Aalborg Universitet

AALBORG UNIVERSITY
DENMARK

pgFMU

Integrating Data Management with Physical System Modelling

Rybnytska, Olga; Siksnys, Laurynas; Pedersen, Torben Bach; Neupane, Bijay

Published in:
Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020

DOI (link to publication from Publisher):
10.5441/002/edbt.2020.11

Creative Commons License
CC BY-NC-ND 4.0

Publication date:
2020

Document Version
Accepted author manuscript, peer reviewed version

Link to publication from Aalborg University

Citation for published version (APA):
Rybnytska, O., Siksnys, L., Pedersen, T. B., & Neupane, B. (2020). pgFMU: Integrating Data Management with Physical System Modelling. In *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020* (pp. 109-120). OpenProceedings.org. https://doi.org/10.5441/002/edbt.2020.11

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal -

Take down policy
If you believe that this document breaches copyright please contact us at vbn@aub.aau.dk providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from vbn.aau.dk on: November 08, 2024

pgFMU: Integrating Data Management with Physical System Modelling

Olga Rybnytska, Laurynas Šikšnys, Torben Bach Pedersen, and Bijay Neupane
Department of Computer Science, Aalborg University
olga@cs.aau.dk,siksnys@cs.aau.dk,tbp@cs.aau.dk,bj21.neupane@gmail.com

ABSTRACT

By expressing physical laws and control strategies, interoperable physical system models such as Functional Mock-up Units (FMUs) are playing a major role in designing, simulating, and evaluating complex (cyber-)physical systems. However, existing FMU simulation software environments require significant user/developer effort when such models need to be tightly integrated with actual data from a database and/or model simulation results need to be stored in a database, e.g., as a part of larger user analytical workflows. Hence, users encounter substantial complexity and overhead when using such physical models to solve analytical problems based on real data. To address this issue, this paper proposes pgFMU - an extension to the relational database management system PostgreSQL for integrating and conveniently using FMU-based physical models inside a database environment. pgFMU reduces the complexity in specifying (and executing) analytical workflows based on such simulation models (requiring on average 22x fewer code lines) while maintaining improved overall execution performance (up to 8.43x faster for multi-instance scenarios) due to the optimization techniques and integration between database and an FMU library. With pgFMU, cyber-physical data scientists are able to develop a typical FMU workflow up to 11.74x faster than using the standard FMU software stack. When combined with an existing in-DBMS analytics tool, pgFMU can increase the accuracy of Machine Learning models by up to 21.1%.

1 INTRODUCTION

Cyber-physical system experts, cyber-physical data scientists, and cyber-physical developers often need to analyze, predict, and simulate physical systems. For this purpose, *physical system models* are often used to capture time-dependent behaviour and dynamics of such systems [1]. They offer powerful, rigorous, and cost-effective means to characterize and reason about such systems, without the need to build, interact, and/or interfere with such systems. Physical system models (*physical models* for short) are well supported by a number of physical system modelling software tools and environments. However, each such modelling environment often uses a specialized form and format of a physical model with limited possibilities to utilize such models across different tools and environments. To mitigate this problem, *Functional Mock-up Interface (FMI)* [2] has emerged as a de-facto standard [3] to facilitate physical model exchange and co-simulation across a large number of modelling tools. In FMI, physical models are compiled into a standard representation, denoted as *functional mock-up units* (FMUs). FMUs reflect real physical systems composed of physical and digital components interacting in complex ways according to a set of pre-defined control strategies and physical laws. FMUs allow accurately defining system behaviour even in the physical states that are not observable in the real world, unlike what is required by traditional AI/ML models (e.g., artificial neural networks). Due to these advantages, FMUs continue gaining popularity in relevant physical modeling communities [5]. FMI has already been broadly adopted and supported by 130+ software tools, including well-known simulation environments Simulink (Matlab) [4] and EnergyPlus [6] (80.000+ downloads), as well as JModelica [8]/Open Modelica [10] (more than 20 companies and 30 universities in the consortium) with 1600+ model components and 1350+ functions from many domains available in a standard library alone.

Despite comprehensive physical modelling support, existing FMI-based simulation environments and tools offer poor database (DB) integration and lack built-in support for conveniently including physical models into user-defined analytical workflows. Thus, user data (e.g., model parameters, measurements, control inputs) cannot be conveniently supplied to the model from a DB and model results cannot be effectively used in larger analytical workflows (e.g., those encompassing multiple simulations) while offering convenient declarative approaches to manage such physical models within a common data management and physical modelling environment. Without these capabilities, model-driven analytical tasks become complicated and slow in terms of both development time and execution, and less usable for users working with, e.g., prescriptive analytics applications [12]).

As a running example, consider a prediction problem from the energy field. The aim is to predict and analyze indoor temperatures inside a house that is heated by an electric heat pump (HP) under different heating scenarios (e.g., *no heating*, *heating at max power*). For this task, a physical model represented as an FMU needs to be calibrated and simulated using measurements and weather data stored in the database. Predictions need to be stored in the database, for further analysis and visualization. In this case, as shown in Figure 1, the user has to pick relevant FMU-compliant software tools (e.g., JModelica [8] or Python [7] + PyFMI [14] + ModestPy [3]) and then use these tools to (1) load a pre-generated FMU file or manually build an FMU file from a model specification file, (2) read historical measurements and (future) control inputs from a database, (3) recalibrate the model (e.g, using ModestPy) in case the model cannot ensure the good fit with the historical measurements, (4) validate the model against real measurements, and update the model and/or parameters values, (5) simulate the updated model to generate temperature predictions for different scenarios (e.g, using PyFMI), (6) export predicted values back to the database, and (7) perform further analysis utilizing a DBMS. This imposes limitations for the user in terms of number of software tools and libraries to use, and in terms of the overall complexity and ability to effectively utilize physical models in larger analytical workflows where both simulation and optimization are required. In this and similar user workflows, interleaved data exchange between a database and a

© 2020 Copyright by the author(s). Published in Proceedings of the 23rd International Conference on Extending Database Technology (EDBT), March 30-April 2, 2020, Copenhagen, Denmark, ISBN 978-3-89318-083-7 on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

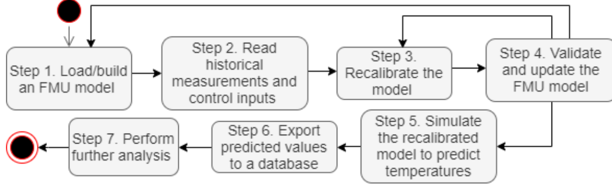

Figure 1: Running example workflow

modelling tool makes the workflows difficult to specify and evaluate, leading to significant implementation complexity, developer and performance overheads. This calls for new solutions offering more tight integration between a database and a modeling tool.

This paper proposes pgFMU – an extension to the PostgreSQL [9] DBMS to address these issues. pgFMU is a SQL-based model- and data management environment that brings benefits to cyber-physical data scientists and cyber-physical software developers. pgFMU facilitates FMU model storage, simulation, and parameter estimation tasks by effectively integrating FMI inside PostgreSQL. For this purpose, pgFMU offers a number of User-Defined Functions (UDFs) accessible by simple SQL queries for each necessary operation. As such, our extension exhibits the following advantages: (1) increased user productivity (on average 11.74x faster for user-defined workflows in terms of development time) due to the usage of a single integrated system for FMU-specific use cases, (2) reduced system complexity (22x fewer code lines), (3) increased performance due to the reduction in I/O operations and optimization for multi-instance workflows (up to 8.43x faster), (4) reduced number of error-prone actions by minimizing the number of software systems and tools a data scientist is required to work with, and 5) when used in combination with traditional Machine Learning models, pgFMU can increase model accuracy by up to 21.1%. These are supported by the results of our experiments which are based on the real-world use cases. Lastly, pgFMU is an open-source project and can be found on GitHub[1].

The rest of the paper is organized as follows: Section 2 elaborates on solving the aforementioned use case using traditional stack, Section 3 discusses the related work Section 4 highlights technical challenges of integrating FMUs inside a FMS, Section 5 discusses how pgFMU tackles model storage and specification, Section 6 presents our pgFMU-based approach for parameter estimation, Section 7 discusses the model simulation, Section 8 presents the experimental evaluation, and Section 9 concludes the paper and suggests future research directions.

2 RUNNING EXAMPLE

In this section, we follow Figure 1 and elaborate on the steps of solving the exemplified case of predicting indoor temperatures of a house. This example illustrates a typical process, steps, and software packages a cyber-physical data scientist would adopt to address this specific and similar problems based on FMU models.

