ModestPy: An Open-Source Python Tool for Parameter Estimation in Functional Mock-up Units

Krzysztof Arendt[1] Muhyiddine Jradi[1] Michael Wetter[2] Christian T. Veje[1]

[1]Center for Energy Informatics, University of Southern Denmark, Denmark, {krza,mjr,veje}@mmmi.sdu.dk
[2]Lawrence Berkeley National Laboratory, USA, mwetter@lbl.gov

Abstract

The paper presents an open-source Python tool for parameter estimation in FMI-compliant models, called *ModestPy*. The tool enables estimation of model parameters using user-defined sequences of methods, which are particularly helpful in non-convex problems. A user can start estimation with a chosen global search method and subsequently refine the estimates with a local search method. Several methods are available already and the tool's architecture allows for easily adding new ones. The advantages of having a single interface to multiple methods and using them in sequences are highlighted on a case study in which the parameters of a Modelica-based gray-box model of a building zone (nonlinear, multi-output) are estimated using 9 different combinations of methods. The methods are compared in terms of accuracy and computational performance.

Keywords: FMI, parameter estimation, Python, open-source

1 Introduction

1.1 Background

The Functional Mock-up Interface (FMI) is becoming a *de facto* standard co-simulation interface, as of 2018 being supported by over 100 simulation tools (`http://fmi-standard.org/tools/`). The support of the FMI in simulation tools varies from full support, especially in Modelica tools, to a subset of FMI 1.0 / 2.0, import / export, co-simulation / model exchange. Models compliant with FMI are referred as Functional Mock-up Units (FMUs).

Using FMUs allows for a flexible co-simulation between models developed in different software environments. It is also attractive for resource-limited embedded systems with no need to install a GUI-based simulation environment, or because relying on FMI makes the system less dependent on specific tools and vendors. In addition, the common interface to models developed in different software environments makes it possible to develop generic tools for co-simulation, system identification, or optimization. System identification methods are commonly used to calibrate models with respect to the real system. In the case of FMUs, the model structure is typically already defined in the FMU, so the problem is narrowed down to the estimation of model parameters and/or states. In fact, several FMI-specific tools for parameter and state estimation have been developed in recent years.

Bonilla et al. (2017) developed a GUI tool for static (batch) parameter estimation in FMUs (compliant with Model Exchange 2.0), based on a global-search Multi-Objective Genetic Algorithm (MOGA). The main motivation for the development of this tool was to facilitate the coupling of different modeling languages, tools and optimization algorithms, while being customizable. The authors stated that the implementation of additional optimization algorithms is planned.

Bonvini et al. (2014) developed a state and parameter estimation Python tool based on Unscented Kalman Filter (UKF), compliant with Model Exchange 1.0. The UKF is a recursive estimation method, meaning it is suitable for on-line applications where the states or parameters need to be continuously updated. The authors presented the capabilities of the tool on a fault detection and diagnosis (FDD) case study, in which it was used to identify a faulty valve in a simple theoretical system. In another exemplary application, the tool has been used for an on-line estimation of the number of occupants in a building based on indoor temperature and CO_2 (Sangogboye et al., 2017).

Vanfretti et al. (2016) developed the Rapid Parameter Identification toolbox (RaPId), used for parameter estimation in FMUs. The tool is developed as a Matlab/Simulink plug-in and can be called using the Matlab command line interface, but is also equipped with a GUI allowing it to be used as a standalone application. Multiple optimization methods are available, such as `fmincon` from Matlab and heuristic algorithms, like a particle swarm optimization (PSO) implemented by the authors.

Kampfmann et al. (2017) proposed a work flow for parameter estimation in FMUs, based on open-source tools. The performance of the work flow was demonstrated on a real industrial problem of a three arm Delta Robot. The estimation problem is formulated using the maximum likehood approach and the underlying optimization problem is solved using the Levenberg-Marquardt algorithm.

De Coninck et al. (2016) developed a more specialized Python toolbox for gray-box system identification for buildings. The tool automates both the model selection (out of a group of predefined models) and parameter estimation. The toolbox relies on the JModelica.org platform which is used for compilation of models, simulation and optimization. The direct collocation method is used to solve the underlying optimization problem. Since the collocation method is a local optimization method, the initial guesses of parameters are either inherited from the previous estimation runs or a set of initial guesses is constructed using the Latin hypercube sampling method. The toolbox is currently in use in MPC systems in existing buildings (De Coninck and Helsen, 2016).

Andersson et al. (2012) implemented in *JModelica.org* three derivative-free optimization algorithms suitable for parameter estimation in FMUs. The algorithms include the Nelder-Mean simplex method, a differential evolution algorithm, and a genetic algorithm. The algorithms can be accessed through a Python interface.

Despite the diverse landscape of the tools and work flows presented, there are still some niche applications not specifically addressed. For example, there are no lightweight, generic, open-source tools that would enable automated testing of multiple algorithms, including algorithm sequences (for global and local search), and be easily deployable on the target machines and integrable into other codes. The available tools are either tied to specific optimization algorithms, specific proprietary platforms, or large software environments. This paper presents a new tool that potentially fills this niche.

1.2 Paper Objective

The objective of this paper is to present a new open-source Python tool for parameter estimation in FMUs, called *ModestPy* (Arendt, 2017). Through the PyFMI (Andersson et al., 2016) the tool is compatible with FMI 1.0/2.0 and Co-Simulation/Model Exchange. The main novelty of the tool compared to the ones presented is that it includes several optimization methods and enables easily running the methods in user-defined sequences.

The initial motivation of this work was to facilitate parameter estimation in models of buildings and HVAC systems for the purpose of MPC. One of the often mentioned requirements for MPC to become economically viable in buildings is an automated creation and updating (e.g. through parameter estimation) of predictive models (Rockett and Hathway, 2017). Building and HVAC models are often non-linear and in general can be non-differentiable. In this context, *ModestPy* is a generic tool with no assumptions on the model structure, offering a high flexibility in terms of combining different algorithms. Since the performance of specific methods is model-dependent, the aim of the tool is to support multiple algorithms. Currently, the tool supports two algorithms implemented by the authors (genetic algorithm and pattern search) in addition to several algorithms from the *SciPy* eco-system (Jones et al., 2001).

