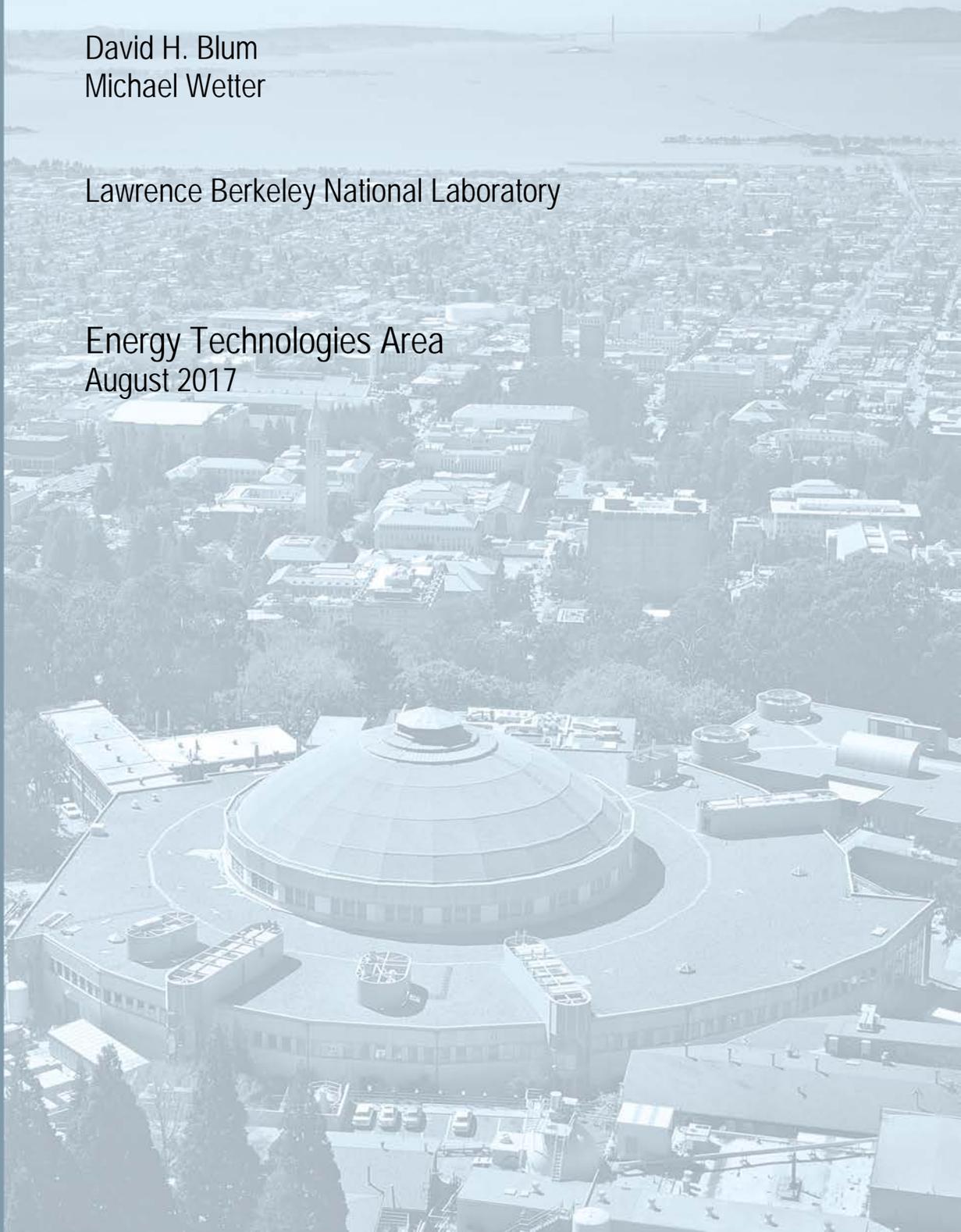**Lawrence Berkeley National Laboratory**

MPCPy: An Open-Source Software Platform for Model Predictive Control in Buildings

David H. Blum
Michael Wetter

Lawrence Berkeley National Laboratory

MPCPy: An Open-Source Software Platform for Model Predictive Control in Buildings

David H. Blum[1] and Michael Wetter[1]

[1]Lawrence Berkeley National Laboratory,

Energy Technologies Area

Building Technology and Urban Systems Division

Berkeley, CA, U.S.A.

Abstract

Within the last decade, needs for building control systems that reduce cost, energy, or peak demand, and that facilitate building-grid integration, district-energy system optimization, and occupant interaction, while maintaining thermal comfort and indoor air quality, have come about. Current PID and schedule-based control systems are not capable of fulfilling these needs, while Model Predictive Control (MPC) could. Despite the critical role MPC-enabled buildings can play in future energy infrastructures, widespread adoption of MPC within the building industry has yet to occur. To address barriers associated with system setup and configuration, this paper introduces an open-source software platform that emphasizes use of self-tuning adaptive models, usability by non-experts of MPC, and a flexible architecture that enables application across projects.

Introduction

Background

In an effort to limit climate change and decrease operating costs, energy systems have become the focus of widespread concern. This is especially true with those systems associated with buildings, which account for approximately 71% (EIA 2016a) of electricity use and 40% (EIA 2016b) of total primary energy use in the U.S. While buildings play the largest role in energy use, they are largely ill-equipped to handle new performance requirements brought about by new concerns. These requirements include energy or carbon minimization, peak demand minimization, integration with electrical and thermal district energy system operations, and occupant and operator feedback and connectivity. Many of these requirements depend upon a building being able to consider time-based incentives in the operation of multiple subsystems towards a common objective. Examples include shifting peak afternoon cooling loads towards morning hours, reducing energy use during times of high energy prices, coordinating PV generation, electric vehicle charging, and occupant service to limit the stress on the electric grid, and responsiveness to occupants.

Advancing the State of the Art

Current state of the art building control systems rely on a combination of PID feedback control and schedule-based setpoint managing without consideration of all of the necessary information to decide an optimal performance trajectory for a given objective. This includes forecasts of weather, energy prices, and building occupancy. In addition, the current control systems do not provide meaningful feedback to operators about the impact of certain control actions on system performance, which may help operators better manage systems according to their objectives.

