Master Thesis 2016

Ole Magnus Brastein

130482

Grey-box models for estimation of heating times for buildings

Telemark University College
Faculty of Technology

Telemark University College
Faculty of Technology

Kjølnes ring 56
3918 Porsgrunn
http://www.hit.no

Abstract

Energy-usage in buildings is responsible for a large part of the total demands on energy production. Models are required to estimate the heating and cooling times of buildings, thus allowing a control system to accurately maintain comfort temperature only when strictly needed, lowering the demand for energy used for heating.

Grey-box models based on Thermal Network Resistor-Capacitor equivalents are used to predict thermal behavior of buildings. Model structures are based on cognitive or intuitive understanding of thermodynamic behavior of buildings.

Working with models and data sets requires software tools both for treatment of data, simulation of models and identification of parameters. In this project software is developed both in c# and MATLAB as and when applicable.

Grey-box models are shown to accurately predict temperature over the prediction horizon, leading to accurate estimation of heating time, when compared to measurement data. Further, the proposed control strategy is shown to overcome some of the shortcomings of standard heater control systems.

Table of contents

Overview of figures

Overview of tables

Preface

This project report and the underlying software development and modeling work have been a part of the authors' master studies at Telemark University Collage at the program Systems and Control Engineering. The methods and tools described in this project presents a foundation for future work with regards to the use of grey-box modeling to predict heating and cooling time of buildings, as well as some conceptual ideas for a working control system. Developed software can serve as tools for future work on the subject, in particular with regards to data pretreatment and simulations of models.

The report outlines several suggestions and recommendations with regards to data collection, which is hoped to be useful for future work. In particular, ranges for training data in both time and variation of inputs is treated in detail. Further, suggestions to future work and improvements on the presented methods are given where applicable, especially with regards to model structures and control algorithm.

I would like to thank my supervisor Nils-Olav Skeie for all guidance, support and advice throughout the project, and PhD candidate Wathsala Perera for her contribution, particularly in the form of collected data and her work in deriving white-box models which is used as a reference in this project. Finally, I would like to thank my family for their support during this project.

Larvik, 25.01.2016

Ole Magnus Brastein

1 Introduction

Energy-usage in buildings is responsible for a large part of the total demands on energy production, particularly in countries with cold climates. According to the International Energy Agency (IEA) some 32% of the total energy consumed is used for heating, cooling and lighting of commercial and residential buildings [1]. In some countries, taking Norway as an example, heating of buildings consume as much as 48% of the total energy production (Appendix A).

Modern building techniques, particularly in Scandinavian countries, go along way in making houses more energy efficient. However, the renewal rate of buildings is slow. Taking France as an example, the renewal rate is reported as only 1% pr year [2]. Upgrading existing buildings to modern standards of energy economy is expensive. Therefore, it is of importance to develop methods and systems which is capable of reducing energy demands for existing buildings.

The use of electricity for heating buildings generates peak demands on the power grid, due to simultaneous use of large amount of energy, hence prediction models for buildings are of interest on a city-wide basis as well [2]. Some literature also suggest that increased use of renewable energy sources may lead to higher demands for predictability in consumption of electric energy [3].

The motivation for this project is developing models that can predict heating and cooling times for buildings. This type of model can then be used in a control system to predict the thermal behavior. Energy usage for heating can be lowered by allowing the building to cool to lower temperatures when not in use (Appendix A).

A study of current literature, with focus on models for thermal behavior in buildings combining physical laws and data driven modeling, is carried out as a summary of previous work in the field. The result of this study, which is presented in a separate chapter, suggest that using grey-box modeling techniques is a typical approach to creating models of thermal behavior in buildings. Therefore, grey-box models are the primary choice of modeling technique [4] in this project.

A grey-box model is named as such because it combines concepts from both white-box and black-box modeling. The definition of grey-box model used in this project is taken from [4], where grey-box models is defined as models where the basic structure is first formed by using the laws of physics and then using measurement data from a system to identify the parameters of the model.

An advantage of the grey-box modeling technique is that the physical structure of the model can be designed based on intuitive understanding of a system [5] rather then complex mathematical equations. Thermal behavior of buildings has been shown to depend on many factors, such as weather conditions, thermal insulation, usage patterns and thermal mass of building structure and furniture [1]. Hence, building models can become complex, if all of these factors are to be modeled by physical laws.

According to the literature, a typical choice for model structures in grey-box modeling of buildings is a Thermal Network [5], where the building is represented as a simplified lumped parameter model [2]. The thermal resistance and capacitance of a building is modeled as a Resistor-Capacitor (RC) electrical equivalent circuit [6]. A detailed discussion on these concepts is given in a separate chapter.

Much of the relevant literature is concerned with the models themselves, and on evaluating a models ability to predict temperature relative to a known, measured, reference [2, 3, 7]. The focus of this project is on predictions of heating and cooling times. The grey-box modeling technique is used to derive models for several different buildings, both from simulated and physical measurement data. The derived models are shown to predict heating time of buildings with good accuracy. This project adds to the previous work in the field by looking at grey-box models for heating and cooling time estimation in the frame-work of a specific predictive control system.

A detailed discussion of requirements for the measurement data is given, with focus on what is important for a models ability to accurately predict temperature, as well as heating and cooling times. A suggested predictive control scheme is given as a framework of these time predictions. The results presented in this report shows that the thermostats of electric heaters have sub-optimal performance with respect to minimizing heating time. This problem is also discussed in the framework of the suggested predictive control system.

In any data-driven modeling technique a system for pre-processing of data files is required. A software tool is developed especially for this purpose. Detailed presentations of the various pre-processing operations of interest to this project are given.

Other software, developed as part of this project, is focused on the simulation of models based on Ordinary Differential Equations (ODE) such as those derived from Thermal Network models. Simulations of models are implemented both in MATLAB and c# and results are compared with respect to accuracy and computational time. Discussions of solvers for ODE's is given, with focus on fixed step solvers using the Runge-Kutta 4th order method [8].

An overview of the system and software, including descriptions of key concepts, sensors, data sets, background information and proposed control system design, is given in chapter 2. In chapters 6 and 7 the results of the 22 separate cases, each using different configurations of data sets and model structures to identify parameters, are presented and discussed. Cases are compared to illustrate interesting aspects in regards to using grey-box models for heating and cooling time estimation in control systems. The report is concluded in chapter 8 with a short summary.

2 System overview

In this chapter information about the developed system, together with relevant background, information is presented. The primary purpose of development of thermal building models is to create better temperature control systems, thereby saving energy and increasing thermal comfort for occupants (Appendix A). This type of control system is based on using models to predict heating and cooling times of a building. The chapter starts with an overview of the system and software, followed by a definition of heating and cooling times, as well as a proposed control algorithm. Next, an introduction to the concepts Thermal Networks, Single-zone models and grey-box modeling is given. Then follows a description of the data collection systems, test buildings, sensors, and data sets this project is based on.

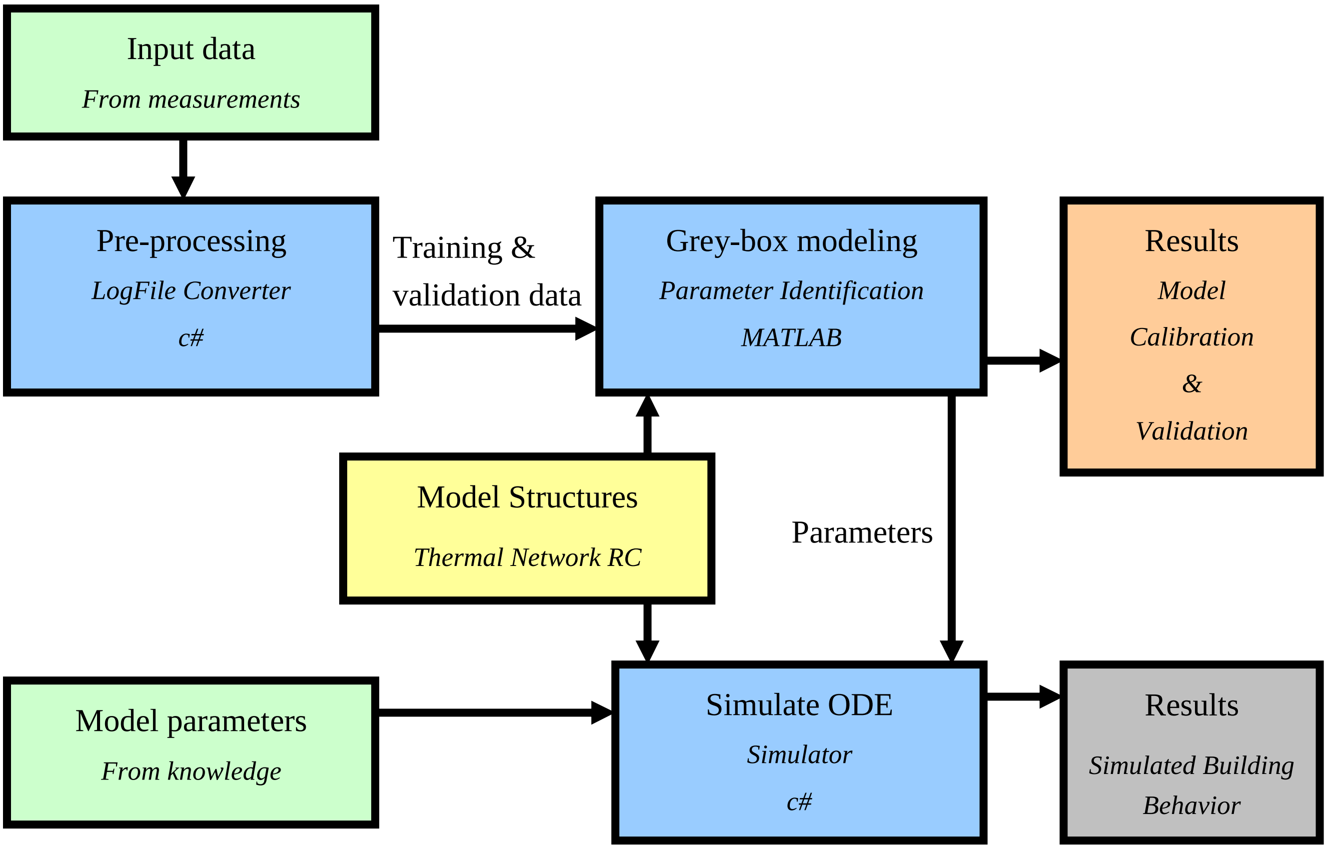


Figure 1 - System Overview

An overview of the system developed in this project is given in Figure 1. Three software tools have been created, shown with blue blocks. They are each responsible for separate processing tasks in the system. Each of these softwares will be discussed in details in a later chapter. The first tool is called “LogFile Converter” and is used to pre-process raw data files from different measurement systems in order to generate training and validation data for the modeling process. The second software tool is the “Simulation” software which can simulate a model given as a set of Ordinary Differential Equations (ODE). Both these softwares are developed in c#. The last software tool is developed in MATLAB. This software is used to identify the parameters of a model based on measurement data. It is also responsible for validating the model against independent test data. Heating and cooling times presented in the results are computed with MATLAB, but can also be found using the “Simulation” software.

The “Model Structures” block, shown in yellow, represents the models for a building. These models are a simplified structural representation of the thermodynamic behavior of buildings. Models are derived as Thermal Networks using Resistor-Capacitor (RC) circuit equivalent descriptions [5]. Both these concepts are described in this chapter and with more details in a later chapter on theory.

The inputs to the system are shown in the two green blocks. The first input is the measured raw data that will be pre-processed into usable training and validation data sets. This data comes from experiments performed on buildings using sensors to measure parameters of interest to the model. The second input block represents known building parameters. The “Simulation” software is not restricted to simulating only grey-box models with identified parameters, but can also use parameters computed by other means, e.g. expert knowledge.

The results of the system are shown in two blocks, grey for the results from the c# simulator and orange for the output from the MATLAB parameter identification software. The MATLAB results are statistics and plots generated from the parameter identification, and from the validation of the model based on independent data. These results are discussed in detail in later chapters. The results from the c# simulator are presented in grey because these are not extensively used in this report. The software is still important for the overall system. Results from c# are interesting in terms of testing the computational speed relative to MATLAB and for use in future control systems, where MATLAB is not available.

2.1 Prediction of heating/cooling times

The key attribute of interest is the heating and cooling times of a building. The reason for deriving building models is primarily improved temperature control, both for comfort of occupants, and for saving energy used for heating [1, 4]. The energy saving gains is realized by lowering the building temperature, and thus the energy required to maintain a set temperature when the building is not in use [1]. Such a control scheme requires good predictions of the time it takes to re-heat the building before use. Models must also predict the cooling time of a building, i.e. how long time before the building is expected to be unused it is acceptable to start cooling. The time it takes to heat or cool the building, is the primary feature of a buildings thermal behavior that this project aims to predict.

If the temperature can be accurately predicted, within a specific prediction horizon, the heating and cooling times can be computed based on simulations using a model. It is natural to think in terms of Model Predictive Control (MPC) [9] as a means of achieving improved temperature control of buildings. A simpler scheme, using only ON/OFF control of heaters and predictions of heating and cooling times, is here suggested as an alternative control scheme to maintain the required comfort temperature only when building is in use, thereby minimizing the energy consumption. One reason for insisting on a simple control scheme is to develop a system that can be installed in existing buildings. This will be discussed in a later chapter.

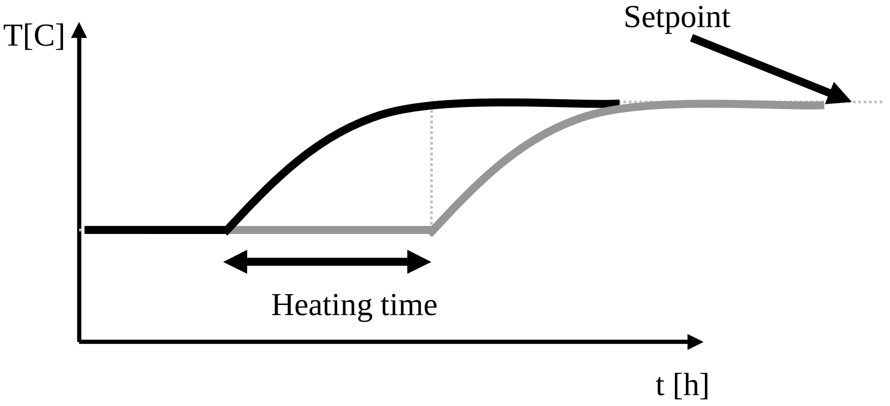


Figure 2 - Heating time and setpoint

In Figure 2 the heating time is defined from the time the heating system is turned on and the temperature starts to increase, until the temperature reaches its target setpoint. Before heating begins, the building is assumed maintained at some lower temperature setpoint. The heating time is defined as the time required for changing the temperature from a low to a high setting.

A control system can use a model to predict the results of control inputs ahead of time, i.e. a predictive controller, as shown by black line in Figure 2. Given an estimate of the model inputs such as weather conditions and building occupancy, it is possible to predict when the buildings heating system must be turned on or off. This allows the system to maintain an optimal setpoint of the buildings air temperature, both with regards to thermal comfort and energy economy. This is in contrast to conventional PID or ON/OFF control (grey line) which only responds to an error, a difference between measurand and setpoint.

As a part of this control scheme, improved control of the heaters it self can be realized. Using electric heaters, where each heater is controlled by an onboard thermostat, is in this project shown to be inefficient with respect to heating time. The thermostat regulates the temperature of the heater, not the building air temperature itself. The thermodynamics between air and heater is what determines the temperature in the building. This will be further discussed in a later chapter.

2.1.1 Definition of heating and cooling time

It is useful for this project to have a definition of heating and cooling time. Heating time is defined as the time from the heater is turned on, until the room temperature has reached its setpoint, i.e. changing from a low to a high temperature setpoint.

Similarly, cooling time is defined as the time for temperature to drop from high to low. For cooling time, the level the temperature drops too is important to specify. When predicting how long time before a building is expected to be unused the heating can be turned off, the temperate the building is cooled to will typically be only 1 degree below the high setpoint, as this is assumed acceptable to the occupants. However, if the time of interest is the time it takes for the building to reach a lower temperature setpoint, such as where the building will be maintained when not in use, the definition of cooling temperature will be different. The first definition is used in this project. The goal is to save energy by more efficient thermal control. Hence the interesting cooling time is the prediction of when heating can be turned off, thus saving energy, without causing uncomfortable temperature decrease in the building.

2.1.2 Control Scheme, local and supervisory control

A controller based on prediction of heating and cooling times will perform simulations on the building as part of a loop, at a specific loop time. In each loop, the controller will compute the time until next change in temperature is expected. If predictions of heating time indicate that the setpoint of the heating system should be changed, the controller will do so, attempting to minimize energy consumption by minimizing the time heating system is on a high temperature setpoint.

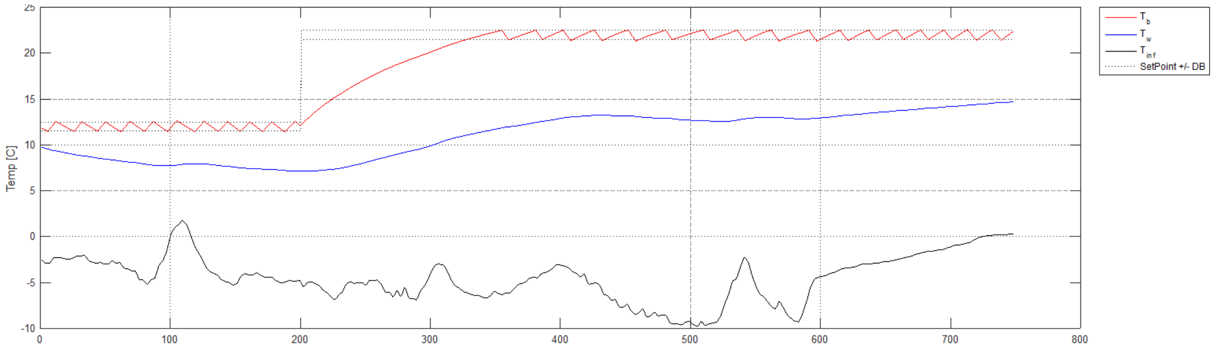


Figure 3 - ON/OFF controller example

The actual heater is typically controlled by a so called ON/OFF controller, as demonstrated by a simulation in Figure 3. This type of controller maintains a setpoint with a deadband, such that if the temperature is below setpoint minus the deadband, the heater is turned ON. If temperature is above setpoint plus deadband the heater is switched OFF. This is the typical way onboard thermostats on electrical heaters work. In the figure, the building temperature is plotted in red, and the setpoint with deadband with black dotted lines. As demonstrated, the temperature is maintained between these two lines.

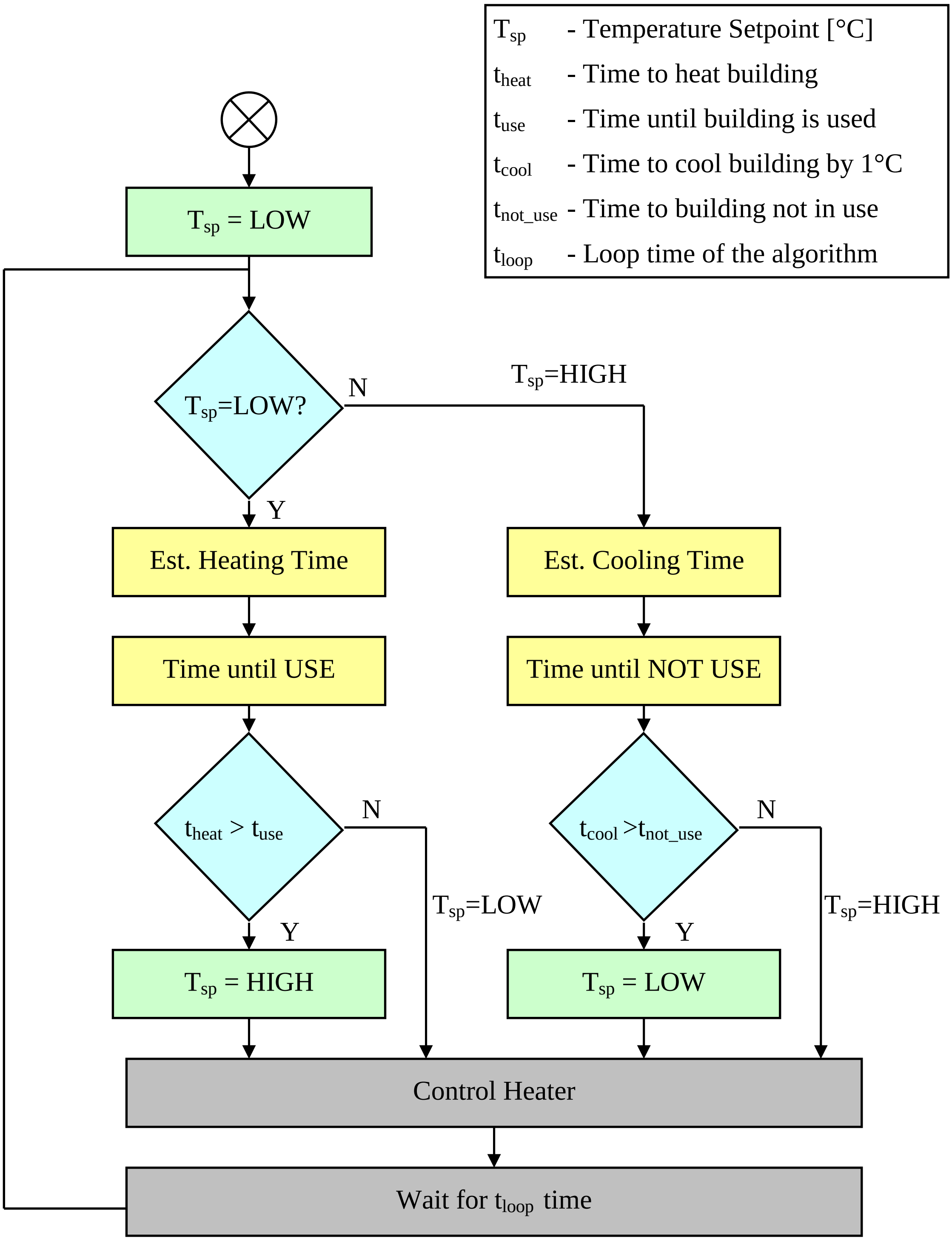

T_{sp} - Temperature Setpoint [°C]
t_{heat} - Time to heat building
t_{use} - Time until building is used
t_{cool} - Time to cool building by 1°C
t_{not_use} - Time to building not in use
t_{loop} - Loop time of the algorithm
T_{sp} = LOW
T_{sp}=LOW?
N
T_{sp}=HIGH
Y
Est. Heating Time
Est. Cooling Time
Time until USE
Time until NOT USE
$t_{heat} > t_{use}$
N
T_{sp}=LOW
Y
$t_{cool} > t_{not\_use}$
N
T_{sp}=HIGH
Y
T_{sp} = HIGH
T_{sp} = LOW
Control Heater
Wait for t_{loop} time

Figure 4 - Flow chart of predictive control scheme

In Figure 4 a flow chart of a suggested predictive control scheme is presented. Initially the building is assumed to be in the unused state and temperature setpoint T_{sp} is set to LOW. The actual values for HIGH and LOW setpoints will depend on the preferences of building occupants. Typically, HIGH will be a comfort temperature of around, say, 22°C, while LOW will be a temperature that is at least a reasonable amount above freezing. The LOW setpoint will likely also be affected by the capacity of the heating system, i.e. how large steps in temperature the heaters can deliver within a reasonable timeframe. It is an important note that even with predictive control the heating system must be able to supply enough heat to meet the requirements of the control system in order for the combined system to be efficient. The capacity of the heating system can limit how low the setpoint during the “unused” phase can be, since it limits the speed of re-heating.

The first condition in the algorithm is to check if the temperature should be in high or low state. If in low state, meaning building is currently not used, the time to re-heat building is estimated from the model (t_{heat}), together with a computed time (t_{use}) before the building is expected to be in use (yellow blocks).

Times for building use can be configured or learned based on machine learning [10], or similar methods. Determination of usage pattern is outside the scope of this project and here it is assumed that the usage pattern is known. If the time to heat is shorter then the time to building will be used, the temperature setpoint is changed to the HIGH state. Note that the loop times is assumed short enough that a slight delay of up to one loop time caused by this inequality is acceptable. Alternatively the condition could be reformed such that t_{heat}-t_{use} < t_{loop}, i.e. if the heater should be turned on before the next loop, it is turned on at present loop, to avoid delay.

With the new setpoint determined, the heater is controlled by this setpoint for time period of one looptime, before the algorithm is repeated.

The second branch of the flowchart for cooling is similar to heating. First the time the building takes to cool by, say, 1 degree below setpoint is estimated from a model (t_{cool}). Then the time before building is expected to be unused is computed (t_{not_use}). If the condition determines that it is acceptable to start letting the building cool down, e.g. that cooling by 1°C takes more then the remaining time until building is no longer used, the heater setpoint is set to LOW, and the system left to control the heater for one looptime.

Without models to predict these times, control systems must rely on “safe” setpoints. This typically means that temperature is dropped by 5 degrees while building is not in use, and that the drop in temperature is initiated after the building is no longer used. Additionally, the heating system will typically be engaged before required, such that the setpoint is achieved to soon. The cooling cycle can not be initiated before the building is known to be unused, without a model to predict how fast the temperature will be unacceptably low.

An advantage of the suggested control scheme in Figure 4 is that building temperature preference can be set simply by defining timeslots of high and low temperature. The control algorithm will use this standard settings information in a way that is better then typical control systems, by using model predictions to minimize the energy needed to maintain the temperature settings.

2.1.3 Infrastructure of building and control system

The “Control Heater” block in Figure 4 will be a “local” controller, where measurements of room temperature is used to determine if the power to the heater should be turned on or off to maintain the given setpoint. The predictive controller is used as a supervisory control, feeding the “local” ON/OFF control of the heater with a setpoint.

Note that “local” in this sense does not mean that the controller is physically inside the heater. Since commercial heaters typically does not have options for external control, a system is required that can control power to the heater. Commercial electric heaters with onboard thermostat control do not actually control room temperature but rather the surface temperature of the heater. This is not desirable in an advanced control system, as it may lead to the heater taking longer then predicted to heat a building. Thermostat control is sub-optimal with respect to heating time as later chapters will demonstrate.

For this type of control system to be applicable in buildings around the world, the system has to be easy to install and maintain. Requiring advanced heat sources with, say, WiFi [11] control signal inputs would be infeasible. A possible solution is to set the heater thermostat to max and plug in a simple remote controllable device between power outlet and heater. The addition of a control device allows the control system to cut power to the heater without interference from the thermostat. Such devices are typically found in so called “Smarthouse” [10] systems, and can even be part of the power outlet itself. Regardless of both new and existing technology, it is an important requirement in this project that a control system should not require any advanced infrastructure, but rather rely on existing power structure of the building. One possible addition is remote control devices that can be plugged in between heaters and wall outlets, without requiring electrical installation by professionals.

The control system is discussed as motivation and background for the project only. Development of controllers can be based on the suggestions, theory and results presented here, but their implementation is outside the scope of this project.

2.1.4 Realtime requirement

An important point to note in the previously discussed algorithm of chapter 2.1.2 is that the estimates, using models, have a real-time requirement, i.e. a requirement to finish the computations within a specific time [11]. The estimation must be completed within one loop-time. If the estimate takes longer then one loop, the result will not be available when the controller needs an updated setpoint, and the supervisory controller will fail.

Additionally, from the flow chart in Figure 4, the Control and Delay blocks, as discussed in chapter 2.1.3, would actually run in parallel with the supervisory control system, possibly as software objects [12] within the same computer or even software. Thus, time used in simulations to estimate heating and cooling time must not stop the controller from maintaining the present setpoint [11], while a new setpoint is computed by the supervisory control system. Since the supervisory and local control must run in parallel this further puts demands on the software implementation to ensure proper synchronization of shared recourses [11].

2.2 Thermal network RC models

The thermodynamic behavior of a building can be described by a so called "lumped parameter" model [4] in the form of an RC equivalent circuit. For a detailed discussion of Thermal Network Modeling, see [5]. The concept of a "thermal network" describes how heat energy can flow between elements of the building and its surroundings, modeled as nodes in a Resistor-Capacitor (RC) circuit.

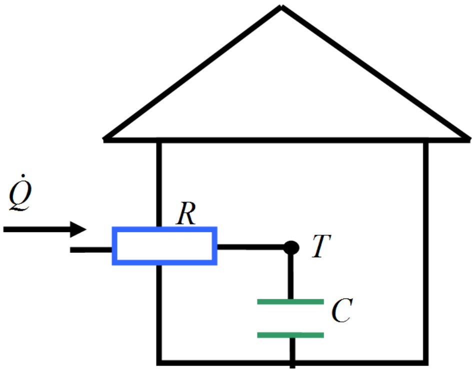


Figure 5 - Thermal model of a building

In Figure 5 a building with a simple RC model is depicted, showing how a resistor can be used as a model of the walls resistance to heat flow, while a capacitor models the buildings capacity to store thermal energy. The node market T here represents the interior of the building. The node is marked by a black dot, but for the remaining schematics in this report the dot will be omitted for simplicity.

Thermal behavior of a building is described by the flow of heat $\dot{Q}$ and the temperature T at specific points [5]. Heat flow, in the unit of Watt (W), can be induced by a heater, solar irradiation or building occupants and it can be driven by a temperature differential, in the unit of Kelvin (K) or Celsius (C). For a differential between two absolute temperatures the units K and C are interchangeable. An example is the differential between indoor and outdoor temperature which drives heat flow between building air and outdoor environment.

Relationship between temperature differential and heat flow, is determined by the thermal resistance that the temperature differential acts across [5], in the unit of Kelvin pr Watt.

Thermal energy can be stored in objects, such as furniture, walls and roof, as determined by the objects thermal capacity, in the unit of Jules pr Kelvin [5]. The amount of energy required to raise the temperature in an object depends on the thermal capacitance.

All of these thermal parameters can be described by electrical equivalents, and the building can then be modeled as a simple Resistor-Capacitor (RC) circuit [5], and analyzed using conventional circuit theory, e.g. Kirchhof’s Laws, potential dividers, Ohms Law and Laplace transformation for impedance computations [13].

Flow of heat is modeled as current in an electric circuit, where the driving potential is the temperature, modeled as voltage. Using this analogy, a resistor becomes thermal resistance, while thermal capacitance is modeled as electrical capacitance [5]. Modeling thermal flow in a building using the electrical circuit analogy has the advantage of being simple to analyze, particularly for those used to working with basic electrical circuits. The intuitive understanding gained from these simple model structures is important when working with grey-box models. Since no accurate physical model, i.e. white-box, is needed, the intuitive, or cognitive [5], derivation of an RC network allows models to be derived based on knowledge about the buildings thermal behavior, without use of complicated thermodynamic laws and equations.

2.3 Thermal zones and single-zone models

In modeling of thermal behavior in buildings, the term “thermal zone” is used to refer to a section of the building which is modeled [5]. The division of a building into thermal zones can be done in many ways, and the particular method used depends on knowledge of the building [5].

A thermal zone has typically a single state for the air temperature of the zone [1], while the envelope can be modeled by multiple states. In particular, walls are often divided into two or more layers, with a separate temperature state for each layer. Roof and floor is treated similarly, but often with a single state. In some models, particularly when Thermal Networks are used, the entire building envelope is modeled as one object with one state for the temperature of walls, roof and floor combined [2, 3].

Thermal zones can be combined to form a more complex model, where each zone is modeled separately, but the models are interconnected depending on the thermal behavior of the building. The term “single zone” is used to describe a model where the whole building is modeled as a single zone [1].

2.4 Grey-box modeling

The terms white-, black- and grey-box models all refer to a classification of models, based on the type of information used to derive the model. In white-box models, physical laws, such as balance laws, or first principles, are used [1], giving a model with a known structure and physical meaning . The term white-box is used in the sense that the internals of the models are fully described by equations. The opposite is a black-box model, where the model is entirely derived from empirical data, with no prescribed physical meaning to the structure. The term black-box is used in the sense that model is not describable, i.e. can not be “seen”.

A grey-box model is a compromise between these two model types. In a grey-box model, the structure is defined based on physical knowledge of the system. In the case of a building the thermal parameters of the building are modeled in a simplified form, often referred to as “lumped parameter” model [2, 3, 6, 7]. The term “lumped” reflects that the model combines several discrete parts of a building into one component. As an example, the thermal resistance of all windows and doors can be combined into a single thermal resistance[5, 6].

The parameters of this simplified model structure are identified based on measurement data. As such, grey-box models combines the strengths of both white-box and black-box models[1, 4]. The grey-box models themselves are not a combination of white and black-box models, but rather the combination of methods to derive them, is the reason for the “grey” term. They are more generally applicable then black-box models, since their internal structure is based on a general model of a building[4]. Further, like black-box models, their accuracy is better then general white-box models, since the grey-box model parameters are fitted to a specific building based on measurements [4].

Grey-box models are particularly interesting in the field of Building Automation Systems (BAS) and Heating Ventilation Air Conditioning (HVAC) modeling, since as reported in [1], a model based on thermodynamic investigation of a building can become complex, due to the large number of parameters that affects the behavior. Heat equations are in nature partial differential equations [1], which complicates the model further since a discretization method is required. Combining simplified model structures with data-driven parameter identification results in a simpler model then the white-box approach [2, 6].

To extract model parameters from a data set the model is simulated with some initial parameter values, and the result is compared to the known empirical reference measurements. Then, some method of optimal estimation is used to modify the parameters iteratively, such that the model will predict the known reference data with minimal error.

Grey-box models are the core of this project, and as such, the concept will be discussed in detail in the following chapters.

2.5 Data set: Sensor types and position

For measurements on buildings, there are several variables that need to be measured or estimated, sometimes for multiple locations. Firstly, the temperature is clearly of importance to all models of thermal behavior. Also the power consumption by any active heating system or other appliances that generate heat in the building requires measurements. Thirdly, the outdoor weather conditions, particularly the temperature, are important. Solar irradiation, or heat gain from solar energy, needs to be measured in some way. Alternatively, solar heat gain can be estimated [2] using e.g. a model and Kalman filter. Some models may also include relative humidity as an input [1]. Occupants of a building also generate heat and may be an important model input. Energy released by occupants may require estimation [6].

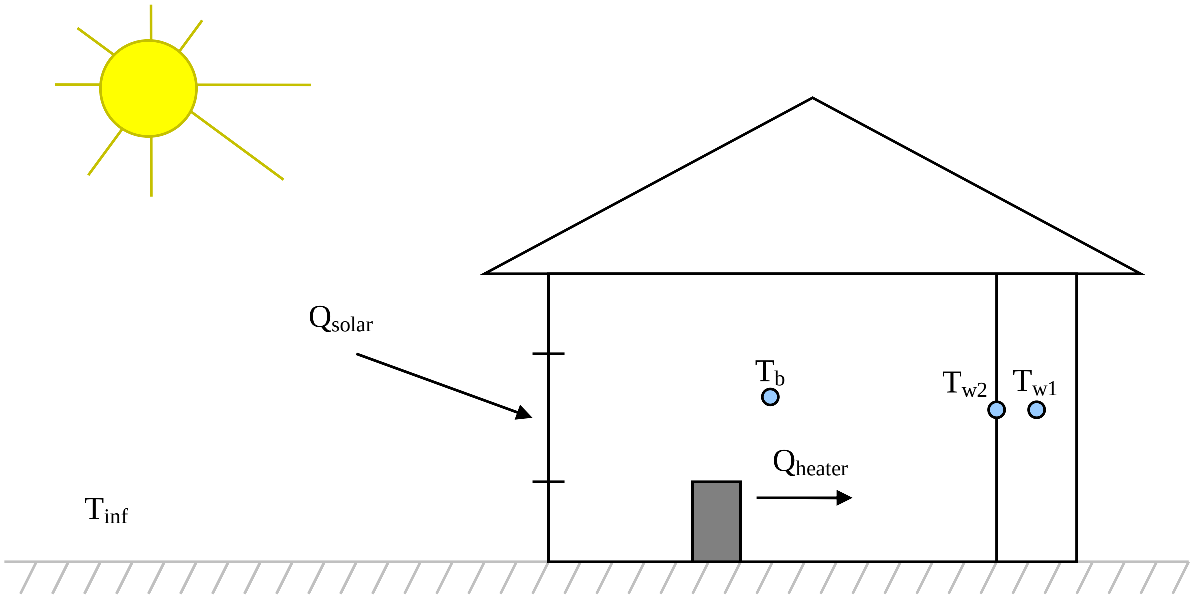


Figure 6 - Example of measured building parameters

In Figure 6, relevant parameters for this project are depicted. The main parameter of interest is the building temperature T_b. To help the models predict T_b, measurements of wall temperature T_w is used. In some models wall temperature is measured both on the surface of the wall inside the building, and in the insulation layer inside the wall. Where only one wall temperature is used, T_{w2} is called T_w. Supplied heat energy from the electric heating system in the buildings is measured by logging the power used in the building. Heat gain from solar irradiation is not measured in this project, and is treated as noise. It is however assumed relevant to the model behavior, and will be discussed further in this report. Buildings are empty during all experiments in this project, such that heat gains from building occupants can be omitted.

2.5.1 Description of test data

The data on which the modeling and simulation in this project is based, is taken from two separate buildings, named “ByggeLab” and “Cabin”.

The data from “ByggeLab” is recorded in an experimental building at Telemark University Collage in Porsgrunn, further described in [14]. The building was constructed partially as a test facility to experiment with building models, and has a high number of sensors installed in strategic locations throughout the building.

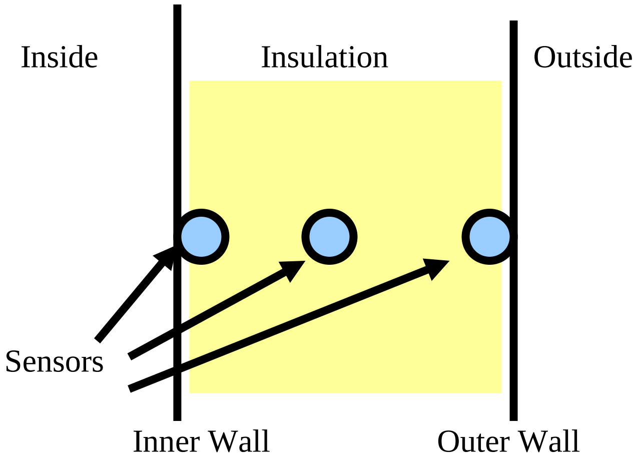


Figure 7 - Building wall with sensors

Temperature and humidity sensors are located in three places in the north, east and south walls, and two places for the west wall. The temperature sensors are of type TMP36, with an accuracy of +/-1°C [15] . The north wall has a door, with one additional sensor above the door. The first sensor is positioned close to the outer wooden cladding. The second sensor is located in the insulation layer, while the third sensor is located on the inside of the inner wall, as shown in Figure 7. Similarly sensors are also located in the roof. All sensors are connected to a custom logging system for the test facility [14].

The building contains an electric heater and a humidifier, together with a computer for logging. Power consumption for all devices is logged under the assumption that all energy is producing heat and energy for other uses is assumed neglitable. Additionally, a weather station is used to log both indoor temperature, and weather conditions such as relative humidity, wind speed, dew point, etc. Further details on the measurement system and sensors, as well as technical details of the building structure, is presented in [14]. The weather station lacks specification of accuracy, but resolution is specified to 0.2° Fahrenheit [16]. It is assumed that the accuracy is in similar range, or better, then the TMP36 sensors, which is typical accuracy for cheap, commercial, temperature sensors without specific accuracy requirements.

Only the temperatures on the north wall are used in this project, together with indoor and outdoor temperature from the weather station, and power consumption measurement. Other sensor readings are not relevant to the simple RC thermal network models presented here. Most of the data used in this project is taken from this building. The building is located in a shaded area, and has only a small window, such that the assumption of solar heat gains being treated as noise is likely good for this particular building.

The second building of interest is a cabin located in Hedmark near Sjusjøen. This building is also used as the basis for [17], which contain more information about the setup. The cabin contains a measurement system with temperature, humidity and one light sensor located in various locations around the building. Since this building was not constructed for experimental use, the sensor locations are not optimal, as they are for the “ByggeLab” building. The data for the cabin is also taken from a weather station, same as for “ByggeLab”. Additionally, in the measurement system, one temperature reading from inside the building floor is used to represent the “wall” temperature. Since the building enclosure is modeled as a single object [2], and not walls, roof and floor separately [17], this is assumed adequate. Power consumption for the entire cabin is read from a smart power meter, with an accuracy of more then 90% [17], which outputs average kWh energy consumption for each hour of the day. This can be directly used as average power in kW. All energy is assumed to go to heat production.

The cabin contains four windows, where two large windows face south [17], and is not surrounded in shade such as “ByggeLab”. Thus for the cabin data, treating solar heating as noise may be to simplistic. This will be further investigated in the following chapters. The cabin measurement setup contains a Light Dependant Resistor (LDR) that can give an approximate indication of the level of sunlight.

Only temperature and power consumption data is used for modeling in this project. Temperature is logged by two types of sensing systems. The sensor setup for the cabin is similar, using same type of weather station, sensors, DAQ and logging software as “ByggeLab”.

2.5.2 Amount of data needed

In all empirical modeling, an important question is what amount of data is required to produce an accurate model. In [2] a summary table of the amount of data used in four other published articles is given, listing the data used in parameter identification process between 4 and 60 days, with timesteps varying from 1 minute to 1 hour. This illustrates a high degree of uncertainty in the field about what amount of data is required. Data sets used in this project vary from around a thousand to twelve thousand samples, with sample times at two, six or ten minutes.

2.5.3 Range of data needed

Unlike much of empirical modeling where the experiment can be freely designed thermal behavior of buildings depends highly on weather conditions which are outside of the experiments control. The building may also be in use such that the usage pattern and the heating requirements of occupants limits the way in which experiments can be designed [6]. This further limits what data can be collected on a building.

The data sets used in this project are step response experiments, where the active heating system in the building is turned to a setpoint value or to a maximum, for a specific time, and then turned low or to a minimum setting, again for a specified time. High/low experiments simulate typical usage in occupied buildings, while running heater on min/max settings gives more extreme temperature variations in the data.

In data driven modeling, the variation of the inputs must cover the range of inputs on which the model is expected to work [2, 18]. A model can not reliably predict the results of any input conditions that were not present in the training data. Therefore the amount of data needed should be considered more in terms of what range of inputs are covered by the data, rather then in terms of time length. A data set of relatively short time containing large variation in temperature, both in terms of measured temperatures but also in terms of rate of change, is likely to give better predictions then a data set of significantly longer time, but with less variance in the data [2, 4]. The question of what amount of data is needed is investigated further in later chapters.

Since the grey-box modeling approach depends highly on the data used for parameter identification, it may be advisable to use several models, depending on seasons and weather conditions. Choice of models can be done using internal logic, e.g. Fuzzy Logic. Such techniques can improve results when sub-optimal training sets is the only available data. By limiting the models to use the same range of inputs as the training data used to produce them, and instead use multiple models depending on conditions, the predictions of the models will be improved.

3 Literature Review

In this chapter, the articles on which much of this project is based, will be reviewed and summarized separately. Some of the articles refer to modeling of Heating, Ventilation and Air Conditioning (HVAC) systems. Modeling of HVAC systems and Building Automation Systems (BAS) are overlapping terms, where the thermal properties of buildings are concerned.

3.1 Review of modeling methods for HVAC systems

In [4] the authors summarize many of the relevant modeling techniques that can be applied to the modeling of HVAC systems. A general introduction and definition of white and black box models is given, followed by a presentation of the grey-box model concept.

A key point in the article is the use of dynamic models, that is models with internal states, such that the predicted results of a model depends on what has happened in the past [4]. Typically, the slow changing elements, such as temperature and humidity, in a HVAC system are modeled using a dynamic model. Faster dynamics of the system can be modeled using static models, as an approximation [4]. By simplifying state equations of fast dynamics to an algebraic equation, the model is simpler to compute. The number of states is reduced, and the computation time can be significantly reduced by eliminating fast dynamics.

The authors further discuss data driven modeling techniques, such as Auto Regressive eXogenous (ARX) and Auto Regressive Moving Average eXogenous (ARMAX) models[4], Support Vector Machines and Artificial Neural Networks. There exist a large number of data driven modeling approaches, and many approaches can be combined to take advantage of their respective strengths [4]. Different methods use different model structures, but in contrast to grey-box models, the structures used are a feature of the modeling technique itself, not of the particular system [4]. Black-box models based on time series regression, are reported to show accurate predictions given that the conditions are similar to the training data [4]. Other black-box modeling techniques are briefly discussed, such as e.g. sub-space identification of state-space models (4SID) [4].

Further, the article describes the development of white-box models, first for a single zone of a building, and next for various typical HVAC elements [4]. Each model is derived from physical balance equations. Of particular interest to this project, is the development of a single zone model. This is similar to the work presented in [1], but giving a significantly less detailed model. Both the model presented in [4] and in [1] are based on “the heat balance method” [4]. An alternative to the heat balance method is the “thermal network” method, where the building is split into several temperature nodes, with associated paths between them of a given thermal resistance. Additionally, thermal capacitance of a node can be modeled [4].

The variations between published works on gray-box models are mainly the method used to identify the parameters. One possibility is the use of optimization algorithms, such as e.g. SQP (Sequential Quadratic Programming) [4]. Objective function and constraints is defined, and a standard optimization algorithm is used to find model parameters. Other alternatives include genetic algorithms [4]. The gray-box model review concludes that this class of models has better accuracy then physics based models, while also being more general than pure empirical models [4]. The drawback of grey-box models is the work required in developing them, since they demand both physical knowledge of the process and knowledge about empirical/statistical modeling [4].

An extensive list of performance criteria, on which models can be compared, are listed[4]. A commonly used criterion is the Root Mean Square Error (RMSE) [18], which can be used to quantify a models ability to predict a variable, relative to the known true value.

The authors conclude the review of data driven, physics based and grey-box with a summary of all the discussed modeling techniques, and the tools required to compare them.

3.2 Development and validation of a gray box model to predict thermal behavior of occupied office buildings

In [2], the focus is placed on the development of gray box models based on thermal network RC models, and comparing different types of models in order to choose which structure is best. Two important shortcomings in traditional black-box models are given. The ARX/ARMAX type models are reported to have problems with non-linear behavior of a system. Secondly, another empirical approach, Artificial Neural Networks, is known to have problems when used to predict systems under control strategies not present in the training data [2].

Four different model structures is investigated [2]. The first structure is the simplest, while the other structures can be seen as expanding the simpler ones, by adding more details, resulting in more parameters.

In modeling thermal behavior of buildings, solar irradiation is important. These are parameters that are considered difficult to measure [2]. The article suggests a model where the solar flux can be estimated based on geometry of building and its surroundings, together with time of day, location of building and cloud coverage data [2]. A simplified model for estimating solar flux on the building interior, through windows, and on the exterior walls is suggested. This model uses only the cloud coverage as an input [2].

The identification of parameters is performed using optimization based on the Interior Point algorithm [2]. This algorithm can handle non-linear objective functions and constraints of the identified parameters. The authors suggest constraints on identified parameters, using one third of the initial value as lower bound, and three times the initial value as upper bound [2]. Furthermore, the objective function is chosen as the product of the separate error sum of power consumption and indoor temperature [2]. Parameters are normalized, such that all parameters have the same weight in the objective function. A convergence criterion is stated to be total change of all parameters less then 10^{-10}.

The four stated models are compared, using simulated data[2]. Data is generated at the extreme cases of cold winter and hot summer. Results are found to be better on data from winter, respective of summer, due to increased heating demands.

Further, model fit is concluded to reach maximum value using two weeks of training data [2]. An interesting result is that a third order model is stated too complex for identification. Second order model gives better performance. The authors conclude that the R6C2 model is the best choice. A good fit, over 80%, is found during high and low outdoor temperature. However, some problems with model predictions are found mid-season with reported model fit less then 80%. The authors conclude redoing the parameter identification on a monthly basis improves results [2].

The final section of the article presents a Sobol analysis which is used to identify which parameters of the model affect the ability to predict temperature and energy consumption in the building. Different parameters may affect the two outcomes to varying degree.

In conclusion, the authors find that the R6C2 model is the best choice. Further, the sensitivity analysis is reported to show that the two-part objective function is required, due to variation of parameter significance for temperature and power consumption prediction [2].

3.3 EKF based self-adaptive thermal model for a passive house

In [6] the focus is also grey-box models, based on lumped parameter thermal network models in the form of RC models. The article is based on using Extended Kalman Filters (EKF) for identification of parameters. A motivation for the article is models usable in Model Predictive Control (MPC). Since the computational load of MPC is known to be large, efficient models is a requirement for successful implementation [6]. Another motivation is a method to deal with changes in building parameters over time, such as degradation of building insulation [6].

In most thermal network models, the area of focus is on a single thermal zone. Hence, multiple rooms or floors are combined into one zone. In [6] the reference temperature measurement is taken as the weighted sum of temperature measurements for all areas within the zone, where the relative volume of each room is used as a weight for its temperature measurement [6]. This method has some problems, as discussed in [6].

To make the models “self-adaptive” the employed method has to estimate the parameters while the model is online. To avoid the problem of averaging temperature for the zone a state estimator is suggested [6].

The key point in [6] is the use of an Extended Kalman Filter (EKF) as a state estimator. By using a so called “dual estimation” technique, the temperature state and the parameters are estimated together [6]. The authors note that the dual estimation problem becomes non-linear, even if the model itself is linear. The model of choice is an R1C1 in the case of [6]. The EKF is a well known method for state estimation in non-linear systems [6]. By augmenting the state equation with a parameter vector in addition to the state vector, both can be computed by the EKF [6].

An advantage of this method is that it can be applied to estimating other model variables, such as the heat flux input from occupants, which is know to be difficult to measure [6]. Disturbance heat gain from occupants is estimated by a second EKF, where the parameters from the first EKF is kept constant [6]. The thermal parameters of the building are estimated while the building is empty. Disturbances from occupants are neglected in the first EKF. The occupants heat gain disturbance is estimated continuously while building is in use [6].

3.4 Quality of grey-box models and identified parameters as a function of the accuracy of input and observation signals

In [3] the influence of several different sources of measurement noise is investigated. Grey-box models are identified based on simulated data from a detailed building model, using IDEAS library in Modelica [3]. The use of simulated data in identification of grey-box models is useful when the objective is to experiment on the identification process, since the simulated data source allows generation of training and validation data with controlled noise parameters. The purpose of the investigation is to test what affects the quality of grey-box model predictions, with respect to use of these models in Model Predictive Control, where the “one day ahead” predictions are of particular interest to the controller [3].

Grey-box models are identified based on state-space form equations, derived from a thermal network, RC-equivalent, model of up to fourth order. Since the model structure is derived from physical understanding of the building thermodynamics, any identified parameters in the model will have a specific physical meaning [3].

In the first part of [3] the focus is on input parameters commonly used in building models that are hard to obtain accurately by measurements, specifically solar heat gains. Next the article tests the effect of white and colored noise on the identified models, and the robustness of the parameter identification algorithm. White noise models measurement uncertainties from sensors, while colored or biased noise may reflect systematic errors, such as a temperature sensor placed close to a heat source [3].

The solar, heater and occupancy heat gains are all distributed across all of the thermal capacitances in the models, based on distribution coefficients, rather then locating each source specifically at one point in the thermal network [3].

Effective solar gains is found using global horizontal irradiance, which can be measured and is often available from local weather stations [3]. Additionally, total irradiance in the vertical plane, along the four cardinal directions, is used as input signals to the models [3]. According to [3] this is a feasible way of including solar gains estimation to the model, without needing to measure the solar heat gains directly.

In conclusion, the authors of [3] finds that while both white and colored noise give an increase in uncertainty for estimated values. There are no significant changes in the estimated physical properties caused by white noise, in contrast to biased noise which gives unreliable predictions [3].

Finally, total irradiation on the vertical plane, together with the domestic electricity demand taken from a smart power meter on the electrical supply of the building, is shown to be a good estimate replacing the need for measurements of the effective solar and internal heat gains [3].

3.5 Gray-box modeling and validation of residential HVAC system for control system design

In [7] the focus is on development of grey-box models for separate parts of a typical HVAC system. The article describes how to derive physical models, based on first principle balance laws, for several typical HVAC subsystems. For each subsystem, general state-space models with unknown parameters are derived, as grey-box models. The parameters are identified from measurement data, using an optimization algorithm.

With completed grey-box models for each subsystem, the units are combined into a complete HVAC system and simulated together [7]. Simulink is used to simulate the final full system model.

As with all optimization algorithms, the initial conditions, which serves as a starting point for the optimization process, has a strong influence on the results, and the time to convergence for the algorithm [7]. Since the optimizer only finds local minima, a starting value for all parameters that is close to the true values is advisable [7].

Splitting a complicated system into subsystems, where each subsystem can be identified from measurements taken specifically for that task, is an interesting approach, which may be applicable to other segmentation of a building beyond the HVAC system.

4 Theory

In this chapter, a theoretical background of the project is given. This chapter explains the theory used in the project, as well as some supporting theory for improved understanding of the key concepts. The nature of the data from thermal behavior modeling is discussed, as well as applicable methods to pre-process and model such data. In particular, the concept of grey-box models and thermal networks is discussed in detail, since they are the basis of the results in this project. Further, the principles of optimization based parameter identification and simulation of Ordinary Differential Equations (ODE) is discussed.

4.1 Time dependence in data driven modeling

In the study of thermal behavior, the system responses are clearly time dependant. Since the system is dynamic in nature, Ordinary Differential Equations (ODE) are required to describe the behavior. By discretizing the time derivative, say using Forward Euler [8]:

$$\frac{df}{dt} = \frac{f(t_{k+1}) - f(t_k)}{\Delta t} \rightarrow f(t_{k+1}) = f(t_k) + \Delta t \frac{df}{dt}$$

From this equation it is clear that the value of a derivative at time t_{k+1} depend on the value at time t_k, which shows how samples at consecutive time instants are not independent samples, but depend on each other.

Because of this time dependence, any modeling method applied must be able to support and use the time depencany in the data. A method that requires or assumes samples are independent of each other may not be advisable to use when studying thermal dynamics.

4.2 Data driven modeling

Data driven modeling, also called empirical modeling, includes a large number of methods and techniques that can be used for finding a relationship between data. In data driven modeling the relationship between input and output data of a system or process is studied [18], whiteout any assumption on the specific structure of the system, i.e. black-box modeling. Some modeling methods, such as polynomial models [19], assume a generic structure where the output of the system is seen as a function of previous system inputs and outputs.

Other methods use co-variance between variables in the data-set to find a model which describes the structure in the data, or a relationship between input data and reference data [18]. This type of models is called multivariate regression models [18] and include methods such as Partial Least Squares Regression (PLS-R) and Principle Component Regression

(PCR). While there have been reported research that show some success in the application of regression models to study the thermal behavior of buildings [17], specifically the energy requirements for heating, the use of polynomial models, or time series regression models, seems to be a better choice [4]. In [4] a comprehensive summary of different modeling approaches to black box modeling used in HVAC systems is given. It is interesting to note that multivariate regression methods, such as PLS-R and PCR is not discussed there [4].

Since thermodynamic behavior of buildings requires a study of time varying effects, i.e. dynamic effects, rather then studying steady state conditions, time series regression models [4], on time discrete polynomial form, will be the focus of the remainder of this chapter. This type of model is ideally suited to describe the dynamic time-dependant nature of thermal behavior of buildings [4].

4.2.1 Time series regression models

Time series regression models can be expressed as a polynomial [19, 20] of past inputs, outputs and estimation errors. The errors are assumed to be white noise, i.e. random, for a good model. Since they consist of inputs and outputs, but not system states, this type of models is sometimes called “input-output models”. State-space time discrete models can, in the linear case, be written as input-output models by state elimination.

There exist many variations and formulations of this type of models. The common factor, or base principle, is using an equation of past inputs and outputs, together with an error term, to formulate a polynomial with coefficients for each of the variables in the model [19]. The coefficients are the model parameters that have to be fitted to a system. When the error term is white noise, the model fit is said to be optimal, in the sense that it is the best possible description of the data [19, 20]. A detailed discussion on so called “Prediction Error Methods” which are concerned with identifying models by regression modeling of time series data is found in [19].

$$a(q^{-1})y_k = \sum_{i=1}^{P} b_i(q^{-1})u_k^i + c(q^{-1})e_k$$

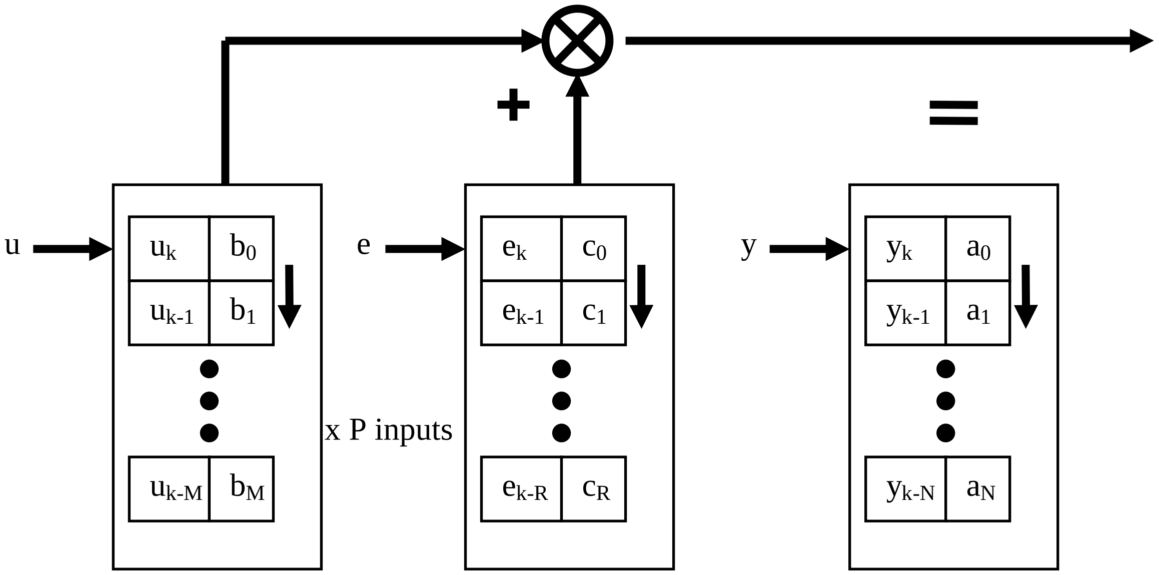


Figure 8 - Input-output model structure

One formulation of a time series regression model is given in Figure 8, both as an equation and corresponding block diagram. This model contains three functions of the time shift operator q^{-1} [19]. These functions a, b and c contain the coefficients that form the model. The function b may exist in a multiple of P if there is more then one input variable u. If all coefficient functions a, b and c are non-zero, the model is said to be an ARMAX model, which stands for Auto Regressive Moving Average eXogenous. Being “Auto Regressive” means that the current output is related to previous outputs as in an Infinite Impulse Response (IIR) filter [21, 22]. The term “eXogenous” defines the model to have external inputs, and “Moving Average” refers to the type of noise filtering that is applied in the model.

Another type of model, where the error term is neglected, i.e. $c(q^{-1}) = 0$, is known as ARX, since the error model is not present, i.e. not “Moving Average”. There exist other types of special configurations of the polynomial models as well. If c = 0 and $a(q^{-1})$ = const, the model becomes a Finite Impulse Response (FIR) filter, where the output y_k at present time depend only on past inputs [19, 21, 22]. If $a(q^{-1}) = c(q^{-1})$, the whole model can be divided by this polynomial in q, forming a denominator for the inputs, and simultaneously stating that the output at time k depends only on the error at time k and not on errors at past time instants. This configuration is called an “Output Error” model [19].

All of these models can be defined as polynomials in q^{-1}, where q is called a “time shift” or “lag” operator [20]. This operator shifts the variable stated at time k by one timestep, such that $q^{-1}*y_k = y_{k-1}$. Similarly, for powers of -n, the variable is shifted n steps backwards in discrete time [19, 21, 22]. This allows the models to be defined on compact form, with the coefficient polynomials describing the structure of the model. This structure is of a general form. The modeling choice, in addition to deciding the model structure, i.e. should any polynomial be set to 0, or should e.g. a = c to give an output error model [19], is the number of coefficients in each polynomial. The number of coefficients equals the polynomial order.

The coefficients are identified from measurement data, and as such the models are black-box models. Even if they do rely on a defined structure by equations that are chosen partially based on system knowledge, the equations are generic and not specific to the system that is modeled. To identify coefficients, Least Squares Regression [21, 22] technique is used, such that the coefficients are optimal, in the defined sense that error terms become white noise.

Time series regression models, Prediction Error Methods, or black-box models in general is not used in the present project, and is included simply for comparison to the grey-box approach. Since thermal behavior of a building, as stated in chapter 4.1, is an example of time series data due to the reliance on ODE's [8], the time series regression polynomial method is used to model HVAC systems in literature [4].

4.2.2 Overfitting

There are multiple challenges when working with data driven modeling [18], which if treated incorrectly can render the model invalid or wrong. One typical concern is "overfitting" [3], meaning deriving a model that describes only the training data, to such a level of detail e.g. noise, that the general applicability of the model to predict the actual system is destroyed [18]. If the model has a high number of parameters to identify, the risk of overfitting increases. This is a well known problem of for example Artificial Neural Networks [2] which can have a high number of free coefficients, allowing them to adapt to any pattern in the data. Even if the model can describe the training data with high accuracy, the model could be describing noise or structure in the data that is not relevant to the system and is thus unique to the particular training data. In such cases, the model will not give reliable prediction of any data set independent of the training data. This type of error is called overfitting.

4.3 Grey box modeling principles

Modeling methods can be divided into three classes of models. In [4] the authors define white-box models as being based in physical principles and black-box models as models where the systems behavior is investigated by modeling inputs effect on outputs. The third class of modeling methods, which is called grey-box models, is defined as models where the basic structure of the model is derived from physical equations or assumptions, and the parameters of that structure is identified using estimation algorithms on measurement data of the system to model.

The advantage of grey-box models is that, similar to white-box models, they are quite generic in nature. The same model can be applicable to multiple similar systems. Also, like black-box models, they can be made to fit a particular system, and thus be more accurate. For prediction of thermal behavior of buildings this is a great advantage, since all buildings share a similarity of structure, e.g. all buildings have walls, windows/doors and a roof. However, their parameters can vary due to e.g. different construction materials such as stone, brick or wood which have different thermal parameters.

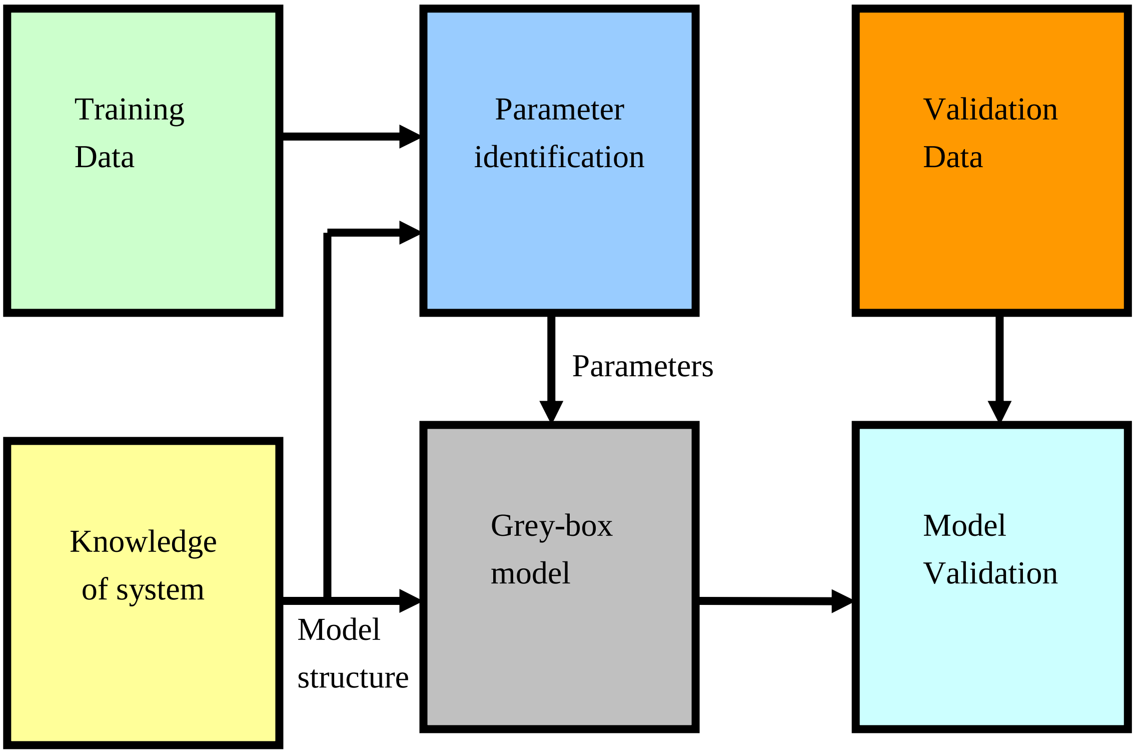


Figure 9 - Grey-box modeling process overview

The first step in deriving a gray box model, as depicted in Figure 9, is to find a set of equations that describe the system (grey box) in a general way [4]. These equations can be derived using a similar method as for white box models, using first principle balance laws [7], or they can be derived using a cognitive approach [5] based on knowledge of the system (yellow block), such as for the thermal network models used here.

Since grey-box model parameters is not necessarily derived from physical equations, i.e. unlike white-box parameters where all parameters are related to a known physical property of the system, parameters does not necessarily have a direct physical interpretation. They are however usually still interpretable in some sense. As an example, a thermal resistance for the whole building envelope has a physical meaning, but it can not be easily computed based on building code standards. In thermal network modeling the true physical parameters of a building is “lumped” into a few parameters of the RC equivalent.

The second step is to identify, or estimate, the parameters (blue block) of the chosen model structure. The estimation of model parameters can be done offline on historical data (green block), typically using some type of optimization algorithm like “Interior Point” [2] or “Sequential Quadratic Programming” [4] or it can use an estimator like a Kalman Filter [6]. Common to all approaches is the extraction of parameters from measured data, or data from a simulated, more complex, building model.

This procedure introduces an element of self learning into the modeling. The models can “learn” from a building how it behaves over time. This has an added advantage that any anomalies in behavior of the building can be identified as an “outlier” [18], i.e. data that does not fit the general pattern of the thermal behavior. An example use of this could be to let the model alert the occupants if a door or window has been left open, or if the insulation of wall or window is degraded, as this would change the thermal parameters of the building significantly. Further, the self learning capabilities of such a model allow the system to adapt to changes and also to some degree capture unmodeled effects in the system.

The third step in the process is to validate the model (cyan block) using independent validation data (orange block), as further discussed in the following sub-chapter.

The disadvantage of grey-box modeling is that it is more complicated to develop the models [4]. In addition to an understanding of the physical behavior of the system required to derive the structure equations, knowledge of data driven modeling is needed [18]. This is previously discussed in chapter 4.2.

4.3.1 Model calibration and validation

In all data-driven modeling methods, it is vital to have a good strategy for validation of the model [18]. To validate a model, means to use the model to predict results for a set of data with known reference values, where the data in the validation set is indepenant of the data used to derive the model. Using independent test data for the validation is considered the best practice [18]. An independent test set will test the model on its ability to predict results on new data thus detecting if the model was incorrectly identifying patterns in the calibration data, such as overfitting [18]. The concerns regarding overfitting and independent test data is of particular importance to black-box modeling, but equally so to grey-box modeling [3], since it relies on data driven modeling principles. This illustrates the need for grey-box modeling to use knowledge of both white- and black-box modeling methods.