2 Software Description

ModestPy is designed with the ease of use and installation in mind. It is compatible with both Python 2.7 and 3 and was tested on Windows and Linux. The package can be installed using the command `pip install modestpy`.

The package structure is modular (Fig. 1). A user needs to interact only with one class, `Estimation` from `estimation.py`, and its two methods: `estimate()` and `validate()`.

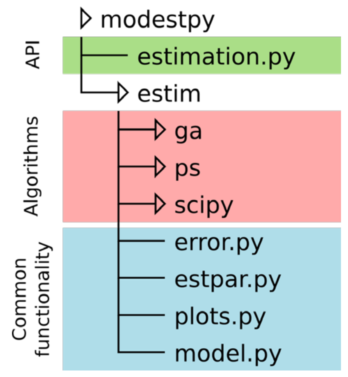

Figure 1. Package structure (testing, logging and auxiliary modules excluded for clarity).

The implemented algorithms are kept in separate modules under `modestpy.estim`. The algorithms share some common functionality covering the error calculation (`error.py`), estimation parameter interface (`estpar.py`), plotting (`plots.py`), and FMU model interface (`model.py`). The interaction with FMUs is provided by *PyFMI* (Andersson et al., 2016). Two types of error metrics are implemented, the total mean-square error (MSE_{tot}) and the total normalized mean-square error ($NMSE_{tot}$), calculated as follows:

$$MSE_{tot} = \sum_i \frac{\sum_{t=1}^{N} (\hat{Y}_i^t - Y_i^t)^2}{N}, \quad (1)$$

$$NMSE_{tot} = \sum_i \frac{MSE_i}{\bar{Y}_i^2}, \quad (2)$$

where $\hat{Y}_i^t$ is the measured value of variable i at time step t, Y_i^t is the simulated value of variable i at time step t, $\bar{Y}_i$ is the mean measured value of variable i, N is the number of time steps, and MSE_i is the mean-square error for variable i. Eq. (1) is suggested for single-output models ($i = 1$) or models with physically comparable outputs. Eq. (2) is suggested for multi-output models ($i > 1$) with physically incomparable outputs, e.g. temperature and CO_2. It should be noted that $\bar{Y}_i$ in Eq. (2) needs to be non-zero. Other norms can be easily implemented in `error.py` if needed.

2.1 Algorithms

A user can estimate parameters using a single algorithm or an arbitrarily designed sequence of algorithms. The sequences typically contain two methods, the first for global search and the second for local search, but if needed it is possible to queue more methods, e.g. multiple genetic algorithms with different evolution parameters (best solution from each run is saved and propagated to the next one). It is, therefore, possible to quickly test and evaluate different methods and combination of methods on a given problem.

Currently two in-house algorithms are implemented, a genetic algorithm (GA) and a generalized pattern search (GPS), in addition to the interface to several well-known algorithms from the *SciPy* eco-system (Jones et al., 2001), specifically those compatible with the function `scipy.optimize.minimize()`. The *SciPy*'s algorithms tested by the authors with results presented in this paper include the sequential least squares programming (SLSQP) (Kraft, 1988), the limited memory Broyden–Fletcher–Goldfarb–Shanno with box constraints (L-BFGS-B) (Byrd et al., 1995), and the truncated Newton method (TNC) (Nash, 1984).

The implemented GA (Algorithm 1) is based on the standard operations of tournament-based selection and crossover. The mutation is adaptive and depends on the population diversity, defined as the ratio of shared genes among the individuals. If the population is diverse, a standard mutation is applied in which a low mutation rate (10% by default) is used, but genes can mutate within a wide range (entire parameter range by default). If the population diversity is low, the population is split into 1/3 and 2/3 parts. The larger group (2/3) undergoes the standard mutation (low mutation rate, wide range of possible values), and the smaller group undergoes mutation with an increased rate (33% by default), but the genes are allowed to change only within a small range (equivalent to a stochastic local search). By tuning $mutrate_1$, $mutrate_2$, $range_1$, and $range_2$ in Algorithm 1, a desired balance between the exploratory and local search of the evolution can be found. The algorithm continues until the error stops decreasing or the maximum number of generations is reached. The tolerance criterion checking can be delayed by a user-defined number of generations (10 by default), allowing for continuing the evolution for some time even if the subsequent generations do not improve the solution. GA is a metaheuristic global search algorithm, which is often able to deal with complex non-convex functions, but does not guarantee that the global optimum is found. In fact, the solution might be worse than in the case of local search methods. It is, therefore, often coupled with some local search methods, as GPS, least squares or Newton-based methods.

The implemented GPS (Algorithm 2) is a simple gradient-free local search method. The algorithm requires an initial guess $\mathbf{x}_0$ and an initial step size s. Thereafter it starts a series of orthogonal exploratory moves, by changing one parameter at a time (in both positive and negative directions) and evaluating the cost function $f(\mathbf{x})$. The parameter vector propagated to the next iteration is the one with the lowest cost function value. If the given step size does not yield a reduction in the cost function value, it is reduced by a factor of 2 and the procedure is repeated. The algorithm stops when the solution does not improve despite k_{max} reductions of the step size. GPS usually requires more iterations than gradient-based methods, but the method can deal with models that are not differentiable or continuous. A similar algorithm is implemented in the generic optimization program GenOpt (Wetter, 2001).

