm A, editors. Agent-oriented software engineering. Berlin, Heidelberg: Springer Berlin Heidelberg; 2014, p. 27–53. http://dx.doi.org/10.1007/978-3-642-54432-3_3.
[40] Kravari K, Bassiliades N. A survey of agent platforms. J Artif Soc Soc Simul 2015;18(1):11. http://dx.doi.org/10.18564/jasss.2661.
[41] Palanca J, Terrasa A, Julian V, Carrascosa C. SPADE 3: Supporting the new generation of multi-agent systems. IEEE Access 2020;8:182537–49. http://dx.doi.org/10.1109/ACCESS.2020.3027357.
[42] Association M. Functional mock-up interface 2.0.2. Technical report, Modelica Association; 2020, p. 130.
[43] Cirillo F, Solmaz G, Berz EL, Bauer M, Cheng B, Kovacs E. A standard-based open source IoT platform: FIWARE. IEEE Internet Things Mag 2019;2(3):12–8. http://dx.doi.org/10.1109/IOTM.0001.1800022, arXiv:2005.02788.
[44] Storek T, Lohmöller J, Kümpel A, Baranski M, Müller D. Application of the open-source cloud platform FIWARE for future building energy management systems. J Phys: Conf Ser 2019;1343(1):012063. http://dx.doi.org/10.1088/1742-6596/1343/1/012063.
[45] Boyd S. Distributed optimization and statistical learning via the alternating direction method of multipliers. Found Trends® Mach Learn 2010;3(1):1–122. http://dx.doi.org/10.1561/2200000016.
[46] Yang Q, Wang H, Wang T, Zhang S, Wu X, Wang H. Blockchain-based decentralized energy management platform for residential distributed energy resources in a virtual power plant. Appl Energy 2021;294:117026. http://dx.doi.org/10.1016/j.apenergy.2021.117026.
[47] Hany Elgamal A, Kocher-Oberlehner G, Robu V, Andoni M. Optimization of a multiple-scale renewable energy-based virtual power plant in the UK. Appl Energy 2019;256:113973. http://dx.doi.org/10.1016/j.apenergy.2019.113973.
[48] Li Z, Ma C. A temporal–spatial charging coordination scheme incorporating probability of EV charging availability. Appl Energy 2022;325:119838. http://dx.doi.org/10.1016/j.apenergy.2022.119838.
[49] Xu Z, Hu Z, Song Y, Zhao W, Zhang Y. Coordination of PEVs charging across multiple aggregators. Appl Energy 2014;136:582–9. http://dx.doi.org/10.1016/j.apenergy.2014.08.116.