Prescriptive Analytics for Physical Systems Models

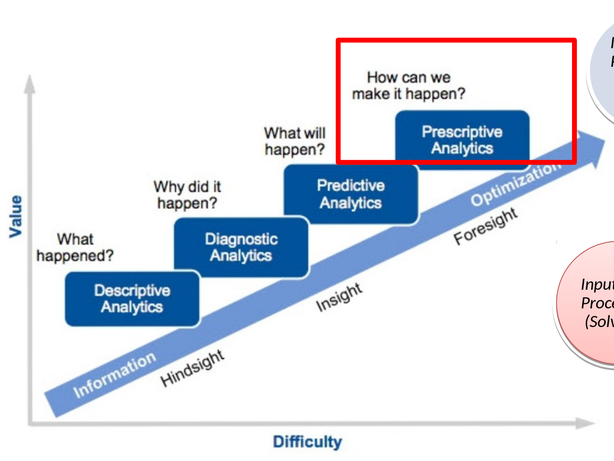
Olga Rybnytska[1]

Supervisors: Torben Bach Pedersen[1], Robert Wrembel[2]

Aalborg University [1]
Aalborg, Denmark

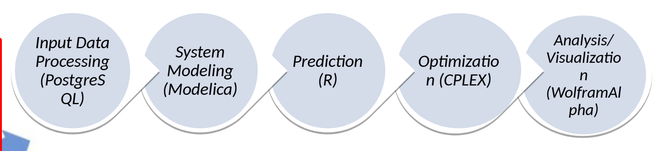
Poznan University of Technology[2]
Poznan, Poland

Prescriptive Analytics: Motivation and Challenges

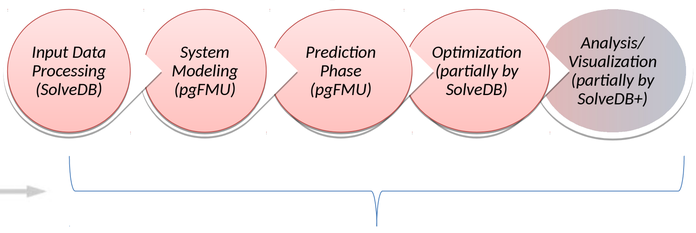


Analytic Ascendancy Model [1]

Example of a Typical Workflow (adapted from [3]):



Our Approach:



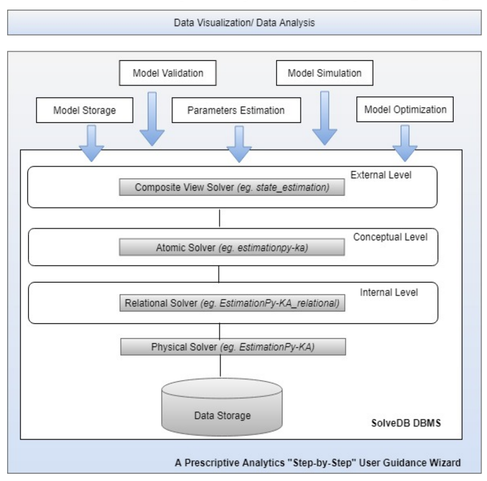
Unified PA tool (an integration of SolveDB and upgraded pgFMU version to support Dynamic Optimization tasks)

SolveDB DBMS

- SolveDB [2] - a PostgreSQL-based DBMS with the native support for in-DBMS optimization problem solving.
- Provides a set of built-in solvers for Linear Programming (LP)/Mixed Integer Programming (MIP), Global Optimization (GO) and domain-specific problems.
- Allows SQL-based problem specification and integrates solver into DBMS backend, therefore, making database-based problem solutions more easy and user-friendly.

```
SOLVESELECT col_name [, ...] IN ( select_stmt ) [AS alias]
      [ WITH col_name [, ...] IN ( select_stmt )  AS alias [, ...] ]
[ MINIMIZE ( select_stmt ) [ MAXIMIZE ( select_stmt ) ] |
  MAXIMIZE ( select_stmt ) [ MINIMIZE ( select_stmt ) ] ]
[ SUBJECTTO ( select_stmt ) [, ...] ]
[ USING solver_name [. ...] [( param[:= expr] [, ...] )] ]
```