4.4 Thermal network models

In [5] the process of modeling thermal behavior by a network of lumped parameter objects is described as a “cognitive process”. The model is derived by understanding of the system rather then first principle physical laws. The building is modeled as a network of interconnected parts, where temperature in each part drives the flow of heat between each object, or node, in the circuit [5]. This technique is intuitive by nature, rather then based in physical laws, but still yields a dynamic model in the form of ordinary differential equations, once the thermal network circuit is analyzed.

In a Resistor-Capacitor (RC) equivalent model, the thermal behavior is described as an equivalent electric circuit. Thermal resistance is modeled as electrical resistance, and thermal capacitance is modeled as electric capacitance. It follows that the voltage potential in a circuit is equivalent to temperature differential, and electric current is equivalent to heat flow.

By modeling thermal behavior as an electric circuit, the system can be analyzed by standard circuit theory, such as Kirchhof's laws [13, 23], which say that the flow into a node in the circuit must equal the flow out. At each node in an RC circuit, a potential is defined as a variable. Since the capacitors determine the dynamic behavior, the potential over each capacitor forms the dynamic states of the system. Hence, the system order is equal to the number of capacitors. Since RC circuits are simple in nature, they can be easily analyzed and configured as simplified models of a system, based on an intuitive understanding of a buildings thermal behavior, or by inspection of the dynamics in a data set.

Circuits may also contain potential or flow sources, representing a driving temperature or heat flow source. Outdoor temperature, which is assumed independent of the buildings thermal states, is an example of a driving potential source, while an electric heater, or solar irradiation heat gains, are examples of heat flow sources.

In this project RC circuit is used to refer to any circuit, electrical or thermal, that consists only of resistive and capacitive elements.

4.4.1 Resistor and capacitor elements

A resistor limits the flow of heat. As a consequence it has a temperature drop across it. A capacitor can store energy, where the temperature across the capacitor depends on its energy charge.

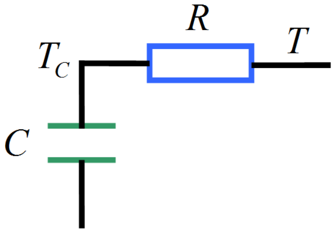


Figure 10 - RC Circuit

As shown in Figure 10, a resistor and capacitor can be connected to form a circuit. This circuit models a first order dynamic response to a step in temperature T. The state in the system is the temperature T_C across the capacitor C.

4.4.2 Ventilation as thermal resistance

Ventilation in a building will typically introduce a heat loss as heat from the building is released to the outside environment when warm air is ventilated out [1]. This effect can be modeled simply in an RC equivalent model as a resistor [2] that varies with the volume of air that the ventilation is replacing per time unit (m^3/h).

This ventilation model requires a parameter that can be tuned, such that the computed resistance accurately represents the loss of heat through the ventilation system.

In this project the model for a ventilation system is given as:

$$R_v = \frac{1}{G_{vent} \cdot N}$$

N is the ventilation setting in cubic meter pr hour, and G_{vent} is a tuning parameter, i.e. the thermal conductance of the ventilation system when N = 1 m^3/h. Conductance is defined as the inverse of resistance, such that the tuning parameter can be defined as a resistance where:

$$G_{vent} = \frac{1}{R_{vent}}$$

Special attention must be given to the case of N = 0, i.e. ventilation is switched off, to avoid division by zero in the model. When the ventilation is off, there is no heat loss through the ventilation, and the thermal resistance is thus infinity. Finally, the resistive model for ventilation is found as:

$$R_v = \begin{cases} \dfrac{1}{G_{vent} \cdot N} & N \neq 0 \\ \infty & N = 0 \end{cases}$$

This model will be used in this project where ventilation is required. For the purpose of parameter identification, if N = 0, the optimizer algorithms will find that changing R_{vent} has no effect on the resulting objective score and simply leave the parameter at some value that does not violate constraints. When N is zero, R_{vent} has no affect on results.

4.4.3 Potential and flow in RC equivalent

Potential in a thermal circuit is described by the temperature across objects, or at a specific object node. Flow in the circuit is the heat flow between nodes.

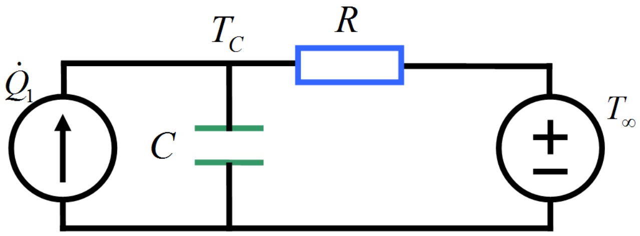


Figure 11 - RC circuit with potential and flow source

In Figure 11 the simple RC circuit is expanded with added temperature and heat flow sources. These sources are used to model energy sources that affect the building, such as temperatures not affected by building dynamics, i.e. outdoor or ground temperatures (potential source), or heat sources such as electric heaters, solar irradiation or building occupants (flow source).

4.4.4 Lumped parameter of buildings

Since the RC equivalent thermal network circuit models are a type of “lumped parameter” model [4] it follows that their parameters, as represented by the resistors and capacitors in the circuit, does not correspond directly to a single physical part of the building. Each element may model several structural parts of a building. As an example, all outer walls are typically modeled as a single resistance, while any heat loss directly from indoor to outdoor temperature, such as through windows and doors, is modeled by a separate resistor. Similarly, the energy storage capacity of all walls is modeled as a single capacitance. This example illustrates how the thermal behavior of the building, by a cognitive, i.e. not based on physics equations, analysis determines the structure of the RC thermal network. The physical structure of the building itself could conceivably be used to model each wall separately, including any windows and doors, but from a thermal behavior viewpoint the above approach is more meaningful [6].

4.4.5 Thermal network model structures

A total of five model structures are used in this project. Four of them are taken from [2] while the fifth one (R5C3) is a modified version of the R4C2 model, where the modification is based on inspection of data from simulations on the single zone white-box model presented in [1].

4.4.5.1 R4C2

The first structure, R4C2, contains four thermal resistances and two capacitances.

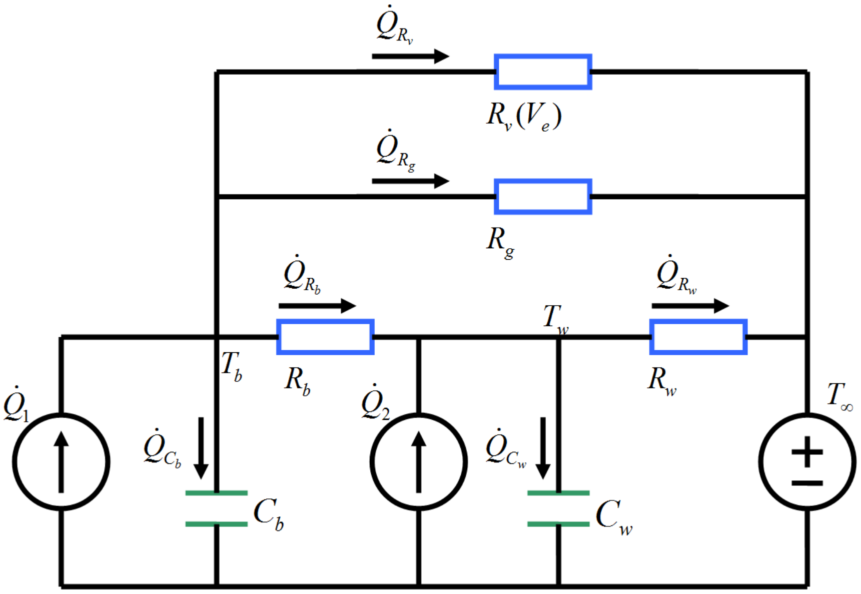


Figure 12 - R4C2 model

Figure 12 shows a schematic of the R4C2 model which gives the state equations for the model as:

$$\frac{dT_b}{dt}=\frac{1}{C_b}\dot{Q}_1-\frac{1}{C_bR_b}(T_b-T_w)-\frac{1}{C_bR_g}(T_b-T_\infty)-\frac{1}{C_bR_v(V_e)}(T_b-T_\infty)$$

$$\frac{dT_w}{dt}=\frac{1}{C_w}Q_2-\frac{1}{C_wR_b}(T_w-T_b)-\frac{1}{C_wR_w}(T_w-T_\infty)$$

One capacitance is used to represent the energy storage capability of the air inside the building (C_b), together with any furniture, and other contents of the building volume. The second capacitance(C_w) represents the energy storing capacity of the walls of the building[2]. One thermal resistance is used to represent the convective resistance of the building volume (R_b), between the temperature of the room and the adjacent wall temperature. This temperature can be considered as the boundary temperature of the inside wall [6]. Another resistance represents the combined thermal resistance of the wall itself (R_w). This contains the resistance of both construction material and insulation of the wall.

The potential, temperature in the case of thermal network models, across the wall resistor is taken as the temperature difference between the inside building and outdoor temperature. One resistance is taken directly between the building temperature and the outside (R_g), bypassing the walls. This resistance represents any heat flow through other elements of the building, such as windows, doors, etc. The forth and final resistance (R_v) is used to model the ventilation system, as discussed in chapter 4.4.2, and is a variable resistance, given as a function of the ventilation [2].

Three heat sources are included in the model. The first is a heat flow source, a combination of internal heat gains ($\dot{Q}_1$), such as active heating from HVAC or general heating, such as electric, oil, wood and other sources, combined with heating from appliances and a portion of the heat generated by occupants of the building. The second source of heat acts on the wall node, and consist of solar energy heating up the walls ($\dot{Q}_2$), combined with the remaining heat generated by occupants [2]. Finally, a third source of energy is modeled as a constant potential source (T_∞), which is the outdoor temperature. This configuration signifies that the outdoor temperature is independent of the model.

4.4.5.2 R6C2

The second model, R6C2, expands on the first, by adding two more resistances and a third heat flow source. This is done to better take into account solar heating gains in the model [2].

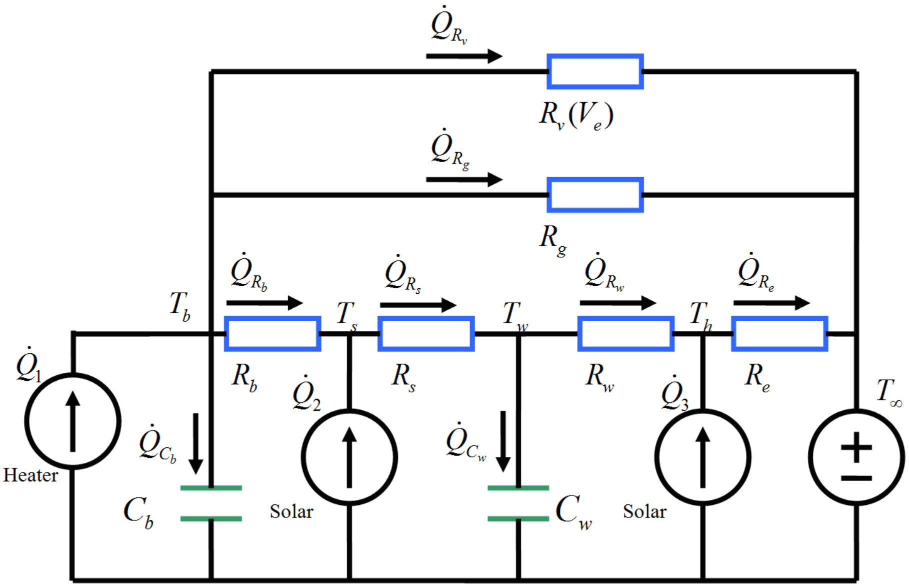


Figure 13 - R6C2 model

Figure 13 shows a schematic of the R6C2 model which has the state and output equations:

$$\frac{dT_b}{dt} = \frac{1}{C_b}\dot{Q}_1 - \frac{1}{C_bR_b}(T_b - T_s) - \frac{1}{C_bR_g}(T_b - T_\infty) - \frac{1}{C_bR_v(V_e)}(T_b - T_\infty)$$

$$\frac{dT_w}{dt} = -\frac{1}{C_wR_s}(T_w - T_s) - \frac{1}{C_wR_w}(T_w - T_h)$$

$$T_s = \frac{R_bR_s\dot{Q}_2 + R_bT_w + R_sT_b}{R_b + R_s}$$

$$T_h = \frac{R_eR_w\dot{Q}_3 + R_eT_w + R_wT_\infty}{R_e + R_w}$$

The first additional resistance (R_s) is added between the rooms convective resistance and the wall temperature [6]. Additionally the solar heat energy ($\dot{Q}_1$) now acts on the wall temperature through this new resistance. This helps to model the different effect solar irradiation has on light furniture compared to heavy walls [2]. The second change is the addition of another thermal resistance from the wall temperature to the environment (R$_e$). This allows the model to show how a part of the incoming solar energy is discharged directly to the outdoor environment, while a part of the energy is absorbed by the wall thermal capacitance, causing an increase in temperature[2]. External solar energy is neglected in the first model, but is included here as a heat source that acts on the outside of the wall ($\dot{Q}_3$).

4.4.5.3 R6C3

The third model, R6C3, adds one more thermal capacitance to model the heat energy storage in furniture of the building, and is an expansion of the second model[2].

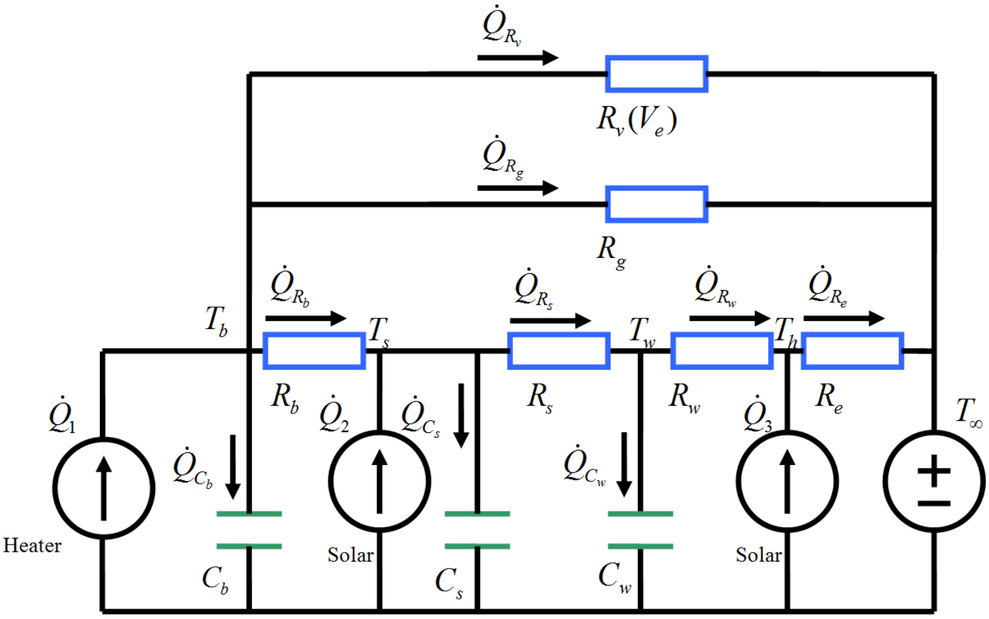


Figure 14 - R6C3 model

As shown in Figure 14, there is now a capacitor (C$_s$) connected to the T$_s$ node. The state and output equations are:

$$\frac{dT_b}{dt}=\frac{1}{C_b}\dot{Q}_1-\frac{1}{C_bR_b}(T_b-T_s)-\frac{1}{C_bR_g}(T_b-T_\infty)-\frac{1}{C_bR_v(V_e)}(T_b-T_\infty)$$

$$\frac{dT_w}{dt}=-\frac{1}{C_wR_s}(T_w-T_s)-\frac{1}{C_wR_w}(T_w-T_h)$$

$$\frac{dT_s}{dt}=\frac{1}{C_s}\dot{Q}_2-\frac{1}{C_sR_b}(T_s-T_b)-\frac{1}{C_sR_s}(T_s-T_w)$$

$$T_h=\frac{R_eR_w\dot{Q}_3+R_eT_w+R_wT_\infty}{R_e+R_w}$$

4.4.5.4 R7C3

The fourth structure, R7C3, is also an expansion on the R6C2 model.

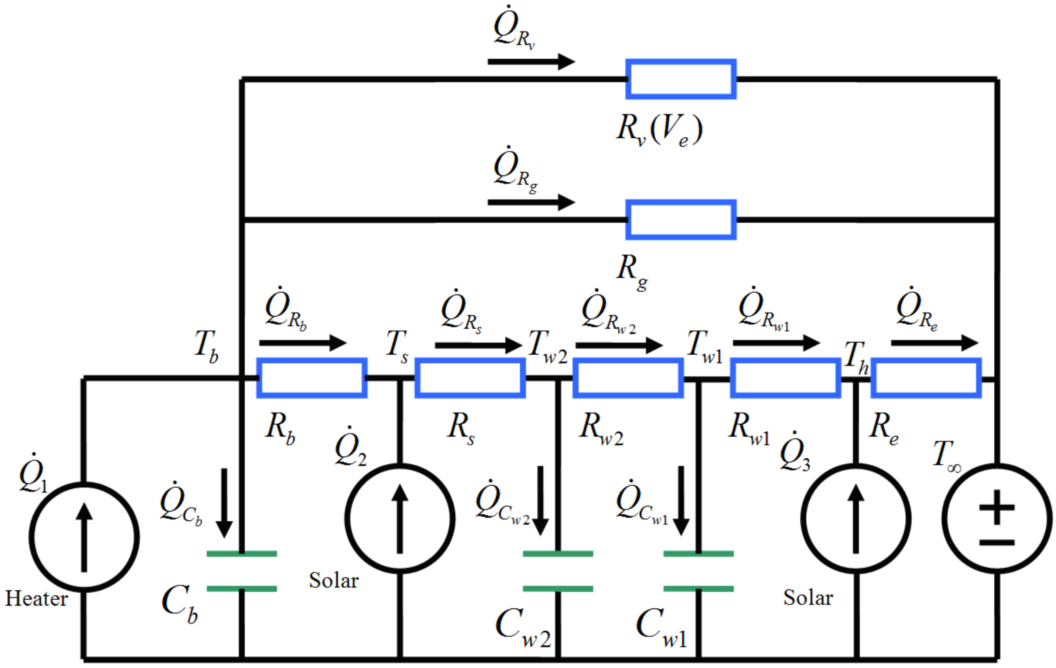


Figure 15 - R7C3 model

In the fourth model is shown in Figure 15. State and output equations are found to be:

$$\frac{dT_b}{dt}=\frac{1}{C_b}\dot{Q}_1-\frac{1}{C_bR_b}(T_b-T_s)-\frac{1}{C_bR_g}(T_b-T_\infty)-\frac{1}{C_bR_v(V_e)}(T_b-T_\infty)$$

$$\frac{dT_{w1}}{dt}=-\frac{1}{C_{w1}R_{w2}}(T_{w1}-T_{w2})-\frac{1}{C_{w1}R_{w1}}(T_{w1}-T_h)$$

$$\frac{dT_{w2}}{dt}=-\frac{1}{C_{w2}R_s}(T_{w2}-T_s)-\frac{1}{C_{w2}R_{w2}}(T_{w2}-T_{w1})$$

$$T_s=\frac{R_bR_s\dot{Q}_2+R_bT_{w2}+R_sT_b}{R_b+R_s}$$

$$T_h=\frac{R_eR_{w1}\dot{Q}_3+R_eT_{w1}+R_{w1}T_\infty}{R_e+R_{w1}}$$

The wall capacitance is divided in two parts (C_{w1} and C_{w2}), with a thermal resistance (R_{w1}, also R_w is now R_{w2}) modeling the transfer of heat between the two capacitances [2]. This results in a slightly more complex version of the R6C2 model, with the addition of another temperature node inside the wall.

4.4.5.5 R5C3

In the last model, a modification is made to the R4C2 model, in an attempt to find a model that will better fit with the single zone model presented in [1]. As the authors note in the article, there is a dynamic in this model that is significantly slower then the rest. This is shown clearly in the step response of the model. After the initial step change in states, there is a slow incline, rather then a straight steady state, meaning that there is a dynamic that takes a long time to reach its steady value. The assumption is that this is caused by slow release of heat energy from furniture [1].

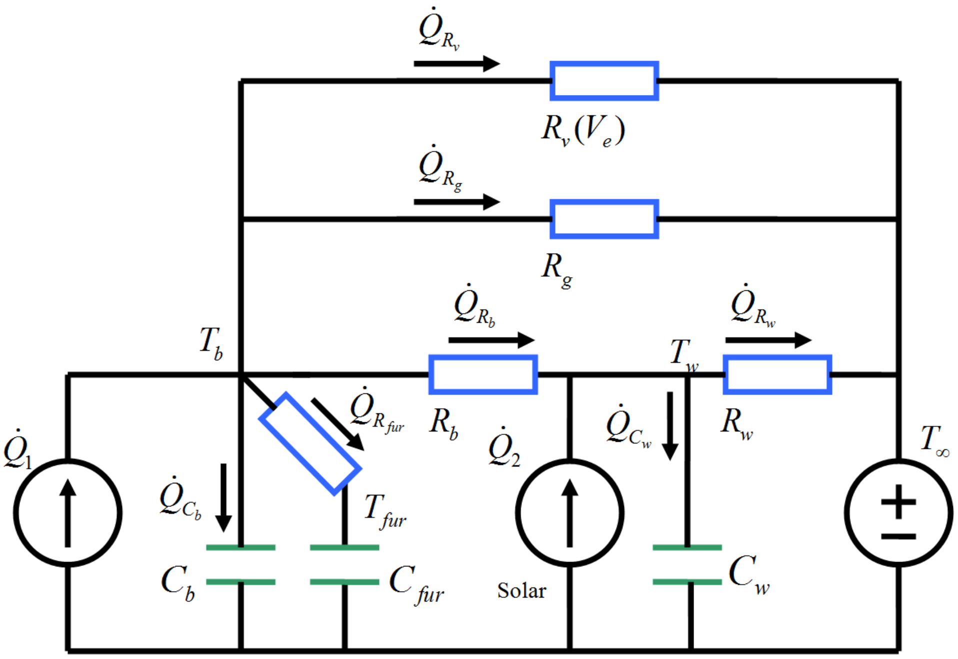


Figure 16 - R5C3 model

To model this as an RC thermal network, as shown in Figure 16, a new capacitor with a series resistor is added to the model. The furniture draws and releases heat from the room temperature and thus the resistor that models the thermal resistance between room air temperature and furniture is added (R_{fur}). Since furniture is assumed to also store energy [1] a capacitor (C_{fur}) is added. This gives the model a new state (T_{fur}). In this way, the slow release of heat from furniture, as well as the energy storage capacity can be modeled. State equations are found to be:

$$\frac{dT_b}{dt} = \frac{1}{C_b}\dot{Q}_1 - \frac{1}{C_b R_b}(T_b - T_w) - \frac{1}{C_b R_{fur}}(T_b - T_{fur}) - \frac{1}{C_b R_g}(T_b - T_\infty) - \frac{1}{C_b R_v(V_e)}(T_b - T_\infty)$$

$$\frac{dT_w}{dt} = \frac{1}{C_w} Q_2 - \frac{1}{C_w R_b}(T_w - T_b) - \frac{1}{C_w R_w}(T_w - T_\infty)$$

$$\frac{dT_{fur}}{dt} = -\frac{T_{fur} - T_b}{C_{fur} R_{fur}}$$

4.5 Data pre-treatment

When working with data sets for empirical modeling there are a number of challenges that needs to be met in order to produce a usable set of measurements. Data from different sources needs to be combined and pre-processed into finished data sets. Typically, data sets for modeling are divided into training and validation sets.

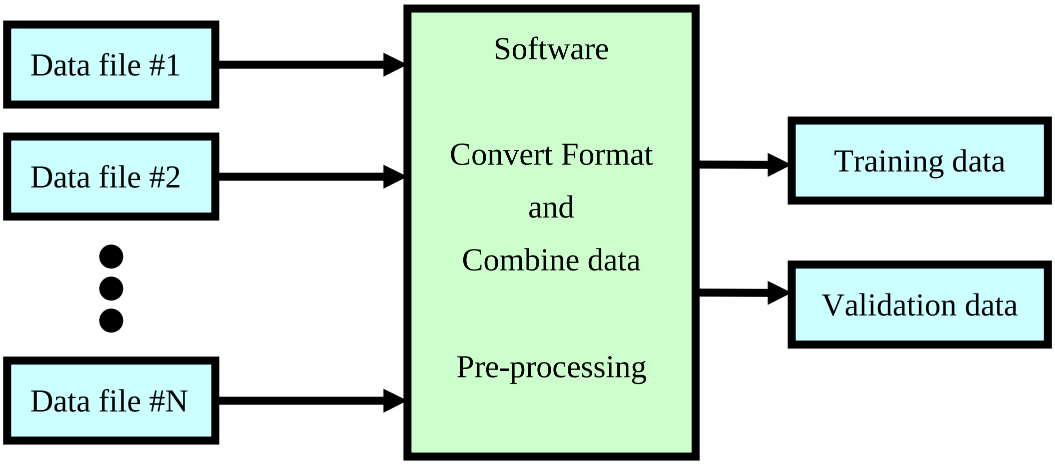


Figure 17 - Data file conversion software

Typically, a large number of sensors and measurement points are required. Therefore, data sets can be of significant size, and often from multiple sources or logging systems. When working with modeling, data from these different sources must be combined and aligned in time, to give a usable data set for the modeling process. Additionally, any outliers and noise in the data should be removed by pre-treatment steps. Figure 17 depicts how software can be used to read each of the input data files, do the necessary pre-processing, and export the finished data-sets on a format readable by the modeling software.

4.5.1 Combining data sets

Measurements from a building contain data from different sources. Typically an internal logging system is used to record temperature and humidity reading throughout the building. A weather station records the outdoor conditions, while a smart energy meter measures power consumption of the active heating systems. Some times external metrological data is also used. Since measurements can come from different logging systems, their output may be on different format. Hence, a tool for converting different formats of data into one format which is usable by the modeling software is typically required. Different countries have different standards for formats of files, such as different decimal point, field separators, etc. Additionally, the different systems may use different timesteps and offsets. Sampling times may not align, even if the sampling intervals are the same. In the final data set, each sample must represent the system at a specific time instant, and have a format that the modeling software can use. Hence, the combination of data from multiple sources requires a significant amount of pre-processing, before modeling can begin.

It is also a likely scenario that the input data sets from each measurement system contain a lot of information not relevant to the modeling, both in terms of extra variables, and also extra samples outside the time range of interest to the modeling. Therefore, some type of extraction is also required in the software that will retrieve only the portion of the input data set that is of interest.

4.5.2 File formats

Typically, data is stored in a Comma Separated Values (CSV) file, which is simple ASCII encoded text, where numeric values is stored as text strings, separated by some field separator. Typically, as the name suggests, a comma or in some countries a semi-colon is used. Further, the decimal separator can vary from country to country. Finally, the line endings can also be different. In Windows computers the ASCII codes CR + LF (10 + 13) is used. Other systems use a single character line ending, such as just LF (13) for Unix.

While these format concerns may seem trivial, they are in fact vital to the data treatment steps. Data must be delivered to the modeling tool, which may have strict requirements on the data format used. As an example, when using MATLAB and its “readcsv” function, as in this project, the data file must use comma as field separator, dot as decimal point, and Unix style line endings of LF. If a file has CR+LF as line endings, this will cause every other row in the data matrix to become zeros.

Figure 17 demonstrates the process of reading data from multiple files and converting them into two data sets, training and validation. Each input file is read and converted from a specific format, while the exported data sets are on the format required by the modeling software.

4.5.3 Re-sampling

In addition to having different formats, different logging systems will typically have different time-scales. Some data requires a high sampling rate because of fast dynamics, such as power consumption of a heater, while other data changes slowly, such as temperature. Therefore, logging systems must use a sampling frequency suitable to the data its intended to record. The sampling frequency must not violate the Nyquist-Shannon Sampling theorem [21]. If there are fast changing signals that need to be logged, sampling rates must be high.

The sampling time of the raw data may not be a good choice for the final data set. If a higher sampling time, i.e. a longer time step, is used in the finished data set, filtering of high frequency signal information may be required to not violate the sampling theorem and avoid aliasing [21].

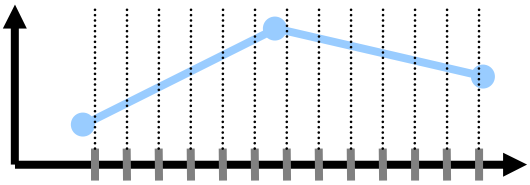


Figure 18 - Re-sampling of data by linear interpolation

In addition to different sampling intervals, the finished data set requires values for all variables at the same time instant. Hence all readings must be “temporally aligned”, meaning aligned in time. This can be done by using re-sampling of the original data, into a new time frame, which is shown in Figure 18, where the vertical dotted lines represent the new time instants where measurements is required, and the blue line is the original measurements taken with longer timestep and a slight offset. By using the original time-frame of the input data, and computing an estimate by linear interpolation of each measured variable at the time required for the output data set, the finished data will have all measurements at the same time instant. This constitutes a single sample at one time instant, even if taken from multiple logging systems whit initially varying and/or unaligned time frames.

The new value between two of the original samples is computed using linear interpolation [22].

$$x(t_k) = x(t_i) + \frac{x(t_{i+1}) - x(t_i)}{t_{i+1} - t_i}(t_k - t_i)$$

This equation can be used for computing the measurement x at time t_k, where time t_k is larger then t_i but smaller then t_{i+1}, and x is known at t_i and t_{i+1}.

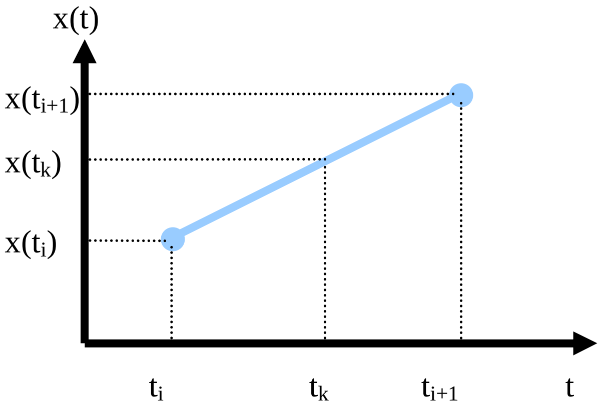


Figure 19 - Linear interpolation

In Figure 19 the process of linear interpolation as described by the above equation is shown graphically. The original data set is parsed to find the two samples at t_i and t_{i+1}, such that t_k lies between them in time.

4.5.4 Filtering

In all measurement systems, there will be measurement noise. Typically this will consist of random signal variations that are not related to the measurand, but are induced into the system as an error. Random noise is often called white noise, because the frequency spectra of random variations contain all frequencies at same magnitude [13]. To remove this kind of noise a filter can be used. There are several kinds of filters, such as time discrete low-pass filters (LP), Weighted Moving Average (WMA) filters, FIR, IIR, and other types [21].

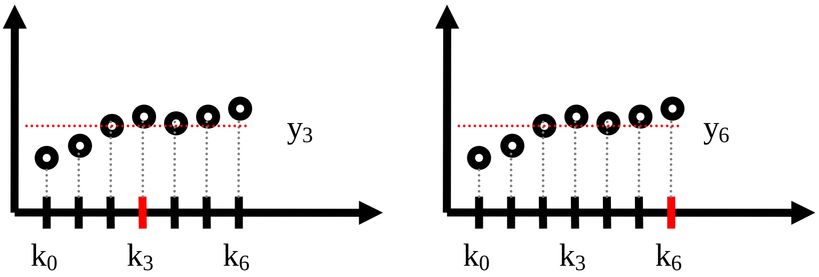


Figure 20 - Centered (left) and Uncentered (right) WMA

In this project a Weighted Moving Average filter is used [22]. This can be of two types, centered or non-centered. If centered, the filter uses an equal quantity of samples from both past and future time to compute the output at the present time instant. Since this filter requires samples from future timesteps, it is only possible to use it on recorded data set, e.g. historical data, and not on data processed continuously from a process.

A WMA filter that is not centered uses only data from previous timesteps to compute the filter output, and will always introduce a delay in the filtered signal. The difference between centered and uncentered WMA filter is shown in Figure 20. The horizontal red line represents the filter output. Both filters give the same output since they average the same seven samples. The difference is in which timestep the output is used for. The uncentered WMA gives the same output as the centered, but three timesteps later, which is what creates the delay in the filter.

The equations for centered (left) and uncentered (right) WMA filters are:

$$x(t_k) = \frac{\sum_{i=k-\frac{N}{2}}^{k+\frac{N}{2}} w_i x(t_i)}{\sum_{i=1}^{N} w_i} \qquad x(t_k) = \frac{\sum_{i=k}^{k+N} w_i x(t_i)}{\sum_{i=1}^{N} w_i}$$

In both filters a set of weights can be defined for each of the N samples used in the filter, where N is the filter window width. Often, all weights equal to 1 and the denominator of both equations become simply N.

Centered WMA filters are used in this project since all data are recorded in files, i.e. historical data.

4.5.5 Outlier detection

The last pre-treatment step of interest in this project is detection of outliers. An outlier is defined as a sample that does not fit the general pattern in the data [18]. There can be many reasons why certain samples should be classified as outliers, such as sensor malfunctions, noise, or changes in conditions that the model is not expected to predict and hence is outside the region of interest. In general, an outlier is a sample that is likely erroneous and would have a large, unwanted, impact on the model, if not removed from the data set.

Different modeling techniques have different approaches to dealing with outliers. In multivariate regression, inspection of score and loading plots can be used to identify outliers [18]. As a general data treatment pre-processing step, outliers can be defined as any sample that has an unlikely magnitude of change, respective of the neighboring samples. By defining a window around each sample and computing an average over this window, each sample can be compared against this average and a limit placed on the maximum deviation acceptable.

$$abs\left(x_k - \frac{\sum_{i=k-\frac{N}{2}}^{k+\frac{N}{2}} x_i}{N} \right) > x_{\lim}$$

If the absolute value of the difference between sample at time k and the average of N neighboring samples is larger then some limit x_{lim}, the sample is classified as an outlier. If an outlier is found the sample can be deleted from the data-set and an interpolated value used instead.

It is worth noting that the process of detecting outliers is often not trivial. The distinction between an outlier and an extreme sample, i.e. a sample that does fit the general trend, but has high values of some relevant variables, is not always obvious [18] and automatic outlier removal can be a risky process. Hence, some manual supervision and tuning of the detection parameters is advised.

Note that automatic detection of outliers is not a guarantied method. Data sets should always be inspected, and if certain samples seem to contain unlikely values, they can be classified as outliers and removed or replaced by an average value. A typical outlier scenario for buildings is when a door or window is left open. This “breaks” the thermal model of the building, and any samples taken on a building in such condition will be outliers in a larger data set.

4.6 Solar irradiation

Solar irradiation is in literature [2, 6, 7] considered an important input to the thermal behavior of a building. In the data on which this project is based little information about solar irradiation is available. It is difficult to measure this parameter directly [3]. In relevant literature an estimation or model based on solar irradiation angles relative to the buildings location is used. Additionally, data from cloud coverage and geometry of building and its surroundings, such as shade from trees and other buildings, is used as input to the estimation.

The data used in this project are taken from buildings in Norway during October, November and December, where daylight is low, sun angle relative to ground is low through out the day, and solar irradiation is also low, as a consequence. It is assumed that solar gains from the relevant time period can be neglected. Particularly for the “ByggeLab” data set, the surrounding buildings provide a high degree of shading, such that treating solar gain as noise in the modeling is a valid assumption. For the “Cabin” data set, this assumption may be an oversimplification, but without better data, or a known, well-defined, method of solar irradiation estimation, the only option left is to treat solar heating as noise.

4.7 Model performance criteria

When comparing results of different combinations of model structures and datasets, a defined quantifiable criterion is needed. The Root Mean Square Error (RMSE) is a commonly used such statistic [18]. Two types of RMSE are used in this project. First, the Root Mean Square Error of Calibration (RMSEC) is used to quantify how well a model fits a specific training data set.

$$RMSEC = \sqrt{\frac{\sum_{i=1}^{N}\left(T_i^m - T_i^c\right)^2}{N}}$$

The error in this equation is the difference between reference in the calibration data set and prediction of the model. T^m is the temperature predicted by the model, while T^c is the reference temperature in the calibration data. It is important to note that the RMSEC is not a measure of the models ability to predict behavior in general, but rather how close the model is able to predict the data set from which the models parameters is identified. As such, it is a measure of the models fit with the calibration data.

At each data point in the training set, the model prediction error is computed, squared and summed over all N samples. The result is divided by the number of samples and the square root is taken. This preserves the unit of the RMSEC such that in the case of temperatures, the RMSEC is given in the unit of °C.

A similar statistic, Root Mean Square Error of Prediction (RMSEP), is used to quantify a models ability to predict indepenant test data [18]. Since the data is now independent of the calibration of the model, RMSEP is a measure of the models ability to predict system behavior. RMSEP is therefore the most important quantity on which combinations of models and data sets can be compared.

$$RMSEP = \sqrt{\frac{\sum_{i=1}^{N}\left(T_i^m - T_i^p\right)^2}{N}}$$

As the equations shows, the difference between RMSEC and RMSEP is the use of T_p in the error term, i.e. a reference from independent test data, rather then the data used to calibrate the model. In empirical modeling this is an important distinction.

4.8 White-box reference model

In [1] a set of equations derived from first principle balance laws is used to model a single-zone building. The resulting model is an example of a white-box model, entirely derived from physical laws and an understanding of thermal dynamics. The result is a set of ODE’s containing 17 states, a number of algebraic equations for boundary conditions and around 70 parameters. This model is complex, and difficult to analyze, but likely to be dynamically accurate, assuming that the modeling assumptions holds and parameters are correct [1].

Parameters of the model are derived from building characteristics and standard code such as the TEK10 regulations [1]. Using a grey-box approach on a model with 70 unknown parameters would not be a feasible solution, simply due to the number of parameters to identify. In literature [2, 6, 7] a model of fifth order (5 states) is considered complex for use in parameter identification.

An interesting use of white-box models such as in [1] is to use it for simulation and using the result data to train a grey-box model. Working with simulated data, rather then real measurement data, gives more flexibility in sensor placement, and allows easier experimentation with various aspects of the building. An example of a study based on simulated data is given in [3].

4.9 Simulation of models

Simulation of models is required for the identification of the parameters of a grey-box model. A numerical computation on the model must be performed, such that the results can be compared with measurement data. It is important to be aware of the timescale, such that the simulated and measured results are aligned in time. The simulated results must be known at fixed time steps, the same time instants as the measured data. Measured data can be re-sampled and aligned to fit the required timescale, and simulations can be designed to produce results at the same time instants as the measured data.

Models used in this project will be on state-space form, using Ordinary Differential Equations (ODE) and will be linear. Simulations on these models will be required both in the c# software, for performing simulations on known models, and in MATLAB, for identification of parameters. In MATLAB there are many well known options for ODE solvers, but for c# the selection is limited, and there are no standard built in features for this. Therefore, implementation of an ODE solver is a required part of this project.

The simplest type of solver to implement is a fixed step solver, where the timestep is independent of the solution, and given to the solver as a configurable setting. In this project a fixed step Runge-Kutta 4th order solver [8] is implemented and used, both in c# and in MATLAB. This allows direct comparison of results and also gives improved computational speed over MATLAB's built in solvers, as later chapters will show. Since parameter identification requires the optimizer to simulate the model repeatedly, solver speed is of vital importance.

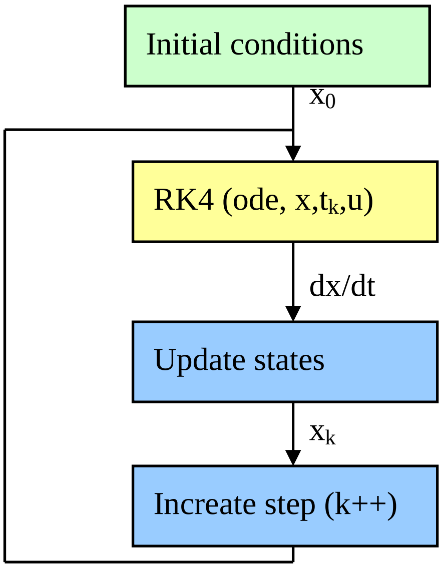


Figure 21 - Simulation loop

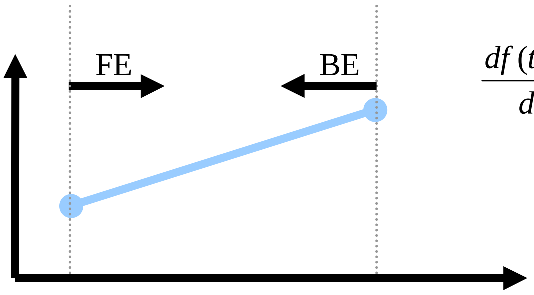
With a fixed step solver, a simple simulation loop like in Figure 21 is used. At each timestep, the model states are computed based on previous computed states. ODE's is evaluated by a solver, such as RK4 in the yellow block. The state variables and time k is updated in the blue blocks. The updating is done in an iterative loop. Before the simulation starts, the initial conditions are set up in the green block. Inputs to the model are provided to the solver to evaluate the ODE'S.

4.10 ODE solving in discrete time

To simulate the models based on ODE’s a discrete time solver is used. These solvers use a numerical method of discretizing the ODE from continuous time domain and into a sampled discrete time domain. In many scientific computing languages, such as MATLAB, this is a standard function that can be invoked when needed. In generic programming languages such as c#, a third party library is typically used, since there is little or no support for numerical computation natively.

The simplest form of discretization is the Forward and Backward Euler (first order accurate), where the time derivative is approximated by the difference in value of the function, between two adjacent samples, i.e.

$$\frac{df(t_k)}{dt} \approx \frac{f(t_{k+1}) - f(t_k)}{\Delta t}$$



$$\frac{df(t_{k+1})}{dt} \approx \frac{f(t_{k+1}) - f(t_k)}{\Delta t}$$

Figure 22 - Backward and Forward Euler approximations

The important difference between Forward Euler (left), also called Explicit Euler, and Backward Euler (right), also called Implicit Euler, is when the difference in value over the sample step is equated to the derivative [24], as depicted in Figure 22. If the derivative at the beginning of the step is used, the approximation is the Forward Euler, since derivative is used forward in time. If the derivative at the end of the sample step is approximated to the difference, the approximation is the Backward Euler [8].

Both these approximations are of first order accuracy. In a Taylor series expansion approximation terms above first order is ignored or truncated [24]. Other approximations have better accuracy. Euler approximation is not used in this project, but it is included here as a background for discretization of ODE’s.

4.10.1 Fixed or variable step solver

In some linear dynamic systems, there may be a large variation in time constants of the different ODE’s. Since the sample time used in discretization depend on the time constants of the ODE’s for the solution to be numerically stable [8, 22, 24], the smallest time constant in the system determines the size of the time step used in the solvers. It may be that this time step is not required for the entire simulation horizon, but rather only when there is a change in the faster dynamics of the system. In this type of system, it would be beneficial if the timestep length could be varied, depending on the present numerical solution.

In this project, all the models are linear, which is typical for thermal networks. Further, the time constants can all be assumed to be of approximately similar magnitude. The heat flow through a wall is typically not drastically different from the heat flow through a roof, window or other building elements. Since variable step solvers are more complicated to implement, only a simple fixed step solver is used here.

During simulation on the complex 17 state single-zone model from [1] some problems with significantly different time constants where encountered, presumably due to how the layers in the walls are modeled with very different thermal properties. Some layers have very fast dynamics compared to the rest of the building because they have a very low thickness and/or thermal resistance causing rapid thermal responses through them. In this case, a variable step solver would have improved the simulation speed significantly, since for fixed step solvers the only remedy is using shorter timesteps [24], resulting in more simulation steps.

4.10.2 Runge Kutta 4th Order solver

An improvement on the Euler approximation is the so called Runge Kutta 4th order approximation. As the name suggest, this approximation is 4th order accurate, and offers significant improvements in accuracy and stability over Euler approximations [8, 24].

Where the Euler schemes approximate the derivative only at the beginning or end of the timestep, the RK4 method uses a central approximation, where the derivative is also estimated in the middle of the sample step [8]. Additionally, an initial “guess” of the true derivative is used and updated with improved computations. Detailed discussion on RK4 implementation is found in literature [8].

The RK4 scheme can be formulated by a set of equations as:

$$\begin{aligned}
k_1 &= \Delta t \cdot f\left(t_n, x_n\right) \\
k_2 &= \Delta t \cdot f\left(t_n + \frac{\Delta t}{2}, x_n + \frac{k_1}{2}\right) \\
k_3 &= \Delta t \cdot f\left(t_n + \frac{\Delta t}{2}, x_n + \frac{k_2}{2}\right) \\
k_4 &= \Delta t \cdot f\left(t_n + \Delta t, x_n + k_3\right) \\
x_{n+1} &= x_n + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right)
\end{aligned}$$

First, the algorithm computes an initial guess for the change in x between time n and n+1. Half of this change is then used to compute a new guess for the derivative in the middle of the step. Again, the guess is updated using the best available previous guess, before the derivative at the end of the timestep is computed, using the guess in k_3 as an estimate. Finally, all four guesses is combined into an estimate of the change in x between time n and n+1, and x is updated, thus advancing the algorithm one timestep. While this method is more computationally heavy then the Euler schemes, it can be shown to be fourth order accurate [22, 24], meaning that in terms of a Taylor series approximation, the 5th and higher order terms is truncated. The RK4 method is the chosen approximation of the derivative in this project, since it is simple to implement in code (c# and MATLAB) and also faster then the built in solvers in MATLAB.

4.11 Optimization based parameter identification

As described in chapters 2.4 and 4.3, a critical part of the grey box modeling procedure is the identification of parameters from measurement data [4]. There are many methods possible for the identification of these model parameters [4]. In the this project the method of choice is the use of an optimization algorithm, specifically the “Interior Point” algorithm, which is also used in [2]. The actual implementation is done in MATLAB and using the non-linear optimizer “fmincon”.

4.11.1 Optimizer working principle

The working principle of an optimization algorithm is to find a minimal point for the objective function. As such, the objective function defines what goal the optimizer is trying to achieve. This process is similar to finding the minimal point on a graph by equating the first order derivative to zero and the second order derivative to be negative, i.e. finding a minimal point. Optimization algorithms do this numerically, without need for computing the derivative/gradient.

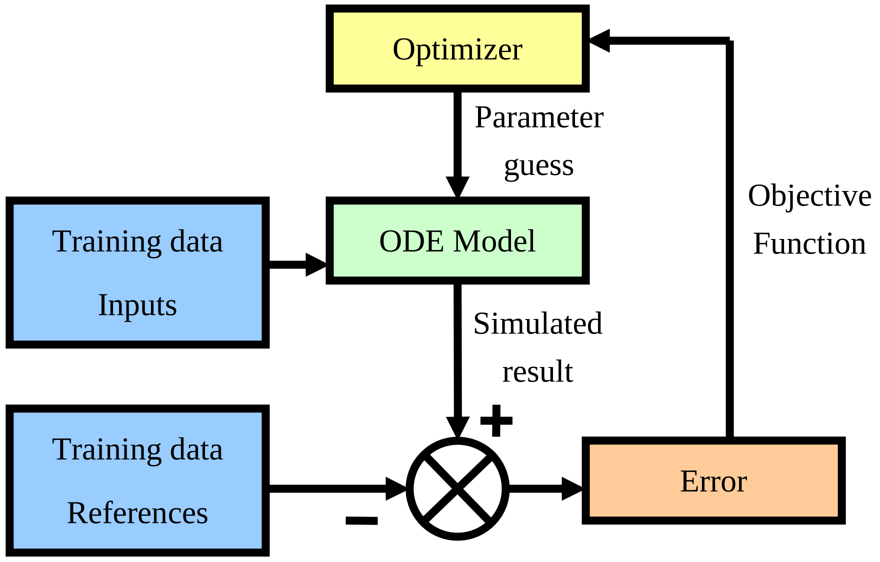


Figure 23 - Optimization based parameter identification

The principle of optimization used in parameter identification, as depicted in Figure 23, is based on repeated simulation of the model (green block), using the optimizer algorithm (yellow block) to search for an optimal set of parameters that minimizes the error (orange block) between simulated results from the model compared with the measured reference values of the real system (blue blocks). The objective consists of the sum of squared error for all reference variables as compared to model outputs. In this way, the end result of the algorithm, at convergence of the optimizer, is a set of parameters that when used in simulation with the given model, gives the best possible fit to the training data set. RMSEC can be used to quantify the degree to which a model fits a data set.

At all iterations of the optimizer algorithm a guess for all values for the “decision variables”, in this case the model parameters, are suggested. The way the different algorithms comes up with new guesses, based on the success or failure of the previous guesses, is the difference between various algorithms, such as SQP, Interior Point, Active Set, Nelder-Mead and others. To compute a numeric value representing the success of each guess in achieving the objective given to the optimizer, the model must be simulated over the time range present in the training data. The simulation results are used to compute the objective function, the accumulated square error, over the whole training data set.

At the initialization of the algorithm an initial guess of the model parameters, also called decision variables in an optimizer, is given. Finding a good starting point for the algorithm to work from is important, as there can exist multiple locally minimal solutions [2]. Finding a good general solution for the model may require some trial and error in initial model parameters [2].

In addition to the defined objective, a set of constraints may also be formed for the decision variables. In [2] the authors suggest using minimum constraints on the parameters as 1/3 of the initial parameters and a maximum constraints of 3 times the initial parameters, thus limiting the range around the initial conditions for where the optimizer is allowed to search for an optimal solution. By repeatedly running the optimizer, changing the initial conditions after each run, it is possible to find a set of initial parameter values that produces a solution where none of the parameters are on the constraint boundries [2]. The exception would be if there is no solution to the problem, i.e. no set of parameters that can make the model fit the given data.

5 Software Implementation

There are several software tools developed for this project, some in c# and some in MATLAB. General syntax details for c# and MATLAB, and also for GUI programming e.g. WinForms, can be found in literature [12] and is not included here. Software modeling [12] and implementation of the key software elements in this project is discussed in detail. Where applicable, results of tests on individual software parts are also included here, to demonstrate the software results. A selection of source code for the important parts of the developed software is included in Appendix C. A selection of class diagrams is presented as a model of the software. These class diagrams are somewhat simplified relative to the source code and some details have been omitted.

An overview of the developed software and required input data is given in Figure 1 and discussed in chapter 2. Since all the experiments are done based on data sets from multiple sources, a software tool that can combine and pre-process the data into suitable training and validation data sets is required. The model calibration, i.e. the identification of parameters, is performed in MATLAB, because it requires support for optimization algorithms. Since the parameter identification, as discussed in chapter 4.11, requires simulation of the model, the models and the RK4 solver, discussed in chapter 4.10, is also implemented in MATLAB. Additionally, solvers and models are implemented in c# in a tool designed primarily for simulation of models, but also with the option to export data. It can be used to generate simulated data from more complicated models, such as the white-box model in [1].

The three main software tools cooperate to solve the tasks in this project. LogFile Converter delivers pre-processed training and validation data, made from different input data sets, to the Parameter Identification software in MATLAB. The identified parameters can then be given to the simulator, which can store simulation settings and parameters in files for easy access. Simulations can be run on both identified and pre-configured models with known parameters. Results from model calibration and validation are produced by MATLAB code. Results from simulating models and estimating the heating/cooling time of buildings is also done in MATLAB, but can be produced by the c# software as well.

Note that all result plots in this project are made by the MATLAB parameter identification software. Simulation results from the c# “Simulator” software is used as training and validation data for some test cases, and also as proof of working software. The most important results from the c# simulation is the reduced computational time achieved by the fixed step RK4 solver, compared to MATLAB built in solvers, as discussed later in this chapter. Further, the c# results compared to MATLAB results show that simulating models in c# is viable, which is a requirement for future predictive control system implementation.

5.1 Software development process

All software in this project is designed using Object Oriented Analysis and Design (OOAD) [12] principles, thus creating software that can be modified and expanded. Examples of where this type of thinking is important, is the model classes in c#. A common base-class is used with inheritance [12] for multiple models. This allows the solvers to use the polymorphism concept of OOAD [12] to solve any model defined by derived classes. This means the software can be expanded with new model classes, should other thermal network structures be investigated as for grey-box modeling.

The developed software can be considered experimental software, since the specification of the software was somewhat unclear at start of implementation. Further, the software was expected to grow with the needs of the project. Since there is only one developer in the project, a simple development process suitable for such software was used. At the initial stages of software development, a conceptual design of each software was made, using a class diagram [12]. These are efficient tools to model software classes and cooperation between classes, to ensure a hierarchal inheritance model in the software that will allow the system to grow. By using the concepts of inheritance and virtual overloaded methods (polymorphism), new classes, e.g. new solvers, models, operations, can be added into the existing framework.

For the purpose of documenting software functionality, a Use-Case diagram [12] for each software was made. These diagrams show the main functions of each software, and what each of the function need in terms of external input/output from actors [12].

By getting the modular design of the software right from conception, each class and each module of the software can be developed and tested. For example, each operation in the "LogFile Converter" software can be tested separately, once the controller-class framework is implemented and tested. This ensures that each part of the software is working, before the whole software is tested as a combined tool.

5.2 Data treatment application

The data treatment application, called simply "LogFile Converter", is the software implementation of the pre-processing steps described in chapter 4.5. Each of the steps is implemented in its own class, called an "operation". Operations can be linked together in a "macro", to perform a complete transformation from raw input data into usable training and validation data sets. Operations include reading files, extracting data, re-sampling, filtering [10], outlier detection and exporting data to CSV file.

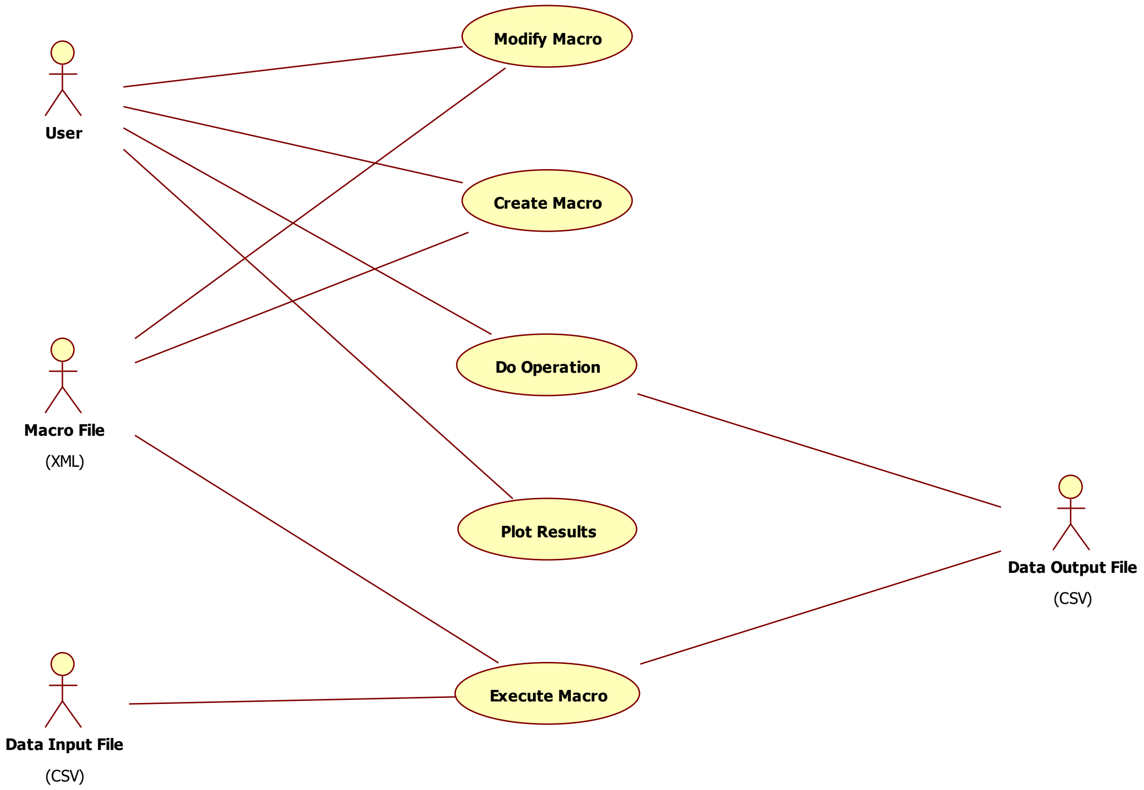


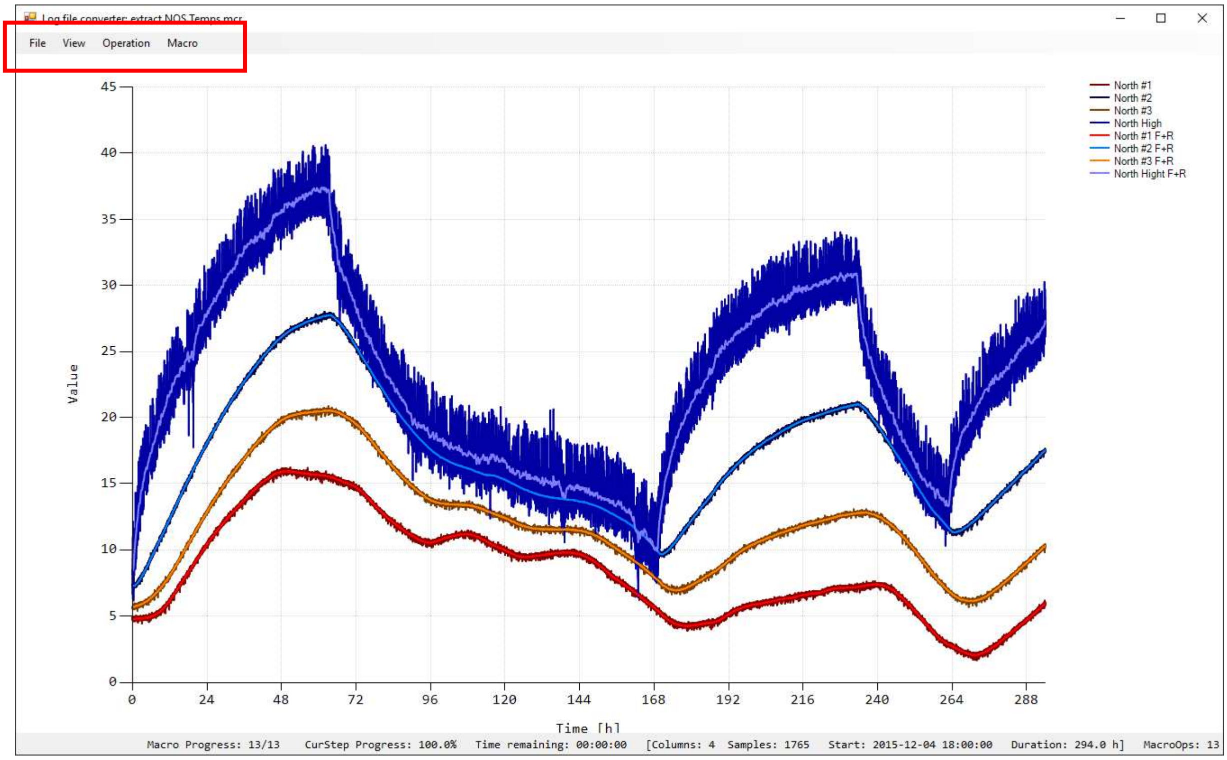
Figure 24 - Use-case diagram for LogFile Converter software

As shown by the use-case diagram in Figure 24, the software is focused around the creation, modification and execution of a "Macro". As described, a Macro is a set of operations required to convert the input data into the desired output, and can include plotting of results. The most important use-case [12] is "Do Operation", which is responsible for executing each of the operations defined in chapter 4.5. The "Execute Macro" use-case relies on "Do Operation" to perform each operation in a sequence determined by the Macro. "Plot Results" has the task of presenting results on GUI, either as a graph or a matrix with data. "Create Macro" and "Modify Macro" is responsible for recording operations into a macro file, and modifying a pre-existing macro loaded into the software, respectively.

The first actor in Figure 24 is the "User", the person who uses the software to create data sets for modeling. The actors "Data Input File" and "Data Output File" is stored information on disk, typically Comma Separated Values (CSV), as raw untreated and resulting treated data, respectively. The final actor "Macro File" represents a stored Macro, as an eXtensible Markup Language (XML) file, a set of operations and settings, including settings for plotting, stored on disc, such that the macro can be reloaded into the software and the resulting data set recreated directly from the raw data.

All operations work on a worker data set, which is internal to the software and therefore not included as an actor. When an operation is executed it is given this worker data set as input. On completion, the worker data set is updated with the results of the completed operation.

Then, the next operation to be executed repeats the process, building on the results of the last operation.

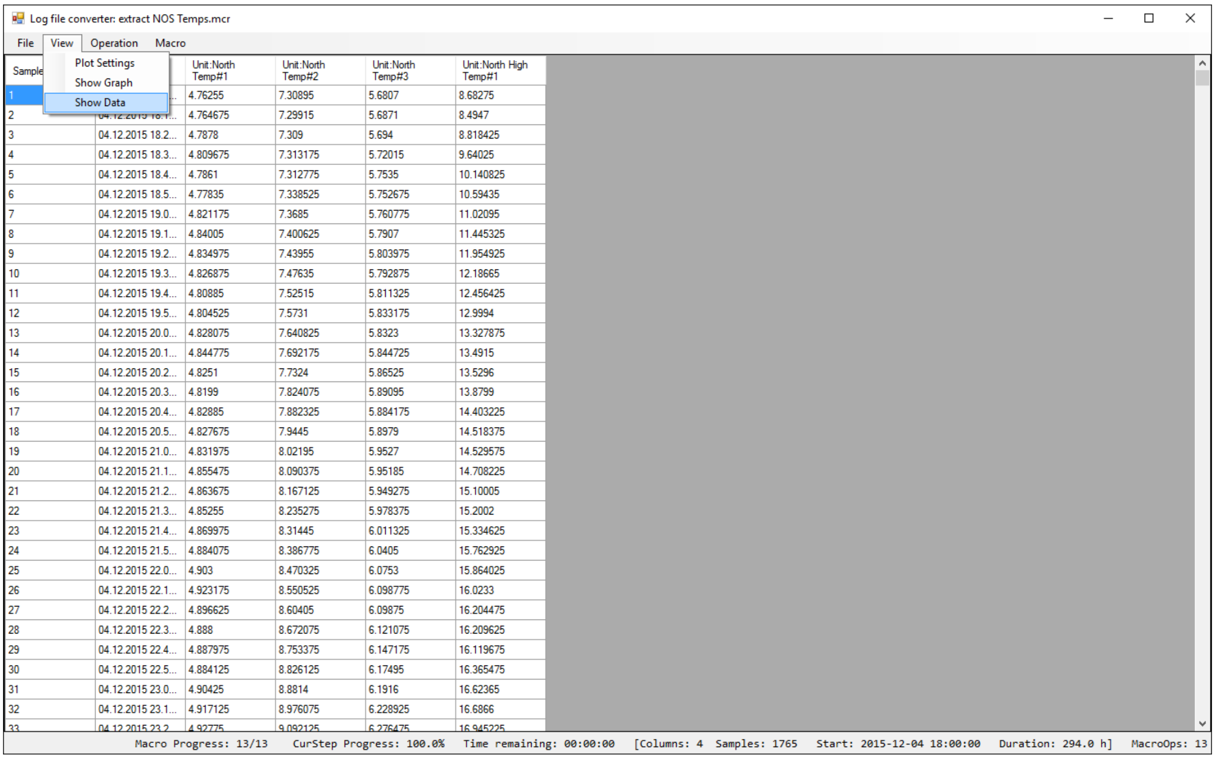


```
Macro Progress: 13/13    CurStep Progress: 100.0%    Time remaining: 00:00:00    [Columns: 4   Samples: 1765    Start: 2015-12-04 18:00:00    Duration: 294.0 h]    MacroOps: 13
```

Figure 25 - LogFile Converter Screenshot

A screenshot of LogFile Converter software is given in Figure 25. The main part of the GUI is used for either a plot of up to 8 graphs, or it can be used to show a table or matrix type presentation of the current worker data set in the software.

A menu bar, marked with red square in Figure 25, allows users access to various commands, such as handling macros, configuration and setup. At the bottom of the GUI, a status line shows current statistics of the active data set, including start time, length, number of samples and variables. Additionally, the number of steps in the loaded macro is shown. At the left side, the progress of the currently loaded macro is given. Since some operations can take a significant amount of time, particularly re-sampling, an estimate of the remaining time to completion is also given.



Sample		Unit:North Temp#1	Unit:North Temp#2	Unit:North Temp#3	Unit:North High Temp#1
1	...	4.76255	7.30895	5.6807	8.68275
2	04.12.2015 18.1...	4.764675	7.29915	5.6871	8.4947
3	04.12.2015 18.2...	4.7878	7.309	5.694	8.818425
4	04.12.2015 18.3...	4.809675	7.313175	5.72015	9.64025
5	04.12.2015 18.4...	4.7861	7.312775	5.7535	10.140825
6	04.12.2015 18.5...	4.77835	7.338525	5.752675	10.59435
7	04.12.2015 19.0...	4.821175	7.3685	5.760775	11.02095
8	04.12.2015 19.1...	4.84005	7.400625	5.7907	11.445325
9	04.12.2015 19.2...	4.834975	7.43955	5.803975	11.954925
10	04.12.2015 19.3...	4.826875	7.47635	5.792875	12.18665
11	04.12.2015 19.4...	4.80885	7.52515	5.811325	12.456425
12	04.12.2015 19.5...	4.804525	7.5731	5.833175	12.9994
13	04.12.2015 20.0...	4.828075	7.640825	5.8323	13.327875
14	04.12.2015 20.1...	4.844775	7.692175	5.844725	13.4915
15	04.12.2015 20.2...	4.8251	7.7324	5.86525	13.5296
16	04.12.2015 20.3...	4.8199	7.824075	5.89095	13.8799
17	04.12.2015 20.4...	4.82885	7.882325	5.884175	14.403225
18	04.12.2015 20.5...	4.827675	7.9445	5.8979	14.518375
19	04.12.2015 21.0...	4.831975	8.02195	5.9527	14.529575
20	04.12.2015 21.1...	4.855475	8.090375	5.95185	14.708225
21	04.12.2015 21.2...	4.863675	8.167125	5.949275	15.10005
22	04.12.2015 21.3...	4.85255	8.235275	5.978375	15.2002
23	04.12.2015 21.4...	4.869975	8.31445	6.011325	15.334625
24	04.12.2015 21.5...	4.884075	8.386775	6.0405	15.762925
25	04.12.2015 22.0...	4.903	8.470325	6.0753	15.864025
26	04.12.2015 22.1...	4.923175	8.550525	6.098775	16.0233
27	04.12.2015 22.2...	4.896625	8.60405	6.09875	16.204475
28	04.12.2015 22.3...	4.888	8.672075	6.121075	16.209625
29	04.12.2015 22.4...	4.887975	8.753375	6.147175	16.119675
30	04.12.2015 22.5...	4.884125	8.826125	6.17495	16.365475
31	04.12.2015 23.0...	4.90425	8.8814	6.1916	16.62365
32	04.12.2015 23.1...	4.917125	8.976075	6.228925	16.6866
33	04.12.2015 23.2	4.92775	9.092125	6.276475	16.945225

```
Macro Progress: 13/13   CurStep Progress: 100.0%   Time remaining: 00:00:00   [Columns: 4  Samples: 1765   Start: 2015-12-04 18:00:00   Duration: 294.0 h]   MacroOps: 13
```

Figure 26 - LogFile Converter Data view

Figure 26 shows another screenshot of LogFile Converter, where the software is in the “data view” mode. Here, all data in the worker data set can be inspected in a table or matrix form.

