Integrating Generative Machine Learning Models and Physics-Based Models for Building Energy Simulation

Luigi Vanfretti[1] Christopher R. Laughman[2] Ankush Chakrabarty[2]

[1]ECSE Department, Rensselaer Polytechnic, Troy, NY, USA, vanfrl@rpi.edu
[2]Mitsubishi Electric Research Laboratories, Cambridge, MA, USA, {achakrabarty,laughman}@merl.com

Abstract

This paper describes the integration of generative deep learning models for data-driven building energy simulation. The generative models (GMs) are trained to learn distributions of building input signals from data using Python and PyTorch and interfaced with physics-based Modelica models. The developed integration requirements provide background on typical needs that focus on building energy simulation performance. Simulation examples using models from the Buildings library, refactored to receive GM inputs, are presented to illustrate the benefits of the proposed integration approach and how GMs can be used for building energy performance analysis.

Keywords: Machine learning, generative models, Buildings library, building energy simulation

1 Introduction

Building simulation tools are frequently used during the design phase to size equipment and perform simulation-based studies that help estimate annual energy use or carbon emissions. The demand for such simulation studies, combined with the emergence of new design scenarios such as building electrification, has driven the creation of advanced physics-based building simulation models. The Modelica Buildings library (Wetter, Wangda Zuo, T. S. Nouidui, et al. 2014) is one of the best-known collections of such models, which enable the simulation of the coupled dynamic behavior of building envelopes and heating, ventilation, and air conditioning systems (Chakrabarty, Maddalena, Qiao, et al. 2021; Zhan, Wichern, Laughman, et al. 2022). Modelica-based tools offer distinct benefits in analyzing building performance, as they facilitate systematic controller design (Wetter, Ehrlich, Gautier, et al. 2022) and realistic closed-loop control performance assessment (Stoffel, Maier, Kümpel, et al. 2023).

While such physics-based Modelica models can effectively simulate the energy and mass transfer processes for the building envelope, together with the thermofluid physics of HVAC systems, there are other processes that influence the heating and cooling load that the HVAC system will experience that are not driven by physics alone, but also by human actions. Building occupants generate and absorb latent, sensible and radiant heat, and their actions can significantly impact the efficiency of an HVAC system in terms of energy usage, comfort levels, and indoor air quality, among other factors (Mirakhorli and Dong 2016). As models of such behavior are also required for building design and performance analysis, a common practice is to make engineering assumptions that define a 'nominal' behavior for variables such as occupancy (number of people occupying a zone), activity level and schedule, and then augmenting this nominal model by representing the time-varying behavior as an input disturbance (e.g. a constant or ramp) during simulation. The reliability of simulation outcomes is compromised by such a limited representation of human behavior variables influenced by human behavior, as these variables are challenging to model using physics-based or first-principles methods.

Data-driven approaches have demonstrated their efficacy in characterizing the observed distribution of single-output operational building profiles, such as energy usage (Ye, Strong, Lou, et al. 2022), thermal comfort (Das, Tran, Singh, et al. 2022), and occupancy patterns (Chen and Jiang 2018). A diverse set of building simulation scenarios, including typical or extreme occupancy patterns for a specific building, could be created by integrating these machine learning-based generative models with Modelica-based building models. Such tools could assist in pinpointing improvement opportunities in the existing HVAC system (i.e. retrofitting) or assess the effectiveness of a particular control scheme for existing buildings. and would also enable the creation of data-driven occupancy models to be used in building design when specifying and calibrating the HVAC system to be deployed.

Although explicit neural network (NN) models have previously been created in Modelica (Codeca and Casella 2006), this approach does not enable integration with major ML platforms for NN design (e.g., PyTorch), and attempting it would require a ground-up reimplementation in Modelica. This would require a parallel effort to that in the discipline of machine learning, where advancements in ML platforms are made rapidly and by a large community compared to that of Modelica specialists. Recent efforts have also been made to integrate NNs with physics-based simulators, including the use of the Functional Mock-Up Interface (Modelica Association 2019) to exchange the trained NN model with other frameworks (The MathWorks n.d.). Two approaches of note that

support the use of trained NN models directly in Modelica include the SmartInt (XRG Simulation GmbH n.d.) library, which allows the use of TensorFlow TFLite models (XRG Simulation GmbH n.d.) and the NeuralNet library, which supports the Open Neural Network Exchange (ONNX) format (Wolfram Research n.d.). Unfortunately, limitations in the implementations of both of these libraries make them unsuitable for this work. In particular, the lack of support for NN models trained in PyTorch and the added complexity caused by its dependencies (e.g. TensorFlow API) made it difficult to use the SmartInt library, whereas the NeuralNet library is only available for use in the Wolfram SystemModeler Modelica tool where, as of the time of writing, models from the Buildings library cannot be compiled. Moreover, this tool requires the use of the ONNX Runtime library and its C API, which adds not only complexity but also overhead in software integration.

In this study, we use trained Generative Models (GMs) to provide input signals for building models, such as building occupancy and power demands. These GMs are specified via a set of input parameters and provide causal outputs to the building models, and as such provide valuable "component models" for the overall building representation. While this could have been integrated with the building models via FMI, we chose to build a direct connection between the GMs and Modelica tools to facilitate the iterative refinement of the building models and avoid the inherent trade-offs that model exchange or co-simulation has on simulation performance (Schweiger, Gomes, Engel, et al. 2019). We thus implement a requirements-based light-weight integration of generative models trained in Pytorch with building simulation models in a Modelica environment, which is accomplished via the external function standard interface, as is also done in the SmartInt and NeuralNet libraries. This approach can be of value to other simulation researchers or practitioners as, for simulation purposes, it maintains a minimum number of dependencies and attempts to prioritize simulation performance.

The remainder of this paper is organized as follows. Section 2 lists the requirements considered for the proposed implementation shown in Section 3. The simulation results obtained through this integration approach are illustrated using a model from the Buildings library, and a GM trained in PyTorch using real-world data are presented in Section 4. Conclusions and further work are summarized in Section 5, which concludes the paper.

Conventional Modelica notation is used extensively in this paper. The typewriter font is used along the dot notation to reference the syntax of the Modelica language, including the names of Modelica libraries, models names, etc. Furthermore, the typewriter font is used to refer to the names of other software packages. Meanwhile, the dot notation is used to specify hierarchy in object-oriented modeling. As an example, consider the model of a building containing a zone named zon, which itself is composed of a room named roo. To access a parameter value, for example, the constant convection coefficient for room-facing surfaces of opaque constructions, hIntFixed, the dot notation would be building.zon.roo.hIntFixed. Finally, syntax in code listings follows same as that in the Modelica Language Specification[1].

2 Model Integration Requirements

To combine GM models with building simulation models based on Modelica, various factors have to be taken into account. We outline the scope in three main categories: 1) training and modeling of GM, 2) integration of GM models with building models, and 3) automation of the simulation workflow. Figure 1 illustrates these categories and shows how they interact to support building simulation.

