July 2018

# A Python-Based Toolbox for Model Predictive Control Applied to Buildings

Javier Arroyo
*KU Leuven, Belgium; EnergyVille, Belgium*, javier.arroyo@kuleuven.be

Bram Van Der Heijde
*KU Leuven, Belgium; EnergyVille, Belgium; VITO, Belgium*, bram.vanderheijde@kuleuven.be

Alfred Spiessens
*KU Leuven, Belgium; EnergyVille, Belgium; VITO, Belgium*, fred.spiessens@vito.be

Lieve Helsen
*KU Leuven, Belgium; EnergyVille, Belgium*, lieve.helsen@kuleuven.be

# A Python-Based Toolbox for Model Predictive Control Applied to Buildings

Javier Arroyo[1,2,3*], Bram van der Heijde[1,2,3], Alfred Spiessens[2,3], Lieve Helsen[1,2]

[1] University of Leuven (KU Leuven), Department of Mechanical Engineering, Leuven, Belgium

[2] EnergyVille, Thor Park, Waterschei, Belgium

[2] VITO NV, Boerentang 200, Mol, Belgium
* Corresponding Author

## ABSTRACT

The use of Model Predictive Control (MPC) in Building Management Systems (BMS) has proven to outperform the traditional Rule-Based Controllers (RBC). These optimal controllers are able to minimize the energy use within building, by taking into account the weather forecast and occupancy profiles, while guaranteeing thermal comfort in the building. To this end, they anticipate the dynamic behaviour based on a mathematical model of the system. However, these MPC strategies are still not widely used in practice because a substantial engineering effort is needed to identify a tailored model for each building and Heat Ventilation and Air Conditioning (HVAC) system.

Different procedures already exist to obtain these controller models: white-, grey-, and black-box modelling methods are used for this end. It is hard to determine which approach is the best to be used based on the literature, and the best choice may even depend on the particular case considered (availability of building plans, Building Information Models (BIM), HVAC technical sheets, measurement data). Nevertheless, the vast majority of researchers prefer the grey-box option.

In this paper a Python-based toolbox, named Fast Simulations (FastSim), that automates the process of setting up and assessing MPC algorithms for their application in buildings, is presented. It provides a modular, extensible and scalable framework thanks to its block-based architecture. In this layout, each of the blocks represents a feature of the controller, such as state-estimation, weather forecast or optimization. Moreover, the interactions between blocks occur through standardized signals facilitating the inclusion of new add-ons to the framework.

The approach is tested and verified by simulations using a grey-box model as the controller model and a detailed Modelica model as the emulator. A time-varying Kalman filter is applied to estimate the unmeasured states of the controller model.

FastSim is developed and used in a research environment, however this automated process will also facilitate the implementation of MPC for different building systems, both in virtual and real life.

*Keywords: Python toolbox, FastSim, model predictive control, MPC, building management system, HVAC*

# 1. INTRODUCTION

The buildings sector is responsible for over 30% of total final energy use of all sectors of the economy worldwide. Despite significant policy efforts to improve energy efficiency in buildings, building energy use has risen by nearly 20% since 2000 (*Building Energy Performance Metrics*, 2015). Model Predictive Control (MPC) is a technique that has proven to increase the energy efficiency of buildings while keeping their indoor comfort (Coninck & Helsen, 2016). These controllers use a mathematical model of the building together with weather and occupancy forecast to optimize the control signal. Concerns have shifted from whether MPC will maintain thermal comfort at a reduced cost to how easily and reliably it can be implemented.

Therefore, there is an actual need of tools that facilitate the implementation of efficient controllers in buildings and that allow a fair comparison between them. This paper describes the *Fast-Simulations-Toolbox* or FastSim, a framework to set-up and assess control algorithms. Section 2 elaborates on the description of FastSim. Section 3 verifies the toolbox with a simulation example where a detailed model written in Modelica is used as an emulator. Finally, Section 4 draws the main conclusions of this paper.

# 2. DESCRIPTION OF THE TOOLBOX

FastSim is a Python based toolbox that helps in the implementation and assessment of control algorithms in buildings. It provides a modular environment for the application and testing of the controllers. Even though several types and variations of control algorithms exist, all of them are interfaced in the same way: at the beginning of each sampling time period $T_s$, measurements are taken from the plant, then the controller processes this information and returns a signal consisting of the controllable inputs that should be applied during that time-step. FastSim standardizes these signals with a pandas data-frame format, which is the most commonly used package for data-handling in Python, allowing the users to easily plug other control or plant blocks without hampering the overall functionality of the simulation.

Within FastSim each block is a class that inherits all attributes from the parent class. At the same time, each class is instantiated inside its parent class. The parent class that holds all the others is called "Simulation". This class instantiates the controller and plant classes, that are going to inherit the attributes from Simulation (simulation time, sampling time, states names, ...). In the same way, the controller and plant blocks may instantiate other sub-classes. This architecture results in two main advantages:

- Each block can use all those attributes down-streamed from its parent class for its functionality, without the need to declare or calculate them again. This ensures that the attributes used in each sub-block are the same as those from the parent.
- Each block is contained inside the parent class, which makes hierarchy much clearer and natural. Moreover, this architecture helps in the organization of the attributes generated during the simulation because they will be stored within the blocks that generated such attributes. Finding and using these attributes is an important task in the post-processing needed for controller assessment.