5.2.1 Data storage classes

The software is modeled using class diagrams, to insure that future maintenance and expansion of the software is possible [12]. Likely extensions can be adding more pre-processing operations, or expanding existing operations, perhaps adding different types of filters or data file formats.

Class diagram for the software is split up in two parts to make discussion of the implementation easier to follow.

Since the software is concerned with handling large quantities of information from log files [10], with a varying number of variables (columns) and samples (rows), and also with varying formats, text or numbers, and time scale, both timestamps and intervals, it is useful to define an internal data storage structure. This structure can be used by the different operations to handle data in a uniform way, regardless of the original data source. By using a set of classes to hold the data, other classes can rely on the data conversion and access methods in these classes to do basic operations on the data, such as retrieving field values as text or numbers, or accessing a particular sample in the data set.

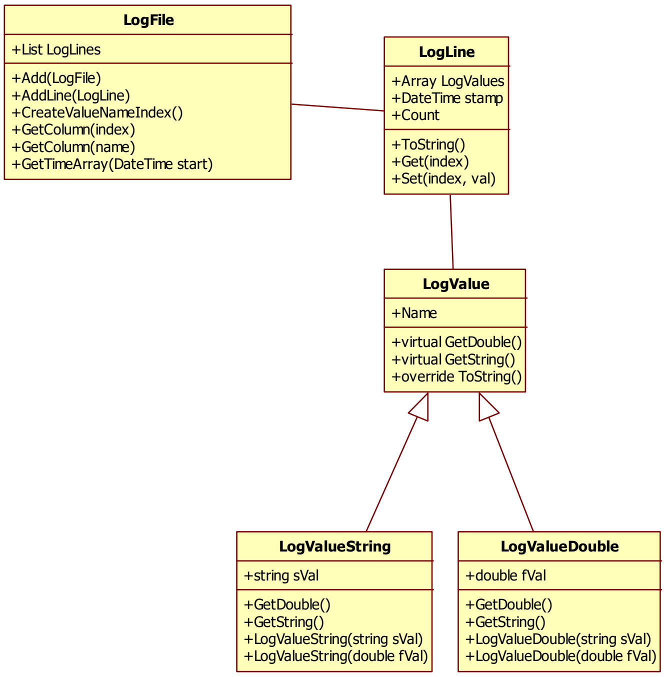


Figure 27 - Class diagram for LogFile objects

Figure 27 shows the class diagram for the data storage classes. A LogFile class is a set of samples, either raw data read and converted from a file, combination of multiple files, or it can hold the results of any of the defined operations. The class has some accessor methods that allow operations to retrieve a column, i.e. variable in the data, by index or name, or get a list, in the form of a c# dictionary, of all columns/variables by name and index. For purpose of plotting, the class can return an array of time offsets from a given start time. Note that the c# classes DateTime and TimeSpan are used to handle all time information allowing efficient computations with time and conversion to and from text strings.

The LogFile class has a List-structure of LogLines, which represent the samples in the data set. This class also stores the timestamp of each sample, either as read from file, or as created by an operation, e.g. re-sampling.

LogLine holds an array of LogValues, which is where the actual data is stored. LogValue class itself is never used directly, but serves as a base class with two overloaded child classes [12]. LogValueDouble is designed to hold a value of numeric format, while LogValueString can hold a value of text. Both classes have overridden the base class virtual methods GetString and GetDouble. This allows operations to simply retrieve any value in a LogFile as a double or string, depending on what the operation needs. As an example, filtering only needs numeric values, and will retrieve a field as GetDouble, while exporting a file to CSV uses GetString to retrieve the value of each stored element as a string.

Finally, all the classes use overloaded [] (“bracket”) operators, which allows LogFile objects to be accessed like a two-dimensional array, where the first bracket holds the sample index and the second bracket returns a specific indexed value from within the LogLine.

```
//make a list of all variable names in the file
Dictionary<string, int> dct = Input.CreateValueNameIndex();
int nColumns = dct.Count;

double[][] fVal = new double[nColumns][];
for(int nC = 0; nC < nColumns; nC++) {
        fVal[nC] = new double[Input.Count];
        for(int nSamp = 0; nSamp < Input.Count; nSamp++) {
                fVal[nC][nSamp] = Input[nSamp][nC].GetDouble();
        }
}
```

Figure 28 - Code example of accessing LogFile class

Using operators in this way gives a simple and readable code that is easy to maintain, as shown in Figure 28. The “Input” data set is an object of type LogFile, which in the above code is converted to a 2D double array. Note that the double array is “transposed” in the sense that the indices in brackets are reversed with respect to the LogFile object. First a dictionary of all columns is retrieved using “CreateValueNameIndex”. Next, the code loops over the entire LogFile Input, all samples and variables, returning the double value of each of them using GetDouble virtual method. As shown, this makes for very simple and readable code, which is important in all software, particularly in software that is expected to be expanded in the future.

5.2.2 Data operations classes

The theory behind all the operations is discussed in details in chapter 4.5. In this chapter, only the implementation details are covered.

The data operations consist of six operation classes, with a common base class “DataOperation”. A framework, or Controller pattern class [12], “DataOperationCtrl”, is used to hold a queue of operations that is executed on a background thread. Since operations can take a long time to complete, they must be executed in a background thread [11], and not on the GUI thread of the application, otherwise the software will appear to “hang” while operations are performed. This would give an unresponsive, and not user friendly, software.

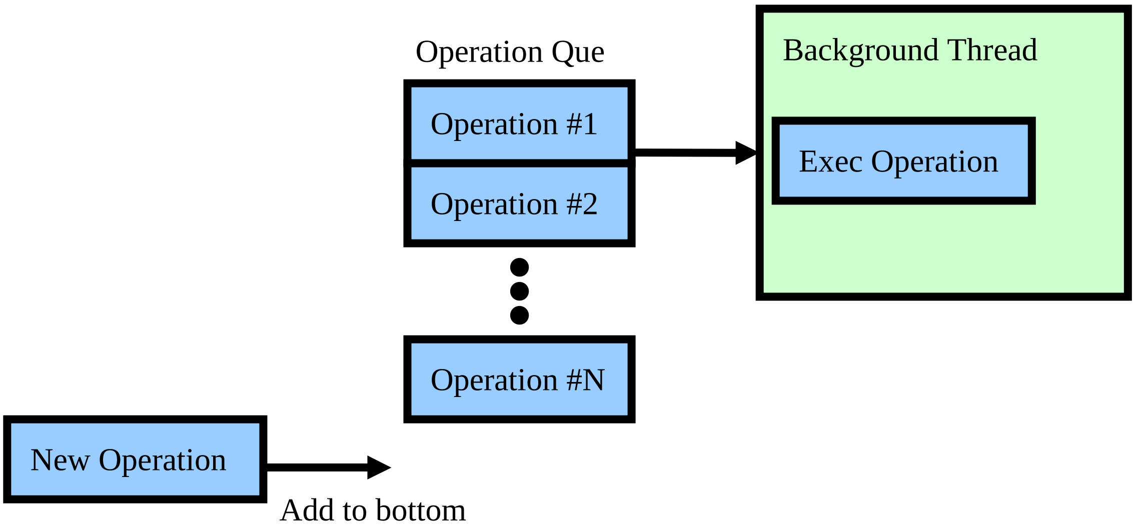


Figure 29 - Operation execution in thread

In Figure 29 it is shown how a queue of operation is stored in a controller class containing a worker thread [11]. New operations are added to the bottom of the queue, while the top operation is executed in the background thread. When the operation is complete, the next operation in the queue is ready for execution.

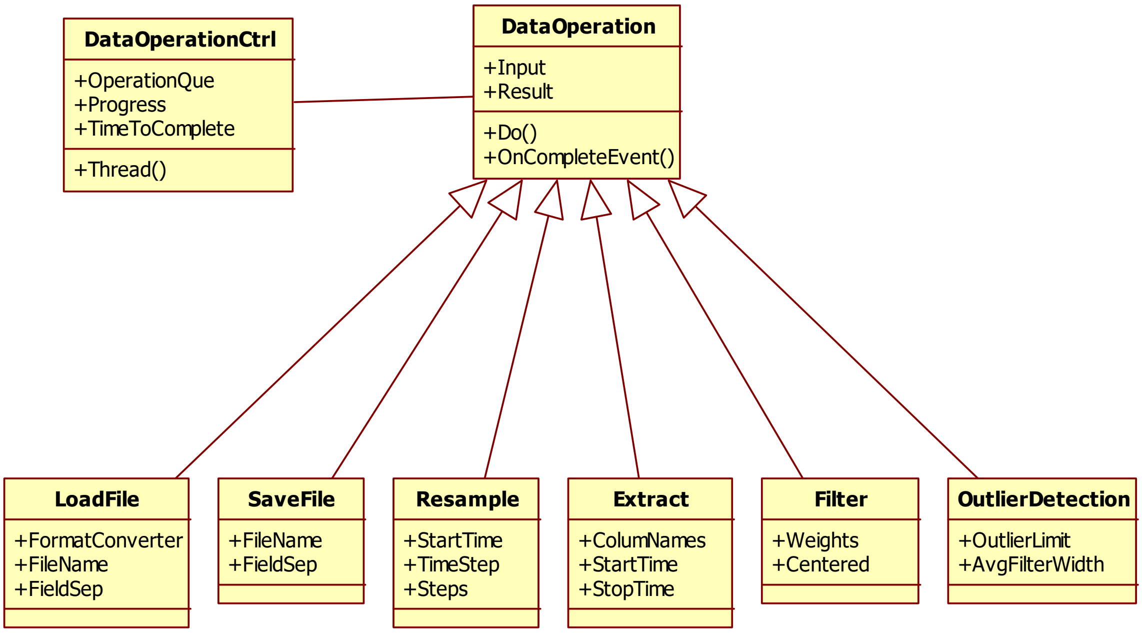


Figure 30 - Class diagram for data operations

Figure 30 shows a class diagram of the operations and the controller class. All operations are inherited from a common base class called ”DataOperation”. The base class has a virtual method named ”Do” which is called by the framework to perform the operation. Each operation child class overrides the “Do” function, and performs whatever actions on the data it requires. Additionally, the base class has two LogFile type data sets called Input and Result.

When an operation is created, it is given a LogFile object as Input, and once complete, it returns the data in Result. Additionally, the base class contains an “OnCompleteEvent” that the controller will call [12] once the operation has completed. Since the operation is implemented as an object, all the required information for use of the results is encapsulated inside [12]. The user can configure the operation object before passing it to the controller, including setting the callback event on completion.

DataOperationCtrl class has a property called Progress that can be used to monitor progress of the current operation. Similarly, “TimeToComplete” is of type TimeSpan and returns an estimated time to completion for the currently executed operation.

In Figure 25 an example of an executed macro is shown. Here, a data set is loaded and a list of columns in the original data set is extracted for a specific time interval. The results are filtered. Both raw and filtered data is plotted to show what the filtering operation has done with the data.

Pre-processing data in this way gives users of the software much flexibility in configuring the pre-processing steps, as well as opportunity to see what happens at each step, thus monitoring the process of creating the training and validation data sets.

5.2.2.1 LoadFile operation

The first operation in any Macro will be to load data from a file. Since the LogFile Converter is expected to handle many types of formats, the “LoadFile” operation uses a special “converter” class called FieldConverter, which can take one line from a file, in the form of a string, and return the values as a LogLine object, including timestamp and field names.

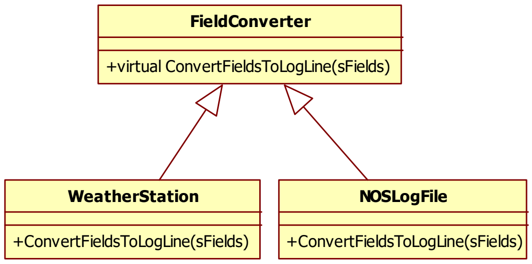

FieldConverter
+virtual ConvertFieldsToLogLine(sFields)
WeatherStation
+ConvertFieldsToLogLine(sFields)
NOSLogFile
+ConvertFieldsToLogLine(sFields)

Figure 31 - File format conversion classes

As shown in Figure 31, the FieldConverter class can be overloaded, where each child class overrides the ConvertFieldsToLogLine method. This allows the software to support new file formats, simply by defining new derivatives of FieldConverter classes. By having specialized conversion classes, a large number of file formats can be included, depending on project needs.

5.2.2.2 Filtering Operation

The filtering operation, as detailed in chapter 4.5.4, is demonstrated in Figure 25. From the figure, it is demonstrated how a noisy data set can be filtered by averaging the data over a specified window. In the figure, the lighter colored plot is overlaid on top of the raw data, to show how the filter extracts the general trend in the data, but removes the random variations from sensor readings, termed as noise. The filtering of data is a standard operation and much literature exist where detailed discussions on filtering can be found [10, 21, 22].

5.2.2.3 Resampling Operation

The theory behind the resampling operation is discussed in chapter 4.5.3.

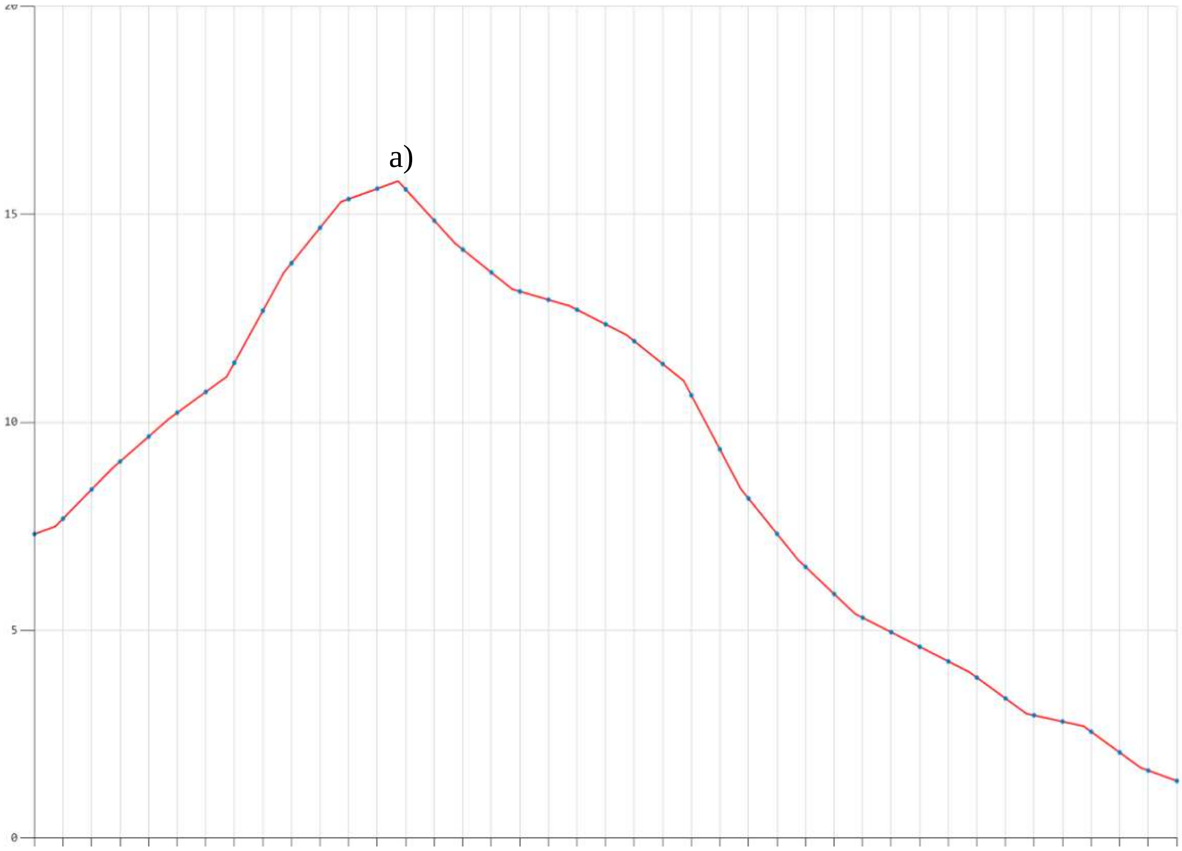


Figure 32 - Screenshot of resampling operation

In Figure 32 the result of applying the resampling operation is shown. The red line is the original data, while the blue dots represent values at the required time instants, also shown by vertical grey lines. Each point is computed by linear interpolation. This is most clearly seen at the peak marked a) in the red curve, where the resampled values on both sides of the highest value original sample are clearly computed by linear interpolation between relevant two closest samples.

5.2.2.4 Outlier Detection Operation

The outlier detection operation is described in detail in chapter 4.5.5. In outlier detection, there is always a certain amount of human judgment involved, even when applying automatic algorithms. In the field of chemometrics, outlier detection is often done manually [18]. Here, an automatic algorithm is used, and as such the human judgment factor is expressed by the settings used in the algorithm. What should be defined as an outlier, and what should be included as extreme samples, is not always obvious.

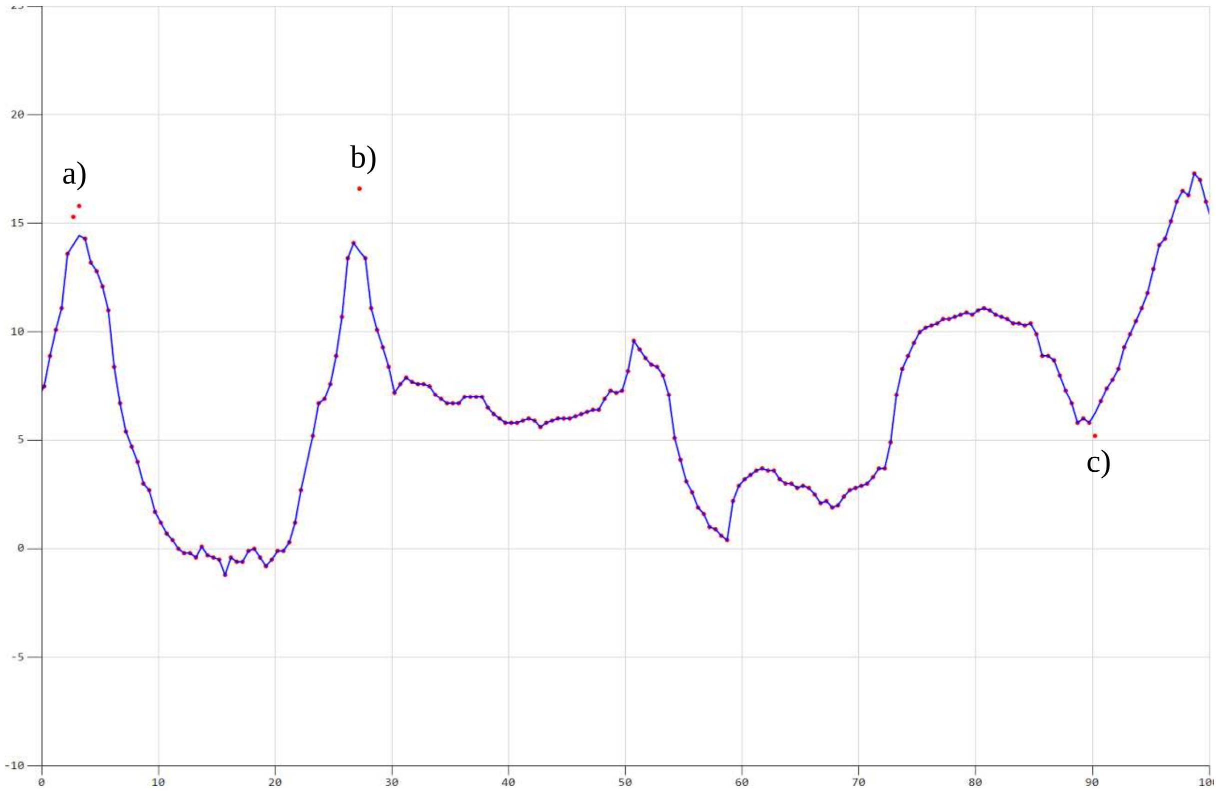


Figure 33 - Screenshot of outlier detection operation

In Figure 33 the results of the outlier detection operation is shown. Red dots are original data points, while the blue line plot is the data after outliers have been replaced with an average value of nabouring samples, using interpolation similar to in resampling. From the plot in Figure 33, four outliers have been identified at points a), b) and c), which are shown as red dots without the blue line plot going through them. If these samples are actually outliers is a matter of judgment call. The purpose here is simply to demonstrate how the algorithm works.

5.2.3 Files with macro steps

It is likely that a set of pre-processing steps could be used several times, perhaps with slightly varying input files and output file names. Also, it is considered an advantage in data driven modeling, if all training and validation sets can be re-created from raw data, with possibility to vary pre-processing parameters such as outlier limits and filter window width. This allows users to experiment with the pre-processing in an efficient way.

LogFile Converter handles this by storing a so-called “macro” in a file. A list of data operations, together with steps for plotting if required, can be saved in an XML file [11]. This allows a macro to be recorded, by doing the procedure step by step the first time, and then saved to file. Later, the macro can be loaded and executed again by the software, or users can edit the macro, execute it step by step, or add more steps to the end of the macro, thus expanding the data pre-processing procedure.

5.3 Simulation Application

The second tool developed in this project, is the “Simulation” software in c#. This software is intended for simulation of a set of known model structures. Implemented models include a selection of thermal network models, a low-pass filter test model and the model from [1]. The parameters and inputs can be varied. The model structures are coded as separate classes, and thus the software has a limited selection of models it can simulate. Due to the OOAD based software design [12] it is easy to expand the software with more classes and new models in the future.

The simulations could have been performed with only MATLAB. In fact, they all need to be for the parameter identification process described in chapter 4.11. However, having a platform in c# for simulating ODE based models is useful also for future work, both to demonstrate results of models, and also to have flexibility in choices of platforms in future controller implementations, such as discussed in chapter 2.1.2. It can not be assumed that all building has a computer capable of running MATLAB.

The simulation software is not used in the generation of the results presented later in this report. These are all generated by the MATLAB code. However, development of c# simulations for models is a requirement of the project (Appendix A). In particular, the c# Simulation software can be used to simulate and compute heating times for models, which is what a future control system based on this project would require. The c# tool is also used to generate simulated data on for the cases A-D in the results in later chapters. Finally, it is of interest to test the computational speed of ODE solvers in c#.

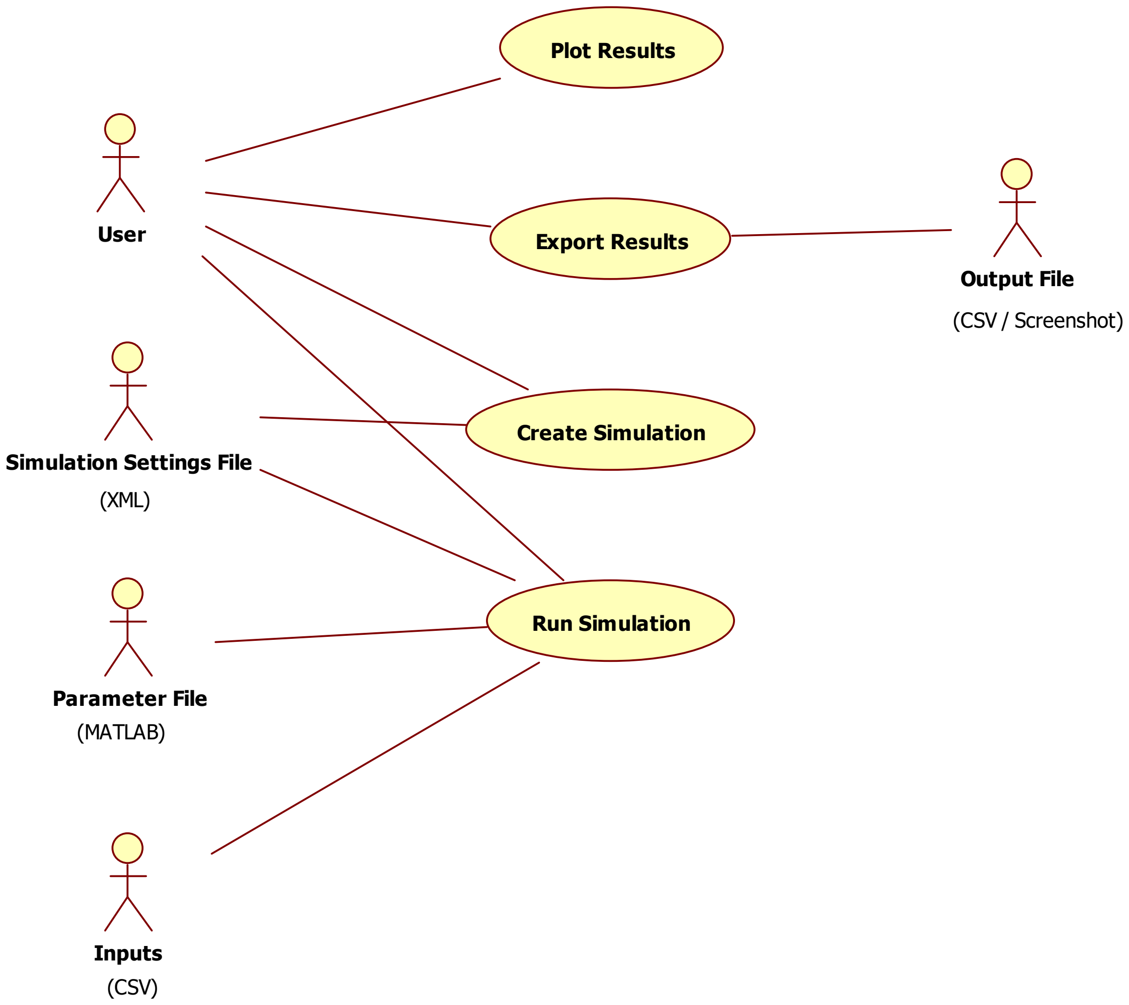


Figure 34 - Use-Case diagram for Simulation software

The most important use-case [12] for the Simulation software, from the diagram in Figure 34, is “Run Simulation”. On user request it will execute a simulation, computing the results, given inputs and parameters. The use-case “Create Simulation” is responsible for setting up a simulation, with all necessary configuration details, and storing it to a file. When a simulation is finished results can be plotted by “Plot Results” or exported to a file as screenshot or as a CSV data file by “Export Results”.

The actor [12] “User” is the person operating the software, while “Simulation Settings File” and “Parameter File” are files on disc storing simulation settings and model parameters, respectively. Parameter files use MATLAB code style syntax so the same file can be read directly by MATLAB as well. This file format is also human readable. The actor “Input” is the simulation model inputs, which can be either a separate file or part of the “Simulation Settings File”.

Simulation configuration is stored in files, containing both settings for the simulation and for the plotting of results. The software is made to be flexible, so that users can customize simulation and plotting with a high degree of freedom. Keeping simulation settings in files, allows users to load, modify and re-run previous simulations. Inputs to the models can be stored in separate files, which can be created by measurement systems from real world conditions, perhaps using LogFile Converter, as described in chapter 5.2, for converting data, or the input files can be created manually. Alternatively, input to models can be specified in the simulation file via settings in the software.

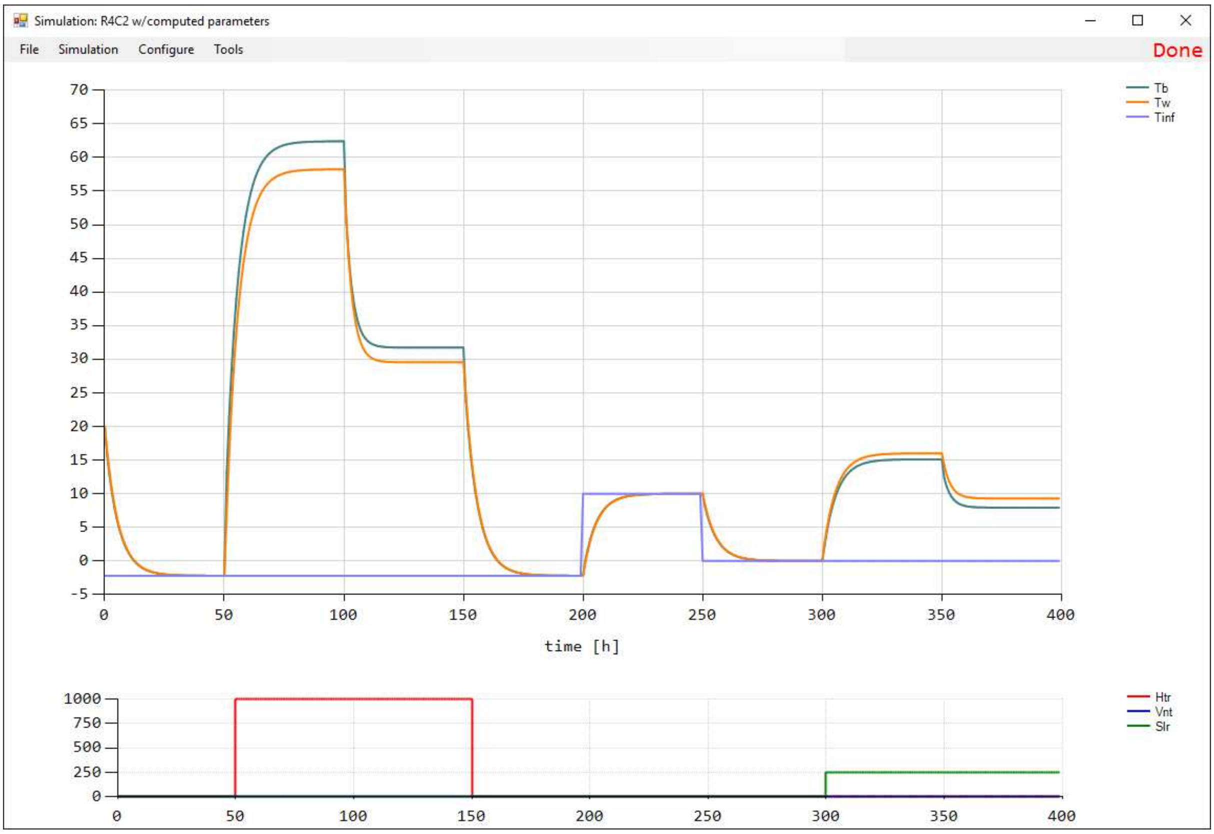


Figure 35 - Screenshot of Simulation software

Figure 35 presents a screenshot of the “Simulation” software tool, after a simulation has been completed. Settings and operations such as starting a simulation or loading a file can be accessed from the menu bar on the top of the GUI. The main part of the GUI is reserved for two separate plots, each with up to 8 graphs, where the user can configure plotting of model states, outputs or inputs, to give a customizable presentation of the simulation results. The height of each plot can be configured, depending on the importance of the contents. Axis range can be defined for both plots separately, allowing flexibility in presentation of data.

In the top right corner of the GUI, a progress indication is shown. When simulating a large number of timesteps, the running time may be significant, so an indication of progress is useful. All computations are done by a background thread, such that the GUI remains responsive while computations are performed.

5.3.1 ODE Models and Solvers

The backbone of the “Simulation” software is the computations performed to solve an ODE model for a given timestep length and a given number of timesteps. The computations are divided into two types of classes, both derived from its own base-class.

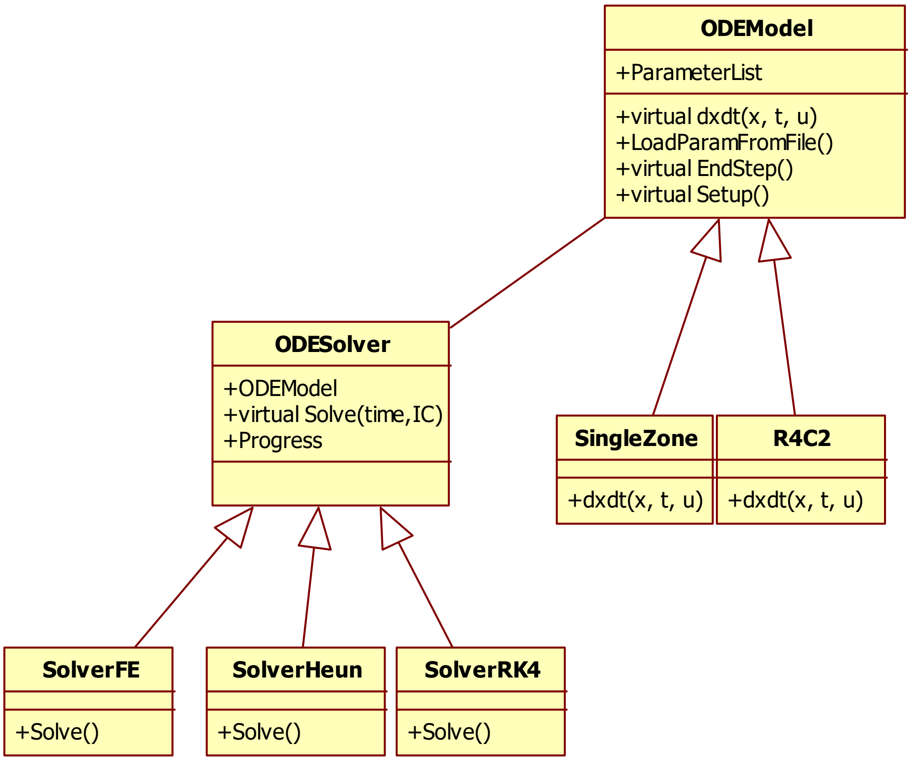


Figure 36 - Class diagram of ODE Model and ODE Solver classes

As shown by the class diagram in Figure 36, the software contains an “ODE Model” class, which is used as a base class for all the models developed. The principle way ODE’s are solved numerically requires a callback function that can compute the current value of the ODE, as described in chapter 4.10. This callback function computes a value for the differential equation, given a set of states, inputs and the current time. These callback functions are implemented as the ODEModel derived classes, one class for each model the software is able to solve. Further, the parameters of the model are read from a file, such that the simulation configuration can specify the model and define a location for the parameter file.

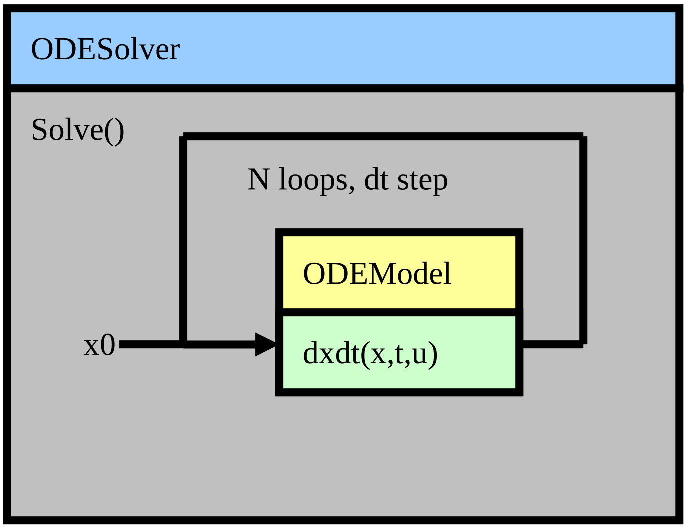


Figure 37 - ODESolver calling dxdt in ODEModel as part of simulation loop

The virtual method “dxdt” (green block) is the callback function, shown in Figure 37, named dxdt since the value it returns is the time derivative of the model states, at a particular time instant. This dxdt method is where the mathematical equations of each model are implemented in code.

A second callback function called “EndStep” is used to allow models to do any computations required *after* the solver has finished computing one step. In solvers like RK4, dxdt will be called four times for each timestep, thus some computations that are only required to do once for each timestep can be placed in “EndStep”. An example of this use can be computation of outputs as a function of the newly computed states.

The method “Setup” is also a virtual method that child classes can overload. This method is called by the framework such that the model can do preparations before the solver starts to work. Typically, this will include loading model parameters from file, thus avoiding that parameters must be read more then once.

Further, the software has a second base class, called ODESolver. Once the models are defined, the system needs an implementation of a solver for ODE equations. These solvers all have some commonalities between them, such as reliance on a callback function for evaluating the dx/dt function. The ODE Solver class has an attribute which is an object of an ODEModel derived class it is tasked with solving. The virtual method “Solve” is overridden by each child, to implement its own way of numerically solving the ODE. Solver schemes are discussed in chapter 4.10. Figure 37 shows how the two classes ODESolver and ODEModel cooperate, and how the Solve method calls the dxdt method to return values for the ODE’s.

In the class diagram of Figure 36 there are three solvers, Forward Euler, Heun and RK4. The latter is discussed in detail in chapter 4.10.2. Heuns method is similar to RK4 but with only one “guess improvement” and is 2nd order accurate. This method has not been used in the project, but it is implemented in the software.

5.3.2 Software Testing, quality of simulations

During software development, each class is tested separately [12]. Since this software is relatively simple in structure, the most critical item to test is the accuracy of the computations. Since this requires all the classes of the software to cooperate it tests the software as a whole.

To test the solvers, a simple model of a standard Low-Pass filter (LP) is implemented into the system as an Ordinary Differential Equation (ODE) model. This model can be used to test the solvers, since the results are very well known and solvable by hand computations for comparison.

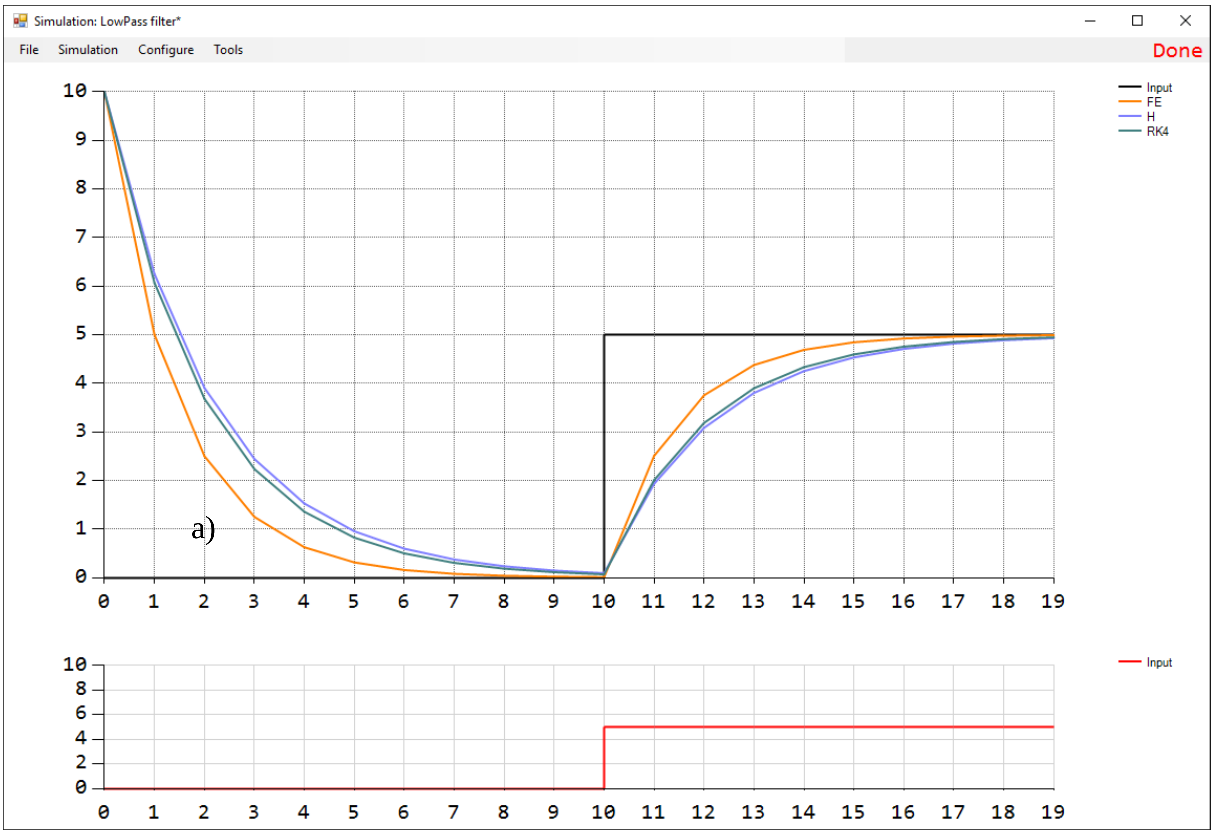


Figure 38 - LP simulation with three solvers

In Figure 38 the results of simulating an LP filter with Time-constant = 10s and initial state 10 for 10 timesteps of 5 sec and then applying an input of 5 for 10 steps is given. From the manual solution to the first order LP filter, it is known that when applying a step change, after one time constant, ~63% of the total change will have happened. From Figure 38, at timestep 2 at point a), where t = 10 sec equals one timeconstant, the RK4 simulation gives a value for the LP filter state of ~3.7 which indeed corresponds to a change of ~6.3, or ~63% of the total change.

By inspecting the values of the solution at this timestep (value after t = 1 time constant, 10 sec), comparing to hand computation, the simulation results with RK4 solver is found to be of acceptable accuracy. Simulated solution gives after 10 seconds 3.6817. The correct hand computed value is 3.6788 (=10e^{-1}) which gives 0.3% error with timestep of 5 seconds. Reducing timestep to 2.5 seconds gives a simulated result of 3.6789, where the error has dropped to 0.01%. This shows that the solvers appear to be performing their task of numerically solving the ODE's correctly, as long as an appropriate timestep for simulation is chosen.

It is interesting to note from Figure 38 that the RK4 and Heun solvers give almost the same results, while the FE solver is dropping to fast when the state variable declines, and rising to fast when the state increases. This can be attributed to the FE methods use of the derivative at the start of each time step, as an approximation for the whole time-step. Hence, when the state is increasing, the derivative is computed to high, and similarly when the state is decreasing, the derivative estimate is to low.

By shortening the timestep significantly, all solutions will converge towards the true solution, but at a cost of increased computing time. In the remainder of this project, the RK4 solver will be preferred, due to increased accuracy and stability, compared to FE and Heun, as discussed in chapter 4.10.2.

5.3.3 Simulations and configuration classes

With the backbone, the computational engine of the simulations, implemented, the remaining part of the Simulation software is concerned with configuration and storage of configuration in files. In this software a "simulation" is defined as a set of "experiments", where each experiment is the solution of a single ODE model by a single solver. Thus, a simulation can contain multiple models and solver combinations. This is useful to compare different solvers against the same model, or multiple models against each other. In all result cases in this project, a simulation file will contain only one such model and solver pair.

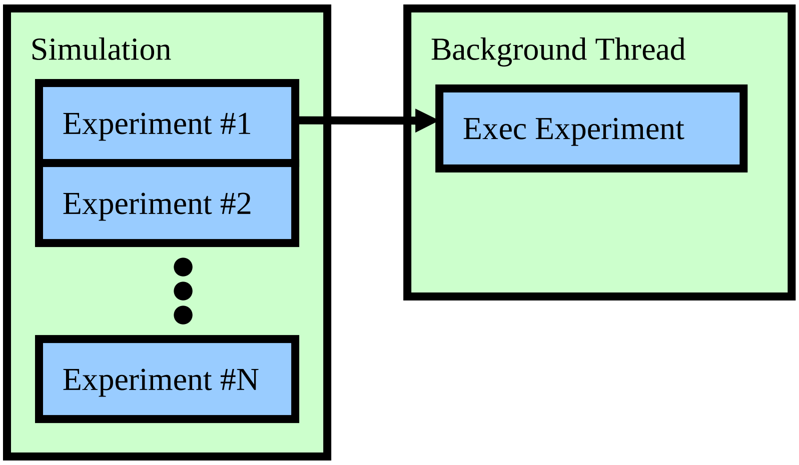


Figure 39 - Simulation experiments running in a background thread

In Figure 39 it is shown how a simulation can contain multiple experiments as defined previously. The combination of one model and one solver gives one experiment. Each experiment is solved in a queue in a background thread, similar to the que, thread and controller setup used in the LogFile Converter software, discussed in chapter 5.2.

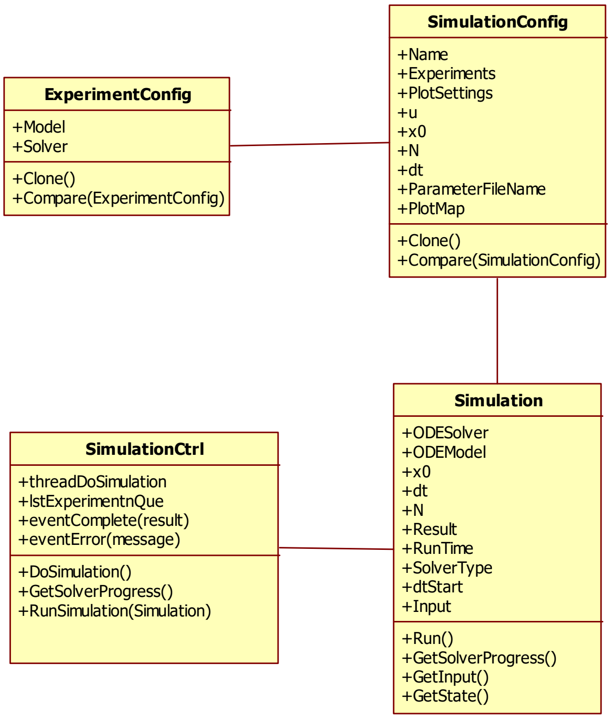


Figure 40 - Class diagram of Simulation related classes

A class diagram of the software is shown in Figure 40. The Simulation class is a worker object in the software, where objects of classes derived from ODESolver and ODEModel is linked together with configuration data required for the simulation, such as initial state x0, timestep dt, number of steps to simulate N and the Input data to use. The simulation objects has a Property “RunTime” which can be read to get an estimate of the expected time to completion. Methods GetState and GetInput are used to retrieve results from the simulation object, once the model is solved. By keeping all relevant information encapsulated in the simulation object, the software can treat it as a complete package, where all information to interpret the results is contained.

The “SimulationConfig” class is a representation of the settings required by the “Simulation” class. This class holds all the settings in a structure that can be exported to an XML [11] file for storage. “ExperimentConfig” is simply a helping class to store combinations of Model and Solver in a XML file. Both of these configuration classes have overloaded methods for cloning and comparing, allowing software to check two configurations against each other for changes. This is useful to detect when the software should prompt user to save results, and to make a deep copy of the configuration. A deep copy is a value by value copy that does not have any memory references to the original object.

The final class, SimulationCtrl contains the background thread object which executes the solvers computations. A queue of experiments is used to hold both current and future experiments for a simulation. Once an experiment has finished, the next one in the queue will run, until all experiments in the simulation is finished.

When each experiment is complete, a callback event is fired to alert host application of the results. Applications can use this event to trigger plotting of the results, or store results to file. Similarly, an error event is called if the solver detects a problem, or an exception is thrown from the worker thread, such that the parent application can be alerted that something has gone wrong.

5.3.4 Input handling

Typically, inputs to simulations is either configured as a simple stepwise constant or read from a file. To accommodate both these input types, the "Simulation" software can be configured to read data from a CSV file or given a set of input lines directly in the configuration file, which is part of the setup functions in GUI.

```
100x       1,     2,      3
```

Figure 41 - Input line example

Each input line, either from configuration or read from file, is assumed to be a comma separated string, using comma as field separator and dot as decimal sign. Any whitespace characters such as space or tab is filtered out, allowing files to be more user friendly formatted. The final syntax detail is the use of an "x" symbol in the beginning of a line, only preceded by an integer number. If an "x" is detected in a line, the preceding integer K is assumed to mean "repeat this line K times". Thus, stepwise constant inputs, such as the one shown in Figure 35 can be configured, without repeating each line K times. The software assumes that each input line is used for a single timestep, thus using the multiplier to repeat each line multiple times allows the input on that line to be used for multiple timesteps. In Figure 41 an example of an input line, as expected by the Simulation software, is shown. This line would be interpreted as having three inputs u0, u1 and u2, with a value of 1, 2 and 3 respectively, all for 100 consecutive timesteps. Note that the spaces are ignored by the software reading the input line text string.

5.4 Model implementation

Since implementations of the models are required in MATLAB for the parameter estimation, as discussed in chapter 4.11, as well as in c# for simulation software, it is natural to implement them in a similar fashion. Models are implemented in MATLAB using functions similar to the dxdt virtual method of the ODEModel class in chapter 5.3.1.

```
function [dxdt,y] = ModelR4C2(t,x,u,theta)
      *extract parameters from theta
      *extract states from x
      *extract inputs from u
      *compute ventilation resistance
      *compute differential equations
      *assign return value dxdt
end
```

Figure 42 - MATLAB model implementation

Figure 42 shows an example of what a model implementation looks like in MATLAB. The dxdt function has the same structure for all models, and both implementations, MATLAB and c#.

In modeling, the parameters are typically contained in a vector called theta. The first step is to extract parameters from this theta vector. This is done such that the differential equations can be written on a more natural form using parameter names that are the same as in the model derivation and RC schematic drawings, e.g. Figure 12. For the same reason, states are extracted from the state vector x, as well as inputs from vector u, and given names from the model development. This avoids the ODE equations referencing the x (current states), u (inputs) and theta (model parameters) vectors by indices, and they can instead be written in a form similar to the mathematical expressions.

$$\frac{dT_b}{dt} = \frac{1}{C_b}\dot{Q}_1 - \frac{1}{C_b R_b}(T_b - T_w) - \frac{1}{C_b R_g}(T_b - T_\infty) - \frac{1}{C_b R_v(V_e)}(T_b - T_\infty)$$

$$\frac{dT_w}{dt} = \frac{1}{C_w}Q_2 - \frac{1}{C_w R_b}(T_w - T_b) - \frac{1}{C_w R_w}(T_w - T_\infty)$$

In code:

```
dT_b = 1 / C_b * Q1 - 1 / (C_b * R_b) * (T_b - T_w) - 1 / (C_b * R_g) * …
      (T_b - T_inf) - 1 / (C_b * R_v) * (T_b - T_inf);

dT_w = 1 / C_w * Q2 - 1 / (C_w * R_b) * (T_w - T_b) - 1 / (C_w * R_w) * …
      (T_w - T_inf)
```

Figure 43 - Model on state-space form and as code

In Figure 43 an example of how two ODE equations in a state-space model can be converted to code in a way that is recognizable. MATLAB and c# code is similar when implementing this type of equations, but the code in Figure 43 is taken from the MATLAB implementation. This is significantly easier to debug then using the x, u and theta vectors with indices directly.

5.4.1 Ventilation as resistance model

Modeling ventilation as a variable thermal resistance is discussed in chapter 4.4.2.

```
function R_v = Ventilation(N,R_vent)
    R_v = Inf;

    if(N > 0)
        G_vent = 1 / R_vent;
        R_v = 1 / (G_vent * N);
    end
end
```

Figure 44 - MATLAB implementation of resistance ventilation model

The implementation of this model in MATLAB as a function that can be called by models to compute a resistance value is given in Figure 44. The function simply implements the model, by testing if the ventilation volume pr time parameter N is zero or not. If zero, the resistance returned is infinity, otherwise it is computed according to the given model.

5.4.2 Comparing MATLAB to c# to verify

With models implemented in both c# and MATLAB, simulations are run on both implementations and the results compared.

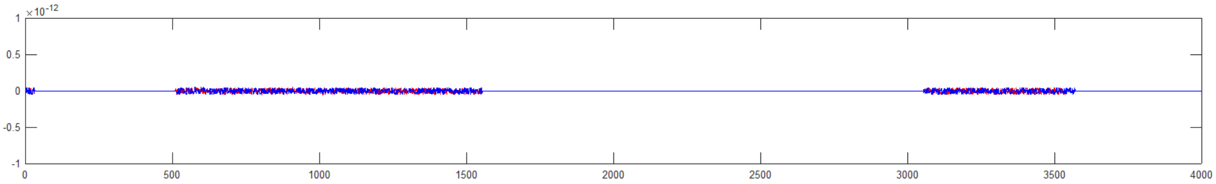


Figure 45- difference between c# and MATLAB for R4C2

By subtracting the states for model R4C2 as simulated in MATLAB from the results of the c# application the plot in Figure 45 can be created. Note that the y-axis is in scale 10^{-12}, such that the differences are on order of 10^{-14}. This test shows that the model implementations give identical results in both languages, down to the precision of the double data type.

5.5 RK4 implementation and MATLAB built in solvers

Fixed step RK4 solver is used in MATLAB, as well as in c#, as described in 4.10. In c# there is no built in choice of ODE solvers, but for MATLAB there are a number of ready made solvers that could have been used. Rather then using any of these built in functions, such as the commonly used “ode45”, this project uses a fixed step implementation of the RK4 algorithm [8]. This is done for two reasons. Firstly, it allows a direct comparison of the results in the two implementations, c# and MATLAB, since exactly the same algorithm is used. The only expected difference is caused by rounding in double number format, and is in the range of the double data type precision, as shown in chapter 5.4.2. Further, the fixed step solver is faster in simulation then the built in ode45, since no computation time is used to find the best step length.

```
function [dxdt] = RK4(x,t,u,dt,model)
    %F1d
    [F1d,y] = model(t,x,u);

    %F2d
    xt = x + dt / 2.0 * F1d;
    [F2d,y] = model(t + dt/2.0,xt,u);

    %F3d
    xt = x + dt/2.0 * F2d;
    [F3d,y] = model(t + dt/2.0,xt,u);

    %F4d
    xt = x + dt * F3d;
    [F4d,y] = model(t + dt,xt,u);

    %compute next state
    dxdt = 1 / 6.0 * (F1d + 2.0 * F2d + 2.0 * F3d + F4d);
end
```

Figure 46 - MATLAB implementation of RK4 method

MATLAB code for implementation of the RK4 method is given in Figure 46. The details of this method are discussed in chapter 4.10.2. The implementation follows the well defined scheme of the RK4 method [22], where the first step is simply to evaluate the model at the present timestep in simulation. Second step uses that evaluation to estimate the state half a timestep ahead, and uses that to re-evaluate the model. This is repeated in step 3, but now using the results of step 2 as the state estimate. Finally in step 4, the state at the end of the present timestep is evaluated, at t + dt, using the result in step 3 as an estimate for the state. With all four steps completed, the final state estimate is computed by a weighted sum of each of the four previous results. The returned results is an estimate for the time derivative of the state vector x. Thus, the simulation is advanced as shown in Figure 21, simply by adding the returned dx/dt value to the previous state estimate, i.e.:

$$x_{k+1} = x_k + \Delta t \cdot dxdt$$

Note that in the implementation in Figure 46 the return value has not yet been multiplied with the step length dt. This implementation detail is done to have compatibility directly with the Forward Euler method [8] when using the RK4 function in a simulation loop.

To test the implementation of RK4, which is already shown to be identical in c# and MATLAB, the R4C2 model is simulated using the built in ode45 in MATLAB.

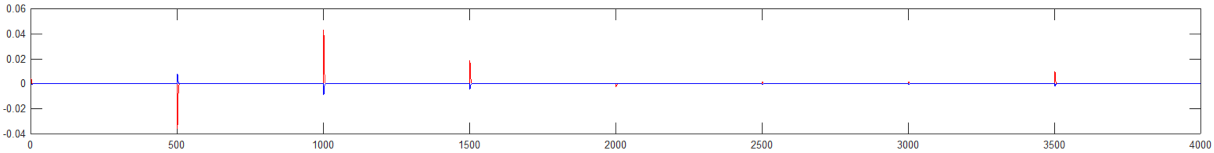


Figure 47 - Simulating R4C2 in MATLAB with ode45

Figure 47 shows the result of this test. As expected the error between ode45 and RK4 implementation are close to zero. The only discrepancy is at the timesteps when inputs are changed. Since ode45 does not use the exact same timesteps as RK4 there is a slight difference, but the magnitude is around 0.05 degrees centigrade which is significantly less then the precision of any sensors used, hence deemed to be neglitable.

It is interesting to note that simulating 4000 steps with ode45 takes around 8.8 seconds, while using the RK4 fixed step solver takes only 0.5 seconds. This is a reduction of nearly twenty times in computational time. Further, when running the same simulation in c#, the computation time is much lower, around 0.01 second. When using the simulation in the optimizer for parameter identification, the simulation is run repeatedly, possibly thousands of times. Hence the computational time of one simulation is important for this project.

Simulation time is also important to this project because the proposed final control structure has realtime requirements [11], as discussed in chapter 2.1.4. Based on the simulations performed here, the realtime requirement should be fulfillable for any reasonable loop times in the supervisory controller. Looptimes for a system tasked with temperature control is assumed to be no shorter then 1-5 minutes.

5.6 Parameter identification in MATLAB

Identification of model parameters is done exclusively by MATLAB code, and not in c#. This is mainly due to use of optimizer algorithms in MATLAB. There are alternatives for c# but MATLAB’s “fmincon” function is well known and ideal for use in this type of project.

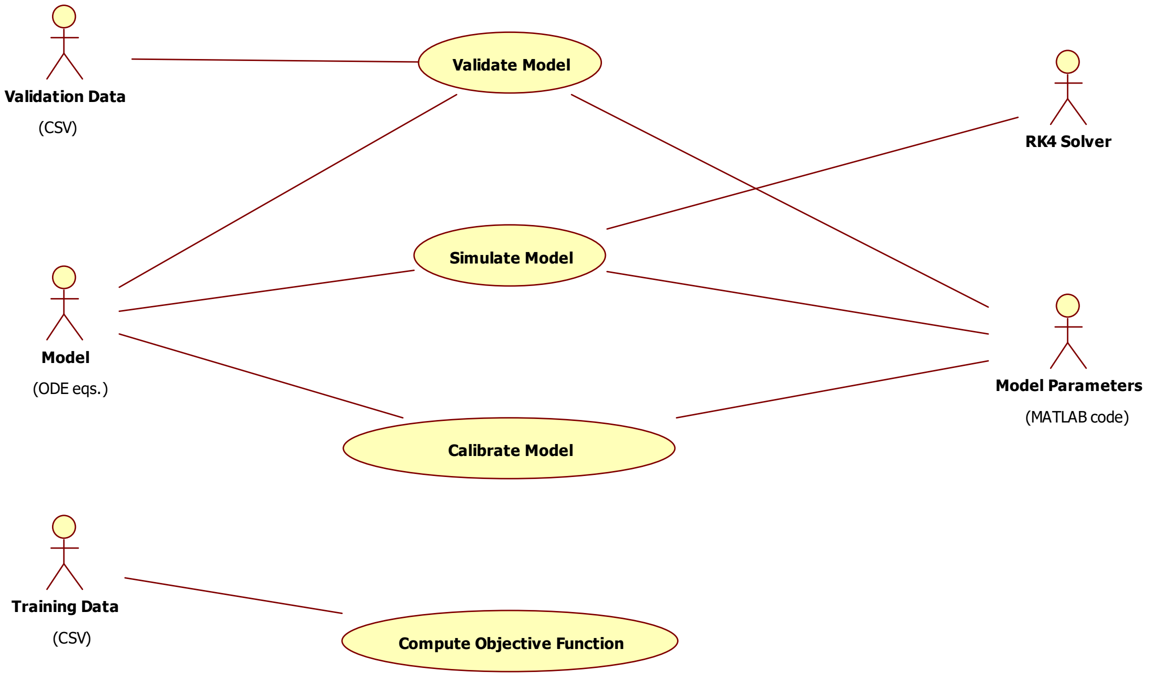


Figure 48 - Use-Case for MATLAB code

A use-case diagram showing all the main functionality in the parameter identification code in MATLAB is given in Figure 48. The most important use-case here is “Calibrate Model”, implemented in the “Optimize” function. This use-case is responsible for identification of model parameters, which is the purpose of this software. To identify parameters, “Calibrate Model” relies on “Compute Objective Function”, implemented in “Compute” function, to evaluate the quality of any given parameter set, both with respect to the objective score, but also with respect to inequality constraints for each parameter. To find the objective score of a given parameter set, the model is simulated over a given range of inputs and time, which is handled by “Simulate Model” use-case. Finally, at convergence of parameters, i.e. when optimizer have found the “best” parameters, a local minima of objective function, the model must be validated using independent testset data, which is done by the use-case “Validate Model” Validation returns a Root Mean Square Error of Prediction (RMSEP) [18] indicating how well the model is able to predict a set of known reference data. Model parameters are stored in a vector, typically called theta in modeling.

The actors “Training Data” and “Validation Data” are independent CSV data sets. “Model” is the actor representing the model structure as ODE’s. “Model Parameters” is the output of the code, in the form of model parameters for the chosen model. The file uses a MATLAB code style format such that it is directly readable by MATLAB, and also readable by humans. Finally, the actor “RK4” represents the solver used to compute the time evolution of the model.

The principle behind using an optimization algorithm to identify parameters of a model is discussed in detail in chapter 4.11. In short, the optimizer algorithm uses trial and error to “guess” the model parameters. Each guess is simulated and the results compared with known reference data. Inputs are also measured on the real process, and those values are used as input to the model. An objective score is computed for each parameter “guess” based on how well the simulation fits the known references.

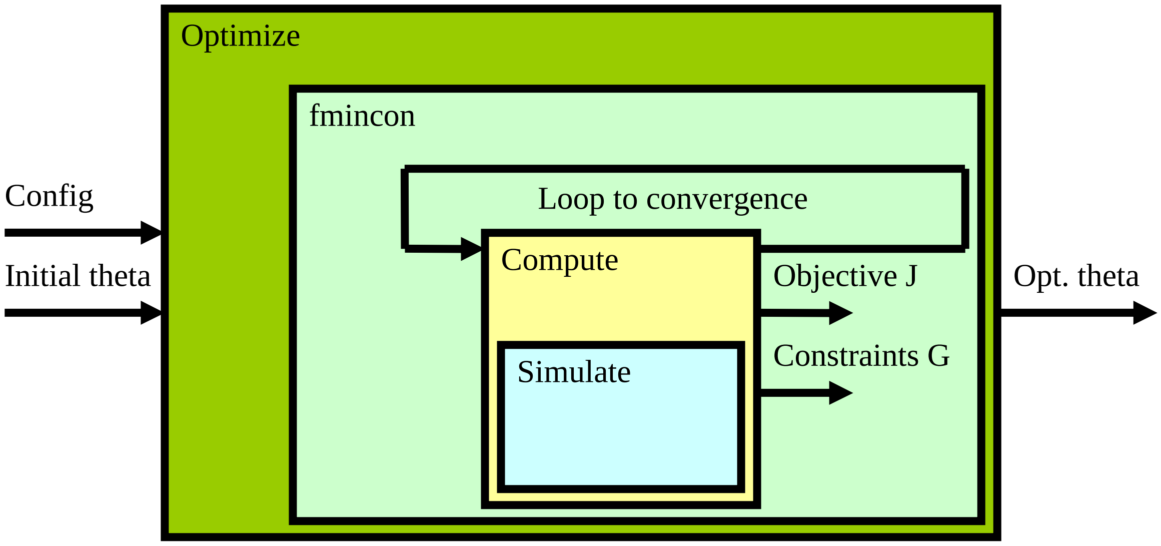


Figure 49 - MATLAB Parameter identification overview

The software that implements this concept consists of three main functions, Optimize, Compute and Simulate. Optimize uses “fmincon”, a MATLAB built in function, which runs the actual optimization algorithm. Figure 49 shows how these software functions depend on each other to accomplish the task of parameter identification.

In addition, there are multiple models implemented, as well as the RK4 solver function. Some additional functions for plotting and exporting results to CSV files are also implemented as a support framework. The functionality in Optimize and Compute is discussed in detail below.

The Simulate function simulates the model over a specific number of timesteps, sampling time and a given initial state. The same simulation implementation is used in both MATLAB and c#, which is discussed in chapter 4.9, 4.10, 5.4 and 5.3. The simulation is in principle an iterative calling of the dxdt evaluation to advance the simulation for a given number of timesteps.

5.6.1 Optimize

The Optimize function takes two kinds of inputs, firstly configuration settings such as timestep length, inputs to the model, reference data, etc, and secondly an initial guess for parameters (theta). The optimize function is simply tasked with calling fmincon with appropriate callback functions for computing the objective function and constraints.

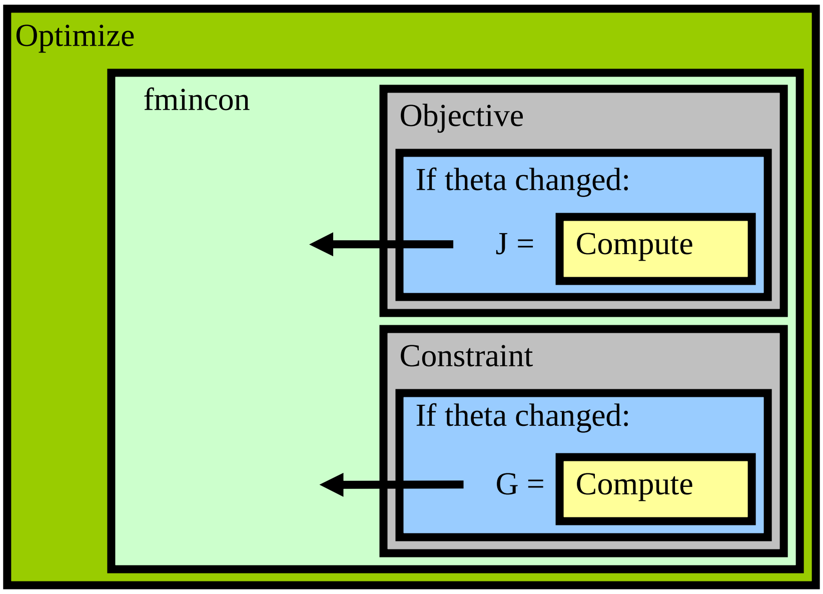


Figure 50 - Optimize function in MATLAB with internal callback functions

The fmincon function in MATLAB has two callback function arguments, as shown in Figure 50, one for the objective and one for constraints. In optimization there are two types of constraints [9], equality and inequality. For parameter identification, only inequality constraints are used. Since both evaluation of constraints and computation of the objective require the model to be simulated, the optimizer contains two local functions that are given as callback functions. Both of them call on the Compute function to evaluate the model, if and only if the “guess” for parameter vector “theta” has changed since the last time Compute was called. If theta has not changed, the resulting objective score and constraints evaluation vector will not be changed by calling Compute either. This is simply an issue of computational efficiency, avoiding simulating the model twice for each new parameter vector “theta” that is evaluated.

Optimize is an interface to fmincon, which resolves the problem of requiring to simulate the model twice, once for the objective and once more for the constraints. It also configures the fmincon parameters, such as what type of algorithm to use. The fmincon function will come up with guesses for model parameters, in the theta vector, based on the chosen optimization algorithm, “Interior Point” in this project. For each guess, fmincon calls the Objective and Constraint callback functions. The return value from Objective is the objective score J, while the constraints are returned as a column vector G. For fmincon the standard form is that the values of G must all be negative or zero. A positive value constitutes a violation of a constraint. The return value from Objective is the value the fmincon optimizer tries to minimize.

5.6.2 Compute

The compute function is tasked with calling the simulate function to evaluate a given set of model parameters and comparing the results to a given reference and returns two values J and G. Actually, the function returns three values, a second constraint vector for equality constraints H is demanded by fmincon, but H is always returned as a null matrix, since equality constraints are not used in parameter identification. The simulated result is compared with the known reference and the Root Mean Square error is computed as:

$$J = \sum_{k=1}^{nx} \sum_{i=1}^{N} \left(x_i^k - r_i^i \right)^2$$

Here, nx is the number of states, which are temperatures in the case of buildings. Further, x is the simulated states, r is the measured states and N is the number of samples in the training data set. For each state in the model, the error is squared for each sample and summed up. This is repeated for all states, and the results are summed up. This gives the objective function used to evaluate the parameter guesses provided by fmincon. By minimizing J, the optimal parameters are identified. The parameters are optimal in the sense that they minimize the square error between the model prediction and the known true values of the real building.

