Available online at www.sciencedirect.com

ScienceDirect

IFAC PapersOnline

CONFERENCE PAPER ARCHIVE

IFAC PapersOnLine 56-2 (2023) 9588–9593

A simple framework for working with MATLAB and Home I/O

Javier Jiménez [*] Elena M. Mosquera [*] José M. Maestre [*]

[*] *Higher Technical School of Engineering, Universidad de Sevilla, Camino de los Descubrimientos S/N, 41092, Sevilla, Spain (e-mail: jjsicardo@us.es, emguerrero@us.es, pepemaestre@us.es).*

Abstract: A software framework that communicates MATLAB and Home I/O software, which provides a pedagogic digital twin of a smart home, is presented. This software is packaged as a MATLAB class code and allows reading and writing to Home I/O from MATLAB. It also includes useful tools to simplify the implementation of learning and research algorithms using its built-in methods. A basic thermal identification and a simple MPC (Model Predictive Controller) with all rooms' heaters are built using this framework to prove its functionality.

Keywords: Control education, virtual and remote labs, centralized internet repository, predictive control, system identification.

1. INTRODUCTION

Labs and test facilities are key elements in the education of engineers because they allow students to develop their skills in a controlled environment. Nevertheless, these practical activities are very often limited due to budget or space constraints and students may miss a valuable learning opportunity. Digital twins offer one alternative to relieve this situation because they provide realistic simulation environments where multiple educational activities can be developed. In the case of Control Engineering, digital twins can be a useful resource at all educational stages, from primary (Screpanti et al., 2022) to University level (Čech and Vosáhlo, 2022).

In this paper, we are interested in Home I/O, which is proposed as a pedagogic digital twin of a smart home simulation software (Riera et al., 2020), developed jointly by CReSTIC –from the French University of Reims Champagne-Ardenne– and the Portuguese company Real Games (Riera et al., 2016a,b; Riera and Vigário, 2017; Riera et al., 2019). The pedagogic intent of Home I/O is expressly clear by visually providing a gamified virtual home with its surroundings, allowing the user to interact with all elements of the simulation, e.g., weather conditions, actuators and sensors via the keyboard and mouse or an Xbox-like gamepad controller. Also, it provides programmers with an API that allows to connect and send and retrieve data from these sensors and actuators in addition to other environmental values that can only be read from the simulation data. This API is shipped in form of a DLL (Dynamic Link Library) using Microsoft's .NET Framework (`EngineIO.dll`).

To help disseminate their software, Home I/O developers presented a Simulink interface to interact with the simulation. While this is a very valuable educational resource, a package for interacting with Home I/O via MATLAB scripting commands is not available to the best of our knowledge. To remedy this situation, we have developed a MATLAB class that employs the `EngineIO.dll` library in a transparent way for the user. This way, we provide a much simpler access to this digital twin via scripts and command line. Additionally, this framework avoids referring to the documentation tables to check for the memory addresses of the devices, providing automatic and manual methods to update the memory shared with the simulator.

The class has been implemented in MATLAB r2021b (The MathWorks, Inc., 2021a) and is expected to also work in later MATLAB versions. To illustrate its usage, two different MATLAB example scripts are provided: the first one performs a thermal identification of the Home I/O building by turning on and off all heaters in the house at full power and measuring temperatures in all rooms. After saving the input/output data, a MIMO system identification is performed using MATLAB System Identification Toolbox (The MathWorks, Inc., 2021b). The second example is a basic model predictive controller (MPC) (Camacho and Alba, 2013) using the previously identified model to regulate the temperature in all rooms via the heaters. The MPC is implemented in YALMIP (Löfberg, 2004) and Gurobi (Gurobi Optimization, LLC, 2022) and controls the digital twin in real time through the MATLAB class. This MATLAB class and both examples are available in a GitHub repository (Sicardo et al., 2022).

