SoftwareX 35 (2026) 102759

Contents lists available at ScienceDirect

SoftwareX

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

Original Software Publication

TRNSYS-AgentControl: A Python-based framework for agent-driven supervisory control workflows in TRNSYS

Masoud Haddad [a,*], Somayeh Asadi [b], Xiaoshu Lü [a,c]

[a] Renewable Energy and Built Environment, University of Vaasa, P.O. Box 700, FIN-65200, Vaasa, Finland
[b] Department of Civil and Environmental Engineering, University of Virginia, Charlottesville, VA, United States
[c] Department of Civil Engineering, Aalto University, P.O. Box 11000, FIN-02150, Espoo, Finland

ARTICLE INFO

Keywords:
TRNSYS
Python coupling
Supervisory control
Agent-based optimization

ABSTRACT

This software provides a Python-based workflow for agent-driven supervisory control in TRNSYS. It supports baseline data preparation from TRNSYS-exported datasets, control-environment construction, external agent training, and runtime inference through the TRNSYS Python interface. The framework is intended for researchers and engineers performing energy simulation and analysis in TRNSYS, enabling agent-based optimization in place of conventional rule-based supervisory logic while remaining configurable for project-specific states, actions, rewards, and operational constraints. Although the implementation demonstrates a Deep Q-Network (DQN) controller for building-scale photovoltaic-battery energy management, the workflow is not restricted to a single algorithm and can be extended to Python-based decision agents.

1. Motivation and significance

TRNSYS is widely used for transient simulation of building and energy systems [1], but supervisory control in many TRNSYS applications still relies on rule-based logic, schedule-based operation, or external optimization environments. As a result, researchers and engineers working on energy simulation and analysis have faced practical limitations not only in optimizing supervisory decisions, but also, more recently, in implementing agent-based methods within a simulation environment that remains predominantly structured around rule-based operation. In earlier and more recent workflows, optimization has often depended on external tools and case-specific couplings, including MATLAB-based interfaces [2], GenOpt-assisted TRNSYS optimization studies [3], and TRNSYS couplings with external Python-based optimization workflows [4], which can limit methodological flexibility and make the integration of newer Python-based optimization and control approaches less straightforward. At the same time, reinforcement learning and agent-based control have gained attention in energy system applications more broadly [5]. Related software tools and coupling approaches address parts of this problem and provide different functions, such as Python coupling with TRNSYS, TRNSYS workflow automation, control benchmarking, and model exchange. For example, the TRNSYS Python CFFI interface, implemented through Type 3157 [6], enables Python code and Python libraries to be called from within TRNSYS simulations. The pytrnsys package supports Python-based building, execution, post-processing, plotting, and reporting of TRNSYS simulations [7]. BOPTEST [8] provides a framework for evaluating and benchmarking building control strategies. CityLearn [9] offers a separate Gymnasium-based environment, independent of TRNSYS, for multi-agent reinforcement learning in building energy coordination and demand response. FMI-based workflows [10] support tool-independent model exchange and co-simulation across simulation tools. Some of these tools complement the proposed workflow, while others operate in distinct simulation or benchmarking environments. TRNSYS-AgentControl focuses on existing TRNSYS simulation models, where the physical and transient simulation environment remains in TRNSYS, while Python is used for control-environment construction, external agent training, saved-policy deployment, and runtime inference through the TRNSYS Python connector. Therefore, TRNSYS-AgentControl is positioned as a TRNSYS-oriented workflow layer around existing TRNSYS and Python capabilities. Its main significance is not in introducing a new optimization algorithm, but in providing a structured software workflow that links existing TRNSYS models with Python-based agent control.

The intended use scenario is straightforward. A user first exports simulation outputs from a TRNSYS model, then prepares training-ready

* Corresponding author.
E-mail address: masoud.haddad@uwasa.fi (M. Haddad).

https://doi.org/10.1016/j.softx.2026.102759
Received 16 April 2026; Received in revised form 16 May 2026; Accepted 29 May 2026
Available online 2 June 2026

data and defines project-specific states, actions, rewards, and operational constraints in Python. After agent training, the trained model is manually reintegrated into the TRNSYS Python connector for runtime inference. The workflow keeps the TRNSYS model setup under user control, because users manually connect the Python-based agent to the required TRNSYS inputs and outputs according to the needs of the project and the target optimization problem. The practical motivation for this workflow is consistent with earlier TRNSYS-based energy management and supervisory control studies, including prior application-oriented work by the authors [11].