Algorithm 1 Genetic algorithm implemented in *ModestPy*

```
Initialize population $pop_1$ with $N$ individuals
while generation $g < g_{max}$ and error decreasing do
    for all individuals in $pop_1$ do
        Evaluate cost function
    end for
    Initialize empty population $pop_2$
    Add fittest individual from $pop_1$ to $pop_2$ {elitism}
    for $i = 0$ to $N-1$ do
        $ind1 \leftarrow$ tournament($pop_1$)
        $ind2 \leftarrow$ tournament($pop_1$)
        $child \leftarrow$ crossover($ind1, ind2$)
        Add $child$ to $pop_2$
    end for
    if $pop_2$ is diverse then
        $pop_2 \leftarrow$ mutate($pop_2$, $mutrate_1$, $range_1$)
    else
        $X\%$ of $pop_2 \leftarrow$ mutate($mutrate_1$, $range_1$)
        $mutrate_2 \leftarrow mutrate_1 K$ {$K > 1$}
        $range_2 \leftarrow range_1 L$ {$1 > L > 0$}
        $(100-X)\%$ of $pop_2 \leftarrow$ mutate($mutrate_2$, $range_2$)
    end if
    $pop_1 \leftarrow pop_2$
    $g \leftarrow g+1$
    $\mathbf{x} \leftarrow$ parameters of fittest individual
end while
return $\mathbf{x}$
```

2.2 Usage

ModestPy does not have a GUI (although there are plans to develop one in the future) and is aimed to be used directly in Python scripts. First, the user has to instantiate the `Estimation` class. All the estimation and validation settings are specified during the instantiation. The required arguments are as follows: path to the working directory (string), path to the FMU (string), data frame with the inputs (pandas DataFrame), known inputs (dictionary), parameters to be estimated (dictionary), and measured data (pandas DataFrame). The user can control other aspects of the estimation with the optional arguments, as discussed in the documentation (Arendt, 2017). Secondly, the estimation and validation methods are called to retrieve the results.

The exemplary Python code setting up an estimation using GA+GPS and using the MSE cost function is as follows:

```
from modestpy import Estimation

session = Estimation(
    workdir,  # string
    fmu_path, # string
```

Algorithm 2 Generalized pattern search algorithm implemented in *ModestPy*

```
Require: Initial guess vector $\mathbf{x}_0$ with $N$ parameters, initial
  step $s_0$, max. no. of tries $k_{max}$ to decrease step
  $s \leftarrow s_0$
  $k \leftarrow 0$
  $\mathbf{x} \leftarrow \mathbf{x}_0$
  $y \leftarrow f(\mathbf{x})$
  while $k < k_{max}$ do
    $y_n \leftarrow y$
    for $n = 0$ to $N$ do
      $\hat{\mathbf{x}}_n \leftarrow$ $N$-dim. versor with $n$-th element equal to 1
      $\mathbf{s}_n \leftarrow s\hat{\mathbf{x}}_n$
      $y_n^+ \leftarrow f(\mathbf{x}+\mathbf{s}_n)$
      $y_n^- \leftarrow f(\mathbf{x}-\mathbf{s}_n)$
      if $y_n^+ < y_n$ then
        $y_n \leftarrow y_n^+$
        $\mathbf{x}_n \leftarrow \mathbf{x}_n + \mathbf{s}_n$
      end if
      if $y_n^- < y_n$ then
        $y_n \leftarrow y_n^-$
        $\mathbf{x}_n \leftarrow \mathbf{x}_n - \mathbf{s}_n$
      end if
    end for
    if $y_n < y$ then
      $y \leftarrow y_n$
      $\mathbf{x} \leftarrow \mathbf{x}_n$
      $k \leftarrow 0$
    else
      $s \leftarrow s/2$ {reduce step}
      $k \leftarrow k+1$
    end if
  end while
  return x
```

```
    inputs,    # pandas DataFrame
    known,     # dictionary
    estimate,  # dictionary
    measured,  # pandas DataFrame
    method=('GA', 'GPS'),
    ga_opts={'maxiter': 5, 'tol': 1e-4},
    gps_opts={'maxiter': 500, 'tol': 1e-6},
    ftype='MSE'
)

estimates = session.estimate()
err, res = session.validate()
```

In addition to the estimation and validation results returned by the respective methods, the results are saved in the working directory together with plots of error and parameter trajectories.

3 Example

3.1 System

The functionality of the tool is presented on a case study in which it was used to calibrate a Modelica-based gray-box model of a single building zone to the results of a white-box model developed in EnergyPlus. The case study is a part of a larger project aimed at the development of an MPC framework that will be tested on the OU44 building in Odense, Denmark (Jradi et al., 2017). The developed EnergyPlus model is a downscaled 7-zone version of the OU44 building, but with the same HVAC system and internal heat gains schedules. The EnergyPlus model is used for MPC tests in the project.

The building is equipped with a balanced mechanical ventilation system. The air handling unit (AHU) contains two fans (for supply and exhaust air), a rotary wheel heat recovery system, and a heating coil. No cooling is provided. The fans are controlled to maintain a constant pressure in the duct system. Each zone is equipped with a hydronic radiator and a VAV box. The radiator valve and ventilation damper positions depend on the indoor temperature and CO_2 concentration, respectively. The temperature and CO_2 setpoint schedules are set through the BMS system.

Selected outputs of the white-box model are assumed to represent the measured data for the calibration of the gray-box model. The following zone-level outputs are used: indoor temperature `T` [°C], indoor CO_2 concentration `CO2` [ppm], radiator heat supply `qrad` [W], ventilation airflow rate `verate` [m^3/s], and the number of occupants `occ` [$-$] (notation consistent with the gray-box model variables). The chosen outputs correspond with the available sensors in the OU44 building, so in the later phase of the project the same gray-box model can be used with the real data. In this study, however, the white-box results are used to exclude the effect of uncertain inputs on the results. In addition to the zone-level measurements, the weather data and assumed temperature and CO_2 setpoints are passed to the gray-box model as inputs.

3.2 Modelica Gray-box Model

The gray-box model was developed in Modelica and exported to an FMU using Dymola (Fig. 2). The thermal part of the model is based on the RC network approach and contains two capacitors, one for the indoor air and one for the internal thermal mass, and one resistor representing the external walls. The solar gains and radiator heat gains are modeled with single gain blocks. The radiator heat supply is controlled by a PI controller, depending on the temperature setpoint (input) and calculated indoor temperature. The occupancy contributes to both the indoor heat gains and CO_2 generation. The metabolic heat generation per person is modeled as a linear function of the indoor temperature. The CO_2 balance is modeled with a custom block containing a steady state mass balance equation. The ventilation heat gain is also modeled with a custom block containing a steady state heat balance equation. The rest of the model is based on the components from the Modelica Standard Library.