The second returned value from Compute, shown in Figure 49, is the constraints. The syntax of fmincon requires that the inequality constraints be evaluated and returned as a column vector where all elements are required to be zero or negative. If any single element in the vector is positive, that constitutes a violation of the constraints.

In parameter identification, as suggested in [2], the constraints can be formed by demanding that all parameter guesses from the fmincon must be between 0.3 and 3 times the initial parameter given to the system.

In MATLAB code, this can be written as:

```
theta_rel = theta./theta0;
G = [  theta_rel - 3.0;
       0.3 - theta_rel ];
```

Figure 51 - MATLAB implementation of constraint evaluation

In Figure 51 an example code which sets up the G column vector according to the fmincon standard syntax, based on the parameter identification scheme using fmincon. The theta_rel vector will hold values for each parameter in theta divided by the initial guess for that parameter. The /. operator performs an element by element division. If all elements are less then 3, clearly the first block row of G will be negative. Similarly, if all elements are larger then 0.3 the second block row of G will also be negative. Hence, if both conditions are true, G will contain only negative or zero elements. This informs the fmincon algorithm if a constraint has been broken. What fmincon actually does with that information depends on the chosen algorithm, and is beyond the present project to discuss. Details on optimization algorithms can be found in the literature. The principle behind optimization algorithms and its use in parameter identification is discussed in chapter 4.11.

5.7 Heating time estimation, ON/OFF control simulation

An important part of this project is the estimation of heating and cooling time, as discussed in chapter 2.1. To achieve this, a simulated ON/OFF controller is required. The model by itself is capable of predicting the temperature in buildings, given an input of power supplied to the indoor air in the form of heat. When validating and calibrating models, this input is taken from measurement data. When simulating the building with a heater under ON/OFF control, the power to the heater is instead computed by a controller.

```
        if(T_b < Tsp - db)
              bHeaterOn = true;

        if(T_b > Tsp + db)
              bHeaterOn = false;

        Q1 = bHeaterOn ? Qheater : 0;
```

Figure 52 - c# code for ON/OFF controller

Similar controllers are implemented both in c# and in MATLAB. In c#, the code can be simply written as in Figure 52. The temperature T_b is checked against a setpoint T_{sp} +/- a deadband db. The heater state is then switched accordingly, if the temperature is above or below these limits. At each timestep the heater state, either on or off, determines what power is supplied from the heater to the air. This implements the controller as described in chapter 2.1.2 and specifically demonstrated in Figure 3. By using a deadband around the setpoint, the controller avoids rapid on and off cycles at the boundries. This would otherwise be caused by noise in temperature measurements.

```
    k_sp = 0;
    for i = k0:N
        if(k_sp == 0 && x(i,ctrl_index) >= Tsp)
            k_sp = i;
        end
    end

    Theat = (k_sp - k0)*dt/3600.0;
```

Figure 53 - Computation of heating time from simulated results

Actual computation of heating time can be done either manually by inspecting simulated results, or it can be done automatically by code similar to Figure 53. This code is only implemented in MATLAB. If the x data is noisy, a centered WMA filter should be applied first, to avoid noise spikes indicating a temperature at setpoint, but for simulated results this is typically not required. The variable “control_index” is the state that is under control, i.e. room temperature location in the x state vector. The timestep index where setpoint is reached is found in k_sp, while k0 is the timestep index of when heating cycle is initiated. N is the total number of samples. A simple for loop checks all simulated results from the start of the heating cycle until it finds a temperature at or above setpoint.

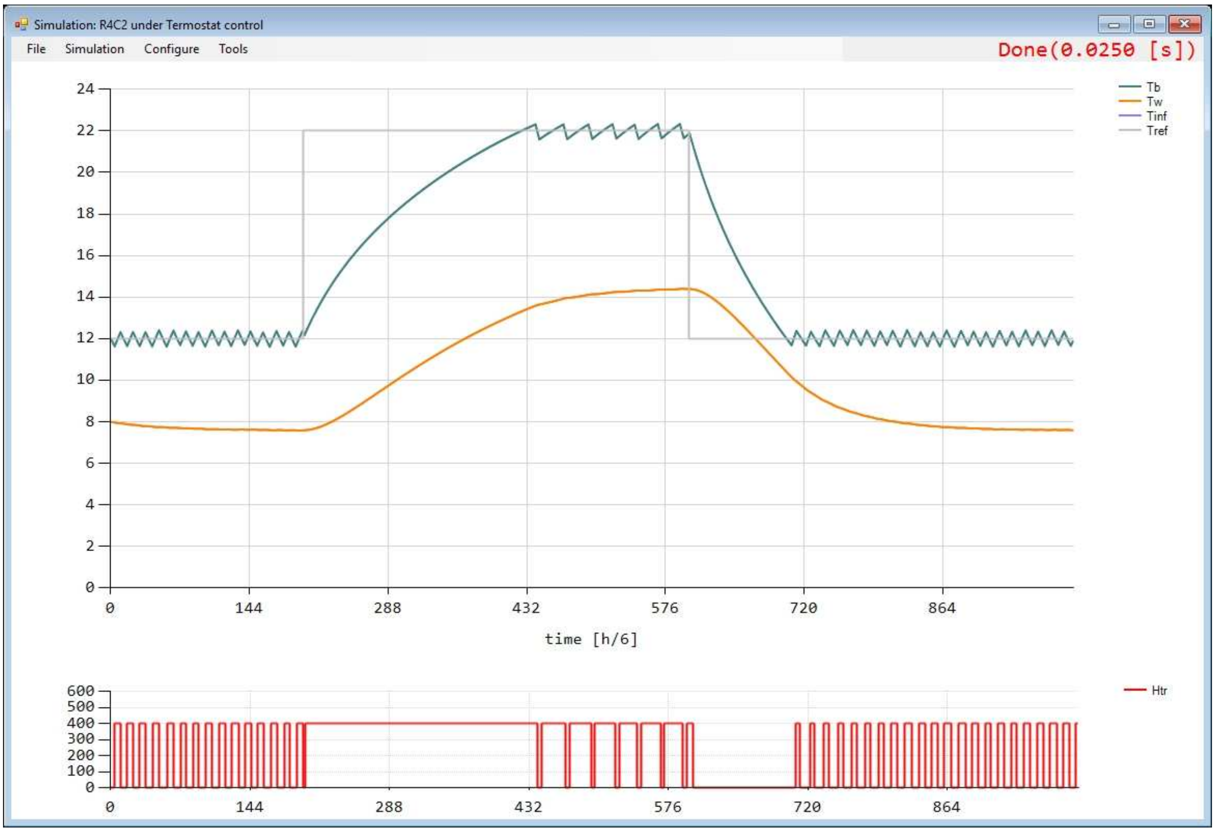


Figure 54 - Simulated ON/OFF control for heating time estimation

In Figure 54, the c# Simulation software is used to simulate the R4C2 model with parameters identified from the ”ByggeLab” data sets, using ON/OFF control to maintain a setpoint. As demonstrated, approximate heating time can be taken directly from the plot by inspection, or more accurately from exported data. Alternatively, an algorithm like in Figure 53 can be added to the c# software as well for automatic computation of heating time. In a control system, this would be the required solution.

6 Results

After implementing software, as described in chapter 5, based on the theory presented in chapter 2 and 4, a total of 22 separate cases of application of grey-box modeling where tested. In each case, named 1-18 for measured data and A-D for data simulated by the single-zone model presented in [1], parameters for a model is identified from data, and validated against independent testset data.

For all cases, an initial guess of parameters is used to start the process. Then, identification is run multiple times, manually adjusting the initial parameters, until a set of initial parameters for each building is found that does not cause the identification to run into the boundary conditions of each parameter. Boundries are 0.3 to 3 times the initial guess. This process takes some manual trial and error, but eventually a set of initial guess parameters is found that allows the optimization to find an optimal set of parameters to fit the given calibration data.

While the development of a simulation tool is an important part of this project, the results presented are all taken from MATLAB plots, as shown in Figure 1. This is a consequence of the parameter identification. Since models are validated as part of the identification process, all necessary plots are produced by MATLAB. However, simulations on the same model using the same parameters as identifies by the MATLAB optimizer code would give identical results, as shown in chapter 5.4.2.

In this chapter, results are presented in table/summary form. More details on results are found in Appendix B. Selected plots are also shown in chapter 7, together with discussions of points of interest in the results.

Note that RMSEP and RMSEC, as discussed in chapter 4.7, values are exported by the software with three decimal points, while the input data uses only one decimal point. The results are presented in the report in the exact way exported by software, but the analysis is carried out using only one significant decimal in the results.

Table 1 - List of all cases with data set, model and timeframe

Case#	Building	Model	Time frame	dt	Samples	Calibration	Validation
A	Simulation	R4C2	Simulated time (4000 samples)	6 min	4000	1 - 2000	2001 - 4000
B	-	R6C3	-	-	-	-	-
C	-	R7C3	-	-	-	-	-
D	-	R5C3	-	-	-	-	-
1	ByggLab	R4C2	04.12.2015 18:00 - 17.12.2015 00:00	10 min	1765	1 - 954	1018 - 1766
2	-	R7C3	-	-	-	-	-
3	-	R4C2	-	-	-	1018 - 1766	1 - 954
4	-	R7C3	-	-	-	-	-
5	-	R4C2	02.11.2015 17:18 - 17.11.2015 07:16	10 min	2100	1 - 1050	1051 - 2100
6	-	R7C3	-	-	-	-	-
7	-	R4C2	-	-	-	1051 - 2100	1 - 1050
8	-	R7C3	-	-	-	-	-
9	-	R4C2	02.11.2015 17:18 - 17.11.2015 07:16	2 min	10500	1 - 5250	5251 - 10500
10	-	R7C3	-	-	-	-	-
11	-	R4C2	-	-	-	5251 - 10500	1 - 5250
12	-	R7C3	-	-	-	-	-
13	-	R4C2	02.11.2015 17:18 - 30.11.2015 03:28	10 min	3950	1 - 2100	2201 - 3950
14	-	R7C3	-	-	-	-	-
15	-	R4C2	-	-	-	2201 - 3950	1 - 2100
16	-	R7C3	-	-	-	-	-
17	Cabin	R4C2	01.10.2015 01:00 - 25.12.2015 23:50	10 min	12378	1 - 6189	6190 - 12378
18	-	R4C2	-	-	-	-	-

In Table 1 the list of all 22 cases is presented with details on which models are used, as well as what data sets and timesteps are used. There are several noteworthy things in Table 1. Firstly, only two models are actually used on measurement data, R4C2 and R7C3. This is because there are no measurements on furniture temperature in any of the data sets which can be compared with the models predicted furniture temperature. Therefore R6C2, R6C3 and R5C3 are not usable with the available data sets. However, as a proof of concept, these models are tested against simulated data, with the exception of R6C2, which is not used in this project. It is however interesting as an intermediate model between R4C2 and R6C3. Based on [1], furniture is found to store significant thermal energy, so a model like R6C2 that only treats the furniture temperature as a potential divided between walls and room is not of particular interest, unless the capacitance is added like in R6C3.

Further, there are data from three sources. Cases A to D use the simulated data taken from simulations of the single-zone white-box model from [1]. The remaining cases use measurement data from two locations, discussed in chapter 2.5. Only two cases are using data from the “Cabin” data set, since these data was found to give unsatisfactory results, due likely to a large influence from solar irradiation, as will be discussed later.

In some cases, a small part of the data is skipped when defining calibration and validation ranges. This is the same as classifying the skipped data as outliers, hence dropping them from the data sets. This is done because the data in the skipped ranges does not fit the excepted dynamic, and thus indicate an anomaly, e.g. that a window or door was left open for a time. An example of this is found in the December data set, samples 955 to 1017. The most interesting results are found in the November data set on the “ByggeLab” location. Outliers for this data set are treated in a separate chapter.

For the simulated data sets, ventilation is on (0.77m^3/h) for the first half of the data set. The difference here between training and validation data sets is that the outdoor temperature is increased from -2.2°C to +5°C. For all measured data, “Cabin” and “ByggeLab”, there is no ventilation system.

Many of the cases include predictions of other states then T_b, i.e. the building air temperature. For a control system based on model predictions, T_b is the most important state to predict. Other states are auxiliary, i.e. important for the quality of predictions of T_b. As such, qualitative comparison of models is mainly based on RMSEP of T_b.

6.1 Summary of results

A summary of the results is presented in table form. For each case, the chosen model is identified from data, and validated. This produces several performance criteria as previously discussed in chapter 4.7, 4.11 and 5.6, in forms of RMSEC and. RMSEC described how well the model is able to fit the calibration data set, while RMSEP describes the models ability to predict a measured state using independent data.

Table 2 - Summary of results, performance criteria for models

Case#	Model	RMSEC	RMSEP T_b	RMSEP T_w (inner)	RMSEP T_w (middle)	RMSEP T_s (furniture)
A	R4C2	2.949	1.871	1.053	-	-
B	R6C3	4.681	1.420	2.225	-	2.824
C	R7C3	2.367	1.737	1.020	1.058	-
D	R5C3	1.000	1.273	0.737	-	-
1	R4C2	1.012	1.869	2.947	-	-
2	R7C3	1.383	1.994	2.826	2.426	-
3	R4C2	0.951	1.510	2.424		-
4	R7C3	1.014	1.400	2.259	2.092	-
5	R4C2	0.602	0.576	2.150	-	-
6	R7C3	0.685	0.639	1.923	1.912	-
7	R4C2	0.837	0.590	1.334	-	-
8	R7C3	1.014	0.645	1.182	1.241	-
9	R4C2	0.600	0.575	2.152	-	-
10	R7C3	0.681	0.637	1.925	1.912	-
11	R4C2	0.836	0.586	1.335	-	-
12	R7C3	1.009	0.641	1.243	1.183	-
13	R4C2	1.101	0.795	0.744	-	-
14	R7C3	1.362	0.811	0.543	0.558	-
15	R4C2	0.815	0.816	0.992	-	-
16	R7C3	0.869	0.847	0.928	0.881	-
17	R4C2	1.662	2.557	2.981	-	-

From Table 2 there are several interesting results to be discussed. Some general remarks are given here, while particularly interesting results are discussed in chapter 7. The details of all cases are found in Appendix B. Note that case 18 is excluded from the results above, since the optimizer fails to identify parameters for this case. Case 18 is further discussed in a later chapter.

The purpose of this project is the prediction of heating times in buildings, where the goal is to reheat a building to a comfortable temperature within a predicted time. In that respect, a prediction error of 1°C is quite adequate.

Not all the presented models can be used in a control system. The best cases from Table 2, cases 5 and 7, give particularly accurate predictions. It is probable that these models would show satisfactory predictions for use in the type of predictive control system discussed in chapter 2.1.2, as will be demonstrated and discussed below.

Table 2 shows that the RMSEP values as are in range 0.5° - 2°C. The sensor accuracy is around +/- 1°C [25] which shows that these results may be considered reasonably good predictions. However, RMSEP is a measure of prediction error over all samples. With respect to model predictions in a control system, as described in chapter 2.1, a better measure of a models ability to predict temperature would be either the error at the end of a heating cycle or the heating time estimation error. Here, RMSEP is used for qualitative comparisons.

6.2 Removal of outliers in November Data

As discussed in chapter 4.5.5, removal of outliers is a partially manual operation. Special attention to outlier removal is given to the November Data because these data turns out to give the most interesting results when used in grey-box modeling, in particular for estimation of heating and cooling times. Hence, some extra care in removing anomalies in the data is justified. This data set contains experiments where the electric heater is used with thermostat control to keep the room temperature approximately constant over a long time period. The experiment results are interesting for the purpose of predicting heating and cooling time of a building.

During the experiment, a person has to physically enter the building to change the thermostat setting. This causes an influx of cold outside air, which disturbs the temperature readings indoors. This is an artificial effect, not caused by building thermodynamic behavior. Clearly, this event will occur in a real building while in use, but for the purpose of identifying building parameters, particularly from a time limited data set, this type of anomaly should be removed e.g. by approximating what the temperature would have been if building was not manipulated by people.

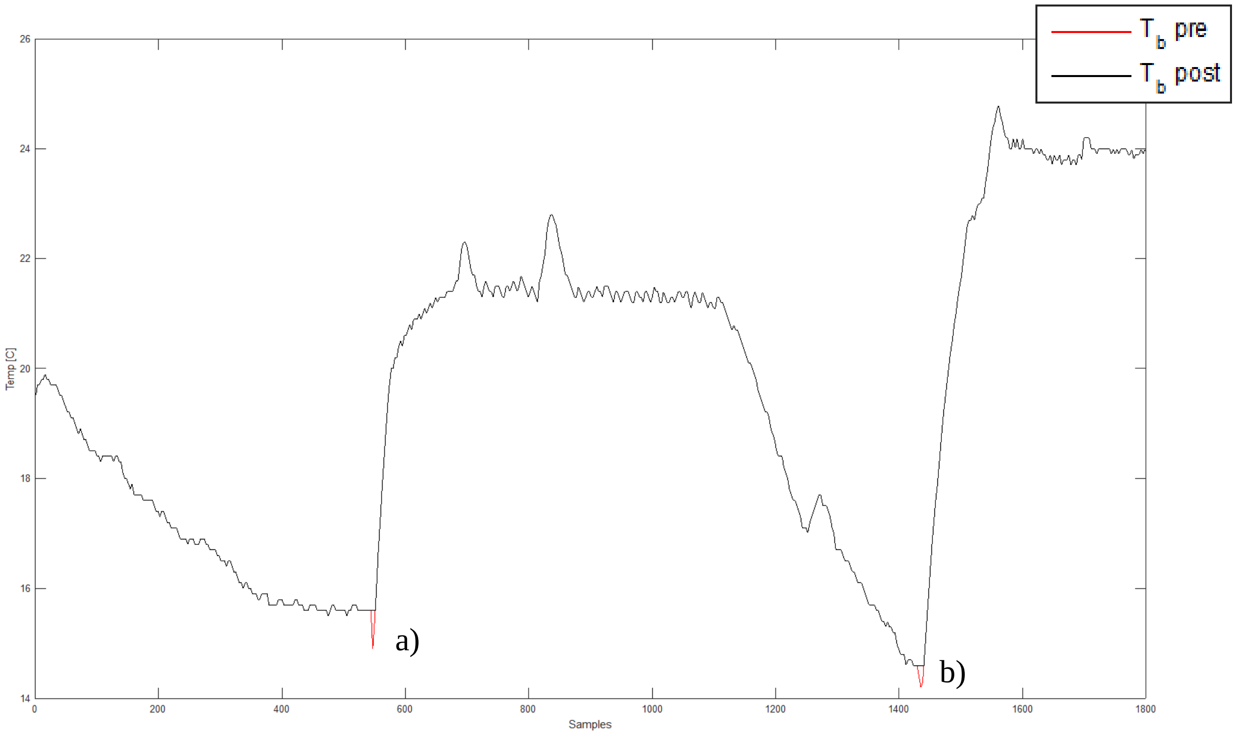


Figure 55 - Manual removal of outliers in November data set

After manual inspection of the data, it was found that the energy consumption of the heater starts to increase at step 544, because the thermostat has been changed. In the next six samples, the temperature declines, as shown in Figure 55 at point a), where the red graph is original data and black is after outliers removed. To remove the anomaly, T_b for samples 545 to 550 is set to 15.6°C over the whole range, as that is the average value before and after the anomaly occurs. Timestep in the data set is 10 minutes, so removing six outliers constitutes one hour of data. A similar procedure is applied to samples 1430 to 1440, at point b), where Tb is set to 14.6°C.

For the third step, which is not shown in the plot of Figure 55, occurring around sample 2984, there is no such anomaly. This is likely because the person entering the building closed the door faster thus releasing less heat. Alternatively the lack of a drop in temperature could be because the building at this time is around 6°C, thus less heat escapes when cold air is entering the building. Since at step 3 the outside temperature is around -5, while at step 1 outside temperature is around 10C and for step 2 around 0, the temperature difference between inside and outside is comparable for all three step changes/heating cycles. Hence it is more likely that the person entering the building kept the door open for a while at step 1 and 2, probably to cool the building further before starting a new heating cycle. The data is not specifically recorded for the purpose of parameter identification, but the experiment was performed as part of the ongoing research project at Telemark University Collage for another purpose.

For these data sets the assumption that the person who changed the thermostat setting kept the door open is not verifiable. A useful extension to the logging system would therefore be to include measurements on doors and windows to detect when they are open and closed. Keeping a window or door open essentially sets the thermal resistance of the building envelope to something closer to zero, forcing abnormal thermal behavior in the building. Since the open door/window will dominate the thermal behavior, this causes an anomaly, a set of outlier samples, in the data. A finished control system based on the principles described in this project could probably detect such events as outliers, and alert users if e.g. a window or door is left open. Another such event could be a change in insulation quality for a part of the building envelope, e.g. a degradation of insulation or windows, causing such kind of anomalies to occur.

7 Discussion

The results of this project consist of 22 cases with differences in models, data sets and sampling interval. The results are presented in table form in chapter 6 and the detailed results of each case are given in Appendix B, with relevant results repeated as needed in this chapter. The results of each case are not of particular interest, beyond identifying which models is likely to work in a control system. Those models hold great interest on a case by case basis. The discussion on these results is best given by comparing different cases to see what effects the results. The discussion is focused on how the grey-box modeling approach, based on RC equivalent thermal network models, can give usable predictions of a buildings thermal behavior. Further, it is interesting to see what parameters in the data and modeling process affects these results.

In this chapter, several interesting points of discussion are presented, based on the case results from chapter 6. Where applicable, a conclusion on each discussion is given at the end of each sub-chapter.

7.1 Cognitive model development, simulated data

The concept of parameter identification was first tested on simulated data. The three models from reference literature in [2] gives RMSEC values in the range 2° - 5°C which are considered unsatisfactory prediction errors. After inspecting the trends in the simulated data, it was found that a very slow dynamic is presented in the model. There is an element that takes a long time to change temperature relative to the rest of the system. The slow dynamic, likely introduced by the furniture sphere model, is further discussed in [1]. The thermal energy storage capacity in this construct is large, hence the R5C3 model was constructed to better fit with the simulated data, as discussed in chapter 4.4.5.5. Indeed, as Table 2 shows, this model improves results significantly, with an RMSEC of 1.0°C. The most important performance characteristic of these models, RMSEP for the building temperature T_b, is also significantly improved.

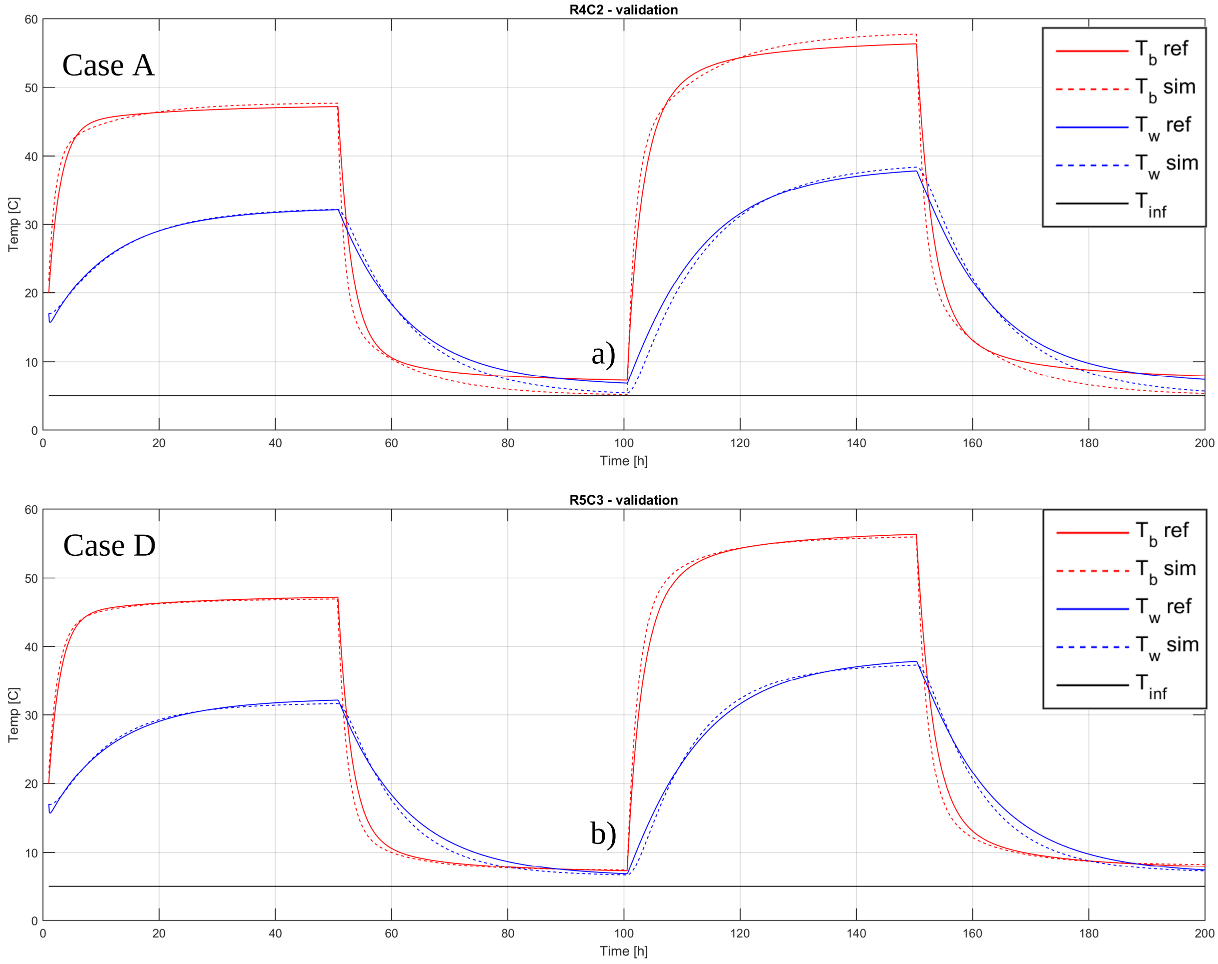


Figure 56 - Comparing case A - R4C2 (top) and D-R5C3 (bottom)

Comparing case A and D, in Figure 56, validation plots show that the addition of thermal energy storage, with a resistor to limit the flow of energy into building air, increases the models ability to predict the simulated data. For case A there is a clear trend that the model (stapled lines) under predicts the temperatures when cooling, shown in point a), and over predict when heating, due to a lack of stored energy in furniture. For case D, this is improved by including a RC model of the furniture, as shown in point b).

7.1.1 Conclusion of models from simulated data

While the results of fitting one model to the simulation of another model, is not practically useful, these results show the advantage of using the cognitively developed thermal network models. These models can be derived, modified and improved, simply by intuitive understanding of the process, rather then through complex mathematics and physics. This is an advantage of the grey-box modeling technique used in this project. The ability to make model structures that fit both data and buildings based on cognitive knowledge is a powerful approach to modeling.

7.2 Range of input and states in training and calibration data

The next cases of interest are cases 1 and 3. Alternatively cases 2 and 4 show the same, but with one more state, hence cases 1 and 3 illustrate the point better.

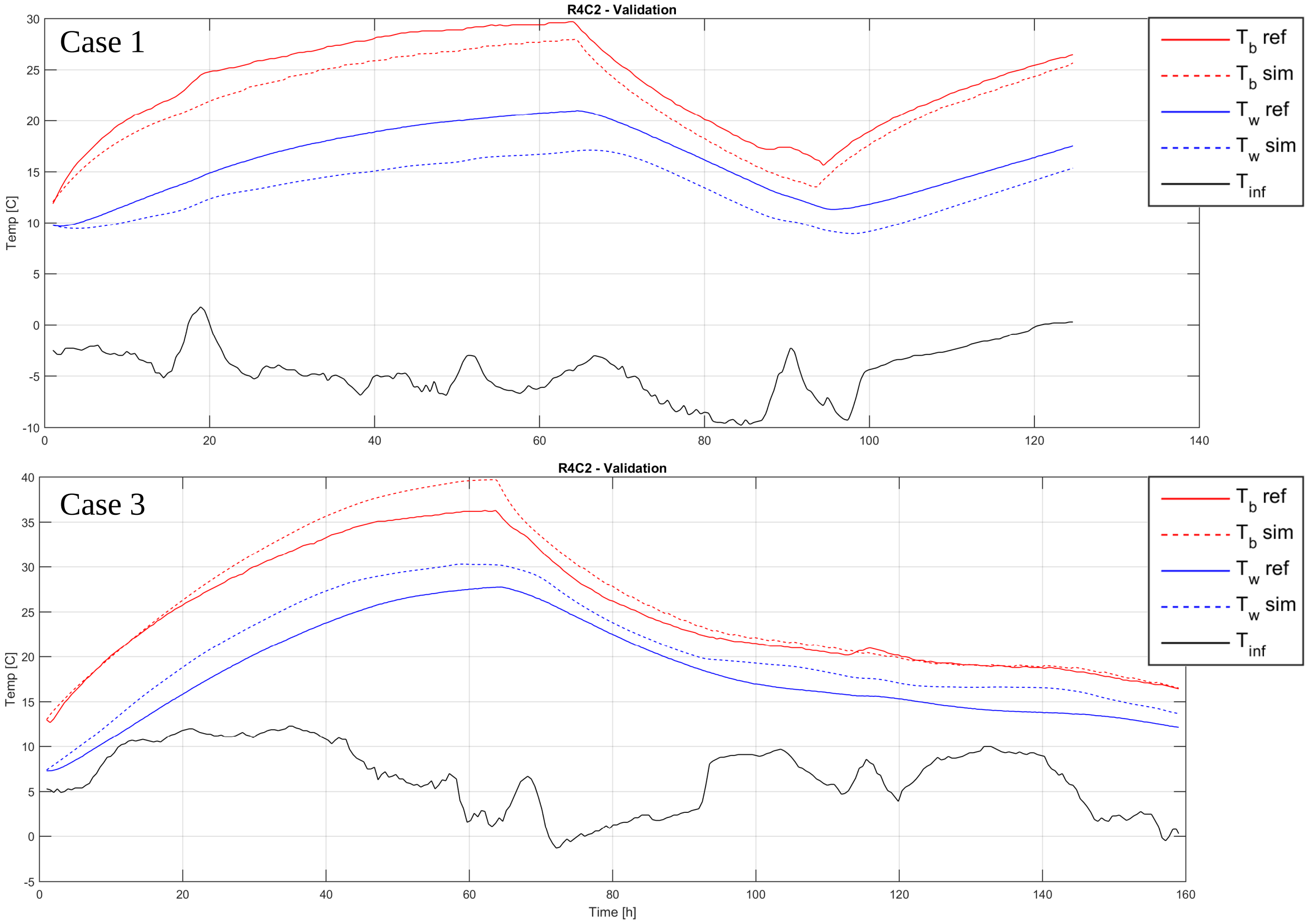


Figure 57 - Comparing cases 1 (top) and 3 (bottom)

Comparing the validation results for cases 1 and 3 in Figure 57, it is interesting to note that for case 1 the model under-predicts the reference, while in case 3 the model over-predicts. In both cases the model and data is the same, but the training and validation sets are swapped. This means that the training data for case 1 is the validation data for case 3, and visa versa. The RMSEC and RMSEP values for both cases are comparable, as shown in Table 2.

The reason for this behavior is seen in the outdoor temperature T_{inf}, plotted as a black line. For case 1, the training data is from first week of December, while for case 3 it is mostly the second week. In case 1, the outdoor temperature is in the range -5 to 0 degrees, while for case 3, the temperature has increased to 0 to 10 degrees. Since there is a mismatch in input ranges between training and validation sets, the model fails to predict the validation data. The prediction error depends on whether the conditions in outdoor temperature in validation data are higher or lower then the training data, hence the prediction error is of opposite sign in case

3 with respect to case 1. This illustrates a very important point when working with any type of empirical models. It is unlikely that a model will give good predictions on data from conditions not present in the training data. This is discussed in theory in chapter 2.5.3, and confirmed by the results of case 1 and 3.

A further point of interest in the discussion of training data, beyond the bespoke requirement of having appropriate matching between the conditions in training data and the data on which the model is expected to predict behavior, is the need for a certain degree of dynamic behavior in the data. Since the models contain capacitance, giving the models internal states, there is a need for variation in these states, in order to identify appropriate values for the capacitors. If there are no dynamic behavior, the RC models could be reduced to their steady state equivalents by using familiar circuit theory [23], where a fully charged capacitor is replaced by an open circuit and only the resistors remain. This illustrates the need for dynamic behavior in the data. If the states are constant the capacitor values can not be identified from the training data.

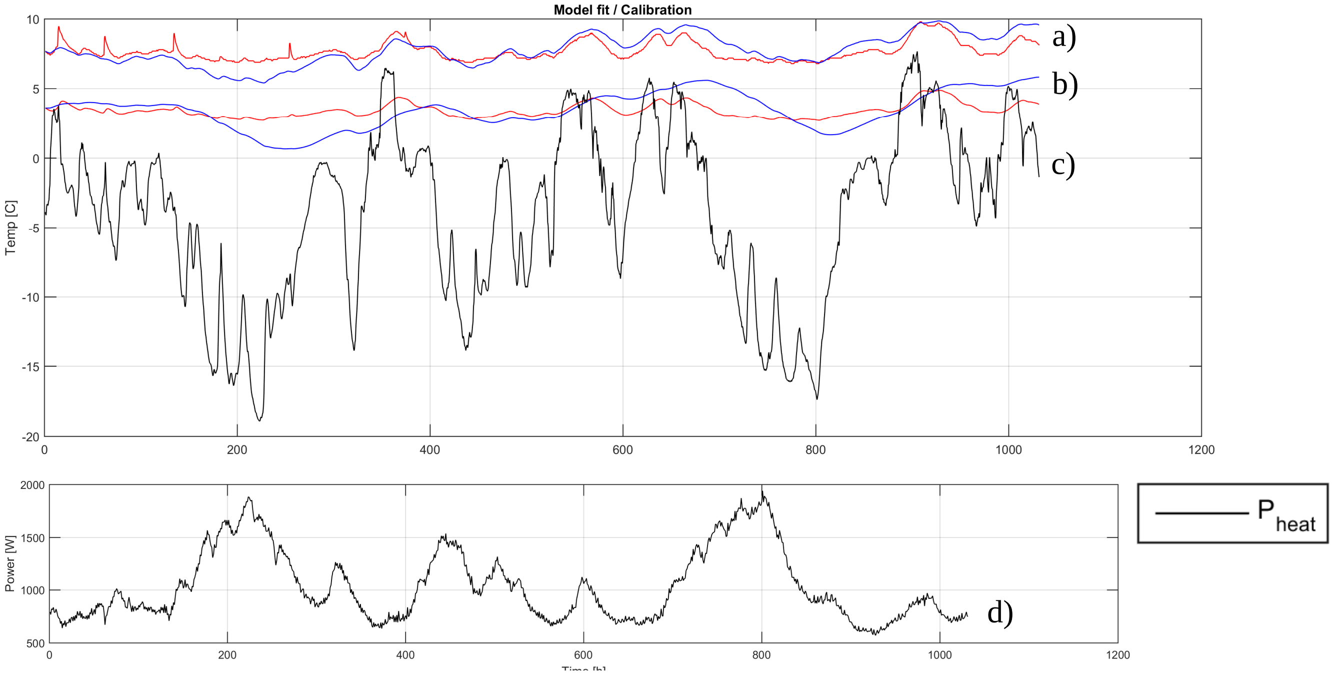


Figure 58 - Case 18 calibration data, red - reference, blue - model, top - T_b, lower - T_w

Figure 58 shows the calibration results from case 18. The temperature states T_b in graphs a) and T_w in graphs b) are kept approximately constant for the entire data set by the heating system in the cabin. Red graphs are the reference measurements while blue graphs are model predictions. Even with large variation in the outdoor temperature, shown in graph c), and the supplied power shown in lower plot marked d), there is not enough dynamic behavior in this training data to identify the value of the capacitors. Since the capacitors can not be identified properly, the optimizer fails to find the resistances as well. Capacitors are driven to the upper boundries (3x initial guess). From case 17 the initial guess for parameters is fairly certain, such that it can be concluded that capacitor values for case 18 should not be as high as the optimizer result suggests. This case illustrates the second requirement for good training data, e.g. variation, or dynamic behavior, of the reference data for model states.

7.2.1 Conclusion of required ranges in inputs and states for measured data

Based on the results in case 18 and comparison of case 1 and 3, two things are found to be important in regards to the type and amount of data required for training of models. Firstly, the input conditions to the model must span the range of inputs that the model is expected to be able to predict. Secondly, there must be a significant amount of dynamic behavior in the states, for the optimizer to be able to identify the dynamic elements of the model. Based on other cases in this project, the dynamic behavior does not have to be as extreme as in case 1 and 3 (10° to 30°+ C), but steps of around 10°C seems to be acceptable. These results agree well with the reference literature in [2] where seasonal variations is found to have an impact on the results, and further that identifying model parameters from winter data gives better results, due to a heavier load on the heating system.

7.3 Length of data sets, number of samples

The next point of interest to discuss is the effect the number of samples, or rather the length of time represented, in each data set has on the model calibration and validation. Cases 5 to 8 are similar to cases 13 to 16, except that the latter use all of November, rather then just the first half. The entire data set for case 5 to 8 composes one half of the data for case 13 to16, used either as training or validation data. Cases 13 to 16 can be seen as re-doing case 5 to 8 with the additional data.

Table 3 - Extract from results, comparing cases 5-8 with 13-16

Case#	Model	RMSEC	RMSEP T_b	RMSEP T_w (inner)	RMSEP T_w (middle)
5	R4C2	0.602	0.576	2.150	-
13	R4C2	1.101	0.795	0.744	-
6	R7C3	0.685	0.639	1.923	1.912
14	R7C3	1.362	0.811	0.543	0.558
7	R4C2	0.837	0.590	1.334	-
15	R4C2	0.815	0.816	0.992	-
8	R7C3	1.014	0.645	1.182	1.241
16	R7C3	0.869	0.847	0.928	0.881

Table 3 is a duplication of selected results from Table 2, but organized differently to facilitate comparison of cases. Based on the performance criteria, RMSEP and RMSEC, in Table 3 there is no clear pattern of improvement when introducing additional data. For some states, the results are improved, but in others the results are worse with larger data sets.

In chapter 2.5.2 a short discussion on the length of data sets for training is given. From the literature, it is found a large variation in the size of training data, in the range 4 - 60 days [2].

7.3.1 Prediction horizon

When discussing timeframe of data sets and predictions, it should be noted that the purpose of these models is the prediction of heating time for a building. In typical daily use, a building is unlikely to require many days worth of heating time. At most, some 2-5 days may be considered the upper limit of what a thermal behavior model is required to predict. In literature the term “day-ahead” predictions is often used in connection with building thermal behavior, signaling that prediction horizons of a single day is a typical length for model validation [3].

Some of the cases presented in Table 2 use validation data over a significantly longer time range. The reason the validation periods in Table 2 are longer then strictly needed is because data sets for calibration and validation is swapped around between cases. E.g. for case 1 first half of December data is used for calibration, and second half for training. In case 2, the data sets are reversed.

The RMSE values given in Table 2 are correct, given the length of the validation sets, but it is possible that shortening the validation period would give lower RMSEP values, given that the same calibration data is used. Since thermal behavior in a building is time dependant, a longer validation period is likely to give less accurate predictions then a short validation period.

Since the RMSEP values are typically in the same range as sensor accuracy, the results as presented here are considered good enough, without shortening validation sets to lengths similar to the expected prediction horizon in a control system.

7.3.2 Conclusion of length of data sets

Introducing more training data, does not necessarily improve prediction results. This suggests that the time frame of the training data is not critical to the results, but rather the range of input values present in the training data is, as discussed in 7.2. As long as the training data covers a sufficient length in time, necessary for the dynamics of the system to be represented, further data does not improve results. The validation results may be improved by using a data set with length closer to the assumed length of predictions required in a control system.

7.4 Timestep length

The effect the timestep length has on results can be investigated from the cases in Table 1, by looking at the effect of reducing the timestep in the resampled data. By reducing timestep dt down from 10 min to 2 min, the number of samples is increased by five.

Comparing cases 5 to 8 with 9 to 12, e.g. 5 with 9, 6 with 10 and so on, looking at the list of performance criteria in Table 2, it is shown that reducing the timestep does not improve the results. Both model fit with training data and the models ability to predict independent test data is the same for all comparisons. In fact, the result numbers are identical for each of the four comparisons, e.g. case 5 has an RMSEP for T_b of 0.576 while case 9 give 0.575. Any differences can be attributed to random variations, sine RMSE values are computed by the software with three decimals but only one significant decimal is used in the data.

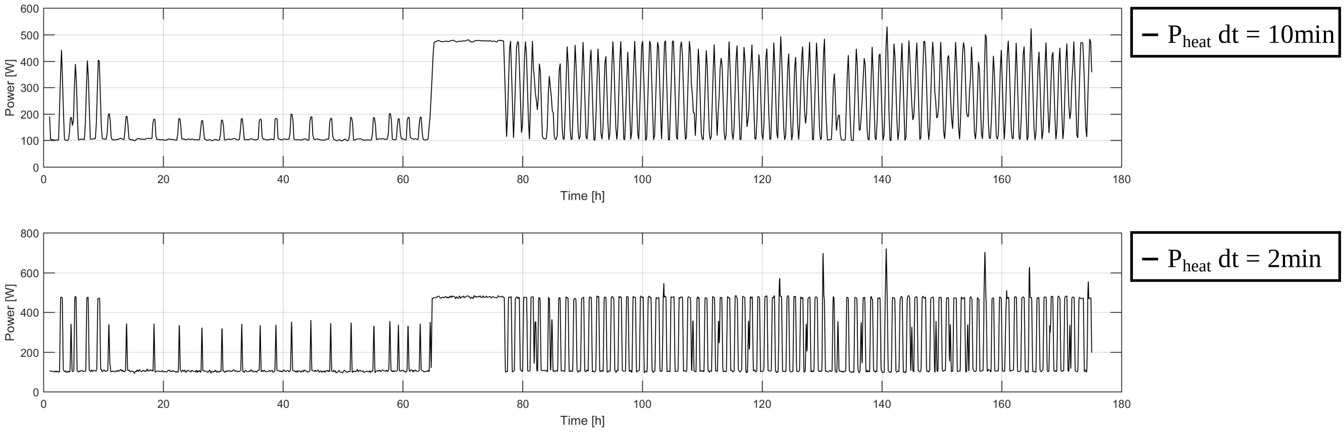


Figure 59 - Comparing power consumption data in case 5 (top) and 9 (bottom)

The main concern when choosing the timestep length is that it has to be short enough to capture any dynamic changes in all the input data. The Nyquist-Shannon sampling theorem states that sampling rate should be twice that of the highest frequency in any sampled signal [10, 13]. This is a requirement for the logging system, but also for the resampling operation. In the data sets of this project, power consumption is by far the fastest changing data.

Since the “LogFile Converter” software can filter the data before resampling, high frequency information such as that found in the fast changing power consumption data can be filtered out [10]. The logging system records power consumption for ByggeLab at dt = 30 sec, such that filtering data in the pre-processing allows smoothing out the signal. By this process, the resampling can be done at a lower sampling frequency, since the filtered data has a lower maximum signal frequency then the original raw data. This process is known as oversampling [10].

In Figure 59, the plots of power consumption data for case 5 and case 9 after processing are shown. The top plot uses a resampling timestep of 10 min while the lower uses a timestep of 2 min. Just by visual comparison it can be seen that the data in case 9 contains significantly more details in the power consumption plot, i.e. sharper edges giving higher frequencies in the information spectra [10]. It is therefore interesting to note that the slight distortion introduced by the filtering and resampling operation in case 5 gives identical results to case 9.

7.4.1 Conclusion of timestep length

Based on the results in Table 3 reducing timestep length from 10 min down to 2 min does not provide any improved prediction results. Based on plots of data, it appears that 10 min resampling operation distorts power consumption data, but this has apparently no effect on the modeling results. This is likely because the average supplied power, over any time interval, to the building is the same for both datasets. Increasing the sample count by five times, increases the time the optimization takes to converge from to 0.6 min up to 2.1 min, for the R4C2 model. For R7C3 the time to convergence goes from 1.2 min to 4.9 min with the shorter timestep[1].

A convergence time of less then 5 min for both models is considered acceptable. Retuning the model is not a real-time operation, i.e. has no specific requirements on completion time. This operation will typically not be run often, maybe once pr day, in a worst case scenario, but retuning models on a monthly basis is a more likely scenario [2].

7.5 Redundant RC model parameters

Due to a lack of measurement data, redundancy in parameters can arise. Parameters are redundant in the sense that two parameters together describe the same part of the model. This is the case with R7C3 model. The model introduces the R_s resistor in series with R_b, and the subsequent addition of the T_s node. This addition models solar irradiation heating light furniture in the building, as discussed in chapter 4.4.5. There are no measurement data that can be used as reference for the temperature T_s and therefore the optimizer will have a problem when identifying R_s and R_b. Only the sum of these two will affect the results without a reference for T_s and the optimizer has in fact an infinite number of solutions. Increasing R_s and decreasing R_b with the exact same amount gives identical objective scores in the optimization when T_s has no affect on the objective. This forms a flat region in the objective function that the optimizer tries to minimize, i.e. gives infinite number of equally acceptable solutions in this region.

[1] The computer used to run the identification has a six-core 3.2GHz CPU using parallel computing in MATLAB

7.5.1 Conclusion of parameter redundancy

It is unclear to what degree this is a problem for the optimizer. However, the problem could have been fixed by simply reducing the model, i.e. removing R_s, such that the model structure fits the measurement system. For future projects, it is recommended to derive the model and measurement setup together, such that one reflects the needs of the other. Since this project is in part a study of the application of grey-box models, the structures where taken from literature, as discussed in chapter 4.4.5, which is then found not to exactly fit the measurement setup. A revised model may give improved results.

7.6 Model complexity

While only two of the models described in chapter 4.4.5 are used with measurement data, it is interesting to compare the results when increasing the model complexity. The difference between R4C2 and R7C3 is mainly the addition of a new state for the temperature in the middle of the wall, and also the R_s and R_e resistors which introduce redundancy due to lack of reference for T_s, as discussed in chapter 7.5.

Looking at the results in Table 2, comparing cases 5 and 6, the results are similar for both models, but the simpler model shows slightly better results. There is a trend that the more complex R7C3 model slightly degrades prediction performance in terms of RMSEP and also has lower fit to the model, in terms of RMSEC. In the reference literature [2, 3] the conclusion is also that increasing model complexity beyond a certain point does not improve the results.

7.6.1 Conclusion of model complexity

As discussed in chapter 7.5, there is a concern about the R7C3 model and redundancy of parameters, which could degrade performance. It is not clear if the degradation of performance of the more complex model is caused by the redundant parameter, or if it is a problem with overfitting [18]. Having more parameters allows models to fit structure in the training data that is not relevant for the model, such as noise. Since that the data sets have only one significant decimal, comparing RMSE values for models should also use only one significant decimal. The differences between the two models are small, on the order of 0.1°C.

A conclusion to draw from these results is that it is important that the model fits the actual measurement setup, not only the building. Further, the measurement setup should be designed together with a set of specific models that will be trained and validated against the measurement data. Because of this uncertainty, and the relatively small differences between results, it is not possible to conclude if the increased model complexity offers any advantages in the prediction results.

7.7 Solar irradiation

As discussed in chapter 4.6, solar irradiation heat gains are in this project considered as noise. This is not an optimal model, since solar heat gain is shown to be important for building thermal behavior [1, 2, 6]. However, due to lack of good measurement data in the data sets, the choice was made to ignore solar heating.

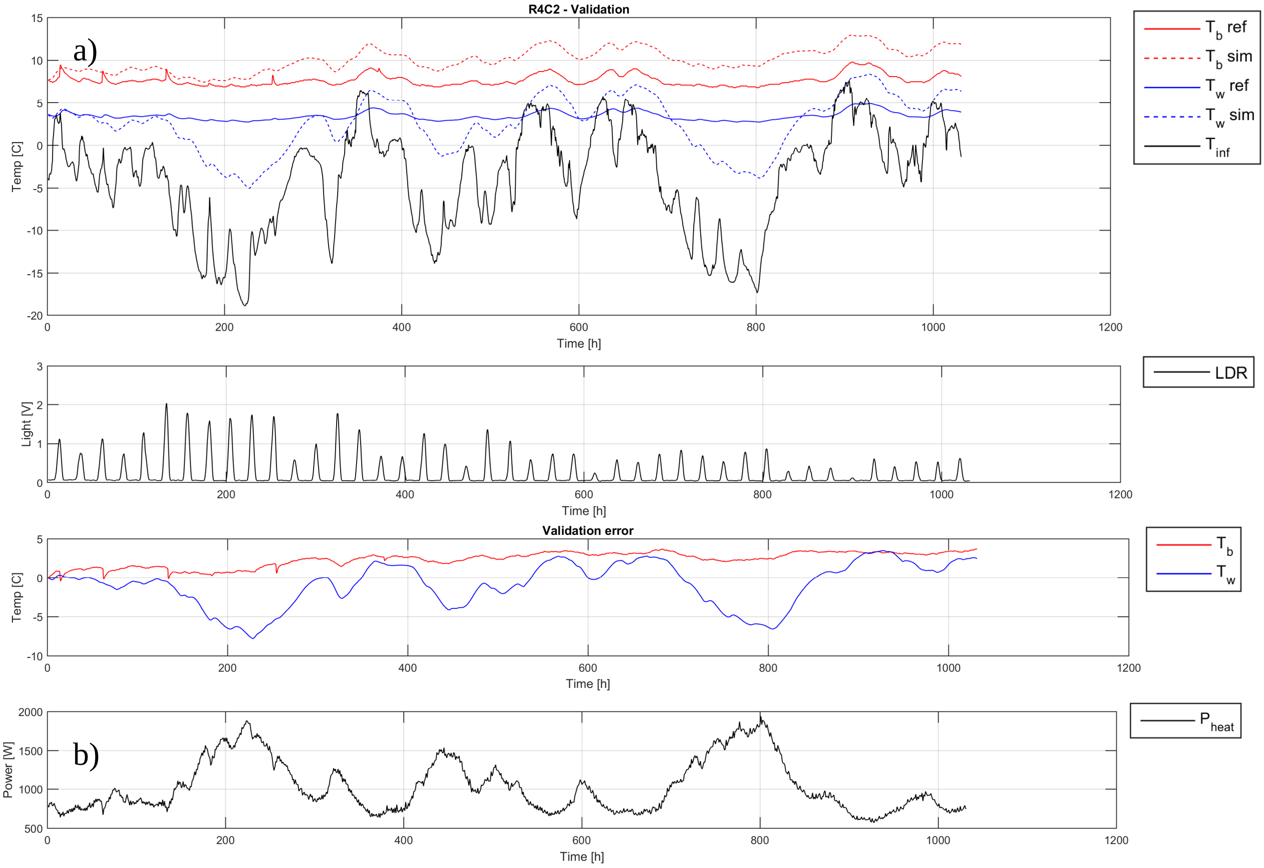


Figure 60 - Validation results for case 17

In Figure 60, the validation results for case 17 are presented. This plot shows some interesting behavior with regards to solar irradiation. Case 17 is the only usable result from the cabin data set. Case 18 failed to identify parameters, as discussed in 7.2. The cabin data set contains a Light Dependant Resistor (LDR) that measures light through a window, shown in the second plot from the top. These measurements could potentially be used to model solar heat gains, but is in this project used only to indicate the level of sunlight affecting the cabin.

By inspection of the plot in Figure 60, it is interesting to observe the spikes in room temperature found in the reference data (red solid line marked a). There is a sudden increase of 2° - 3°C in temperature, which coincides with the solar irradiation spikes in the LDR graph. Further, these spikes are only visible in the reference when the outside temperature is relatively high, e.g. around -5° to 5°C. If the outdoor temperature drops below -5, the spikes disappear from the data, indicating that the solar heat gains does not affect the room temperature anymore.

This can be explained by looking at the lowest plot, the power consumption, marked b). If the power consumption, i.e. the power used to artificially heat the cabin, is high, the solar heat gains does not increase temperature, but rather induce a drop in consumed power. In cold temperatures, there is significant power used to maintain the setpoint temperature of 7°C, thus reduction of power is possible when solar gains is added to the cabin. When little or no power is required to maintain setpoint, the system is unable to lower the heating power further and instead the room temperature rises.

It is also interesting that for the first three days of the data set, e.g. the first three spikes in the LDR plot, day 1 and 3 show higher light levels then day 2. This can be seen on the spikes in T_b as well. There are clear spikes in temperature for days 1 and 3, while day 2 show only a very slight increase in temperature during the day. This may indicate that the LDR measurements could be used for solar heat gains predictions.

7.7.1 Conclusion of solar irradiation

From these results, it is clear that solar gains play a significant role in the heating of the cabin, and thus can not be ignored when modeling the building. This is further evident by the results in Table 2, where the performance criteria for case 17 are worse then for any of the cases for “ByggeLab”. It is also interesting to note the correlation between the LDR measurements and the spike in Tb for the first three days. This may show potential for using LDR’s to estimate the solar gains, together with an identified model.

One important aspect of solar irradiation is that the effect can be introduced to the model indirectly. When the sunlight heats the cabin, it will typically also heat the outside air, causing increases in outdoor temperature, as shown also in Figure 60. This can lead the optimizer to computing a lower R_g resistance then what would be identified if solar heat gains where included in the data set. The optimizer will try to find parameters that, based on given inputs, can predict the states such as Tb. Thus, if outdoor temperature correlates with indoor temperature, i.e. they rise and fall together, the indication is that R_g should be low. However, if the correlation is caused by an external effect, i.e. solar irradiation, this correlation is not indicative of a low R_g. The model identification will give too low thermal resistance between indoor temperatures state T_b and outdoor temperature. The low value of R_g will constitute overfitting [18], and thus give a model with poor prediction performance in conditions with lower solar heat gains. Hence, inclusion of a model of solar heat gains is found to be an important improvement on the presented results, particularly for the data from “Cabin”.

7.8 Best case

As discussed the “Cabin” data set has some problems with respect to identifying a usable model. Hence, the “ByggeLab” is the only building with sufficient data for parameter identification. The best results are found in case 5 and 7, which use the same data set, only with validation and calibration swapped around between the two. The important variable to predict is T_b, the room temperature, which would be the variable that a control system is expected to keep at a certain level at specific times. Hence the RMSEP for T_b is the criteria on which the quality of each case is compared. By that reason, case 5 and 7 are found to be the best models. Results from case 9 is identical to 5, and 11 identical to 7, but cases 9 and 11 is derived from data with five times more samples, as discussed in chapter 7.4. Increasing model complexity, such as cases 6 and 8, gives less accurate predictions of T_b, as discussed in 7.6.

To select one case as the best one for in-depth study, a look at the validation data is needed. In case 7, i.e. with the first half of the data as validation data, a smooth step response is found. The temperature is kept at a relatively constant value of 15.6°C for a time, before the heater power is engaged and the building starts to heat up towards the new setpoint of 21.5°C. Further, case 7 shows slightly better RMSEP for the temperature T_w, compared to case 5, but also a slight increase in RMSEC. Despite slightly larger RMSEC, case 7 is considered the best case result among the 22 cases tested in this project. Therefore, closer inspection of case 7 is of interest.

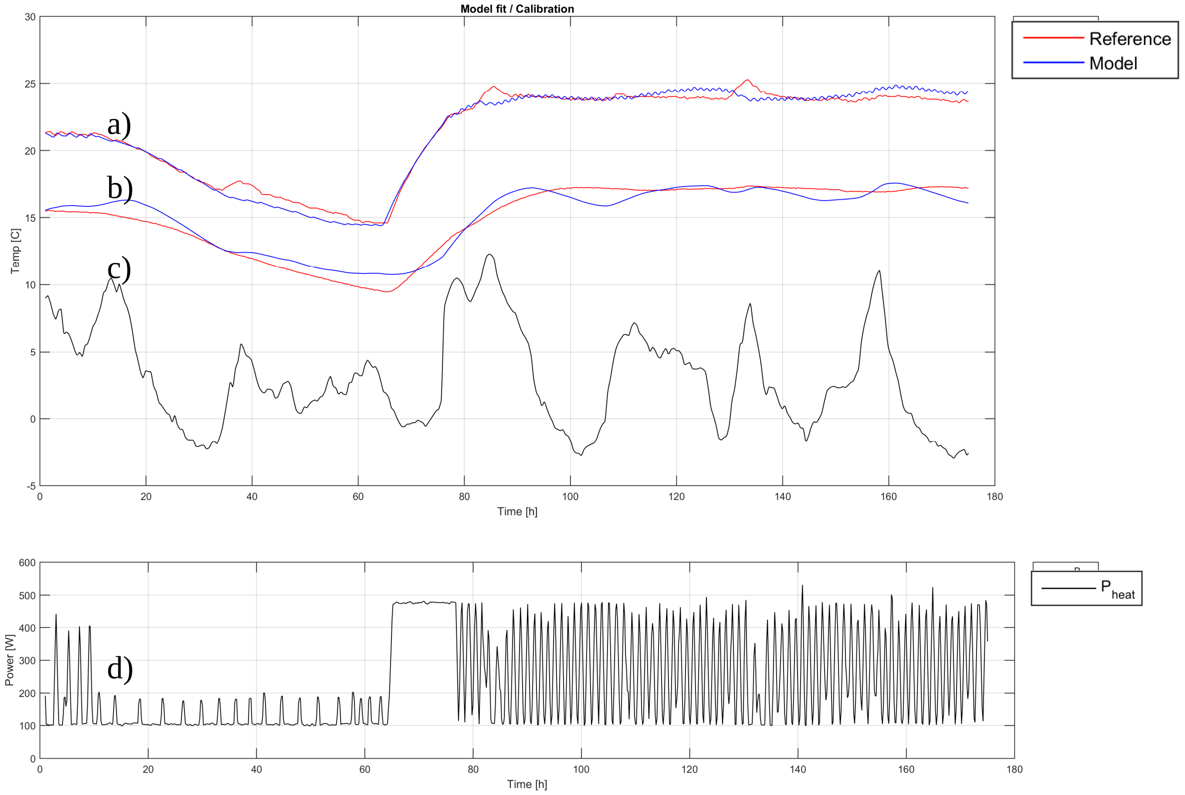


Figure 61 - Calibration plot for case 7

In Figure 61, the results of calibrating the R4C2 model to the training data for case 7 are given. The outdoor temperature in graph c) consists of values in the range approximately 0 to 10C. There is a notable feature around hours 60 to 80, where the temperature T_b in plot a, and also for T_w in plot b), is increased by adjusting the setpoint of the heater, as seen from the used power in plot d). After the temperature reaches its new setpoint, power is supplied to the heater in pulses by the ON/OFF thermostat controller to maintain the setpoint. In Figure 61 the top graph a) is the room temperature T_b, while the bottom graph b) is the wall temperature T_W. Just by inspecting the plot, it is shown that the model fits the calibration data to a high degree. Of particular importance is the models ability to follow the building temperature under a step response caused by increased power to active heating, which is demonstrated in plot a).

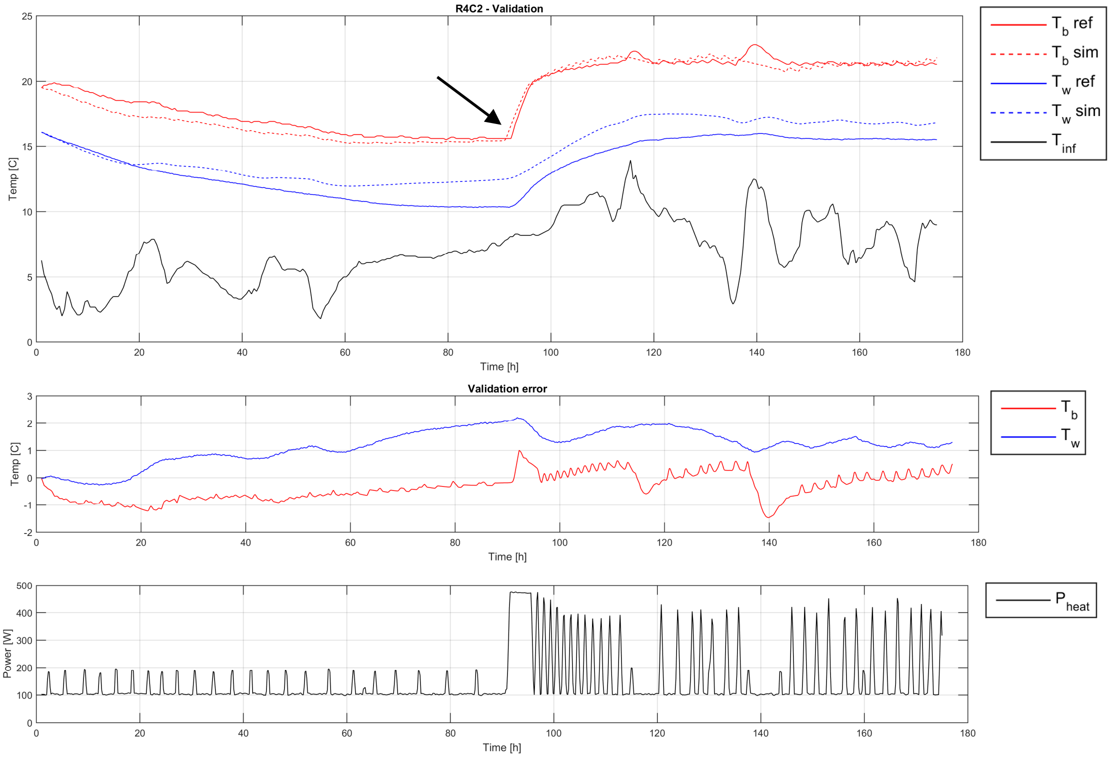


Figure 62 - Validation plot for case 7

The results of validating case 7 against data from the first half of November are shown in Figure 62. The outdoor temperature (T_{inf}) has the same range in validation as in calibration, 0 to 10C. Further, the validation data contain a step around hour 90 as marked by an arrow in the figure. By giving the model the same inputs as the true system, the results show a simulated building temperature close to the measured building temperature. For the purpose of predicting heating time, this is exactly what the models are required to do. Therefore, the results of case 7 are considered promising for the use of this modeling technique in a control system.

7.8.1 Error distribution

If the errors between model and real system, both for model calibration and validation, are just random fluctuations or noise, the error distribution will become a zero mean Gaussian curve [20]. This holds for pure empirical models as well as grey-box models. Hence, it is interesting to look at the error distribution for the results in case 7. Note that error distributions for all cases are shown in Appendix B.

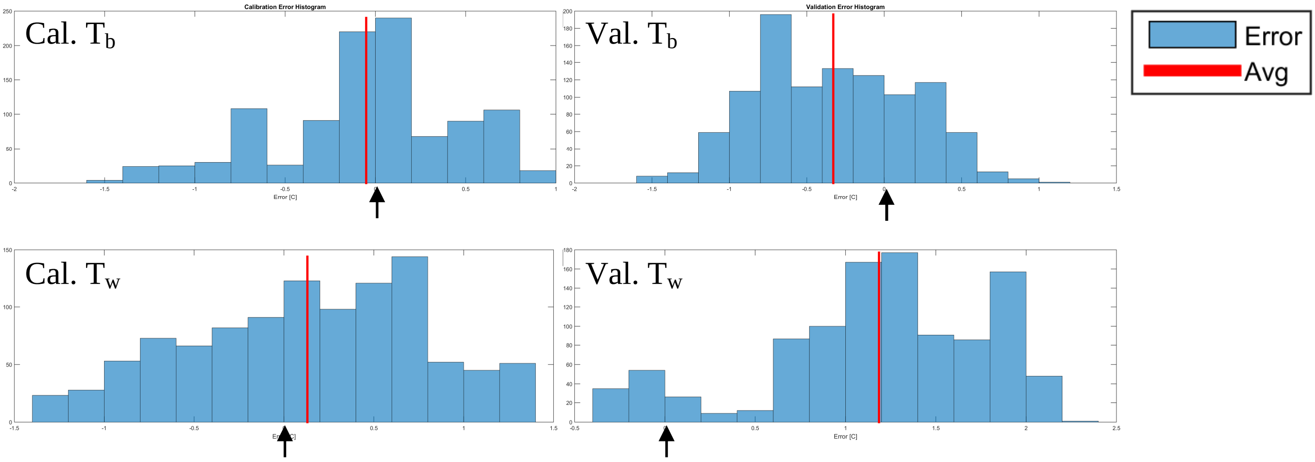


Figure 63 - Error distribution case 7, calibration (left) validation (right)T_b (top) T_w(bottom)

As shown in Figure 63, the error distributions for case 7 shows mean errors in calibration close to zero. For validation of T_b the plots also correspond closely to what is expected if the error is mostly caused by random noise for T_b, but for T_w there is some systemic errors that shifts the error mean up to around 1.2°C. Zero error is marked in the plots by black arrows.

7.8.2 Conclusion of choice of best case

The most interesting feature of these plots is however that the error distribution shows the errors in the same interval or range as the sensor accuracy of +/- 1°C. Given that the sensor accuracy gives a degree of uncertainty in the measurements, it is interesting to note that the error distribution reflects this error range. This further supports the conclusion that the grey-box models, particularly for case 7, are a valid method for predicting thermal behavior of buildings, and especially for prediction of heating and cooling times.

7.9 Estimation of heating time

Case 7 is chosen as the best case for heating and cooling time estimation, because it has a low RMSEP for validation, i.e. less then the sensor accuracy (>1°C). Case 7 also has a clean step response in the reference data, allowing a good comparison between heating time estimation on the model and the real system. In case 7, the first 1050 samples of November (dt = 10min) is used as validation. The step change occurs at sample 545 based on manual inspection of the power used by the electric heater. At this sample, the energy used by the heater increases, hence the heating cycle begins. At this sample, the building temperature T_b is found to be 15.6°C.

To find the end of the heating cycle, an average temperature in the steady state is used. By averaging of samples 720 to 820 gives an average T_b = 21.5°C, hence this is assumed to be the temperature setpoint of the heater thermostat. This setpoint is first reached at sample 679, with a following overshoot. The standard deviation in this range is found to be 0.1°C. The thermostat is in simulations set to have a temperature tolerance (deadband) of +/- 2x stdev = 0.2°C.

The heating cycle is found to take 135 steps of dt = 10 minutes, which computes to 22.5 hours. This is a relatively long heating time, i.e. too long for a residential building in normal use. This means only that the heater is under-dimensioned for the building it is used in, if the step in temperature setpoint used here is representable for the applied final control system. Alternatively, the heater may not be controlled optimally, with respect to using the shortest possible time to reach the new setpoint.

To find the power consumed by the heater, average power consumption when the heater is off, samples 700 - 719, is computed to be 103W. Similarly, samples 547 - 566 when heater is on, gives an average power consumption of 472W. Hence, the electric heater is using approximately 370W and other equipment, such as appliances and computer, in the room uses around 100W. This information is used when simulating the heating time.

The heating time estimation and comparison with reference is started from sample 500. At this time, the temperature in the building is steady at around 15.6°C.