To support this positioning, Table 1 summarizes the functional relationship between TRNSYS-AgentControl and related tools or workflows. The comparison is functional rather than performance-based, because the listed tools do not all address the same software task.

Table 1
Functional positioning of TRNSYS-AgentControl relative to related tools and workflows.

Tool or workflow	Main scope	Relation to TRNSYS-AgentControl
TRNSYS Python CFFI / Type 3157	Provides a TRNSYS component/interface for calling Python code and Python libraries during TRNSYS simulation.	TRNSYS-AgentControl uses this connector as the runtime deployment interface, while adding the surrounding workflow for data preparation, Python-side control-environment definition, external agent training, saved-policy loading, and action/output mapping around existing TRNSYS models.
pytrnsys	Supports Python-based tasks related to TRNSYS simulation workflows, including model building, execution, post-processing, plotting, and reporting.	TRNSYS-AgentControl is not focused on TRNSYS project generation or reporting; it focuses on agent-driven supervisory control and policy redeployment within an existing TRNSYS simulation workflow.
BOPTEST	Provides standardized emulators, APIs, and performance indicators for evaluating and benchmarking control strategies in building and district-energy applications.	TRNSYS-AgentControl operates in a different setting: it targets project-specific supervisory-control workflows around user-defined TRNSYS models rather than a standardized benchmark platform.
CityLearn	Provides a Gymnasium-based environment for reinforcement-learning research in building energy coordination and demand response.	CityLearn provides a standardized RL environment, whereas TRNSYS-AgentControl focuses on existing TRNSYS simulation models and supports Python-based agent training and redeployment through the TRNSYS Python connector.
FMI-based workflows	Support model exchange and co-simulation across tools through Functional Mock-up Units.	TRNSYS-AgentControl does not package the TRNSYS model as an FMU; the physical and transient simulations remain in TRNSYS, while Python is used for agent training and runtime inference via the TRNSYS Python connector.
Previous TRNSYS optimization couplings	Connect TRNSYS with external tools or case-specific optimization and control workflows.	TRNSYS-AgentControl provides a reusable agent-oriented workflow for data preparation, control-environment definition, external training, and redeployment of a saved policy into TRNSYS, rather than a single case-specific coupling.

2. Software description

2.1. Software architecture

TRNSYS-AgentControl is a lightweight Python workflow for preparing, training, and redeploying supervisory agents for TRNSYS simulation models. The software architecture consists of four main layers: optional baseline data preparation, control-environment definition, agent training, and runtime inference. In the illustrative workflow provided in this repository, a training-ready dataset is used as the input for agent development in Python. An environment module constructs the state representation, action space, and reward logic required for agent training. A trainer module uses this environment to train and save a deployable control model. Finally, a runtime module loads the trained model. It performs inference through the TRNSYS Python connector, where selected state variables are passed from TRNSYS to Python, and the predicted control action is returned to the simulation. Fig. 1 summarizes the overall workflow of TRNSYS-AgentControl, from rule-based TRNSYS simulation and data export to Python-side agent training and reintegration into TRNSYS for agent-based closed-loop operation.

Fig. 2 shows the TRNSYS model used to demonstrate the software workflow, including the location of the Python-based runtime component within the simulation structure. In the illustrated setup, TRNSYS provides the selected state variables at runtime; the Python-side agent evaluates the current state and predicts the supervisory action; and the returned output is reintroduced into the same simulation loop for closed-loop control.