Load / build an FMU model. A Functional Mockup Unit (FMU) is a .zip file consisting of a number of XML files, model and underlying solver implementation C-language files and/or their binary compilations for execution on different hardware platforms. The XML files describe model variables and parameters, attributes, default stop time, tolerance, etc., to be used, mostly, by FMU libraries and software tools and not the end-user. The cyber-physical data scientist can either download a pre-compiled FMU file from the (third-party) system modellers, or build an FMU file manually using a modelling tool, e.g., Open Modelica [10]. The latter option requires domain knowledge and modeling skills to mathematically capture all instantaneous thermal energy losses (and gains), intensities, and their balance within the building (e.g., windows, walls). For this problem, at time $t_k, k = 1 \dots n$, temperature dynamics of the house can be captured by a generic *linear time invariant state space* model (1) (adapted from [11]):

$$\begin{aligned} x(t_{k+1}) &= F(x(t_k), u(t_k), \theta(t), t) \\ y(t_k) &= H(x(t_k), u(t_k), \theta(t), t), \end{aligned} \tag{1}$$

where $x(t_0) = x_0$ is the initial state, F is a linear or non-linear function, $F : \Re^n \times \Re^m \times \Re^p \times \Re \to \Re^n$, $x(\cdot) \in \mathbb{R}^n$ is the state variable vector, $u(\cdot) \in \mathbb{R}^m$ is the model input vector, $\Theta(\cdot) \in \mathbb{R}^p$ is the parameter vector, $y(\cdot) \in \mathbb{R}^O$ is the model output vector, and $H : \Re^n \times \Re^m \times \Re^p \times \Re \to \Re^O$ is the output function.

This model often needs to be translated into a representation required by a specific modelling tool, e.g., the Modelica [13] program shown in Figure 2. There, u is the *input variable* – heat pump power rating settings in the range [0 ... 1], corresponding to [0 .. 100%] of HP power operation; x is the *state variable* – indoor temperatures; y is the *output variable* – energy consumed by a heat pump. The parameters are represented by A, B, C, D, and E; $A = \left(-\frac{1}{R \cdot Cp}\right)$, $B = \left(\frac{P \cdot \eta}{Cp}\right)$, $C = P$, $D = 0$, $E = \left(\frac{\theta_a}{R \cdot Cp}\right)$, where $Cp = 1.5kWh/°C$ is the thermal capacitance (the amount of energy needed to heat up by 1 $°C$ within 1 hour); $R = 1.5°C/kW$ is the thermal resistance; $P = 7.8kW$ is the rated electrical power of the heat pump; $\eta = 2.65$ is the performance coefficient (the ratio between energy usage of the heat pump and the output heat energy); $\theta_a = -10°C$ is the outdoor temperature.

Read historical measurements and control inputs. Modelling tools often have poor support for database integration and require model parameters and inputs to be provided in a predefined format, usually a text file. If model inputs are stored in a database, the users are required to either manually export them to use it within the modelling software, or to download and familiarize themselves with often complex library functionality (e.g. Matlab Database Explorer app [4] or *psycopg2* [16] Python package). Yet, storing the measurements in a database has a number of advantages while handling the concurrent I/O operations such as managing the information from the multiple sensors and serving end-user applications. Lastly, the user is required to retrieve the control inputs from the FMU, and manually match them with the input data series obtained from the measurements.

Recalibrate the model. Values of one or more model parameters (e.g., A, B, E in Figure 2) are often either not known, or tuned for another physical system. This may result in poor model predictions, or erroneous decisions. *Parameter estimation* (or model calibration) is the operation of fitting model parameters to actual measurements, such that simulated model states and

```
1  model heatpump "Model of LTI SISO heat pump system"
2  output Real x(start = 20.7507) "State variable of the system - indoor temperature";
3  input Real u "Input of the system - Heat pump power range [0 ... 100%]";
4  output Real y "Output variable - Heat pump energy consumption";
5  parameter Real A "A coefficient of LTI SISO system"
6      annotation(Fixed=false);
7  parameter Real B "B coefficient of LTI SISO system"
8      annotation(Fixed=false);
9  parameter Real C "C coefficient of LTI SISO system"
10     annotation(Fixed=false);
11 parameter Real D "D coefficient of LTI SISO system"
12     annotation(Fixed=false);
13 parameter Real E "E coefficient of LTI SISO system"
14     annotation(Fixed=false);
15 equation
16 der(x) = A*x + B*u +E;
17 y = C*x + D*u;
18 end heatpump;
```

Figure 2: Heat pump LTI SISO model in .mo format

[1] https://github.com/OlgaRyb/pgFMU.git

outputs acceptably match measured system states and outputs. In our problem, the unknown model parameter values are A, B and E (see Figure 2), and the sum of squared errors between the measured and simulated indoor temperatures is to be minimized.

Parameter estimation can be performed, for instance, using the ModestPy Python package [3] and its built-in function `Estimation`. In this situation, `Estimation` requires a path to the working directory, a path to the FMU model file, input measurements, known values of the parameters, values to be estimated, and a dataset with real measurements to calibrate upon. All these formal parameters need to be specified explicitly by the user. Additionally, in case the user needs to further use the updated model, parameters update is required to be done manually by calling a specific function from the *PyFMI* [14] package.

Often, a set of relevant parameters to be estimated has to be identified by the user. This operation demands an extensive domain knowledge, or assistance of the external domain expert. As an alternative, the user can use the functionality of *PyFMI* Python package to retrieve the relevant parameters of an FMU model. However, the list of retrieved parameters always needs to be accompanied by often complex filtering. For example, by default the *PyFMI* fetches the full list of parameters, including the parameters related to the built-in solver internally connected with the FMU model. Such parameters are not relevant for the parameter estimation operation, therefore, should be filtered out.

Validate and update the FMU model. Once parameter estimation is performed, the model needs to be validated before it can be used for subsequent predictions, e.g., using cross-validation. In case the model cannot generate predictions with sufficient accuracy, the model has to be either re-calibrated (e.g., with a different set of parameters) or further refined or replaced (i.e., using another FMU). In our case, validation should be performed using user-defined Python scripts.

Simulate the recalibrated model to predict temperatures. Next, we need to generate predictions by simulating the calibrated FMU model based on the desired input (HP power rating setting in the range [0 ... 1]/heating scenario). During simulation, model outputs and states are computed based on the provided inputs (and previous states), as seen in Figure 3. To simulate the model in this workflow, the user need to write a (Python) program (e.g. using the *PyFMI*) to load the *.fmu* file, read input data, map data to the form required by the simulation library (e.g. using *numpy* and *pandas* [15]), simulate the model (*PyFMI*), and insert simulation results back in the database (*psycopg2*).

Export predicted values to a database. For further analysis and visualization, the user needs to export data either directly to a database or first to a text file and then import the text file into a database using the respective SQL command.

Perform further analysis. Stored predicted indoor temperatures may further be used for subsequent visualization, analysis, optimization, and/or decision support. However, if predictions

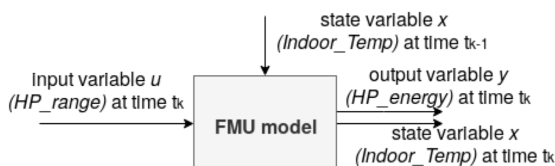

Figure 3: Simplified simulation FMU schematic of the HP system

Table 1: Workflow operations

Operation	Package	Code lines	
		Python	*pgFMU*
Load/build an FMU model	PyFMI	4	1
Read historical measurements and control inputs	psycopg2, PyFMI, pandas	12	-
Recalibrate the model	ModestPy, pandas	15	1
Validate & update the FMU model	PyFMI, pandas	7	-
Simulate the recalibr. model to predict temp.	PyFMI, Assimulo, numpy	24	1
Export predicted values to a DB	psycopg2, pandas	4	-
Perform further analysis	psycopg2, PyFMI	22	1
Total		**88**	**4**

under different model inputs are required, or by using different models, this overall process needs to be repeated. When predictions with multiple FMU model need to be performed, the cyber-physical data scientist faces additional complexity of managing all these models, their parameters values, and predictions. As seen in Table 1, for this particular example, the data scientist would have to perform 7 steps utilizing 6 different Python packages and writing 88 lines of code. The necessity to utilize multiple software packages makes development time-consuming and error-prone. At the same time, the usage of different programming interfaces makes such tasks difficult to perform, manage, and customize. FMU integration inside a DBMS, as in pgFMU, results in an order of magnitude reduction in the code (as seen in last table column).

3 RELATED WORK

FMU model simulation is widely supported by over 130 tools and packages, most of them being commercial ones. Among open-source projects are OpenModelica[10], PyFMI [14], and FMI Library as a part of JModelica[8]. However, even while providing GUI (e.g., OpenModelica), these tools do not have the solid support for the import and export of data from/to a DBMS. For example, OpenModelica requires an intermediary text file to be created as an input to the simulation engine; this text file should follow a strict format to be able to continue the simulation.

Another well-known environment for FMU simulation and parameter estimation is JModelica [8]. JModelica offers a Python interface, and serves the user with a number of functions required for a broad range of simulation- and optimization-related tasks. Another advantage of JModelica is package orchestration: even though some packages (e.g. PyFMI or Assimulo [20]) are claimed to be independent of other software, individual package management still remains a challenge. Nonetheless, JModelica does not provide yet a direct database input support, so the users are required to use extra packages (e.g. psycopg2).