The model takes six inputs and has six outputs. The inputs include the solar radiation `solrad` [Wm^{-2}], outdoor temperature `Tout` [°C], number of occupants `occ`, ventilation air temperature setpoint `Tvestp` [°C], indoor CO_2 setpoint `CO2stp` [ppm], and indoor temperature setpoint `Tstp` [°C]. The outputs are the indoor temperature `T` [°C], indoor CO_2 `CO2` [ppm], ventilation airflow rate `verate` [m^3s^{-1}], total ventilation airflow `vetot` [m^3], radiator heating rate `qrad` [W], and total radiator heating energy `Qrad` [J].

Seven of the model parameters are estimated: outdoor wall resistance `RExt` [m^2WK^{-1}], indoor wall resistance `RInt` [m^2WK^{-1}], infiltration air change rate `Vinf` [h^{-1}], maximum ventilation air change rate `maxVent` [h^{-1}], internal thermal mass `imass` [$JK^{-1}m^{-2}$], solar heat gain coefficient `shgc` [$-$], and indoor air thermal mass `tmass` [$JK^{-1}m^{-3}$]. All parameters excepts `shgc` are scaled by the respective surface areas (e.g. external wall surface area for `RExt`) or the indoor volume (`tmass`). It should be noted that the infiltration rate parameter `Vinf` affects only the CO_2 balance, while the thermal effect of infiltration is included in the resistance `RExt`.

3.3 Estimation Setup

Table 1 provides the lower and upper bounds in addition to the initial guesses of the seven parameters to be estimated in this study. The bounds were chosen considering standard physical and technical specifications of buildings.

Table 1. Initial guess, lower and upper bounds of the gray-box model parameters.

Parameter	Initial guess	Low. bound	Up. bound
`shgc`	5	0.1	10
`tmass`	50	1	100
`imass`	50	1	100
`RExt`	5	0.1	10
`RInt`	5	0.1	10
`Vinf`	5	0.1	10
`maxVent`	5	0.1	10

The cost function is based on the total normalized mean-square error $NMSE_{tot}$, calculated according to Eq. (2). Four white-box model outputs are taken into account: `T`, `CO2`, `verate`, and `qrad`.

9 different method sequences were used to estimate the parameters: (1) GA, (2) GPS, (3) TNC, (4) SLSQP, (5) L-BFGS-B, (6) GA+GPS, (7) GA+TNC, (8) GA+SLSQP, (9) GA+L-BFGS-B.

The GA settings were consistent across the sequences (1) and (6)-(9). In addition a random number seed was used to make the results comparable. The maximum number of generations was set to 100. The initial population contained 40 individuals and was generated using the Latin hypercube sampling. The maximum number of iterations in all the other methods was 500. The absolute solution tolerance was 1e-9 (same for all methods).

All the non-default estimation parameters are specified in the following code:

```
ga_opts = {
    'maxiter': 100, 'tol': 1e-9,
    'lhs': True, 'pop_size': 40
}
gps_opts = {
    'maxiter': 500, 'tol': 1e-9
}
scipy_opts = {
    'solver': 'scipy_solver',
    'options': {
        'maxiter': 500, 'tol': 1e-9
    }
}

session = Estimation(
    wdir, fmu, inp, known, est, out,
    lp_frame=(0, 86400 * 3),
    vp=(86400 * 3, 86400 * 4),
    ga_opts=ga_opts,
    gps_opts=gps_opts,
    scipy_opts=scipy_opts,
    methods=met_seq,
    ic_param=ic_param,
    ftype='NMSE', seed=1
)
```

where `scipy_solver` is one from `['TNC', 'L-BFGS-B', 'SLSQP']`, and `met_seq` is one from `[('GA', ), ('GPS', ), ('SCIPY', ), ('GA', 'GPS'), ('GA', 'SCIPY')]`.

The training period was 3 days long (January 1-3). The validation was performed on the day following the training period (January 4).

3.4 Results

All the 9 considered method sequences yielded different estimates, despite the same bounds and initial guesses (in GPS, SLSQP, L-BFGS-B, TNC) and the random number seed (in the GA-based sequences). Fig. 3 presents a stacked histogram of the obtained estimates. The estimates obtained by the GA-based sequences are close to the ones yielded by the GA alone. Differences are noticeable mainly in the case of `shgc` and `maxVent`. The remaining method sequences (non-GA-based) yielded estimates scattered across the parameter space.

The GA+L-BFGS-B and GA+GPS sequences yielded the lowest training errors, while GA+SLSQP was the most accurate in the validation, suggesting that the parameters are slightly overfitted in former cases (Table 2). GA+L-BFGS-B was the only GA-based sequence that performed worse in the validation than the GA alone. Nevertheless, all GA-based sequences yielded models with a similar accuracy, with training errors below 0.394 and validation errors below 0.379. The rest of the methods (GPS, TNC, L-