Conversely, model predictive control (MPC) can meet the emerging requirements of building control systems. MPC uses system performance models, which include all of the relevant information, to forecast performance and optimize control inputs with respect to a given objective. These models can also provide useful feedback to system operators or building occupants for a number of operating scenarios.

A large body of work has shown that MPC can help enable buildings to meet these new requirements (Rockett and Hathway 2016). However, despite its widespread adoption in other industries (Qin and Badgwell, 2003) and success in research, it has not been widely adopted in the building industry, except for a few companies offering MPC as a software service for commercial buildings (BuildingIQ 2016, QCoefficient 2016) and campus central plants (Johnson Controls 2015). Rockett and Hathway (2016) point out several factors that contribute to the lack of penetration of MPC into industry, with the foremost being 1) the lack of long-term trials showing the effectiveness of MPC and 2) the expense and skill required for installation and maintenance. This is particularly true for initial model configuration and maintaining model accuracy as building operation changes over time. We believe these factors go hand-in-hand, where the high costs of installation and maintenance have prevented numerous long-term trials, and the low number of long-term trials have prevented the development of robust modeling and installation approaches.

Paper Objective

In order to address the problem of high system setup and maintenance costs, increase the number of trials of MPC in buildings, and facilitate widespread adoption of MPC in the building industry, this paper introduces the development of an open-source software platform for MPC in buildings, *MPCPy*, available on the LBNL Simulation Research Group github site at https://github.com/lbl-srg/MPCPy under a modified BSD license. A number of specific features are expected to contribute to the solution:

- An emphasis is put on the use of adaptive models, which use measurements of the building performance to continually update and remain accurate enough for control optimization, as illustrated in Figure 1. Such models are expected to drastically reduce model setup and maintenance costs.

- Automatic model parameter estimation and optimization problem formulation together with flexible data input modules reduce the required MPC and programming expertise of users.

- The use of open software standards enable contributions from other researchers and adoption by industry, while maintaining code maintenability and longevity due to the use of standards that have support in many industrial sectors.

- An extensible architecture enables rapid development and distribution of new MPC methods.
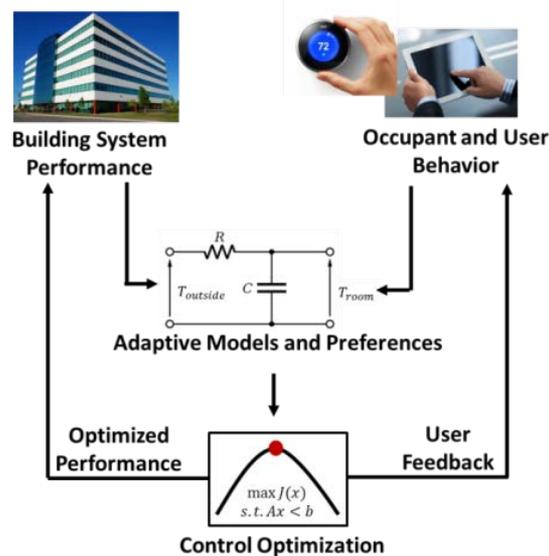

Figure 1 – MPCPy emphasizes the use of self-adapting models for MPC optimization.

While previous research has developed software frameworks for MPC in buildings, most have been developed for specific project requirements (such as data sources, processing methods, building simulation packages, and optimziation solvers) and are not made available for public use. Those that are offered to the public in some way require costly commercial software, such as MATLAB (Zakula et al. 2014, Bernal et al. 2012, Sturzenegger et al. 2014), or are focused on one aspect of the MPC process, such as reduced-order model development (Sturzenegger et al. 2014, DeConinck et al. 2016). In contrast, the framework introduced here looks to address all aspects for building MPC in an externsible way, is freely available, and is provided open-source.

The remainder of the paper will describe the architecture of *MPCPy* and present examples to showcase its capabilities. This paper will not address open research questions about proper adaptive model configurations and learning procedures, nor will it address guidelines for connecting to building automation systems (BAS). However, it is expected that *MPCPy* could be used extensively to answer these and other questions.

Architecture

MPCPy is designed using an object-oriented approach that promotes extensibility and is scripted in Python 2.7. The architecture is adapted from a tool-based architecture approach for computer-aided control design software (Barker et al. 1993 and Jobling et al. 1994), and is shown in Figure 2. In such an architecture, tools are developed to perform very specific functions, and then are supported by common processing agents and combined to perform various tasks. In *MPCPy*, four class modules represent the tools needed for MPC:

- **ExoData** classes collect external data and process it for use within *MPCPy*.

- **System** classes represent real or emulated systems to be controlled, collecting measurements from or providing control inputs to the systems.
- **Models** classes represent system models for MPC, managing model simulation, estimation, and validation.
- **Optimization** classes formulate and solve the MPC optimization problems using Models objects.

Supporting the four tools are three modules for data handling and processing during run-time:

- **Variable** and **Unit** classes together maintain the association of static or timeseries data with units.
- **Utility** classes provide functionality needed across tools and for interactions with external components.

MPCPy does not contain any of its own model specifications or solvers. Instead, it relies on external software, modeling languages, and solvers. While dependencies can be expanded to include other software, these external components are currently based on the Modelica (Mattsson and Elmqvist 1997) and Functional Mock-up Interface (FMI 2016) open standards, with system emulation models and MPC models able to be defined as native Modelica files or as Functional Mock-up Units (FMU). An FMU is a zipped file containing the variables and equations of a model that can be used to exchange the model between simulation programs. Optionally, solvers and additional data can be packaged
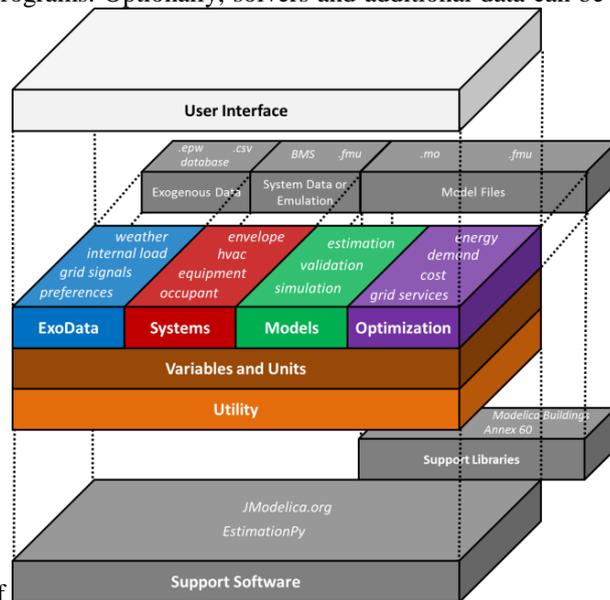
as well. Substantial development of the application of