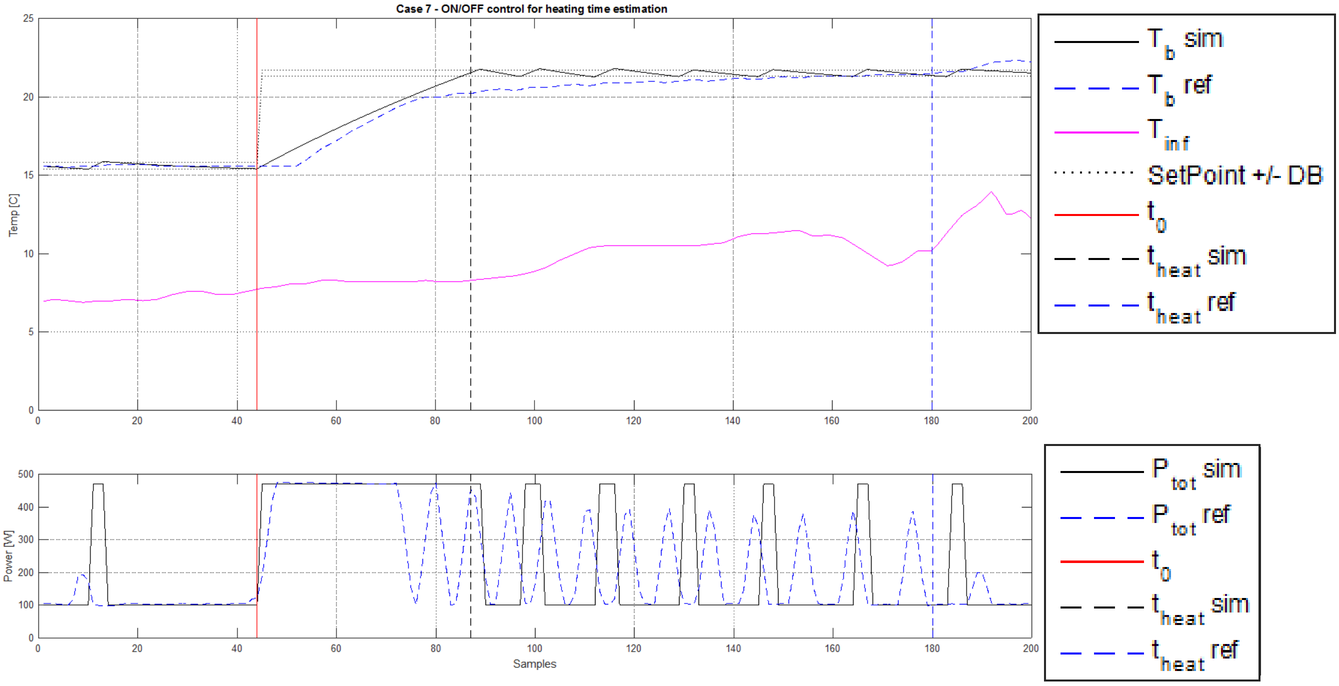


Figure 64 - Estimation of heating time, compared with reference November data set

To simulate the heater under ON/OFF control, a version of the R4C2 model with controller is used, as described in chapters 2.1.2 and 5.7. The simulated heater is configured to supply 370W with a constant 100W supply from appliances, independent of the heaters ON/OFF state. The control deadband is set to +/- 0.2°C, and the starting point of the heating is sample 545, same as found in the data set. The result is presented in Figure 64. Setpoint with deadband is plotted with black dotted line. Black solid line is the simulated temperature, while blue dashed line is the measured reference. The outdoor temperature is plotted in magenta. Vertical lines represent start and stop of the heating cycle. The red vertical line is $t_{0,}$ the start of the step, while black and blue dashed lines are the end of the heating cycle, t_{heat} sim and t_{heat} ref, for simulated and measured data, respectively.

By inspection and interpretation of the results, the following times are computed:

Heating time building to SP:	180-44 = 136 -> 22.5 hours
Heating time building to 20C:	78-44 = 34 -> 5.6 hours
Heating time model to SP:	87-44 = 43 -> 7.6 hours.

Based on these time computations, the model is found to significantly under-predict the time the heater used to reach the setpoint 21.5°C. This happens because the heaters thermostat controller is not reaching the setpoint in the fastest way possible. Electric heaters are typically regulating the temperature of their surface, not the building air temperature. Hence, they switch OFF before the room temperature has reached the setpoint (SP). This can be seen from Figure 64, where the power usage drops *before* the room temperature has reached its setpoint.

This is not efficient with respect to minimizing heating time to a specified setpoint. Since the thermal capacity of the heater can be assumed low relative to the building, there is little chance of overshooting the setpoint by maintaining power to the heater until the setpoint is reached. Even if the heater at this point is hotter then the room temperature, the residual heat energy in the heater, once switched off, will not further increase the room temperature by a large amount causing an overshoot in T_b. Hence, the thermal behavior and power usage in Figure 64 is assumed caused by the bespoke functionality of electric heater thermostats. As evident from the plot in Figure 64, the room temperature in the building quickly reached 20°C, but from 20° to 21.5°C takes significantly longer, due to how the thermostat controller works.

The optimal way to heat a room by electric heater, with respect to minimizing heating time, is for the heater to remain on constantly until the desired temperature is reached. This will give the maximum amount of energy the heater is capable of supplying to the thermal capacitance of the room in the shortest possible time. In case 3, the validation data contain exactly this scenario, i.e. heater is constantly ON.

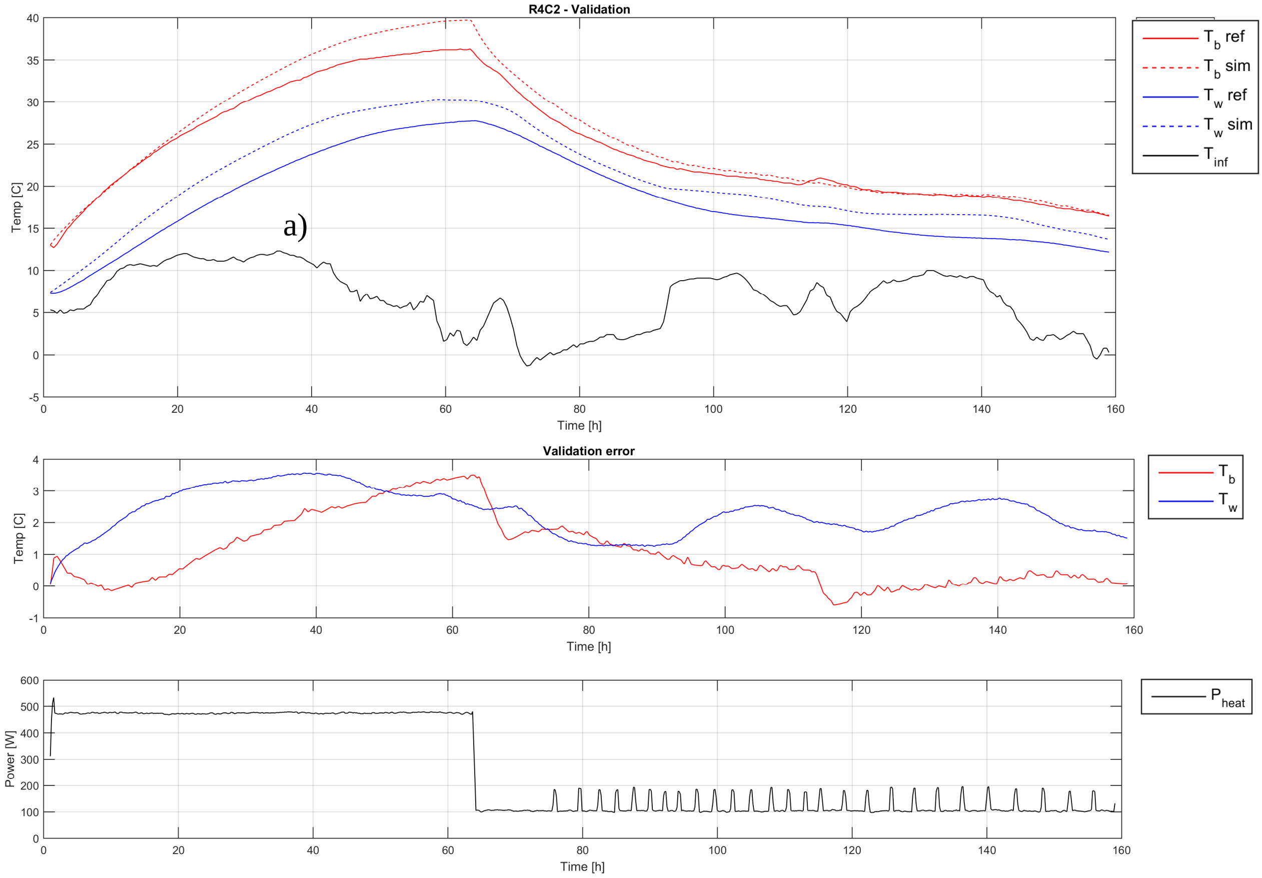


Figure 65 - Case 3 validation results

From Figure 65, it is shown that outdoor temperatures, T_{inf} in graph a), for this data set is similar to that used for calibration of the model in case 7 shown in Figure 61, around 0° to 10°C. The model of case 7 should be able to predict the thermal behavior in the case 3 validation data as well. Heating time can be estimated when the heater is on constantly, i.e. optimal step change from one temperature to another. This is how the heater would behave if under control of a system such as the one described in chapter 2.1.2.

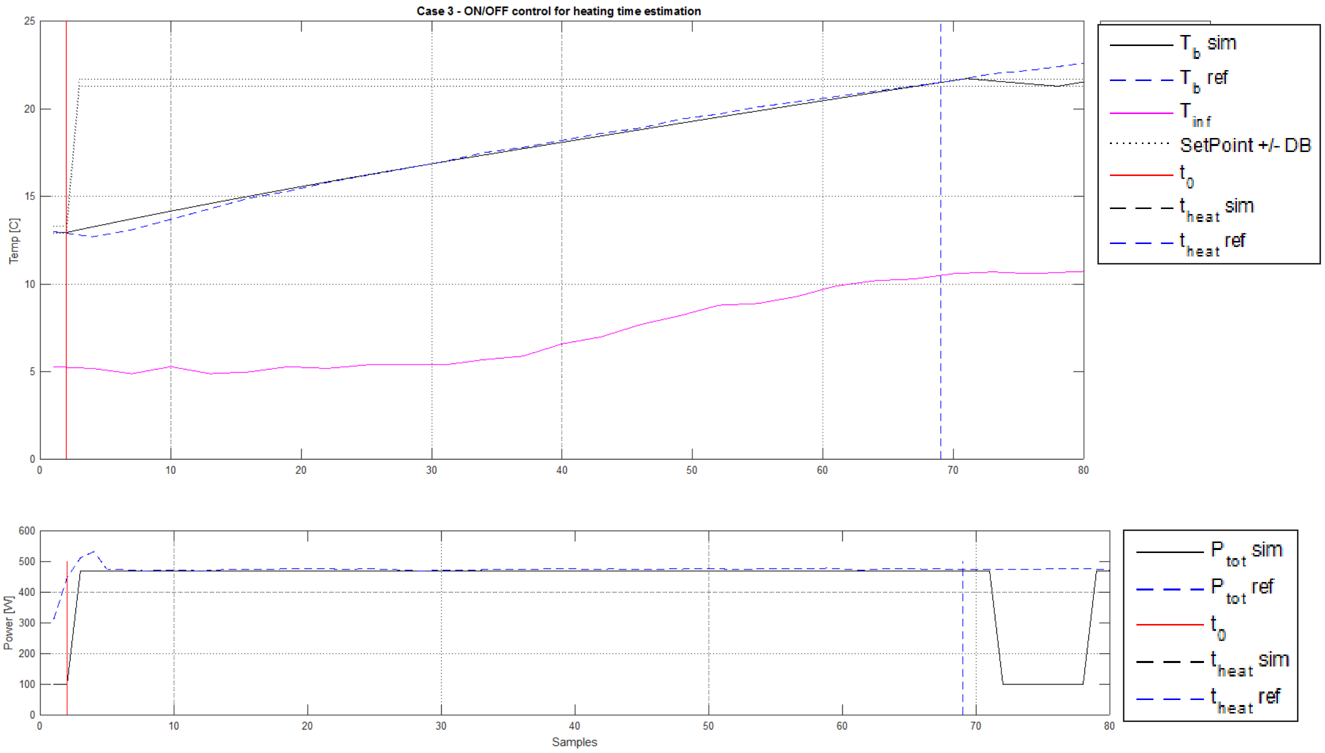


Figure 66 - Heating time estimation on case 3 validation data

By inspecting the data, the heating cycle is found to begin at sample 3. At sample 69 the measured temperature in the building has reached 21.5°C. The same setpoint as in case 7 is used, since case 3 does not actually have a setpoint, or rather the setpoint of the thermostat is unobtainable, so that it heats the building to the highest possible temperature the heater is capable of. The starting temperature of the step in case 3 is found to be 13.1°C. There is a small drop in temperature at the time the heater increases power. This is assumed caused by a person entering the building, thus releasing hot air from the building and reducing temperature. The drop is however small enough that it has little effect on the computations.

Simulations with ON/OFF control, where the simulated heater is kept on until the room temperature reaches it setpoint, is found to produce exactly the same heating time as the reference data. The plot in Figure 66 shows the simulated temperature (black line) reaching the setpoint at exactly the same time, sample 69, as the reference (blue dashed line), with a computed heating time of 69 -3 = 66 samples. The start of the heating cycle t_0 is indicated by vertical red line while the end of the heating cycle is shown as a dashed blue line. This gives a heating time of 10.1 hours. Heating time in case 3 is longer then case 7, because the initial temperature is lower giving a larger step in setpoint.

7.9.1 Delay in heating

In Figure 64, the temperature in the room does not start to increase until a significant time after the heater starts using power. This is likely caused by the thermodynamic behavior of the heater itself. Filtering of power consumption data and outlier removal as described in chapter

4.5 can also explain part of the delay. However, the last outlier sample was found to be sample 550, and the temperature increase does not begin before sample 552, which is approximately 20 minutes later. The filter window used is +/- 15 minutes, using a centered WMA filter, which also does not fully account for the delay in temperature rise. It is therefore likely that the heater dynamics is the main cause of the delay between increased power consumption and rise in temperature. Figure 66 also shows a short delay before the temperature starts to increase, after the heater power is switched on.

To confirm that the heater is indeed the cause of the delay, further research, with measurements of heater surface temperature, is suggested.

7.9.2 Thermodynamics of electric heaters

An interesting improvement on the thermal network models would be to add a first order RC model of the heater itself. If the model can better predict the thermal interaction between heater and air, the heating time estimates may be closer to the measured data. This may be more important for typical Norwegian homes heated by stand-alone electric heaters then models for complex building with central heating systems. For identification of such a model, measuring the temperature of the heater would be required, hence it is not applicable with the data sets used here. However, as a future project, it is likely to give better estimates of heating time under thermostat control if the heater itself is included in the model.

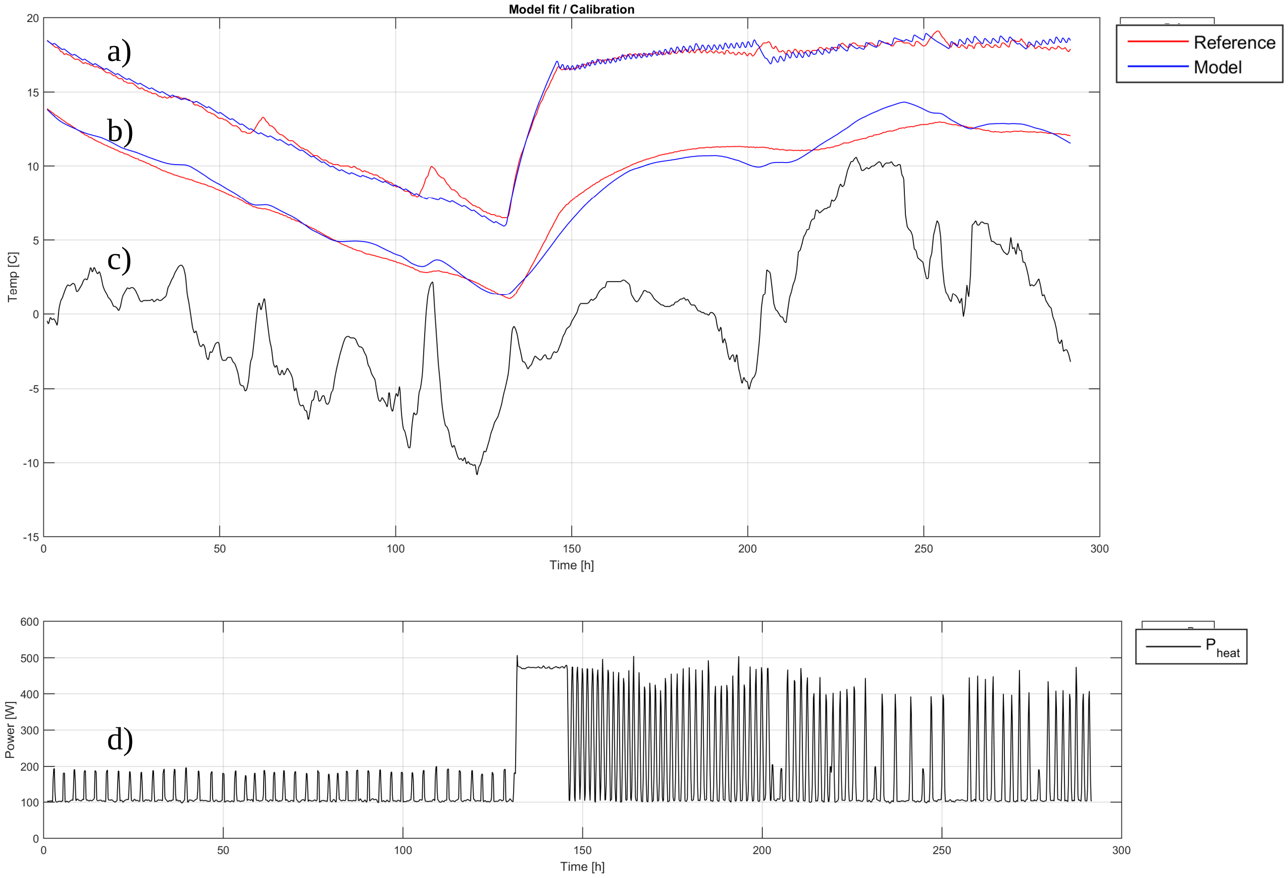


Figure 67 - Calibration results for case 15

In further support of the argument for modeling heaters, the calibration plot results for case 15 are shown in Figure 67. Here, room temperature T_b is marked as graph a) where blue is simulated and red is the reference, graph b) is similarly the wall temperature T_w, and graph c) is the outdoor temperature. Finally, graph d) is the power usage.

The simulated room temperature contains many small spikes that are not found in the reference data. The general trend of the model simulation results is similar to the reference data, but these small spikes are only seen in simulations. This indicates that the power supplied to the heater is not directly introduced into the thermal capacitance of the room air, C_b, such as the model describes. Rather, the heater acts as a low-pass “filter”, where the spikes in supplied energy is smoothed out before affecting the room temperature. Heating of the room is a consequence of raised temperature of the heater.

From the above discussion of the way heat energy from electric heaters is modeled, and also the discussion of delays in temperature increases, it seems likely that improved prediction results may be gained by introducing the heater temperature as a state in the model. By adding a thermal capacitance for the heater, with a separate thermal resistance to the room air, the model would more accurately describe the data. Further, a measurement of the heater surface temperature is required in order to model this temperature as a state, i.e. a reference is needed to identify the parameters of the heater. Where multiple heaters are used, it may be possible to model them as a single heater, using the average surface temperature as a reference for the model state.

The fact that models can be modified and expanded to fit observed data further emphasizes the strength of cognitively derived model structures [5], as discussed in chapter 7.1.

7.9.3 Energy savings by improved control

The results show a potential for energy saving. Since such a system would require a shorter time of high power to the heater, reduction of energy usage could be gained by this type of improved control.

As previously discussed, the heating cycle here is unnaturally long (~22h) and hence the estimate potential savings is hardly general. Further research with a better dimensioned heater, or improved control system, could be done to improve the estimate and perhaps scale it up to potential savings over one year. Regardless, there seems to be evidence in the results that a potential for energy savings by an improved control scheme exist. The following argument and computation is included as an example of how energy savings due to improved control can be calculated.

Assume that sample 180, the time when the reference data reaches the setpoint (SP) temperature, is the time where the building is expected to be at comfort temperature (SP = 21.5°C). The start of the heating cycle would have been delayed from sample 44 to sample 137, based on the model simulated heating time of 43 samples.

Assuming now that the heater is on for the entire 43 samples, and not switched off by the onboard thermostat as in Figure 64, and further assuming that keeping heater on max power for 430 minutes, as the model predicts, is enough to reach the setpoint, there is a time period of 15.5 hours where the heater can be left on the low setpoint of 15.6°C rather then the high setting of 21.5°C. These assumptions are sound, because the heater is in fact capable of reaching the 20°C in a time shorter then the estimated heating time, so keeping the heater on for slightly longer would likely reach the setpoint of 21.5°C within the estimated time.

By averaging the power consumption of the heater at the low and high setpoint, samples 1-300 and 850-950 respectively, i.e. the time when the heater is only maintaining setpoint and not changing it, a power consumption of 116 W for low and 158W for high is found. Thus, if the heater can be kept on low setpoint for 15.5 hours longer, an energy saving of (158-116) W * 15.5h = 0.651kWh is gained. The test-building in this case is very small compared to most residential buildings. Hence, for larger building, the energy savings would increase according to the building size. Also the estimated savings could be realized for each heating cycle.

This reduction in energy consumption is made possible by using heating time estimation to better maintain comfort temperature in the building only when required. Consequently, improved control of the heater itself is achieved by using building temperature, not heater surface temperature, as the control variable.

7.9.4 Performance criteria

In much of this project, the RMSEC and RMSEP statistics are used to quantify models in terms of fit with calibration data and ability to predict independent test data, respectively. Since the purpose of this project, as discussed in chapter 2.1, is the estimation of time, not temperature, a better criterion of model performance would be comparing prediction time of the model with actual building heating cycles. The problem with this method is that due to the insufficiencies of thermostat controllers buildings are not heated optimally, as they would be by a more advanced control system in which models could be used. Hence, comparison of model and measurements is not readily available and can not be used as a performance criterion.

An improvement on this project would therefore be to implement the control system discussed in 2.1.2, at least partially. New experiments where heaters are controlled based on room temperature, thus ensuring that the heaters are controlled optimally with respect to heating time, could be carried out. The results from such an experiment could then be used to train models and compare the predicted heating time of model and building, to find out what model structure performs better in the framework of a predictive control system.

7.9.5 Conclusion of heating time estimation

From the heating time estimates for case 7, it is demonstrated that due to the way thermostats on electric heaters work, the room takes a long time to reach the setpoint. Without introducing the thermostat into the model, this is behavior the model can not predict. Therefore the model gives a much shorter heating time then what actually occurs in the data.

The goal of this project is to show how grey-box models can be used in predictive control systems. In such a system, the heater would not be controlled by the onboard thermostat, but rather by an external controller, thus modeling the thermostat is not relevant for this project.

The difference between the models estimated heating time and actual time it takes to reach the setpoint in case 7 is not an indication of the model being wrong. In fact, based on validation of the model, it predicts quite accurately what will happen to the temperature in the building, given the supplied heater power. The difference in heating times is caused by the models ability to predict thermal behavior producing a significantly better control signal to the virtual heater, compared with what the thermostat on the actual heater is capable of.

The part of the model that does not fit the real system is the simulated ON/OFF controller, or thermostat, which outperforms the controller on the heater because the virtual controller uses the room temperature as a reference, not the heater surface temperature as the physical device does. The difference in heating time should therefore be seen as a positive result of the project. This is an indication of a potential for energy savings if the models and control structure presented here is applied.

When instead using case 3 to compute heating time, where the heater is not controlled by the thermostat, because the setpoint is turned so high that the thermostat is always on, the heating time estimates of the model are accurate to within one sample (dt = 10 min).

This result shows that grey-box models are capable of predicting heating times with the required accuracy for use in predictive control systems. This result should be seen as the most important conclusion of this project.

It is also worth noting that the predictions on case 3 validation data is done with model from case 7, i.e. November data is used to predict December behavior. This further demonstrates that the model is able to predict behavior on independent data, as long as the input ranges are similar to the calibration data.

7.10 Estimation of cooling time, November ByggLab

Estimating cooling time is interesting because by predicting how long it takes the temperature to drop below comfort level tolerance, the control system can choose to turn off heating a predicted time before the building will be unoccupied. Typical control systems in use will switch off heating only once building is empty. With accurate predictions of the cooling time, energy savings can be gained by precise control of building temperature, maintaining comfort temperature only when strictly required.

In the data set for November the high of power consumption peaks decrease from around 400 to around 200W between sample 1102 and 1112. Plotting from sample 1096 gives:

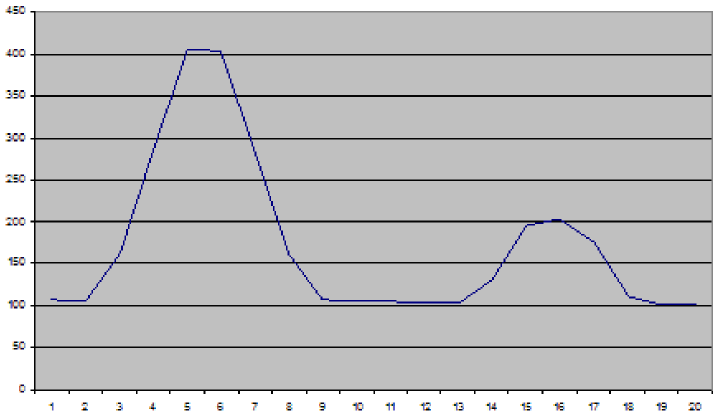


Figure 68 - Peaks in power consumption indicate start of cooling cycle

From Figure 68, it is approximated that the sample between the two peaks, sample 1107 is a reasonable starting point for the cooling time, i.e. when the thermostat was turned down to the lower setting. By assuming a temperature drop of 1°C below the setpoint is acceptable, i.e. 20.5°C, the goal is to estimate how long time it takes the temperature to drop by 1°C. Based on experimental data, 20.5°C is reached at sample 1145.

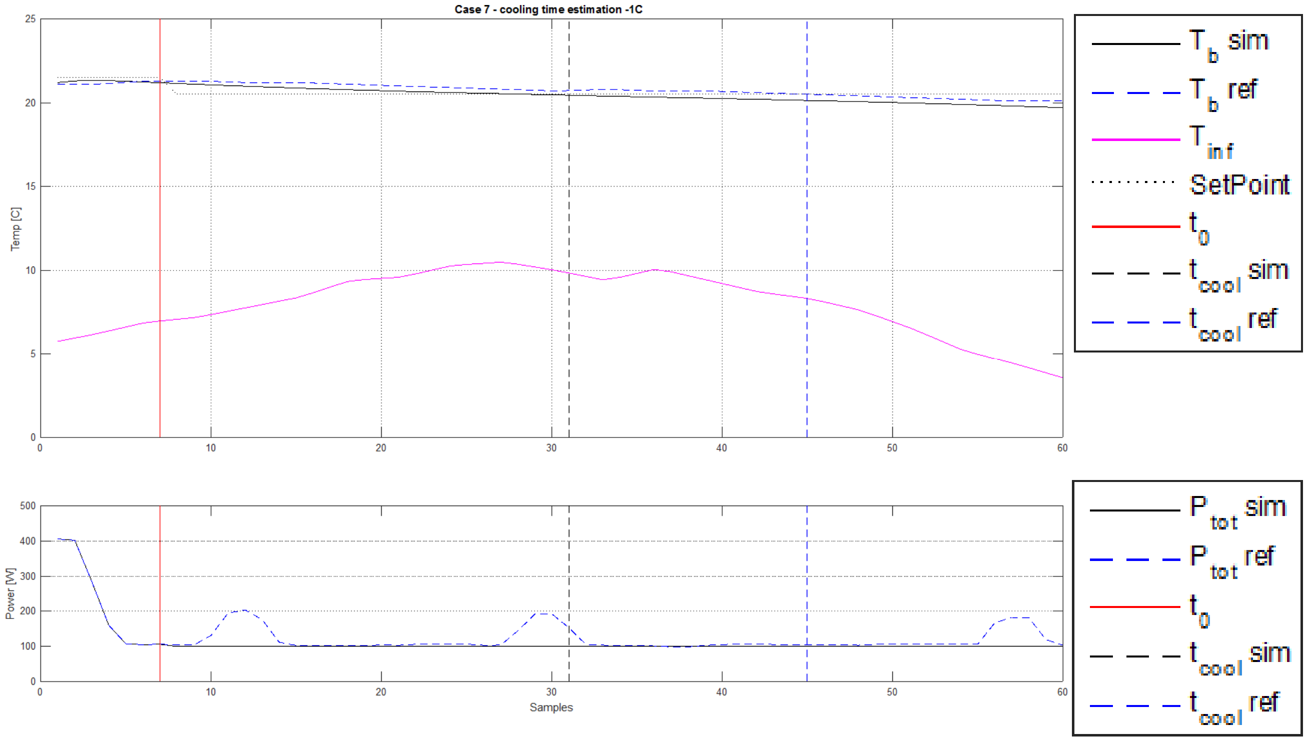


Figure 69 - Estimation of cooling time, November data

From Figure 69 the time it takes from the timestep where temperature setpoint is assumed to change to low, at k = 1107, it takes 24 timesteps in simulations and 38 timesteps in reference data, before temperature has dropped by 1 degree. Vertical lines represent the start of the cooling cycle t_0 (red), time when simulations reach 1°C below setpoint t_{cool} sim (black stapled) and the same for reference data t_{cool} ref (blue stapled).

The model is using actual power data from measurements up to the point where the setpoint is changed, and a constant 100W power to appliances after the setpoint. However, in the reference data there are some peaks in power, suggesting that there is some added heat to the building during the cooling phase. This power usage peaks could be caused by equipment other then the heater. It could also explain why cooling time is longer in the reference data.

The interesting thing to note here is that based on simulations, which are somewhat conservative in their results due to the bespoke power consumption input, it takes about 4 hours (24 samples at dt = 10 min) before temperature has dropped by one degree. By this estimate, the temperature setpoint could have been changed four hours earlier, if a temperature 1°C below the initial setpoint is acceptable.

7.10.1 Estimated energy saving potential

Using the estimated values for power consumption with high and low thermostat settings, the energy savings would compute to (158-116) W * 4h = 0.168kWh. Note again that this is a low number, but also still for the test building which is quite small.

This kind of energy savings, by predictable cooling, could be realized daily, perhaps even twice daily, depending on how often the temperature setpoint changes. As a rough estimate, assume that for 200 days pr year, a residential home of comparable thermal behavior to the test facility, but with five times higher need for heating power, the temperature is lowered both during the day while occupants are at work, and at night while house is unused. By utilizing a model to predict how long it will take the temperature to drop 1°C, choosing a conservative energy savings estimate of 0.5kWh pr cooling cycle for a larger building, based on the estimates for the test facility and its size, the savings pr year comes out to approximately 200kWh. This estimate is a very rough calculation, but it is only taking into account savings based on predictably initiating cooling of buildings a certain time before they are not used. Predictable heating is likely to have significantly higher potential for improved energy economy. The computation is mainly included to show that there is a potential for energy savings by predictable cooling of buildings.

It should be noted that this estimate is based on an outdoor temperature around 8° to 10°C, which may be a good approximate average year-around temperature for southern Norway. In colder climates, the savings might be significantly higher.

8 Conclusion

In this project, several software tools has been developed and tested. Together, they form a tool-set for grey-box modeling by optimization based parameter identification [2], and for simulation of the developed models. Relevant pre-processing of data is shown to be efficiently carried out by the developed software.

Implementation of models in both c# and MATLAB are shown to give identical results. Further, ODE solvers [8, 21, 22] implemented in both languages are demonstrated to solve models quickly and accurately, providing a computational foundation for model based predictions in control systems. The RK4 fixed step solver algorithm is shown to greatly improve the computational speed, relative to the built in MATLAB solvers. The c# implementation improves the speed even further, ensuring that the simulation techniques presented in this report can be implemented in a real-time system [11], as discussed in chapter 2.1.2.

Theory and practice regarding the concept of grey-box models, discussed in chapters 2.4 and 4.3, is given [2, 3], with some additional presented theory, discussed in chapter 4.2, regarding data driven modeling in general [19, 20]. The amount and type of data required for successful parameter identification of a grey-box model is discussed in chapters 7.2, 7.3 and 7.4. Range and variation of inputs and states is found to be the dominating factor in successful parameter identification, as well as use of a proper model structure [3]. Models are mainly taken from literature [2], with one example of how models can be adapted by cognitive reasoning rather then physical equations [5]. Further improvements on models are suggested, particularly with respect to inclusion of the heater to air dynamics for improved thermal control.

Results of using two separate model structures show that increasing model complexity does not yield improved results, as discussed in chapter 7.6. Due to redundancy in model parameters, as discussed in chapter 7.5, no conclusion about increased model complexity is possible without further research.

A proposed control strategy, given in chapter 2.1, 7.9 and 7.10, both for improved local control of energy to electric heaters, and for supervisory control of building temperature is demonstrated to show promising results with regards to the reduction of energy consumption. By use of grey-box models, with parameters identified for a particular building, results show prediction accuracy in the same range as the sensor accuracy. Further, models have been shown to predict heating and cooling times. Analysis shows that the electric heaters employed in the experiments do not efficiently reach the setpoint temperature, with respect to heating time. If the heater is left fully on for the whole heating cycle heating times is shown to be predicted with accuracy of one sample. Solutions for improvements to this control system are suggested in chapter 7.9.2.

The most important result of this project is discussed in chapter 7.9, where a grey-box model of the simplest form (R4C2) [2], using parameters identified from independent data, is shown to predict the heating time of the test building “ByggeLab” with an accuracy of one timestep (10 min). This experiment shows that the combination of grey-box modeling with heating and cooling time predictions is a feasible strategy for advanced temperature control in buildings.

References

[1] D. W. U. Perera, C. F. Pfeiffer, and N.-O. Skeie, "Modelling the heat dynamics of a residential building unit: Application to Norwegian buildings," *Modeling, Identication and Control*, vol. 35, pp. 43-57, 2014.

[2] T. Berthou, P. Stabat, R. Salvazet, and D. Marchio, "Development and validation of a gray box model to predict thermal behavior of occupied office buildings," *Energy and Buildings*, vol. 74, pp. 91-100, 5// 2014.

[3] G. Reynders, J. Diriken, and D. Saelens, "Quality of grey-box models and identified parameters as function of the accuracy of input and observation signals," *Energy and Buildings*, vol. 82, pp. 263-274, 10// 2014.

[4] A. Afram and F. Janabi-Sharifi, "Review of modeling methods for HVAC systems," *Applied Thermal Engineering*, vol. 67, pp. 507-519, 6// 2014.

[5] K. K. Associates, *Thermal Network Modeling Handbook*, 2000.

[6] S. F. Fux, A. Ashouri, M. J. Benz, and L. Guzzella, "EKF based self-adaptive thermal model for a passive house," *Energy and Buildings*, vol. 68, Part C, pp. 811-817, 1// 2014.

[7] A. Afram and F. Janabi-Sharifi, "Gray-box modeling and validation of residential HVAC system for control system design," *Applied Energy*, vol. 137, pp. 134-150, 1/1/ 2015.

[8] A. Tveito, H. P. Langtangen, B. F. Nielsen, and X. Cai, *Elements of Scientific Computing*: Springer, 2010.

[9] L. Wang, *Model predictive control system design and implementation using MATLAB®*. London: Springer, 2009.

[10] N.-O. Skeie, *Lecture Notes: An Introduction to Hard/Soft Sensors in Process Measurements*. Porsgrunn: Telemark University College, 2013.

[11] N.-O. Skeie, *Lecture Notes: Industrial Information Technology*. Porsgrunn: Telemark Univeristy Collage, 2014.

[12] N.-O. Skeie, *Lecture Notes: Object-Oriented Analysis, Design, and Programming using UML and C#*. Porsgunn: Telemark University College, 2014.

[13] T. L. Floyd, *Electronic devices*. Upper Saddle River, N.J.: Prentice Hall, 2002.

[14] R. Oleksandr, "Master Thesis: Estimation of the heating time for buildings," Telemark University College, Faculty of Technology2015.

[15] A. Devices, "Low Voltage Temperature Sensors, Rev H," One Technology Way, P.O. Box 9106, Norwood, MA 02062 - 9106, U.S.A.2015.

[16] T. V. AB, "Manual: Weather Station," in *Art.no. 02049 Ver. 200908*, ed, 2009.

[17] D. W. U. Perera, M. Halstensen, and N.-O. Skeie, "Prediction of Space Heating Energy Consumption in Cabins Based on Multivariate Regression Modelling," *International Journal of Modeling and Optimization*, vol. 5, 2015.

[18] K. Esbensen, *Multivariate analysis in practice*. Oslo: CAMO, 1998.

[19] L. Ljung, "Prediction error estimation methods," *Circuits, Systems and Signal Processing*, vol. 21, pp. 11-21, 2002/01/01 2002.

[20] R. T. Baillie, "Predictions from ARMAX models," *Journal of Econometrics*, vol. 12, 1980.

[21] E. C. Ifeachor and B. W. Jervis, *Digital Signal Processing, A practical approach*: Prentice Hall, 2002.

[22] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical recipes in C++: the art of scientific computing*. Cambridge: Cambridge University Press, 2002.

[23] J. W. Nilsson and S. A. Riedel, *Electric Circuits*, 6th ed.: Prentice Hall, 2001.

[24] S. Linge and H. P. Langtangen, *A Gentle Introduction to Programming for Computations*: Telemark University Collage, 2014.

[25] AnalogDevices, "Low Voltage TEmprature Sensors TMP35/TMP36/TMP37 Rev H," ed.

Appendices

A) Task description

B) Plots of results, calibration and validation for all 22 cases, with description and statistics

C) Code listing of relevant C# classes and MATLAB functions

D) Summary Sheet

Appendix A - Task Description

FMH606 Master's Thesis

Title: Grey box models for estimation of heating times for buildings

TUC supervisor: Nils-Olav Skeie Ver. 11-NOV-15

Co-supervisors: Carlos Pfeiffer

Task background:

The building sector has become one of the largest energy consumers in the world owing to the population growth, economic development and social development. In Scandinavian countries where a harsh cold climate dominates, building energy consumption is at its highest due to the space heating and water heating. In Norway, 48% of the total energy production is allocated for space heating. Recent investigations in Norway have revealed that there is a feasibility of saving more than 50 TWh both from residential and commercial buildings by 2020. To achieve this goal, the Norwegian government has imposed various regulations to force the people to save energy.

A good way to save energy for space heating is to turn the heating off when the building is not in use. Under these situations, it is important to know how much time is required to heat a building from a particular low temperature to a comfortable temperature. The heating time of a building depends on outside weather conditions (outside temperature, solar irradiation, wind, humidity), ventilation, building construction properties, heat sources, etc. To achieve a better prediction of the heating time it is necessary to consider all these consequences simultaneously.

Further, thermal mass of the building plays an influential role in heating of buildings. Higher the thermal mass, it might store energy during the nighttime (when the power is cheap) to release it as heat during the daytime reducing the electricity bill.

A mathematical model must be used for estimating the heating times, and a combination of both a mechanistically model and a data driven model can be a solution for adapting to different type of buildings. A mechanistic model is developed for a building [1] and can be used as a starting point. This model has been testing out on a small building at Telemark University College [2].

Task description:

The main task will be to evaluate how mechanistic and data driven models can be combined to estimate the heating/cooling time of buildings.
The sub-tasks will be:

- Give an overview of the research status for combining mechanistic and data driven models, as grey box models,
- Discuss methods that can be used for developing data driven models, with focus on models for estimating heating time for buildings,
- Discuss combinations of mechanistic and data driven models that can be useful for estimating heating time for buildings,
- Implement models for a single-zone building, preferable in the programming language C# (or Matlab if C# is too time consuming), based on the literature review.
- Use the developed models to simulate and predict the heating and cooling times for several buildings. Experimental data will be available for these buildings. Compare the results.
- Discuss how to adapt these models to any building in an easy and efficient way, with focus on the type of data needed, the amount of data needed, calibration of the mechanistic model, and the training and verification of the data driven model.

Student category:

SCE students

Practical arrangements:

Experimental data for at least three buildings will be available for the models.

Signatures:

18-nov-15

Student (date and signature): Ole M. Brastein ……………………………………………………

Supervisor (date and signature): ……………………………………………………..

References
[1] Perera, D.W.U., C. Pfeiffer, and N.-O. Skeie, *Modelling the heat dynamics of a residential building unit: Application to Norwegian buildings.* Modeling, Identification and Control, 2014. **35**(1): p. 43-57.
[2] Romanets, Oleksandr, *Estimation of the heating time for buildings.* Master Thesis, 2015, Telemark University College.

Appendix B - Details of all result cases

In this appendix the raw results of all 22 cases from chapter 6 is presented. For each case, first, the calibration plot with error distribution is given. The MATLAB code generates these automatically for all models. Red lines are references while blue are model simulated values. In the error distribution plots the red vertical line indicates the average error. If the only error in a model calibration or validation is random noise, the average error should be 0 and the distribution forms an approximate Gaussian distribution.

After calibration plots, a text box with statistics is given. This includes the actual parameter values for the model, with the initial guess the optimizer started from in parenthesis. The statistics include the time it takes to identify the parameters, and the step length of the simulations, which must be same as in the data set. Both RMSEC for all states combined, and RMSEP for each state is given.

Then, the validation results are given as plots of reference temperatures and predicted temperatures, together with relevant inputs.

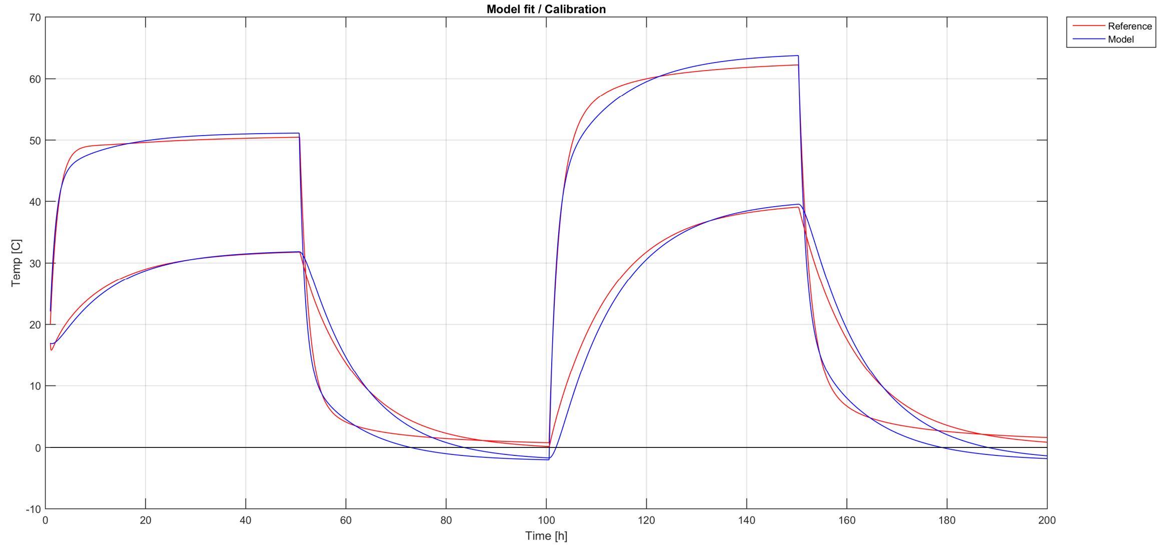
Finally, a small description of each case is given.

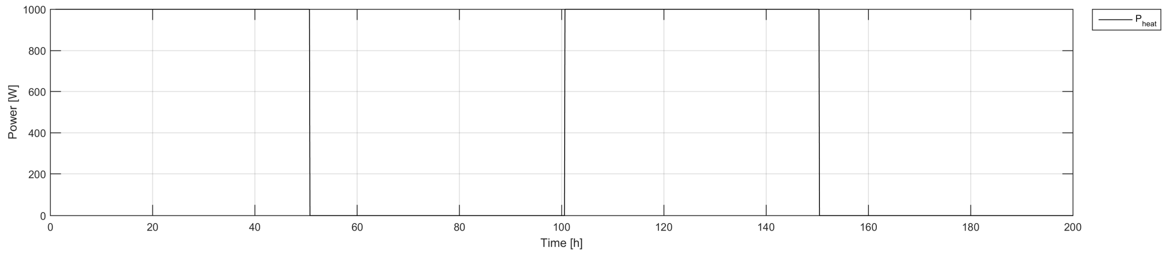
Case 18 is not included since it gives unusable results (parameter identification does not converge to a reasonable solution, i.e. something similar to case 17 which is the same building. See discussion in chapter 7.2)

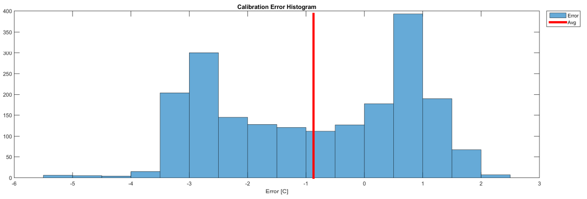
Several of the results presented in Appendix B is also found in chapter 7. The duplicated plots in appendix B is simply included for completenes, such that appendix B contains all results from all cases, even if those results are discussed in detail in previous chapters.

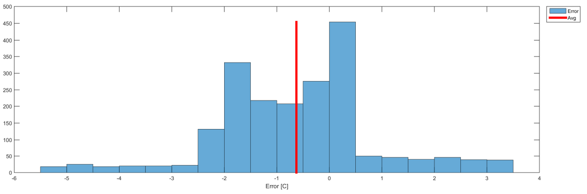
Case A

Calibration



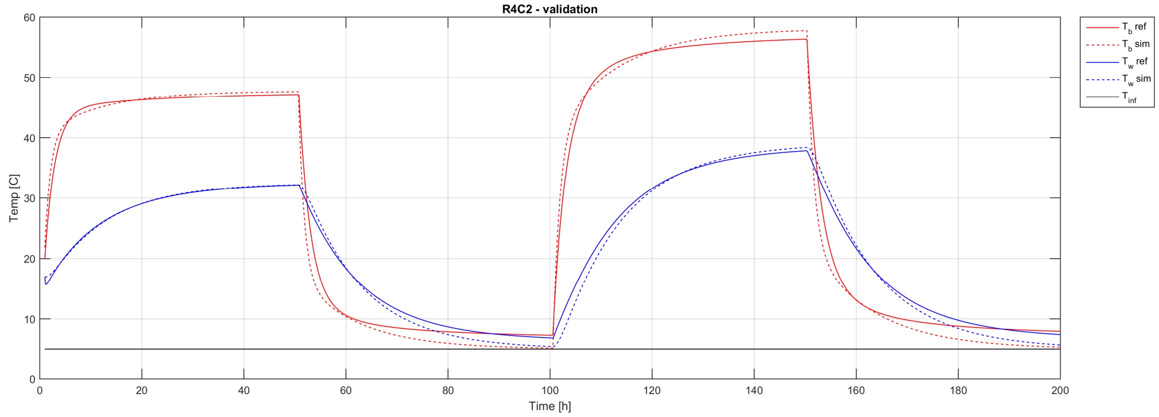


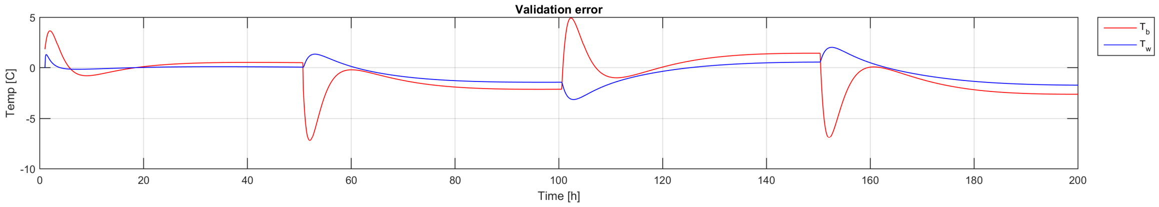


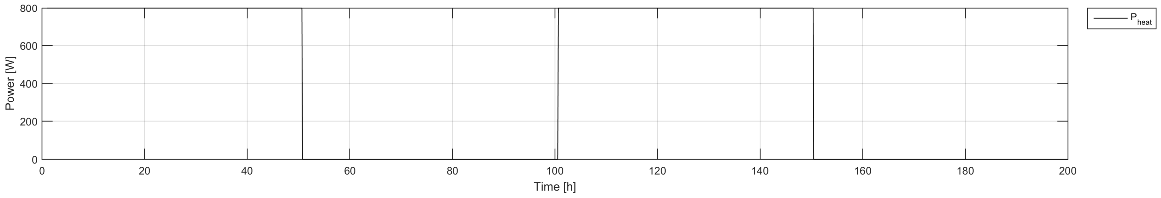


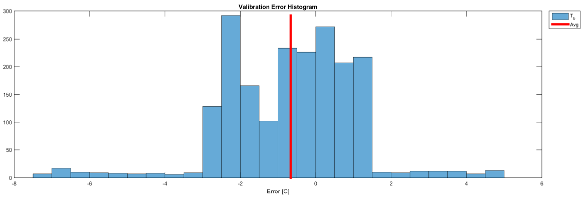
```
Simulation statistics
Total time     : 0.7 min
Timestep       : 360 sec
R_b    =              0.114847             (Initial = 0.060000)
R_w    =              0.205706             (Initial = 0.600000)
R_g    =              0.083690             (Initial = 0.135000)
R_vent =              0.191784             (Initial = 0.200000)
C_b    =              102887.846812        (Initial = 58000.000000)
C_w    =              422925.109635        (Initial = 500000.000000)
RMSEC: 2.492
Model type: R4C2
rmsep Tb: 1.871
rmsep Tw: 1.053
```

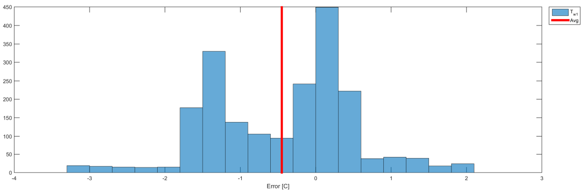
Validation









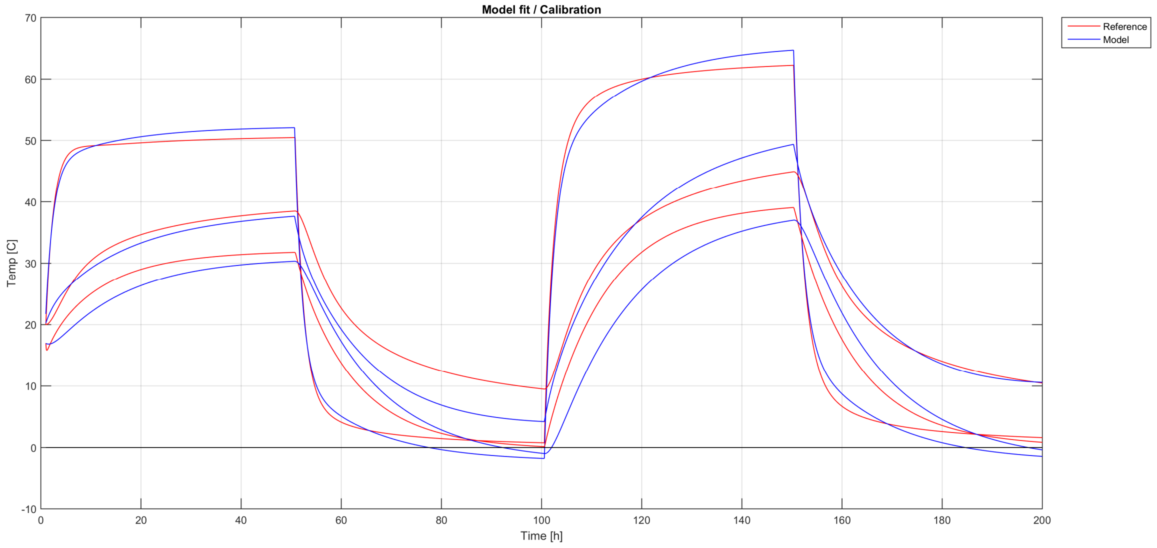


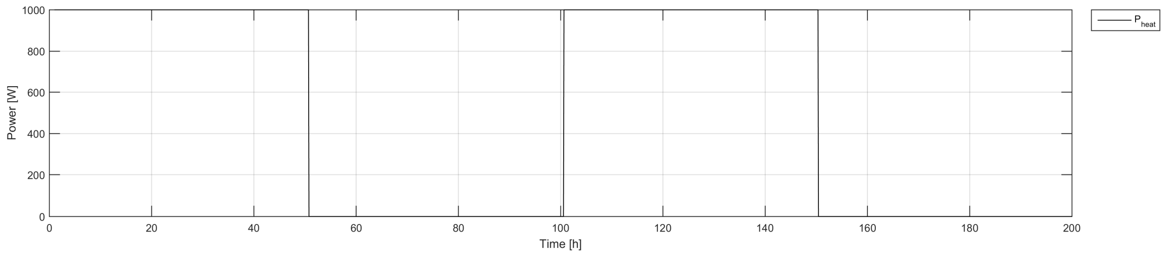
Description

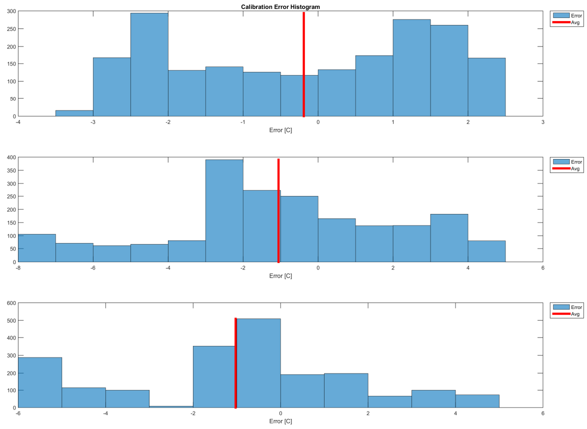
This case is based on simulated data from single-zone model in [1]. The model parameters are taken from mostly from the article, with some additional data from the author and from the MATLAB code used in simulations. The results match those presented in the article. For all simulations ventilation of 0.77 m3/h is used for the first half of the simulation, and no ventilation for the second half. The same is repeated again for validation data, but this time the outside temperature is increased from -2.2 (used in the original article) to +5. In case A R4C2 is the model that is calibrated to the simulated data.

Case B

Calibration

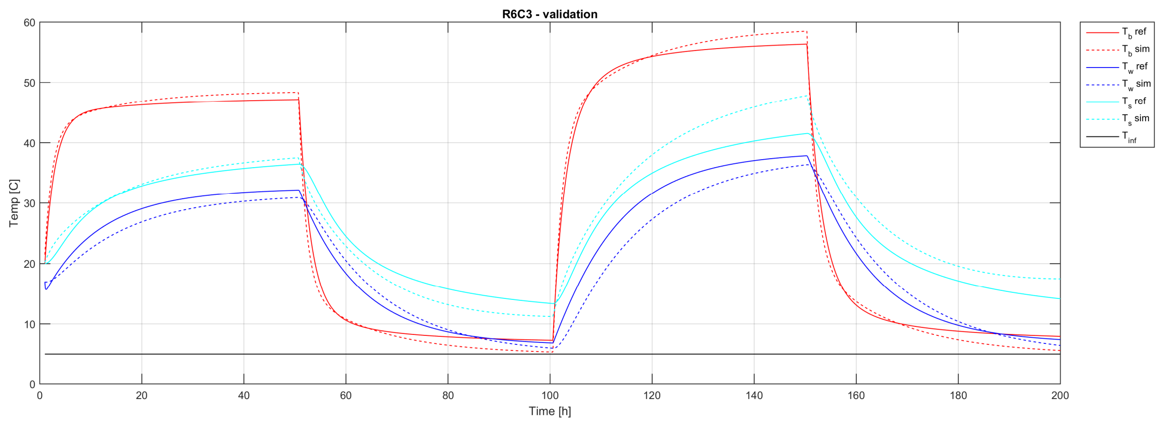


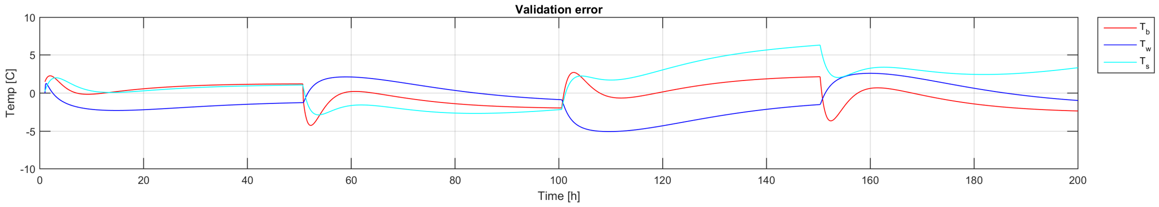


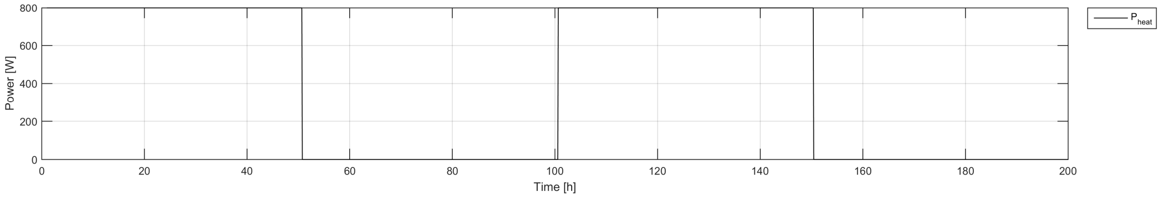


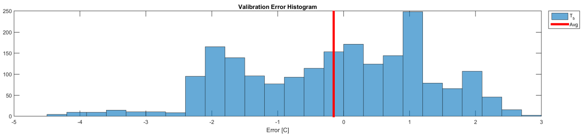
```
Simulation statistics
Total time      : 1.9 min
Timestep        : 360 sec
R_b    =              0.109506            (Initial = 0.200000)
R_w    =              0.154294            (Initial = 0.200000)
R_s    =              0.021128            (Initial = 0.020000)
R_e    =              0.048588            (Initial = 0.020000)
R_g    =              0.085182            (Initial = 0.135000)
R_vent =              0.194271            (Initial = 0.200000)
C_b    =              127525.376848       (Initial = 58000.000000)
C_w    =              521414.292444       (Initial = 500000.000000)
C_s    =              7380.319340         (Initial = 20000.000000)

RMSEC: 4.681
Model type: R6C3
rmsep Tb: 1.420
rmsep Tw: 2.225
rmsep Ts: 2.824
```

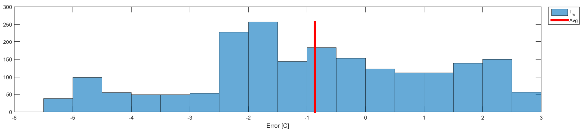
Validation

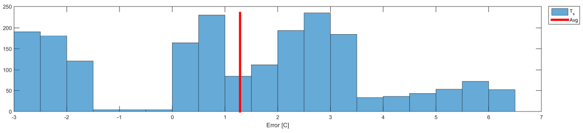










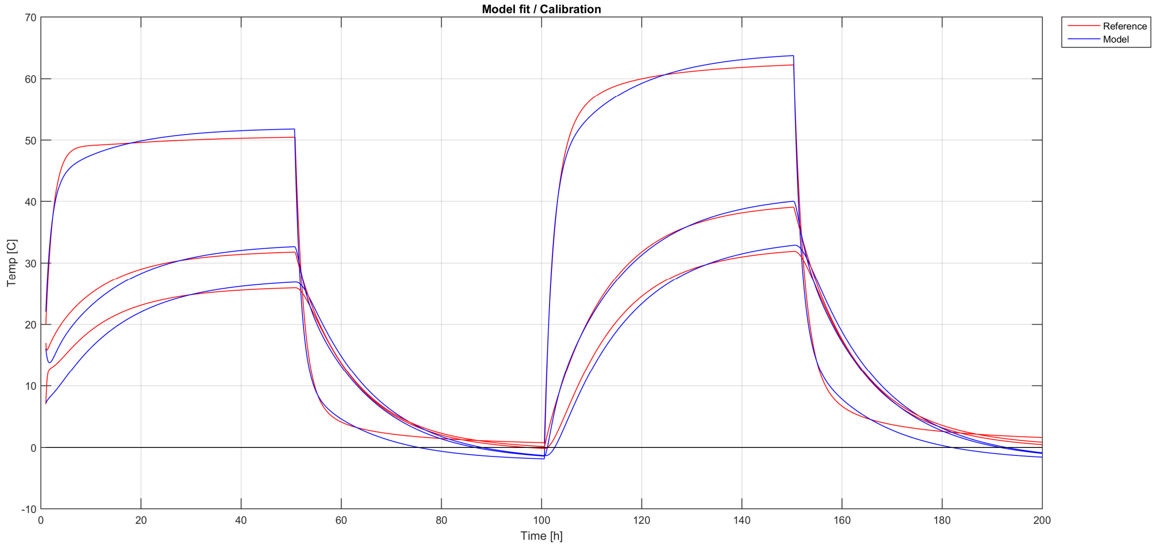


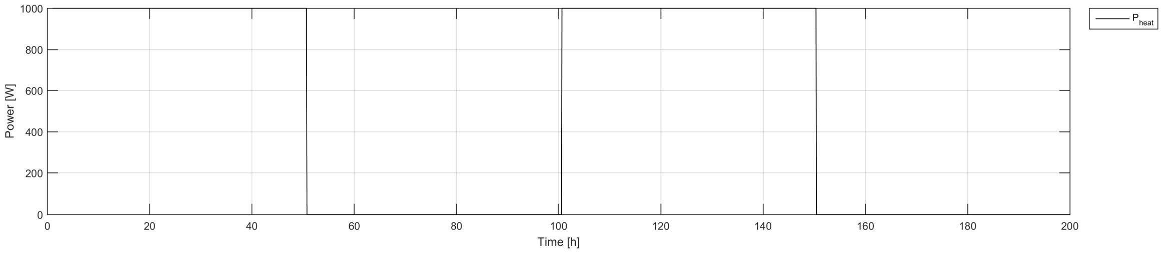
Description

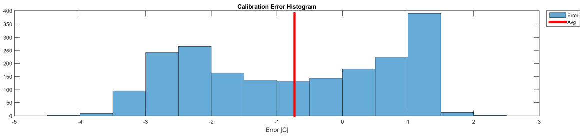
Same as case A, but using model R6C3.

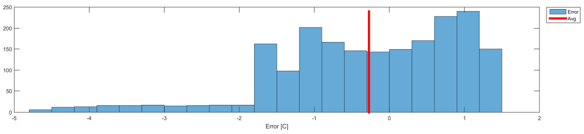
Case C

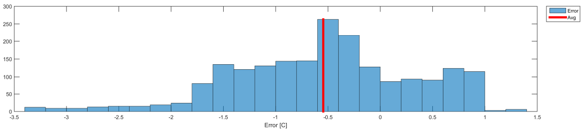
Calibration





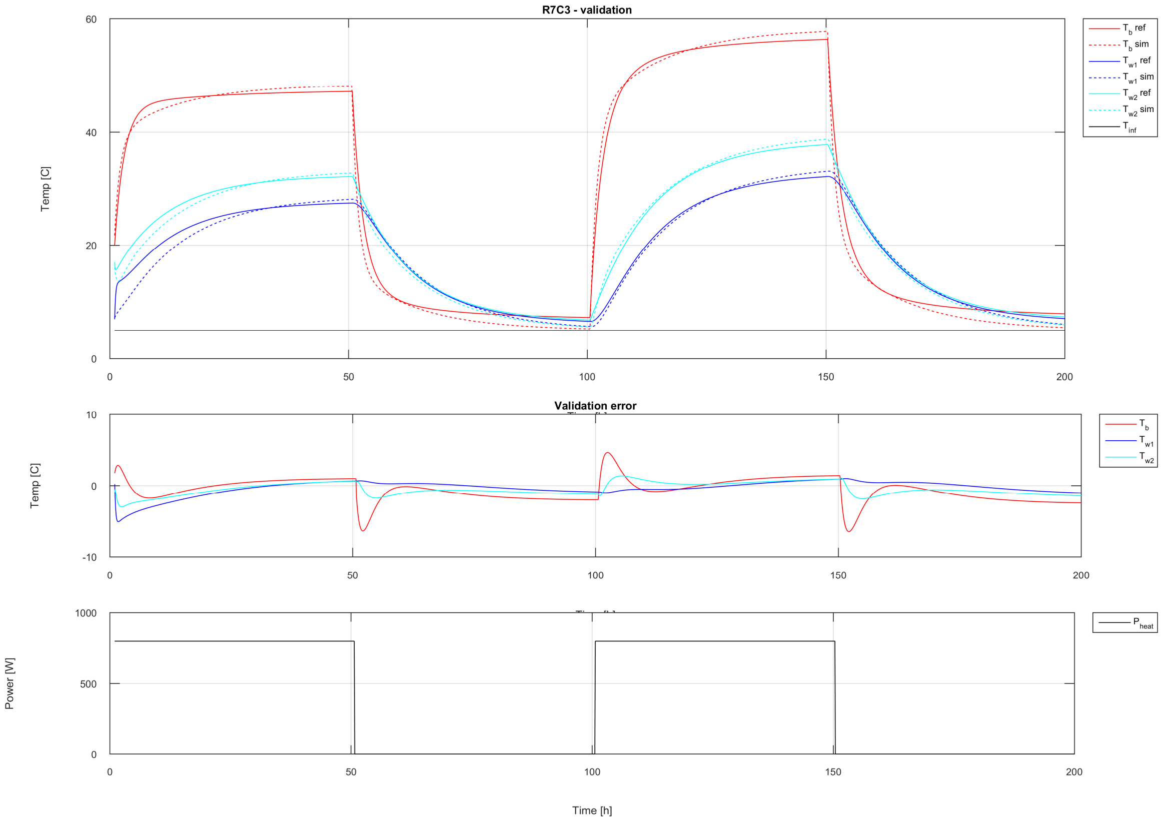


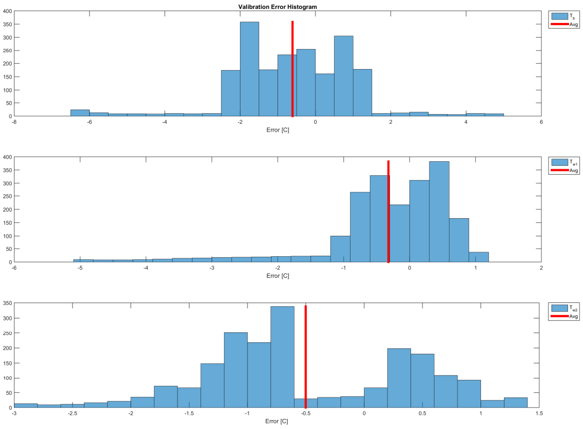




```
Simulation statistics
Total time     : 1.3 min
Timestep       : 360 sec
R_b     =                 0.079525               (Initial = 0.200000)
R_w1    =                 0.124764               (Initial = 0.100000)
R_w2    =                 0.030125               (Initial = 0.050000)
R_s     =                 0.019515               (Initial = 0.020000)
R_e     =                 0.031240               (Initial = 0.020000)
R_g     =                 0.086887               (Initial = 0.135000)
R_vent =                  0.204330               (Initial = 0.200000)
C_b     =                 106171.842680          (Initial = 58000.000000)
C_w1    =                 478522.915426          (Initial = 250000.000000)
C_w2    =                 103737.964106          (Initial = 250000.000000)
RMSEC: 2.367
Model type: R7C3
rmsep Tb: 1.737
rmsep Tw1: 1.058
rmsep Tw2: 1.020
```

Validation



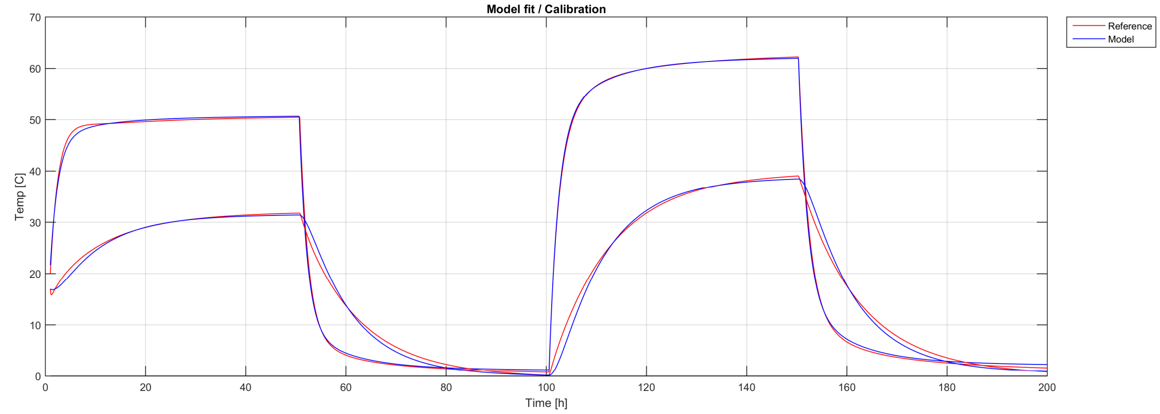


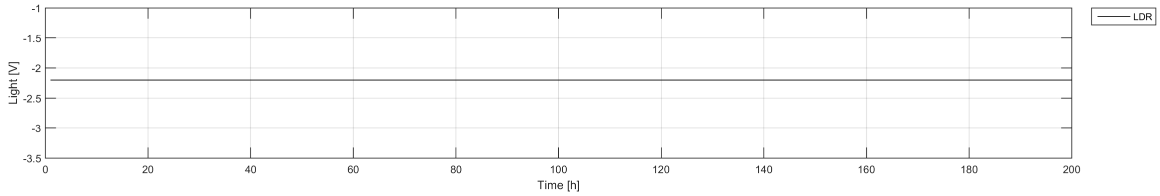
Description

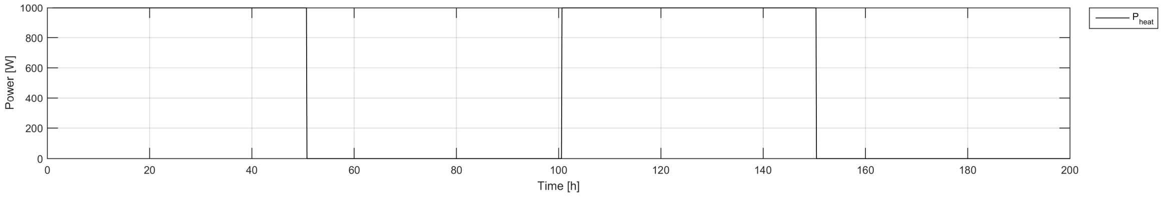
Same as case A, but using model R7C3.