This paper presents the case with MPC as controller and an interface with Dymola as an emulator model (plant). Figure 1 shows a schematic presentation of this case where it is possible to appreciate that the controller also contains other sub-blocks: a predictor, an observer and an optimizer. These sub-blocks inherit the attributes from the controller ensuring a coherent implementation. In the diagram, $m_k$ are the measurements taken from the plant model at the beginning of the actual interval $k$; $\hat{x}_k$ are the states estimated by the observer; $d_{(k,k+N)}$ are the predicted disturbances during the time horizon $T_h$, i.e. for $N$ future intervals; and $u_k$ is the vector with the control signals to be applied during the actual interval $k$. All these signals act between the blocks in a pandas data-frame format. Figure 2 shows the work-flow of the blocks in a time-line scale. At the beginning of each time step the predictor forecasts the future disturbances and the observer provides an estimation of the actual states to solve an optimization problem according to a chosen objective. The solution of the problem is sent as control signals to the plant that returns a measurement allowing the observer to estimate the new state. The process is then repeated shifting the optimization horizon one time step. Each block of the diagram (see Figure 1) is explained at each of the

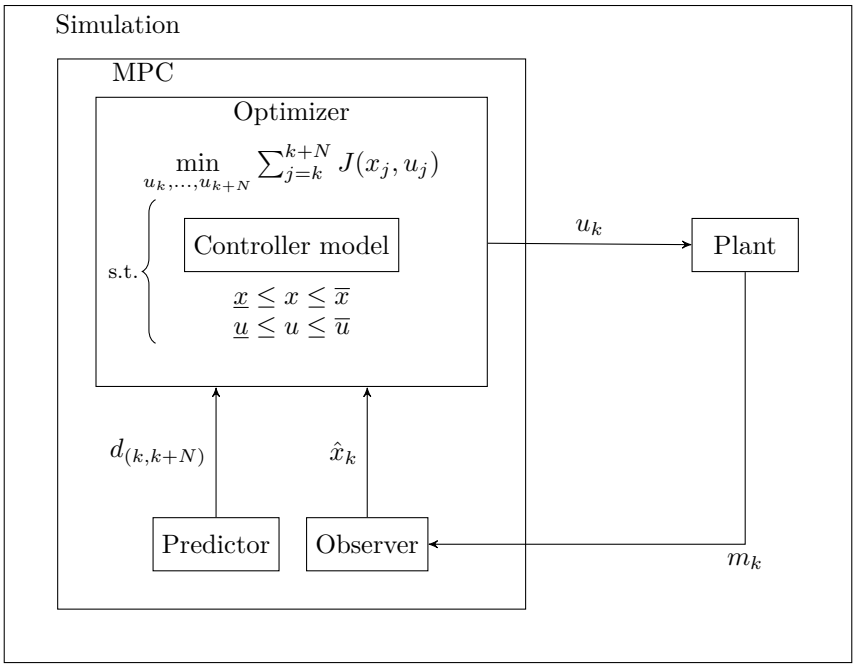


**Figure 1: Block diagram of the MPC**

following sub-sections.

## 2.1 Simulation

The simulation block contains all high-level information like beginning and end time of the simulation, sampling time, variables' names... It also instantiates a controller and a plant block and orchestrates the interaction between them.

## 2.2 Plant

The plant block is an interface with a virtual modelling and simulation environment or with an actual physical building management system. In both cases the plant reads the control signals at the beginning of each time-step and sends the measurements at the end. The tool integrates two plant blocks developed so far.

The first developed plant block uses *Dymola* ("Dymola (Dynamic Modeling Laboratory) Users Manual", n.d.) to emulate the physical behaviour of the building system. For the simulation of the first time-step, a *.mos* file is written. This is a scripting file with the specifications to translate the model and simulate the first interval period. For the following steps, the generated *dymosim.exe* is directly called. This executable file contains all information of the model. Therefore, it enables the simulation from an initial state defined in a text file called dsin.txt and returns the final state into *dsfinal.txt*. Right before each step, the name of the file *dsfinal.txt* is changed to dsin.txt to use the last state vector from the previous time-step as the initial state of the new time-step. The controllable inputs are written in a *dsu.txt* file that is read by *dymosim.exe*.

The second developed plant block uses the Functional Mock-up Interface (FMI) standard (Project, 2018) to emulate the building behaviour from a model in a Functional Mock-up Unit (FMU). The advantage of an FMU is that it can be generated from any modelling environment. In this case, the plant block uses *pyfmi*, which is a Python package with methods to load the FMU, set the initial states, write the inputs and run a simulation (Andersson et al., 2016).