Figure 2 – MPCPy architecture these standards to building simulation and optimization is ongoing (Wetter et al. 2015 and Wetter and Treeck 2015). JModelica (Modelon AB 2009), an open-source Modelica compiler, is used to compile native Modelica models, simulate FMUs, and solve optimization problems, while EstimationPy (Bonvini et al. 2014) can be used for model parameter and state estimation using FMUs. Therefore, any Modelica library can be used to create native Modelica models, including the Modelica Buildings Library (Wetter et al. 2014) or any from the Annex 60 collaboration (Wetter et al. 2015), as can any modeling software capable of generating FMUs. While there exist methods to solve control optimization problems using highly detailed, non-differentiable models (Wetter 2001), we strongly recommend the use of differentiable models for control optimization, which can be solved very efficiently (Wetter et al. 2016).

The following sections will detail the features of the Variables, Units, ExoData, Systems, Models, and Optimization module classes.

Variables and Units

Variable and Units classes allow *MPCPy* to effectively associate data with units as well as handle timeseries data separately from static (constant) data. The Python package Pandas (McKinney 2010) is integrated into the handling of timeseries variables, bringing to *MPCPy* the many useful features Pandas has to offer for data storage, cleaning, resampling, interpolation, and statistical analysis.

Figure 3 presents a basic Unified Modeling Language (UML) class diagram, commonly used in the field of software development, showing the relationship of Variables and Units classes. The Variable class defines common methods for *MPCPy* variables, while the Static and Timeseries classes inherit the Variable class and implement methods more specific to data that is constant and time-varying respectively. Units are added to variable objects upon instantiation and act upon the variable data depending on the action. Each unit is part of a quantity type. For example the unit degC is a temperature quantity. Each quantity has a base unit in which all data is stored, which follows the convention defined by the Modelica Standard Library (Modelica Association 2016). The display unit, however, defines the

conversion of the data to that base unit upon input, or conversion from that base unit upon display or output. The DisplayUnit class defines these required methods, with additional methods specified in quantity-specific classes.
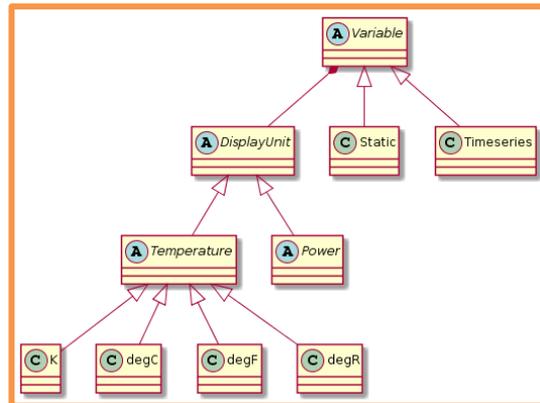

Figure 3 – Variables and Units class diagram. Classes labelled "A" are abstract, "C" are concrete.

ExoData

ExoData classes are responsible for the representation of exogenous data, with methods to collect this data from various sources and process it for use within *MPCPy*. Exogenous data is separated according to a data class and each data class has a specified variable organization in the form of a Python dictionary. This allows for an internal understanding of where objects should look for specific data and the use of specific checks for each data class. At the time of this writing, there are eight types of data classes: Weather, Internal, Controls, Constraints, Prices, Parameters, and Other Inputs.

Figure 4 presents a basic UML class diagram showing the relationship of ExoData classes. At the top is the Type class that contains common methods for all exogenous data class types. Then, there exists an abstract class for each exogenous data class, which inherits the Type class and adds additional methods specific for that data class. These include assertions that particular variables fall within limits or methods for manipulating the data dictionary for that data class. Finally, source-specific classes inherit the type-specific data classes and add methods required to collect data from a particular source. For example, an EPW file or CSV file may be the source of weather data. At the time of this writing, aside from this choice, CSV files must be the source of all other data types. However, similar to the EPW and CSV example, the class designs allow for easy addition of other data class sources and types.
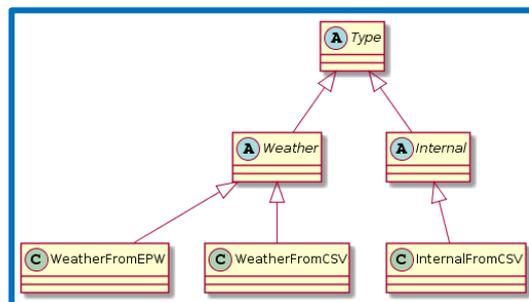

Figure 4 – ExoData class diagram. Classes labelled "A" are abstract, "C" are concrete.

Systems

Systems classes represent the controlled systems, with methods to collect measurements from or set control inputs to the system. This representation can be real or emulated using a detailed simulation model. A common interface to the controlled system in both cases allows for algorithm development and testing on a simulation with easy transition to the real system. Similar to exogenous data, measurement data has a specified variable organization in the form of a Python dictionary in order to aid its use by other objects.

Figure 5 presents a basic UML class diagram showing the relationship of Systems classes. The System class contains common methods for all system types. Then, real and emulated system classes inherit the top-level System class and add methods that are more specific. For example, the emulated system class requires methods to simulate a model, rather than establish a connection with a data server or control system serving a real system. Finally, concrete classes inherit the system type methods and implement methods that are more specific for the emulation or real system source

type. At the time of this writing, *MPCPy* only supports emulation by FMU simulation, however, the class design allows for adding emulation methods and real system sources.
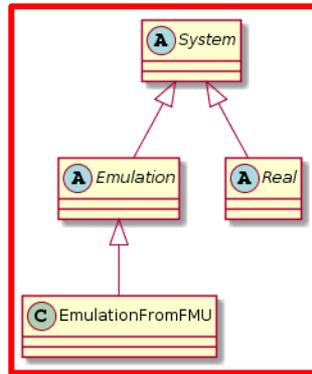

Figure 5 – Systems class diagram. Classes labelled "A" are abstract, "C" are concrete.