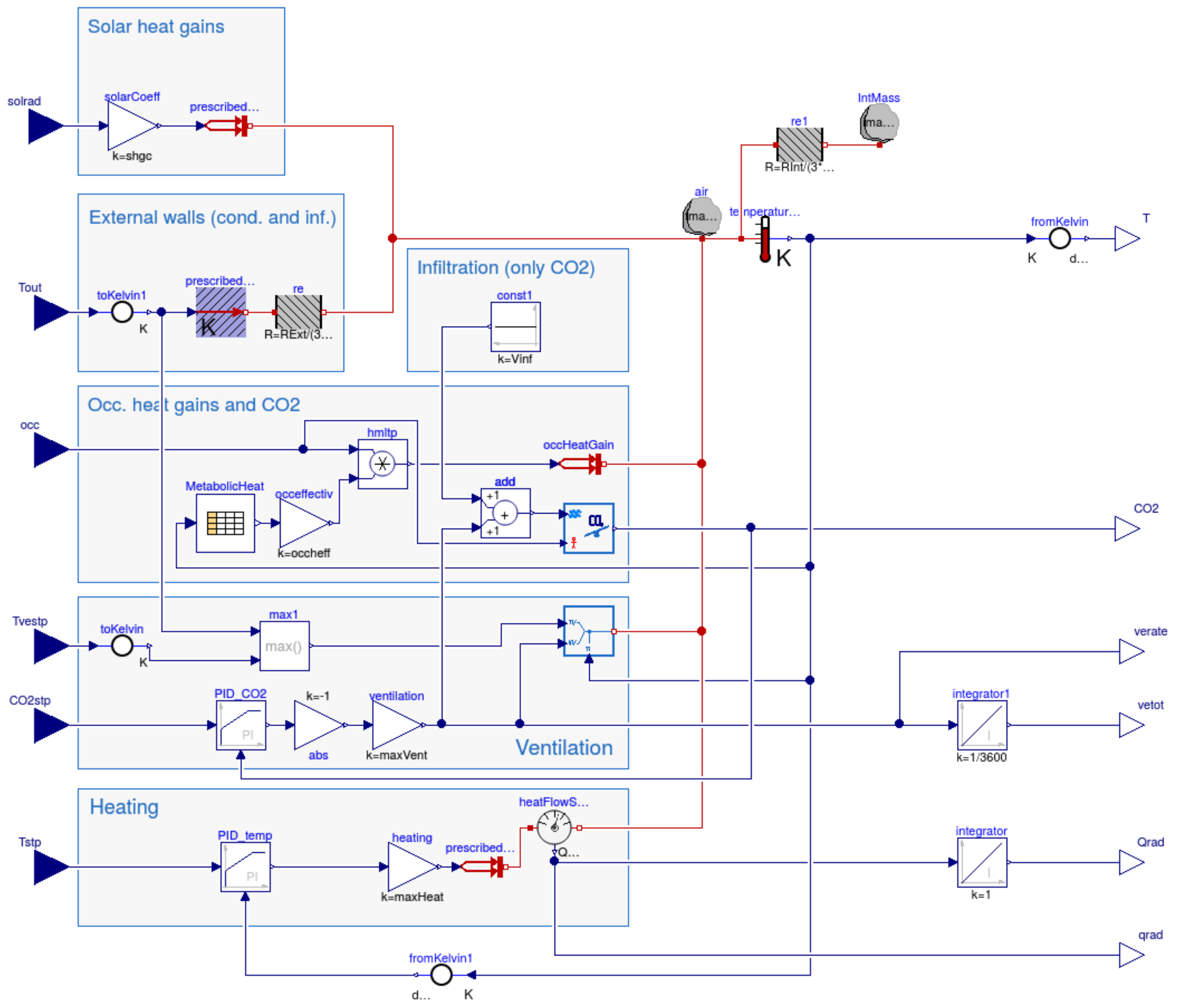

Figure 2. Gray-box zone model developed in Modelica (using Dymola).

BFGS-B, SLSQP) yielded significantly worse estimates, with validation errors above 3.4.

The computational time of the GA-based sequences was mostly dominated by GA (723 s). In the considered problem only GPS was slower than GA (986 s). All gradient-based methods included in the test where much faster, but stuck in local minima. From the time-to-accuracy point of view mixing GA with a gradient-based method seems the most efficient (801-934 s), while GA+GPS is the slowest combination (1319 s).

All computations were performed on a laptop equipped with an Intel Core i7-5600U processor (2.60GHz), 16 GB RAM, and a hard-disk drive (HDD). The disk type is relevant, because *ModestPy* was run in a verbose logging mode, each time saving a log file containing up to 85000 lines with detailed results from each iteration. All simulations were run on a single core – *ModestPy* does not offer parallelized algorithms yet.

Table 2. CPU time and total normalized mean square error ($NMSE_{tot}$) for validation and training, sorted in ascending order by validation $NMSE_{tot}$.

Method	Training $NMSE_{tot}$	Validation $NMSE_{tot}$	CPU Time [s]
GA+SLSQP	0.377	0.353	920
GA+GPS	0.351	0.371	1319
GA+TNC	0.393	0.372	801
GA	0.394	0.373	723
GA+L-BFGS-B	0.349	0.379	934
GPS	1.306	3.428	986
TNC	4.967	5.856	101
L-BFGS-B	4.929	6.808	38
SLSQP	5.040	6.920	12

The different estimates yielded by the respective methods can be due to inaccuracies in the numerically approximated gradients, and due to the non-convexity of the cost function. The non-convexity of the cost function used in this study can be analyzed in Fig. 4. The cost function evaluated on the line connecting estimates yielded by two different methods (GA and SLSQP) has few sections with positive derivatives and local minima. In addition the cost function is very steep in the vicinity of the GA solution, while relatively flat for $s < 0.7$. Although the shape of the function in other directions is not presented, Fig. 4 highlights the need for a good initial guess in the case of gradient-based methods if the cost function is non-convex.