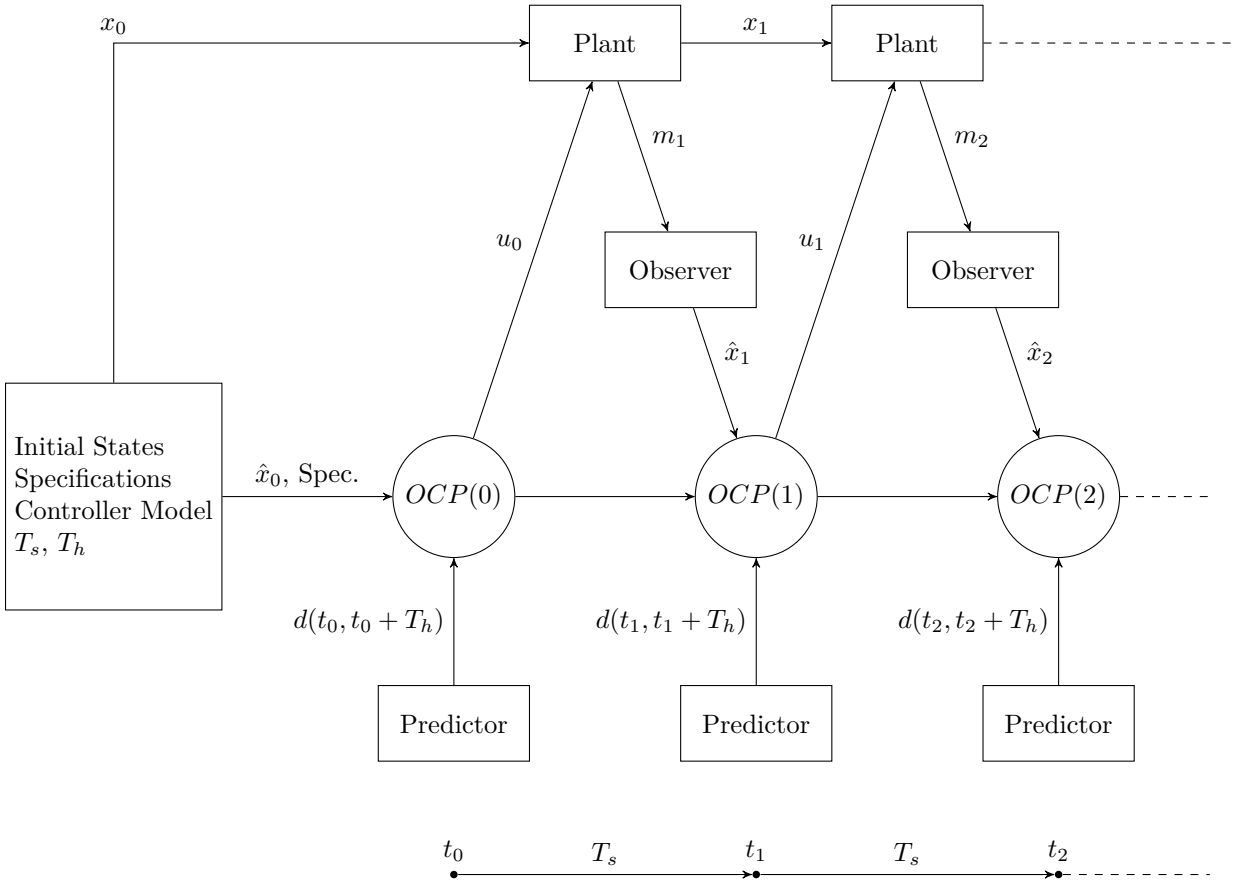


**Figure 2: Work flow of the MPC**

### 2.3 Controller

The main goal of the controller block is to read the measurements from the plant, process information and return the controllable inputs to the plant. Two controllers have been implemented so far: a simple hysteresis controller and an MPC. For the MPC block, a controller model is needed. This is usually a simplified model of the plant, suitable for optimization. This model may also be used by the observer block to estimate the initial states.

The main challenge when developing an MPC controller is related to obtaining a satisfactory controller model (Cigler et al., 2013). For this reason, the developed MPC block contains a package to easily build grey-box models. This type of models use a combination of prior physical knowledge and monitoring data for the estimation of their parameters. They are structured as thermal resistance and capacitance network models with lumped parameters and have demonstrated to be a comprehensive and accurate solution (Coninck & Helsen, 2016; Coninck et al., 2015). The structure of the network is a choice of the designer who can easily build the network connecting nodes with simple commands. The values of the parameters are estimated using the GreyBox Toolbox (Coninck et al., 2015). At any node, the sum of the heat flows flowing into that node is equal to the sum of heat flows flowing out of that node, leading to Eq. 1:

$$\dot{T}_i = \sum_j \frac{T_j - T_i}{C_i R_{ij}} + \sum_k \frac{\dot{Q}_k}{C_i}, \tag{1}$$

where $T_i$ is the temperature of node $i$, $T_j$ is the temperature of node $j$ that is adjacent to $i$; $\dot{T}_i$ is the derivative of the temperature of node $i$; $C_i$ is the thermal capacitance of node $i$; $R_{ij}$ is the thermal resistance between nodes $i$ and $j$; finally, $\dot{Q}_k$ represents each of the heat flows entering the node from internal gains, solar irradiation, radiators... After specifying nodes, resistances, external heat flows and boundary temperatures, the package automatically builds the associated state-space system as shown in Eq. 2.

$$\left.\begin{aligned} \dot{X}_i = \dot{T}_i; \quad \forall i = 1, \ldots, n \\ Y_m = T_m; \qquad \forall m \subset i \end{aligned}\right\} \implies \begin{pmatrix} \dot{X} \\ Y \end{pmatrix} = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} X \\ U \end{pmatrix} \tag{2}$$

In Eq. 2, $n$ is the number of nodes in the network that are represented with the index $i$ and $m$ is the subset of measured states; $U$ are the inputs to the system which concatenates both, the set of controllable inputs, and the set of disturbances; $A, B, C, D$ are the state-space matrices of the system containing all model parameters. These matrices are automatically derived from the RC-model that is used by the controller. It is also possible to introduce the state-space matrices directly into the controller, allowing not only grey-box models but also black-box models and white-box models as controller models as long as they can be represented as state space matrices. A methodology for obtaining linear state space building energy simulation models from white-box models is presented in (Picard et al., n.d.).