Models

Models classes represent system models that can be used for MPC optimization, with methods to simulate the model, estimate parameters or states based on measured system data, and validate the model based on measured system data. These are often reduced-order or simplified models when compared to a system emulation model.

Figure 6 presents a basic UML class diagram showing the relationship of Models classes. The Model class contains common methods for an MPC-suited model. The Model class has two classes for an estimation method and a validation method. The Estimate class contains methods common for solving parameter or state estimation problems. Implementations of the Estimate class contain specific methods required to setup and solve the estimation problem with various algorithms. For example, the estimation problem may be formulated as an optimization problem, to be solved in JModelica, or formulated for an Unscented Kalman Filter (UKF), implemented by EstimationPy. Meanwhile, the Validate class contains methods common for validating the estimation process and concrete implementations of this class apply more specific validation algorithms. For example, this could be calculating the RMSE between measured data and simulated data with estimated parameters. Lastly, the Modelica class inherits the Model class and adds methods to handle Modelica and FMI-based models. Many of these methods are inherited from an FMU class in the Utilities module, not described in detail here. The class design of models is in such a way that allows for adding other estimation and validation methods as well as model formats.
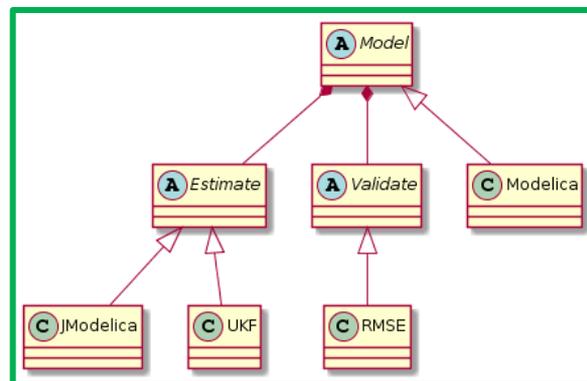

Figure 6 – Models class diagram. Classes labelled "A" are abstract, "C" are concrete.

Optimization

Optimization classes represent MPC optimization problems, with methods to setup and solve such problems.

Figure 7 presents a basic UML class diagram showing the relationship of Optimization classes. The Optimization class implements methods for setting up and solving an MPC optimization problem. It has two classes for a problem and an optimization package. The Problem class contains common methods for defining an optimization problem. Implementations of Problem add specific methods to setup a particular problem type. At the time of this writing, *MPCPy* supports parameter estimation, energy minimization, and energy cost minimization. The Package class

contains common methods for setting up an optimization package to solve the defined optimization problem. Implementations add specific methods for a particular optimization package. At the time of this writing, *MPCPy* supports the use of JModelica as a package. Note that an implementation of a Package should be able to solve any of the problems defined by a Problem class. In this way, the class design allows for the addition of other problem types, such as peak load minimization or demand response maximization, and optimization packages.
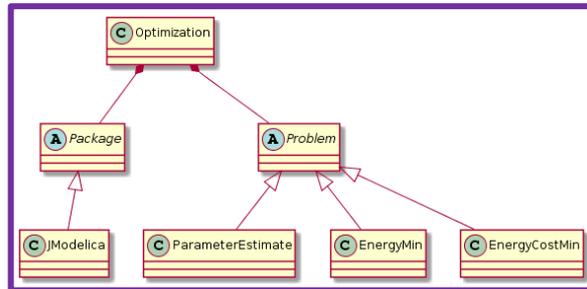

Figure 7 – Optimization class diagram. Classes labelled "A" are abstract, "C" are concrete.

A key feature of *MPCPy* is the ability to automatically setup and solve optimization problems, requiring only model files, exogenous input data, and optimization constraints. At the time of this writing, this feature is supported for models defined in native Modelica and problem types of parameter estimation, energy minimization, and energy cost minimization using JModelica as an optimization package. This automated process begins with automatic modification of the given native Modelica model file (.mo) into an Optimica (Akesson 2008) Modelica file (.mop) based on the optimization problem type. This includes the objective of the problem type chosen, constraints on model states, or identification of parameters or states to be estimated. Figure 8 shows example modifications that are made for an energy cost minimization problem. Next, the .mop file is used by JModelica to instantiate an optimization problem. Upon solver runtime, the optimization problem is combined with solution-specific information and passed to the optimization solver. This solution-specific information includes exogenous input data or measurement data required for model estimation.

Examples

This section presents a number of examples to show the capabilities of *MPCPy*. Examples are for variable and unit management, collecting exogenous data, building emulation, model estimation and validation, and control optimization. Note that the examples are simple and emphasize the processes enabled by *MPCPy*, not the specific outcomes of those processes.
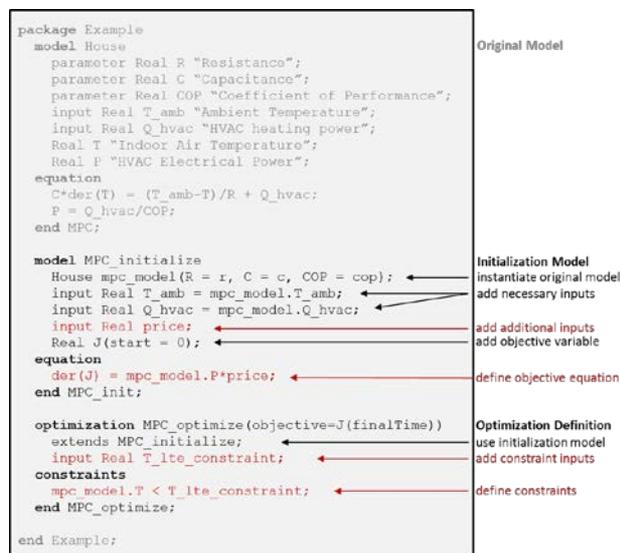

Figure 8 – .mo file to .mop file modifications for an energy cost minimization problem with grey text representing original .mo code, red text representing case-specific code additions, and black code representing general code additions.