The architecture is intentionally modular and application-configurable. The workflow separates baseline preparation, environment definition, training, and runtime deployment into distinct software components. Modular software organization has also been used in energy-system tools that provide separate computational modules and simulation-related components [12]. Control logic can also be organized outside the simulation model, an approach that has been used in software platforms for building control and co-simulation [13]. Project-specific variables such as states, actions, reward terms, and operational constraints are defined on the Python side, while the overall workflow remains unchanged. This design allows the same structure to be reused across different TRNSYS-based energy applications without requiring automatic modification of TRNSYS project files. In the present implementation, the software is demonstrated with a building-scale PV-battery case, but the architecture is not restricted to a single energy system or agent type. Fig. 3 presents the repository structure of TRNSYS-AgentControl and the organization of its main source modules, example case files, trained model directory, and supporting project files.

2.2. Software functionalities

The first major functionality of TRNSYS-AgentControl is control-environment construction, where state variables, action definitions, and reward logic are assembled into a Python-based environment suitable for agent training. The second functionality is external training of supervisory agents using the provided training-ready dataset and the defined environment. In the present implementation, this functionality is demonstrated with a DQN-based controller, but the workflow is not limited to a single algorithm and can be adapted to other Python-based decision agents.

The third functionality is runtime inference for closed-loop supervisory control in TRNSYS. After training, the saved model can be loaded through the TRNSYS Python connector, where the current simulation state is evaluated, and the returned action is mapped to a control signal for the next simulation step. This enables the comparison of agent-based supervisory control against conventional rule-based operation within

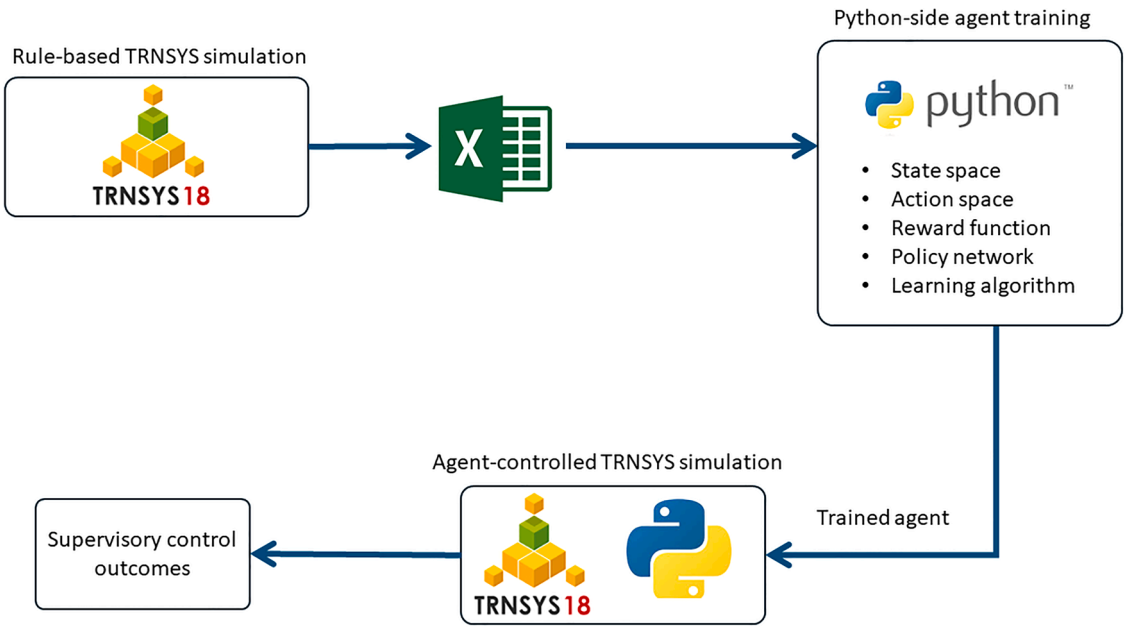

Fig. 1. Overall workflow of TRNSYS-AgentControl, from TRNSYS simulation to Python-side training and reintegration into TRNSYS.