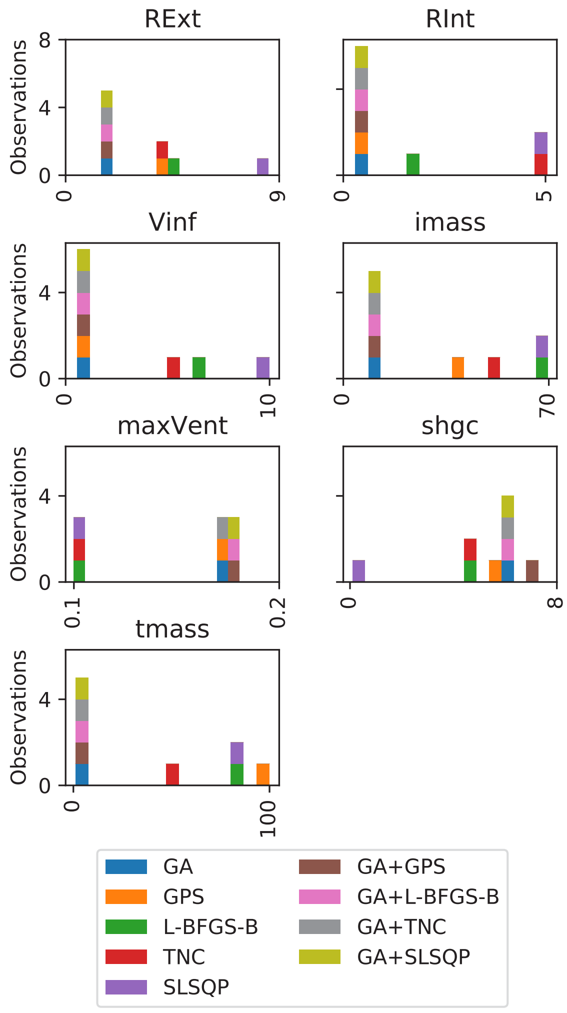

Figure 3. Histogram of estimates yielded by the 9 method sequences.

In the case of non-convex problems, it is sometimes useful to analyze the visualization of the parameter search in GA (Fig. 5). This figure is by default generated by *ModestPy* whenever GA is used in the method sequence. Each dot in the figure represents a solution produced by a single individual. The colors represent the error ($NMSE_{tot}$), with the brightness decreasing with decreasing error. The GA starts with the inaccurate population spread over the entire

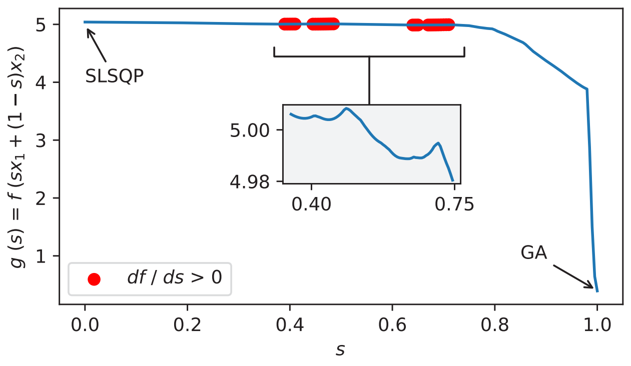

Figure 4. Cost function evaluated on the training data based on linear combinations of parameters yielded by GA ($\mathbf{x}_1$) and SLSQP ($\mathbf{x}_2$). Sections with positive derivatives with respect to s marked in red.

search domain (yellow color marks $NMSE_{tot}$ above 5.0). Over time, the solution quality improves to $NMSE_{tot}$ below 1.0 (purple color). For some parameters (most notably `maxVent` and `Vinf`) the value found early in the evolution does not change much throughout the rest of the evolution. In other cases rapid jumps in parameters with only a minor improvement in the accuracy are observed (e.g. `imass`). Based on this visualization, the user is able to assess whether a desired balance of the exploration and local search was achieved.

Fig. 6 depicts root-mean-square errors (RMSE) for each output variable, calculated for the validation period. RMSE was not used in the cost function, because it is non-differentiable at 0 which is problematic for gradient-based methods, but the metric helps interpreting results. RMSE represents the standard deviation between the predicted and true values. The analysis of the errors reveals that the problems encountered by the non-GA-based sequences were due to both the thermal and CO_2 parts of the model. The best performing models achieved RMSE below 50 ppm for `CO2`, 0.4 °C for `T`, 600 W for `qrad`, and 10 m^3h^-1 for `verate`. The RMSE differences between the best and worst performers are around 180 ppm for `CO2`, 2 °C for `T`, 1600 W for `qrad`, and 20 m^3h^{-1} for `verate`. Interestingly, unlike the gradient-based methods, GPS calibrated well the parameters affecting `CO2` and `verate`, even though it started from the same initial guess.

The validation results for the four output variables explain the large errors yielded by the non-GA-based sequences. The temperature in those cases slowly reacts to the applied heat loads (Fig. 7), which is due to the overestimated thermal mass (`tmass`, `imass`) of the building (Fig. 3).

In the case of CO_2 (`CO2`) and ventilation rate (`verate`) the inaccuracy of the gradient-based methods is due to the overestimated infiltration rate `Vinf` (Fig. 3), which reduced the indoor CO_2, preventing it from reaching the setpoint of 800 ppm and triggering the ventilation. Hence, the ventilation in those cases remained turned off through-

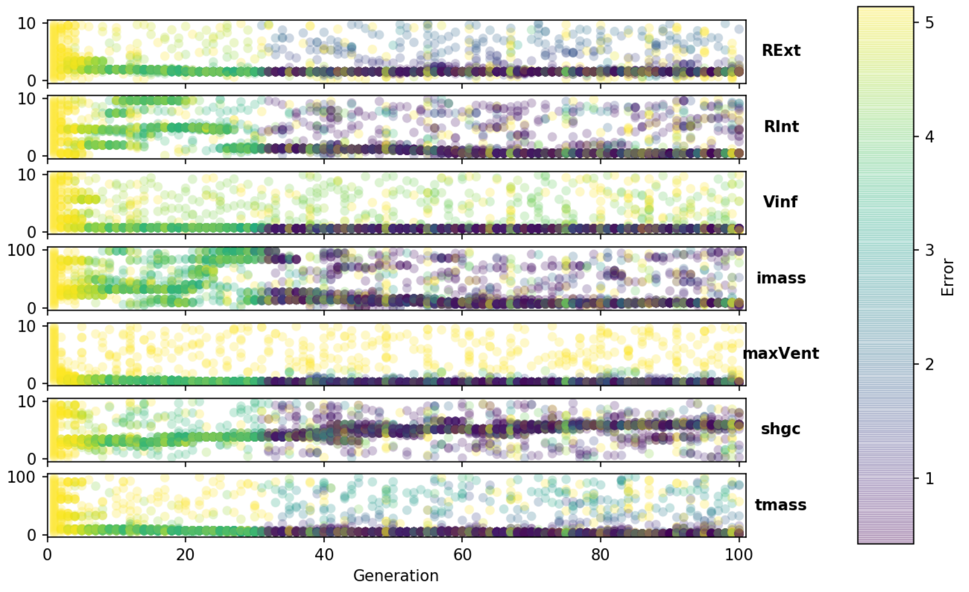

Figure 5. Parameter evolution in the genetic algorithm – color represents the training error (darker more accurate).