Variable and Unit Management

We begin by defining a single static variable with units of $^{\circ}$C, which may represent a thermostat setpoint.

```
>>> from mpcpy import variables
>>> from mpcpy import units
>>> T_set = variables.Static('T_set',20,units.degC)
>>> print(T_set)
Name: T_set
Variability: Static
Quantity: Temperature
Display Unit: degC
```

The first two lines of code import the Variables and Units modules from *MPCPy*. The third line instantiates the static variable with three arguments; name, data, unit. Printing the variable displays relevant information about the variable, including that it has a quantity of temperature. Therefore, the data is actually stored in units of Kelvin. We can check this by getting the base unit and data of the variable.

```
>>> T_set.get_base_unit()
mpcpy.units.K
>>> T_set.get_base_data()
293.15
```

The data can also be displayed in the display units.

```
>>> T_set.display_data()
20.0
```

The display unit can also be changed.

```
>>> T_set.set_display_unit(units.degF)
>>> T_set.display_data()
68.0
```

Note that functionality is the same with timeseries variables, however, data is supplied in the form of a Pandas series variable with a timestamp index instead of a single value. By default, the Pandas timeseries specified is assumed to be in UTC time, however, optional arguments upon instantiation allow for timeseries in local time zones to be specified as well. The timeseries is converted to UTC time for storage, though can be again converted to local upon data display.

Collecting Exogenous Data

Next, we would like to collect exogenous data from a source. First, we collect weather data from an EPW file.

```
>>> from mpcpy import exodata
>>> weather = exodata.WeatherFromEPW('path.epw')
>>> weather.collect_data('1/1/2015', '1/4/2015')
```

The first line of code imports the exodata module from *MPCPy*. The second line instantiates a weather data object with an EPW file as the source, while the third line collects data from the file for the time-period specified. The time-period can be in any form recognizable to Pandas or a custom format, optionally defined upon object instantiation.

Now, we collect weather data from a CSV file.

```
>>> variable_map = {'TemperatureF' :
...                 ('weaTDryBul', units.degF),
...                 'Dew PointF' :
...                 ('weaTDewPoi', units.degF),
...                  'Humidity' :
...                 ('weaRelHum', units.percent),
...                 'Sea Level PressureIn' :
...                 ('weaPAtm', units.inHg),
...                 'WindDirDegrees' :
...                 ('weaWinDir', units.deg),
...                 'Wind SpeedMPH' :
...                 ('weaWinSpe', units.mph)};
>>> weather =
... exodata.WeatherFromCSV(
...     'path.csv', variable_map,
...     time_header = 'TimePDT',
...     tz_name = 'from_geography',
...     geography = [37.8716, -122.2727]);
>>> weather.collect_data('1/1/2015', '1/4/2015')
```

The first line of code defines a mapping dictionary between the header columns of the CSV file and the input variables in the emulation or MPC models. The dictionary keys are the column headers and the value of each key is a tuple of the model input name and data units as displayed in the CSV file. Note that using the Variable and Units classes, the data

is automatically converted to the base unit for use within the model. The second line of code instantiates a weather object with a CSV file as a source and the variable map already defined. A few optional arguments are shown as well, which define which CSV column header name contains the timestamps and in which time zone the timestamps are. By default, the object will read timestamps from a column header "Time" or "Timestamp" and will assume UTC time. Additional arguments for specifying a custom timestamp format and data cleaning algorithms are available, but not discussed further here.

Building Emulation

Next, we look to simulate a building emulation model. The building we will use is based on a test facility on the Berkeley Lab campus, called LBNL71T and shown in Figure 9. It is a three-zone facility, with two of the zones having exterior walls, one facing east and the other facing west. For example purposes, we model each zone having a single convective heater. An emulation model is built using components from the Modelica Buildings Library, namely three thermally connected instances of the Buildings.ThermalZones.Detailed.MixedAir, and exported as a model-exchange FMU v 2.0. Peak internal loads total 20 W/m^2 with a 40%-40%-20% split between convective, radiative, and latent and a typical weekday office schedule. For demonstration purposes the building is located in Chicago, IL, a time zone that is UTC-6:00.
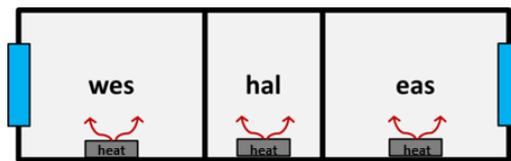

Figure 9 – LBNL71T test building, used for demonstration of MPCPy.

First, we must define what measurements we wish to take from the model. The code shown below defines a measurement dictionary by indicating the name of the variable we wish to measure and its sample rate. Note that upon simulation, the output reporting time-step is equal to the minimum defined measurement sample rate.

```
>>> measurements = {};
>>> measurements['wesTdb'] = {'Sample' :
. . . variables.Static('wesTdb_sample', 600, units.s)};
>>> measurements['halTdb'] = {'Sample' :
. . . variables.Static('halTdb_sample',1200, units.s)};
>>> measurements['easTdb'] = {'Sample' :
. . . variables.Static('easTdb_sample',1200, units.s)};
```

With the measurements defined, we can instantiate our building emulation object, simulate the building, and plot the measurements.

```
>>> from mpcpy import systems
>>> from matplotlib import pyplot as plt
>>> building = systems.EmulationFromFMU(
. . .     'path.fmu', measurements,
. . .     weather_data = weather.data, \
. . .     internal_data = internal.data, \
. . .     control_data = control.data, \
. . .     parameter_data = parameters.data, \
. . .     tz_name = weather.tz_name);
>>> building.collect_measurements(
. . .     '1/1/2015', '1/4/2015');
>>> plt.figure(1)
>>> for key in self.building.measurements.keys():
. . .     variable =
. . .         building.measurements[key]['Measured'];
. . .     variable.set_display_unit(units.degC);
. . .     variable.display_data().plot(label = key);
. . .     plt.legend();
. . .     plt.ylabel(variable.quantity_name + '[' +
. . .         variable.display_unit.name + ']');
```

The first line of code imports the Systems module from *MPCPy* and the second imports the matplotlib package (Hunter 2007). The third line of code instantiates the building system object as an FMU and with the measurement dictionary already defined. Other optional arguments of the instantiation define the exogenous data to use, collected as shown before, parameters of the emulation model, and the building time zone. Note that the weather data comes from the Chicago, IL EPW file, while internal load data and heater control signals come from separate CSV files. The fourth line of code simulates the emulation model for the time-period specified, according to the building time zone. Finally, the last lines of code plot the measurements using the Pandas plot function and *MPCPy* variable attributes. Figure 10 shows the resulting plot.
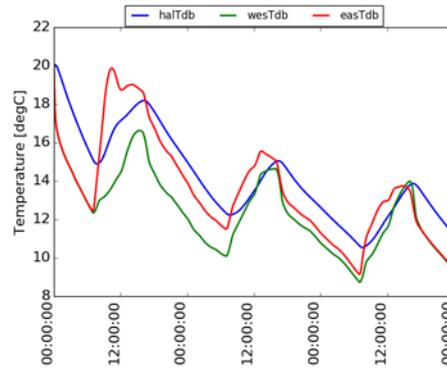
Figure 10 – Mean zone air temperatures of LBNL71T in Chiacgo, IL from January 1-3.