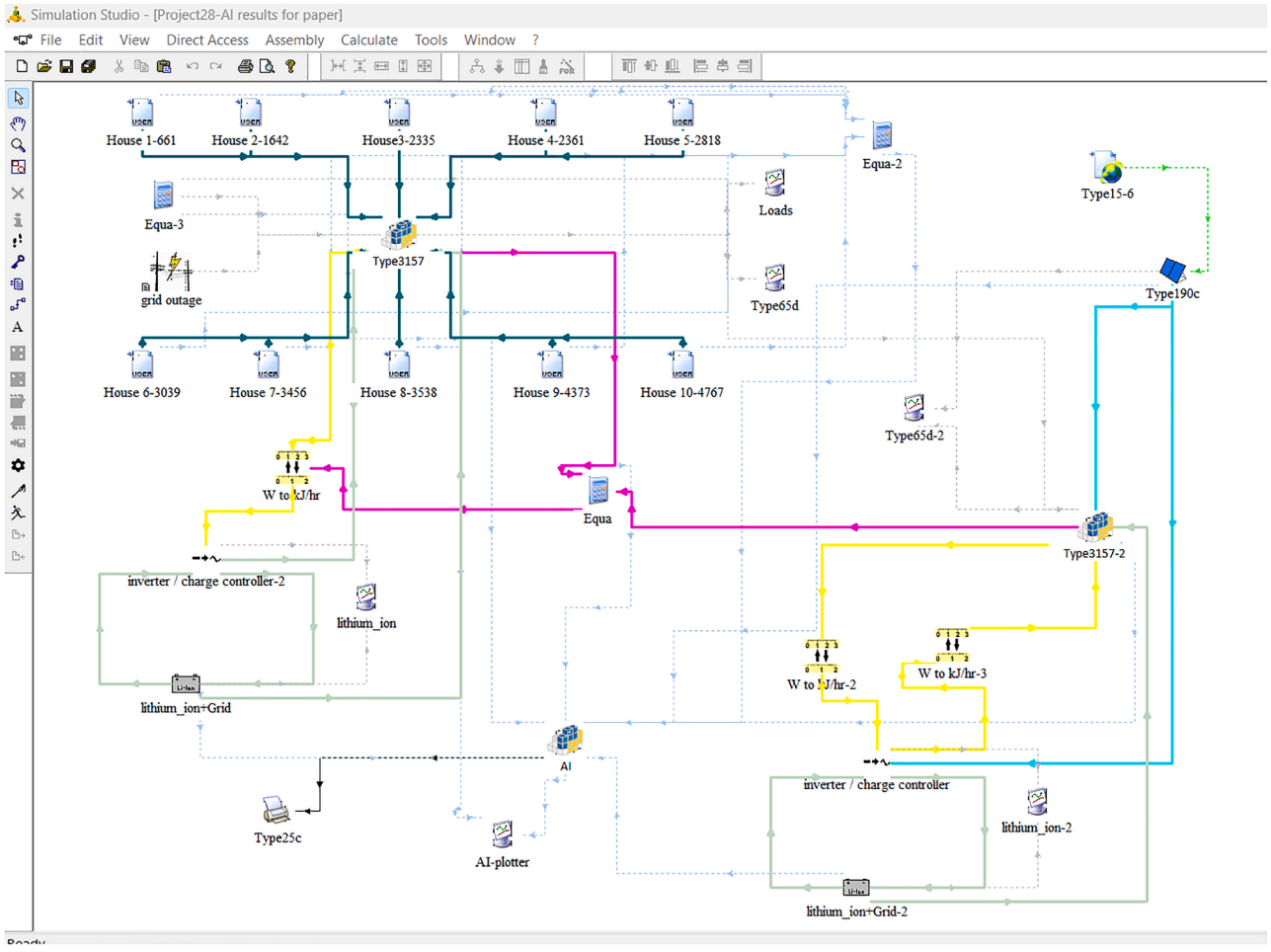

Fig. 2. TRNSYS model for the illustrative PV-battery case, showing the location of the Python-based runtime agent for closed-loop supervisory control.