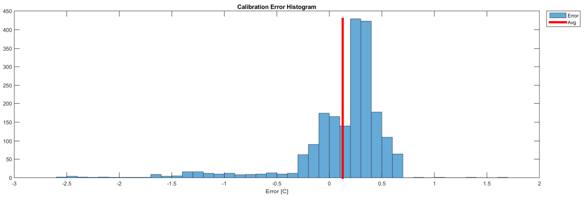
Case D

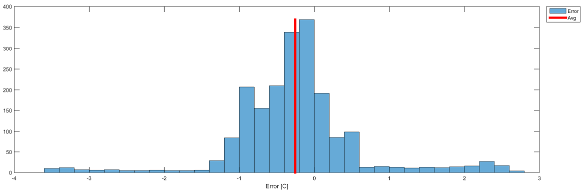
Calibration





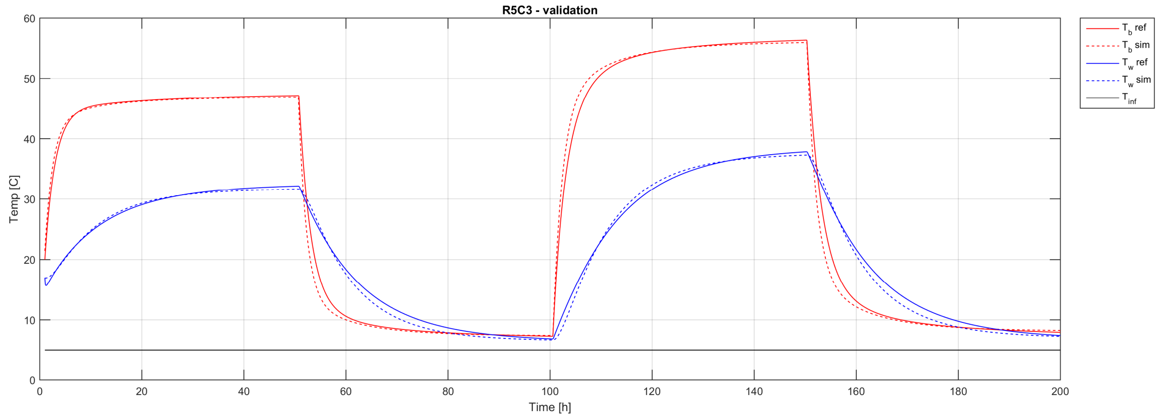


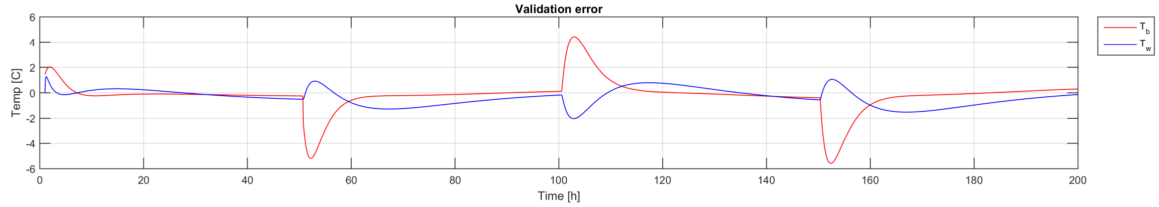


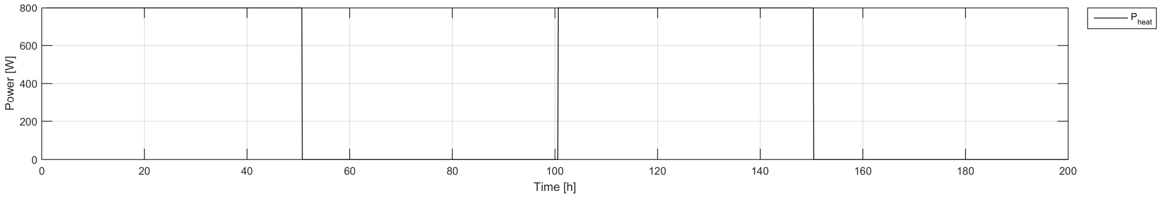


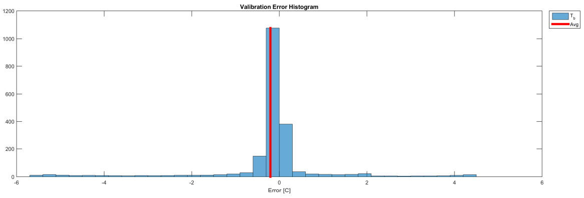
```
Simulation statistics
Total time     : 1.0 min
Timestep       : 360 sec
R_b    =              0.242068              (Initial = 0.300000)
R_w    =              0.426743              (Initial = 0.500000)
R_fur  =              0.337576              (Initial = 0.300000)
R_g    =              0.081158              (Initial = 0.080000)
R_vent =              0.198818              (Initial = 0.200000)
C_b    =              132926.073103         (Initial = 132000.000000)
C_w    =              186273.277238         (Initial = 190000.000000)
C_fur  =              5400055.179060        (Initial = 5400000.000000)

RMSEC: 1.000
Model type: R5C3
rmsep Tb: 1.273
rmsep Tw: 0.737
```

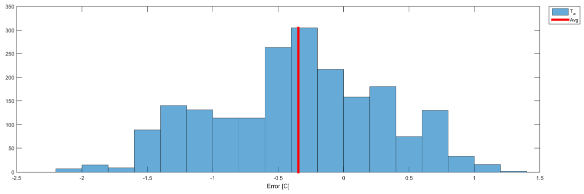
Validation









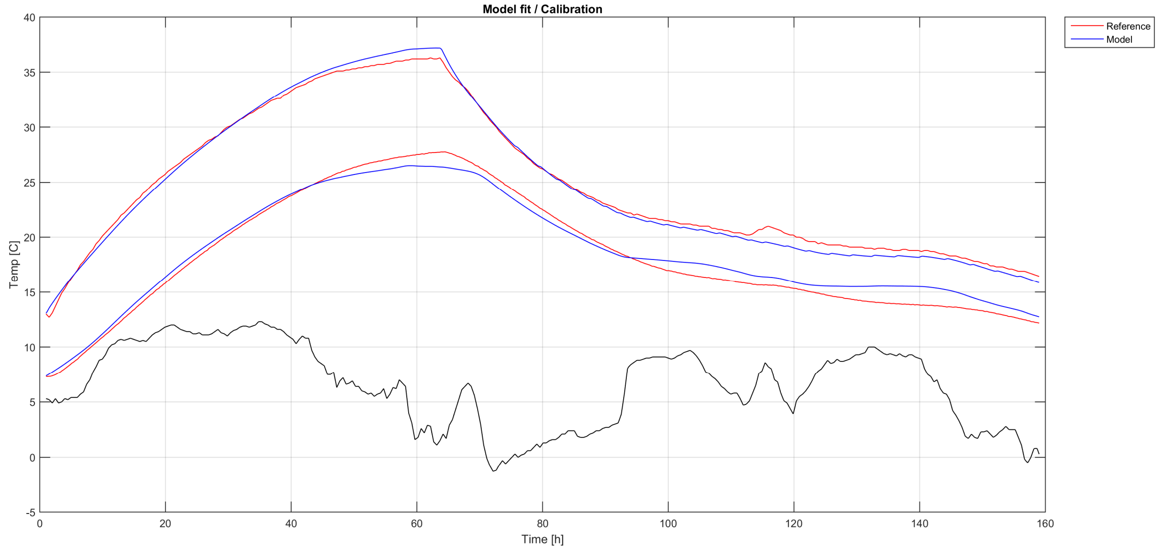


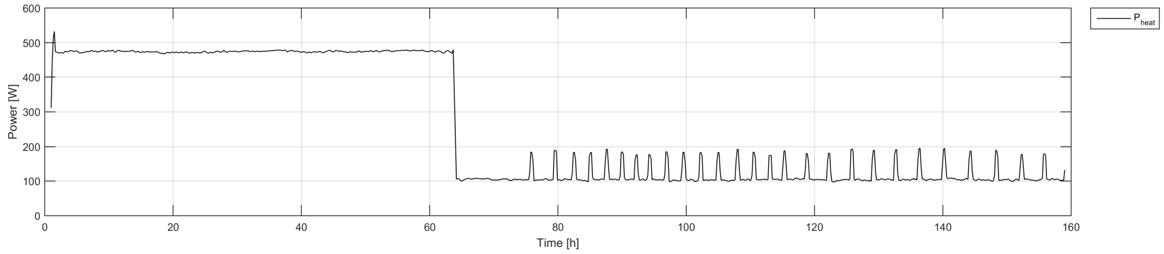
Description

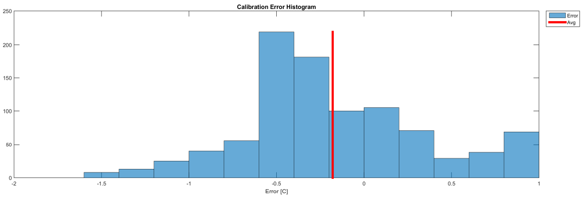
Same as case A, but using model R5C3.

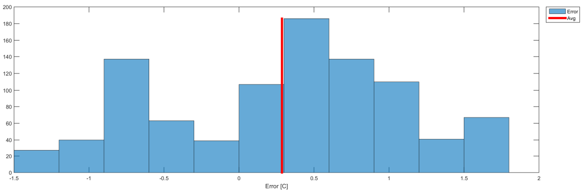
Case 1

Calibration



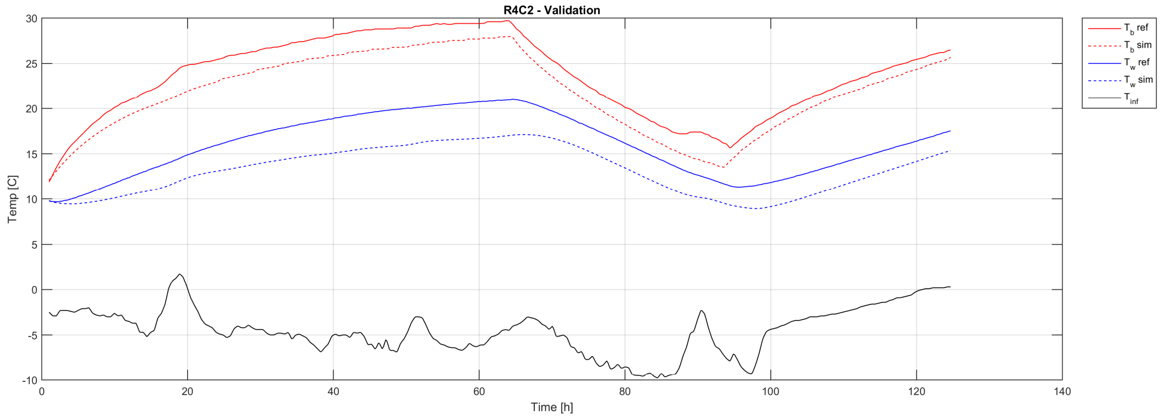


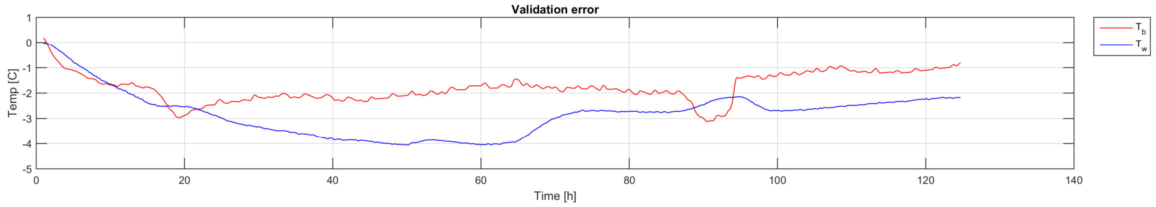


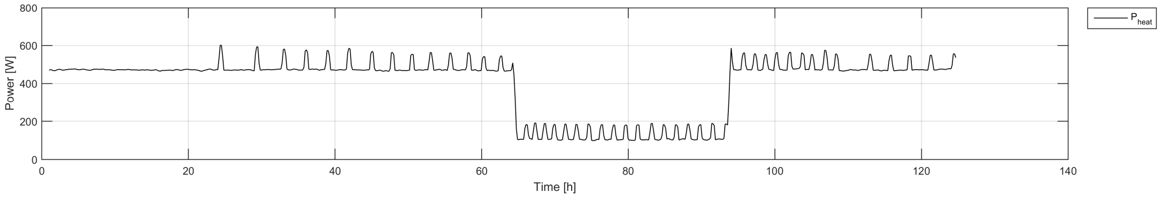


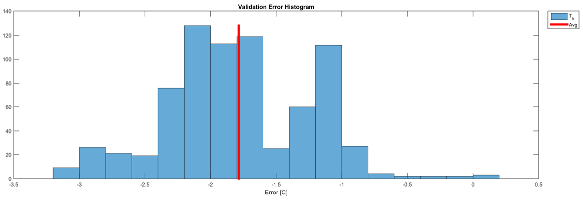
```
Simulation statistics
Total time     : 0.3 min
Timestep       : 600 sec

R_b    =              0.036387            (Initial = 0.040000)
R_w    =              0.080332            (Initial = 0.100000)
R_g    =              0.201898            (Initial = 0.300000)
R_vent =              0.330000            (Initial = 0.200000)
C_b    =              1107894.607319      (Initial = 1110000.000000)
C_w    =              1398321.529381      (Initial = 1400000.000000)

RMSEC: 1.012
Model type: R4C2
rmsep Tb: 1.869
rmsep Tw: 2.947
```

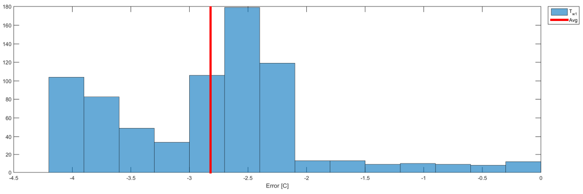
Validation











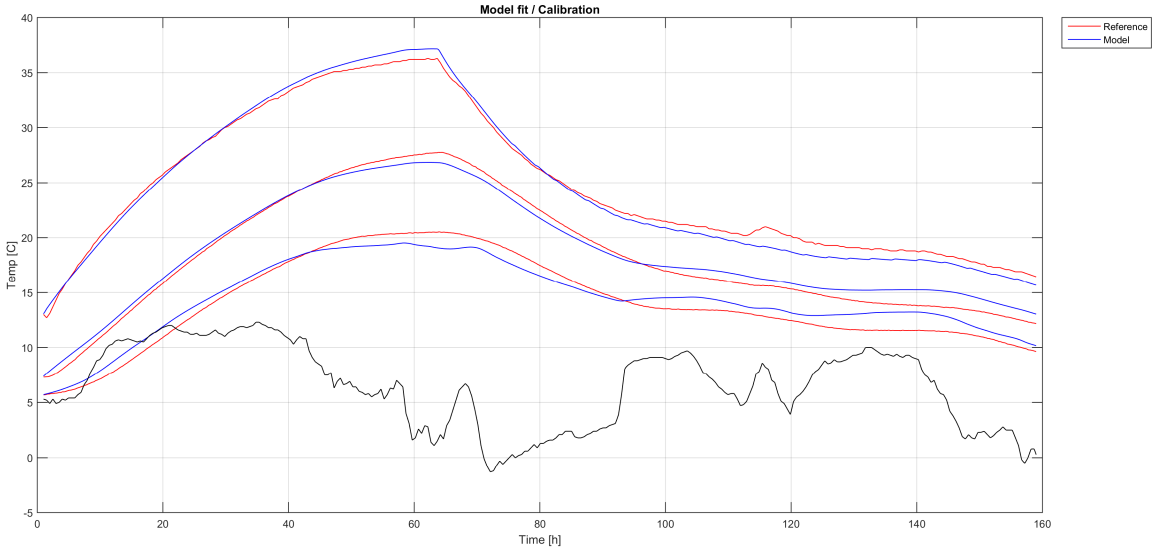
Description

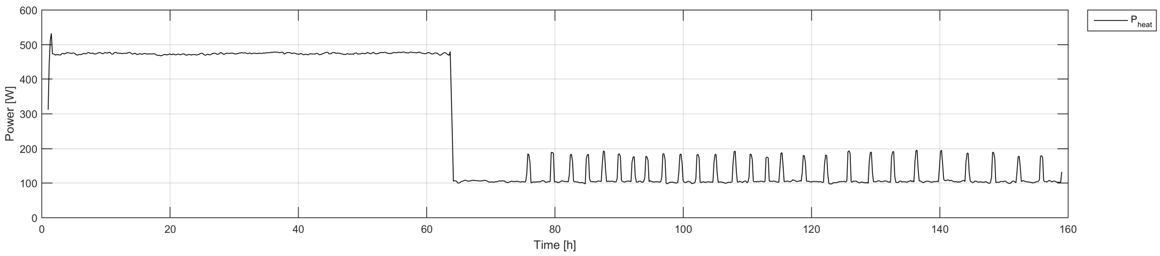
Based on data from “ByggLab” in the timeframe from 04.12.2015 18:00 to 17.12.2015 00:00. Samples 1-964 used for calibration and 1018 - 1766 for validation of the model. Data set contains step responses where heater thermostat is turned to max, thus giving the highest possible temperature in the building that the heater can sustain. This is shown by the heaters constant use of power. The reason for maximum temperature was to have largest possible dynamic changes in thermal behavior.

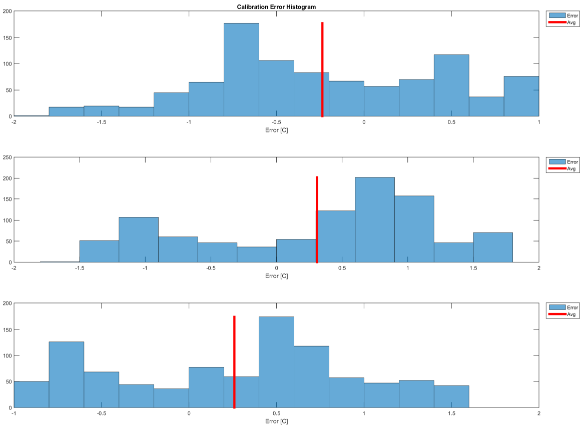
Small spikes in power are assumed caused by other equipment in the building (such as air humidifier). Note that there is a discrepancy in the range of outdoor temperatures between calibration and validation data.

Case 2

Calibration

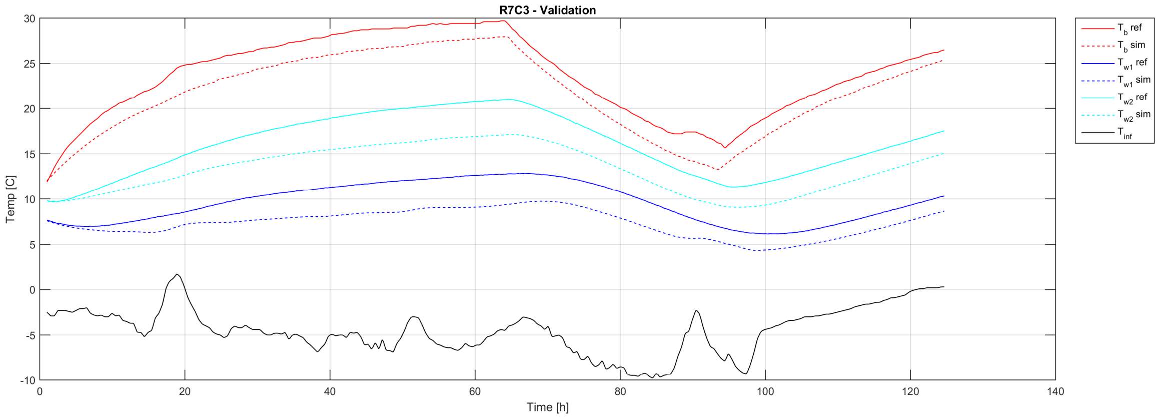


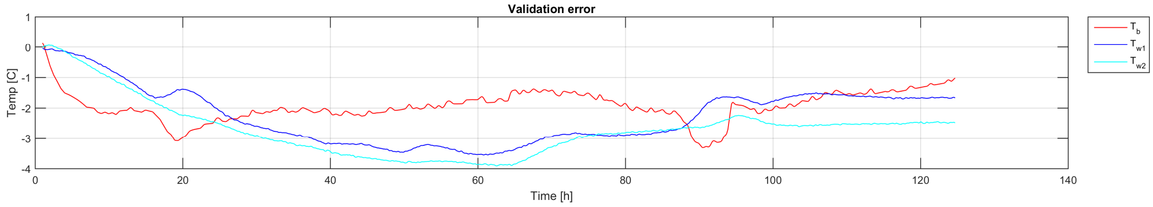


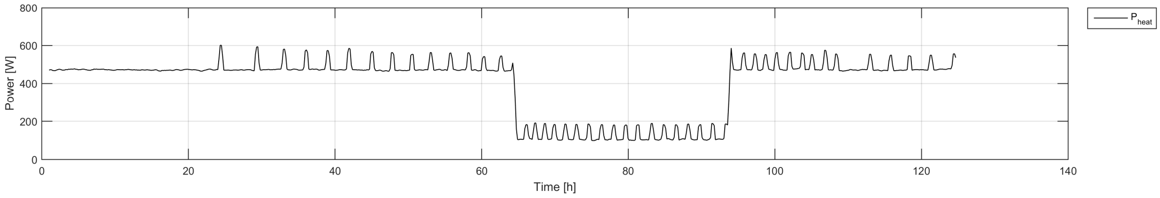


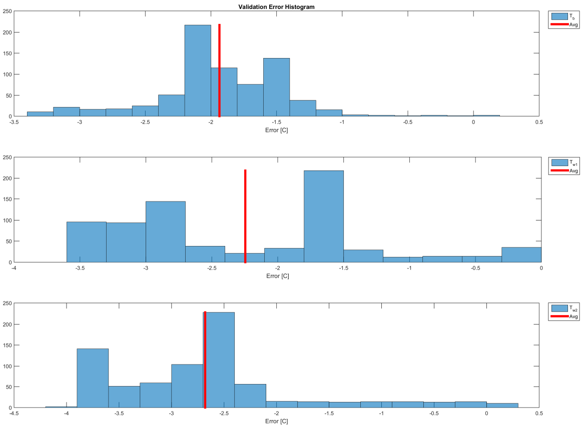
```
Simulation statistics
Total time     : 0.7 min
Timestep       : 600 sec
R_b    =                0.051108            (Initial = 0.060000)
R_w1   =                0.044626            (Initial = 0.080000)
R_w2   =                0.039092            (Initial = 0.020000)
R_s    =                0.001630            (Initial = 0.001000)
R_e    =                0.034231            (Initial = 0.050000)
R_g    =                0.128300            (Initial = 0.135000)
R_vent =                0.330000            (Initial = 0.200000)
C_b    =                1305475.975806      (Initial = 1310000.000000)
C_w1   =                1002396.655774      (Initial = 1000000.000000)
C_w2   =                487131.525411       (Initial = 500000.000000)
RMSEC: 1.383
Model type: R7C3
rmsep Tb: 1.994
rmsep Tw1: 2.426
rmsep Tw2: 2.826
```

Validation







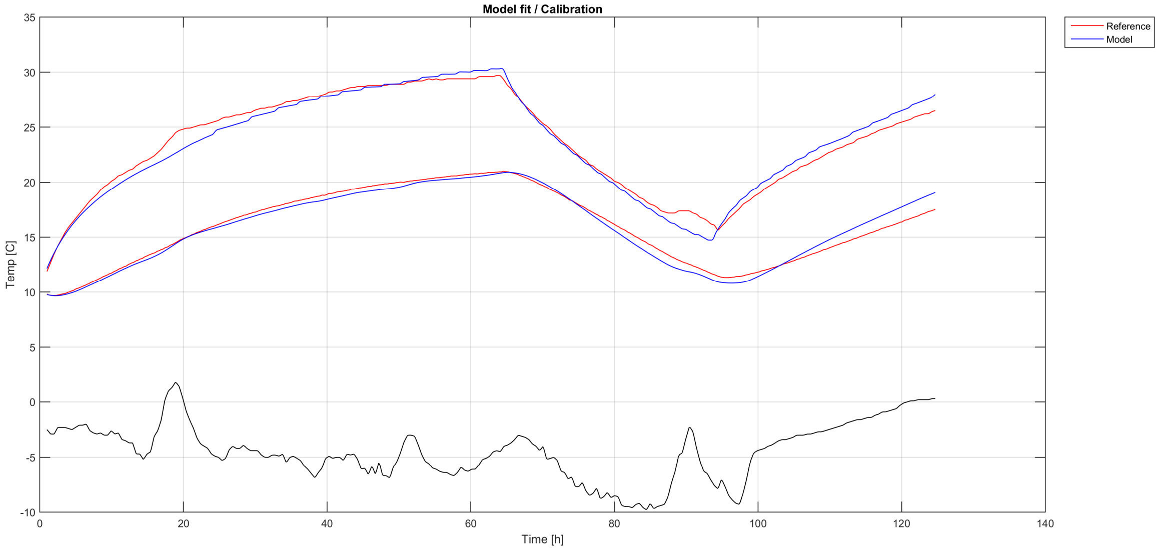


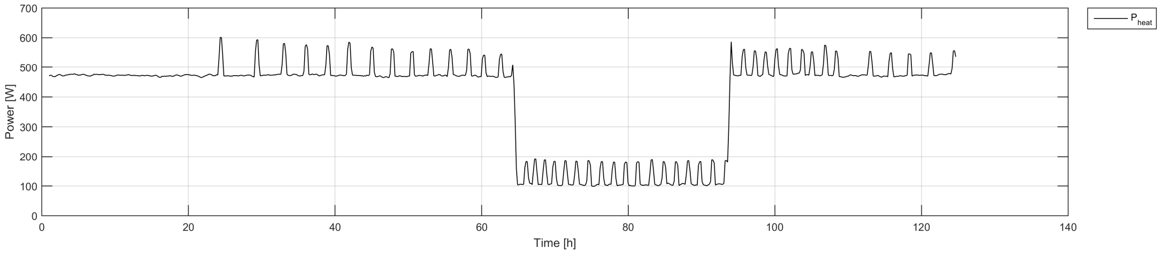
Description

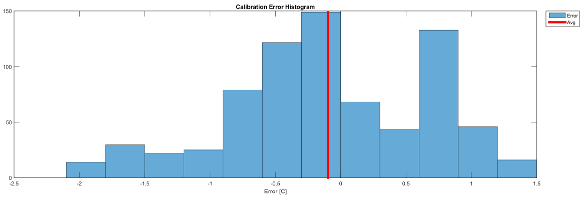
Same as case 1, but with model R7C3

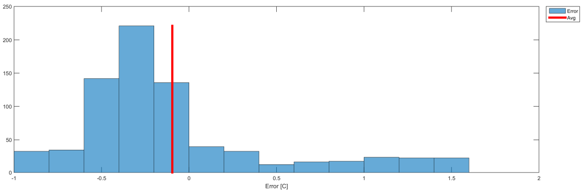
Case 3

Calibration



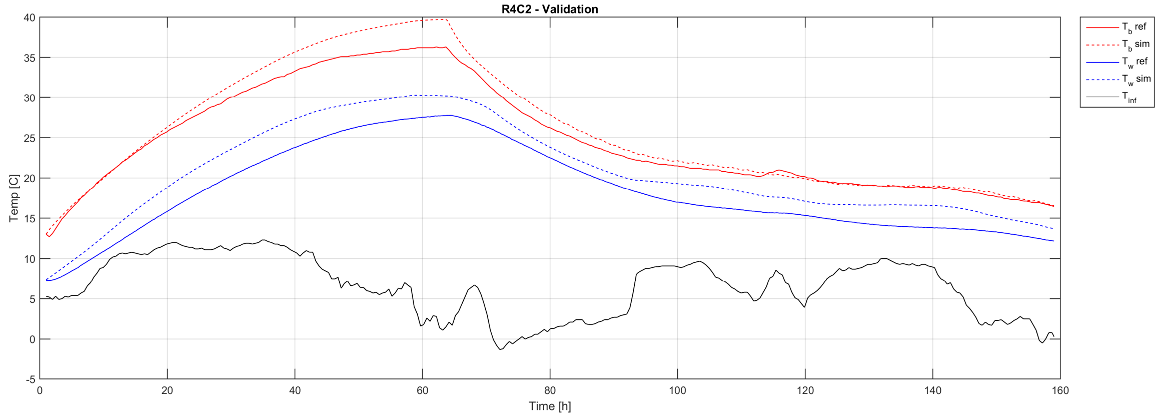


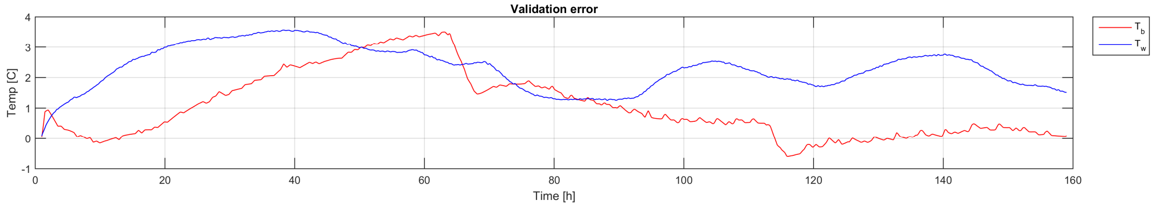


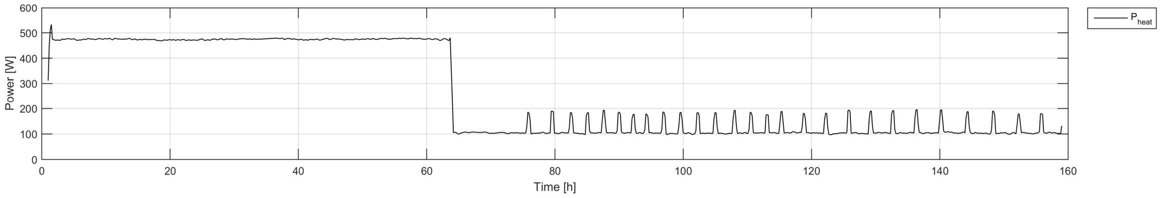


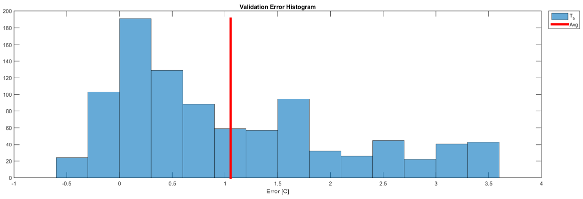
```
Simulation statistics
Total time     : 0.8 min
Timestep       : 600 sec

R_b    =                0.023466              (Initial = 0.040000)
R_w    =                0.069227              (Initial = 0.100000)
R_g    =                0.566107              (Initial = 0.300000)
R_vent =                0.329176              (Initial = 0.200000)
C_b    =                791360.856092         (Initial = 1110000.000000)
C_w    =                1414151.688278        (Initial = 1400000.000000)

RMSEC: 0.951
Model type: R4C2
rmsep Tb: 1.510
rmsep Tw: 2.424
```

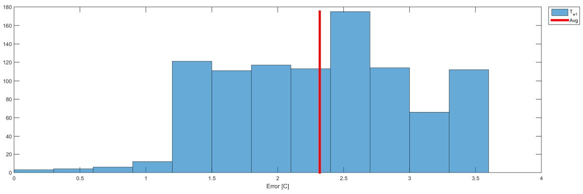
Validation









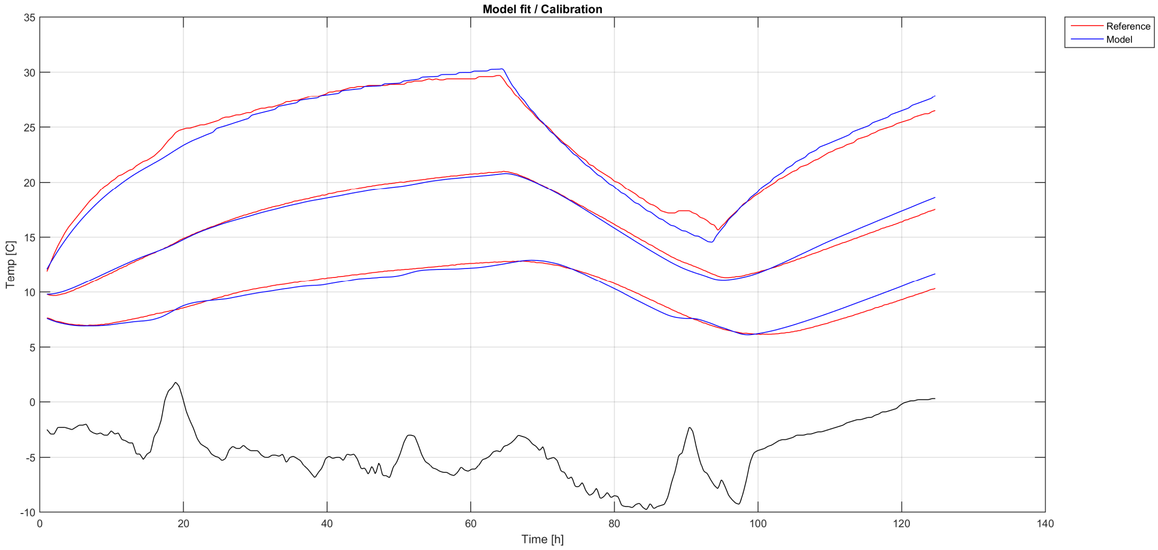


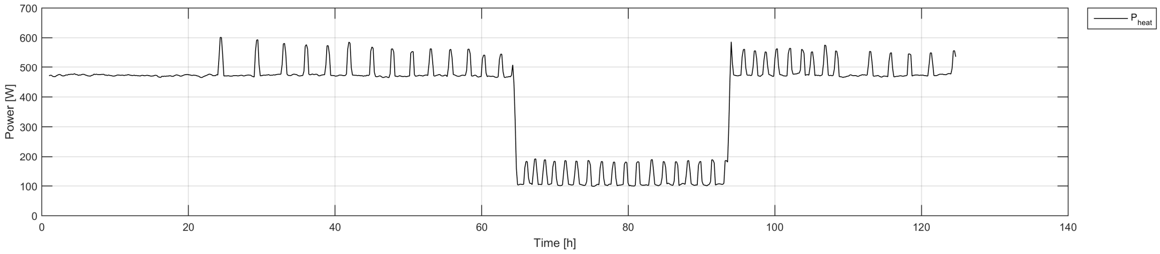
Description

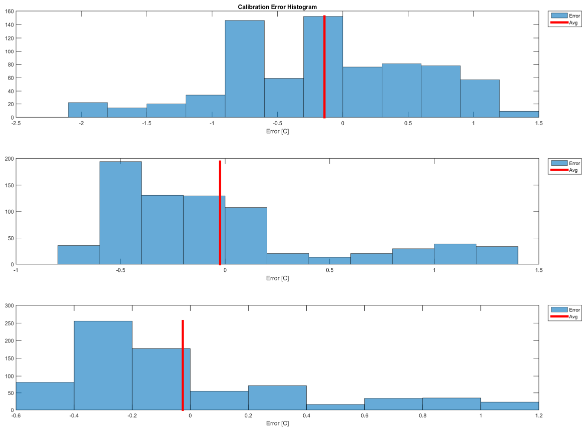
Same as case 1 but with calibration and validation data swapped (i.e. calibration data for case 1 is used as validation data for case 3, etc).

Case 4

Calibration

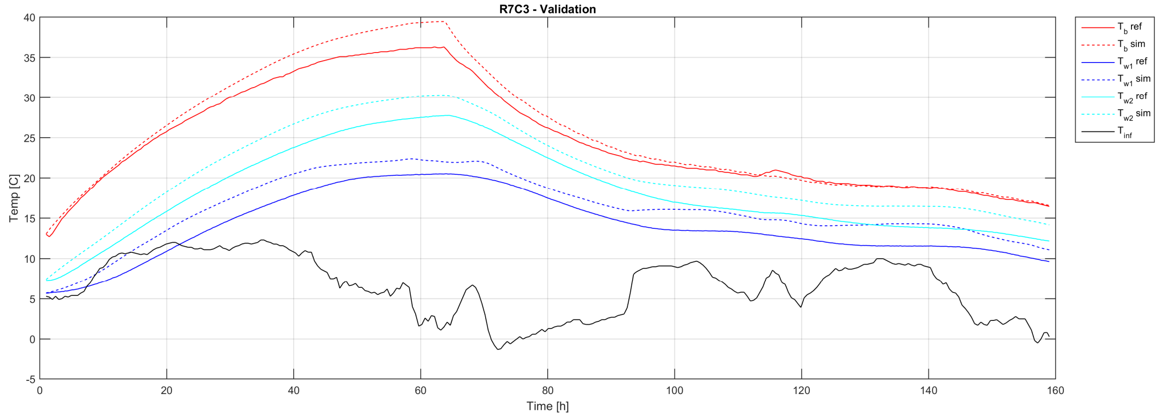


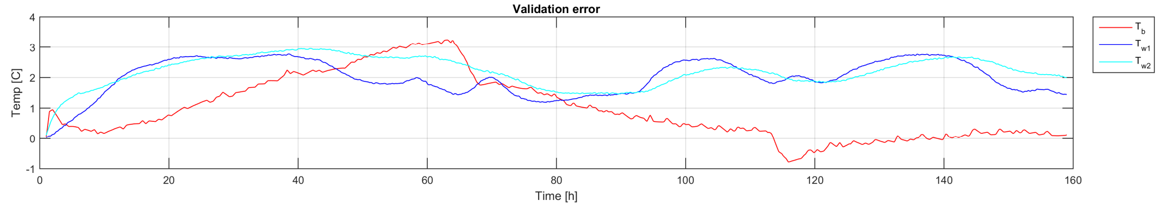


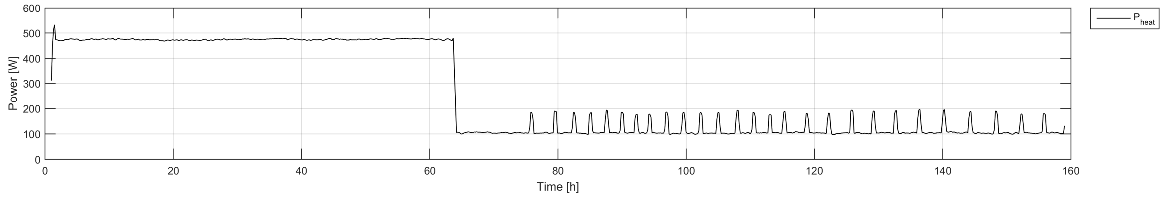


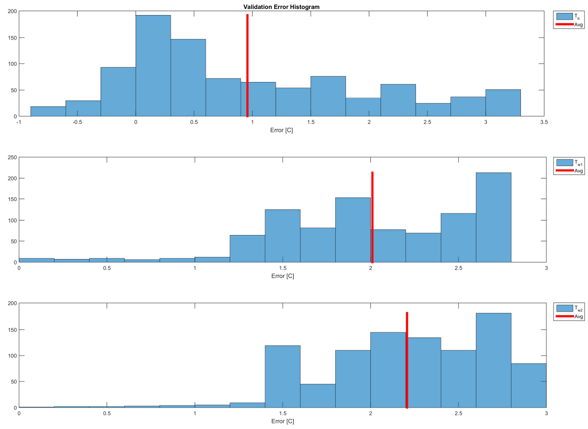
```
Simulation statistics
Total time      : 0.9 min
Timestep        : 600 sec
R_b     =                0.032667               (Initial = 0.060000)
R_w1    =                0.040553               (Initial = 0.080000)
R_w2    =                0.030970               (Initial = 0.020000)
R_s     =                0.001594               (Initial = 0.001000)
R_e     =                0.030686               (Initial = 0.050000)
R_g     =                0.188886               (Initial = 0.135000)
R_vent =                 0.330000               (Initial = 0.200000)
C_b     =                986571.542641          (Initial = 1310000.000000)
C_w1    =                998644.472573          (Initial = 1000000.000000)
C_w2    =                611869.633066          (Initial = 500000.000000)
RMSEC: 1.014
Model type: R7C3
rmsep Tb: 1.400
rmsep Tw1: 2.092
rmsep Tw2: 2.259
```

Validation







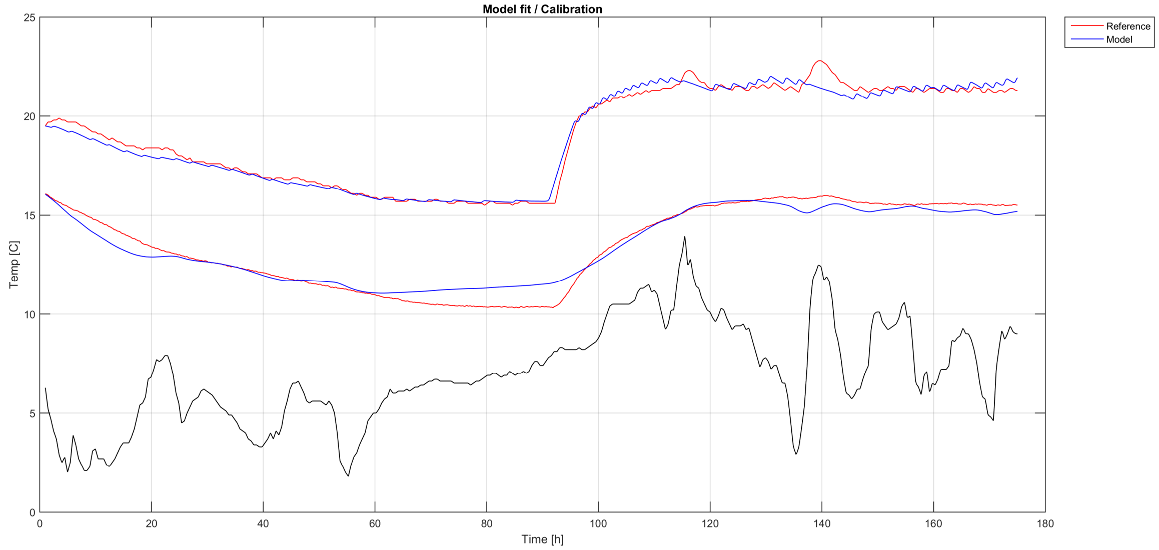


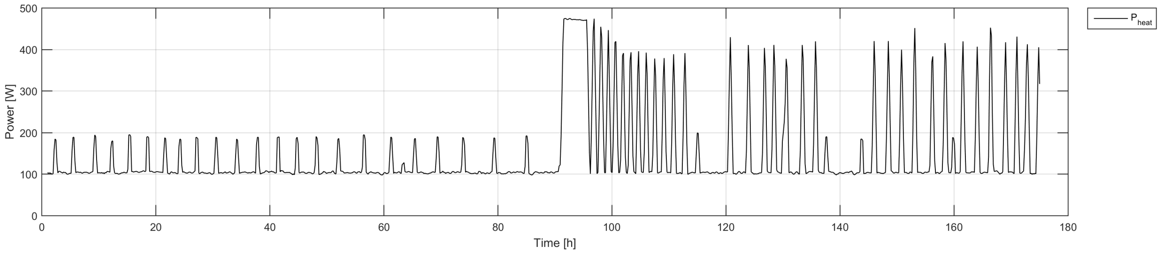
Description

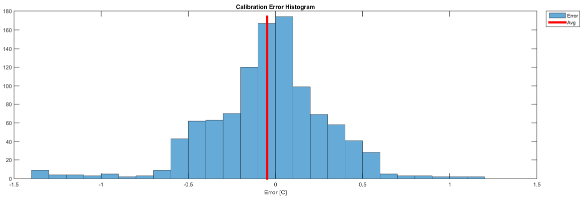
Same as case 3, but using model R7C3.

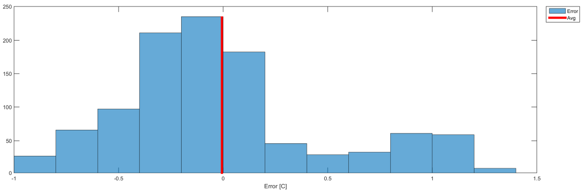
Case 5

Calibration



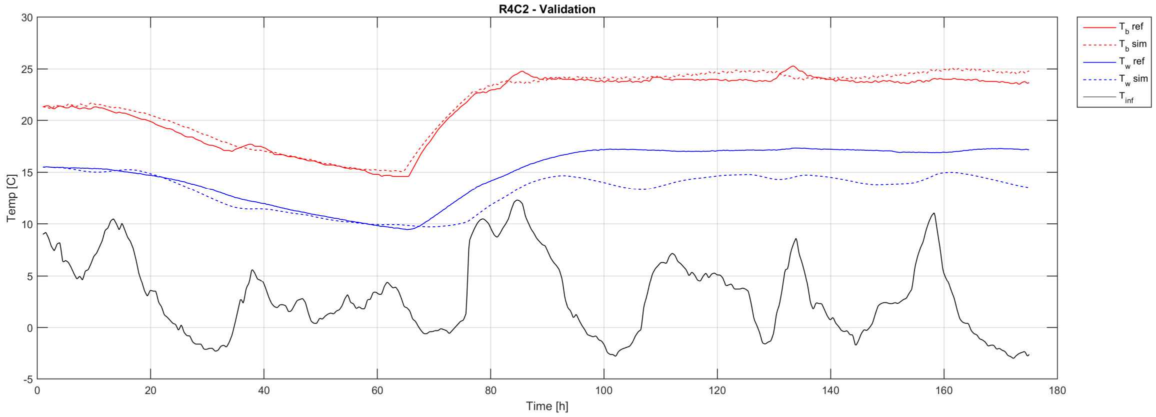


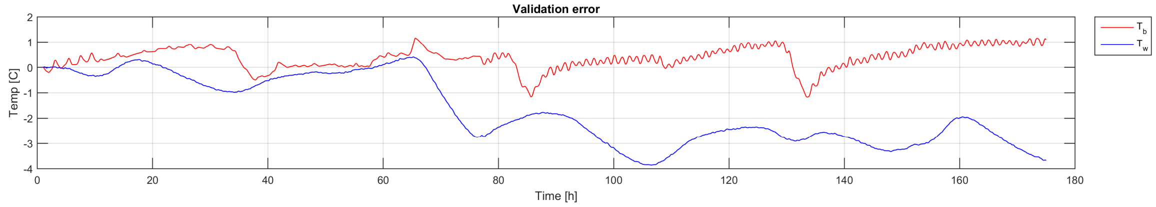


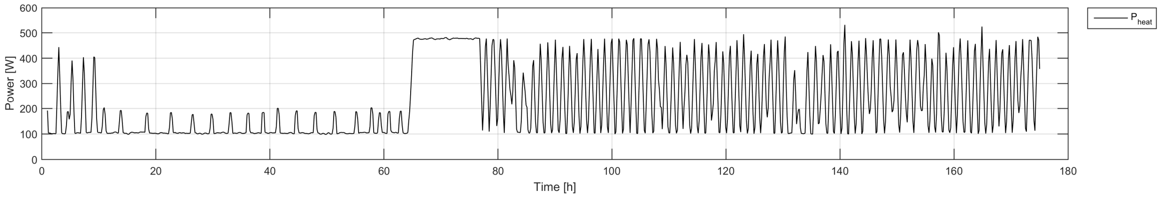


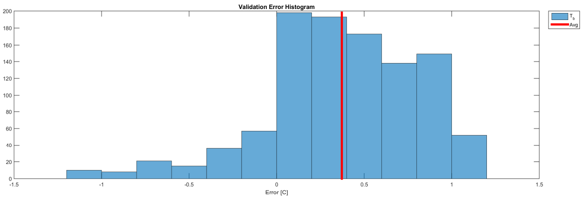
```
Simulation statistics
Total time      : 0.6 min
Timestep        : 600 sec

R_b    =                 0.069268                (Initial = 0.040000)
R_w    =                 0.079561                (Initial = 0.100000)
R_g    =                 0.185601                (Initial = 0.300000)
R_vent =                 0.330682                (Initial = 0.200000)
C_b    =                 1370105.274711          (Initial = 1110000.000000)
C_w    =                 1454574.102325          (Initial = 1400000.000000)

RMSEC: 0.602
Model type: R4C2
rmsep Tb: 0.576
rmsep Tw: 2.150
```

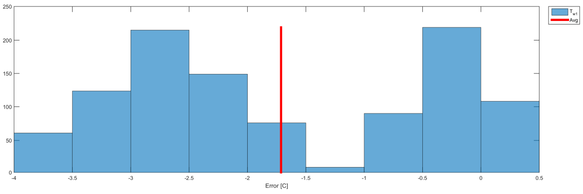
Validation









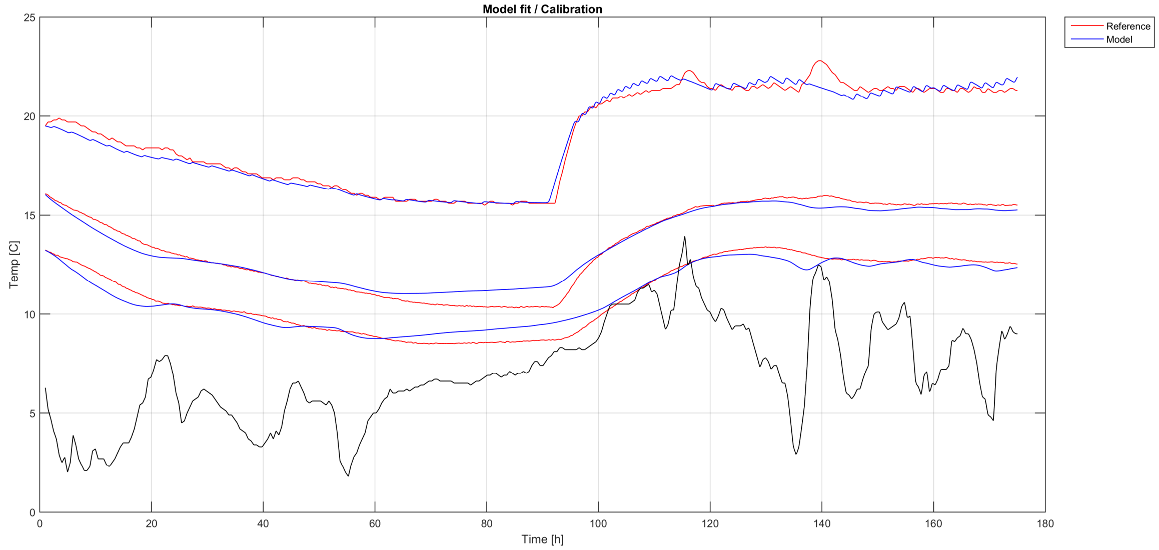


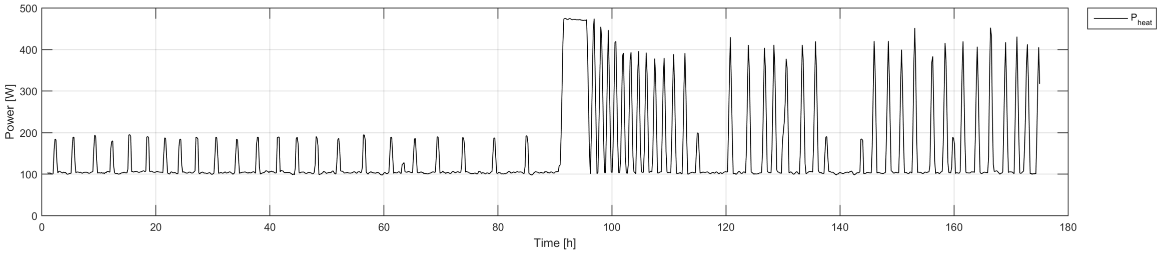
Description

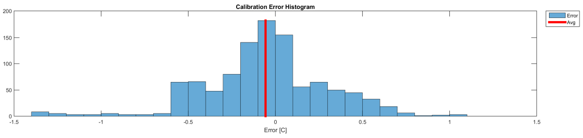
Data logged from “ByggeLab” in the timeframe from 02.11.2015 17:18 to 17.11.2015 07:16. The data set was resampled with timestep 10 minutes, giving a total of 2100 samples. Data is split in two such that samples 1- 1050 are used for calibration of the model, and 1051 to 2100 for validation. In case 5 model R4C2 is used. The data is taken while the building temperature is controlled by the heater onboard thermostat at a specific setpoint. The setpoint is in the range that would be comfortable to human occupants, rather then some maximum settings as in case 1 - 4, typically around 22°C. Calibration and validation data for case 5 show inputs in similar ranges, making them particularly interesting allowing models to be calibrated and validated on data of similar range.

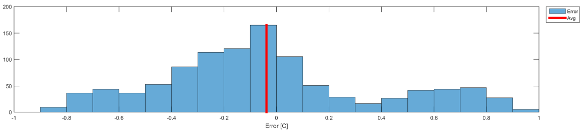
Case 6

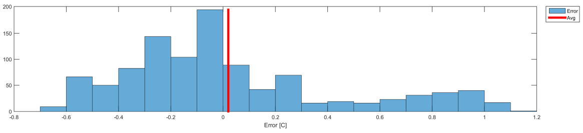
Calibration





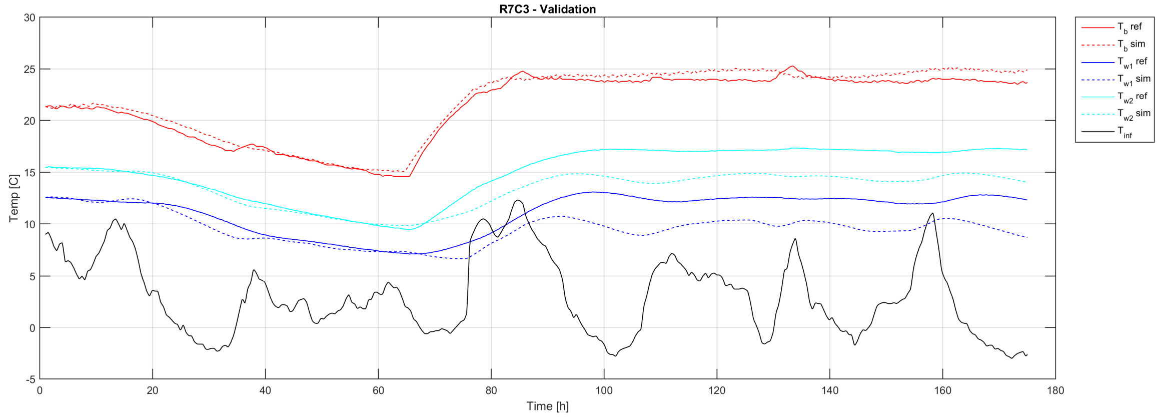


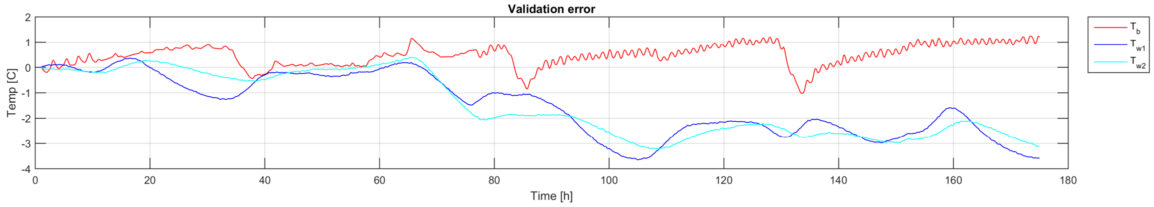


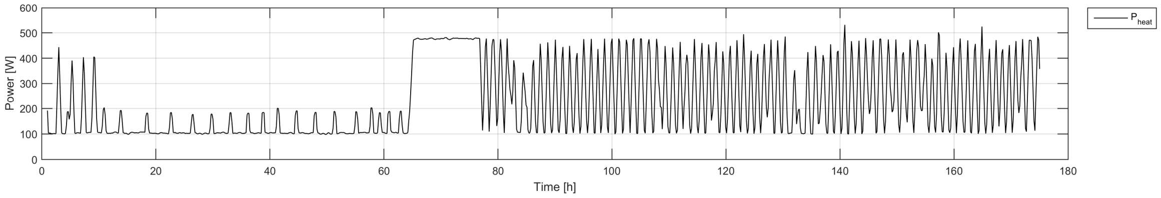


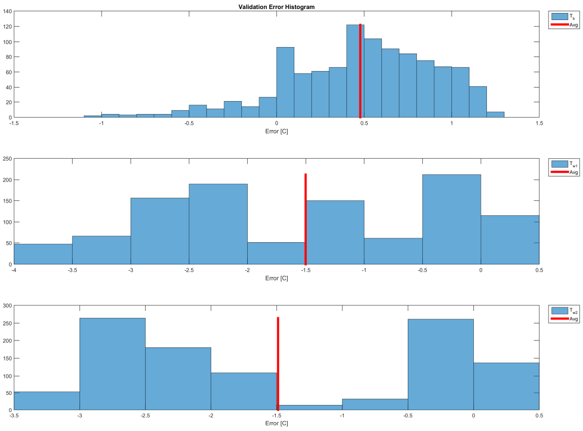
```
Simulation statistics
Total time     : 1.2 min
Timestep       : 600 sec
R_b    =                0.071423              (Initial = 0.060000)
R_w1   =                0.030028              (Initial = 0.080000)
R_w2   =                0.033602              (Initial = 0.020000)
R_s    =                0.001641              (Initial = 0.001000)
R_e    =                0.020918              (Initial = 0.050000)
R_g    =                0.173131              (Initial = 0.135000)
R_vent =                0.329998              (Initial = 0.200000)
C_b    =                1356046.205051        (Initial = 1310000.000000)
C_w1   =                1803684.779566        (Initial = 1000000.000000)
C_w2   =                484190.536511         (Initial = 500000.000000)
RMSEC: 0.685
Model type: R7C3
rmsep Tb: 0.639
rmsep Tw1: 1.912
rmsep Tw2: 1.923
```

Validation







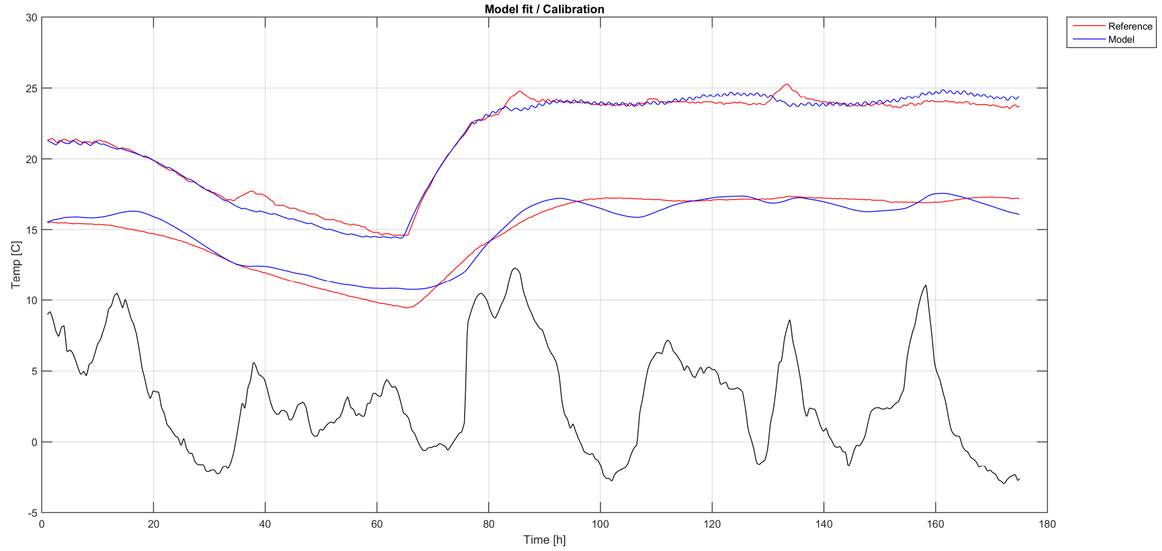


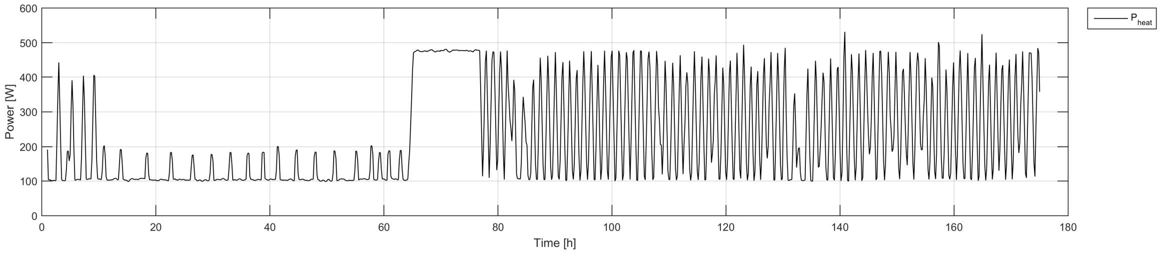
Description

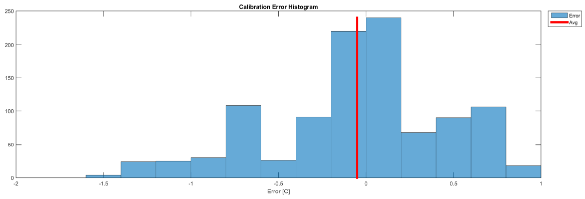
Same as case 5, but using model R7C3

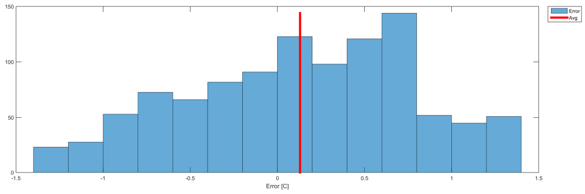
Case 7

Calibration



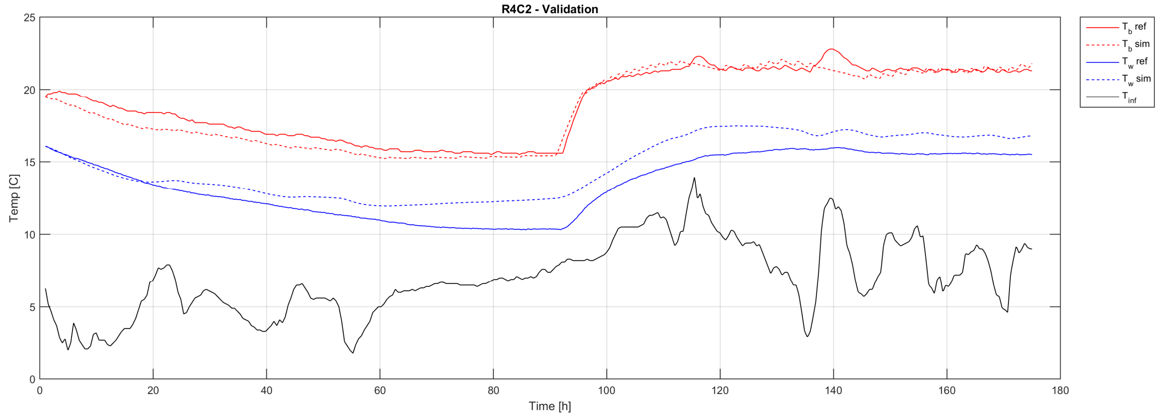


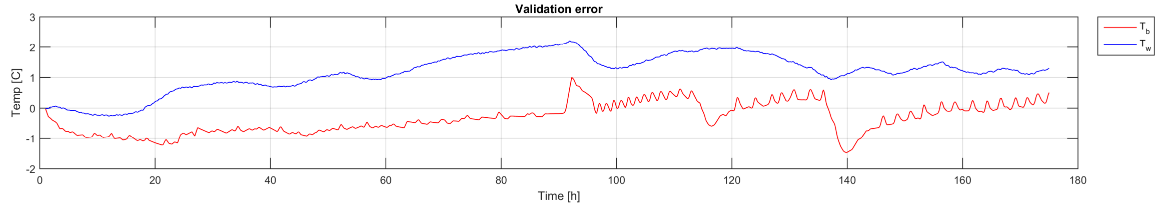


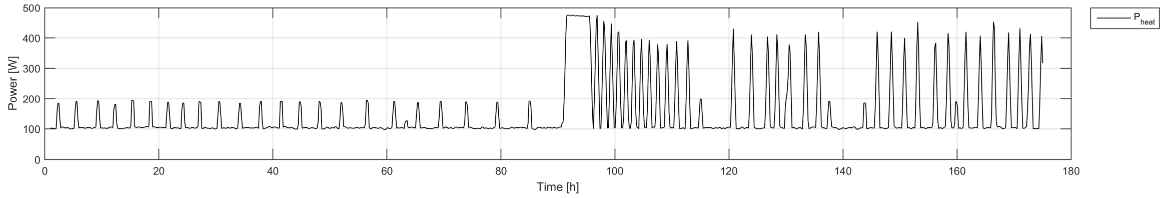


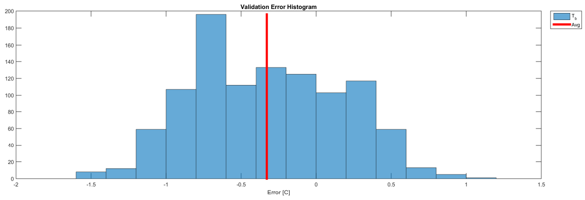
```
Simulation statistics
Total time     : 0.6 min
Timestep       : 600 sec

R_b    =                0.055901            (Initial = 0.040000)
R_w    =                0.108072            (Initial = 0.100000)
R_g    =                0.161105            (Initial = 0.300000)
R_vent =                0.330000            (Initial = 0.200000)
C_b    =                1231400.027247      (Initial = 1110000.000000)
C_w    =                1027141.440197      (Initial = 1400000.000000)


RMSEC: 0.837
Model type: R4C2
rmsep Tb: 0.590
rmsep Tw: 1.334
```

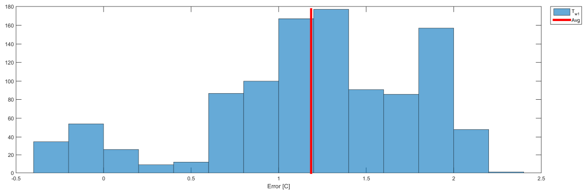
Validation









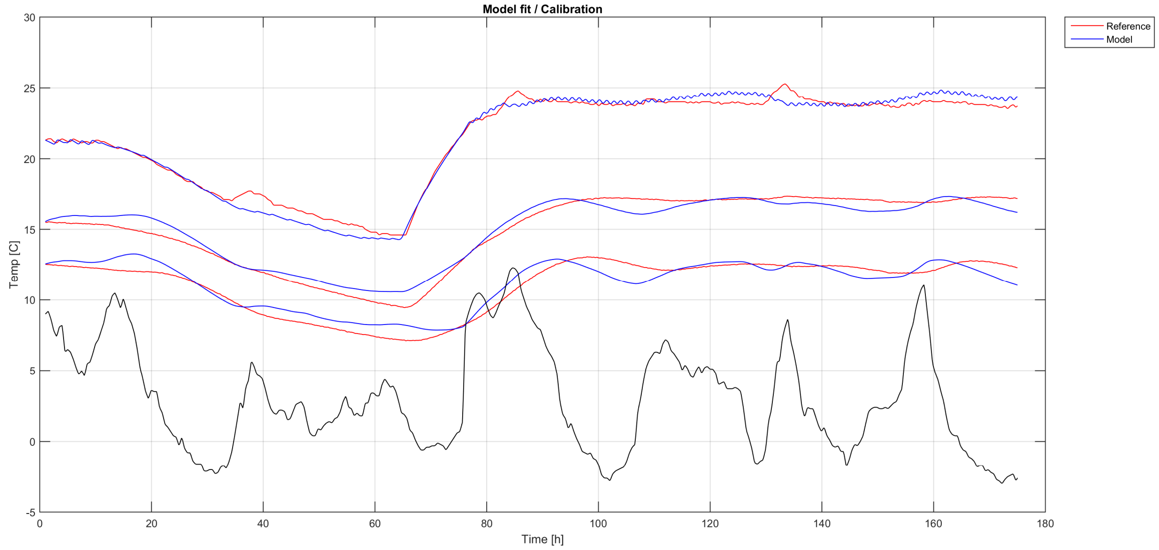


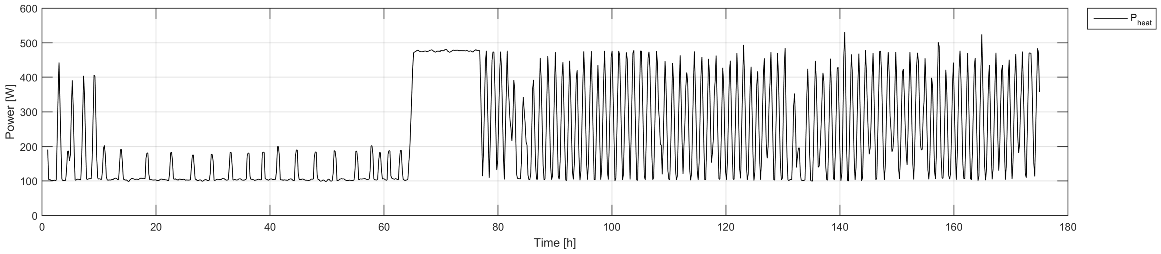
Description

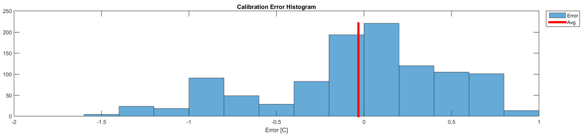
Same as case 5, but with calibration and validation data ranges swapped.