The remainder of this paper is as follows: Section 2 introduces the Home I/O software and provides information about how it is possible to interact with the digital twin of the smart house. Section 3 presents the MATLAB class that has been developed, which can be downloaded with two examples of code that are commented in Section 4. Final remarks and conclusions are given in Section 5.

[⋆] This work has been funded by the European Research Council (ERC) under the Advanced Grant OCONTSOLAR (grant agreement number 789051), by MCIN/AEI/ 10.13039/501100011033 under project C3PO-R2D2 (ref. PID2020 − 119476RB − $I00$), and by the regional government of Andalusia under project GESVIP (ref. US-1265917).

10.1016/j.ifacol.2023.10.262

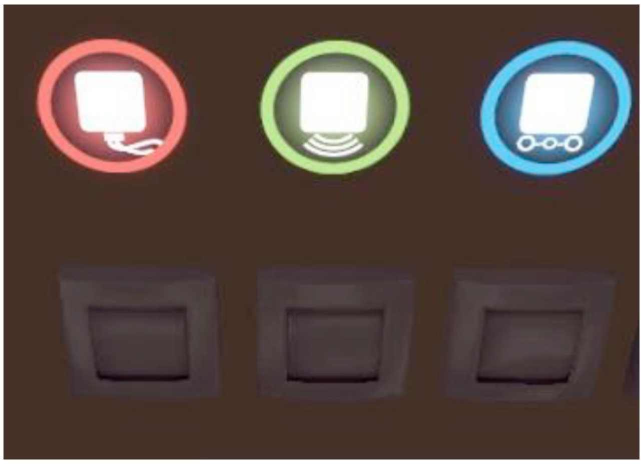

Fig. 1. Three switches in Home I/O showing the different modes (Red: *Wired*, Green: *Wireless*, Blue: *External*)

2. INTRODUCTION TO HOME I/O

Home I/O presents a virtual home with a predefined layout, sensors and actuators, that is fixed. This means that there is no way to extend the simulation itself, and the user is unable to add/remove devices or simulation parameters, or changing their way of functioning. Sensors and actuators are respectively named as *Inputs* and *Outputs*, while the environmental variables and other miscellaneous values are saved in *Memories*. Further details are given in the documentation (Real Games, Unipessoal Lda, 2022a).

While a couple examples regarding energy saving are presented in this paper, Home I/O is meant for a broader training in control engineering than energy concerns, e.g., you can trigger a fire alarm or control a garage door.

2.1 Devices

All devices are distributed throughout the house and its exterior coherently with what an actual home could look like. All the space is sectorized in predefined *Zones* by the documentation, with zones A to N being rooms, and zone O for the exterior of the house). Some devices are not assigned to a zone, e.g., the remote controller device and some simulation parameters in *Memories*.

Additionally, sensor and actuator devices, i.e., *Input* and *Output* devices, have three operation modes (see Figure 1): *Wired* (red color), without automation; *Wireless* (green color), for using with an automation console available via the interface; and *External* (blue color), which is of interest here, where the devices remain visible and available for connection with external technologies via the API. All *Memories* are always visible when connecting to the API.

Through the API, any *Inputs*, *Outputs* and *Memories* can be read, but only *Outputs* can be written. Attempting to write any *Inputs* or *Memories* is silently ignored, because these are either physical sensors or simulation parameters that can only be interacted through the user interface, or are calculated values such as *Zone* temperatures. There are some other objects the user can interact with through the user interface, like doors and windows that can be opened or closed, but their status cannot be retrieved via the API (because they are not sensors or actuators).