SolveDB query example [2]



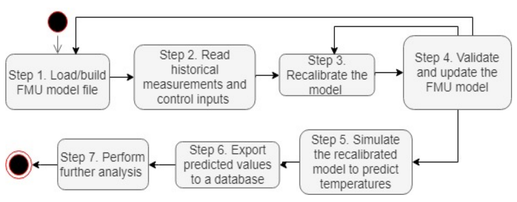
Extended SolveDB Architecture

System Modeling and Prediction Extension

Motivation:

- Complexity of model-driven analytical tasks.
- Lack of Functional Mockup Unit (FMI)-compliant model storage, simulation and calibration comprehensive software support (within typical Python IDE), easy-to-use constructs and support within analytical workflows.
- Limitations when working with large amounts of data stored in non-volatile memory.

Running Example:



User-defined arbitrary workflow to store, calibrate and simulate physical system model

Heat pump model:

$$x(0) = x_0;$$
$$\text{For } k = 1..n:$$
$$x(k) = \left(1 - \frac{1}{R \cdot Cp}\right) \cdot x(k-1) + \left(\frac{P \cdot \eta}{Cp}\right) \cdot u(k) + \left(\frac{\theta_a}{R \cdot Cp}\right);$$
$$y(k) = P \cdot u(k);$$

A, B, C, E

$Cp = 1.5kWh/^\circ C$ is the thermal capacitance (the amount of energy needed to heat up by 1 $^\circ C$ within 1 hour);
$R = 1.5^\circ C/kW$ 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);
$\theta_a = -10^\circ C$ is the ambient temperature;
$x_0 = 21^\circ C$ is the initial temperature;
$u(1), ..., u(n)$ are *input variables* – heat pump power rating setting in the range [0 .. 1], corresponding to [0 .. 100%] of HP power operation;
$k = 1..n$; $x(0), ..., x(n)$ are *state variables* – inside temperatures at $k = 0..n$; and
$y(1), ..., y(n)$ are *output variables* – power consumed by a heat pump at the time intervals $k = 1..n$.

Linear time-invariant (LTI) state-space model of the heat pump heated room

Model ID	Measurements dataset	Inputs	Outputs	Parameters
HP0	NIST Engineering Lab's Net-Zero Energy Residential Test Facility	No inputs	– heat pump power consumption *HP_power* – indoor temperature *Indoor_Temp* (state variable)	– thermal capacitance *Cp*, – thermal resistance *R*.
HP1	NIST Engineering Lab's Net-Zero Energy Residential Test Facility	– heat pump power rating setting in the range [0 .. 100%] *HP_range*	– heat pump power consumption *HP_power* – indoor temperature *Indoor_Temp* (state variable)	– thermal capacitance *Cp*, – thermal resistance *R*.
Classroom	Data from one of the classrooms 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 coefficient *shgc*, – zone thermal mass factor *tmass*, – external wall thermal resistance *REext*, – occupant heat generation effectiveness *occheff*.

FMI physical systems model to test pgFMU upon

pgFMU system implementation, usage scenario and results:

1. Create a DBMS model instance
2. Retrieve model variables
3. Set model variable values
4. Estimate parameters of the model
5. Simulate the model

```
SELECT fmu_create('/home/fmus/model.fmu', 'hp1');
```

```
SELECT * FROM fmu_variables('hp1') AS f WHERE
f.type = 'parameter'
```

```
SELECT * FROM fmu_set('hp1', 'A', 0.56);
```

```
SELECT fmu_param_est('hp1', 'SELECT time, var_name, value
    FROM measurements', 'A, B, C, E', 'hp1')
```

```
SELECT time, var_name, sim_value, real_value
FROM fmu_simulate('hp1', SELECT time, var_name, value
FROM measurements WHERE var_name IN (''u'')') AS f
```