Until now, the derivation of the controller equations assumes continuous-time equations. However, the continuous state-space matrix representation of the system is discretized with a finite time-step in order to explicitly calculate the system's states at all intermediate time steps. The discretization method used is the zero-order hold method, which assumes that the control inputs are constant during each time step. For the discretization, the Python Control Systems Library is used (*Control Systems Library for Python*, 2018).

### 2.4 Predictor

The predictor is the MPC block that provides information about relevant external influences (mostly weather conditions) during the optimization horizon. It predicts the disturbance inputs, i.e. outside temperature, solar irradiation, ... The tool integrates a block able to download weather data from (weather forecast webpage, 2018) provided the latitude and longitude coordinates of the building are known. It also integrates blocks to read weather data statically from *.TMY* and *.CSV* files. A block able to predict occupant behaviour has not yet been integrated.

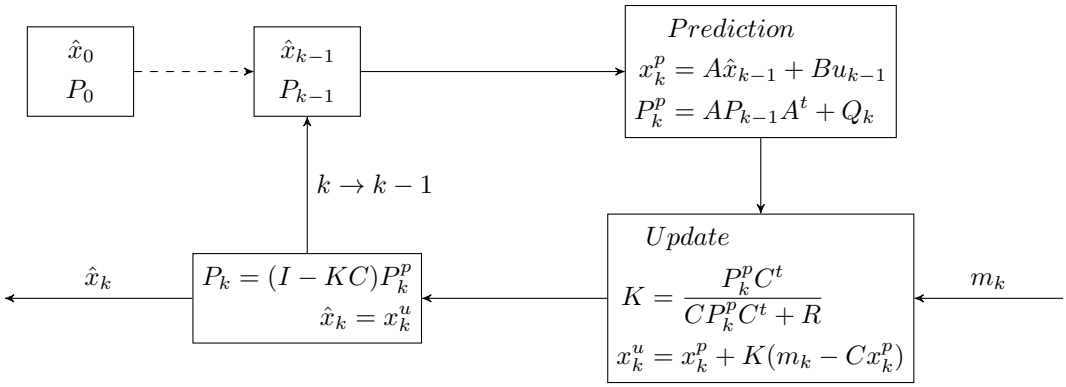


**Figure 3: Block diagram of the time varying Kalman filter**

## 2.5 Observer

The observer estimates the initial states of the system at the actual time-step $\hat{x}_k$ from the previous state estimate $\hat{x}_{k-1}$ and the measurements of the current time-step $y_k$. A time-varying Kalman filter has been developed and is already integrated into the toolbox. It is used for stochastic estimation from noisy sensor measurements. The Kalman filter is essentially a set of mathematical equations that implement a predictor-corrector type estimator that is optimal in the sense that it minimizes the estimated error covariance (Laaraiedh, 2009). It works by assessing the process covariance matrix and accordingly weighting the states estimated by the controller model and the measurements to get a state estimate with the information coming from both sources. The Kalman matrix varies over time, giving more weight to the predicted states or the measurements depending on the process and measurement noise. After some time, a time-invariant matrix should be achieved.

The estimation process is presented in Figure 3. In the first time-step, it is required to initialize the states $\hat{x}_0$ and the covariance matrix $P_0$. The process has two main steps. The first step is the prediction of the new state $x_k^p$ and covariance matrix $P_k^p$. For that purpose, the same state-space matrices of the controller model are used. $u_{k-1}$ is the vector of inputs applied during the previous time-step and $Q$ is the process noise matrix, a diagonal matrix with an estimation of the variance of the model error for each state.

The second step is the update of the Kalman gain matrix, $K$, and state vector $x_k^u$ with the new measurements $m_k$. $R$ is the measurement noise matrix. Note that $K$ is an estimate of the truthfulness of the model compared with the measurements and varies over time. This gain is used to weight the correction factor for the predicted state $x_k^p$. This correction factor is called "innovation" and is the difference between the measurements and the expected values of the measured states obtained with the controller model. Once the update is made, the covariance matrix is also adjusted with the Kalman gain and the process is repeated again for the following time-step.

## 2.6 Optimizer

The optimizer is the algorithm that aims to find the optimal controllable inputs to the building by minimizing the objective function and meeting the constraints associated with the comfort bounds and physical system limitations. The constraints also include the equations of the controller model to include the dynamics of the physical system into the optimal control problem. The objective function penalizes the heating cost and

the discomfort as can be seen in Eq. 3.

$$\min_{x,u} \sum_{k=1}^{N} (u_k + w\delta_k)\Delta t, \tag{3a}$$

$$x_{k+1} = f(x_k, u_k, d_k) \qquad \forall k \tag{3b}$$

$$\underline{x}_k - \delta_k \leq x_k \leq \overline{x}_k + \delta_k \qquad \forall k \tag{3c}$$

$$\underline{u}_k \leq u_k \leq \overline{u}_k \qquad \forall k \tag{3d}$$

$$\delta_k \geq 0 \qquad \forall k \tag{3e}$$

$$\tag{3f}$$

Where $N$ is the number of time steps contained in the prediction horizon, $x$ are all states of the building system that should be kept between the lower and upper comfort bounds: $\underline{x}$ and $\overline{x}$; $u$ stands for the controllable inputs and represents the thermal power used to heat the building; $\underline{u}$ and $\overline{u}$ are the minimum and maximum possible values of the controllable inputs; $\delta$ is the discomfort vector, which represents the lower and upper deviations, respectively, of the actual temperature relative to each comfort bound. This discomfort is weighted with a constant $w$ that accounts for the different orders of magnitude between power and temperature. Note that these temperature deviations are possible because the comfort is introduced as soft constraints into the optimization problem. $f$ is the model of the building that predicts the future states $\boldsymbol{x}$ for a given set of controllable inputs and disturbances $d$.