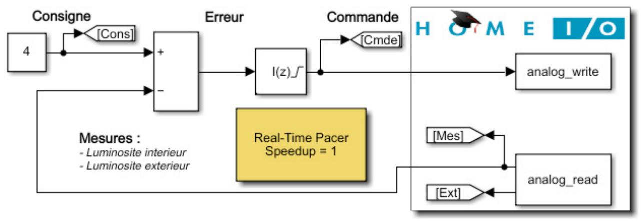

Fig. 2. Simulink model of the light regulation educational resource, in original French language

The full list of devices can be found at the documentation (Real Games, Unipessoal Lda, 2022a), where it is specified the type (*Input*, *Output* or *Memory*), data type (e.g., *Bool*, *Float* and *DateTime*) and memory address, which are required to identify an API data entry. Note also that some devices may work as *Bool* (like a contact) and *Float* (precision value) simultaneously in separate memory entries. Likewise, some *Float* values have a documented operating range of $0 - 10\ V$. Special care should be taken with *Output* entries: any values set out of this range is silently capped within it and, most importantly, the consumed power (W) for that device behaves linear to V up to its maximum power: $W = (V/10) \cdot max\_power$.

Finally, fully exhaustive data loads and saves can be done through XML save files (usually in *Documents*\Home IO\Saves). This save file is not suitable for control purposes, but can be used for setting all devices to *External* mode on a base save via a search and replace operation from *DeviceMode="Wired"* to *DeviceMode="External"*.

2.2 External technologies and educational resources

With all devices of interest in *External* mode, they are available for the Home I/O API and other processes that include the API library file, `EngineIO.dll`. This DLL file is provided in the documentation with code examples for C#, Python and Scratch 2 and 3.

There are also additional examples in the documentation (Real Games, Unipessoal Lda, 2022b) for, e.g., a MATLAB-Simulink implementation for a PID controller to control lights at Zone A via the usage of M-S-Function blocks (see Figure 2, where the simulation has been slightly altered to properly fit here); or using additional applications from the same provider to enable a middleware controller to other additional technologies such as Modbus, KNX, OPC or Siemens PLCs, which are out of the scope of this paper.

2.3 Using the API

The nature of the API is a cached copy of the memory map in the `EngineIO.dll` file. The synchronization of this cached copy with the latest Home I/O data is done on demand with previously fetched memory entries. Thus, a typical recurrent workflow through the Home I/O API would read or write devices and synchronize back with the home simulation data in a programming loop.

These programming actions are well documented and independent of the chosen language (the base is in C#, and the *EngineIO* prefix can be avoided via a smart usage of namespaces in languages that support it):

- Updating the Memory Map:
  EngineIO.MemoryMap.Instance.Update()
- Destroying the Memory Map:
  EngineIO.MemoryMap.Instance.Dispose()
- Retrieving a *Bool* (*Bit*) value:
  EngineIO.MemoryMap.Instance.GetBit(<address>,<memType>)
- Retrieving a *Float* value:
  EngineIO.MemoryMap.Instance.GetFloat(<address>,<memType>)
- Retrieving a *DateTime* value:
  EngineIO.MemoryMap.Instance.GetDateTime(<address>,<memType>)

A list of memory addresses is available at the documentation. Valid *memType*s are:

- for *Inputs*: *EngineIO.MemoryType.Input*
- for *Outputs*: *EngineIO.MemoryType.Output*
- for *Memories*: *EngineIO.MemoryType.Memory*

Instead of updating and manually reading all devices, the API code samples show a more advanced way of reading changed devices via events and handlers, procedures not shown here for brevity.

3. THE FRAMEWORK

The framework is a MATLAB class ready to use (named *HomeIO.m*). Its code is open, available as a GitHub repository (Sicardo et al., 2022), is in active development as of the date of this publication and does welcome suggestions and pull requests to improve and/or extend its functionality. For a proper construction, a *HomeIO* object requires the API DLL file (by default `EngineIO.dll`) and an Microsoft Excel file containing all devices of interest (by default `homeio_full.xlsx`). By this approach, various objectives can be reached in regard to the easiness of use:

- No need to check documentation tables;
- Reduction in human mistakes with addressing;
- Automated updates for the devices (if desired);
- A user-friendly abstraction of the DLL code;
- Checks in place before making the API calls;
- More robustness in expected behaviour.
- Simultaneous reading or writing for multiple devices.