At the same time, attempts were made to develop in-DBMS analytics: SAS [21] in collaboration with Teradata suggested the integration of DBMS with predictive modelling (including regression analysis and time series analytics); Tiresias [22] - a PostgreSQL-based system that supports "how-to" queries for defining and integrated evaluation of constrained optimization problems, and SolveDB - an extensible DBMS for SQL-based optimization problems [23]. The existing DBMSes offer only limited analytical support, such as in-DBMS linear optimization and, to

Table 2: Comparison between in-DBMS analytics tools and pgFMU

Feature	MADlib	Microsoft SQL Server ML Services	pgFMU
Data query language	SQL	SQL	SQL
Model integration approach	UDFs	Stored procedures	UDFs
In-DBMS machine learning	✓	✓	✗
In-DBMS physical models	✗	✗	✓
- FMU management	✗	✗	✓
- FMU simulation	✗	✗	✓
- FMU parameter estimation	✗	✗	✓

some extent, forecasting and simulation. To our best knowledge, no systems have until now supported in-DBMS storage, analysis, simulation, and calibration of the FMI-compliant models.

There exists a number of tools that incorporate traditional Machine Learning (ML) models into a DBMS, e.g., the hugely popular MADlib [27] and the recently released Microsoft (MS) SQL Server ML Services [28]. Table 2 shows a comparison of pgFMU with these systems. As we can see, neither MADlib, nor Microsoft SQL Server ML Services support physical system models (like FMUs), unlike pgFMU. MADlib follows the same integration approach as pgFMU by offering a set of UDFs specialized for ML. MS SQL Server ML Services provides a single T-SQL stored procedure for executing external Python and R scripts for ML, which is considerably less user-friendly as two languages (SQL + Python/R) are involved. While the UDFs of pgFMU can be installed as a DBMS extension, pgFMU can be used as a complement to existing ML tools, e.g., MADlib, which we demonstrate in Section 8.

4 CHALLENGES OF INTEGRATING FMUS INSIDE A DBMS

In the next section, we highlight 3 essential technical challenges of integrating FMUs inside a DBMS:

Challenge 1. Language Integration Challenge: *How to integrate and expose FMUs in database queries?* The challenge is to offer effective new SQL constructs, so that FMUs and FMU instances can be created, analyzed, manipulated, and used effectively together with existing SQL constructs.

Challenge 2. FMU Meta-data Challenge. *How to optimally take advantage of FMU meta-data to semi-automate task specification and data mapping?* Traditionally, the specification of FMU tasks and mapping between (discrete) data stored in a database and the FMU variables is manual and very verbose, but there is a huge opportunity to take advantage of the meta-data stored as a part of a FMU to automate such specifications.

Challenge 3. FMU Performance Challenge. *How to increase system performance when a (generic) FMU needs to be instantiated and used many times?* Individual tasks within FMU workflows (e.g., Figure 1) often require the same FMU to be instantiated and simulated many times. When model parameters and inputs change only marginally, FMU computations/results of FMU simulation can be reused to increase overall performance. The next sections elaborate on how pgFMU addresses these challenges.

5 MODEL SPECIFICATION, STORAGE, AND MANIPULATION

pgFMU aims to facilitate arbitrary user-specified FMU-based workflows by enabling creation, storage, and manipulation of FMU models (e.g., in terms of model parameters to be estimated) in a DBMS through user-specified SQL queries. This leads to Challenge 1: Language Integration Challenge (Section 4). Here, pgFMU takes a "session-like" approach where FMU instances are managed and used by explicitly calling commands in a particular order for FMU creation, deletion, etc. For this, pgFMU employs the same approach as MADlib [27] and offers a set of easy-to-use User-Defined Functions (UDFs) that support the full range of model management operations. pgFMU packs and offers these UDFs as a PostgreSQL extension, for ease-of-use and installation. Thus, FMU-specific functionality of pgFMU can be used with, e.g., the functionality of MADlib, for combined machine learning and physical simulations, as shown later. In this section, we describe in detail the new DBMS constructs / UDFs for model specification and storage, explain how they work, and show how users can take advantage of them when working with FMU models.

In pgFMU, a user can create an instance of an FMU model by using the function (UDF) *fmu_create (modelRef, [instanceId])* ↦ *instanceId*. As input, the function takes a mandatory textual argument *modelRef*, that represents either a path to a pre-compiled FMU (.fmu) file, a Modelica (.mo) file, or inline Modelica model code. If *modelRef* is a Modelica argument, it will be automatically compiled into an FMU file. The function returns a textual model instance identifier *instanceId*, which uniquely identifies the model instance in the subsequent model management function calls. The model instance identifier can be set manually by the user by supplying *instanceId* as an optional argument.

As explained in Section 4, FMU meta-data can be used to semi-automate task specification and data mapping (Challenge 2). For this, pgFMU reads meta-data from the user-defined FMUs once during FMU load and stores it in a *model catalogue* in a database. Thus, pgFMU automatically detects simulation parameters (start/stop timestamp, time step), variable causalities (e.g., *input*, *output*, *parameter*) and data types (e.g., *float*, *integer*, *string*), automatically configures simulation, and performs implicit data conversions when needed. The user can manually override the detected parameters. As seen in Figure 4, the model catalogue consists of four basic tables: *Model*, *ModelVariable*, *ModelInstance*, and *ModelInstanceValues*. *Model* captures the essential model attributes. FMU models are identified with a Universally Unique Identifier (UUID) [25] - a 128-bit string for unique object identification. All loaded models are stored in the *FMU storage* (non-volatile memory). *ModelVariable* describes the model variables: the variable name, variable type, and the initial, minimum, and maximum values. The combination of *ModelId* and *varName* attributes serves as the primary key. *initialValue*, *minValue*, and *maxValue* have the *variant* [24] type - a specialized PostgreSQL data type that allows storing any data type in a column, while keeping track of the original data type. *ModelInstance* stores the information about model instances. It uses *instanceId* as the primary key, and *modelId* as a foreign key. In this way, multiple instances of the same parent model can be stored. *ModelInstanceValues* stores the model instance variable values. Here, the combination of *modelId, instanceId*, and *varName* is a primary key. Lastly, we show some fields in all four tables decoded in regular and italic font. Here, the italic indicates the changes occurred after executing a query example (e.g., *fmu_parest*, Section 6). The initial field value is decoded with regular font.

A user can create an instance of the heat pump model (see Section 2) using the following intuitive SQL query in pgFMU:

```
1 SELECT fmu_create('/tmp/hp1.fmu', 'HP1Instance1');
```

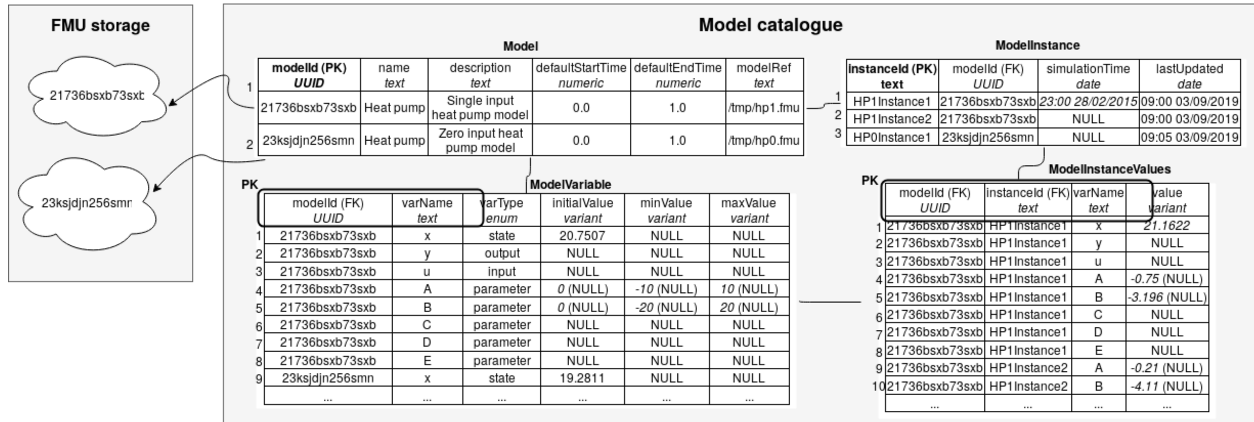

Figure 4: pgFMU model catalogue (filled with example data values)

The result of query execution is shown in Row 1 in *Model*, Row 1 in *ModelInstance*, Rows 1-8 in *ModelVariable* and *ModelInstanceValues*, and the object *21736bsxb73sxb* in *FMU storage*. Similarly, the user can use *fmu_create* to interact with Modelica model files:

```
1 SELECT fmu_create ('HP0Instance1', '/tmp/model.mo');
2 SELECT fmu_create('HP0Instance1', 'model heatpump
      output Real x, ..., y = C*x + D*u;end heatpump;');
```

The result of executing either of these calls corresponds to Row 2 in *Model*, Row 3 in *ModelInstance*, Row 9 in *ModelVariable*, and the object *23ksjdjn256smn* in *FMU storage*. The user can also create a copy of the model instance by using *fmu_copy (instanceId, [InstanceId2])* ↦ *instanceId2* (e.g., when managing many heat pumps of the same type); it takes the initial instance *instanceId* as input, and outputs the copy of this instance with the new instance identifier *InstanceId2* (user-defined or pgFMU-generated). The code snippet below illustrates the usage of *fmu_copy* (leading to changes in Row 2 in *ModelInstance* and Rows 9, 10 in *ModelInstanceValues*):

```
1 SELECT fmu_copy ( 'HP1Instance1', 'HP1Instance2');
```

In pgFMU, the initial copy of the FMU file is reused when either creating a new instance of the same FMU model (Row 2 in *ModelInstance*), copying a model instance, or changing a model state in the database environment. Once loaded (see Figure 4, *FMU storage*), an FMU model is used for all further operations; in this way, we avoid the creation and load of superfluous FMU model files, and we control and manipulate model instances while minimizing memory and computational resources. Algorithm 1 presents the logic behind *fmu_create*.