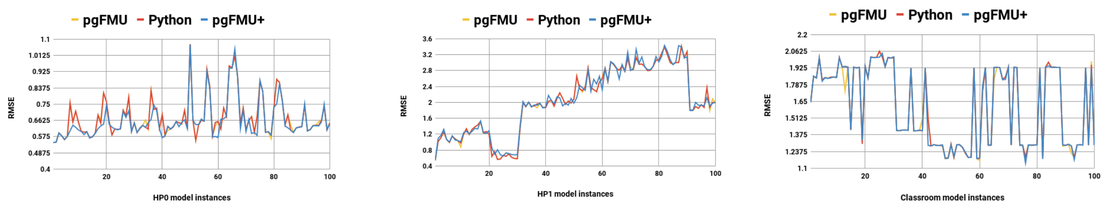
Algorithm 1: fmu_create()
Input: model_id, fmu_file
Output: fmu_model
1: Retrieve a model identifier `model_id` specified by a user;
2: Retrieve `path` to the model;
3: Retrieve model variable names, types and values by means of internal functions `get()`, `get_model_variables()` of *PyFMI* package;
4: Based on Step 3 create an instance of *fmu_model* ;
5: Store the FMU file in a volatile memory ;

Algorithm 2: fmu_simulate()
Input: model_id, SQL query
Output: ModelSimulation table.
1: Retrieve a model identifier `model_id` specified by a user;
2: Retrieve the FMU model instance based on `model_id` ;
3: Retrieve the *Measurements* table;
4: Map the input (`u`), output (`y`) and state (`x`) variables values in the *measurements* table to the input (`HP_range`), output (`HP_power`) and state (`Indoor_Temp`) variables of the FMU model instance ;
5: Perform model initialization by simulating a model from the initial time t_0 to initial time + ε (given ε is a linear interpolation between time values t_0 and t_1);
6: Perform *model_simulation* using PyFMI Python package with `simulate()` function;
7: Store simulation results in *ModelSimulation* table;

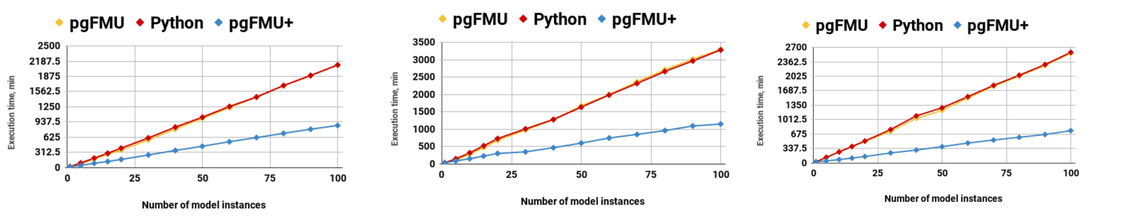
Algorithm 3: fmu_param_est()
Input: {model_id}, {inputoutput_sql}, {pars}
Output: {model_id_out}
1: Retrieve `model_id[0]` from the list of model identifiers specified by a user;
2: Based on the `inputoutput_sql[0]`, retrieve the *measurements* table;
3: Retrieve the input variables values from the *measurements* table ;
4: Retrieve the parameters to be estimated from `pars`;
5: For `model_id[0]`, run Global Search and Local Search estimation algorithms;
6: Store estimation results to *Parameters* table;
7: Update all model instances from `{model_id}` with the new parameter values;
8: For all remaining models (`model_id[1], ... model_id[n]`, n = length(`{model_id}`)) run only Local Search algorithm, using estimation results from Step 6 as initial guess;
9: Store estimation results for `model_id[1], ... model_id[n]` to *Parameters* table;
10: Store the new model instances as `{model_id_out}`

Pseudocode for pgFMU User-Defined Functions (UDFs):
fmu_create ()(Algorithm 1),
fmu_simulate () (Algorithm 2),
fmu_param_est () (Algorithm 3)

PgFMU + - an optimization technique for estimating parameters of multiple model instances. The search space is reduced by using "initial guess" technique. Main idea: after running Global Search + Local Search algorithms for one model instance, reuse the results for the remaining model instances, assuming the model instances are of the same structure and input time series have the same distribution.