All this also enables to put in practice more advanced techniques on the virtual home without worrying about the API or internals of the inter-process communications.

3.1 Implementation considerations

All devices in Home I/O will be provided to MATLAB in a Microsoft Excel file (`homeio_full.xlsx` by default) with all relevant data saved by columns, including the required three entries to identify a device as mentioned in sections 2.1 and 2.3 (*Memory Type*, *Data Type*, *Address*), plus their *Zone* and *Name* (*Zone* is A to O, or "-" if not applicable; while *Name* is the name as per the documentation), and other values that might apply to some devices: *Contact Type* (applies only to *Input Bool*s, fill with "-" for non applicable entries) and Power (applies only to *Outputs*, fill with 0 for others). The `homeio_full.xlsx` file is provided in the GitHub repository with all device names in English.

A partial list of devices is also acceptable, with these devices being the only ones accessible in that case.

In MATLAB, two additional columns are created on object construction to improve runtime performance. Since all data is doubly classified with their *Memory Type* and *Data Type*, requiring different calls (see section 2.3), a numerical *VarType* column has been calculated as shown in Table 1 (eg an *Input Bool* is *VarType* = 1).

Table 1. *VarType* value from *Memory Type* and *Data Type*

	Bool	*Float*	*DateTime*
Inputs	1	2	3
Outputs	4	5	6
Memories	7	8	9

Then, the other additional column is a unique row identifier, *RowID*, The rows are previously sorted by *VarType* and *Address*. Both new columns allow for significatively faster search speeds.

Synchronization and value updates are made in two different ways each. Regarding synchronization, the user can synchronize manually (calling the *updateHomeIO* method on demand), or configuring an included timer that calls the update method on their behalf, which performs the update operation in the background with each trigger (mentioned in section 3.2).

A first approach for the values updates is just fetching every entry in the devices table, which is considerably slow. However, a listener handler for each *Memory Type* has been implemented. They wait for value changed flags and extract and process only changed devices, contained in the event data between update operations. This saves accessing the API for values that have remained unchanged. The full table update approach remains useful for checking smaller sets of devices in terms of API calls (see section 3.3).

Most methods implemented use name-pair arguments, which can be used to the user's advantage:

- Most optional arguments are configuration arguments, with a default behavior already defined;
- Some methods can also assume a default behavior when called with no arguments, providing custom versatility when arguments are present. For example *readValues* will read all values from Home I/O unless the user specifies a set of devices to read.

3.2 Functionalities

This section is a succinct revision of the attributes and methods for the *HomeIO* class. Extensive documentation is available in the code at the GitHub repository.

A *HomeIO* object has two main table attributes: *BaseData* and *FullData*. Both are similar, but *FullData* is the only table that contains values from Home I/O, for performance concerns. All other attributes are a collection of pre-made classifications for all devices. This makes finding them easier, but a given device may appear in more than one of these collections:

- ***Types*** to classify all devices by *Memory Type* or *Data Type*, or both;
- ***Zones*** to classify all devices by the *Zone* they are physically placed (A to O or "-" for None);
- ***Devices*** to classify all devices by what type of devices they physically are (*Light Switches*, *Brightness Sensors...*) *Heater* and *Lights* have separate entries for their *Bool* and *Float* behaviors;
- ***Special*** has some special categories that could be of particular interest:
  - · *InputsNO* for Normally Open contacts;
  - · *InputsNC* for Normally Closed contacts;
  - · *Inputs10V* for *Inputs* in range $0 - 10\ V$;
  - · *Outputs10V* for *Outputs* in range $0 - 10\ V$;
  - · *ConflictInputs* for *Inputs* that should not coexist simultaneously (such as Up/Down Switches driving the same Roller Shades);
  - · *ConflictOutputs* for *Outputs* that should not coexist simultaneously (such as Roller Shades Up and Down);
  - · *BoolFloatOutputs* for devices that work both in *Bool* and *Float* mode, to prevent conflicts caused by using both at the same time.