Model Estimation

Next, we look to estimate the parameters of a reduced-order model of our emulated building, shown in Figure 11. The reduced order model is an RC circuit that models sensible heat transfer only. Each zone interior is modelled with two capacitances with a resistance between them, representing the air capacitance and interior thermal mass capacitance. The zone exterior walls are modelled with two resistances and one thermal capacitance as well as a solar absorbance coefficient for solar irradiation incident on the wall. Windows are modelled with a single resistor and solar transmittance coefficient. Lastly, interior partitions between zones are modelled with two resistors and a single capacitor. The components are modelled using the Modelica Standard Library and Modelica Buildings Library.

While this paper shows the approach of building a library of reduced order model components is promising, the purpose of this model is not to represent answers to questions about what the proper reduced-order model configurations is to use for multi-zone buildings and HVAC systems. Also, this example does not address methods to improve model training, including necessary model excitation or data characteristics. Rather, the model's purpose here is to be a suitable demonstration of *MPCPy* capabilities. These other important questions are saved for future work, and can be exercised using *MPCPy*. We are now ready to instantiate, estimate, and validate the reduced order model.

```
>>> from mpcpy import models
>>> model = models.Modelica(
...       models.JModelica,
...       models.RMSE,
...       building.measurements,
...       moinfo =
...            ('path.mo', 'package.model', libraries)
...       weather_data = weather.data, \
...       internal_data = internal.data, \
...       control_data = control.data, \
...       parameter_data = parameters.data, \
...       tz_name = weather.tz_name);
>>> model.estimate('1/1/2015', '1/4/2015',
...                ['wesTdb', 'halTdb', 'easTdb']);
>>> building.collect_measurements(
...       '1/4/2015', '1/5/2015');
>>> model.measurements = building.measurements;
>>> model.validate('1/4/2015', '1/5/2015',
...                'respath', plot = 1);
```

Figure 11 – RC model representing the LBNL71T test building, used for demonstration of MPCPy. The parameters to train are shown next to each component Modelica diagram.
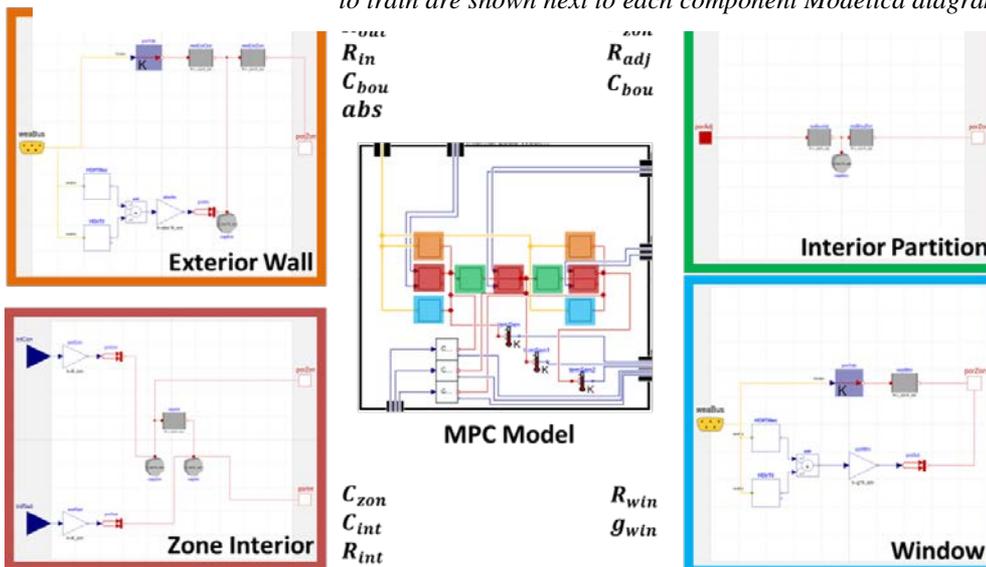
The first line of code imports the Models module from *MPCPy*. The second line of code instantiates the model object with an estimation method (JModelica), validation method (calculate RMSE), system measurements, and

Modelica model information, as well as exogenous data as collected before. The system measurements are passed using the measurement dictionary formulated after collecting measurements with the building object. The model information requires the file path, model path within the file, and any libraries required to compile the model. The parameter data is collected from a CSV as exogenous data and contains information about the parameters in the model to be estimated, including their minimum and maximum values, initial guesses, and covariances (applicable to the UKF estimation method if used). The third line of code estimates the model parameters using the specified period of measurement data (Jan. 1-3) and measurement variables.
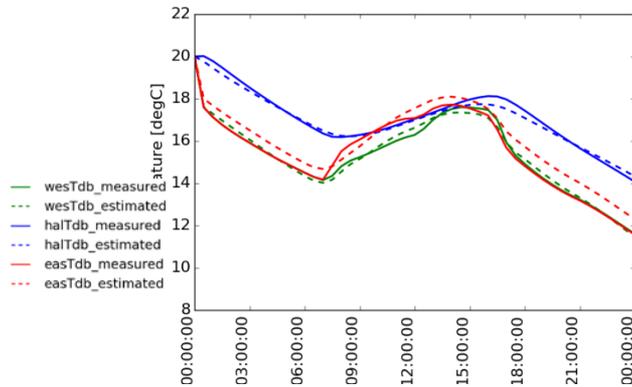

Figure 12 – Mean zone air temperatures over the validation period on Jan. 4 for LBNL71T RC model Shown in local time.

Finally, the last three lines of code validate the model using data from the day after the training period using the specified period of measurement data, with the validation results output to a defined directory path. The results of model validation are shown in Figure 12. The RMSE for wesTdb, halTdb, and easTdb are 0.32 K, 0.27 K, and 0.40 K for the training period, and 0.18 K, 0.20 K, and 0.58 K for the validation period.