2.1 GM Training and Modeling

An important consideration is that the GM models must be designed to interact seamlessly with the building simulation model. The following requirements (Req.) must be met by the GM models and their integration. These requirements result in the implementation shown in Figure 1(A), which is discussed in Section 3.

Req. 1: Deep generative networks should be easily trained (for example, in PyTorch) with real building data. This requirement emerged from the need for a research-focused framework on machine learning that provides flexibility and ease of experimentation to test new methods such as the one in (Salatiello, Wang, Wichern, et al. 2023). Furthermore, the design of the GM neural architecture should be such that the length of the output (e.g., number of days a signal is generated) can be easily provided as a user-input.

Req. 2: The parameters of the trained generative model must be exchanged in a manner that ensures they are stored in the smallest possible file formats. Reading the files must be fast and efficient, especially because modern GM architectures contain a very large number of trained parameters. In this work, our GM has multiple sub-network components to be trained, but for signal generation (i.e., at inference), only a small sub-module of the deep network is required: therefore, only a small subset of the GM weights need to be stored.

Req. 3: The generative model must be incorporated into the simulation environment and should operate with high computational efficiency. The computational load of running the generative models in conjunction with the building model should be minimal (or insignificant) when compared to running the building model by itself.

[1] See Ch. 1.4 Notation in the Modelica Language Specification: https://specification.modelica.org/master/introduction1.html.

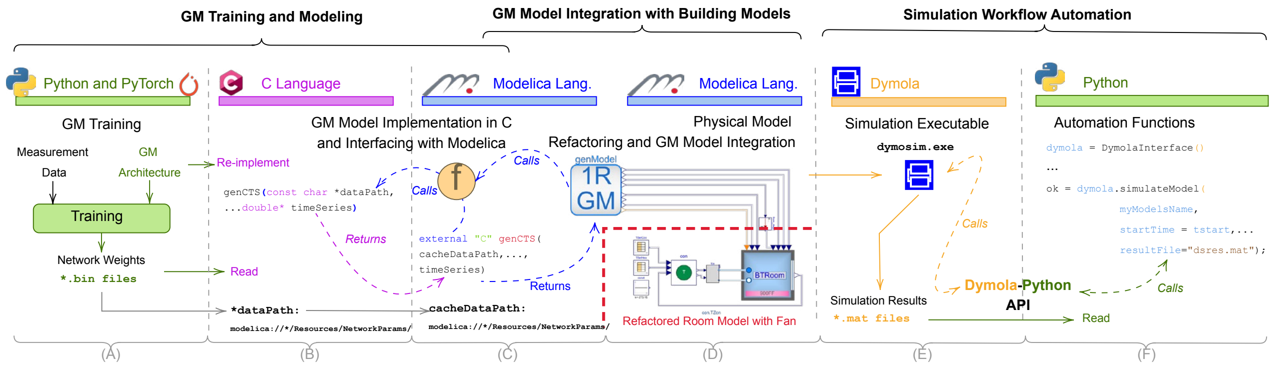

Figure 1. Overview of the Desired Integration of a Generative Model with a Single-Zone Thermal Model of a Building

2.2 Interfacing Deep Generative Networks with Building Simulation Models

Because the building models are implemented in the Modelica language; thus, it is necessary to integrate the GMs with them. To interface them the following requirements are made:

Req. 4: The GMs need to be incorporated into the building simulation model using a tailored `block`. Inputs to the GM should be routed through the Modelica model. Outputs from the GM should be connected via `RealOutput` or `IntegerOutput` interface blocks from the MSL. These interfaces can, for instance, be used to provide the occupancy, equipment, ventilation, and lighting loads of the building.

Req. 5 The GM's output should generate a time series that fills a `CombiTimeTable` from the MSL with predicted variable values at specified time intervals. The lookup table must implement a sample-and-hold mechanism and suitable methods for extrapolation for values outside its defined range for each variable. These features should be included within the `block` described in Req. 4.

Req. 6: The `block` from Req. 4 must also provide the parameters for the deep generative network. In the case of the weights obtained through training, a string parameter will indicate the location of the file(s) storing the weights.

These requirements guided the implementation shown in Figures 1 (see labels (B)-(D)) and 2, which are discussed in Section 3.

2.3 Simulation Workflow Automation

Referring to Figure 1 (see labels (E) and (F)), one final aspect to consider in the scope of this work is that of simulation. As illustrated above label (E), once the GM and building models are integrated, it is possible to create a simulation executable and obtain simulation results. It is beneficial to offer a method for automating the workflow, specifically to alter parameters in the Modelica model programmatically, herein reflected by:

Req. 7: When possible, the simulation executable shall be reused, i.e. limit the re-translation/compilation of the Modelica model, to perform trade-off analysis studies. Such studies shall include changing any of the parameters of the Modelica model, and allow for post-processing of the simulation results.

3 Prototype Implementation

To meet the requirements emerging from the three aspects considered in the previous section, the design choices and implementation pursued are defined next. For illustrative purposes, Figure 1 shows how the implementation was carried out to meet the requirements. A detailed description of the implementation to meet the requirements above labels (B)-(D) in Figure 1, explaining how the interfacing between C and Modelica of the GM models was done, is shown in Fig. 2.

The main goal of the software integration approach used was to minimize dependencies (for both ease of portability and simulation performance purposes) on external software tools other than the C compiler and the Modelica tool, in this case Dymola (Bruck, Elmqvist, Olsson, et al. 2002; Dassault Systemes AB 2023), which requires a `C` compiler itself. Hence, an attractive feature (as discussed in the Introduction) is the use of the standardized external function feature of the Modelica language. Consequently, to minimize dependencies, the integration of GMs with the simulation model must be done solely using `C`. In turn, this requires one to interface the `C` implementation of the GMs with a Modelica model that can call the `C` code. Meanwhile, simulation workflow automation can be achieved by interacting with the Dymola-built simulator executable, which in the case of the approach shown in Figure 1, contains *both* the GM and the building models, with a suitable scripting tool supported by Dymola.

3.1 GM Training and Modeling

Training the NN of the GM was conducted using Python and PyTorch, as illustrated in Figure 1, in the portion over the under-brace labeled with (A), which helps to meet Req. 1. To train the models, measurement data and the designed architecture for the NN are provided in PyTorch to perform the training. This results in the weights of the network, which are stored in `*.bin` files, i.e. binary data, which helps to meet Req. 2.