There are other two attributes whose purpose is to hold configurations and communications variables, not Home I/O devices:

- *Config*, for the attributes used to save configurations;
- *Comm*, for process communications objects such as listener handles or the timer, in case they are used.

Also, a list of all the methods follows, which have been classified by their functionality:

Object construction These methods are related to the construction of the object:

- ***HomeIO***: Constructor method for the *HomeIO* object. All other methods in this subsection are called only by this method. By default it looks for `EngineIO.dll` and `homeio_full.xlsx` on the same directory, listeners are enabled and timer is disabled;
- ***connectHomeIO***: Function wrapper for linking the `EngineIO.dll` library;
- ***readExcelData***: Reads the list of devices from the Excel file;
- ***changePower10V***: Adapts power for $0 - 10\ V$ *Float Outputs* (divide max power by 10). Makes calculations easier for the *estimatePower* method;
- ***splitData***: Splits all data read from the Excel file in the previously mentioned categories.

Shared memory update For updating the current copy of the *Memory Map*:

- ***updateHomeIO***: Function wrapper for calling the instance update of the *Memory Map*.

Listener and timer event handlers These are called when an event happens to the listeners or the timer, if active:

- ***OnTimerCall***: Triggers when the timer times out. Runs a check on Home I/O then launches an update that will trigger the listener events to refresh all changed values;
- ***OnTimerCallNoListener***: Same as *OnTimerCall*, but manually updating all values by using the *readValues* method;
- ***OnValuesChanged***: Triggered by the listeners when values are changed in Home I/O. Sends all data to the *updateFromEvent* for centralized processing.

Read/Write from Home I/O Methods that directly read or write values to the shared *Memory Map* copy. Must be used sparingly because of the API access times. Remember that the main point is that these methods accept more than one value at once:

- ***updateFromEvent***: Reads and updates *FullData* with the values changed that were sent by the listener trigger functions;
- ***readValues***: Reads and updates *FullData* for a specified list of values from the shared Memory Map. If no list of values is specified, reads the whole list of values available;
- ***setValues***: Sends values to be written in the shared *Memory Map*. Ensures these values are *Outputs* and can sanitize them in the appropriate range (0 or 1 for *Bool*; between 0 and 10 for the $10\ V$ *Float* devices). Also can check for *Bool-Float* conflicts.

Getting existent values These methods do not interact with the API, so they can be used without a significant impact on performance:

- ***getRows***: Gets *full rows* from *FullData* property, from rows given from *BaseData*. Makes the check and calls *getRowsFromRowIDs*;
- ***getRowsFromRowIDs***: Gets *full rows* from *FullData*, from a list of given *RowID*s;
- ***getValues***: Gets *only the Value column* from *FullData* property, from rows given from *BaseData*. Makes the check and calls *getValuesFromRowIDs*;
- ***getValuesFromRowIDs***: Gets *only the Value column* from *FullData*, from a list of given *RowID*s;
- ***estimatePower***: Estimates current instantaneous power consumption for all devices in the simulated home ($W_{total} = \sum_i V_i \cdot Power_i$). Useful when working without connecting to Home I/O.

Check methods Methods used for checking to avoid errors in the future:

- ***checkHomeIO***: Checks if Home I/O is active and running and warns otherwise;
- ***checkMember***: Wrapper of MATLAB's *ismember* for this object's tables;
- ***mustMember***: Wrapper of MATLAB's *mustBeMember* for this object's tables.

Default Behavior change For informing MATLAB about how to save, load and delete *HomeIO* objects:

- ***saveobj***: Saves the configuration to be able recreate the *HomeIO* object later;
- ***loadobj***: Recreates a *HomeIO* object from a previously saved *HomeIO* object;
- ***delete***: Disables special behavior items (listeners, timer) if existent, then unlinks the *Memory Map* to allow proper destruction of the *HomeIO* object.