Control Optimization

Lastly, we optimize heater control to minimize energy use and cost while maintaining thermal comfort.

```
>>> from mpcpy import optimization
>>> self.opt_problem =
...      optimization.Optimization(model,
...      optimization.EnergyMin,
...      optimization.JModelica,
...      'Ptot',
...      constraint_data = self.constraints.data);
>>> opt_problem.optimize('1/2/2015', '1/3/2015');
```

The first line of code imports the Optimization module from *MPCPy*. The second line of code instantiates an optimization problem object by defining the model to use, the optimization problem, the optimization solver, the objective variable within the model, which is total power in this case (variable "Ptot"), and constraint data. The model is the model object instantiated as shown previously. The constraint data is collected as an exogenous data object from a CSV file and contains timeseries information about greater-than-or-equal-to and less-than-or-equal-to constraints on state variables and their derivatives, as well as initial value, final value, or cyclic (initial equals final) constraints. For this example, the zone mean air temperatures are constrained to between 22°C and 25°C, their derivatives are constrained to less than 2 K/h, and their initial and final values must be equal. This cyclic constraint assumes that neighbouring days would have similar conditions and preferred performance. The heater control signals are constrained to between 0 and 1. The last line of code solves the optimization problem over the time period specified. Note that all other exogenous data, such as weather and internal loads, are included in the model object that is passed. The optimization will optimize all variables in the "control_data" attribute of the model, which in this case are the heater control signals.

```
>>> opt_problem.set_problem_type(
...      optimization.EnergyCostMin);
>>> opt_problem.optimize('1/2/2015', '1/3/2015',
...      price_data = prices.data);
```

Lastly, the objective is changed to minimize energy cost. Here, we add price data, collected as an exogenous data object as shown previously, to the optimize statement. The price data is automatically integrated with the model optimization, in this case multiplying together with the defined objective variable, "Ptot." In the case demonstrated, the price of electricity is five times higher during the hours of 6:00 PM, 7:00 PM, and 8:00 PM local time than the rest of the day. The solutions to the energy minimization and energy

cost minimization problems are presented in Figures 13 and 14. Notice that temperature constraints are not violated and the inclusion of price spikes incentives a significant shift in heating power away from the hours with higher prices.

Discussion

The examples showed the systematic, yet flexible, approach *MPCPy* takes to setup MPC for buildings, upon which industry and researchers can build and test their implementations. Variables and Units provide flexibility during the input and output of data with respect to data units and timeseries time zones. Exogenous data of various types and with different data sources and formats can be combined together for use during model simulation, estimation, validation, and optimization. Common interfacing with real and emulated systems eases transition from controller development to application. Automatic setup of model parameter estimation and optimization algorithms minimizes the user programming and expertise requirements. Lastly, an extensible architecture enables new data sources, simulation procedures, problem types, and solver methods to be added in the future.

Conclusion

The requirements of building control systems are growing in a way that requires improved information processing and decision making over the state of the art.
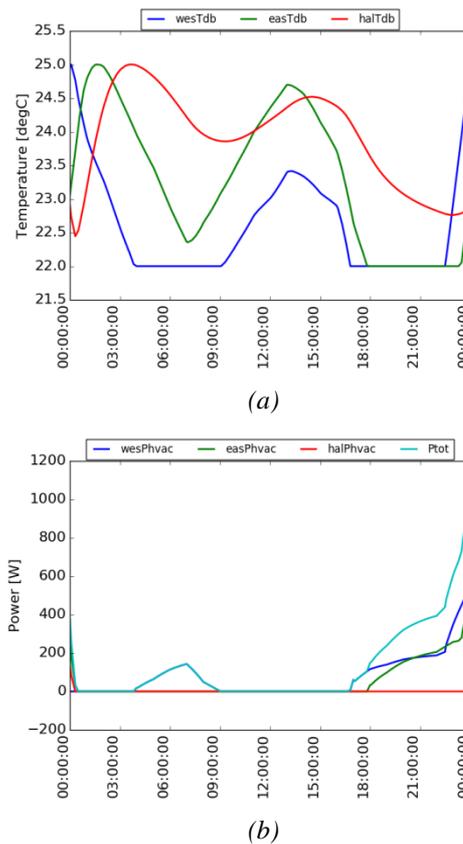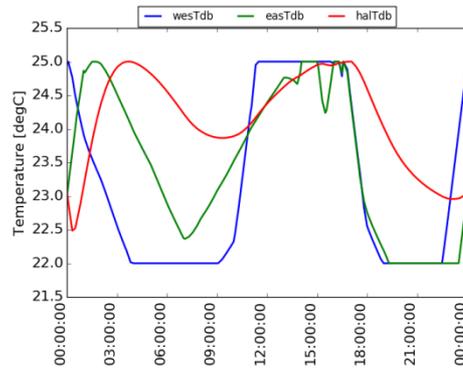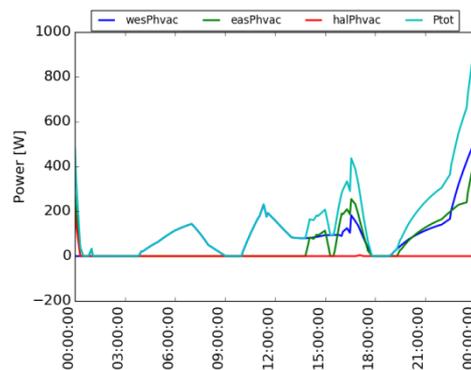
(a)

(b)

Figure 13 – Zone mean air temperature (a) and heater power (b) solutions for energy minimization of LBNL71T RC model on Jan. 2nd. Shown in local time.

Past research and few demonstrations have shown that MPC can meet these new requirements. However, the building industry has yet to see widespread adoption of MPC control systems, likely due to high setup costs and maintenance for individual sites. To address these barriers, this paper introduces a freely available open-source MPC platform for buildings based on open standards, called *MPCPy*, available on the LBNL Simulation Research Group github site at https://github.com/lbl-srg/MPCPy under a modified BSD license. The platform emphasizes the use of adaptive models, whose parameters can be learned over time with measured building data, and the automation of model learning and optimization problem setup and solving. Both of these features are expected to significantly reduce the required system setup time and expertise. In addition, *MPCPy* is designed for extensibility and based on open-source standards so that new data sources, model types, procedures, and problems can be added over time without sacrificing code longevity.