10.3384/ECP207178 DOI

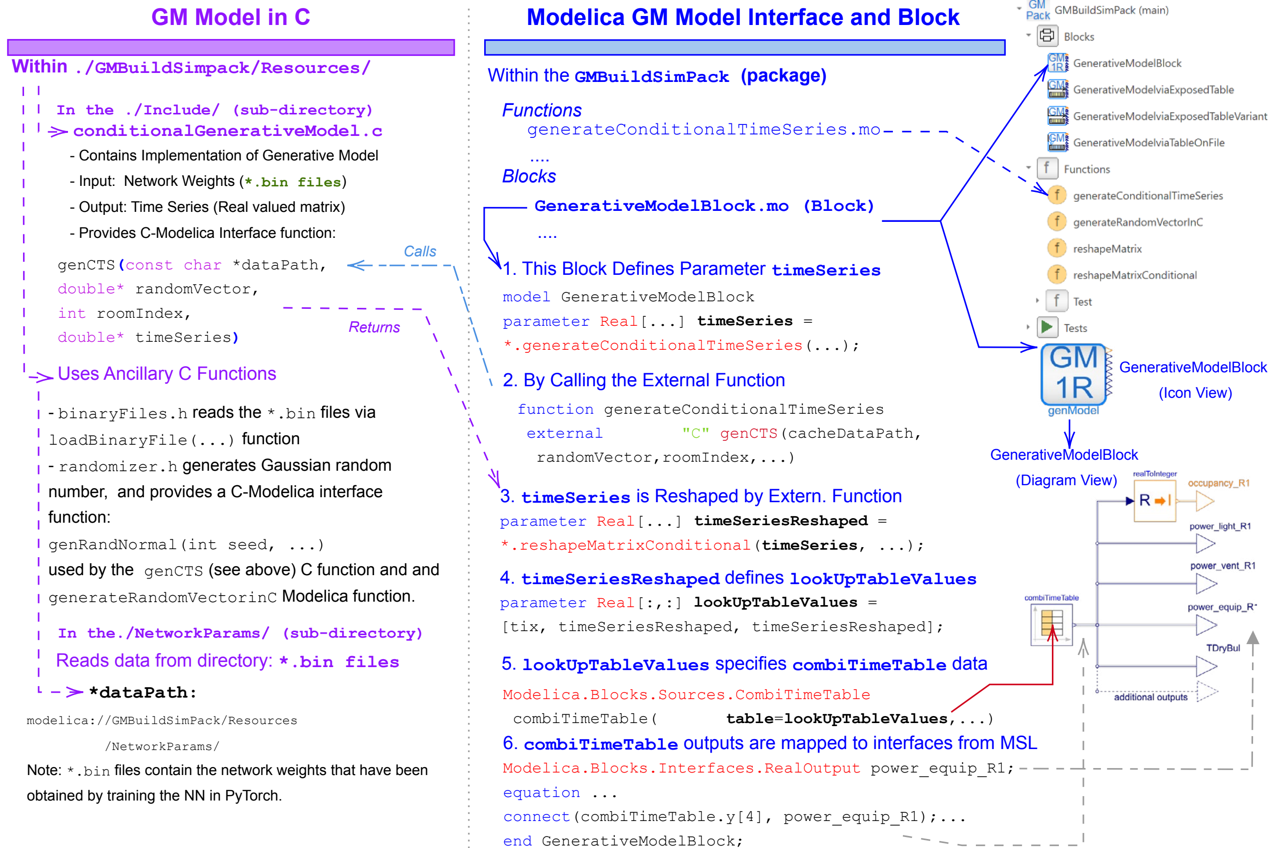

Figure 2. Integration between C and Modelica

To meet the third requirement (Req. 3), the GM needs to be translated to C, as illustrated above label (B) of Figure 1. This implies that the network architecture and different activation layers need to be coded, while the weights resulting from training only need to be read. To illustrate this, consider Listing 1. In lines 2-4, the NN's graph is defined via a structure, i.e., the architecture is defined. Here, only one hidden layer is shown; however, the NN has others (see (Chakrabarty, Vanfretti, et al. 2024)). Next, in lines 6-10, the activation functions are defined, here only the Softplus(x) function is shown; however, other functions are used (see (Chakrabarty, Vanfretti, et al. 2024). Finally, in lines 11-23, the fourth layer's output is calculated using the output of the previous layer, applying the weights for this layer, adding biases, and finally applying the Softplus(x) function.

Listing 1. Excerpt of the GM Implementation in `conditionalGenerativeModel.c`

```
/* Structure to represent network graph */
typedef struct {...
    cLayer hidden4_output;
} cGenerativeModel;
...
/* Sample Activation Functions */
double c_SoftPlus(double x) {
    return log(1 + exp(x));
}
...
/* Sample Network Layer and Output */
double* c_forward(cGenerativeModel network,
    double *input){
...
    // Hidden Layer to Output
    for (int i = 0; i < c_layerSizes[5]; i++) {
        output[i] = 0;
        for (int j = 0; j < c_layerSizes[4]; j
            ++) {
            output[i] += hidden4[j] * network.
                hidden4_output.weights[i *
                c_layerSizes[4] + j];}
        output[i] += network.hidden4_output.
            biases[i];
        output[i] = c_SoftPlus(output[i]);
```

While C-implementation of such kind of NN's is not trivial, this choice was intentionally made taking into account the needs to minimize dependencies and maximize simulation performance. With the proposed approach, the GM model becomes part of the source code of the simulation executable, in Dymola (called `dymosim.exe`, see Figure 1(F)). In addition to this, other C functions are needed, e.g., to load the binary files, and provide other functionalities (see Figure 2). Finally, in addition to implementing the network, the C code needs to include a function to interface with Modelica, as discussed next.

3.2 Interfacing GM Models with Building Models

To fulfill the requirements pertaining this aspect, i.e. Req. 4-6, it is useful to refer to Figure 1, paying

attention to what is presented above the labels (B)-(D). This gives a high-level overview of the main parts of the interfacing. As can be observed, R4 and R5 are fulfilled by invoking a Modelica function, `generateConditionalTimeSeries` (see above label (C) 1), which provides input parameters and invokes the C-function, `genCTS` (see above label (B) 1), evaluating the GM and asking for its output to fill a lookup table called `timeseries`. Meanwhile, to fulfill Req. 6, the `timeseries` output is provided to a table within the `genModel`, which is a `block` that is interfaced with the building model (see above label (D) in Figure 1).

To expand on this overview and understand the integration, refer to Figure 2, which focuses only on the integration between C and Modelica for the GM alone. In the LHS, the C implementation is shown, in the middle the Modelica text is shown, and in the RHS the Modelica graphical views for the developed package and the main block component are shown. This figure should be perused from left to right to understand the implementation details and from right to left to understand how the different pieces work together from a user perspective.

Reading from the right-hand side (RHS), the RHS corner of Figure 2 shows the structure of the `GMBuildSimPack` package. It contains the Modelica function and `block` that help to integrate the GM model with the building. For the user, the `GenerativeModelBlock` would be used to drag, drop, and connect with the building model inputs. When graphically instantiated as `genModel`, as shown by the icon view, one `IntegerOutput` and several `RealOutput` interfaces that route the output of the GM model. This fulfills Req. 4. Meanwhile, the diagram view of this block helps to see them in more detail, and to observe that they are connected to a `CombiTimeTable`, which was what Req. 5 requested. This table will be populated with the `timeseries` output of the GM model, which requires a few steps that are explained next.

Now, to understand how the `CombiTimeTable` gets the GM model data, it is necessary to understand how the Modelica function, `generateConditionalTimeSeries`, interfaces with the C function `genCTS`. This is illustrated in Listing Listing 2 that shows the call to the `external "C"` function in line 5, while also passing input parameters to run the NN (lines 2-4), and obtaining the GM's output in line 5 of the listing, i.e., `Real[384] timeSeries` [2].