3.3 Limitations

The main drawback for synchronizing the full list of devices is the time it takes to access the `EngineIO.dll` API, which is slow via MATLAB. Using listener handlers to reduce API calls for unchanged devices requires to dedicate about 6 ms for each call of the *updateFromEvent* method[1]). This happens at least for *Inputs* and *Memories* if no *Outputs* are changed, leaving very little time for other tasks in case of working frame-by-frame, with 16ms per frame @60fps. In practice, this means that there might be difficulties to run Matlab with simulations in Home I/O that are accelerated at more than $\times 500$.

Also, comparison between table rows is slow as the *HomeIO* tables have 9-10 fields, half of them are texts. For this reason and to avoid big speed impacts (others than API access), the *checkMember* and *mustMember* accept table rows and send just the *RowID*s for checking. A basic column vector of device IDs should be sent to the *ismember* and *mustBeMember* functions if skipping the wrappers.

4. CODE EXAMPLES

Two code examples are uploaded in the GitHub page for this framework: a thermal identification of the home and the regulation of the temperature of the simulated house using MPC and MATLAB. This code can be run efficiently in MATLAB and we were able to accelerate the simulation of the house in Home I/O by setting its speed at $\times 500$, with a sample time of 30 simulated seconds.

Also, notation is reversed here: for the control user, an input is a controllable device (which are labelled *Outputs* in Home I/O) and an output become a measurable state (*Inputs* in Home I/O).

4.1 Thermal identification of the home

This example identifies the thermal behavior of the simulated home for all rooms with a heater by employing step response identification methods. In particular, a model for the home can be identified via MATLAB's System Identification Toolbox (The MathWorks, Inc., 2021b) after recording all temperatures from a sequence of steps.

In real-time simulation, the thermal behavior follows a model that takes into account radiation phenomena and the physical properties of building materials (Riera and Vigário, 2017), none of which are modifiable by the user, although they all influence the behavior identified.

Here, we have selected as control inputs all heaters ($n_u = 12$) and as outputs the temperatures in Kelvin of all the corresponding rooms ($n_x = 11$). Also, disturbances are modelled as the outer air temperature and the outside brightness ($n_d = 2$). Likewise, the *Memory* date and time data are saved for convenience and representation ($n_{\text{other}} = 1$). So, the total data being tracked is 26 devices.

Since the thermal dynamics are slow, each heater remains active for two full simulated days, followed by a relaxation period of other two days. This pattern is done for each of the n_u heaters until a final two-day period without any heater active is performed. This results in a total 50 simulation days, or approximately 150 minutes of simulation at $\times 500$ speed. Some diagnostics are also collected to check whether the simulation has been successfully completed.

Once the simulation ends, data are deoffseted and a MIMO system is identified in canonical form[2] as

$$x_{t+1} = A_c x_t + B_c u_t, \tag{1}$$

considering $C_c = I_{n_x}$ and $D_c = 0$. The inputs for identifying the system are the control inputs (heaters) plus the disturbances (outer temperature and brightness); and the outputs are the measured temperatures.

4.2 Model Predictive Control of temperatures

In the second example, the previously developed model is fed to an MPC controller that has been adapted from Camacho and Alba (2013). Here, input matrix B_c of Eq. 1 is split considering the controllable inputs and disturbances:

- $B_u \in \mathbb{R}^{n_x \times n_u}$ is the control input matrix.
- $B_d \in \mathbb{R}^{n_x \times n_d}$ is the disturbance input matrix.

The physical limits of the system are taken into account as constraints:

- $x_{\min} = -offset$ (i.e., $x_{\min} = 0K$ for room temperatures),
- $u_{\min} < u < u_{\max}$ ($0 - 10\ V$ as per documentation).

The MPC optimizes a quadratic stage cost during a horizon of $N_p = 30$ time steps with weighting matrices $Q = I$ and $R = 0.001 \cdot I$, where I is the identity matrix of the corresponding size. Therefore, we have the stage cost:

$$\ell(x(k), u(k)) = (e(k))^T Q(e(k)) + (u(k))^T R(u(k)), \tag{2}$$

where $e(k) = ref(k) - x(k)$.