FastSim automatically builds the optimization problem from the controller model in Pyomo which is a compiler and solver interface for optimization problems in Python (*http://www.pyomo.org/*, 2018). Pyomo can interface with a multitude of optimization solvers, and depending on the solver's capabilities both linear and non-linear problems with or without integer variables can be solved. Bonmin (COIN-OR, 2018) has been chosen as the default solver for the optimization block of the toolbox because of its proven good performance and its possibility to solve mixed-integer non-linear problems.

## 3. A SIMULATION EXAMPLE

A simulation of four winter months has been carried out to verify and test the performance of FastSim. The MPC controller block is used with the time varying Kalman filter and the optimizer presented in the previous section. The plant is an interface with Dymola where a detailed emulator model is running. Each time-step the emulator runs a simulation of one sampling time period that is chosen to be one hour. During this period, the heating inputs are those calculated by the MPC and the weather information is taken from a TMY file. This detailed emulator model is considered to be an accurate representation of a nine-zones residential building. However, to challenge the controller in an even more realistic scenario, the plant adds white noise with a standard deviation of $0.3°C$ to each measurement before passing the measurement vector to the controller.

At the controller side, each time-step the predictor reads the same TMY weather file used by the emulator to take the (perfect) weather forecast during the prediction horizon; then the Kalman filter estimates the initial states of the controller model from the noised measurements; finally, the optimizer estimates the optimal inputs according to this information. These inputs are read by the plan and the process is repeated again.

The lower and upper comfort bounds are time varying, being less restrictive during night-time and during the weekend periods. The only technical constraint is a maximum heat power supply of 2 $kW$ to each zone. No cooling is considered in this example.

### 3.1 Emulator model

The emulator model is used for the virtual representation of a real building. In this example, the emulator model is a detailed white-box model of a nine-zones residential dwelling that has been modelled using the

Modelica IDEAS library (Baetens et al., 2015). The IDEAS (Integrated District Energy Assessment by Simulation) Modelica library is a modelling environment that enables thermal simulation of the building envelop and its HVAC systems.

The emulator model is considered to be an accurate representation of a real building. In this particular example, the model has 426 continuous time states and 26078 differential-algebraic equations defining the heat transfer and other physical processes of the building at one minute sampling time accuracy. However, only the nine air-zones temperatures are being measured and communicated to the controller. The heating inputs are idealized, i.e. we assume that we can inject a desired amount of heat to each zone.

To initialize the emulator model it is run for one day with realistic inputs right before the beginning of the four-months simulation. At the first time-step the observer assigns the value of $20°C$ to the unmeasured states.

### 3.2 Controller model

A decentralised multi-zones grey-box model is identified as a controller model. The model is decentralised because no interaction between zones is included. However, at the end of the identification process, the zones are aggregated and the model is treated as a whole. Centralised models have been tried as well, but they increased the complexity of the overall model without improving its prediction performance.

The month of January is chosen to generate data from the emulator to train and cross-validate the controller model. Thus two sets of data are generated during this period. For the sake of data generation, an hysteresis controller tries to keep the temperature of the building within $21°C$ and $23°C$ by tracking only the temperature of the living room. The controller switches on 3 $kW$ of thermal power when the sensor detects that the temperature is under the lower comfort bound and switches it off when the higher comfort bound is surpassed. The heat is distributed proportionally to the area of each zone. This type of control mimics a traditional, rudimentary rule-based controller and leads to enough level of excitation to obtain rich data.

Once the data is obtained it is split in two weeks for training and two weeks for cross-validation. The GreyBox toolbox (Coninck et al., 2015) is used to estimate the parameters of the RC model. This toolbox enables the estimation of grey-box model parameters in a semi-automated process and from model structures built in Modelica. The model of each zone is identified independently, trying several zone-structures between one and three states and different initial guesses for each. At the end, the attempt that provides the best result in cross-validation is chosen. The controllable input to each zone model is the heating delivered to the zone and the disturbances are the ambient temperature and solar irradiation.

The controllability of the aggregated model has been successfully tested. Then, it has been used as controller model following the steps explained in Section 2.3. The green bars in Figure 4 show the Root Mean Square Error (RMSE) for the obtained model in auto- and cross-validation in units of $°C$. These are the RMSEs of the model for a pure simulation during the two weeks of training data and the two weeks of cross-validation, respectively.