Furthermore, pgFMU provides a number of utility functions to analyse and manipulate the model instance variable values. For example, the function *fmu_variables (instanceId)* ↦ *(instanceId, varName, varType, initialValue, minValue, maxValue)* returns the details of all variables and parameters of the supplied model instance *instanceId*. The result includes the variables' initial, minimum, and maximum values, which can also be retrieved using the function *fmu_get(instanceId, varName)* ↦ *(initialValue, minValue, maxValue)*. In addition, a user can set the initial, maximum and minimum values of *HP1Instance1 A* and *B* model instance parameters using *fmu_set_initial(instanceId, varName, initialValue)* ↦ *instanceId*, *fmu_set_minimum (instanceId, varName, minValue)* ↦ *instanceId*, and *fmu_set_maximum (instanceId, varName, maxValue)* ↦ *instanceId* (Row 4 in *ModelVariable*, italic font):

```
1 SELECT fmu_set_initial('HP1Instance1', 'A', 0);
2 SELECT fmu_set_minimum('HP1Instance1', 'A', -10);
3 SELECT fmu_set_maximum('HP1Instance1', 'A', 10);
```

Furthermore, a user can retrieve all "HP1Instance1" model variables that serve, for example, as model parameters using the following query (the query output is shown in Table 3):

```
1 SELECT * FROM fmu_variables('HP1Instance1') AS f WHERE
2 f.varType = 'parameter'
```

The model instance can be brought back to its initial state using *fmu_reset (instanceId)* ↦ *instanceId* (see Rows 4, 5 in *ModelVariable*, and Rows 4, 5, 9, 10; the regular font indicates the initial values for the specific model instance). The user can delete either a specific model instance using *fmu_delete_instance (instanceId)*, or an entire FMU model using *fmu_delete_model (modelId)*. In the

Algorithm 1: fmu_create

Input:
Stored model representation: `modelRef`;
Optional: *Unique model instance identifier* `[instanceId]`;
Output:
Model instance identifier `instanceId`;
1: **if** `modelRef` = .fmu file **then**
2: *fmuModel* ← Construct a model object using *PyFMI* function `load_fmu(modelRef)`;
3: **else**
4: **if** `modelRef` = .mo file or inline Modelica model specification **then**
5: *fmuModel* ← Construct a model object using *PyFMI* function `compile_fmu(modelRef)`;
6: **end if**
7: **end if**
8: **if** `instanceId` is not given **then**
9: `instanceId` ← pgFMU-generated `instanceId`;
10: **end if**
11: Store the FMU model file in *FMU storage*;
12: Retrieve model variable names `varName`, types `varType` and values `value` by means of *PyFMI* function *fmuModel.get_model_variables*;
13: Insert the related sets of records into *Model, ModelVariable, ModelInstances, ModelInstancesValues*;
14: Return `instanceId`;

Table 3: ***fmu_variables* example query output**

instanceId	varName	varType	initial-Value	min-Value	max-Value
HP1Instance1	A	parameter	0	-10	10
HP1Instance1	B	parameter	0	-20	20
...	...	...	...	...	...

latter case, all model instances associated with this FMU model will be automatically removed from the database.

6 MODEL PARAMETER ESTIMATION

Model parameter estimation is a key functionality in pgFMU. Model instance parameters in pgFMU can be estimated using *fmu_parest (instanceIds, input_sqls, [pars])* $\mapsto$ *estimationErrors*. It takes a list of model instances *instanceIds* as input, updates the model instances with updated parameter values, and returns the list of estimation errors for each model instance *estimationErrors* (Root Mean Square Errors (RMSEs) by default). The user must also specify a list of SQL queries (*input_sqls*, one SQL query for each model instance) that produce the data to use in the parameter estimation, i.e., model training input and output variable pairs at different time instances. By default, the function estimates all model parameters. Optionally, the user can override this list by supplying a list of specific parameter names *pars*.

For example, the user can estimate the model parameters "A" and "B", and then store the updated model instance in the model catalogue using the following query (output is shown in Rows 4 and 5 in *ModelInstanceValues*, italic font):

```
1 SELECT fmu_parest('{HP1Instance1}', '{SELECT * FROM
      measurements}', '{A, B}')
```

fmu_parest() adopts the functionality of the ModestPy [3] Python package, which uses multiple optimization runs of the Global (denoted as G) and the Local (denoted as LaG) Search algorithms on different subsets of inputs to ensure result optimality even in the case of non-convex problems (later Figure 5 will illustrate the intuition behind this). In pgFMU, we use the ModestPy genetic algorithm implementation as G, and gradient-based method implemented by scikit-learn as LaG.

fmu_parest is designed to reduce the required amount of computation when parameters of multiple FMU instances need to be estimated (Challenge 3). Therefore, it automatically detects the number of model instances supplied and uses different algorithms when estimating parameters of a single model instance (SI scenario) or multiple instances (MI scenario).

Single instance parameter estimation Algorithm 2 provides implementation details for *fmu_parest* within the SI scenario. *fmu_parest* not only estimates the parameters of the FMU model instances, but also validates and updates the model instance with the new parameter values.

Multi-instance optimization In many scenarios, a user has a number of instances of the same model, e.g., 20 instances (*HP1Instance1*, *HP1Instance2*,.., *HP1Instance20*) of the heat pump model *HP1* corresponding to 20 different houses located in the same neighbourhood. In this case, pgFMU can apply its MI optimization when estimating parameters for multiple instances.

For example, the user can estimate the parameters "A" and "B" of *HP1Instance1* and *HP1Instance2* using the following query (leading to Rows 4, 5, 9, and 10 in *ModelInstanceValues*, italic font):

Algorithm 2: fmu_parest_SI

Input:
  Unique model instance identifier: `instanceId`;
  Query to retrieve measured data: `input_sql`;
  Optional: *List of parameters* `[pars]`;
Output:
  `estimationError`;
1: Result set `measurements` ← Execute `input_sql`;
2: *uuid* ← Retrieve FMU model UUID from *ModelInstance* table identified by `instanceId`;
3: **if** `pars` is not given **then**
4:   `pars` ← Retrieve parameter variables from *ModelVariable* table identified by *uuid*;
5: **end if**
6: Retrieve the input variable values from the result set `measurements`;
7: `parsEstimated, estimationError` ← Run G & LaG for `pars`;
8: Update *ModelInstanceValues* with `parsEstimated`;
9: Return `estimationError`;

```
1 SELECT fmu_parest('{HP1Instance1, HP1Instance2}', '{
      SELECT * FROM measurements, SELECT * FROM
      measurements2}', '{A, B}')
```

Figure 5 illustrates the logic behind the MI optimization of *fmu_parest*. This optimization occurs in two stages. During the first stage, we estimate the parameters of *HP1Instance1* (solid blue line). In most cases, only the lower and upper bounds of each model parameter are known, and such bounds are usually defined by real-world physical constraints (e.g., HP performance coefficient cannot be negative). We set the initial parameter values to random numbers between the lower and the upper bounds mentioned above. Then, we follow the steps described in Algorithm 2, i.e., firstly, we run Global Search (G) to reduce the search space. Here, *1_G*, *2_G*, and *3_G* (filled circles) indicate the G iterations for *HP1Instance1*. After finding a good place in the search space, Local Search after Global (LaG) finetunes the parameter values to find optimal ones. We denote LaG iterations as *1_LaG*, *2_LaG*, and *3_LaG* (empty circles). *3_LaG* (exemplified by Rows 4, 5 in *ModelInstanceValues*) is the optimal parameter values of *HP1Instance1*.

The next stage is to perform parameter estimation for *HP1Instance2* (red dashed line). First, we check that the model instances belong to the same parent FMU. Next, we check whether the condition for the MI optimization invocation holds, i.e., we only

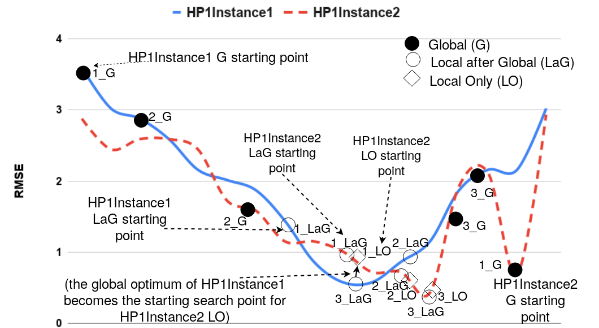

Figure 5: *fmu_parest* MI optimization

invoke the MI optimization after ensuring similarity (by calculating the L2 norm) between the input (and output) measurements of *HP1Instance1* and *HP1Instance2*. L2 norm (or the Euclidean norm) is one of the simplest and widely known, but still accurate and robust metrics for measuring the similarity between time series [26]. It is important to check the similarity between model instances time series to make sure the optimal solutions for these model instances lie within the same neighbourhood (see Figure 5). In case the difference between measurement time series is greater than a threshold, we do not invoke the MI optimization, and instead, run Algorithm 2 (a combination of G+LaG) for every instance. The *1_G*, *2_G*, and *3_G* points (filled circles) indicate the G iterations, while *1_LaG*, *2_LaG*, and *3_LaG* (empty circles) illustrate the LaG iterations. *3_LaG* would, in this case, be the optimal parameter values of *HP1Instance2*.