The optimization problem for the MPC across the prediction horizon and adding the restrictions is:

$$\min_{u(k)\dots u(k+N_p-1)} \sum_{l=0}^{N_p-1} \ell(x(k+i), u(k+i)) \tag{3}$$

subject to

$$x(l+1) = A_c \cdot x(l) + B_u \cdot u(l) + B_d \cdot d(l), \tag{4}$$
$$x_{min} < x(l) \quad l = 1, 2, \dots, N_p, \tag{5}$$
$$u_{min} < u(l) < u_{max} \quad l = 0, 1, \dots, N_p - 1, \tag{6}$$
$$x(0) = x(k), \tag{7}$$

where $x(k)$ represents the current measurement of the state.

The MPC been implemented via YALMIP (Löfberg, 2004) bridge with Gurobi solver (Gurobi Optimization, LLC, 2022) and the constrained optimization problem is solved at each time step following a receding horizon fashion so that only the first optimized input of the sequence is actually applied to Home I/O. Diagnostics are also saved to track the performance in each step.

As a reference for this example, we can see in Figure 3 the result of incrementing all rooms' temperature reference by 10 ºC with respect to the operation point (which is substracted in the plots).

[1] On a Windows 10 Home 64 bits, 16 GB RAM, 11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz system

[2] As Home I/O rooms are named A to N, a subscript is added to state space matrices A_c, B_c, C_c, D_c to avoid ambiguity

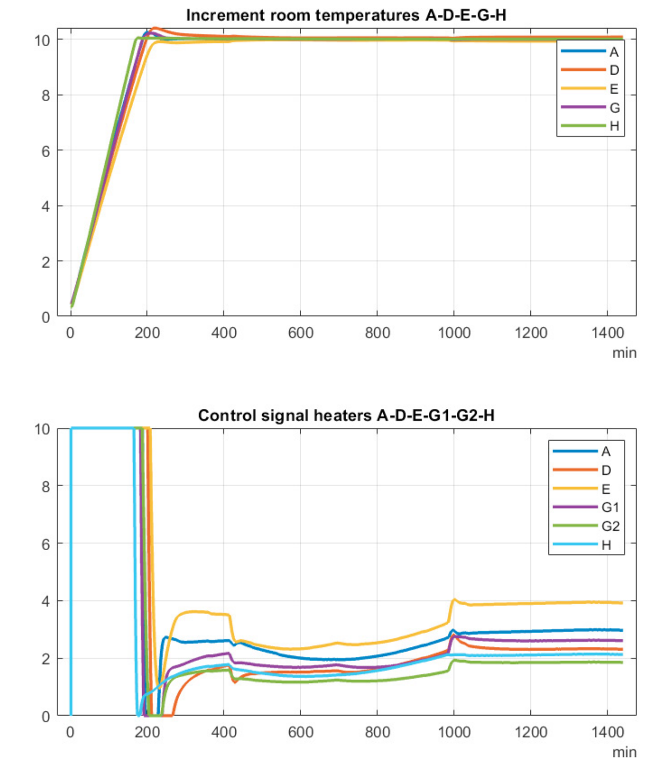

Fig. 3. Temperature evolutions (up) and control signals (down) from the operating point after a 10 ºC step in the reference for rooms A, D, E, G, H. The other rooms are omitted in the plot.

As expected, this MPC control scheme tries to reach the target temperature as soon as possible, neglecting the cost for control efforts since reference tracking has been made more important.

5. CONCLUSION

A MATLAB framework to write scripts that can interact with the simulated smart home provided by the software Home I/O has been presented. This prevents human mistakes by easing the access to simulated devices via code, which allows to exploit the power of MATLAB as a teaching and research instrument.