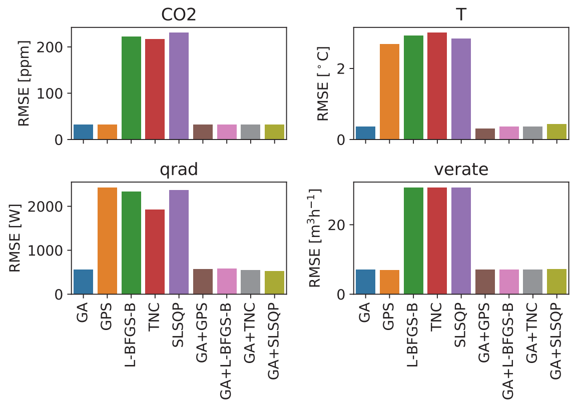

Figure 6. Validation root-mean-square error (RMSE) per output variable.

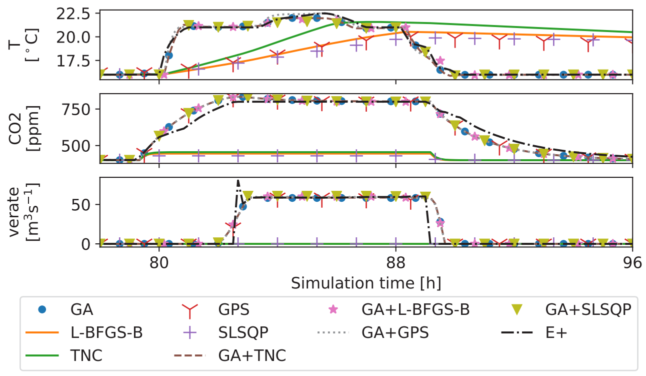

Figure 7. Validation results: temperature (`T`), CO_2 (`CO2`), ventilation airflow rate (`verate`).

out the entire validation period (`verate` equal to 0).

Similarly, the models with the inaccurate indoor temperature yielded inaccurate radiator heating power `qrad` (Fig. 8). Only the GA-based method sequences were able to achieve a good accuracy with respect to `qrad`. In all cases, however, there is a mismatch in the initial 8 hours of the validation period due to the wrong initial condition for `qrad`. If the grey-box model is to be used for short-term predictions, the initial value should be based on the measurements, as in the case of indoor temperature and CO_2.

4 Discussion

The results are not generalizable to other models, estimation settings (initial guesses, number of iterations, parameter bounds etc.), or even training data. The specific methods can perform differently on different problems. The gradient-based methods heavily rely on the quality of the initial guess, while the GA results depend on the random seed. Both, the initial guess and the random seed were fixed in this study. However, the results highlight the importance of such inter-method comparisons, especially in the cases where the parameter estimation is a non-convex problem. In many practical applications it may be difficult to assess whether or not the problem is non-convex, or what should be the initial guess for parameters.

The GA results presented in this paper are surprisingly good in terms of the accuracy and computational time. Typically GA requires much more time to converge than gradient-based methods. However, since GA is stochastic in nature, the results could be different if the experiment was repeated without a fixed random seed. The computa-

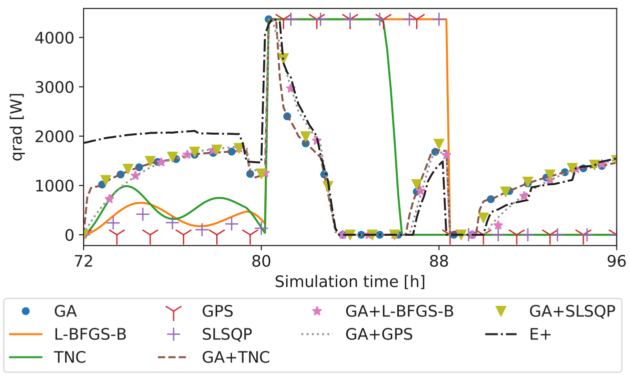

Figure 8. Validation results: radiator heating rate (`qrad`).

tional times reported in this study are based on single trials and therefore may be slightly biased.

The authors advise to always couple GA with another gradient-based method and rerun the estimation several times if there are no strict time constraints, possibly with various GA settings. *ModestPy* allows for setting up the number of estimation runs with an optionally moving training period during the instantiation of the `Estimation` class. An alternative approach could be to test the gradient-based methods with multiple random initial guesses.

It should be noted, that the gradient-based methods (SLSQP, L-BFGS-B, TNC) would be much faster if the gradient of the cost function with respect to the estimated parameters was known. In a general case, this gradient is not known and has to be evaluated numerically, as in this study. However, the FMI standard allows to provide directional derivatives (optional feature), and some tools support it. If the computational time is an issue, other tools that are able to perform algebraic differentiation should be used, e.g. JModelica.org (Åkesson et al., 2009).

5 Conclusions

Automated parameter estimation is crucial in many industrial applications, including MPC and other cyber-physical systems. The FMI standard provides an attractive simulation interface that allows for using models outside their dedicated environments, and developing model-agnostic tools. This paper introduces a new Python tool for parameter estimation in FMI-compliant models, called *ModestPy*. The tool supports several optimization methods, both gradient-free and gradient-based (numerically approximated), that can be queued in user-defined sequences. *ModestPy* enables easily testing multiple methods on a given model and find the most suitable approach and estimation settings.

The tool was tested on a case study in which it was used to estimate parameters of a non-linear multi-output model of a building zone. The results showed that the local optimization methods (GPS, L-BFGS-B, TNC, SLSQP) were unable to calibrate the model, marking that the the initial guess for parameters was poor. The addition of GA as the initial global search method significantly improved the model accuracy. The results also validated the two in-house algorithms (GA and GPS).

It should be noted, 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.

The current functionality of the tool is already sufficient for a general use. It is used by the authors for calibrating gray-box models of buildings and HVAC systems for the use in MPC.