Using the `GenerativeModelBlock` in Listing 3, will call `genCTS` in line 7, while providing it with different required parameters, see lines 2-5. The function `generateConditionalTimeSeries` in Listing 3 provides the `timeSeries` output that will pass its data to `lookUpTableValues` in Line 9. Next, in line 7, `CombiTimeTable` is instantiated and data are provided through the `lookUpTableValues` parameter. There are several steps required in this process, which are listed as steps 3 and 4 in Fig. 2. Note that in 10 of Listing 3, several modifiers that are needed have been ommitted; these include those required to set-up the sample-and-hold and periodic extrapolation (i.e. `smoothness=...` and `extrapolation=...` modifiers), and the `timeScale=...` modifier defines the GM's output rate, which will be 15 min in the examples in the forthcoming section. Finally, in line 11, `power_equip_R1` instantiates a `RealOutput` interface that is connected in Line 14 to the corresponding output of the table, i.e. `CombiTimeTable.y[4]`.

[2]Note that in this prototype implementation the size of the output `timeSeries` is fixed to 384 for illustration purposes. In a more generic implementation, this parameter could be propagated to make it easier for the user to modify it.

Listing 2. External Function in Modelica Linking the GM's Output

```
function generateConditionalTimeSeries                              1
  input String cacheDataPath = Modelica.                            2
      Utilities.Files.loadResource("modelica://
      GMBuildSimPack/Resources/NetworkParams/");
  ...                                                               3
  output Real[384] timeSeries;                                      4
  external "C" genCTS(cacheDataPath,                                5
      randomVector, conditionalInputs,
      timeSeries) annotation (
    IncludeDirectory="modelica://GMBuildSimPack/                    6
        Resources/Include",
    Include="#include \"                                            7
        conditionalGenerativeModel.c\"");
end generateConditionalTimeSeries;                                  8
```

Listing 3. Excerpt of the Source Code of the Generative Model Block

```
model GenerativeModelBlock                                          1
    constant String cacheDataPath=Modelica.                         2
        Utilities.Files.loadResource("modelica
        ://GMBuildSimPack/Resources/
        NetworkParams/");
    /* Network Input Parameters */                                  3
    constant Integer latentDim = 8;                                 4
    ... /* Other parameters ommited.                                5
    /* Network Output */                                            6
    parameter Real[nSamplesPerDay*nSignals]                         7
        timeSeries = GMBuildSimPack.Functions.
        generateConditionalTimeSeries(
        cacheDataPath, randomVector,
        conditionalInputs);
    ... /* Ommitting: reshape timeSeries into                       8
        lookUpTableValues */
    Modelica.Blocks.Sources.CombiTimeTable                          9
        combiTimeTable( table=lookUpTableValues
    ... /* other modifiers ommitted */)                             10
    Modelica.Blocks.Interfaces.RealOutput                           11
        power_equip_R1;
    ... /* Other interface instantiations                           12
        ommitted */
equation                                                            13
 connect(combiTimeTable.y[4], power_equip_R1);                      14
 ... /* many connect statmenets ommited */                          15
end ConditionalGenerateTimeSeriesModel;                             16
```

Under this category, only one requirement needs to be addressed, Req. 6. To understand how this is fulfilled, it can be observed in both Listings 2 (see line 2) and 3 (see

line 2) that the string called `cacheDataPath` points to a specific directory where the `*.bin` files are located. It should be noted in this Listing 2 the `annotation` points Dymola to the location where the file that includes `genCTS` is located, so that it can be included as part of the integrated simulator code.

Finally, additional Modelica and C functions are built to support the workflow. For example, the NN must be initialized with a random vector, this is done through ancillary C code whose functions are depicted on the RHS of Fig. 2, one of which has an accompanying Modelica function, i.e. `generateRandomVectorInC`. Meanwhile, as it can be observed in steps 3-4 in Fig. 2, additional Modelica functions (i.e., `reshapeMatrix`), reshape the GM's output before feeding it to the table. We omit further details about the integration of these features, as it is similar to that explained for `genCTS` and `generateConditionalTimeSeries`.

3.3 Simulation Workflow Automation

Finally, to facilitate the automation of the simulation workflow and meet Req. 7, the only design choice to make is the selection of one of the available scripting interfaces provided by Dymola. Among the various interfaces, the most attractive is the Python Interface for Dymola (Dassault Systemes AB 2023), an API to execute Dymola commands using a Python program. This choice was made because Python is already being used to train NN models via PyTorch. Using this interface, the models parameters, weather data files, etc., can be specified and used for specific simulation cases.

As shown in Figure 1 the Dymola-Python Inteface allows to change the value of the models parameter within the simulation executable, by instantiating the interface (i.e. `dymola = DymolaInterface()`), and running a simulation through one of its commands (i.e. `dymola.simulateModel(...)`. To avoid the need of retranslating/compiling the model, one option is to first translate the model (see Chp. 1.3 of the Modelica Specification) using the `translate` command of the Dymola-Python interface, which generates the code of the simulator that can simulate the model. Thus, every time that parameters are changed[3] within a look, the model does not need to be translated; i.e. code generator is avoided, reducing time.

4 Results

4.1 Building Models and GM Training Data

4.1.1 Building and System Model

In this section, examples demonstrating the integration of GM and building models will be presented using the system model depicted in Figure 3, which is divided into three parts. The segment labeled (A) is designated for setting the temperature setpoints (`TSetCoo` and `TSetHea`), the segment labeled (B) includes the physics-based and GM models that will be elaborated on later, and finally, the segment (C) carries out calculations to track building performance metrics such as the zone's temperature (`TRoom`).

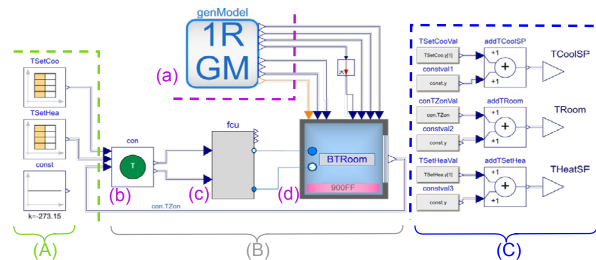

Figure 3. GM integrated with a Re-factored Single-Zone Building Model including a Fan Control Unit

Let us now describe the physics-based models. Above the segment labeled (B) in Figure 3, there are four components. GM models are identified with the label (a) and have been described in detail in Section 3.2. Labeled (c), a simple fan coil unit (FCU) is included to condition the building, which is shown in Figure 4. The FCU is regulated by a simple thermostat, which is modeled by a dual proportional and integral (PI) controller with dual set point, shown in Figure 5, to maintain room temperature within the set points of heating and cooling. When the