Immediate future work is three-fold, having to do with functionality, demonstration, and support for the public release. Additional functionality includes methods for real system implementation, support for additional optimization problems, such as peak load minimization, and incorporation of occupant behaviour models.

(a)

(b)

Figure 14 – Zone mean air temperature (a) and heater power (b) solutions for energy cost minimization of LBNL71T RC model on Jan. 2nd. Shown in local time.

Future functionality also includes application of *MPCPy* at various system scopes, including the room, building, and campus levels. Development of a Modelica component library for MPC applications is planned by Wetter and Treeck (2015). Demonstrations of *MPCPy* at real sites are planned to take place over the next five years by LBNL and collaborators. However, we hope that the public release provides others an opportunity to demonstrate its capabilities as well, contribute to its development, and create a community of users.

The current version of *MPCPy* is v0.1. Development is ongoing for the support of the public release, including additional features, documentation, error handling, regression testing, and interface adjustments. Updates will be released periodically according to user feedback and developer progress. To contribute to this project's development or inquiry about collaboration, visit the *MPCPy* github site or contact the authors.

Acknowledgements

References

Akesson, J. (2008). Optimica – An extension of Modelica supporting dynamic optimization. In Proceedings of the 6th International Modelica Conference, Bielefeld, Germany, March 3-4, 57-66.

Barker, H. A., M. Chen, P. W. Grant, C. P. Jobling, and P. Townsend (1993). Open architecture for computer-aided control engineering. *IEEE Control Systems 13(2)*, 17-27.

Bernal, W., M. Behl, T. X. Nghiem, and R. Mangharam (2012). MLE+: A tool for integrated design and deployment of energy efficient building controls. *In Proceedings of Buildsys 2012*, Toronto, Canada, November 6.

Bonvini, M., M. Wetter, and M. D. Sohn (2014). An FMI-based framework for state and parameter estimation. In *Proceedings of the 10th International Modelica Conference*, Lund, Sweden, March 10-12, 647-656.

BuildingIQ (2016). https://buildingiq.com/. Last accessed Nov. 18, 2016.

De Coninck R., F. Magnusson, J. Akesson, and L. Helsen (2016). Toolbox for development and validation of grey-box building models for forecasting and control. *Journal of Building Performance Simulation 9(3)*, 288-303.

EIA (2016a). Monthly Energy Review October 2016, Table 7.6. Available online at http://www.eia.gov/totalenergy/data/monthly/pdf/sec7_19.pdf. Last accessed Nov. 18, 2016.

EIA (2016b). Monthly Energy Review October 2016, Table 2.1. Available online at http://www.eia.gov/totalenergy/data/monthly/pdf/sec2_3.pdf. Last accessed Nov. 18, 2016.

FMI (2016). Functional Mock-up Interface. Available online at https://www.fmi-standard.org/start. Last accessed Nov. 18, 2016.

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science and Engineering 9(3)*, 90-95.

Jobling, C. P., P. W. Grant, H. A. Barker, and P. Townsend (1994). Object oriented programming in control system design: a survey. *Automatica 30(8)*, 1221-1261.

Johnson Controls (2015). Johnson Controls helps Stanford University drastically reduce water and energy use in new central plant. Available online at http://www.johnsoncontrols.com/media-center/news/press-releases/2015/06/15/johnson-controls-helps-stanford-university-drastically-reduce-water-and-energy-use-in-new-central-plant. Last accessed Nov. 18, 2016.

Mattsson, S. E. and H. Elmqvist (1997). Modelica – An international effort to design the next generation modeling language. In *7th IFAC Symposium on Computer Aided Control Systems Design*, Gent, Belgium, April 28-30.

McKinney, W. (2010). Data structures for statistical computing in Python. In *Proceedings of the 9th Python in Science Conference*, Austin, Texas, June 28 – July 3, 51-56.

Modelica Association (2016). Modelica Standard Library. https://github.com/modelica/Modelica. Last accessed Nov. 18, 2016.

Modelon AB (2009). Jmodelica.org – Open source Modelica platform for modeling, simulation, and optimization. Available online at http://jmodelica.org/. Last accessed Nov. 18, 2016.

QCoefficient, Inc. (2016). http://qcoefficient.com/. Last accessed Nov. 18, 2016.

Qin, S., and T. Badgwell (2003). A survey of industrial model predictive control technology. *Control Engineering Practice 11(7)*, 733-764

Rockett, P. and E. Hathway (2016). Model-predictive control for non-domestic buildings: a critical review and prospects. *Building Research & Information*, DOI: 10.1080/09613218.2016.1139885.

Sturzenegger, D., D. Gyalistras, V. Semeraro, M. Morari, and R. S. Smith (2014). BRCM Matlab Toolbox: Model generation for model predictive building control. *In Proceedings of the 2014 American Control Conference (ACC)*, Portland, OR, June 4-6, 1063-1069.

Wetter, M. (2001). GenOpt – A generic optimization program. *In Proceedings of the 7th International IBPSA Conference*, Rio de Janeiro, Brazil, August 13-15, 601-608.

Wetter, M., W. Zuo, T. S. Nouidui, and X. Pang (2014). Modelica Buildings library. *Journal of Building Performance Simulation 7(4)*, 253-270.

Wetter, M., M. Fuchs, P. Grozman, L. Helsen, F. Jorissen, M. Lauster, D. Muller, C. Nytsch-Geusen, D. Picard, P. Sahlin, and M. Thorade (2015). IEA EBC Annex 60 Modelica Library -- An international collaboration to develop a free open-source model library for buildings and community energy systems. In *Proceedings of the 14th Conference of International Building Performance Simulation Association*, Hyderabad, India, Dec. 7-9.

Wetter, M., and C. van Treeck (2015). IBPSA Project 1. Available Online at https://ibpsa.github.io/project1/. Last accessed Mar. 3, 2017.

Wetter, M., M. Bonvini, and T. S. Nouidui (2016). Equation-based languages – A new paradigm for building energy modelling, simulation and optimization. *Energy and Buildings 117*, 290-300.

Zakula, T., P. R. Armstrong, and L. Norford (2014). Modeling environment for model predictive control of buildings. *Energy and Buildings*, 85, 549-559.