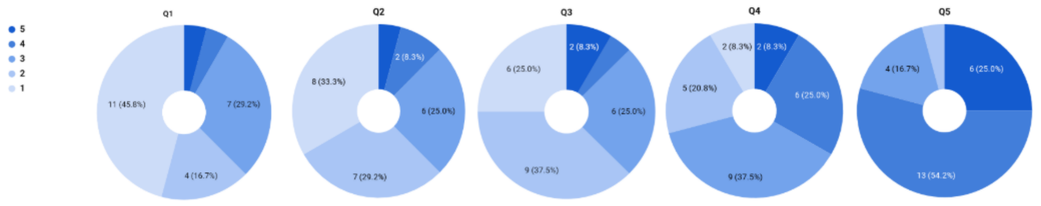
RMSE comparison for 100 model instances of *HP0*, *HP1* and *Classroom*. RMSE for all three models is almost identical meaning pgFMU and pgFMU+ delivers as accurate results as typical setup (Python)

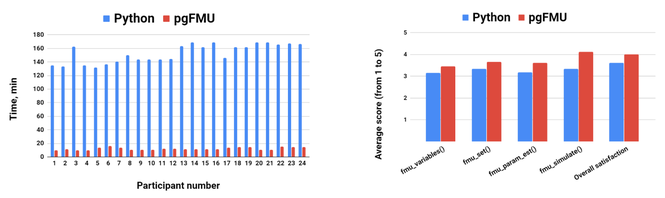


Step 1-5 execution time comparison for 100 model instances of *HP0*, *HP1*, *Classroom*. For *HP0*, pgFMU+ is 2.43x faster, for *HP1* – 2.85x faster, for *Classroom* – 3.4x faster

Usability evaluation :

Pre-assessment questionnaire (1 - very little, 5 - very much):
Q1. How can you estimate your knowledge in energy systems and physical systems modeling?
Q2. How familiar are you with model simulation and model calibration process?
Q3. How familiar are you with model simulation software(s)?
Q4. How comfortable are you with using Python IDE?
Q5. How comfortable are you with using SQL?





Steps 1-5 student workflow time comparison for *HP1* and *Classroom* (left) and average user satisfaction with setups functionality comparison (right)

Operation	Python	pgFMU(+)
1. Create a DBMS model instance	4	1
2. Retrieve model variables	12	1
3. Set model variable values	4	1
4. Estimate parameters of the model	15	1
5. Simulate the model	24	1
Total	59	5

Lines of code comparison within typical setup (Python) and pgFMU(+)

Conclusions:

- pgFMU - the first DBMS extension to support simulation, calibration and validation of Functional Mockup Interface (FMI) – compliant physical systems dynamic models within a single DBMS environment.
- On average 2.9 times faster in comparison to the traditional workflow, and 11.8 times less code lines.
- pgFMU is up to 12.5x faster in terms of development time for the arbitrary user-defined workflow.

References

1. Gartner. https://www.gartner.com/technology/home.jsp Accessed: 2017-12-05.
2. Šikšnys, L., & Pedersen, T. B. (2016, July). Solvedb: Integrating optimization problem solvers into sql databases. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management* (p. 14). ACM.
3. Frazzetto, D., Nielsen, T. D., Pedersen, T. B., & Šikšnys, L. (2019). Prescriptive analytics: a survey of emerging trends and technologies. *The VLDB Journal*, 1-21.

This Ph.D. is jointly funded by IT4BI-DC and AAU, Denmark