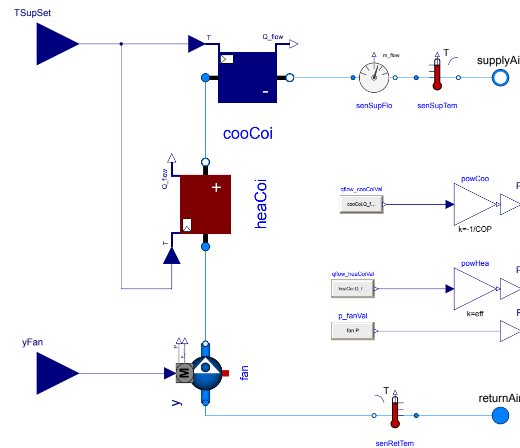

Figure 4. Fan Coil Unit labeled (c) in Figure 3

FCU is activated, the supply fan runs at a constant speed to circulate air through the heating and cooling coils, indicated in Figure 4 as `heaCoi` and `cooCoi`, respectively. The heating and cooling set points are converted to the supply air temperature set point by the PI controller shown in Fig. 5[4], and the coils are activated to reach the set point. The conditioned air is then supplied to the building through the `supplyAir` interface in Figure 4, where it is assumed to be well mixed. For the illustrative purposes of the examples herein, the energy impact of FCU is

[3] Provided that the parameter to be changed is non-structural (Modelica Association 2017).

[4] Observe that the goal here was to provide a simple implementation of the thermostat. To avoid numerical issues that could appear due to the use of the `Modelica.Blocks.Logical.GreaterThreshold` block set to > 0 could be made.

simplified, i.e., simple electrical models are used to determine the consumed power. The electric heating coil has a constant efficiency of 0.9, and the cooling coil operates at a constant coefficient of performance (COP) of 3.0.

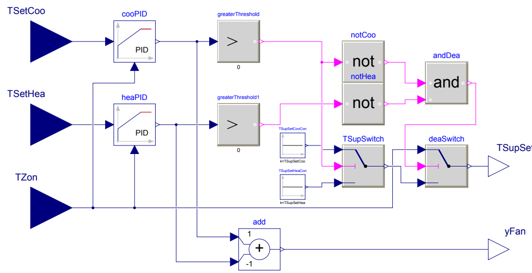

Figure 5. Thermostat Model labeled (b) in Figure 3

Finally, consider the component labeled (d) in Figure 3, this is the building model, which is expanded in Figure 6. The modeling hierarchy is shown for three layers. In the layer labeled (A), it is shown how the GM model is interfaced, and certain variables are scaled to match the model's units. In layer (B), further refactoring and computations are performed, mainly the interfacing with a weather data block with the GM and the computation of the total radiant, convective, and latent heat gains from the prediction of the GM model. Finally, layer (C) the underlying model refactored from the `Buildings` library. Although the specific case shown here corresponds to `Case900FF` of the ASHRAE BESTEST validation models (ASHRAE 2007), the refactoring provides an object-oriented hierarchy that allows one to easily modify other models by matching the interfaces for radiant, convective, and latent heat flow (which are constant in the original model), and move the weather data block to layer (B) to adapt weather conditions based on input from the GM.

The building represents a single zone with a window on the south wall and a constant infiltration mass flow rate. For the examples considered here, there are two variations of construction, the light weighted `Case600FFF` and the heavy weighted `Case900FFF`. The exterior walls and roof of `Case600FFF` and 900 are, respectively, plaster board with fiberglass insulation and concrete block with foam insulation. The floor of `Case600FFF` is timber construction and the floor of `Case900FFF` is concrete slab.

4.1.2 Measurement Data for GM Training

To train generative models, we use measurement data collected from SUSTIE, a cutting edge net zero-energy commercial office building located in Japan[5]. The name SUSTIE combines the words "Sustainability" and "Energy" and the building is designed to investigate and demonstrate technologies that can lead to energy savings and worker health and comfort. The four-story SUSTIE building has a total floor area of approximately 6456 m^2 which includes nine office spaces (experimental rooms) regularly used by around 260 office workers, an open atrium area, a cafeteria and a gym.

The building management system at SUSTIE gathers data on electrical energy usage, weather conditions, indoor environmental parameters, occupancy levels, and equipment operations to monitor and manage energy consumption and comfort throughout the building's operations. The electrical energy consumption is measured separately for different types of equipment (air conditioning, ventilation, lighting, hot water supply, and elevators) and for each room. The occupancy, i.e. the number of people in each room, is counted by the access control system using card readers installed in each area. This constitutes hundreds of sensing instruments installed throughout the building measuring more than 2,500 unique data signals throughout the year, 24 hours a day, with a sampling rate of 1 minute.

In this work a data set collected at SUSTIE over 20 consecutive months from January 2021 to August 2022 is used for training the GM's used in the examples below. For more information on the steps required for pre- and post-processing of data and GM training, see (Chakrabarty, Vanfretti, et al. 2024).

4.2 Illustrative Example

Let us now present some simulation results obtained by simulating the model shown in Figure 3.

First, we present the GM predictions in Figure 7, which correspond to the signals from `genModel`, (a) in Figure 3 that are fed to the building model (d) in Figure 3. These figures display the mean of the distribution from the measured data (*nom*) as well as a realization from the generative model (GM). The output of the GM corresponds to several variables (e.g., occupancy, equipment power, etc.) that influence the radiant, convective, and latent heat gains of the room shown, which is represented by a light blue square with a gray edge in Figure 6 (C). As can be seen in Figure 7, the GM model provides a time series that reflects what is expected of such types of building operations. For example, in Figure 7 (a) the occupancy increases in the morning and falls to zero during the day, while in Figure 7 (b) the power consumed by the equipment is highest in the morning and drops to a minimum at night. This illustrates the expressiveness of these generative models, as multiple simulations can be used to characterize the effect of the uncertainty in these input quantities.

The influence of the GM model upon the radiant, convective, and latent heat gains is shown in Figure 8. Observing the flow of radiant heat in Figure 8 (a) while at the same time observing both the power of the equipment in Figure 7 (b) and the global horizontal radiation in (f), it can be observed that during the beginning of the day both variables influence the radiant heat. Meanwhile, the convective heat flow in Figure 8 (b) is not as drastically affected at the beginning of each day by these variables. Furthermore, it should be noted that the latent heat flow in

[5] See `https://www.mitsubishielectric.com/en/about/rd/sustie/index.html`.