If the measured time series are sufficiently similar, i.e., the difference between them is less than *threshold*, we invoke the MI optimization. This means the optimal parameters values of *HP1Instance1* (solid blue line, *3_LaG*) become the initial parameter values of *HP1Instance2* (red dashed line, *1_LO*). In this way, we run Local Only Search (LO) (which is essentially the same algorithm as LaG, but with different initial parameter values) to obtain optimum parameter values of *HP1Instance2*, as its solution lies within the neighbourhood of the best solution of *HP1Instance1*. The points *1_LO*, *2_LO*, and *3_LO* (diamonds) are converging to those found by LaG. *3_LO* (exemplified by Rows 9, 10 in *ModelInstanceValues*) is the optimum parameter values of *HP1Instance2*. Algorithm 3 describes MI optimization for n model instances.

In pgFMU, the MI optimization significantly speeds up parameter estimation for multiple model instances. This speed-up is possible because G is much more expensive than LaG (see Section 8), and instead of G+LaG only LO is run. pgFMU provides the gradient-based Local Search algorithm with the good initial parameter values. In [3] authors emphasize, that "the initial global search would not be needed if the approximate initial values of parameters were known. In such a case the gradient-based methods would easily outperform GA [genetic algorithm]". As it will be shown later in Section 8, this statement holds. It should also be noted that the quality of the solution is dependent on the internal ModestPy algorithms. Within pgFMU, we adapt and enhance the functionality of ModestPy to best capture user preferences, streamline parameter estimation, and facilitate user interaction. The empirical evaluation of the MI parameter estimation shows identical accuracy with and without MI optimization.

7 MODEL SIMULATION

Users can simulate models by utilizing the function *fmu_simulate (instanceId, [input_sql], [time_from], [time_to])* $\mapsto$ *(simulationTime, instanceId, varName, values)*, which performs simulation on the supplied model instance and returns simulation results as a table of a timestamp, a model instance identifier, a variable name, and the variable's simulated value. By default, the simulation results for all state and output variables are returned. It is also possible to supply a time series of model input variable values by specifying an SQL query *input_sql*. If desired, the user can also specify a time window for simulation using the *time_from* and *time_to* parameters; otherwise, the start and end time will be determined by *defaultStartTime* and *defaultEndTime*. The system raises an error, e.g., if insufficient model input time series or an incomplete simulation time interval is provided.

Algorithm 3: fmu_parest_MI

Input:
List of unique model instance identifiers: `instanceIds`;
List of queries to retrieve measured data: `input_sqls`;
Optional: *List of parameters [pars]*, float *[threshold]*.
Output:
List of `estimationErrors`;

```
1:  if pars is not given then
2:      pars ⟵ Retrieve parameter variables from ModelVariable
        table identified by InstanceIds;
3:  end if
4:  for i = 0 to length (instanceIds)-1 do
5:      Result set measurements[i] ⟵ Execute input_sql[i];
6:      parsEstimated[0], estimationError[0] ⟵ Run
        fmu_parest_SI for instanceIds[0];
7:      for i = 1 to length (instanceIds)-1 do
8:          if modelId[0] ! = modelId[i] then
9:              Run fmu_parest_SI for instanceIds[i];
10:         else
11:             δ = L2 norm of measurements[i] from
                measurements[0];
12:             if δ ≥ threshold then
13:                 Run fmu_parest_SI for instanceIds[i];
14:             else
15:                 Update all the initial parameter values of
                    instanceIds[i] to parsEstimated[0];
16:                 parsEstimated[i], estimationError[i] ⟵
                    Run LO for instanceIds[i];
17:             end if
18:         end if
19:     end for
20:     Update ModelInstanceValues with parsEstimated;
21: end for
22: Return estimationErrors;
```

Table 4: *fmu_simulate* example query output

simulationTime	instanceId	varName	value
08:00 28/02/2015	HP1Instance1	x	20.7507
08:30 28/02/2015	HP1Instance1	y	0.0041
...	...	...	...

For example, the user can simulate the model by supplying model input values from the table *measurements* (Table 6) using the following query (also updating *ModelInstanceValues*, Row 1 and *ModelInstance*, Row 1, values in italic):

```sql
SELECT simulationTime, instanceId, varName, value
FROM fmu_simulate('HP1Instance1','SELECT * FROM
    measurements') WHERE varName IN ('y', 'x');
```

Table 4 shows the output of this query. If needed, the user can filter values of the desired columns using the standard WHERE clause predicates. We note that returning variable values as separate columns is not possible due to PostgreSQL limitations. In PostgreSQL, UDFs require fixed output schemas; flexible schemas are not supported in a convenient and easy-to-use way.

If multi-instance simulations are needed, the user can call a standard LATERAL join:

```sql
SELECT * FROM generate_series(1, 100) AS id,
LATERAL fmu_simulate('HP1Instance' || id::text ,
'SELECT * FROM measurements') AS f
```

Using *fmu_simulate*, model simulation is performed in two stages. In the first stage, an input object required for model simulation is created according to the FMU meta-data. This object consists of a set of input time series, automatically transformed for each input variable by taking into account their data types and variabilities (Challenge 2). In the second stage, an underlying model instance is simulated within the time interval (user-specified or from the model catalogue) while feeding the simulation algorithm with an input object. The simulation results are then emitted as a table (Table 4). Algorithm 4 depicts the logic behind *fmu_simulate*.

The UDFs described in Sections 5, 6, and 7 can be combined to solve specific problems. For instance, the example regarding HP temperature prediction can be specified using a single query (producing the same results and updates as *fmu_create, fmu_parest*, and *fmu_simulate* example queries from Sections 5, 6, and 7):

```
1 SELECT time, value
2 FROM fmu_simulate (fmu_parest {fmu_create
3  ('HP1Instance1', 'C:\temp\hp1.fmu')},
4 '{SELECT * FROM measurements}', '{A, B}'),
5 'SELECT time, 'u' AS varName, value
6  FROM generate_series('''2015-01-01'''::timestamp,
7 '''2015-01-02'''::timestamp, '''1 hour'''::
8 interval) AS time WHERE varName = 'x'
```

In addition, there are few important points. Firstly, all pgFMU UDFs are independent of each other and can be used in any order. Similar to other modelling software, e.g., Matlab and JModelica, the user is only requested to create the in-DBMS instance of an FMU model. pgFMU provides full flexibility when configuring user workflows: if parameter estimation is not required, the user can simulate the model right away, or change the order of actions and perform model simulation followed by parameter estimation. Next, in pgFMU, we eliminate explicit I/O operations (e.g. historical measurements import, or simulation results export from a third-party software into DBMS) as all the computations are done "in-place" inside DBMS. This affects the essential operations (parameter estimation, model simulation) due to the data-driven nature of these operations. Furthermore, when operating many instances of the same model, we eliminate the necessity to load the same FMU file multiple times. We store one single FMU model, and operate with in-DBMS model instances. These instances share the essential information about the initial FMU through model catalogue, and can be considered as sufficient substitutes of FMUs. pgFMU is also able to operate several in-memory FMU model files at a time. Lastly, as we use prepared SQL queries, we avoid the repeated reevaluation of database queries for measurements retrieval.

Algorithm 4: fmu_simulate

Input:
Unique model instance identifier: `instanceId`;
Optional: *Query to retrieve measured data: [input_sql], [time_from], [time_to];*

Output:
Output table (simulationTime, instanceId, varName, values);

1: *uuid* ← Retrieve FMU model UUID from *ModelInstance* table identified by `instanceId`;
2: `fmuModel` ← Load FMU model identified by *uuid* from *FMU storage*;
3: Result set `measurements` ← Execute `input_sql`;
4: *inputs* ← Retrieve input variables from *ModelVariable* identified by *uuid*;
5: `input_object` ← Empty hash map of (name, time series) pairs;
6: For each name *n* in *inputs*: Insert (*n*, `measurements[n]`) into *input_object*;
7: **if** `time_from` and `time_to` are not given **then**
8:     (`time_from, time_to`) ← (`Model.DefaultStartTime, Model.DefaultEndTime`) where Model.modelid = *uuid*;
9: **end if**
10: *result* ← `fmuModel.simulate(input_object, time_from, time_to)`;
11: *time* ← `(time_from, time_to)`
12: `output` ← [];
13: **for** time *i* in *result.time* **do**
14:     **for** varName in *result.variables* **do**
15:         Append (i, instanceId, varName, result[varName][i]) to `output`;
16:     **end for**
17: **end for**
18: Return `output`;

8 EXPERIMENTAL EVALUATION

In this section, we evaluate pgFMU in real-world circumstances. Firstly, we describe our experimental setup. Then, we present the results of the evaluation with regard to model quality, performance, and usability.