the same TRNSYS simulation context. A further practical functionality of the software is that it separates external training from in-simulator deployment, which makes the workflow easier to reproduce and adapt while avoiding dependence on direct manipulation of legacy TRNSYS

```
TRNSYS-AgentControl/
├── src/                              # Core Python modules
│   ├── baseline_processor.py         # Rule-based baseline preprocessing
│   ├── env.py                        # RL environment definition
│   ├── trainer.py                    # Agent training workflow
│   ├── runtime_agent.py              # Deployed runtime control logic
│   └── utils.py                      # Utility functions
├── examples/
│   └── pv_battery_case/              # Example case study
│       ├── Meta_data_RuleBased_Result.xlsx
│       ├── best_model.zip
│       └── trnsys_runtime_example.py
├── logs/                             # Training and evaluation logs
├── models/
│   └── dqn_model_sb3.zip             # Trained model
├── README.md                         # Documentation
├── requirements.txt                  # Python dependencies
└── LICENSE.txt                       # License
```

Fig. 3. Repository structure of TRNSYS-AgentControl, showing the main source modules, example case files, trained model directory, and supporting project files.

project structures. To support stable deployment, the workflow preserves consistent input-output mapping, time-step alignment, normalization consistency between training and runtime, action scaling within physical limits, and fallback to baseline control when needed. In addition, the repository includes an optional preprocessing utility that can be used to generate a rule-based comparison dataset when such a step is needed in a project-specific workflow.

3. Illustrative examples

A representative illustrative example is provided for a building-scale PV-battery energy management problem modelled in TRNSYS. In this example, the input data consist of time-series outputs exported from the TRNSYS PV-battery simulation, including the variables selected for the control state and the baseline signals used for comparison. These exported data are converted in Python into a training-ready dataset, where the intermediate information used by the agent consists of aligned time-step records, selected state columns, action mapping, reward-related quantities, and baseline comparison outputs. The Python control environment then represents the TRNSYS-derived PV-battery management problem: at each time step, the agent observes the selected state variables, selects a discrete battery-use action, and receives a reward based on the engineering objective defined for the project. In the present implementation, a DQN-based controller is used to demonstrate the workflow, although the software architecture is not restricted to this algorithm. In TRNSYS-AgentControl, the state vector, action set, reward function, constraints, normalization, and fallback logic are defined on the Python side and are therefore project-specific rather than fixed by the software. For reproducibility, the illustrative PV-battery example uses the configuration summarized in Table 2. This example shows how these elements can be specified for a single TRNSYS model; users can modify the same definitions for other TRNSYS-based applications.

After training, the saved model is manually reintegrated into the TRNSYS Python connector for runtime inference using the Type 3157/CFFI data-exchange structure. In practice, the user defines the required number and order of Python-component inputs and outputs in the TRNSYS connector, connects the Python inputs to the relevant TRNSYS output signals, and connects the Python outputs back to the required control inputs of the TRNSYS components. In the illustrative example, these connector inputs follow the same order as the 15-variable state vector defined above. The runtime script then reads the input array, converts it into the agent state, loads the trained DQN policy, predicts the supervisory action, and maps the selected action to the corresponding battery-use signal. The calculated outputs are then written back to the TRNSYS connector and can be connected to the same downstream control or equation-block locations that would otherwise receive rule-based control signals. In this example, the returned outputs are household grid use, battery use, and the selected action index, in this order. The initialization function prepares the runtime history variables, while the inference step follows the TRNSYS component call sequence so that the control decision remains synchronized with the hourly simulation time step. Together, these steps demonstrate the full workflow from data preparation and external training to in-simulator deployment. The resulting agent-based control can then be compared with a transparent rule-based baseline under the same TRNSYS simulation context. In the illustrative case, annual grid energy consumption was reduced from 473,512 kWh under the rule-based baseline to 385,500 kWh under agent-based control, corresponding to an 18.6% reduction under the tested conditions. The learned policy was evaluated by comparing annual grid energy use with the rule-based baseline within the same TRNSYS simulation context and by verifying that the runtime action mapping remained within the defined battery-use limits. For problem scale, the provided example represents a low-dimensional supervisory-control case with 15 state inputs, five discrete actions, hourly control decisions, and a one-year simulation horizon. Larger applications can be addressed by redefining the project-specific state, action, reward, and constraint settings and by selecting a training algorithm and network size appropriate to the resulting problem dimension. Table 3 summarizes the main example files included in the illustrative PV-battery case and their role in the workflow.