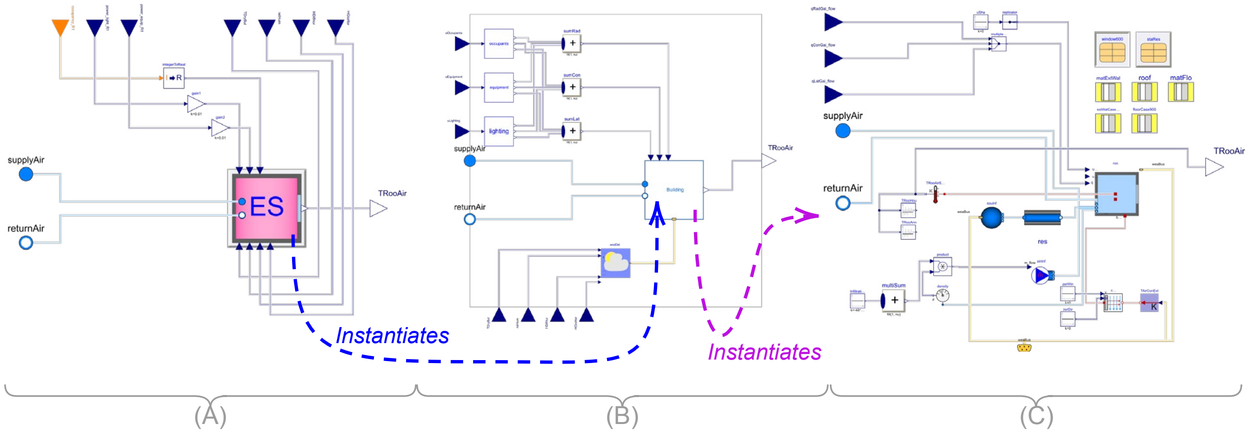

Figure 6. Hierarcichal Layers of the Single-Zone Building-Model labeled (d) in Figure 3. Observe that layer (C) is a refactored model from the `Buildings` library, corresponding to `Case900FF` of the ASHRAE BESTEST validation models (ASHRAE 2007).

Figure 8(c) is dominated by occupancy. It is worth noting that in the case of the latent heat flow, not having a good estimate such as those from the GM can lead to a substantial underestimation of latent heat, as shown by the case where the variables are assumed to be some 'nominal' heat loads obtained from a nominal schedule of energy use in the zone.

The GM's output also influences the heat and mass balance in the moist air of the room. Within the room in Figure 6 (C), the component `MixedAirHeatMassBalance` determines the heat and mass balance of moist air (M., W. Zuo, and T. Nouidui 2011), as can be seen in the sensed air temperature and flow rate of water added to the air using the `MixingVolumeMoistAir` component from the `Buildings` library. The resulting effect of the difference between the nominal inputs and a realization from the generative model can can be observed in Figure 9(a)-(c); the GM output allows a better estimation of the resulting heat flow in the room (see (a)) and a more realistic temperature estimate (see (b)), which can serve in the sizing of the HVAC system and/or improving its control. It should be noted that, although small, it is also possible to quantify the rate of extraction of water from moist air in Figure 9 (c), which would otherwise be underestimated when using nominal values instead of those of the GM.

Finally, to maintain the room temperature shown in Figure 9 (b) within the specified set points, the FCU and the thermostat in Figures 4 and 5 must cool the air. The resulting set point provided by the thermostat to regulate the cooling is shown in Figure 10 (a), with the air flow from the FCU shown in Figure 10 (b). From these figures, it can be seen that the impact of including the GM serves to adapt the performance of the cooling system according to the operating needs of the building. Observe in Figure 10 (a) that the new setpoints adapt to changes in the building conditions that are not prescribed by the mean of the experimental measurements, allowing the user to quantify the uncertainty in the system performance related to variations in the building operation.

5 Conclusions

Bringing together physics-based buildings models with models that describe variables driven by human interactions has the potential to substantially improve the performance of existing buildings or to develop better informed building designs, particularly when considering heating and cooling requirements that impact HVAC systems. To explore this potential, this paper has presented the requirements and a prototype implementation of the integration of machine learning generative models and physics-based building models. Once the generative models were trained, they were linked to a building model by exploiting the external function interfaces for C defined in the Modelica language specification. This enabled the reuse of Modelica building models from the `Buildings` library, while simultaneously leveraging real-world occupancy, power consumption, and other data for building energy simulations, as demonstrated by the provided examples.

Although the prototype implementation proposed here has proven beneficial in the development of novel building control performance analysis techniques (Chakrabarty, Vanfretti, et al. 2024), it has several other application domains. For example, it can be used similarly to characterize load patterns and perform power system control performance evaluations (Bombois and Vanfretti 2024). There is also a great deal of room for improvement, for example, to be able to use multiple and different types of generative model architectures, which will be subject to future work.

Acknowledgements

Part of the work reported in this article was carried out by the first author during his sabbatical at Mitsubishi Electric Research Laboratories (MERL). The author thanks MERL for the opportunity and support.

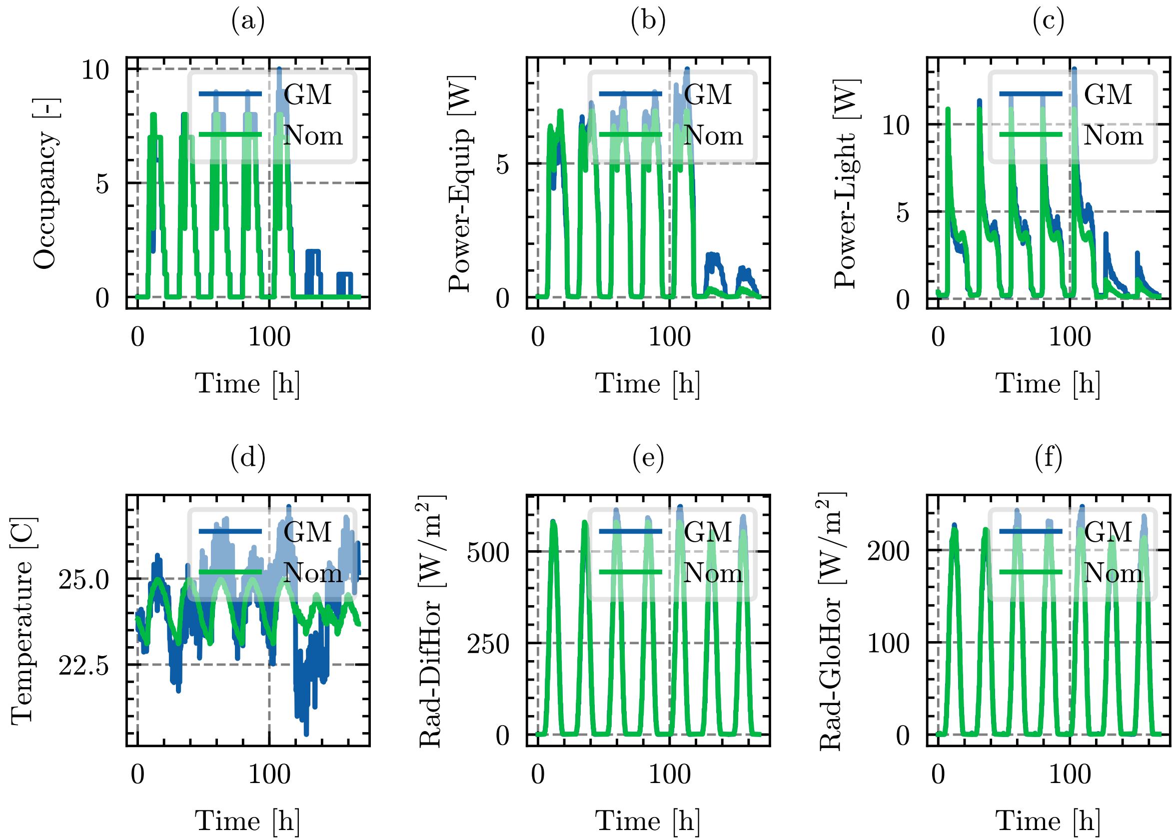

Figure 7. Outputs of the GM model, (a) in Figure 3, for a 7-day period. (a) Occupancy. (b, c) Power consumed by equipment and lighting. (d) Outdoor temperature. (e,f) Solar radiation, scattered and global.