The proposed framework can be very useful for different courses in engineering degrees. Some potential applications of our framework are the modeling and control of the thermal dynamics a virtual homes, which can be of interest, e.g., for courses in control and smart home technologies.

Nevertheless, a variety of improvements still remain open. For example, in further works, the framework could include the operation at ×5000 speed simulation that Home I/O provides. It could also be interesting to extend the different uses of the framework proposed.

Finally, the code developed can be downloaded freely from a Github repository, with new functions being currently under development.

REFERENCES

Camacho, E.F. and Alba, C.B. (2013). *Model predictive control*. Springer science & business media.

Čech, M. and Vosáhlo, M. (2022). Digital twins and hil simulators in control education–industrial perspective. *IFAC-PapersOnLine*, 55(17), 67–72.

Gurobi Optimization, LLC (2022). Gurobi Optimizer Reference Manual. URL `https://www.gurobi.com`.

Löfberg, J. (2004). Yalmip : A toolbox for modeling and optimization in matlab. In *In Proceedings of the CACSD Conference*. Taipei, Taiwan.

Real Games, Unipessoal Lda (2022a). *Home I/O Documentation*. Gondomar, Porto, Portugal. URL `https://docs.realgames.co/homeio/en/`.

Real Games, Unipessoal Lda (2022b). *Home I/O Documentation (in French)*. Gondomar, Porto, Portugal. URL `https://docs.realgames.co/homeio/fr/`.

Riera, B., Annebicque, D., and Vigário, B. (2016a). Home i/o: an example of human-machine systems concepts applied to stem education. *IFAC-PapersOnLine*, 49(19), 233–238. doi: https://doi.org/10.1016/j.ifacol.2016.10.530. 13th IFAC Symposium on Analysis, Design, and Evaluation ofHuman-Machine Systems HMS 2016.

Riera, B., Emprin, F., Annebicque, D., COLAS, M., and Vigário, B. (2016b). Home i/o: a virtual house for control and stem education from middle schools to universities. *IFAC-PapersOnLine*, 49(6), 168–173. doi: https://doi.org/10.1016/j.ifacol.2016.07.172. 11th IFAC Symposium on Advances in Control Education ACE 2016.

Riera, B., Philippot, A., and Annebicque, D. (2019). Teaching the first and only logic control course with home i/o and scratch 2.0. *IFAC-PapersOnLine*, 52(9), 109–114. doi: https://doi.org/10.1016/j.ifacol.2019.08.133. 12th IFAC Symposium on Advances in Control Education ACE 2019.

Riera, B., Ranger, T., Saddem, R., Emprin, F., Chemla, J.P., and Philippot, A. (2020). Experience feedback and innovative pedagogical applications with home i/o. *IFAC-PapersOnLine*, 53(2), 17610–17615. doi:https://doi.org/10.1016/j.ifacol.2020.12.2676. 21st IFAC World Congress.

Riera, B. and Vigário, B. (2017). Home i/o and factory i/o: a virtual house and a virtual plant for control education. *IFAC-PapersOnLine*, 50(1), 9144–9149. doi:https://doi.org/10.1016/j.ifacol.2017.08.1719. 20th IFAC World Congress.

Screpanti, L., Scaradozzi, D., Gulesin, R., and Ciuccoli, N. (2022). Control engineering and robotics since primary school: an infrastructure for creating the digital twin model of the learning class. *IFAC-PapersOnLine*, 55(17), 267–272.

Sicardo, J.J., Mosquera, E.M., and Maestre, J.M. (2022). A simple framework for working with matlab and home i/o. URL `https://github.com/GESVIP/matlab-homeio`.

The MathWorks, Inc. (2021a). *MATLAB version 9.11.0.1809720 (R2021b) Update 1*. Natick, Massachusetts, United States. URL `https://mathworks.com/`.

The MathWorks, Inc. (2021b). *System Identification Toolbox*. Natick, Massachusetts, United States. URL `https://mathworks.com/products/sysid.html`.