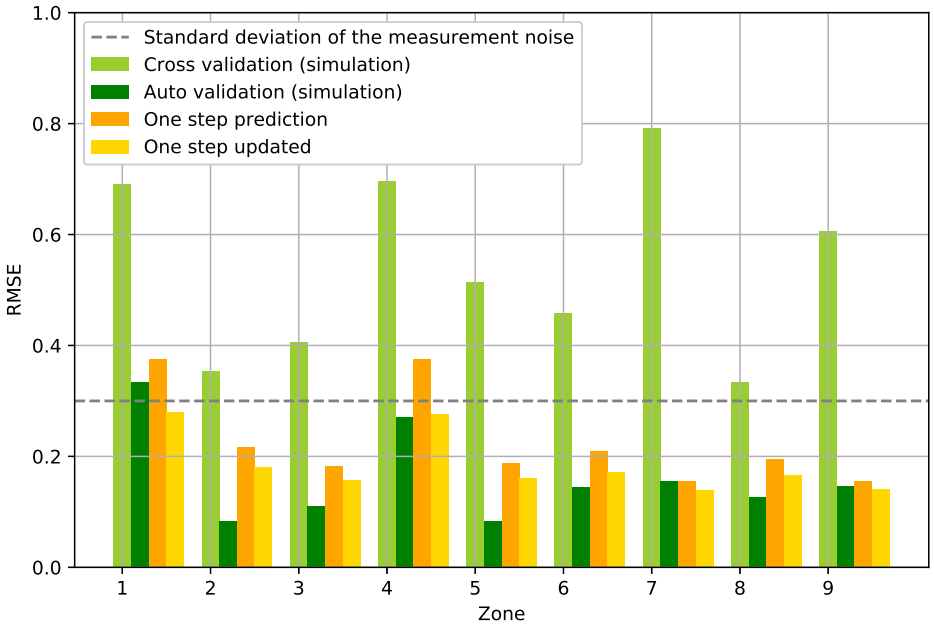


**Figure 4: RMSE of the controller model for each zone temperature for pure simulation of the training and cross-validation period, for the one-step ahead prediction and for the one-step ahead prediction updated with the Kalman filter. The dashed line represents the standard deviation of the white noise associated with the measurement error. The states updated with the Kalman filter are always more accurate than the one-step ahead predictions and the measurements.**

### 3.3 Results

The simulation results prove the good performance of the MPC within the FastSim framework. The first sub-plot of Figure 5 shows the emulated temperatures for the nine zones, the comfort bounds (in green) and the ambient temperature (in blue). The second sub-plot depicts the thermal power released to each zone and the total solar irradiation per square meter (direct plus diffuse). From these graphs, it is possible to appreciate how this controller keeps all zone temperatures at the lower part of the comfort range and takes advantage of the less restrictive periods ensuring comfort at the lowest operational cost.