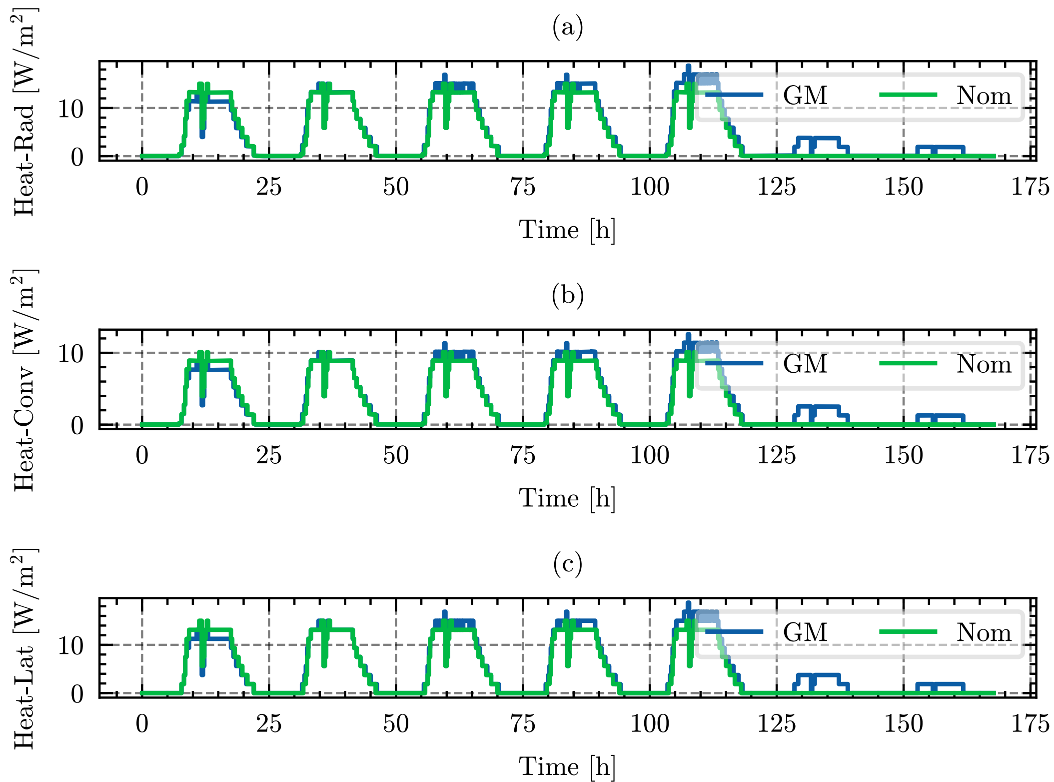

Figure 8. (a) Radiant, (b) convective and (c) latent Heat Flow resulting from the GM and a Nominal signal.

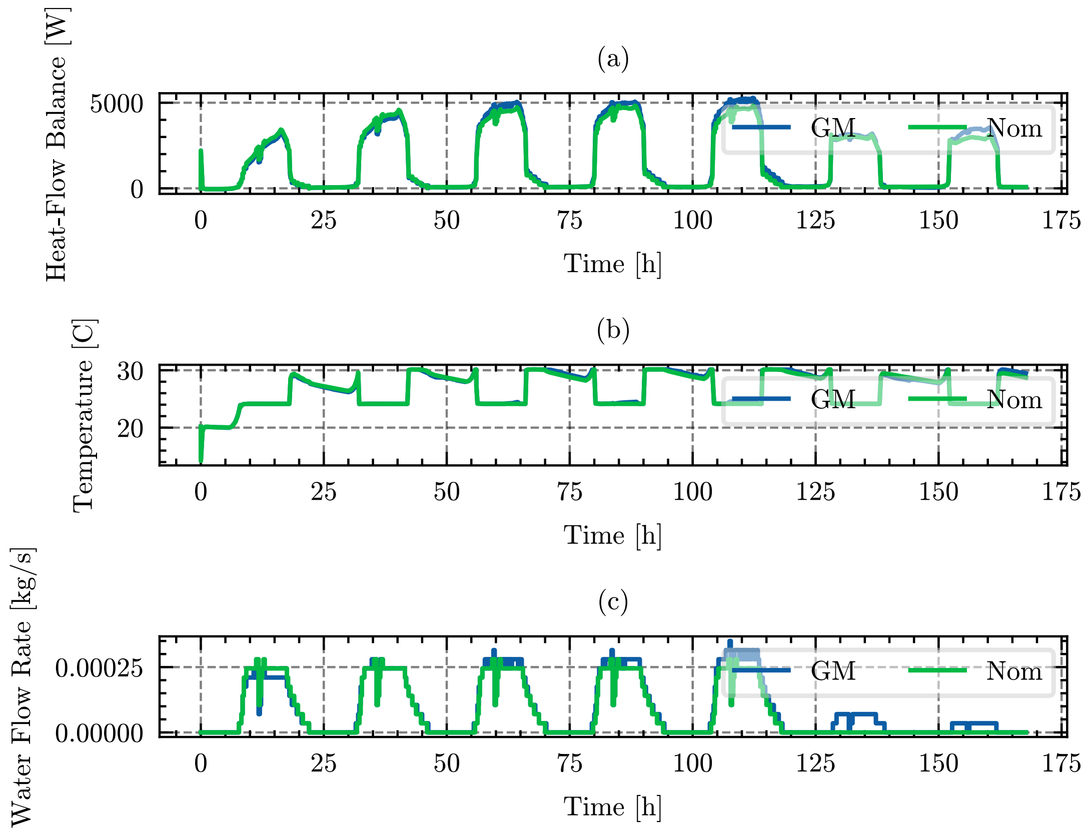

Figure 9. Mixed air conditions in the room in Figure 6 (C). (a) Heat flow balance, (b) temperature and (c) water flow rate.