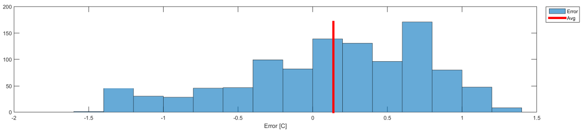
Case 8

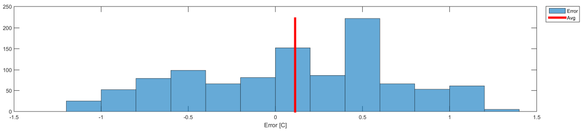
Calibration





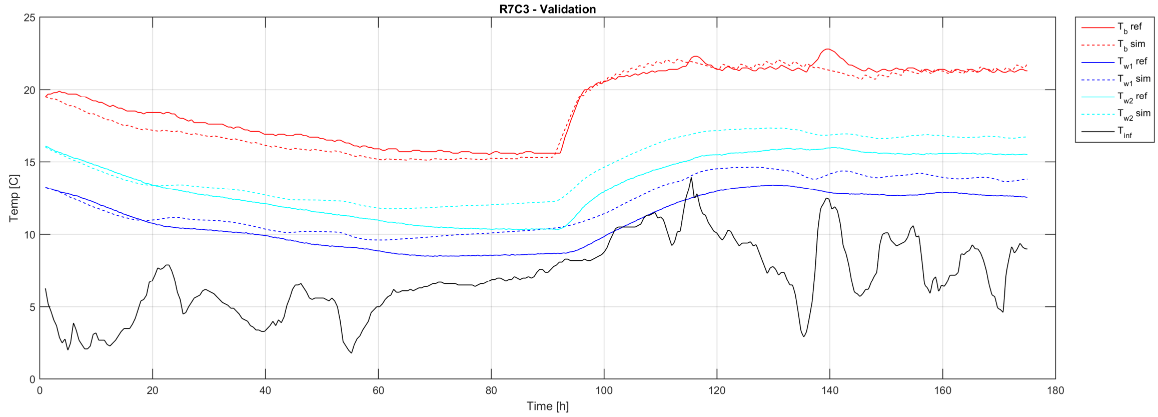


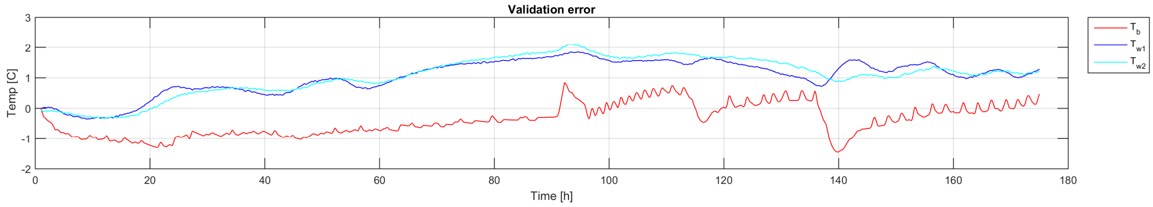


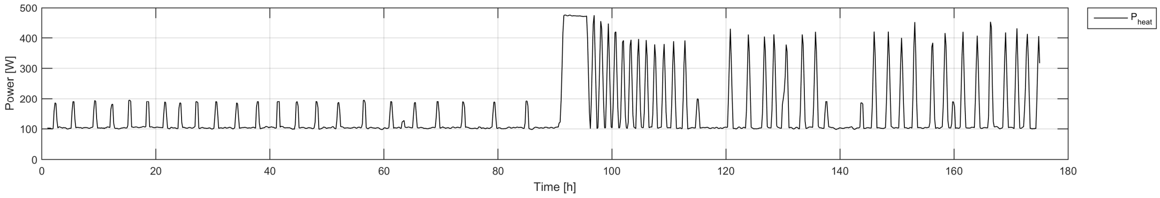


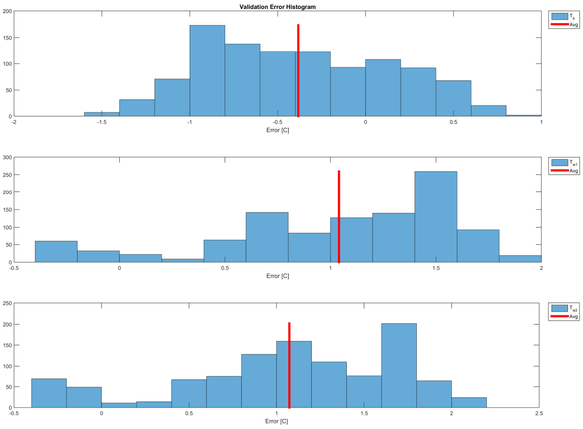
```
Simulation statistics
Total time     : 0.8 min
Timestep       : 600 sec
R_b    =                 0.065599               (Initial = 0.060000)
R_w1   =                 0.047750               (Initial = 0.080000)
R_w2   =                 0.041396               (Initial = 0.020000)
R_s    =                 0.001639               (Initial = 0.001000)
R_e    =                 0.036868               (Initial = 0.050000)
R_g    =                 0.139050               (Initial = 0.135000)
R_vent =                 0.330000               (Initial = 0.200000)
C_b    =                 1298656.225866         (Initial = 1310000.000000)
C_w1   =                 1070878.621398         (Initial = 1000000.000000)
C_w2   =                 212718.029709          (Initial = 500000.000000)
RMSEC: 1.014
Model type: R7C3
rmsep Tb: 0.645
rmsep Tw1: 1.182
rmsep Tw2: 1.241
```

Validation







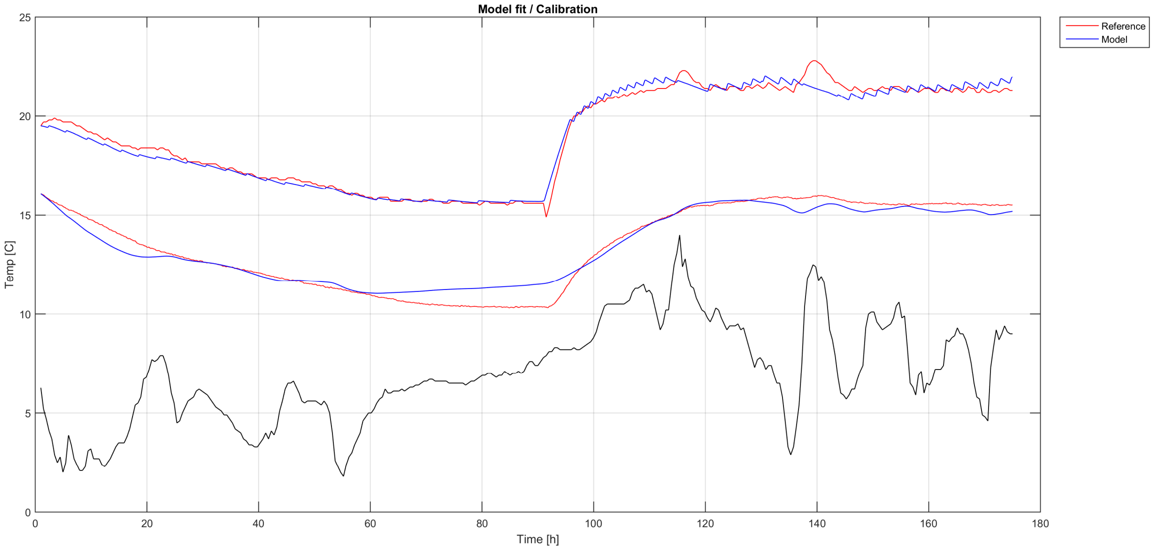


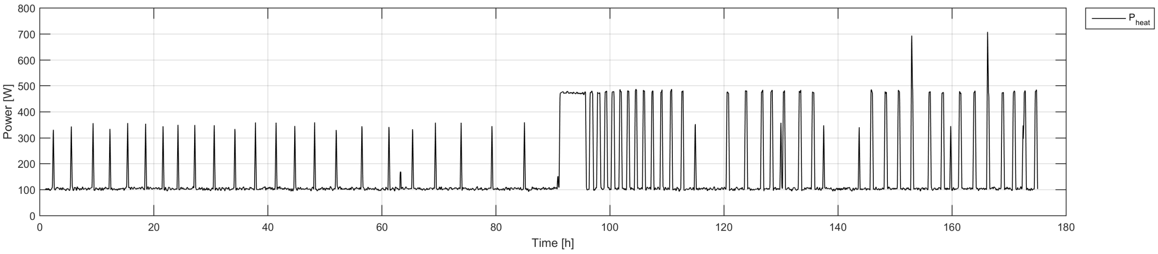
Description

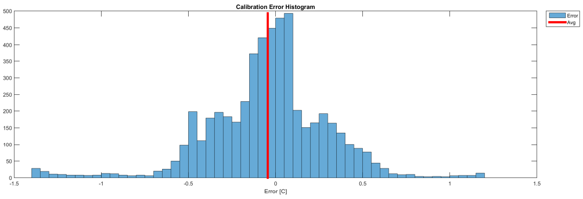
Same as case 6, but with calibration and validation data ranges swapped.

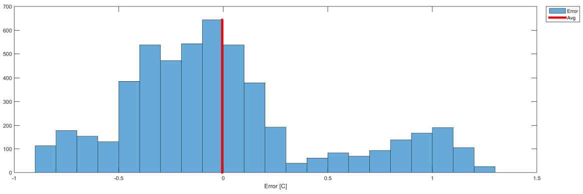
Case 9

Calibration



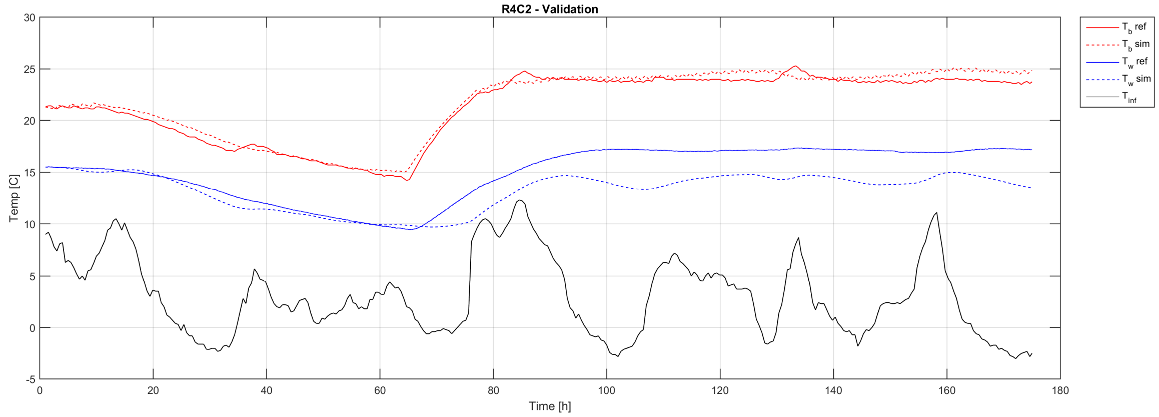


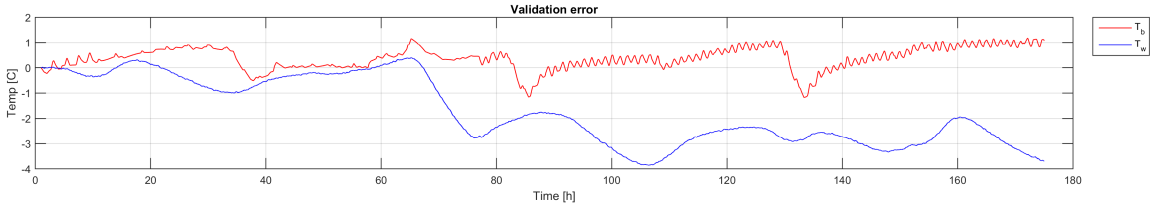


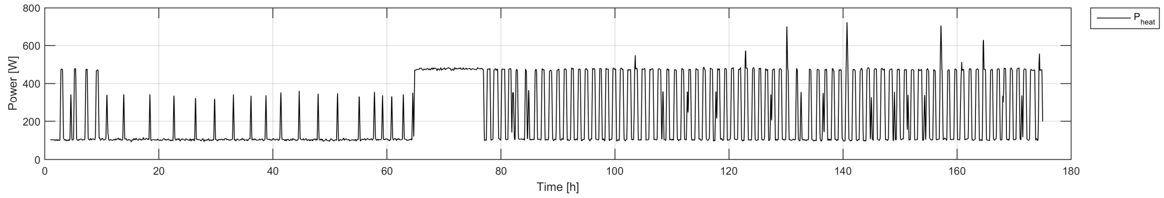


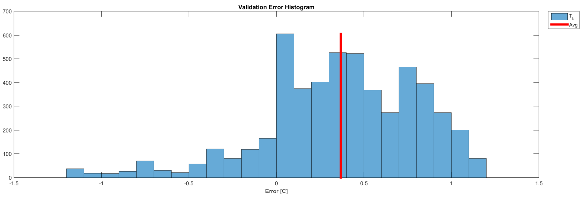
```
Simulation statistics
Total time     : 2.1 min
Timestep       : 120 sec
R_b     =                0.066961              (Initial = 0.040000)
R_w     =                0.076891              (Initial = 0.100000)
R_g     =                0.193564              (Initial = 0.300000)
R_vent =                 0.330228              (Initial = 0.200000)
C_b     =                1348657.593801        (Initial = 1110000.000000)
C_w     =                1492990.498436        (Initial = 1400000.000000)
RMSEC: 0.600
Model type: R4C2
rmsep Tb: 0.575
rmsep Tw: 2.152
```

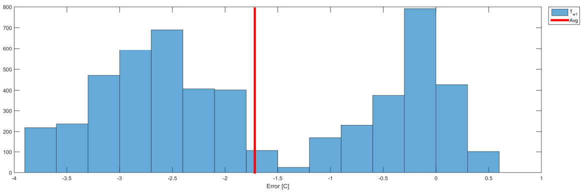
Validation









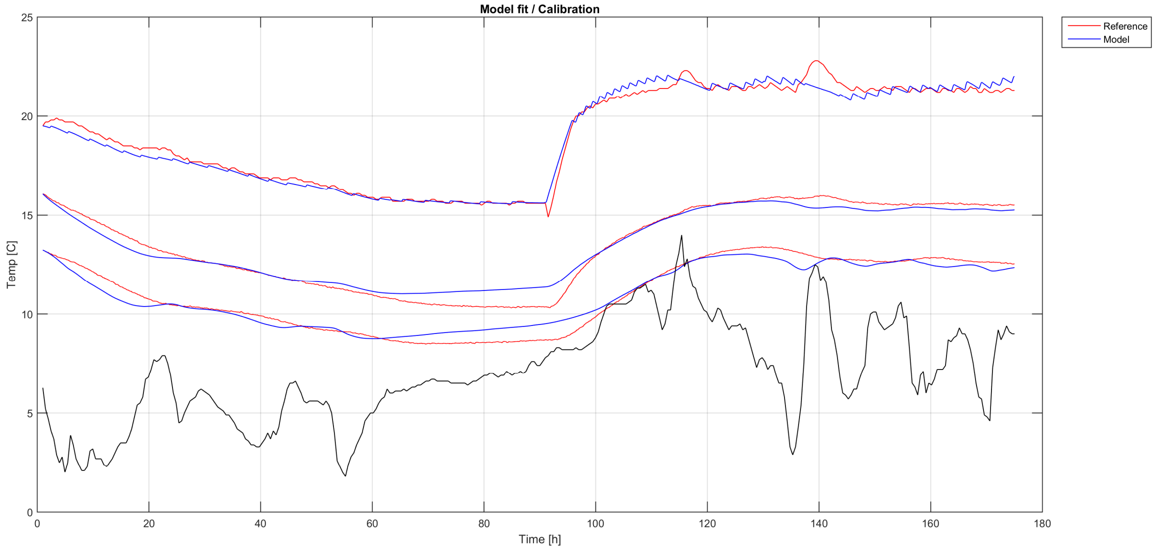


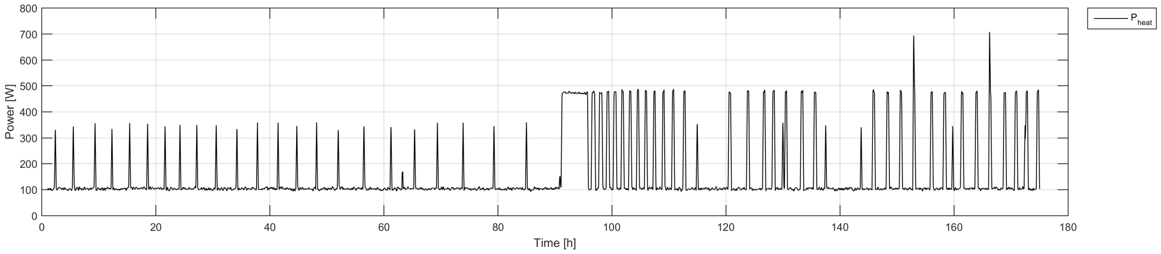
Description

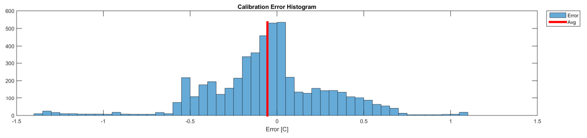
Same as case 5, but with 2 min intervals in the data (after resampling), thus increasing sample count by five. Calibration range is then 1 - 5250 and validation in range 5251 - 10500.

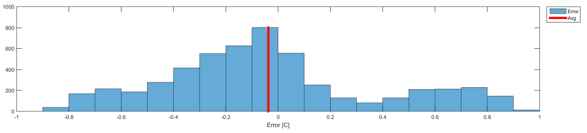
Case 10

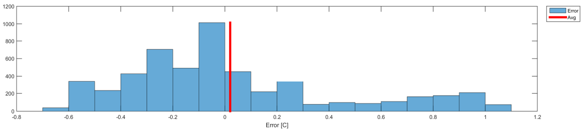
Calibration





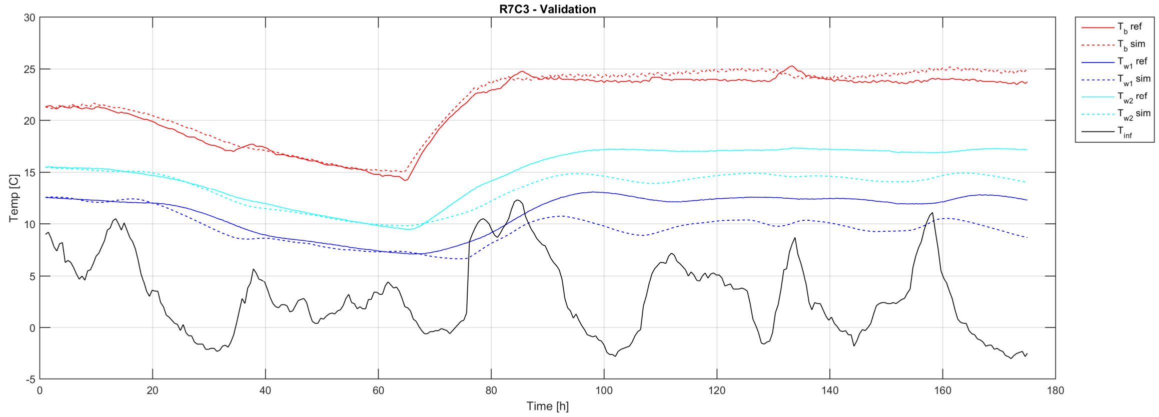


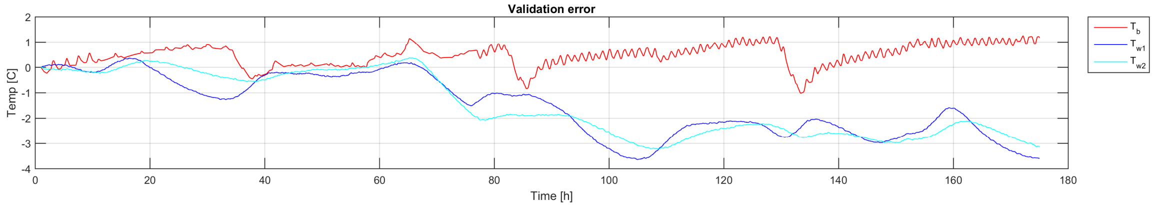


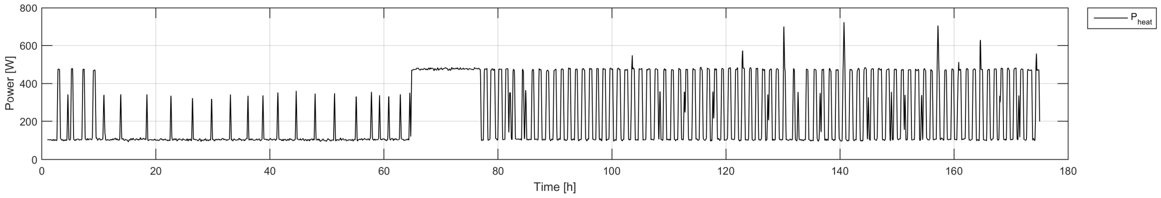


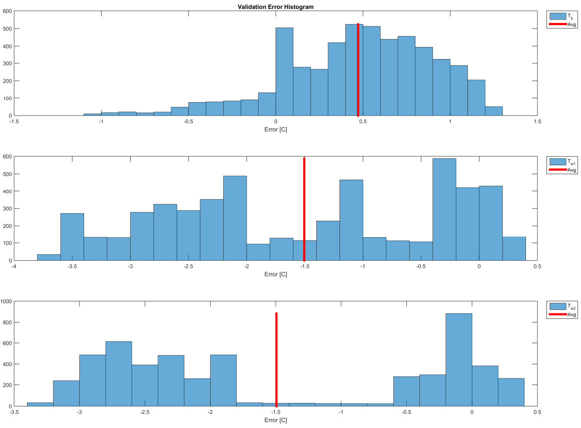
```
Simulation statistics
Total time     : 4.9 min
Timestep       : 120 sec
R_b    =                0.070494            (Initial = 0.060000)
R_w1   =                0.029697            (Initial = 0.080000)
R_w2   =                0.033170            (Initial = 0.020000)
R_s    =                0.001641            (Initial = 0.001000)
R_e    =                0.020600            (Initial = 0.050000)
R_g    =                0.175438            (Initial = 0.135000)
R_vent =                0.329990            (Initial = 0.200000)
C_b    =                1347356.243500      (Initial = 1310000.000000)
C_w1   =                1821632.125297      (Initial = 1000000.000000)
C_w2   =                484679.215477       (Initial = 500000.000000)
RMSEC: 0.681
Model type: R7C3
rmsep Tb: 0.637
rmsep Tw1: 1.912
rmsep Tw2: 1.925
```

Validation







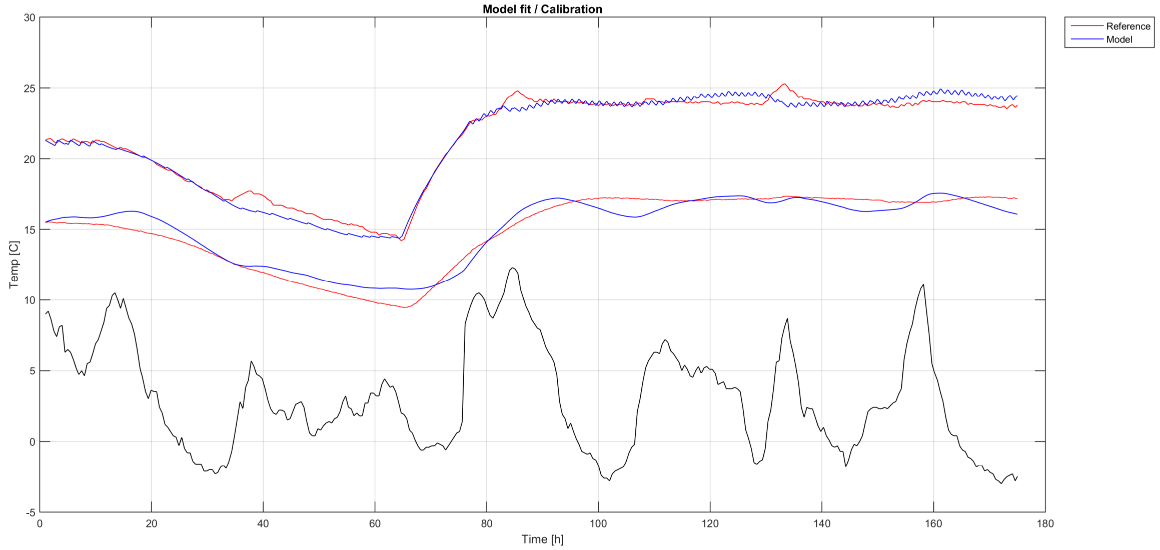


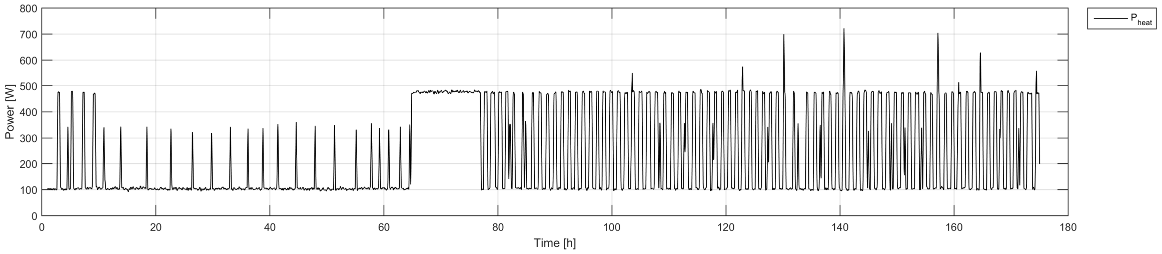
Description

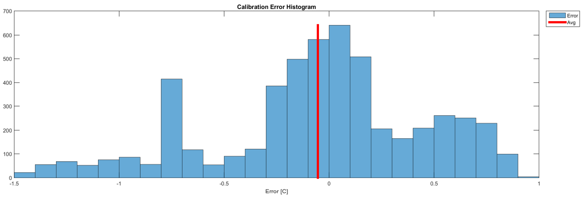
Same as case 9, but using model R7C3

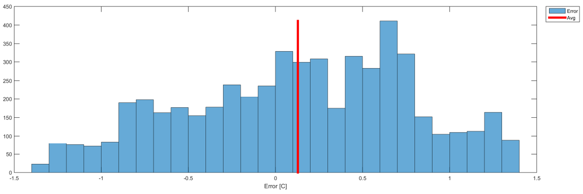
Case 11

Calibration



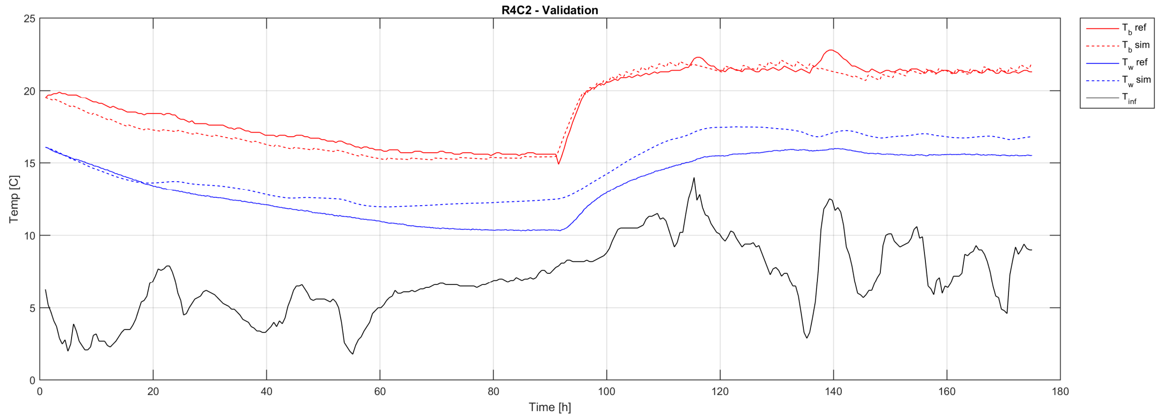


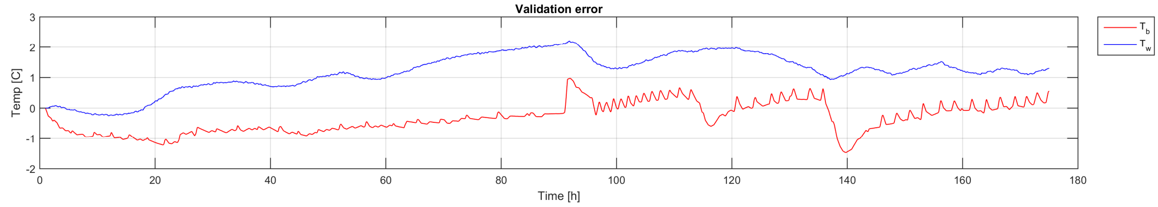


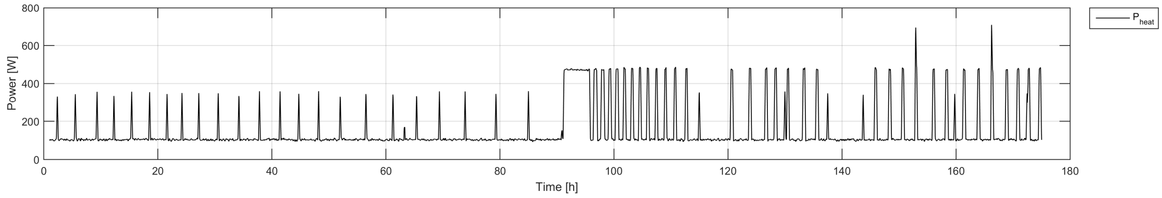


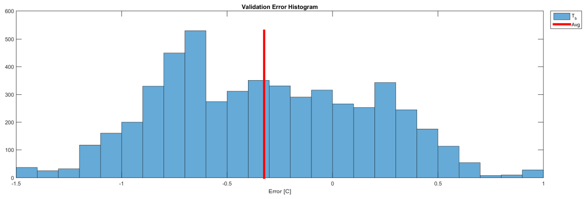
```
Simulation statistics
Total time     : 2.4 min
Timestep       : 120 sec

R_b    =                0.054909              (Initial = 0.040000)
R_w    =                0.106218              (Initial = 0.100000)
R_g    =                0.163950              (Initial = 0.300000)
R_vent =                0.329894              (Initial = 0.200000)
C_b    =                1221509.512024        (Initial = 1110000.000000)
C_w    =                1044410.223996        (Initial = 1400000.000000)

RMSEC: 0.836
Model type: R4C2
rmsep Tb: 0.586
rmsep Tw: 1.335
```

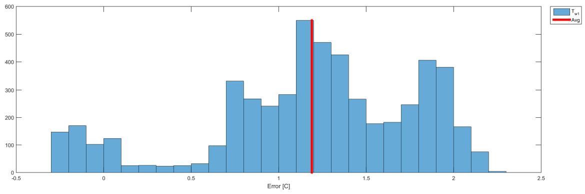
Validation









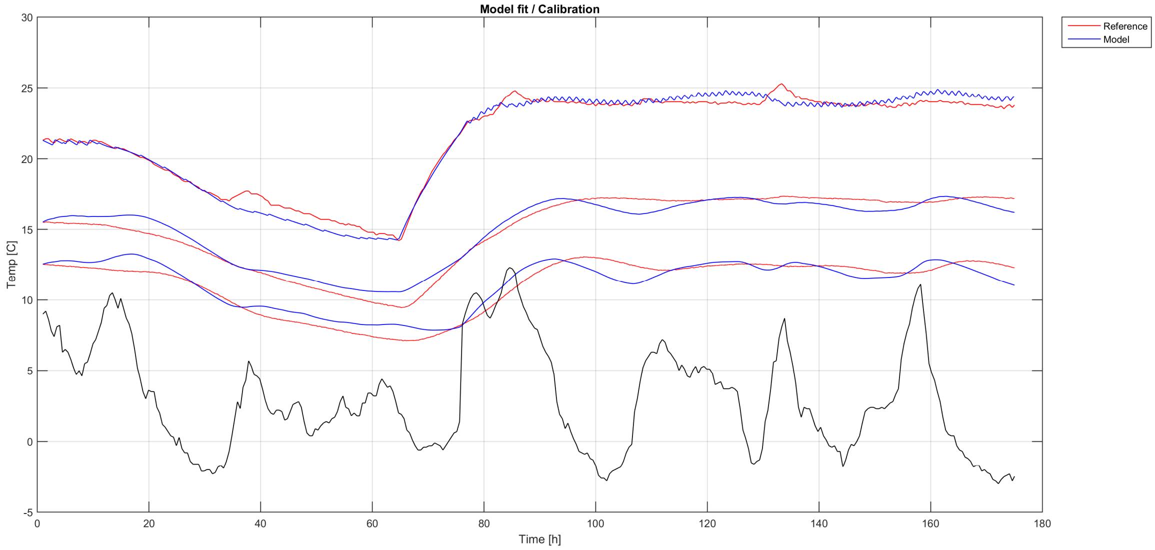


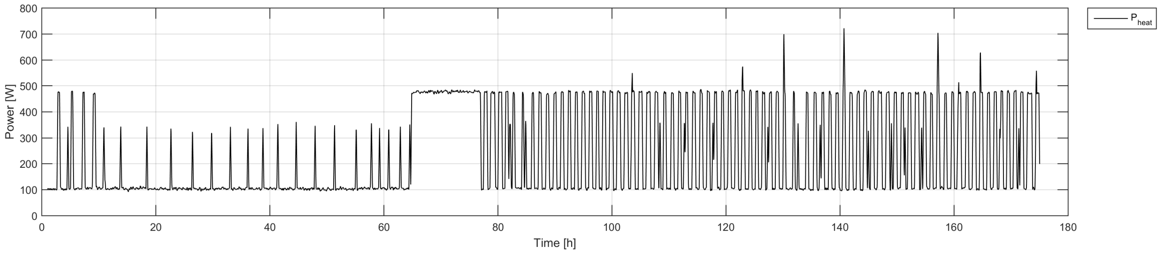
Description

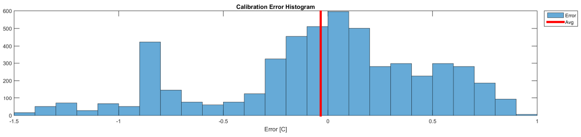
Same as case 9, but with calibration and validatin ranges swapped.

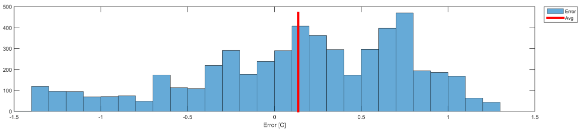
Case 12

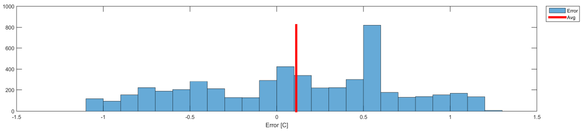
Calibration





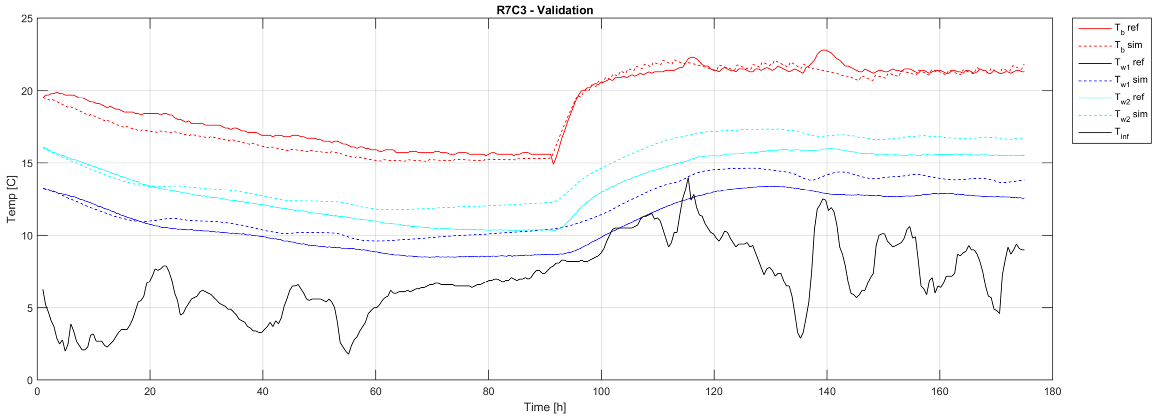


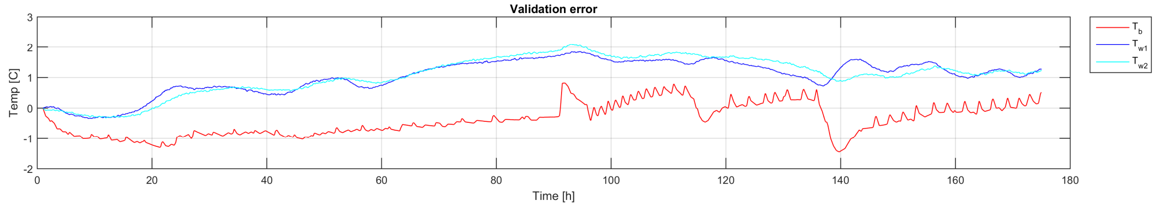


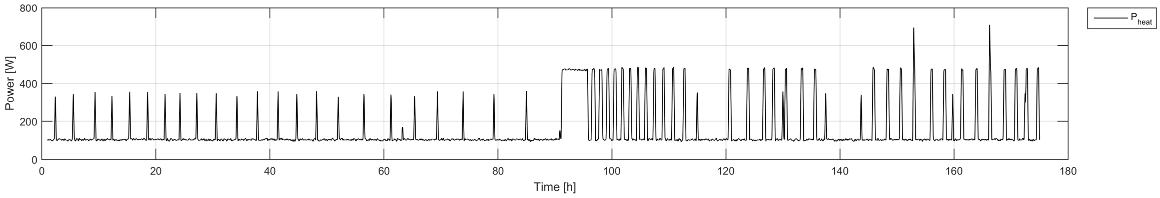


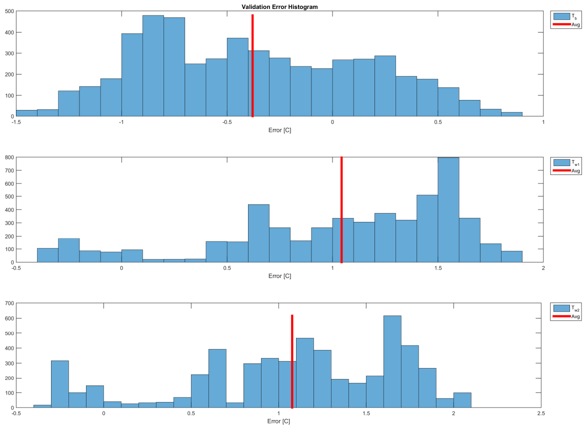
```
Simulation statistics
Total time     : 3.0 min
Timestep       : 120 sec
R_b    =                0.065126            (Initial = 0.060000)
R_w1   =                0.047461            (Initial = 0.080000)
R_w2   =                0.041141            (Initial = 0.020000)
R_s    =                0.001639            (Initial = 0.001000)
R_e    =                0.036619            (Initial = 0.050000)
R_g    =                0.139713            (Initial = 0.135000)
R_vent =                0.330052            (Initial = 0.200000)
C_b    =                1295058.575120      (Initial = 1310000.000000)
C_w1   =                1076282.116937      (Initial = 1000000.000000)
C_w2   =                212833.480922       (Initial = 500000.000000)
RMSEC: 1.009
Model type: R7C3
rmsep Tb: 0.641
rmsep Tw1: 1.183
rmsep Tw2: 1.243
```

Validation







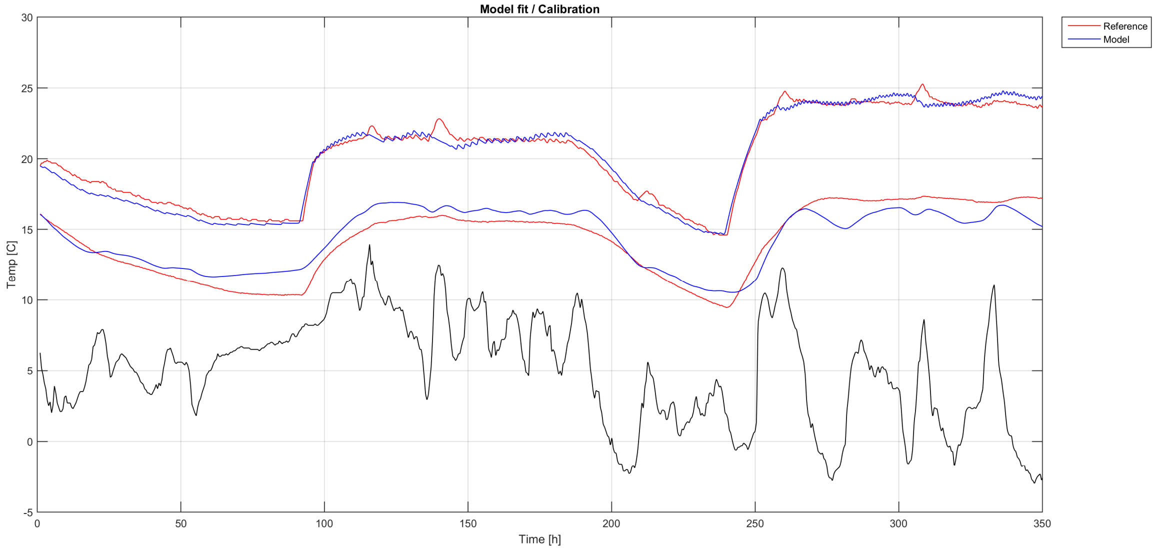


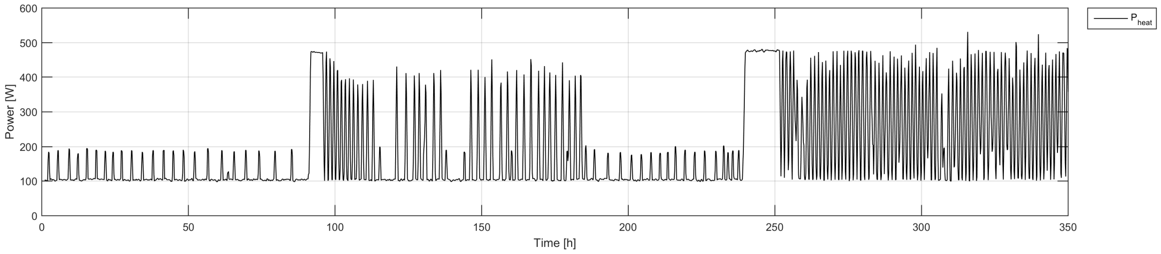
Description

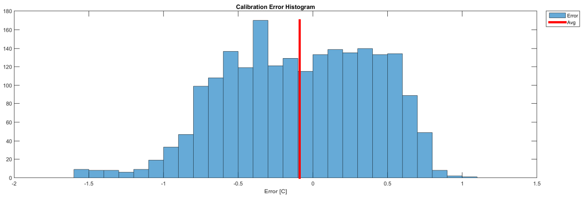
Same as case 11, but using model R7C3

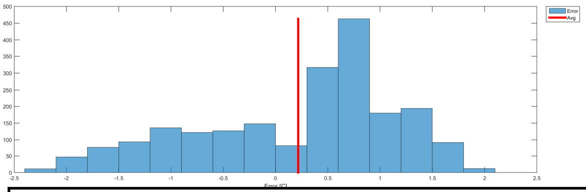
Case 13

Calibration



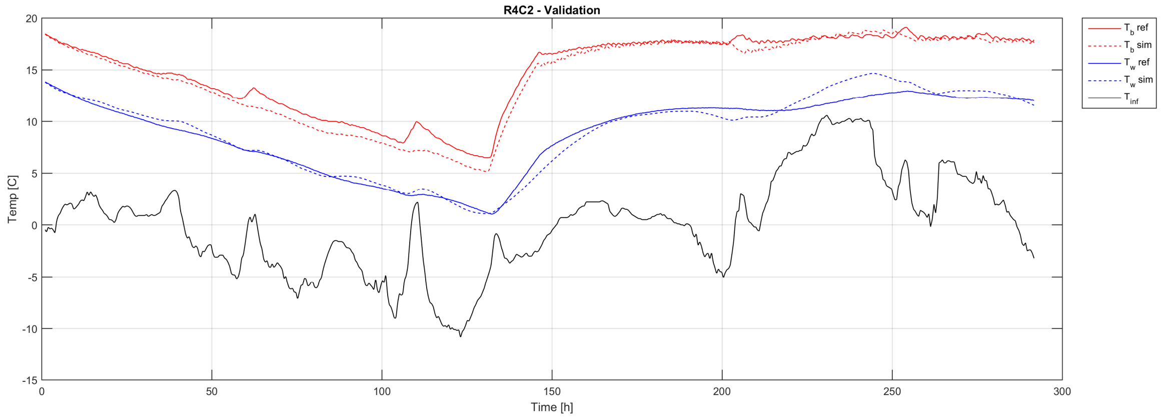


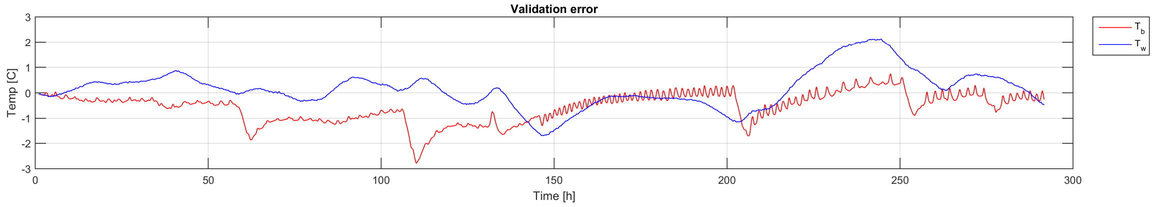


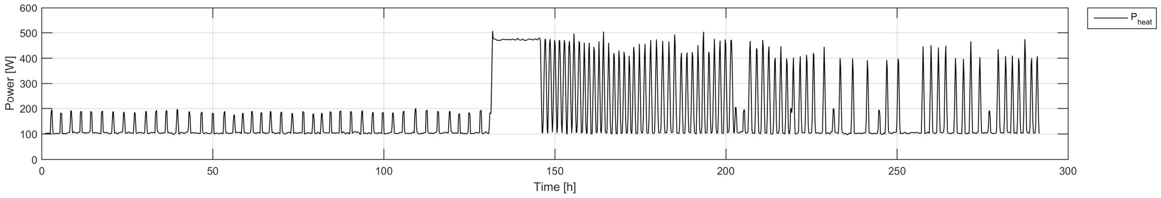


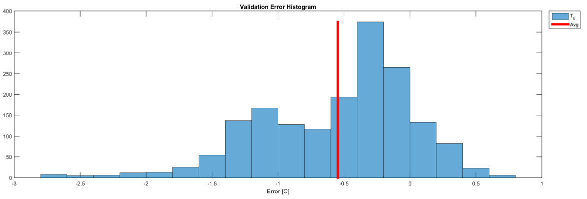
```
Simulation statistics
Total time     : 1.0 min
Timestep       : 600 sec
R_b    =                    0.054775              (Initial = 0.040000)
R_w    =                    0.089732              (Initial = 0.100000)
R_g    =                    0.184655              (Initial = 0.300000)
R_vent =                    0.330000              (Initial = 0.200000)
C_b    =                    1232103.711683        (Initial = 1110000.000000)
C_w    =                    1231242.811396        (Initial = 1400000.000000)
RMSEC: 1.101
Model type: R4C2
rmsep Tb: 0.795
rmsep Tw: 0.744
```

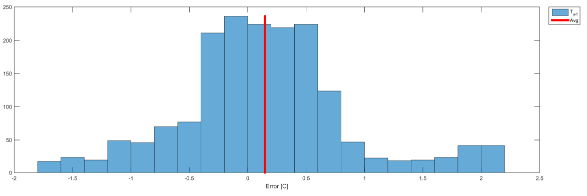
Validation









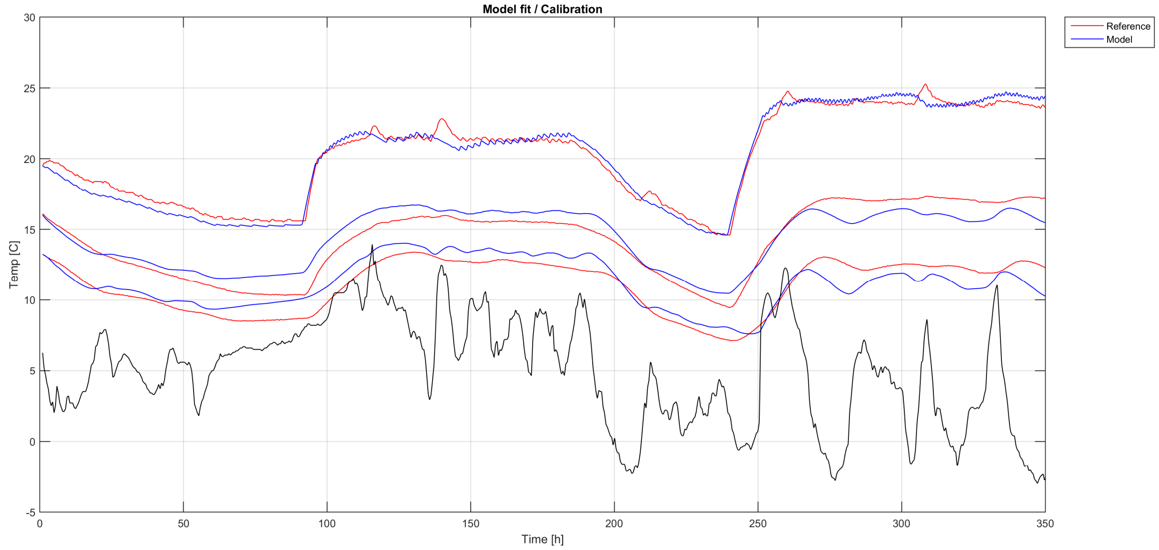


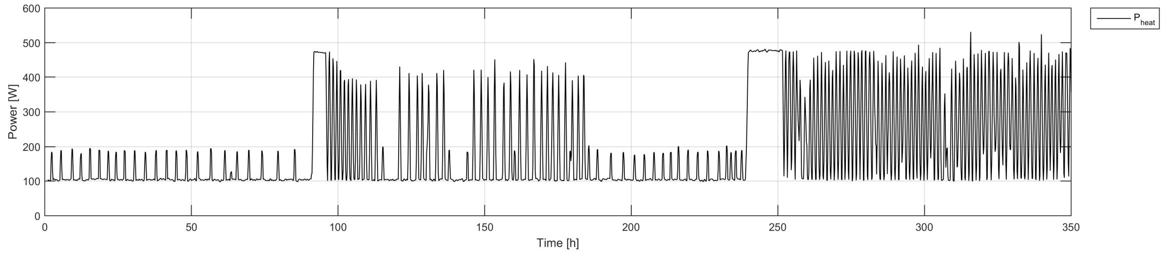
Description

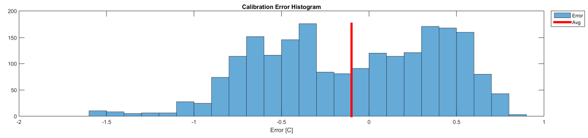
Case 13 is similar to case 5, but with more data. Here the rest of November is included, thus extending the data set from 02.11.2015 17:18 to 30.11.2015 03:28, resampled with 10 min interval for a total of 3950 samples. The data set is then nearly twice the size as in case 5. Samples 2100 - 2200 were classified as outliers due to rapid dynamics assumed caused by venting of the building and where excluded from the modeling process. The data is split such that 1 - 2100 forms the calibration. Note that this is the entire data set from case 5-8.2201-3950 forms the validation data. Note also that the outdoor temperature range in the last half (validation) of the data set has a range of -10 to 10C (calibration, as shown for case 5 has a range of 0 - 10C). There is a mismatch in input ranges for this case. Its also interesting to note large prediction errors during large fluctuations of outdoor temperature(hour 65 and 110).

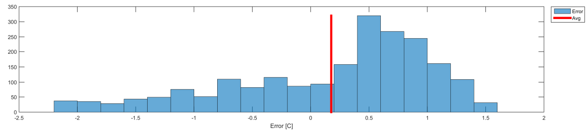
Case 14

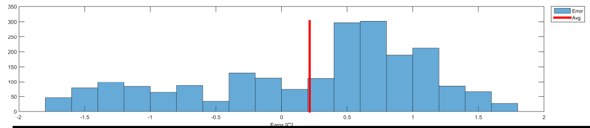
Calibration





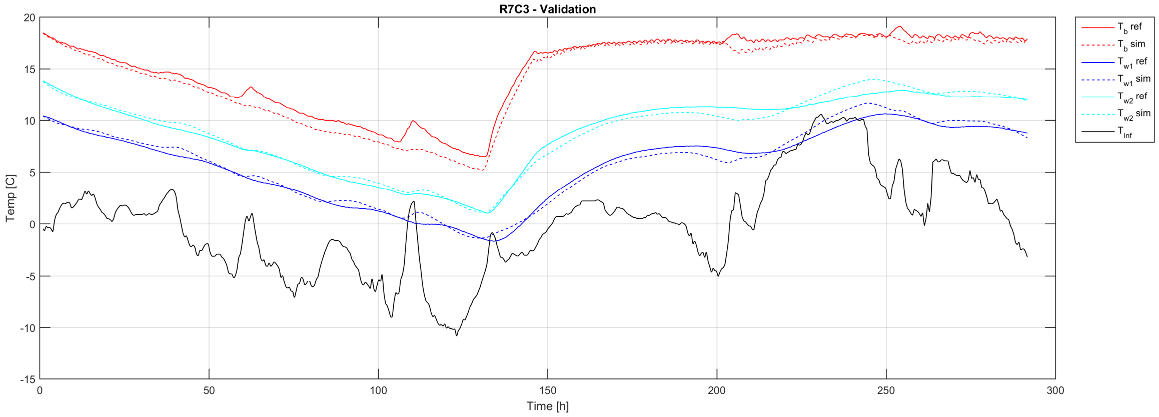


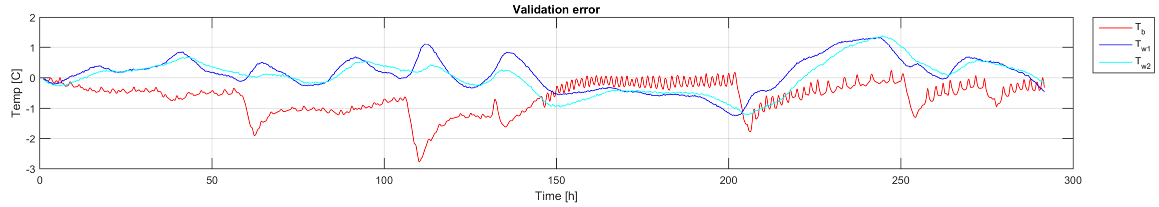


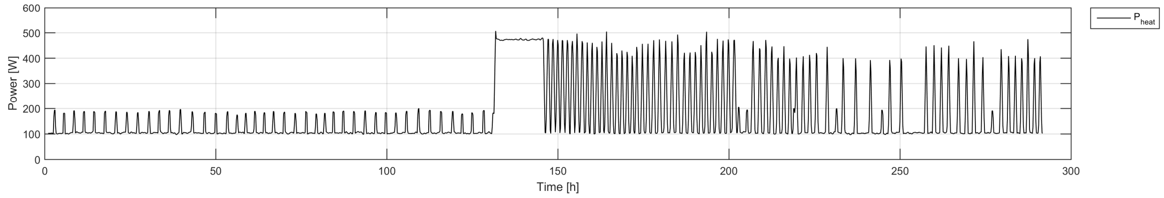


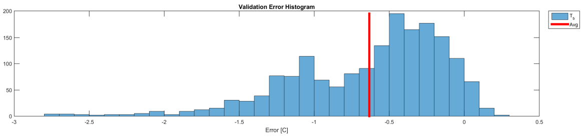
```
Simulation statistics
Total time     : 2.0 min
Timestep       : 600 sec
R_b     =                0.055127            (Initial = 0.060000)
R_w1    =                0.034297            (Initial = 0.080000)
R_w2    =                0.032163            (Initial = 0.020000)
R_s     =                0.001635            (Initial = 0.001000)
R_e     =                0.025033            (Initial = 0.050000)
R_g     =                0.176442            (Initial = 0.135000)
R_vent =                 0.330044            (Initial = 0.200000)
C_b     =                1246052.641217      (Initial = 1310000.000000)
C_w1    =                1605478.615390      (Initial = 1000000.000000)
C_w2    =                293211.937609       (Initial = 500000.000000)
RMSEC: 1.362
Model type: R7C3
rmsep Tb: 0.811
rmsep Tw1: 0.558
rmsep Tw2: 0.543
```

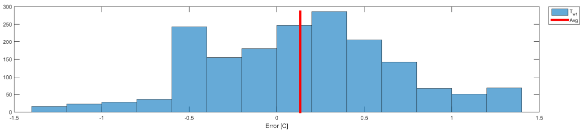
Validation

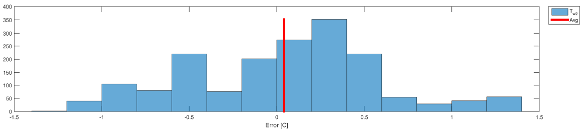










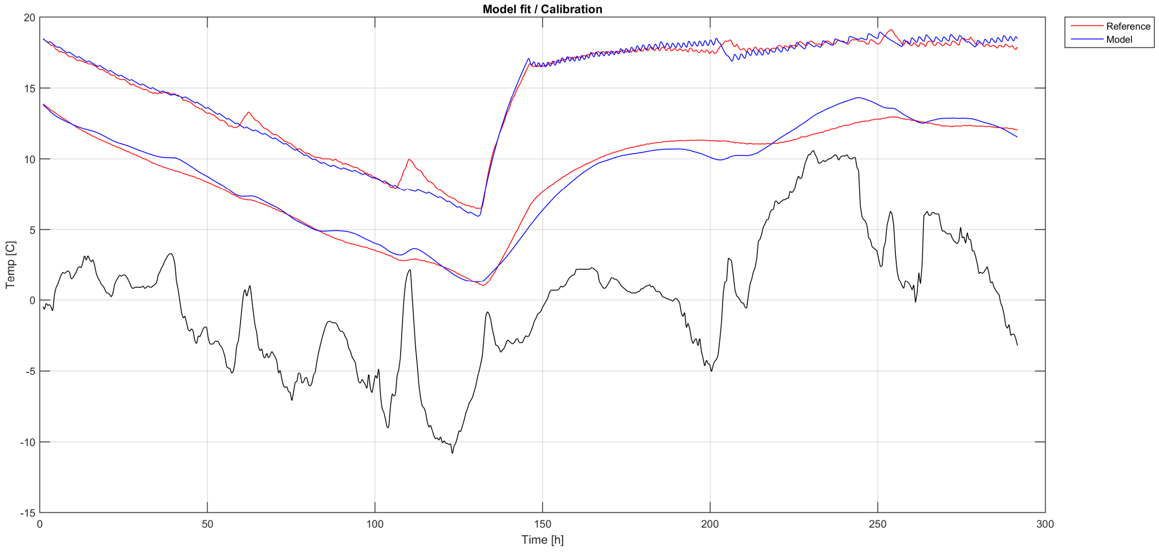


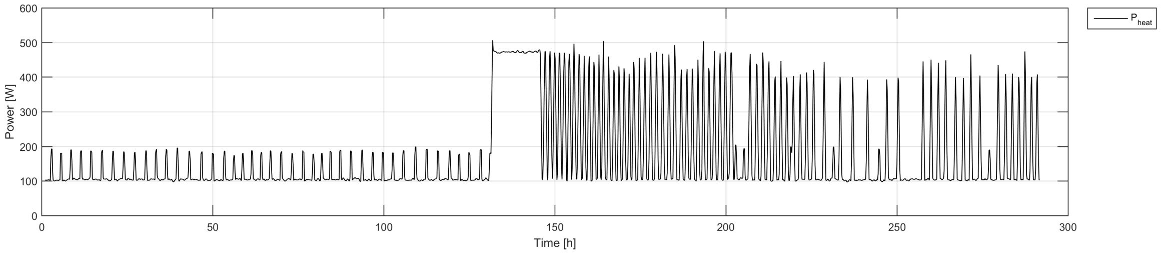
Description

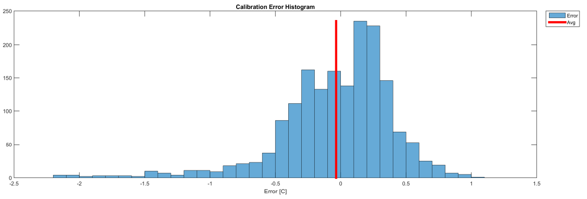
Same as case 13, but using model R7C3

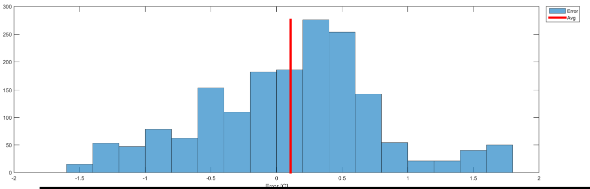
Case 15

Calibration



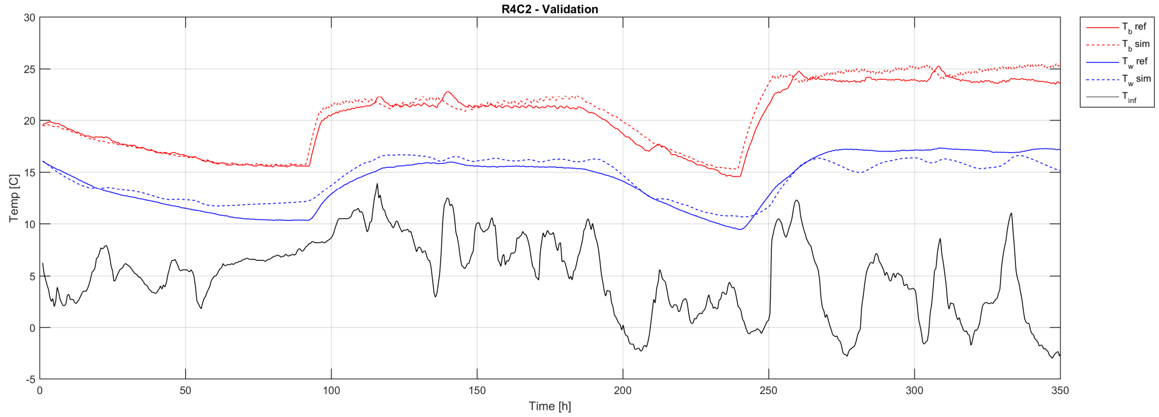


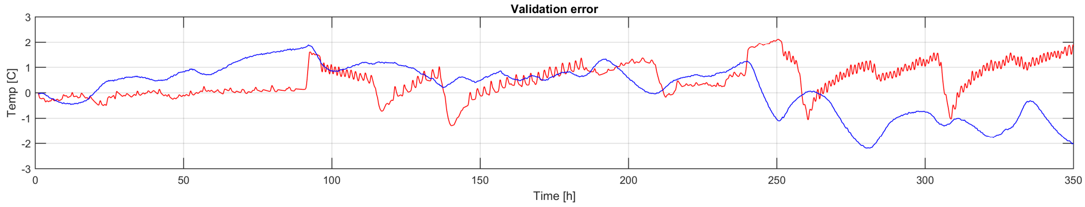


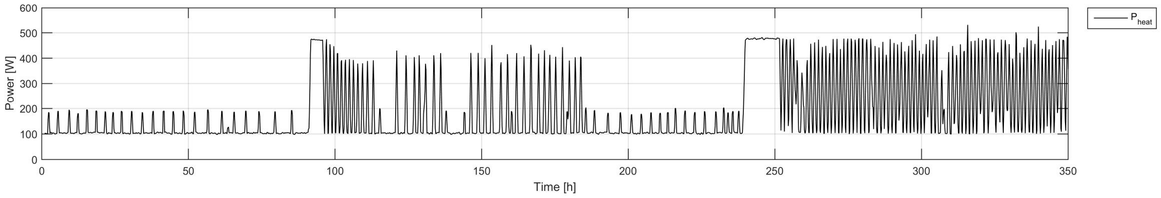


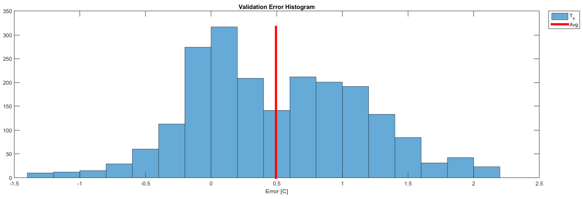
```
Simulation statistics
Total time      : 0.9 min
Timestep        : 600 sec

R_b    =                  0.037364                (Initial = 0.040000)
R_w    =                  0.055217                (Initial = 0.100000)
R_g    =                  0.899999                (Initial = 0.300000)
R_vent =                  0.330007                (Initial = 0.200000)
C_b    =                  957331.551375           (Initial = 1110000.000000)
C_w    =                  2012721.530961          (Initial = 1400000.000000)

RMSEC: 0.815
Model type: R4C2
rmsep Tb: 0.816
rmsep Tw: 0.992
```

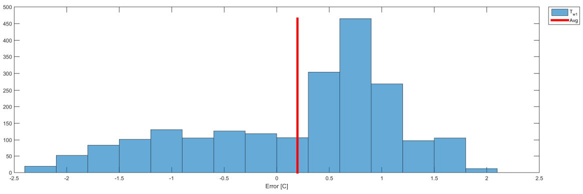
Validation









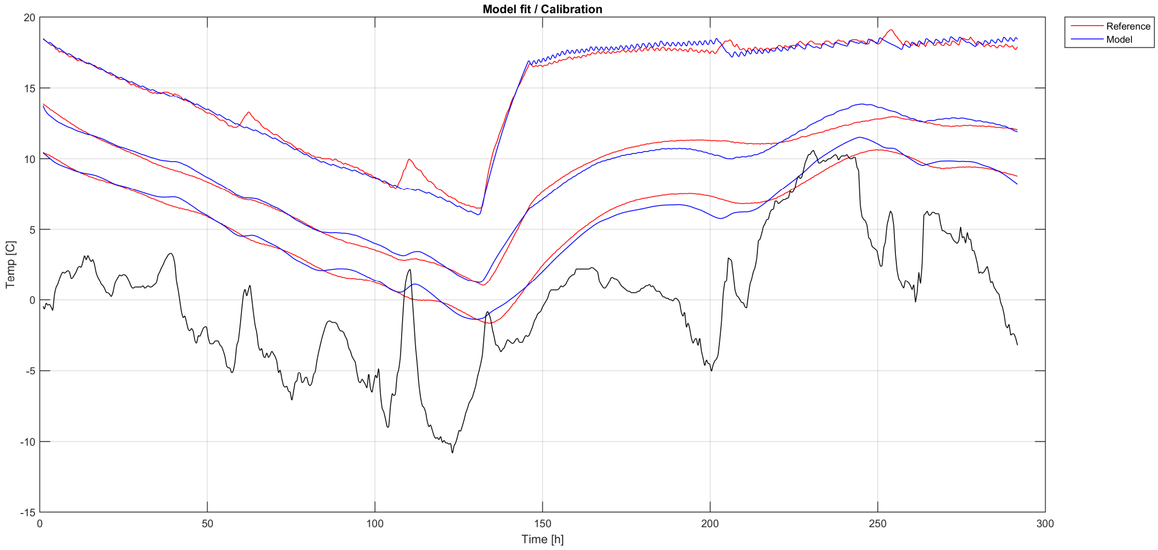


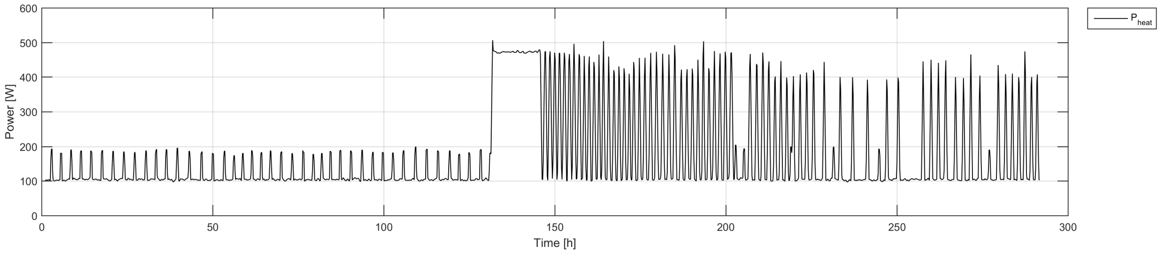
Description

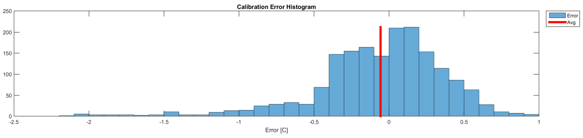
Same as case 13 but with calibration and validation ranges swapped.