8.1 Experimental Setup

As a baseline, we follow a traditional workflow for model storage, calibration, simulation, and validation, and perform the steps described in Figure 1. We consider two scenarios: *single instance (SI) scenario* and *multi-instance (MI) scenario*. In the SI scenario, we execute parameter estimation and simulation using a single model instance only. In the MI scenario, we execute parameter estimation and simulation for many model instances.

Three main system configurations are compared: (1) workflow execution within a Python IDE (referred as *Python*), (2) workflow execution using the non-optimized version of pgFMU (referred as *pgFMU-*), and (3) workflow execution using the optimized version of pgFMU (referred as *pgFMU+*). *Python* is based on the usage of standard Python packages. In *pgFMU-* configuration we use the pgFMU system with no multi-instance optimization activated, and *pgFMU+* takes advantage of the MI optimization. All three configurations are evaluated using Ubuntu 17.1 OS on a Lenovo ThinkPad with a four-core Intel i7 processor, and 8 GB of DDR3 main memory. For parameter estimation within pgFMU, we utilize genetic algorithm (GA) for Global Search and sequential quadratic programming (SQP) for Local Search. The study argues [3] that this combination of algorithms produces optimal results in terms of accuracy and performance. For GA, we use default settings with a fixed randomly derived seed. For SQP, the default settings were used as well. More information about default parameters settings is available on the library homepage.

Generalizability was one of the main criteria for choosing the test models. The models should represent real-world physical systems with varying numbers of inputs and outputs, and different physical meaning of model parameters. We follow a workflow process identical to the one outlined in Figure 1, i.e., we estimate the unknown parameters of the model (model parameter estimation), validate the model with regard to the real input measurements (model validation), and simulate the model to predict

the values of the model state variable (model simulation). We have chosen three different FMU models, each of them representing a real-world physical system. These models are denoted *HP0, HP1*, and *Classroom*, respectively. *HP1* corresponds to the running example model described in Section 2. *HP0* is a modification of *HP1* with zero inputs, such that we keep the heat pump power at a constant rate. *Classroom* is a thermal network model [17] represented by a classroom in a 8500 m^2 university building at the SDU Campus Odense (Odense, DK). Table 5 summarizes the inputs, outputs and parameters of the three models. For experimental evaluation within the MI scenario, we construct 100 synthetic datasets for each FMU model. We multiply the original dataset time series values with a constant $delta$ from the numerical range $\delta \in \{0.8, ..., 1.2\}$, meaning we amplify or decrease the numerical values by up to 20% while ensuring the same data distribution as the original datasets. In Section 6 we explain the reasoning behind such numerical range of δ. We also ensure that the datasets respect the physical constraints of the real-world systems.

8.2 Model quality

In this subsection, we compare and evaluate model quality for *Python*, *pgFMU-*, and *pgFMU+*. We perform parameter estimation and model simulation within the *SI scenario* and *MI scenario*.

SI scenario. Within *Python*, parameter estimation is performed using the ModestPy package, whereas *pgFMU-* and *pgFMU+* utilize the *fmu_parest* UDF. *HP1* was calibrated using the NIST dataset [18]. *HP0* was calibrated using the same dataset with u being kept at a constant rate of 1.38%. For both models, we estimate the parameters based on the hourly aggregated data from February 1-21, while using February 22-28 for validation. The *Classroom* model was calibrated using measurements data from University building O44 in Odense, DK. Table 6 shows an excerpt of the datasets for all three models. We compare parameter values and error value for *HP0, HP1* and *Classroom* for *Python*, *pgFMU-*, and *pgFMU+*. Table 7 shows this comparison. To evaluate the quality of the model, we use the RMSE metric. RMSE and Mean Absolute Error (MAE) are two commonly used metrics for model evaluation. However, a study [19] argues that RMSE is more appropriate to be used when unfavourable conditions should be given a higher weight, i.e., RMSE penalizes large errors stricter than MAE. In our case, we want to distinguish every occurrence when a model fails to secure a good fit with the measured data, therefore, we choose RMSE for model quality evaluation.

For *HP0* (Table 7), the model parameter values have converged to the same values within *Python*, *pgFMU-* and *pgFMU+*. The RMSEs when performing parameter estimation for *Python*, *pgFMU-*, and *pgFMU+* are near identical (the relative difference is only 0.013%). For *HP1*, the RMSEs in all three configurations are exactly the same, and for *Classroom* the relative difference is at most 0.018%. We consider these differences negligible. Thus, *pgFMU-* and *pgFMU+* handles single model workflow computations with the same accuracy as *Python*. The identical accuracy for all three configurations is achieved through the usage of the same Python ModestPy library; however, both *pgFMU-* and *pgFMU+* use a modified version of ModestPy to provide generalizability and handle all types of FMU models, and perform data binding and pre-processing steps discussed in Section 2.

MI scenario. For the MI scenario, we compare RMSE values for 100 instances of each model (*HP0, HP1* and *Classroom*) side-by-side. Each model instance is supplied with a synthetic dataset based on the measured data. For *Classroom*, the RMSE values for all three configurations are matching, resulting in the same average RMSE values (1.61°C). For *HP1*, RMSE values differ a bit more, but are still very close, with either *Python*, *pgFMU-*, or *pgFMU+* as the better one. The average RMSE values yield in 2.03°C for all three configurations. Thus, the model quality is also the same here. We observe a similar behaviour for *HP0*, where for *pgFMU-* and *Python* the average RMSE is 0.68°C, while *pgFMU+* estimates the model parameters with 0.66°C average accuracy.

When enabling MI optimization, one must remember the conditions for this feature to produce acceptable results. By default, parameter estimation is performed using Algorithm 2 (Section 6), unless the user alters the threshold value. In our case, we have set the threshold to 20% (Section 8.1) based on the series of experiments reflected in Figure 6. In this Figure, the x-axis represents dataset dissimilarity in terms of L2 norm distances, the y-axis encodes the corresponding RMSE values, and the secondary y-axis shows the execution time of G+LaG and LO. RMSE is represented using the same unit as the dependent variable (for *HP1*, the indoor temperature in °C). The Figure shows that the execution time of G+LaG is significantly larger than LO. We see that G takes approximately 90% of the execution time (considering that LaG and LO are the same algorithms, with different initial parameter values). The Figure shows that there is no difference in G+LaG and LO RMSEs until maximum dissimilarity reached approximately 30%; after this, the difference grows linearly. This is because the optimal solution for *instanceId[0]* used as initial parameter values for the remaining models *instanceId[1], ..., instanceId[n-1]* (see Algorithm 3) was not able to provide satisfactory results. Further, the choice of a threshold value is always contextually dependent, and a user has to decide about the acceptable RMSE values.

Combining pgFMU and MADlib. To improve model quality, pgFMU can be used in combination with other in-DBMS analytics tools, e.g., MADlib. If in our *Classroom* model the number of occupants in the room is unknown, we can use MADlib to predict occupancy, e.g., using the ARIMA model and the following query to train the model:

```
SELECT arima_train(
'occupants', -- Source table
'occupants_output', -- Output table
'time', -- Timestamp column
'value'); -- Timeseries column
```

In this experiment, we used the original dataset to train the model and perform the prediction. We divided the dataset into training (80%) and validation (20%) sets. We created two model instances: without occupancy information, and with occupancy values predicted by the MADlib ARIMA model. Then, we simulated the two models, and compared model RMSEs. The *Classroom* model

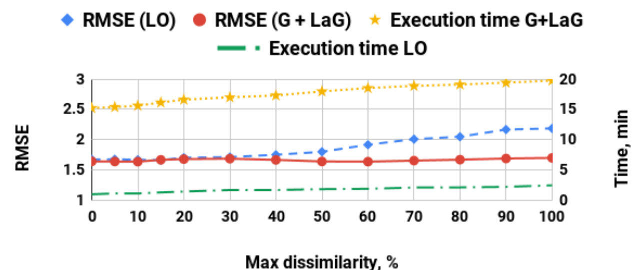

Figure 6: Avg. RMSE & execution time overhead of LO & G+LaG for datasets of different dissimilarity (HP1 model)

Table 5: FMU models

ModelID	Measurements dataset	Inputs	Outputs	Parameters
HP0	NIST Engineering Lab's Net-Zero Energy Residential Test Facility	No inputs	HP power consumption *y*, Indoor temperature *x* (state variable).	Thermal capacitance *Cp*, thermal resistance *R*.
HP1	NIST Engineering Lab's Net-Zero Energy Residential Test Facility	HP power rating setting in the range [0 .. 100%] *u*	HP power consumption *y*, Indoor temperature *x* (state variable).	Thermal capacitance *Cp*, thermal resistance *R*.
Classroom	Data from the classroom in the test facility in Odense, DK	Solar radiation *solrad*, outdoor temperature *tout*, number of occupants *occ*, damper position *dpos*, radiation valve position *vpos*.	Indoor temperature *t* (state variable)	Solar heat gain coeff. *shgc*, zone thermal mass factor *tmass*, ext. wall thermal resistance *RExt*, occupant heat generation effectiveness *occheff*.