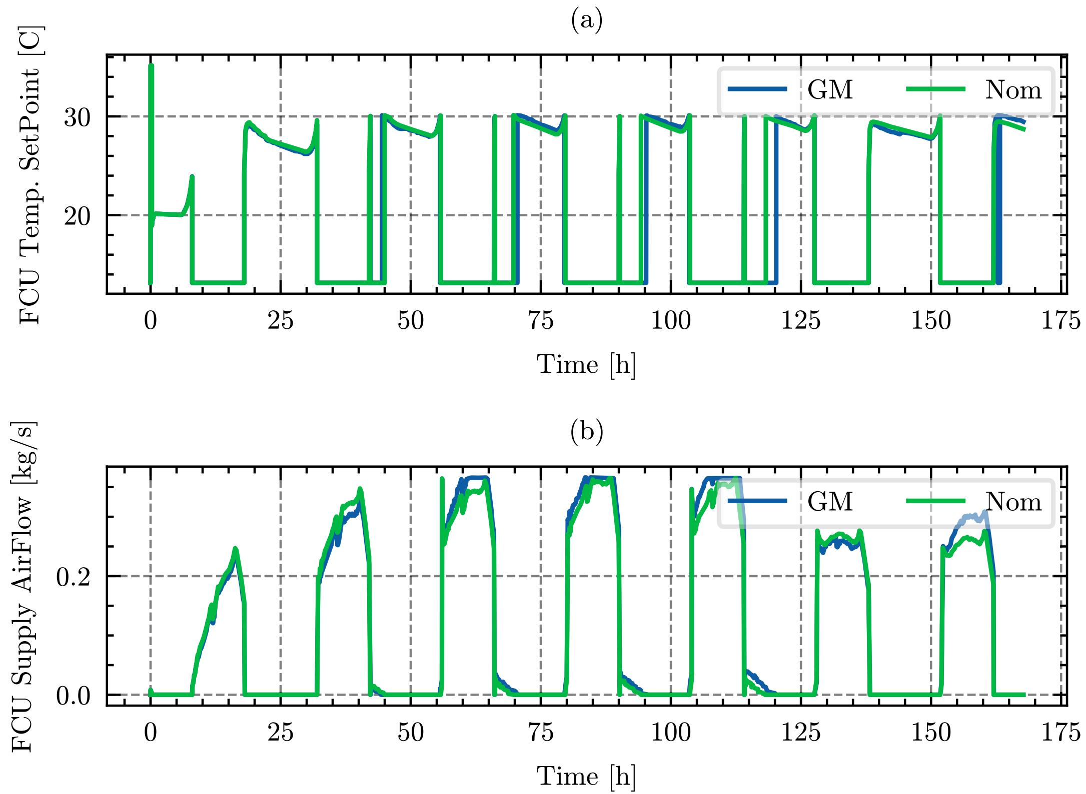

Figure 10. (a) Thermostat `TSupSet` output (see Figure 5) and (b) Supply air output measured from `senSupFlo` (see Figure 4)

References

ASHRAE (2007). *140: Standard Method of Test for the Evaluation of Building Energy Analysis Computer Program*. Tech. rep. ASHRAE.

Bombois, Xavier and Luigi Vanfretti (2024). "Performance monitoring and redesign of power system stabilizers based on system identification techniques". In: *Sustainable Energy, Grids and Networks* 38, p. 101278. ISSN: 2352-4677. DOI: https://doi.org/10.1016/j.segan.2024.101278.

Bruck, D., H. Elmqvist, H. Olsson, et al. (2002). "Dymola for Multi-Engineering Modeling and Simulation". In: *2nd International Modelica Conference*, pp. 55–1 — 55–8. URL: https://modelica.org/events/Conference2002/papers/p07_Brueck.pdf.

Chakrabarty, Ankush, Emilio Maddalena, Hongtao Qiao, et al. (2021). "Scalable Bayesian optimization for model calibration: Case study on coupled building and HVAC dynamics". In: *Energy and Buildings* 253, p. 111460.

Chakrabarty, Ankush, Luigi Vanfretti, et al. (2024). "Assessing Building Control Performance using Physics-Based Simulation Models and Deep Generative Networks". In: *8th IEEE Conference on Control Technology and Applications (CCTA)*. IEEE, pp. 1–8.

Chen, Zhenghua and Chaoyang Jiang (2018). "Building occupancy modeling using generative adversarial network". In: *Energy and Buildings* 174, pp. 372–379.

Codeca, F. and F. Casella (2006). "Neural Network Library in Modelica". In: *5th International Modelica Conference*, pp. 549–557. URL: https://modelica.org/events/modelica2006/Proceedings/sessions/Session5c3.pdf.

Das, Hari Prasanna, Ryan Tran, Japjot Singh, et al. (2022). "Conditional synthetic data generation for robust machine learning applications with limited pandemic data". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 36. 11, pp. 11792–11800.

Dassault Systemes AB (2023-09). *Dymola — Dynamic Modeling Laboratory — Full User Manual*. Lund, Sweden.

M., Wetter, W. Zuo, and T. Nouidui (2011). "Modeling of Heat Transfer in Rooms in the Modelica "Buildings" Library". In: *12th Conference of International Building Performance Simulation Association*, pp. 1096–1103.

Mirakhorli, Amin and Bing Dong (2016). "Occupancy behavior based model predictive control for building indoor climate—A critical review". In: *Energy and Buildings* 129, pp. 499–513.

Modelica Association (2017). "Modelica Specification, Version 3.4". In: URL: www.modelica.org.

Modelica Association (2019). "Functional Mockup Interface for Model Exchange and Co-Simulation, Version 2.0.1". In: URL: www.fmi-standard.org.

Salatiello, Alessandro, Ye Wang, Gordon Wichern, et al. (2023). "Synthesizing Building Operation Data with Generative Models: VAEs, GANs, or Something In Between?" In: *Companion Proceedings of the 14th ACM International Conference on Future Energy Systems*, pp. 125–133.

Schweiger, Gerald, Cláudio Gomes, Georg Engel, et al. (2019). "An empirical survey on co-simulation: Promising standards, challenges and research needs". In: *Simulation modelling practice and theory* 95, pp. 148–163.

Stoffel, Phillip, Laura Maier, Alexander Kümpel, et al. (2023). "Evaluation of advanced control strategies for building energy systems". In: *Energy and Buildings* 280, p. 112709.

The MathWorks (n.d.). *Export Network as FMU*. Accessed: Feb. 2, 2024. Available since v. 2023b. URL: https://www.mathworks.com/help/deeplearning/ug/export-network-to-fmu.html.

Wetter, Michael, Paul Ehrlich, Antoine Gautier, et al. (2022). "OpenBuildingControl: Digitizing the control delivery from building energy modeling to specification, implementation and formal verification". In: *Energy* 238, p. 121501.

Wetter, Michael, Wangda Zuo, Thierry S Nouidui, et al. (2014). "Modelica buildings library". In: *Journal of Building Performance Simulation* 7.4, pp. 253–270.

Wolfram Research (n.d.). *NeuralNet: Library that provides support for the use of neural networks in modeling*. Accessed: Feb. 2, 2024. URL: https://reference.wolfram.com/system-modeler/libraries/SystemModelerExtras/SystemModelerExtras.NeuralNet.html.

XRG Simulation GmbH (n.d.). *SmartInt: Simple Modelica Artificial Intelligence Interface*. Accessed: Feb. 2, 2024. URL: https://github.com/xrg-simulation/SMArtIInt.

Ye, Yunyang, Matthew Strong, Yingli Lou, et al. (2022). "Evaluating performance of different generative adversarial networks for large-scale building power demand prediction". In: *Energy and Buildings* 269, p. 112247.

Zhan, Sicheng, Gordon Wichern, Christopher Laughman, et al. (2022). "Calibrating building simulation models using multi-source datasets and meta-learned Bayesian optimization". In: *Energy and Buildings* 270, p. 112278.