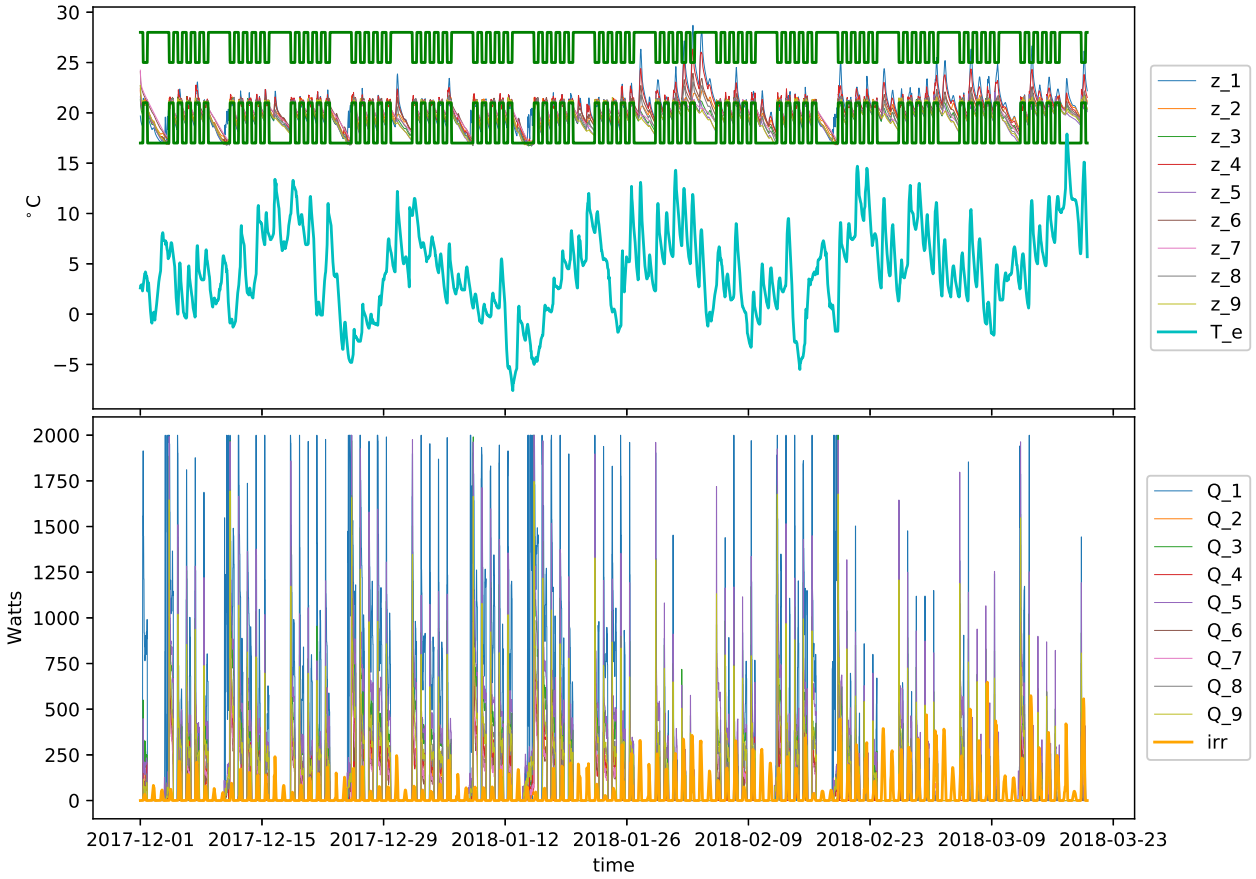


**Figure 5: Evolution of the zone temperature and the heat inputs during four months of simulation. The MPC controller keeps the temperatures of the zones within the comfort bounds while minimizing the energy use.**

Even though the controller model complexity is noticeably lower than the complexity of the emulator model (23 states versus 426), it forecasts the future thermal behaviour of the building with enough accuracy thereby keeping the zone temperatures well within the comfort bounds. The total discomfort for all zones is of 540 Kh for the four months of simulation.

The Kalman filter safeguards the balance between measurement and process error by estimating the initial states of the controller model. In Figure 4 the RMSE for the one-step ahead prediction of the controller model and the RMSE when the Kalman filter updates such predictions with the measurements are also shown. These are the two main steps of the Kalman filter process. From Figure 4 we appreciate that the two-weeks simulation leads to larger prediction errors in cross validation. It is also worth noting that the updated states are always more accurate than the predictions of the controller model and than the measurements that are white-noised around their true value with a standard deviation of $0.3°C$ (see dashed grey horizontal line in Figure 4). This standard deviation is considered a reasonable value for a conventional thermostat.

As explained in sub-section 2.5, the elements of the time-varying Kalman filter vary over time to find a

trade-off between model predictions and measurements. The evolution of these elements' values over the three first days of simulation are depicted in Figure 6, which shows that the filter needs around two days to achieve a time-invariant value of the elements of its matrix gain. Once they converge to their steady values, they remain constant during the rest of the simulation. However, in a real case where the noise is not perfectly white they would keep on changing, adapting themselves over time.

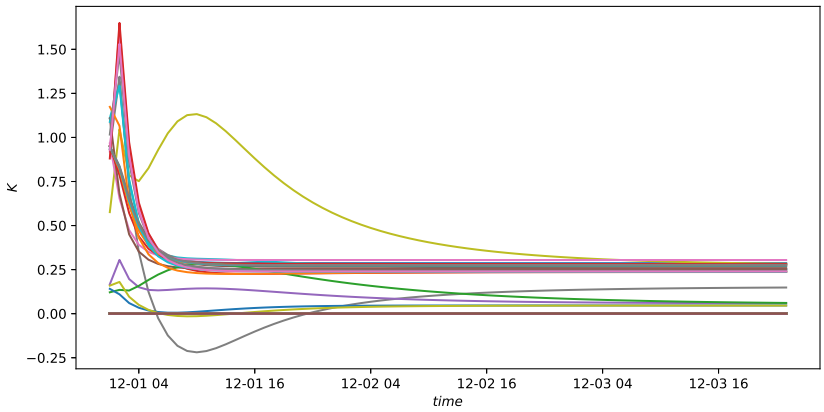


**Figure 6: Evolution of the elements of the Kalman gain matrix for the first three days of simulation. These elements weigh the information coming from the measurements and the one-step ahead predictions of the controller model to estimate the initial states. After two days, the filter converges towards a steady gain that remains constant for the rest of the simulation.**

The results for the first zone (living room) during the first week of December are plotted in Figure 7 to show in more detail the evolution of the temperature. The blue line represents the temperature as emulated by the emulator model, i.e. what is considered in this simulation example the real temperature value. The blue triangles are the measurements as given to the controller model. They do not coincide with the blue line. This is due to the measurement error that has been artificially added. The red line is the living room temperature as predicted with the controller model, i.e. its one-step ahead forecast. Finally, the yellow line is the temperature state updated with the Kalman filter from the measurement and model prediction information. Notice that this updated temperature is always between the predictions of the controller model and the measurements.

One benefit of the Kalman filter is that it provides a Gaussian distribution for the state estimates, whose mean is given by $\hat{x}(t)$ and mean square value of the estimation error is given by the trace of the covariance matrix $P$ (Moura & Zhang, 2016). Consequently, the standard deviation around each state estimate is calculated as the square root of the trace of $P$ and enables to draw a confidence interval around $\hat{x}(t)$ that is depicted in the figure with the dashed yellow lines. According to the properties of a Gaussian distribution, we can ensure that the true state should be within such interval with a probability of 68.2% (Ribeiro, 2004). Such covariance enables to pursue robust control strategies, but this has not been implemented in this paper.