Table 6: Dataset for *HP0, HP1* (top), *Classroom* (bottom)

No	Timestamp	x	y	u
1	2015/02/01 00:00	20.7507	0	0
2	2015/02/01 01:00	23.6231	0.1381	0.0177
...	...	...	...	...

No	Timestamp	T	solrad	Tout	occ	dpos	vpos
1	2018/04/04 08:00	21.5727	364.37	11	19.7	0	13.165
2	2018/04/04 08:30	20.8667	396.05	10.5	20.033	0	19.4
...	...	...	...	...	...	...	...

with the occupancy values predicted by MADlib ARIMA showed up to 21.1% increased accuracy in terms of RMSE.

Reversely, pgFMU can be used to improve the quality of traditional ML models. In the next experiment, we used the indoor temperatures of the *Classroom* computed using pgFMU to increase the accuracy of the logistic regression model that identifies the position (open/closed) of the ventilation damper *dpos*. When we include indoor temperature *t* into the feature vector of the model, this yields 5.9% increased model accuracy.

8.3 Performance evaluation

In this subsection, we look into the performance comparison of *Python*, *pgFMU-*, and *pgFMU+* for *HP0, HP1*, and *Classroom* within SI and MI scenarios.

Single instance scenario. To evaluate pgFMU performance within the SI scenario, we compare the execution time for all three models. Table 8 shows this comparison. For all the models, we do not observe a significant difference in execution time between *Python*, *pgFMU-* and *pgFMU+* (for *HP0*, *Python* is faster by 0.14%, for *HP1* and *Classroom* *pgFMU-* and *pgFMU+* perform better by 0.10%, and 0.12%, respectively). We conclude that within *Python*, *pgFMU-*, and *pgFMU+* the performance is practically identical, as expected. However, in Table 8 we see that model calibration takes more than 99% of execution time. This imposes limitations when estimating the parameters of multiple model instances.

Table 7: SI scenario, model calibration comparison

	Python		*pgFMU-, pgFMU+*	
	Param. values	RMSE	Param. values	RMSE
HP0	Cp: 1.53	0.7701	Cp: 1.53	0.7702
	R: 1.51		R: 1.51	
HP1	Cp: 1.49	0.5445	Cp: 1.49	0.5445
	R: 1.481		R: 1.481	
Classroom	RExt: 4	1.6445	RExt: 4	1.6442
	occheff: 1.478		occheff: 1.478	
	shgc: 3.246		shgc: 3.246	
	tmass: 50		tmass: 50	

MI scenario. For the MI scenario, we use *Python*, *pgFMU-* and *pgFMU+*, and scale the number of model instances to 100 for each FMU model. We match the synthetic dataset with each FMU model instance. Figure 7 shows the execution time for *Python*, *pgFMU-*, and *pgFMU+* for *HP0, HP1* and *Classroom*. The execution workflow of storing, calibrating, simulating, and validating 100 *HP0* model instances takes 1083.7 min (18.06 hours) for *Python*, 1073.7 min (17.9 hours) for *pgFMU-*, and 204.2 min (3.4 hours) for *pgFMU+*. As we can see, *pgFMU+* outperforms *Python* and *pgFMU-* by 5.31x. The execution workflow of storing, calibrating, simulating, and validating 100 *HP1* model instances takes 1329.63 min (22.16 hours) for *Python*, 1319.48 min (21.97 hours) for *pgFMU-*, and 241.47 min (4.02 hours) for *pgFMU+*. In this case, *pgFMU+* outperforms *Python* and *pgFMU-* by 5.51x. We observe an even larger difference in workflow execution time for 100 *Classroom* model instances. It takes 1380.68 min (23.01 hours) for *Python*, 1370.95 min (22.85 hours) for *pgFMU-*, and 163.6 min (2.73 hours) for *pgFMU+*; *pgFMU+* is faster by 8.43x.

As we can see, the execution time grows linearly with more model instances, but the growth rate is different for different models. The common pattern for all three models is that *Python* and *pgFMU-* exhibit a similar growth rate. For *pgFMU+*, the execution time also grows linearly, but slower compared to *Python* and *pgFMU-*. This means *pgFMU+* performs user-defined workflows based on the FMU model calibration and simulation on average 6.42x faster. Such gains in the runtime are achieved through a number of optimization steps described in Sections 6 and 7.

8.4 Usability

We conducted usability tests to evaluate how the functionality of pgFMU reflects user needs. We asked a group of 6 PhD-candidates and 24 master students from four different universities in Denmark, Poland, Spain, and Belgium to individually perform the task of *HP1* and *Classroom* model calibration and simulation. All the participants were asked to complete the SI workflow based on Figure 1 using *Python* and *pgFMU*. The execution of the MI workflow was optional. During the usability testing, we recorded the issues with both configurations, and observed the learning curve of the participants as they progressed with the task. All participants were timed.

Table 8: Configurations comparison, SI scenario

ID	Operation	execution time, s					
		HP0		HP1		Classroom	
		Python	pgFMU±	Python	pgFMU±	Python	pgFMU±
1	Load FMU	0.02	0.025	0.02	0.021	0.03	0.03
2	Read historical measurements & control inputs	0.02	0.021	0.03	0.031	0.04	0.041
3	(Re)calibrate the model	842.99	844.18	834.68	833.88	830.2	829.16
4	Validate and update FMU model	0.01	-	0.01	-	0.01	-
5	Simulate FMU model	0.16	0.214	0.2	0.22	0.35	0.44
6	Export predicted values to a DBMS	0.06	-	0.06	-	0.05	-
	Total	**843.26**	**844.44**	**835**	**834.15**	**830.68**	**829.67**

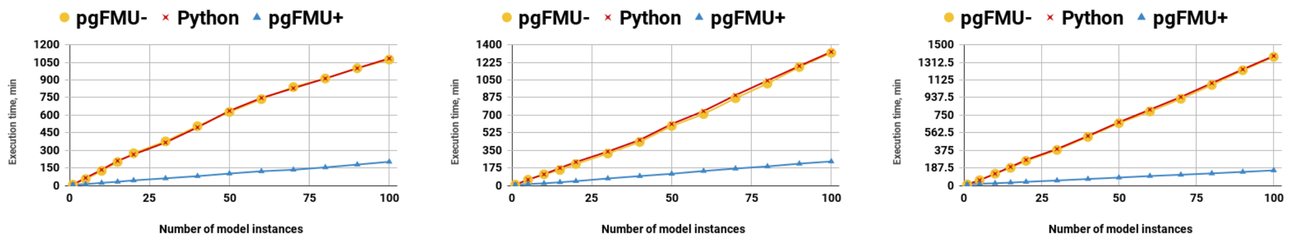

Figure 7: *HP0, HP1, Classroom* (from the left to the right) parameter estimation execution time

In the beginning, the users were asked to answer a pre-assessment questionnaire aimed at identifying their knowledge about physical systems modelling. By using the scale from 1(very little) to 5(very much), the participants answered the following questions:

Q1. How familiar are you with energy systems and physical system modelling?
Q2. How familiar are you with model simulation and calibration?
Q3. How familiar are you with model simulation software?
Q4. How comfortable are you with using Python IDE(s)?
Q5. How comfortable are you with using SQL?

Based on the participants' answers, they did not consider themselves experts in the energy domain (Q1). Only 2 people estimated the familiarity with energy systems as "very much". For Q1, Q2 and Q3, the majority (27 out of 30 for Q1 and Q2, 26 out of 30 for Q3) ranked their knowledge in energy systems, model simulation and calibration processes, and model simulation software as "very little" or "little". When speaking about programming environments, 25 out of 30 students knew "much" or "very much" about the SQL, with only 14 out of 30 giving the same score to the Python. We concluded that graduate and post-graduate students possessed more knowledge about SQL, and felt more comfortable with using SQL-based functions rather than Python packages.

For this session, we set the time limit to 3 hours. The participants tested the *pgFMU* functionality first, then the *Python*. All participants but one were able to finalize the task within the defined time range. Figure 8 illustrates the time distribution for every user performing the steps described in Figure 1 for *HP1* and *Classroom* models. With pgFMU it took under 20 minutes for all participants to become familiar with the syntax, and complete the task. The minimum learning time for pgFMU was reported to be 9.6 minutes, and the maximum was 17.6 minutes, respectively. On average, with pgFMU the participants finished the task (excluding the runtime) 11.74x faster than with *Python*.