4. Impact

TRNSYS-AgentControl extends the practical use of TRNSYS from conventional rule-based supervisory operation to configurable agent-based control workflows implemented in Python. As a result, the software makes it possible to investigate research questions that are difficult to address with fixed supervisory logic alone, including how different state definitions, reward structures, action mappings, and learning-based or agent-based decision policies influence energy management

Table 2
Control-environment definition used in the illustrative PV-battery example.

Item	Definition in the illustrative example
State vector	The runtime state is a 15-element numerical vector passed from TRNSYS to Python at each time step. The vector follows this input order: TIME, EV_Grid_Usage, PV_Generation, EV_Load, surplus_to_houses_from_EV, EV_Batt_SOC, Num_Ev_Arrivals, total_houses_load, total_surplus, total_deficit, total_pool_coverage, total_battery_supply, battery_SOC, houses_grid_usage, and Total_houses_pv.
Simulation horizon and time step	The example represents a single annual simulation with hourly control decisions; therefore, the simulation horizon is 8760 h, and the control time step is $\Delta t = 1$ h.
Action set	The DQN output is a discrete action mapped to five battery-use fractions: 0, 0.2, 0.4, 0.6, and 0.8.
Action scaling and battery constraint	In the runtime script, the selected action fraction is applied to a battery_SOC-based signal, with the maximum mapped fraction limited to 0.8. This example-specific scaling limits the commanded battery-use signal within the defined runtime mapping.
Reward function	The training reward penalizes grid import and unmet load, with a small penalty on SoC variation to reduce unnecessary battery cycling: $r\_t = -(\alpha G\_t + \beta U\_t) - \varepsilon\lvert\Delta SoC\_t\rvert$. In the example, $\alpha = 1$, $\beta = 50$, $\varepsilon = 0.01$, and $\Delta t = 1$ h.
Tariff assumption	Tariff assumptions are treated as project-specific settings for grid-cost evaluation of simulated grid-import results. Users can define or extend these settings for their own application, including flat-rate or time-based tariffs.
Runtime outputs	Python returns three values to TRNSYS: grid_usage_houses, battery_used, and action_selected.
Fallback control	If the trained policy is not used, the same TRNSYS model can be operated using the rule-based baseline logic/dataset for comparison.
Normalization	Input variables can be normalized in Python to improve learning stability. The same scaling must be used during runtime inference, and control outputs must be mapped back to the physical scale required by TRNSYS components.
Inverter/power limits	Inverter- and power-related limits are represented through the TRNSYS component setup and the battery-control signal used in the runtime mapping. Users can redefine these limits according to their own TRNSYS model configuration.
User configurability	These definitions are specific to the illustrative example. Other users can redefine the state variables, action mapping, reward terms, and constraints according to their own TRNSYS model.

Table 3
Main example files included in the illustrative PV-battery case and their role in the workflow.

File	Role in workflow	Stage
Meta_data_RuleBased_Result.xlsx	Training-ready dataset used for control-environment construction and agent training	Preprocessed training input
dqn_model_sb3.zip	Trained example model produced after external training in Python	Deployment artifact
trnsys_runtime_example.py	Example runtime script used for manual reintegration of the trained agent into the TRNSYS Python connector	Runtime deployment

performance in the same simulation environment [5]. It also supports more flexible experimentation than earlier TRNSYS workflows that relied on external optimization couplings or case-specific optimization setups [2,4].

This structured path supports reproducible comparison of alternative supervisory policies within the same TRNSYS model, while the physical simulation remains in TRNSYS and the control logic is developed externally in Python. At the present stage, the software has been demonstrated through the illustrative PV-battery case included in this article and is aligned with TRNSYS-based energy management applications explored in prior work. Its current impact is therefore methodological and practical rather than widespread in terms of user adoption. No commercial deployment or spin-off activity is claimed in the present work. However, the software provides a reusable basis for future TRNSYS-based studies that require flexible supervisory control and integration of Python-based decision agents.