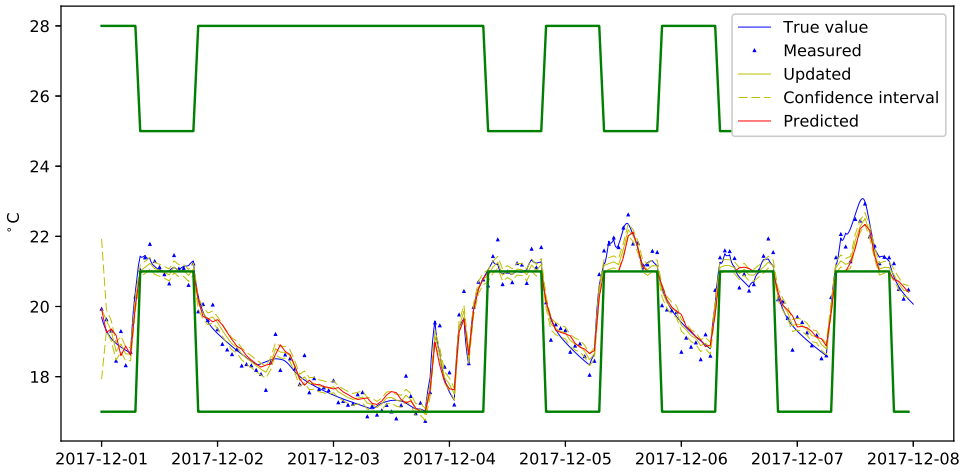


**Figure 7: Evolution of the states of the living room as simulated with the emulator model (true values), predicted with the controller model and updated with the Kalman filter for the first week of simulation. The Kalman update always falls between the measurement and the one step ahead prediction of the controller model.**

## 4. CONCLUSIONS

This paper describes FastSim, a Python-based toolbox that facilitates the implementation and assessment of control algorithms in buildings. More specifically, it automatizes the process of setting-up an MPC controller in a building, enabling a broader implementation of MPC. Moreover, the toolbox has a modular and extensible architecture and is projected to be used not just for testing at simulation level but also for real applications.

FastSim has successfully been tested by simulation for a four months Winter period. In the simulation, a detailed Modelica model plays the role of an emulator from where artificially noise-corrupted measurements are taken. MPC is used with a decentralised multi-zones grey-box model as controller model. This model has demonstrated satisfactory prediction performance even for a simulation of two weeks in cross-validation. In general, the MPC manages to keep thermal comfort within all zones at a low energy cost. A time-varying Kalman filter is used to estimate the initial states of the controller model. The matrix-gain elements of the filter achieve convergence after two days. Moreover, it has been shown that the states updated with the Kalman filter always lead to higher accuracy than the one-step ahead prediction of the controller model and the measurements.

As a conclusion, we can state that the FastSim toolbox is a step forward towards the broad implementation of MPC in practice.

## ACKNOWLEDGEMENT

The authors gratefully acknowledge the financial support of the European Commission in the H2020 programme under Grant Agreement no. 731231 Flexible Heat and Power that has made possible the research leading to these results. The work of Bram van der Heijde is financed by VITO through a PhD Fellowship, and through the project "Towards a Sustainable Energy Supply in Cities", which receives the support of the European Union, the European Regional Development Fund ERDF, Flanders Innovation & Entrepreneurship and the Province of Limburg.

## REFERENCES

Andersson, C., Åkesson, J., & Führer, C. (2016). *Pyfmi: A python package for simulation of coupled dynamic models with the functional mock-up interface* (Vol. LUTFNA-5008-2016) (No. 2). Centre for Mathematical Sciences, Lund University.

Baetens, R., Coninck, R. D., Jorissen, F., Picard, D., Helsen, L., & Saelens, D. (2015). Openideas - an open framework for integrated district energy simulations. *Proc. 14th Conference of International Building Performance Simulation Association, Hyderabad, India, Dec. 7-9*. (`https://github.com/open-ideas`)

*Building energy performance metrics* (Tech. Rep.). (2015). International Energy Agency.

Cigler, J., Gyalistras, D., Siroky, J., Tiet, V.-N., & Ferkl, L. (2013). Beyond theory: the challenge of implementing model predictive control in buildings. *Proc. 11th REHVA World Congress Clima 2013*.

COIN-OR. (2018, April). (`https://projects.coin-or.org/Bonmin`)

Coninck, R. D., & Helsen, L. (2016). Practical implementation and evaluation of model predictive control for an office building in Brussels. *Energy and Buildings*, *111*, 290–298.

Coninck, R. D., Magnusson, F., Akesson, J., & Helsen, L. (2015). Toolbox for development and validation of grey-box building models for forecasting and control. *Journal of Building Performance Simulation*, *9*, 288–303.

*Control systems library for python.* (2018, April). `http://github.com/python-control/python-control`.

Dymola (dynamic modeling laboratory) users manual [Computer software manual]. (n.d.).

Laaraiedh, M. (2009, June). Implementation of Kalman Filter with Python Language. *The Python Papers Journal*.

Moura, P. S., & Zhang, G. D. (2016). *Ce 295: Energy systems and control* (Vol. LUTFNA-5008-2016). University of California, Berkeley.

Picard, D., Jorissen, F., & Helsen, L. (n.d.). Methodology for obtaining linear state space building energy simulation models. *Proceedings of the 11th International Modelica Conference. September 21-23, 2015, Versailles, France*.

Project, M. A. (2018, April). *The functional mock-up interface standard.* (`http://fmi-standard.org/`)

Ribeiro, I. (2004, February). Gaussian probability density functions: Properties and error characterization. , 5.

(2018, April). (`http://www.pyomo.org/`)

weather forecast webpage, D. (2018, April). (`https://darksky.net/dev`)