The main criticism regarding the suggested setup was the necessity to use multiple Python packages, and the inability to

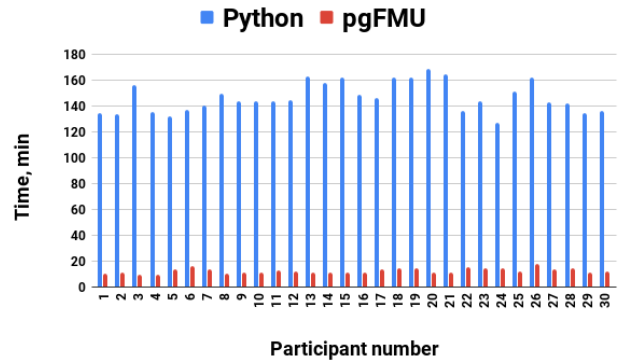

Figure 8: Users learning and development time (combined)

perform all the tasks with a single programming package/tool. The whole workflow was described as "intuitive" and "easy to understand", but the need to use different libraries was characterized as "confusing" and "unsettling". One of the participants reported that both PyFMI and ModestPy packages in their documentation use domain-specific language which was difficult to grasp. Table 9 shows highlighted feedback that the participants expressed about the strengths and weaknesses of both systems.

At the end, all users were asked to answer a post-assessment questionnaire aimed at identifying participants' opinion regarding the functionality of *Python* and *pgFMU*. By using the scale from 1(very unsatisfactory) to 5(very satisfactory), the participants were asked to answer the following questions :

Q1. How was it to retrieve information about model variables?
Q2. How was it to set model parameters?
Q3. How was it to calibrate the model?
Q4. How was it to simulate the model?
Q5. How was your satisfaction with systems' functionality?

Table 9: Strengths and weaknesses of *Python* and *pgFMU*

Strong points of *Python*	Strong points of *pgFMU*
"Better to debug, analyze"; "More functionality"; "Control over program flow"; "Data visualization option";	"Data and model in one place"; "Easy to run and understand"; "Simplicity, no need to use or import external tools"; "Familiar SQL syntax";
Weak points of *Python*	Weak points of *pgFMU*
"A lot of unknown new modules and packages"; "No one ready package to do everything"; "A lot of code to set up configuration, some functions not intuitive"; "You need practise[*sic*] to understand";	"Not so much configuring available"; "I don't see any significant [*weak points*] besides maybe installation of the package on postgres [*PostgreSQL*];" "Specific database implementation";

The questionnaire results show a clear advantage of using *pgFMU* over *Python*. When performing the workflow depicted in Figure 1, *pgFMU* scores better when retrieving (3.7 out of 5 for pgFMU vs 3.1 out of 5 for Python) or setting the model variables (3.83 out of 5 vs 3.26 out of 5 respectively), calibrating (3.66 out of 5 vs 3.1 out of 5 respectively) and simulating the model. In particular, the model simulation functionality of *pgFMU* was commended, scoring 4.17 out of 5 (for the same operation, *Python* scored only 3.53 out of 5). *pgFMU* outperforms *Python* in terms of overall participants' satisfaction, scoring 4.2 and 3.6 out of 5, respectively.

Lastly, *pgFMU* requires substantially fewer lines of code than does *Python*; Table 1 shows this comparison. In addition, *pgFMU* does not require any customization, meaning it is capable of working with any number of model inputs, outputs and parameters without UDF adjustment for every specific model or use case. On the contrary, *Python* requires the user to manually retrieve and match model inputs with measurements data.

9 CONCLUSION AND FUTURE WORK

This paper presents pgFMU - the first DBMS extension to support storage, simulation, calibration, and validation of physical system models defined as Functional Mock-up Units (FMUs) within a single DBMS environment. This extension is developed for cyber-physical data scientists and cyber-physical software developers. For such users, pgFMU provides a set of powerful User-Defined Functions (UDFs) invoked by traditional SQL queries. The UDFs are designed as stand-alone functions within pgFMU, and can be used in a user-defined sequence. Furthermore, pgFMU provides efficient functionality to store, simulate an arbitrary number of FMU model instances, and estimate the parameters of such instances. The aforementioned properties all contribute to supporting FMU model simulation and parameter estimation tasks within a DBMS environment. Due to its in-DBMS implementation, pgFMU demonstrates increased performance (up to 8.43x, and on average 6.42x faster for multi-instance workflows) for data-dependent workflows and improves user productivity (11.74x times faster in terms of development time). Moreover, it can increase Machine Learning model accuracy by up to 21.1% when used in combination with existing in-DBMS analytics tools, e.g., MADlib.

pgFMU has been tested by the entry level cyber-physical data scientists. The usability testing showed positive results, reporting the development time more than an order of magnitude lower than the traditional approach. As reported by the participants, pgFMU was capable of addressing the major point of dissatisfaction, namely the variety of software packages and libraries the user is obliged to use, and the domain-specific nature of such packages. pgFMU simplifies the overall analytical procedure, and minimizes the efforts required to specify and calibrate a particular model; the users are able to perform the necessary operations with approximately 22x fewer code lines. pgFMU was also reported to provide a more intuitive way of scripting and a better data organisation.

Future work will continue the development of functionality to support in-DBMS FMU-based dynamic optimization. This includes the adoption of various model predictive control means, covering the optimization of control inputs. Additionally, we will look into FMU integration challenges in the Big Data setting, including just-in-time (JIT) FMU compilation to optimize user queries, and scheduling FMU execution on multi-core multi-node environments.

REFERENCES

[1] Ilic, M. D., Xie, L., Khan, U. A., & Moura, J. M. 2010. Modeling of future cyber–physical energy systems for distributed sensing and control. In *IEEE Trans. on SMC-Part A: Systems and Humans.*

[2] Functional Mock-up Interface. 2018. https://fmi-standard.org/

[3] Arendt, K., Jradi, M., Wetter, M., & Veje, C. T. 2018. ModestPy: An Open-Source Python Tool for Parameter Estimation in Functional Mock-up Units. In *American Modelica Conference.*

[4] MATLAB & Simulink. 2019. https://www.mathworks.com/

[5] Modelica Association. 2019. https://www.modelica.org/association/

[6] EnergyPlus. 2019. https://energyplus.net/

[7] Python.org. 2019. https://www.python.org/about/

[8] Åkesson, J., Gäfvert, M., & Tummescheit, H. 2009. Jmodelica—an open source platform for optimization of modelica models. In *MATHMOD.*

[9] Momjian, B. 2001. PostgreSQL: introduction and concepts.

[10] Fritzson, P., Aronsson, P., Pop, A., Lundvall, H., Nystrom, K., Saldamli, L., ... & Sandholm, A. 2006. OpenModelica-A free open-source environment for system modeling, simulation, and teaching. In *IEEE ISCSIC.*

[11] Bonvini, M., Wetter, M., & Sohn, M. D. 2014. An fmi-based framework for state and parameter estimation. In *10th International Modelica Conference.*

[12] Frazzetto, D., Nielsen, T. D., Pedersen, T. B., & Šikšnys, L. 2019. Prescriptive analytics: a survey of emerging trends and technologies. In *VLDBJ.*

[13] Fritzson, P., & Bunus, P. 2002. Modelica-a general object-oriented language for continuous and discrete-event system modeling and simulation. In *35th Annual Simulation Symposium.*

[14] Andersson, C., Åkesson, J., & Führer, C. 2016. PyFMI: A Python package for simulation of coupled dynamic models with the functional mock-up interface.

[15] McKinney, W. 2012. Python for data analysis: Data wrangling with Pandas, NumPy, and IPython.

[16] PostgreSQL + Python. Psycopg. 2019.http://initd.org/psycopg/

[17] Arendt, K., Veje, C. T. 2019. MShoot: an Open Source Framework for Multiple Shooting MPC in Buildings. In *IBPSA.*

[18] Healy, W., Fanney, A. H., Dougherty, B., Payne, W. V., Ullah, T., Ng, L., and Omar, F. 2017. Net Zero Energy Residential Test Facility Instrumented Data; Year 1. https://doi.org/10.18434/T46W2X

[19] Chai, T., & Draxler, R. R. 2014. Root mean square error (RMSE) or mean absolute error (MAE)?–Arguments against avoiding RMSE in the literature. In *GMD.*

[20] Andersson, C., Führer, C., & Åkesson, J. 2015. Assimulo: A unified framework for ODE solvers. In *IMACS.*

[21] Webb, B. 2008. Sas in-database processing with teradata: an overview of foundation technology. https://support.sas.com/resources/papers/teradata08.pdf

[22] Meliou, A., & Suciu, D. 2012. Tiresias: the database oracle for how-to queries. In *SIGMOD.*

[23] Šikšnys, L., & Pedersen, T. B. 2016. Solvedb: Integrating optimization problem solvers into sql databases. In *SSDBM.*

[24] variant: Variant data type for PostgreSQL / PostgreSQL Extension Network. 2019. https://pgxn.org/dist/variant/0.7.0/doc/variant.html

[25] The Telecommunications Standardisation. 2019. https://www.ietf.org/rfc/rfc4122.txt

[26] Wang, X., Mueen, A., Ding, H., Trajcevski, G., Scheuermann, P., & Keogh, E. 2013. Experimental comparison of representation methods and distance measures for time series data. In *DMKDFD.*

[27] Hellerstein, J. M., Ré, C., Schoppmann, F., Wang, D. Z., Fratkin, E., ... & Kumar, A. 2012. The MADlib analytics library: or MAD skills, the SQL. In *VLDB.*

[28] SQL Server Machine Learning Services (Python and R). 2019. https://docs.microsoft.com/en-us/sql/advanced-analytics/