5. Conclusion

TRNSYS-AgentControl provides a practical Python-based workflow for agent-driven supervisory control studies in TRNSYS. The software supports baseline data preparation, control-environment definition, external agent training, and runtime inference through the TRNSYS Python interface. Its main contribution is not the introduction of a new optimization algorithm, but the provision of a reusable and configurable workflow that helps TRNSYS users move beyond conventional rule-based supervisory logic toward Python-based agent control.

The illustrative PV-battery case demonstrates that the workflow can be implemented from exported TRNSYS data to trained-model redeployment within the TRNSYS simulation loop. More generally, the software offers a structured basis for future energy simulation studies in which supervisory decisions are developed externally and then reintroduced into TRNSYS for closed-loop evaluation.

CRediT authorship contribution statement

Masoud Haddad: Writing – review & editing, Writing – original draft, Visualization, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Somayeh Asadi:** Writing – review & editing, Supervision, Software. **Xiaoshu Lü:** Writing – review & editing, Supervision, Funding acquisition.

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.

Acknowledgements

The authors acknowledge the financial support provided by the Research Council of Finland (SustainSchool Grant No. 359189 and DISCAS Grant No. 362751).

References

[1] SAea Klein. TRNSYS 18: a transient system simulation program. Solar Energy Laboratory, University of Wisconsin; 2017.
[2] Meiers J, Frey G. Interfacing TRNSYS with MATLAB for building energy system optimization. Energies (Basel) 2025;18(2):255.
[3] Haddad M, Javani N, Rezaie B. Energy storage management in a near zero energy building using Li-ion, lead-acid, flywheel, and photovoltaic systems with TRNSYS simulation. Process Saf Environ Prot 2025;196:106898.
[4] Kotegov R, Abokersh M, Mateu C, Shobo A, Boer D, Vallès M. Multi-method optimization of solar district energy systems with battery and thermal energy storage via real-time TRNSYS-Python coupling. Appl Energy 2025;400:126528.
[5] Shen R, Zhong S, Wen X, An Q, Zheng R, Li Y, Zhao J. Multi-agent deep reinforcement learning optimization framework for building energy system with renewable energy. Appl Energy 2022;312:118724.
[6] Bernier N, Marcotte B, Kummert M. Calling Python from TRNSYS with CFFI. Polytech Montréal 2022. https://doi.org/10.5281/zenodo.6523078.
[7] Carbonell D., Birchler D., Hobé A. pytrnsys documentation: a Python-based framework to build, post-process, plot, and report TRNSYS simulations. Available from: https://pytrnsys.readthedocs.io/.
[8] Blum D, Arroyo J, Huang S, Drgoňa J, Jorissen F, Walnum HT, et al. Building optimization testing framework (BOPTEST) for simulation-based benchmarking of control strategies in buildings. J Build Perform Simul 2021;14(5):586–610. https://doi.org/10.1080/19401493.2021.1986574.

[9] Vazquez-Canteli JR, Dey S, Henze G, Nagy Z. CityLearn: standardizing research in multi-agent reinforcement learning for demand response and urban energy management. Available from, https://arxiv.org/abs/2012.10504; 2020.
[10] Blockwitz T, Otter M, Akesson J, Arnold M, Clauss C, Elmqvist H, et al. Functional mockup interface 2.0: the standard for tool independent exchange of simulation models. In: Proceedings of the 9th International MODELICA Conference; 2012. p. 173–84. September 3-5Munich, Germany2012.
[11] Haddad M, Asadi S, Alemazkoor N. Integrated sizing and management of residential energy systems for electric and hydrogen vehicle charging. J Build Eng 2025;116:114675.
[12] Román F, Hensel O. Latent-dry: Solar drying with latent heat storage in Python, Fortran and TRNSYS. SoftwareX 2025;29:101999.
[13] Pallonetto F, Mangina E, Milano F, Finn DP. SimApi, a smartgrid co-simulation software platform for benchmarking building control algorithms. SoftwareX 2019; 9:271–81.