However, the development work continues and there are plans to include the following functionality:

- a simple graphical user interface to attract users less experienced in the Python programming language,
- support for on-line estimation methods (e.g. Kalman filter),
- support for multi-period stochastic gradient descent training,
- support for parallel processing methods.

6 Acknowledgments

The authors thank David Blum for comments that helped to improve the manuscript and the development of *ModestPy*.

This work was supported by the Innovation Fund Denmark for the project COORDICY (4106-00003B).

This research was supported by the Assistant Secretary for Energy Efficiency and Renewable Energy, Office of Building Technologies of the U.S. Department of Energy, under Contract No. DE-AC02-05CH11231.

This work emerged from the IBPSA Project 1, an international project conducted under the umbrella of the International Building Performance Simulation Association (IBPSA). Project 1 will develop and demonstrate a BIM/GIS and Modelica Framework for building and community energy system design and operation.

References

Johan Åkesson, Magnus Gäfvert, and Hubertus Tummescheit. JModelica—an open source platform for optimization of Modelica models. In *Proceedings of MATHMOD 2009 - 6th Vienna International Conference on Mathematical Modelling*, Vienna, Austria, February 2009. TU Wien.

Christian Andersson, Sofia Gedda, Johan Åkesson, and Stefan Diehl. Derivative-free parameter optimization of functional mock-up units. In *Proceedings of the 9th International Modelica Conference - Munich, Germany, September 3, 2012*. Modelica Association, 2012.

Christian Andersson, Johan Åkesson, and Claus Führer. *PyFMI: A Python Package for Simulation of Coupled Dynamic Models with the Functional Mock-up Interface*, volume LUTFNA-5008-2016 of *Technical Report in Mathematical*

Sciences. Centre for Mathematical Sciences, Lund University, 2016.

Krzysztof Arendt. ModestPy: Parameter Estimation in FMI-compliant Models, 2017. URL `https://github.com/sdu-cfei/modest-py`. [Online; accessed April 25, 2018].

Javier Bonilla, Jose Antonio Carballo, Lidia Roca, and Manuel Berenguel. Development of an open source multi-platform software tool for parameter estimation studies in fmi models. In *Proceedings of the 12th International Modelica Conference, Prague, Czech Republic, May 15-17, 2017*, number 132, pages 683–692. Linköping University Electronic Press, Linköpings universitet, 2017.

Marco Bonvini, Michael Wetter, and Michael D. Sohn. An FMI-based Framework for State and Parameter Estimation. In *Modelica Conference 2014*, 2014.

Richard H. Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM J. Sci. Comput.*, 16(5):1190–1208, September 1995. ISSN 1064-8275. doi:10.1137/0916069.

Roel De Coninck and Lieve Helsen. Practical implementation and evaluation of model predictive control for an office building in brussels. *Energy and Buildings*, 111:290 – 298, 2016. ISSN 0378-7788. doi:https://doi.org/10.1016/j.enbuild.2015.11.014.

Roel De Coninck, Fredrik Magnusson, Johan Åkesson, and Lieve Helsen. Toolbox for development and validation of grey-box building models for forecasting and control. *Journal of Building Performance Simulation*, 9(3):288–303, 2016. doi:10.1080/19401493.2015.1046933.

Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001. URL `http://www.scipy.org/`. [Online; accessed April 25, 2018].

Muhyiddine Jradi, Fisayo Caleb Sangogboye, Claudio Giovanni Mattera, Mikkel Baun Kjærgaard, Christian Veje, and Bo Nørregaard Jørgensen. A world class energy efficient university building by danish 2020 standards. *Energy Procedia*, 132:21 – 26, 2017. ISSN 1876-6102. doi:https://doi.org/10.1016/j.egypro.2017.09.625. 11th Nordic Symposium on Building Physics, NSB2017, 11-14 June 2017, Trondheim, Norway.

Rüdiger Kampfmann, Danny Mösch, and Nils Menager. Parameter Estimation based on FMI. In *Proceedings of the 12th International Modelica Conference, Prague, Czech Republic, May 15-17, 2017*, number 132, pages 313–319. Linköping University Electronic Press, Linköpings Universitet, 2017.

Dieter Kraft. *A Software Package for Sequential Quadratic Programming*. Deutsche Forschungs- und Versuchsanstalt für Luft- und Raumfahrt Köln: Forschungsbericht. Wiss. Berichtswesen d. DFVLR, 1988.

Stephen G. Nash. Newton-Type Minimization Via the Lanczos Method. *SIAM Journal on Numerical Analysis*, 21(4):770–788, 1984. ISSN 00361429.

Peter Rockett and Elizabeth Abigail Hathway. Model-predictive control for non-domestic buildings: a critical review and prospects. *Building Research & Information*, 45(5):556–571, 2017. doi:10.1080/09613218.2016.1139885.

Fisayo Caleb Sangogboye, Krzysztof Arendt, Ashok Singh, Christian T. Veje, Mikkel Baun Kjærgaard, and Bo Nørregaard Jørgensen. Performance comparison of occupancy count estimation and prediction with common versus dedicated sensors for building model predictive control. *Building Simulation*, 10(6):829–843, Dec 2017. ISSN 1996-8744. doi:10.1007/s12273-017-0397-5.

Luigi Vanfretti, Maxime Baudette, Achour Amazouz, Tetiana Bogodorova, Tin Rabuzin, Jan Lavenius, and Francisco José Goméz-López. Rapid: A modular and extensible toolbox for parameter estimation of modelica and fmi compliant models. *SoftwareX*, 5:144 – 149, 2016. ISSN 2352-7110. doi:https://doi.org/10.1016/j.softx.2016.07.004.

Michael Wetter. Genopt - a generic optimization program. In Roberto Lamberts, Cezar O. R. Negrão, and Jan Hensen, editors, *Proc. of the 7th IBPSA Conference*, volume I, pages 601–608, Rio de Janeiro, 2001. URL `http://www.ibpsa.org/proceedings/BS2001/BS01_0601_608.pdf`.