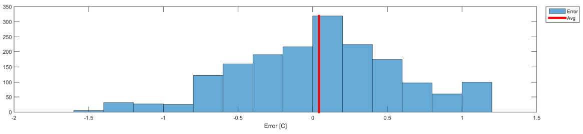
Case 16

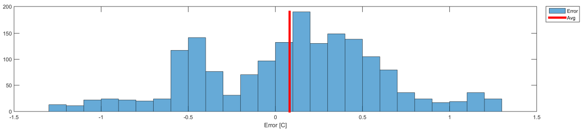
Calibration





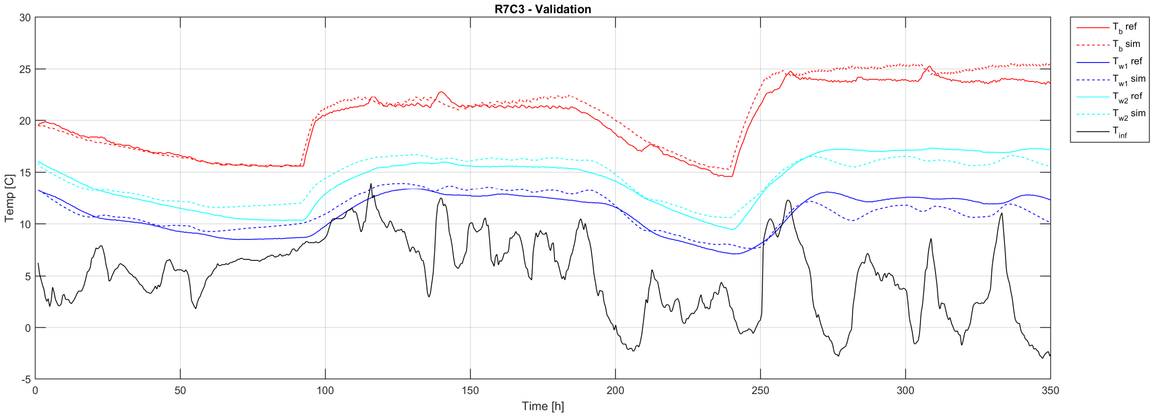


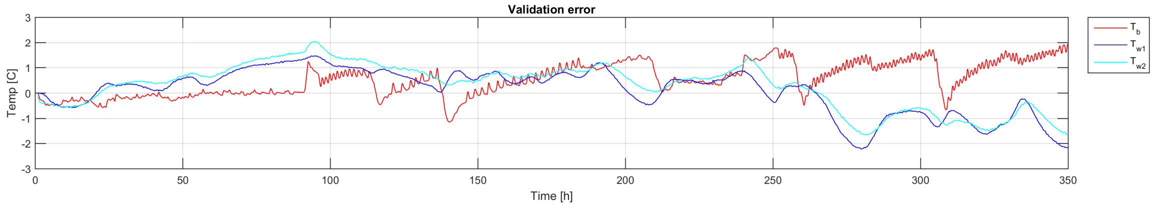


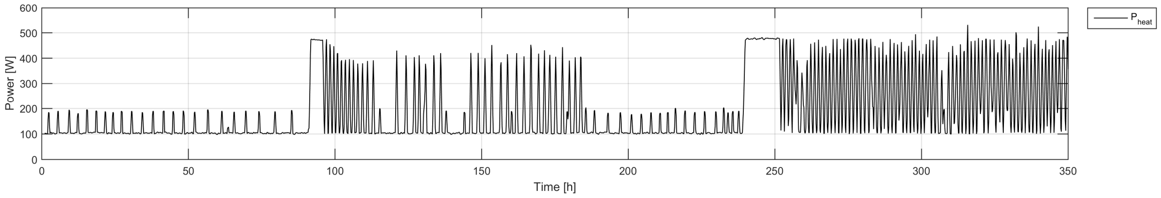


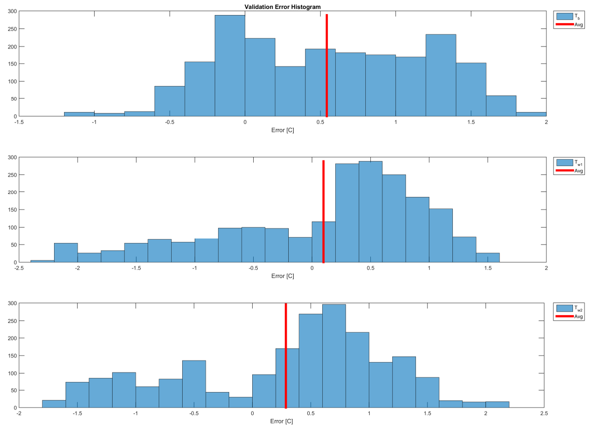
```
Simulation statistics
Total time      : 1.7 min
Timestep        : 600 sec
R_b     =                0.041221            (Initial = 0.060000)
R_w1    =                0.024549            (Initial = 0.080000)
R_w2    =                0.023117            (Initial = 0.020000)
R_s     =                0.001622            (Initial = 0.001000)
R_e     =                0.015548            (Initial = 0.050000)
R_g     =                0.405000            (Initial = 0.135000)
R_vent  =                0.330196            (Initial = 0.200000)
C_b     =                1173163.947254      (Initial = 1310000.000000)
C_w1    =                2435485.178205      (Initial = 1000000.000000)
C_w2    =                168343.332890       (Initial = 500000.000000)
RMSEC: 0.869
Model type: R7C3
rmsep Tb: 0.847
rmsep Tw1: 0.881
rmsep Tw2: 0.928
```

Validation







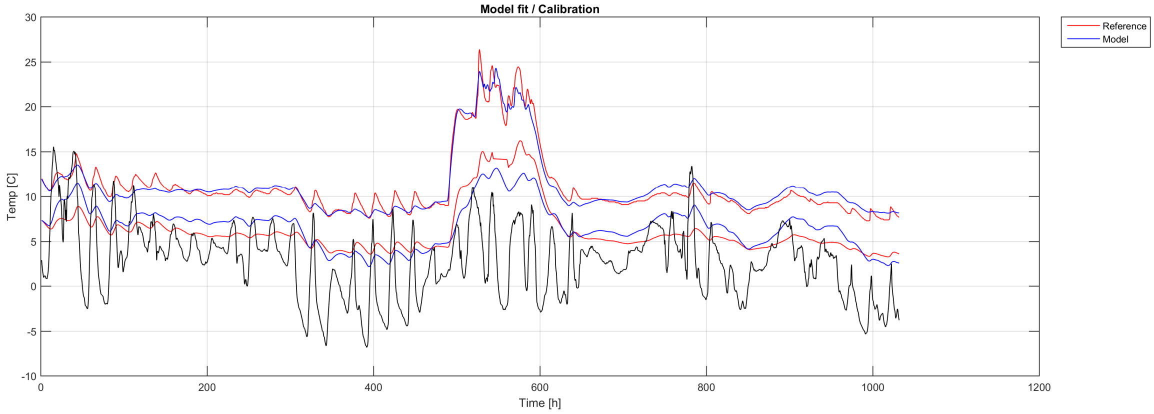


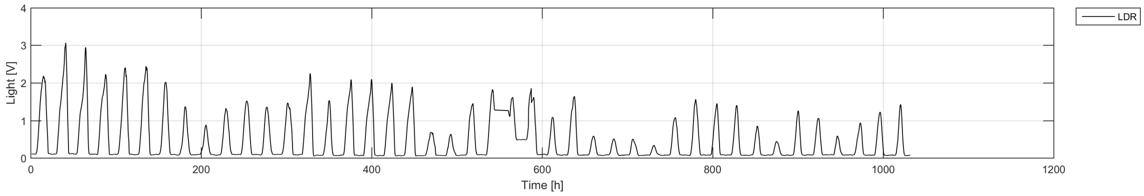
Description

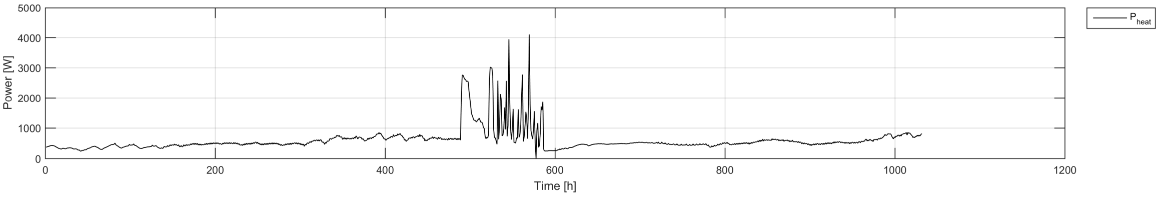
Same as case 14, but with validation and calibration ranges swapped.

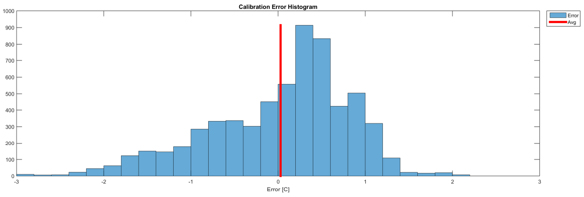
Case 17

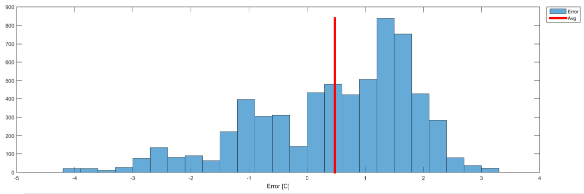
Calibration





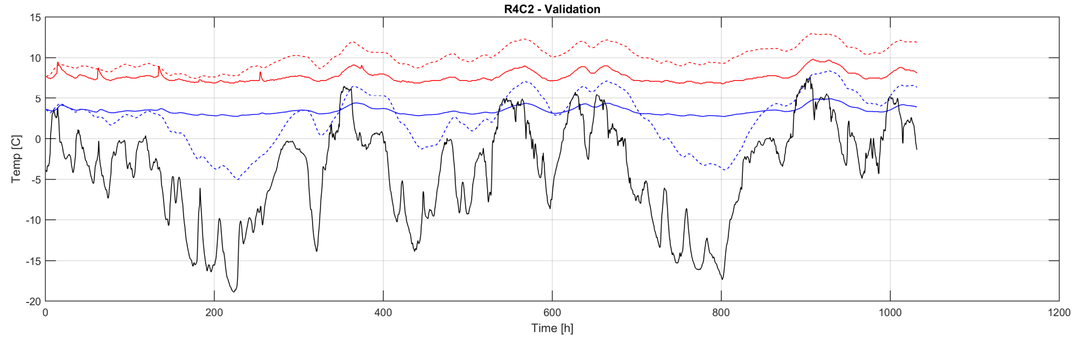


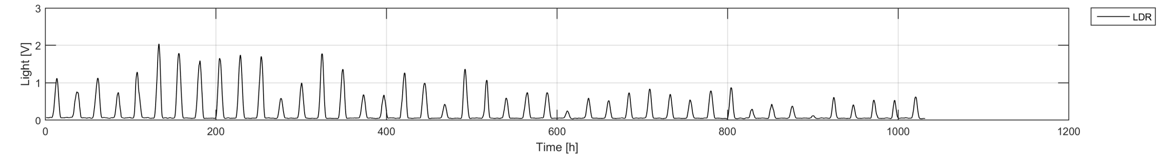


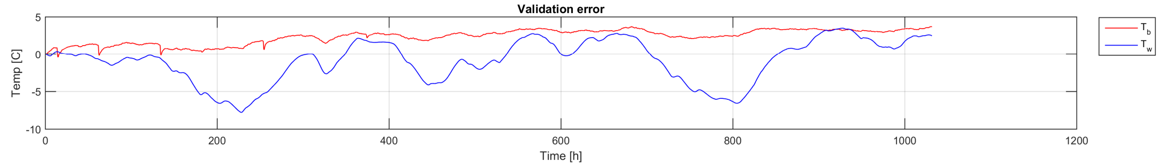


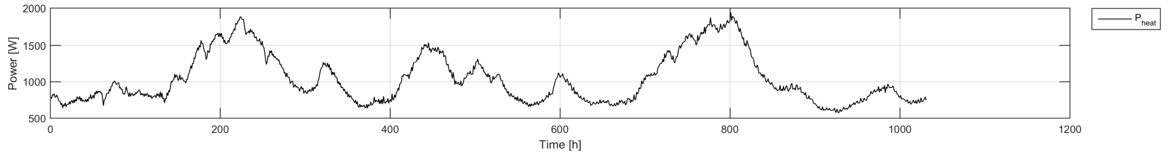
```
Simulation statistics
Total time     : 0.9 min
Timestep       : 600 sec

R_b    =                 0.018120               (Initial = 0.018000)
R_w    =                 0.015629               (Initial = 0.015000)
R_g    =                 0.022717               (Initial = 0.022000)
R_vent =                 0.330002               (Initial = 0.200000)
C_b    =                 6099999.999947         (Initial = 6100000.000000)
C_w    =                 6999999.999960         (Initial = 7000000.000000)

RMSEC: 1.662
Model type: R4C2
rmsep Tb: 2.557
rmsep Tw: 2.981
```

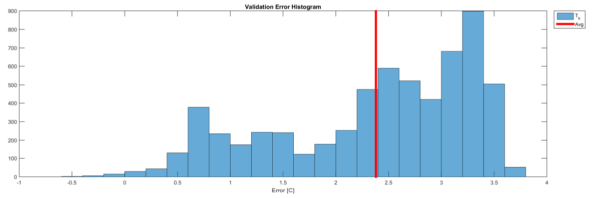
Validation

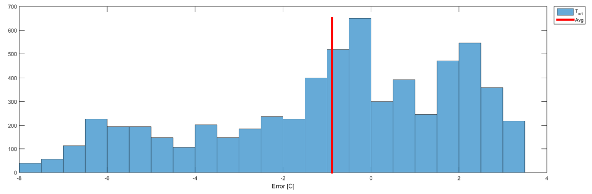












Description

Case 17 is the only case based on “Cabin” data that is somewhat successful in identifying parameters for a model. Only R4C2 is tested, since there are no temperature readings from inside the insulation layer of the walls (required for R7C3). This case has also the longest time range, from 01.10.2015 01:00 to 25.12.2015 23:50, close to three months worth of data. This data set is also the only one with measurements of light. For most of the data, the building is maintained at setpoint of 7°C by the heaters, except for a short periode around hours 500 to 600, where there is a step response up to 20° - 23°C. The lack of dynamic variation in the indoor temperature Tb is a noteworthy problem with this particular data (also the reason case 18 failes).

Appendix C - Code Listing

This appendix lists the source code for the most important classes in c# and functions in MATLAB. These form the business layer of the software in this project, responsible for all the computations and simulations required to achieve the results. Most of the classes and functions described in chapter 5 are listed below, but some minor discrepancies between model and code in naming may occur. In the software models of chapter 5 only the most important methods and attributes are included in the diagrams, for simplicity.

Log file data structures (LogFile Converter, c#)

```
class LogValue {
        public string Name{get;set;}
        public LogValue(string name) {
                Name = name;
        }
        /// <summary>
        /// Get LogValue as double
        /// </summary>
        /// <returns></returns>
        public virtual double GetDouble() {
                return 0.0;
        }
        /// <summary>
        /// Get log value as string
        /// </summary>
        /// <returns></returns>
        public virtual string GetString() {
                return "";
        }

        public override string ToString() {
                return Name + " = " + GetString();
        }
}

class LogValueDouble:LogValue {
        private double _fVal;
        string _sFormat = "";
        public LogValueDouble(string name, double fVal, string sFormat = "")
                : base(name) {
                _sFormat = sFormat;
                _fVal = fVal;
        }
        public LogValueDouble(string name, string sVal, string sFormat = "")
                : base(name) {
                _sFormat = sFormat;
                if(!double.TryParse(sVal, out _fVal))
                        _fVal = double.NaN;
        }
        public override double GetDouble() {
                return _fVal;
        }
        public override string GetString() {
                return _fVal.ToString(_sFormat);
        }
}
```

```
class LogValueString :LogValue{
    private string _sVal;
    public LogValueString(string name, string sVal)
        : base(name) {
        _sVal = sVal;
    }
    public LogValueString(string name, double fVal, string sFormat = "0.0")
        : base(name) {
        _sVal = fVal.ToString(sFormat); ;
    }
    /// <summary>
    /// return value as double
    /// </summary>
    /// <returns></returns>
    public override double GetDouble(){
        double fVal;
        try {
            if(double.TryParse(_sVal, out fVal))
                return fVal;
            else
                return double.NaN;
        }
        catch(Exception ex){
            return double.NaN;
        }
    }

    //return value as string
    public override string GetString() {
        return _sVal;
    }
}
```

```
class LogLine {
        public DateTime TimeStamp { get; set; }
        private LogValue[] _values;

        public int Count {
                get {
                        if(_values == null)             return 0;
                        else                                    return _values.Length;
                }
        }

        public LogLine(int nValues) {
                _values = new LogValue[nValues];
        }

        /// <summary>
        /// Check if the LogLine contains value of a given name
        /// </summary>
        /// <param name="sName"></param>
        /// <returns></returns>
        public bool ContainsName(string sName) {
                if(Count == 0)
                        return false;
                else {
                        for(int i = 0; i < _values.Length; i++)
                                if(_values[i].Name == sName)
                                        return true;
                }
                return false;
        }

        public LogValue this[string sName] {
                get {
                        int index = IndexOf(sName);
                        if(index == - 1)        return null;
                        else                            return _Get(index); }
                set {
                        int index = IndexOf(sName);
                        if(index != -1)
                                _Set(value, index);
                }
        }

        public LogValue this[int index] {
                get { return _Get(index); }
                set { _Set(value, index); }
        }
        public int IndexOf(string sName) {
```

```
        if(_values == null)
            throw new Exception("No data to search");

        for(int i = 0; i < _values.Length; i++) {
            if(_values[i].Name == sName)
                return i;
        }
        return -1;
    }
    private void _Set(LogValue val, int index) {
        if(_values != null && index < _values.Length && index >= 0)
            _values[index] = val;
        else
            throw new Exception("index out of range");
    }

    private LogValue _Get(int index) {
        if(_values != null && index < _values.Length && index >= 0)
            return _values[index];
        else
            throw new Exception("index out of range");
    }

    public override string ToString() {
        if(_values != null) {
            string[] sVals = new string[_values.Length];
            for(int i = 0; i < _values.Length; i++) {
                sVals[i] = (this[i] != null ? this[i].ToString() : "");
            }
            return TimeStamp.ToString() + ": " + String.Join(" | ",
                                                       sVals);
        }
        else
            return "";
    }
}
```

```
class LogFile {
    protected List<LogLine> _lstLogLines;

    public int Count {
        get {
            if(_lstLogLines == null)    return 0;
            else                        return _lstLogLines.Count;
        }
    }

    #region constructor
    public LogFile() {
        _lstLogLines = new List<LogLine>();
    }
    #endregion

    #region file operations

    /// <summary>
    /// Return a dictionary index over all value names found in the LogFile
    /// </summary>
    /// <returns></returns>
    public Dictionary<string, int> CreateValueNameIndex() {
        if(_lstLogLines == null || _lstLogLines.Count == 0)
            throw new Exception("No LogLines in this file");

        Dictionary<string,int> dct = new Dictionary<string,int>();

        for(int j = 0; j < _lstLogLines.Count; j++){
            LogLine line = _lstLogLines[j];

            for(int i = 0; i < line.Count; i++) {
                if(line[i] != null && !dct.ContainsKey(line[i].Name))
                    dct.Add(line[i].Name,dct.Count);
            }
        }
        return dct;
    }

    /// <summary>
    /// Add all LogLines in another file to this one
    /// </summary>
    /// <param name="file"></param>
    public void AddFile(LogFile file) {
        for(int i = 0; i < file.Count; i++) {
            AddLine(file[i]);
        }
    }
```

```
        /// <summary>
        /// Add line to a file
        /// </summary>
        /// <param name="line"></param>
        public void AddLine(LogLine line) {
                _lstLogLines.Add(line);
        }
        #endregion

        #region Get columns
        public void GetStartStopTime(out DateTime dtStart, out DateTime dtStop) {
                dtStart = DateTime.MaxValue;
                dtStop = DateTime.MinValue;

                for(int i = 0; i < this.Count; i++) {
                        if(this[i].TimeStamp > dtStop)
                                dtStop = this[i].TimeStamp;

                        if(this[i].TimeStamp < dtStart)
                                dtStart = this[i].TimeStamp;
                }
        }

        /// <summary>
        /// Return an array of values, one from each LogLine, with the name sName
        /// </summary>
        /// <param name="sName"></param>
        /// <returns></returns>
        public double[] GetColumn(string sName) {
                double[] fClm = new double[_lstLogLines.Count];
                for(int k = 0; k < _lstLogLines.Count; k++) {
                        LogLine line = _lstLogLines[k];

                        for(int i = 0; i < line.Count; i++) {
                                if(line[i] != null && line[i].Name == sName)
                                        fClm[k] = line[i].GetDouble();
                        }
                }
                return fClm;
        }

        /// <summary>
        /// Return all values in column index, based on the CreateValueNameIndex list of
columns and indecies
        /// </summary>
        /// <param name="index"></param>
        /// <returns></returns>
        public double[] GetColumn(int index) {
                Dictionary<string, int> dct = CreateValueNameIndex();
        if(index >= dct.Count)
            throw new Exception("index out of range");
        if(_lstLogLines == null || _lstLogLines.Count == 0)
            throw new Exception("No data in file");
        string sName = dct.FirstOrDefault(x => x.Value == index).Key;
        return GetColumn(sName);
    }

    public enum TimeUnit { Sec,Min,Hour};
    /// <summary>
    /// Get the timestamps of file relative to some starting point. Configurable time
unit Sec,Min,Hour
    /// </summary>
    /// <param name="dtStart">Start time</param>
    /// <param name="tm">Time unit</param>
    /// <returns></returns>
    public double[] GetTime(DateTime dtStart, TimeUnit tm) {
        double[] fTime = new double[_lstLogLines.Count];
        for(int k = 0; k < _lstLogLines.Count; k++) {
            LogLine line = _lstLogLines[k];
            switch(tm) {
                case TimeUnit.Sec:
                    fTime[k] = (line.TimeStamp -
                            dtStart).TotalSeconds;
                    break;
                case TimeUnit.Min:
                    fTime[k] = (line.TimeStamp -
                            dtStart).TotalMinutes;
                    break;
                case TimeUnit.Hour:
                    fTime[k] = (line.TimeStamp -
                            dtStart).TotalHours;
                    break;
            }
        }
        return fTime;
    }
    #endregion
    #region operators
    public LogLine this[int index] {
        get {
            if(_lstLogLines == null || index >= _lstLogLines.Count)
                throw new Exception("index out of range or no lines
                                    found");
            return _lstLogLines[index];
        }
    }
    #endregion
}
```

Data Operations (LogFile Converter, c#)

```
class DataOperationCtrl {
        private Thread _thrOperation;
        private bool _bRunSim = true;
        private List<DataOperation> _lstQue;
        public delegate void OperationFailed(string sError);
        public event OperationFailed OnOperationFailed;
        DateTime _dtOpStart;

        public double CurProgress {
                get {
                        if(_lstQue != null && _lstQue.Count > 0)
                                return _lstQue[0].OpProgress;
                        else
                                return 100;
                }
        }

        public TimeSpan TimeRemaining {
                get {
                        TimeSpan ts = DateTime.Now - _dtOpStart;
                        if(CurProgress > 0) {
                                double fPercentRemaining = 100 - CurProgress;
                                double fSecUsed = (double)ts.TotalSeconds;
                                double fSecPrPercent = fSecUsed / CurProgress;
                                double fSecRemaining = fSecPrPercent * fPercentRemaining;

                                return new TimeSpan(0, 0, (int)fSecRemaining);
                        }
                        else return new TimeSpan(0);
                }
        }

        public DataOperationCtrl() {
                _lstQue = new List<DataOperation>();
                _thrOperation = new Thread(DoOperation);
        }

        public void Start() {
                _bRunSim = true;
                _thrOperation.Start();
        }
        public void Stop() {
                _bRunSim = false;
                _thrOperation.Abort();
        }
```

```
    public void Que(DataOperation op) {
        _lstQue.Add(op);
    }
    //Worker thread proc
    private void DoOperation() {
        while(_bRunSim) {
            if(_lstQue.Count > 0) {
                try {
                    _dtOpStart = DateTime.Now;
                    _lstQue[0].Do();
                    _lstQue[0].Finish();

                }
                catch(Exception ex) {
                    string sType = ex.GetType().ToString();
                    if(ex.GetType().ToString() !=
"System.Threading.ThreadAbortException") {
                        if(OnOperationFailed != null)
                            OnOperationFailed(ex.Message);
                    }
                }
                finally {
                    _lstQue.RemoveAt(0);
                }
            }
            else
                Thread.Sleep(100);
        }
    }
}
```

```
class DataOperation {
    public delegate void OperationComplete(DataOperation op);
    public event OperationComplete OnOperationComplete;
    public LogFile Input { get; set; }
    public LogFile Result { get; set; }
    public double OpProgress { get; set; }

    public DataOperation() {
        Input = new LogFile();
        Result = new LogFile();
    }
    public virtual void Do() {
    }

    public void Finish() {
        if(OnOperationComplete != null)
            OnOperationComplete(this);
    }
}

class LoadFileOperation:DataOperation {
    protected string[] sLines = null;
    string _sFileName;
    FieldConverter _converter;
    bool _bHasHeader = false;
    char _cFieldSeparator;
    public LoadFileOperation(string sFileName,char cFieldSeparator,bool bHasHeader,
FieldConverter converter) {
        _sFileName = sFileName;
        _converter = converter;
        _bHasHeader = bHasHeader;
        _cFieldSeparator = cFieldSeparator;
    }

    public override void Do() {
        base.Do();
        OpProgress = 0;
        if(Input != null)
            Result = Input;
        else if(Result == null)
            throw new Exception("No Results LogFile object found (cannot be
null)");

        if(File.Exists(_sFileName))
            sLines = File.ReadAllLines(_sFileName);
        else
            throw new Exception("Failed to load file");

        OpProgress = 10;
        //remove line one, if header
        string sHeader = null;
        if(_bHasHeader) {
            sHeader = sLines[0];
            string[] sa = new string[sLines.Length - 1];
            Array.Copy(sLines, 1, sa, 0, sa.Length);
            sLines = sa;
        }

        for(int i = 0; i < sLines.Length; i++) {
            LogLine line =
_converter.ConvertFields(sLines[i].Split(_cFieldSeparator),
sHeader.Split(_cFieldSeparator));
            Result.AddLine(line);
            OpProgress = 10 + 90.0 * (double)i / (double)sLines.Length;
        }
        OpProgress = 100;
    }
}

class ExtractOperation :DataOperation{
    string[] _sClmNames;
    DateTime _dtStart;
    DateTime _dtStop;

    public ExtractOperation(string[] sClmNames, DateTime dtStart, DateTime dtStop) {
        _sClmNames = sClmNames;
        _dtStart = dtStart;
        _dtStop = dtStop;
    }
    public override void Do() {
        base.Do();

        //make a look table
        Dictionary<string, int> dct = new Dictionary<string, int>();
        for(int i = 0; i < _sClmNames.Length; i++)
            dct.Add(_sClmNames[i], i);

        //scan this file for any fields matching the list, using the indecies from
list to rearange columns
        for(int nLine = 0; nLine < Input.Count; nLine++) {
            LogLine line = Input[nLine];
            if(line.TimeStamp >= _dtStart && line.TimeStamp <= _dtStop) {
                LogLine lineOut = new LogLine(dct.Count);
                lineOut.TimeStamp = line.TimeStamp;

                for(int nField = 0; nField < line.Count; nField++) {
                    string sName = line[nField].Name;
                    if(dct.ContainsKey(sName)) {
                        lineOut[dct[sName]] = new
LogValueString(line[sName].Name, line[sName].GetString());
```

```
                    }
                }

                //add line if timestamp within bounds
                Result.AddLine(lineOut);
            }
        }
    }
}

class FilterOperation:DataOperation {

    double[] _weights;
    bool _bCentered;

    public FilterOperation(double[] weights, bool bCentered) {
        _weights = weights;
        _bCentered = bCentered;
    }
    public override void Do() {
        base.Do();

        //make a list of all variable names in the file
        Dictionary<string, int> dct = Input.CreateValueNameIndex();
        int nColumns = dct.Count;

        double[][] fVal = new double[nColumns][];
        for(int nC = 0; nC < nColumns; nC++) {
            fVal[nC] = new double[Input.Count];
            for(int nSamp = 0; nSamp < Input.Count; nSamp++) {
                fVal[nC][nSamp] = Input[nSamp][nC].GetDouble();
            }
        }

        double[][] fFilterd = FilterWMA(fVal);
        for(int nSamp = 0; nSamp < Input.Count; nSamp++) {
            LogLine line = new LogLine(nColumns);
            line.TimeStamp = Input[nSamp].TimeStamp;
            for(int nC = 0; nC < nColumns; nC++) {
                line[nC] = new
LogValueDouble(Input[nSamp][nC].Name,fFilterd[nC][nSamp]);
            }
            Result.AddLine(line);
        }
    }

    private double[][] FilterWMA(double[][] fX) {
        double fSumXW = 0.0, fSumW = 0.0;
        int N,i,j;
        int nC = fX.Length;
        int nR = fX[0].Length;

        double[][] fY = new double[nC][];
        for(int c = 0; c < nC; c++) {
            N = Math.Min(fX[c].Length, _weights.Length);
            fY[c] = new double[nR];
            for(int r = 0; r < nR; r++) {
                fSumW = 0.0;
                fSumXW = 0.0;

                for(i = 0; i < N; i++) {
                    //compute x index, offset by N/2 if ArrayOffset mode,
centers window on cur output value => optimal noise reduction
                    j = i + r - (_bCentered ? (N / 2) : 0);

                    if(j < fX[c].Length && j >= 0) {
                        fSumW += _weights[i];
                        fSumXW += _weights[i] * fX[c][j];
                    }
                }
                if(fSumW == 0.0) fY[c][r] = 0.0;
                else fY[c][r] = fSumXW / fSumW;
            }
        }
        return fY;
    }
}

class RemoveOutliersOperation:DataOperation {
    double _fOutlierLimit;
    int _N;

    public RemoveOutliersOperation(double fOutlierLimit, int N) {
        _N = N;
        _fOutlierLimit = fOutlierLimit;
    }

    public override void Do() {
        base.Do();

        //make a list of all variable names in the file
        Dictionary<string, int> dct = Input.CreateValueNameIndex();
        int nColumns = dct.Count;

        double[][] fVal = new double[nColumns][];
        for(int nC = 0; nC < nColumns; nC++) {
            fVal[nC] = new double[Input.Count];
            for(int nSamp = 0; nSamp < Input.Count; nSamp++) {
                fVal[nC][nSamp] = Input[nSamp][nC].GetDouble();
            }
        }

        double[][] fFilterd = RemoveOutliers(fVal);
        for(int nSamp = 0; nSamp < Input.Count; nSamp++) {
            LogLine line = new LogLine(nColumns);
            line.TimeStamp = Input[nSamp].TimeStamp;
            for(int nC = 0; nC < nColumns; nC++) {
                line[nC] = new LogValueDouble(Input[nSamp][nC].Name,
fFilterd[nC][nSamp]);
            }
            Result.AddLine(line);
        }
    }
    private double[][] RemoveOutliers(double[][] fX) {
        int nC = fX.Length;
        int nR = fX[0].Length;
        int i, j;
        double fSum, fN, fAvg;

        double[][] fY = new double[nC][];
        for(int c = 0; c < nC; c++) {
            fY[c] = new double[nR];
            for(int r = 0; r < nR; r++) {
                fN = 0;
                fSum = 0;
                for(i = 0; i < _N; i++) {
                    //compute x index, offset by N/2
                    j = i + r - (_N / 2);

                    if(j < fX[c].Length && j >= 0) {
                        fN += 1.0;
                        fSum +=  fX[c][j];
                    }
                }

                //compute the average over N samples (centered)
                fAvg = fSum / fN;

                if(Math.Abs(fX[c][r]-fAvg) > _fOutlierLimit)
                    fY[c][r] = fAvg;
                else
                    fY[c][r] = fX[c][r];

            }
        }

        return fY;
    }
}
```

```
class ResampleOperation :DataOperation{
        DateTime _dtStart;
        TimeSpan _tsStep;
        int _N;
        public ResampleOperation(DateTime dtStart, TimeSpan tsStep, int N) {
                _dtStart = dtStart;
                _tsStep = tsStep;
                _N = N;
        }

        public override void Do() {
                base.Do();
                if(Input == null || Input.Count == 0)
                        throw new Exception("No data in file");

                //make a list of all variable names in the file
                Dictionary<string, int> dct = Input.CreateValueNameIndex();
                int nColumns = dct.Count;

                //loop through all the times in the new file
                DateTime dtCur = _dtStart;
                for(int i = 0; i < _N; i++) {
                        OpProgress = 100.0*(double)i/(double)_N;

                        //create a new line and add it to the resampled file
                        LogLine line = new LogLine(nColumns);

                        //set the timestamp
                        line.TimeStamp = dtCur;

                        //loop over all variable/column names in the file, and interpolate
values for each of them. Find closest value before and after the current dt
                        for(int nc = 0; nc < nColumns; nc++) {
                                //get current column name
                                string sName = dct.Keys.ToList()[nc];

                                //find line directly before and after which contains a
variable of name sName
                                int nLineBefore, nLineAfter;
                                _FindLineBeforeAfter(dtCur, sName, out nLineBefore, out
nLineAfter);
                                if(nLineBefore == -1 || nLineAfter == -1)
                                        throw new Exception("Failed to find datapoints before
and/or after a specified sample time");

                                //now we have the line we want, get the values
                                double fBefore = Input[nLineBefore][sName].GetDouble();
                                double fAfter = Input[nLineAfter][sName].GetDouble();
                //interpolate
                double fTotalTime = (Input[nLineAfter].TimeStamp - 
Input[nLineBefore].TimeStamp).TotalMilliseconds;
                double fDeltaTime = (dtCur - 
Input[nLineBefore].TimeStamp).TotalMilliseconds;
                double fDeltaVal = fAfter - fBefore;

                double fVal = (fDeltaVal / fTotalTime) * (fDeltaTime) + 
fBefore;

                //set value for this colum
                line[nc] = new LogValueDouble(sName, fVal);
            }
            Result.AddLine(line);
            dtCur = dtCur + _tsStep;
        }

        OpProgress = 100;
    }

    private void _FindLineBeforeAfter(DateTime dtCur, string sValueName, out int 
nLineBefore, out int nLineAfter) {
        //find closest lines in time before and after the current time

        TimeSpan tsBefore = TimeSpan.MinValue;
        TimeSpan tsAfter = TimeSpan.MaxValue;

        nLineBefore = -1;
        nLineAfter = -1;

        for(int nLine = 0; nLine < Input.Count; nLine++) {
            LogLine line = Input[nLine];
            if(line.ContainsName(sValueName)) {
                TimeSpan tsDelta = line.TimeStamp - dtCur;

            //check this line is closer then previously closed line AFTER dt
                if(line.TimeStamp > dtCur && tsDelta < tsAfter) {
                    nLineAfter = nLine;
                    tsAfter = tsDelta;
                }

            //check this line is closer then previously closed line BEFORE dt
                if(line.TimeStamp <= dtCur && tsDelta > tsBefore) {
                    nLineBefore = nLine;
                    tsBefore = tsDelta;
                }
            }
        }
    }
}
```

```
class SaveFileOperation:DataOperation {
        string _sFileName;
        string _sFieldSep;
        bool _bHeader;
        public SaveFileOperation(string sFileName, string sFieldSep, bool bHeader) {
                _sFieldSep = sFieldSep;
                _sFileName = sFileName;
                _bHeader = bHeader;
        }
        public override void Do() {
                base.Do();

                Dictionary<string, int> dct = Input.CreateValueNameIndex();

                //number of columns out
                int nLen = dct.Count;

                //list of lines to write to file
                List<string> lstLines = new List<string>();

                //get header
                if(_bHeader) {
                        List<string> keyList = new List<string>(dct.Keys);
                        lstLines.Add("Timestamp" + _sFieldSep + String.Join(_sFieldSep,
keyList.ToArray()));
                }

                //get all lines
                for(int k = 0; k < Input.Count; k++){
                        LogLine line = Input[k];

                        string[] sFields = new string[nLen + 1];
                        sFields[0] = line.TimeStamp.ToString();
                        for(int i = 0; i < line.Count; i++) {

                                //get the column for this value
                                int clm = dct[line[i].Name];
                                sFields[clm + 1] = line[i].GetString();
                        }
                        lstLines.Add(String.Join(_sFieldSep, sFields));
                }

                File.WriteAllLines(_sFileName, lstLines.ToArray());
                OpProgress = 100;

                Result = Input; //no changes in data
        }
}
```

Format Conversion (LogFile Converter, c#)

```
class FieldConverter {
        /// <summary>
        /// Create the LogLine, throw exception if problems with fields vectors
        /// </summary>
        /// <param name="sFields">string array of fields to convert</param>
        /// <returns></returns>
        protected LogLine MakeLogLine(string[] sFields) {
                if(sFields == null || sFields.Length == 0)
                        throw new Exception("Invalid field vector");

                return new LogLine(sFields.Length);
        }
        /// <summary>
        /// Convert all fields to strings
        /// </summary>
        /// <param name="sFields">string array of fields to convert</param>
        /// <returns></returns>
        public virtual LogLine ConvertFields(string[] sFields, string[] sHeader) {
                LogLine line = MakeLogLine(sFields);

                bool bHeaderOK = true;
                if(sHeader == null || sHeader.Length < sFields.Length)
                        bHeaderOK = false;
                for(int i = 0; i < sFields.Length; i++)
                        line[i] = new LogValueString(bHeaderOK ? sHeader[i] : "Field" +
i.ToString(), sFields[i]);

                return line;
        }
}

class NOSLogSystem : FieldConverter {
        string[] _sTimeHeaders = new
string[7]{"TimeStamp","Day","Month","Year","Hour","Minute","Second"};

        public NOSLogSystem() {
        }
        public override LogLine ConvertFields(string[] sFields,string[] sHeader) {
                int _nChannels;
                _nChannels = (sHeader.Length - 7) / 4;

                LogLine line = new LogLine(_nChannels * 4);

                //handle the timestamp and time information
                int[] nTime = new int[6];
                for(int i = 1; i < 7; i++)
                        nTime[i - 1] = Convert.ToInt32(sFields[i]);
                        line.TimeStamp = new DateTime(nTime[2], nTime[1], nTime[0], nTime[3],
nTime[4], nTime[5]);

                        for(int i = 0; i < _nChannels * 4; i++) {
                                line[i] = new LogValueString(sHeader[7 + i], sFields[7 + i]);
                        }

                        return line;
                }
}


class WeatherStationFormat:FieldConverter {
                string[] _sFieldFormat = new string[19] { "0", "0", "0.0", "0.0", "0.0", "0.0", "",
"0.0", "0.0", "0.0", "0.0", "0.0", "0.0", "0.0", "0.0", "0.0", "0.0", "0", "0" };
                string[] _sFieldNames = new string[19] { "Interval", "Indoor Humidity", "Indoor
Temperature", "Outdoor Humidity", "Outdoor Temperature", "Absolute Pressure)", "Wind",
"Gust", "Direction", "Relative Pressure", "Dewpoint", "Windchill", "Hour Rainfall", "24
hour Rainfall", "Week Rainfall", "Month Rainfall", "Total Rainfall", "Wind Level", "Gust
Level" };
                public override LogLine ConvertFields(string[] sFields,string[] sHeader) {

                        //call this one just to check sFields vector. Ignore the return line
                        MakeLogLine(sFields);
                        LogLine line = new LogLine(19);

                        bool bHeaderOK = true;
                        if(sHeader == null || sHeader.Length < sFields.Length)
                                bHeaderOK = false;

                        //get the timestamp
                        DateTime stamp;
                        if(DateTime.TryParse(sFields[1], out stamp))
                                line.TimeStamp = stamp;

                        for(int i = 0; i < 19; i++) {
                                if(i == 8) //wind direction
                                        line[i] = new LogValueString(bHeaderOK ? sHeader[2 + i] :
_sFieldNames[i], sFields[2 + i]);
                                else
                                        line[i] = new LogValueDouble(bHeaderOK ? sHeader[2 + i] :
_sFieldNames[i], sFields[2 + i], _sFieldFormat[i]);
                        }
                        return line;
                }
}
```

ODE Solver (Simulation, c#)

```
public enum SolverType { FE,Heun,RK4};
public class ODESolver {
       public int Progress { get; protected set; }
       protected ODEModel _model;

       public ODESolver() {
              Progress = 0;
              _model = null;
       }
       public void SetModel(ODEModel model) {
              _model = model;
       }
       public virtual double[][] Solve(double[] x0, double[][] u, double tStart, double dt,
int N, out double[][] y) {
              Progress = 0;

              if(_model == null)
                     throw new Exception("No model asigned to solver");
              if(x0 == null)
                     throw new Exception("Initial state cannot be null");
              if(u == null)
                     throw new Exception("Input cannot be null");
              if(x0.Length <= 0)
                     throw new Exception("Initial state cannot be length less then 1");

              y = new double[N][];
              return null;
       }
}

public class SolverFE : ODESolver {
       public override double[][] Solve(double[] x0, double[][] u, double tStart, double
dt, int N, out double[][] y) {
              base.Solve(x0, u, tStart, dt, N, out y);

              int nx = x0.Length;
              double[][] x = new double[N][];
              double[] dxdt = x0;

              //set initial values at t=0
              x[0] = new double[nx];
              y[0] = _model.Measurments(x0);
              Array.Copy(x0, x[0], nx);

              for(int k = 1; k < N; k++) {

                     //evaluate function at current state/time
                     dxdt = _model.dxdt(x[k - 1], tStart + k * dt, u[k - 1]);
```

```
            //set next state
            x[k] = new double[nx];
            for(int i = 0; i < nx; i++)
                x[k][i] = x[k - 1][i] + dt * dxdt[i];

            _model.EndStep();
            y[k] = _model.Measurments(x[k]);
            Progress = (int)(k * 100 / (N - 1));
        }
        return x;
    }
}

public class SolverHeun : ODESolver {
    public override double[][] Solve(double[] x0, double[][] u, double tStart, double 
dt, int N, out double[][] y) {
        base.Solve(x0, u, tStart, dt, N, out y);

        int nx = x0.Length;
        double[][] x = new double[N][];
        double[] dxdt = x0;
        double[] xt = new double[nx];

        //set initial values at t=0
        x[0] = new double[nx];
        y[0] = _model.Measurments(x0);
        Array.Copy(x0, x[0], nx);

        for(int k = 1; k < N; k++) {

            //evaluate function at current state/time
            double t = tStart + k * dt;

            //FE
            double[] Ffe = _model.dxdt(x[k - 1], t, u[k - 1]);

            //x_fe
            for(int i = 0; i < nx; i++)
                xt[i] = x[k - 1][i] + dt * Ffe[i];

            double[] Fhn = _model.dxdt(xt, t + dt, u[k - 1]);


            //compute next state
            x[k] = new double[nx];
            for(int i = 0; i < nx; i++)
                x[k][i] = x[k - 1][i] + dt / 2.0 * (Ffe[i] + Fhn[i]);

            _model.EndStep();
```

```
                y[k] = _model.Measurments(x[k]);
                Progress = (int)(k * 100 / (N - 1));
            }
            return x;
        }
    }

public class SolverRK4 : ODESolver {
        public override double[][] Solve(double[] x0, double[][] u, double tStart, double
dt, int N, out double[][] y) {
            base.Solve(x0, u, tStart, dt, N, out y);

            int nx = x0.Length;
            double[][] x = new double[N][];
            double[] dxdt = x0;
            double[] xt = new double[nx];

            //set initial values at t=0
            x[0] = new double[nx];
            y[0] = _model.Measurments(x0);
            Array.Copy(x0, x[0], nx);

            for(int k = 1; k < N; k++) {

                //evaluate function at current state/time
                double t = tStart + k * dt;

                //F1d
                double[] F1d = _model.dxdt(x[k - 1], t, u[k - 1]);

                //F2d
                for(int i = 0; i < nx; i++)
                    xt[i] = x[k - 1][i] + dt / 2.0 * F1d[i];
                double[] F2d = _model.dxdt(xt, t + dt / 2.0, u[k - 1]);

                //F3d
                for(int i = 0; i < nx; i++)
                    xt[i] = x[k - 1][i] + dt / 2.0 * F2d[i];
                double[] F3d = _model.dxdt(xt, t + dt / 2.0, u[k - 1]);

                //F4d
                for(int i = 0; i < nx; i++)
                    xt[i] = x[k - 1][i] + dt * F3d[i];
                double[] F4d = _model.dxdt(xt, t + dt, u[k - 1]);

                //compute next state
                x[k] = new double[nx];
                for(int i = 0; i < nx; i++)
                    x[k][i] = x[k - 1][i] + dt / 6.0 * (F1d[i] + 2.0 * F2d[i] +
2.0 * F3d[i] + F4d[i]);
```

```
            _model.EndStep();
            y[k] = _model.Measurments(x[k]);
            Progress = (int)(k * 100 / (N - 1));
        }
        return x;
    }
}
```

ODE model (Simulation, c#)

```
public class ODEModel
{
        public Dictionary<string,double> Parameters{get;set;}

        public ODEModel() {
                Parameters = new Dictionary<string, double>();
        }

        public virtual double[] dxdt(double[] x, double t, double[] u){
                return null;
        }
        public virtual void EndStep() {
        }
        public virtual void Setup() {
        }
        public virtual double[] Measurments(double[] x) {
                return null;
        }
        protected double par(string sName) {
                if(Parameters.ContainsKey(sName))
                        return Parameters[sName];
                else
                        throw new Exception("No parameter by the name " + sName + " found");
        }
        public void LoadParamFile(string sFileName, bool bErase = true) {
                if(Parameters == null)
                        Parameters = new Dictionary<string,double>();

                if(bErase)
                        Parameters.Clear();

                if(!File.Exists(sFileName))
                        throw new Exception("Parameter file: " + sFileName + "not found!");

                string[] sLines = File.ReadAllLines(sFileName);

                for(int i = 0; i < sLines.Length; i++) {
                        if(sLines[i].Length > 0 && sLines[i][0] != '%' ) {

                                //params in one line
                                string[] sParams = sLines[i].Replace(" ", "").Replace("\t",
"").Split(';');

                                for(int k = 0; k < sParams.Length; k++) {
                                        if(sParams[k].Length > 0 && sParams[k][0] != '%' &&
sParams[k].Contains('=')) {
                                                string[] sTemp = sParams[k].Split('=');
                        if(sTemp.Length != 2)
                            throw new Exception("Format error on line " + (i + 1).ToString() + " of parameter file");

                        double fVal;
                        if(double.TryParse(sTemp[1], out fVal)) {
                            string sParamName = sTemp[0];

                            if(Parameters.ContainsKey(sParamName))
                                Parameters[sParamName] = fVal;
                            else
                                Parameters.Add(sParamName, fVal);
                        }
                    }
                }
            }
        }
    }
}

class ModelLP:ODEModel {
    public override double[] dxdt(double[] x, double t, double[] u) {
        base.dxdt(x, t, u);

        //get parameters
        double T = par("T");

        int nLen = Math.Min(x.Length,u.Length);

        double[] dxdt = new double[nLen];
        for(int i = 0; i < nLen;i++)
            dxdt[i] = 1 / T * (u[i] - x[i]);

        return dxdt;
    }
}

class ModelR4C2 : ModelRC {
    //parameters
    double R_b, R_w, R_g;
    double C_b, C_w;

    public override void Setup() {
        base.Setup();

        //PARAMETERS

        //THERMAL RESISTANCES
        R_b = par("R_b");
        R_w = par("R_w");
```

```
                //connecting thermal resistances for window, door, floor and roof in paralell
                R_g = par("R_g");


                //THERMAL CAPACITANCE
                C_b = par("C_b");
                C_w = par("C_w");
        }
        public override double[] dxdt(double[] x, double t, double[] u) {
                base.dxdt(x, t, u);

                #region extract states and inputs from arguments
                //number of states, alocate return array
                int nx = 2;
                if(x.Length != nx)
                        throw new Exception("Not enough states in x0. Found " +
x.Length.ToString() + ", need " + nx.ToString());
                int nu = 7;
                if(u.Length != nu)
                        throw new Exception("Not enough inputs in u. Found " +
u.Length.ToString() + ", need " + nu.ToString());

                //EXTRACT STATES
                double T_b = x[0];
                double T_w = x[1];

                //EXTRACT INPUTS

                //heat sources
                double Qheater = u[0];
                double Qpeople = u[1];
                double Qappliences = u[2];
                double Qsolar = u[3];
                double Qextsolar = u[4];
                //outside weath condition parameters
                double T_inf = u[5];

                //ventilation
                double V_e = u[6];
                #endregion

                #region Model

                //sum up heat sources
                double Q1 = Qheater + Qappliences;
                double Q2 = Qsolar;

                //ventialtion equivalent resistance
                double R_v = Ventilation(V_e);
```

```
              //DIFFERENTIAL EQUATIONS
              double dT_b = 1 / C_b * Q1    -    1 / (C_b * R_b) * (T_b - T_w)    -    1 /
(C_b * R_g) * (T_b - T_inf)    -    1 / (C_b * R_v) * (T_b - T_inf);
              double dT_w = 1 / C_w * Q2    -    1 / (C_w * R_b) * (T_w - T_b)    -    1 /
(C_w * R_w) * (T_w - T_inf);

              #endregion

              #region return differentials
              //RETURN DIFFERENTIALS
              double[] dxdt = new double[nx];
              dxdt[0] = dT_b;
              dxdt[1] = dT_w;
              return dxdt;
              #endregion
       }
}
```

Simulation Framework (Simulation, c#)

```
public class SimulationCtrl {

        private Thread _thrSimulation;
        private bool _bRunSim = true;

        public delegate void SimulationComplete(Simulation sim, double nRunTimeMS);
        public event SimulationComplete OnSimulationComplete;

        public delegate void SimulationFailed(string sError);
        public event SimulationFailed OnSimulationFailed;

        private List<Simulation> _lstSimQue;

        public int GetSolverProgress() {
                if(_lstSimQue != null && _lstSimQue.Count > 0)
                        return _lstSimQue[0].GetSolverProgress();
                else
                        return 100;
        }

        public SimulationCtrl() {
                _lstSimQue = new List<Simulation>();
                _thrSimulation = new Thread(DoSimulation);

        }
        public void Start() {
                _bRunSim = true;
                _thrSimulation.Start();
        }
        public void Stop() {
                _bRunSim = false;
                _thrSimulation.Abort();
        }

        public void QueSimulation(Simulation sim) {
                _lstSimQue.Add(sim);
        }

        private void DoSimulation() {
                while(_bRunSim) {
                        if(_lstSimQue.Count > 0) {
                                try {
                                        DateTime dtStart = DateTime.Now;
                                        _lstSimQue[0].Run();
                                        TimeSpan tsRunTime = DateTime.Now - dtStart;
                                        if(OnSimulationComplete != null)
                        OnSimulationComplete(_lstSimQue[0],
tsRunTime.TotalMilliseconds);
                }
                catch(Exception ex) {
                    string sType = ex.GetType().ToString();
                    if(ex.GetType().ToString() !=
"System.Threading.ThreadAbortException") {
                        if(OnSimulationFailed != null)
                            OnSimulationFailed(ex.Message);
                    }
                }
                finally {
                    _lstSimQue.RemoveAt(0);
                }
            }
            else
                Thread.Sleep(100);
        }
    }
}

public class Simulation {
    public ODESolver Solver { get; protected set; }
    public ODEModel Model { get; protected set; }
    public double[] x0 { get; private set; }
    public int N { get; private set; }
    public double dt { get; private set; }
    public double[][] Result { get; protected set; }
    public double RunTime { get; protected set; }
    public int Index { get; protected set; }
    public string ParamFileName { get; protected set; }

    public string[] sInput { get; protected set; }
    public double[][] fInput { get; protected set; }
    public double[][] Measurments { get; protected set; }

    private DateTime _dtStart;

    public string SolverType {
        get {
            if(Solver == null)          return "None";
            else                        return
Solver.GetType().ToString().Replace("NumSimLib.Solver.","");
        }
    }
    public int GetSolverProgress() {
        if(Solver == null)          return 0;
        else                        return Solver.Progress;
    }
```

```
        public double[] GetState(int nState, int nStep, double fMin = double.MinValue, 
double fMax = double.MaxValue) {
                double[] fRes;
                fRes = new double[N / nStep];
                for(int i = 0; i < N / nStep; i ++) {
                        fRes[i] = Math.Min(Math.Max(fMin, Result[i * nStep][nState]), fMax);
                }
                return fRes;
        }
        public double[] GetInput(int nInput, int nStep, double fMin = double.MinValue, 
double fMax = double.MaxValue) {
                double[] fRes;
                fRes = new double[N / nStep];
                for(int i = 0; i < N / nStep; i ++) {
                        fRes[i] = Math.Min(Math.Max(fMin, fInput[i * nStep][nInput]), fMax);
                }
                return fRes;
        }
        public double[] GetMeasurment(int nInput, int nStep, double fMin = double.MinValue, 
double fMax = double.MaxValue) {
                double[] fRes;
                fRes = new double[N / nStep];
                for(int i = 0; i < N / nStep; i++) {
                        fRes[i] = Math.Min(Math.Max(fMin, Measurments[i * nStep][nInput]), 
fMax);
                }
                return fRes;
        }
        public void Setup(int index, ODESolver solver, ODEModel model, double[] 
initial_conditions, int sim_horizion, double time_step,string[] input,string parfile) {
                Index = index;
                Solver = solver;
                Model = model;
                x0 = initial_conditions;
                N = sim_horizion;
                dt = time_step;
                sInput = input;
                ParamFileName = parfile;
                RunTime = 0;


        }


        public void Run() {
                //set the model to solver
                if(Solver == null)
                        throw new Exception("No solver configured for simulation");
                StartTimer();
                //set model parameters
                Model.LoadParamFile(ParamFileName);
        //call setup to let the model init itself
        Model.Setup();

        //set the model to solver
        Solver.SetModel(Model);

        //decode u strings
        List<string> lstInputs = new List<string>();
        for(int i = 0; i < sInput.Length; i++) {
            string sU = sInput[i].Replace(" ", "").Replace("\t", "").ToLower();

            //has multiplication
            int nStart = sU.IndexOf('x');
            int nMultiplier = 1;
            if(nStart != -1) {
                if(!int.TryParse(sU.Substring(0, nStart), out nMultiplier))
                    nMultiplier = 1;
                sU = sU.Substring(nStart + 1);
            }
            for(int k = 0; k < nMultiplier; k++) {
                lstInputs.Add(sU);
            }
        }
        //create the input vector for LP simulation (all 0), decode u strings
        fInput = new double[N][];
        for(int i = 0; i < N; i++) {
            //extract CSV values
            string[] sVals = lstInputs[i % lstInputs.Count].Split(',');
            fInput[i] = new double[sVals.Length];
            for(int k = 0; k < sVals.Length; k++) {
                double.TryParse(sVals[k], out fInput[i][k]);
            }
        }

        //solve the model and store results
        double[][] y;
        Result = Solver.Solve(x0, fInput, 0, dt, N, out y);
        Measurments = y;
        StopTimer();
    }

    protected void StartTimer() {
        _dtStart = DateTime.Now;
    }
    protected void StopTimer() {
        TimeSpan ts = DateTime.Now - _dtStart;
        RunTime = ts.TotalMilliseconds;
    }
}
```

Parameter Identification (MATLAB)

```
function [theta_opt] = optimize(theta0,theta,u,x0,N,dt,model,r,Ew)
    %% init local worker variables
    thetaLast = [];
    myJ = [];
    myG = [];
    myHeq = [];

    %% run optimizer
    options=optimset('Algorithm','interior-point','Display','off','MaxIter',300,'UseParallel','always');

    % check bounds in fmincon speed, group inputs
    [theta_opt] = fmincon(@objective,theta0,[],[],[],[],[],[],@constraints,options);

    %% objective func
    function J = objective(theta)
        if ~isequal(theta,thetaLast)
            [myJ, myG, myHeq] = compute(theta,theta0,u,x0,N,dt,model,r,Ew);
            thetaLast = theta;
        end
        %return myJ
        J = myJ;
    end

    %% constraints func
    function [G, Heq] = constraints(theta)
        if ~isequal(theta,thetaLast)
            [myJ, myG, myHeq] = compute(theta,theta0,u,x0,N,dt,model,r,Ew);
            thetaLast = theta;
        end
        %return myG and myHeq
        G = myG;
        Heq = myHeq;
    end
end


function [myJ myG myHeq] = compute(theta,theta0,u,x0,N,dt,model,r,Ew)
    %% Solve ODE with current theta
    [x,y,u] = simulate(u,x0,N,dt,model,theta);
    theta_rel = theta./theta0;

    %% compute J
    J = 0;
    for i = 1:size(r,2)
        e = x(:,i)-r(:,i);
        J = J + e'*e*Ew(i);
    end


    %% return J,G,Heq
    myG = [  theta_rel - 3.0;
             0.3 - theta_rel ];
    myJ = J;
    myHeq = [];

end

function [x_store,y_store,u] = simulate(u,x0,N,dt,model,theta)

    %init sim loop
    nx = size(x0,1);
    ny = 2;
    x_store = zeros(N,nx);
    y_store = [];

    x = x0;
    for i = 1:N
        u_cur = u(i,:)';

        %solve the ODE one timestep at a time
        [dxdt,y] = RK4(x,i*dt,u_cur,dt,@(t,x,u) model(t,x,u,theta));
        x = x + dt * dxdt;

        x_store(i,:)=x;

        if isempty(y_store)
            ny = size(y,1);
            y_store = zeros(N,ny);
        end

        if ny > 0
            y_store(i,:) = y;
        end
    end
end

function [dxdt,y] = RK4(x,t,u,dt,model)
    %F1d
    [F1d,y] = model(t,x,u);

    %F2d
    xt = x + dt / 2.0 * F1d;
    [F2d,y] = model(t + dt/2.0,xt,u);
    %F3d
    xt = x + dt/2.0 * F2d;
    [F3d,y] = model(t + dt/2.0,xt,u);

    %F4d
    xt = x + dt * F3d;
    [F4d,y] = model(t + dt,xt,u);

    %compute next state
    dxdt = 1 / 6.0 * (F1d + 2.0 * F2d + 2.0 * F3d + F4d);
    [F,y] = model(t + dt,x+dt*dxdt,u);
end
```

Model implementations (MATLAB)

```
function [dxdt,y] = ModelR4C2(t,x,u,theta)
    % load parameters
    R_b         = theta(1);
    R_w         = theta(2);
    R_g         = theta(3);
    R_vent      = theta(4);
    C_b         = theta(5);
    C_w         = theta(6);

    %extract states
    T_b = x(1);
    T_w = x(2);

    %heat sources
    Qheater = u(1);
    Qpeople = u(2);

    Qappliences = u(3);
    Qsolar = u(4);
    Qextsolar = u(5);

    %outside weath condition parameters
    T_inf = u(6);

    %ventilation
    N = u(7);


    %sum up heat sources
    Q1 = Qheater + Qappliences;
    Q2 = Qsolar;

    %ventialtion equivalent resistance
    R_v = Ventilation(N,R_vent);

    %DIFFERENTIAL EQUATIONS
    dT_b = 1 / C_b * Q1    -    1 / (C_b * R_b) * (T_b - T_w)    -    1 / (C_b *
R_g) * (T_b - T_inf)    -    1 / (C_b * R_v) * (T_b - T_inf);
    dT_w = 1 / C_w * Q2    -    1 / (C_w * R_b) * (T_w - T_b)    -    1 / (C_w *
R_w) * (T_w - T_inf);

    %RETURN DIFFERENTIALS
    dxdt = zeros(2,1);
    dxdt(1) = dT_b;
    dxdt(2) = dT_w;

    y = [];
end
```

```
function [dxdt,y] = ModelR6C2(t,x,u,theta)
    % load parameters
    R_b         = theta(1);
    R_w         = theta(2);
    R_s         = theta(3);
    R_e         = theta(4);
    R_g         = theta(5);
    R_vent      = theta(6);
    C_b         = theta(7);
    C_w         = theta(8);

    %extract states
    T_b = x(1);
    T_w = x(2);

    %heat sources
    Qheater = u(1);
    Qpeople = u(2);

    Qappliences = u(3);
    Qsolar = u(4);
    Qextsolar = u(5);

    %outside weath condition parameters
    T_inf = u(6);

    %ventilation
    N = u(7);


    %sum up heat sources
    Q1 = Qheater + Qappliences;
    Q2 = Qsolar;
    Q3 = Qextsolar;

    %ventialtion equivalent resistance
    R_v = Ventilation(N,R_vent);

    %ALGEBRAIC NODE EQUATIONS (store for later return by Measurments)
    T_s = (R_b * R_s * Q2 + R_b * T_w + R_s * T_b) / (R_b + R_s);
    T_h = (R_e * R_w * Q3 + R_e * T_w + R_w * T_inf) / (R_e + R_w);

    %DIFFERENTIAL EQUATIONS
    dT_b = 1 / C_b * Q1    -    1 / (C_b * R_b) * (T_b - T_s)    -    1 / (C_b *
R_g) * (T_b - T_inf)    -    1 / (C_b * R_v) * (T_b - T_inf);
    dT_w =                 -    1 / (C_w * R_s) * (T_w - T_s)    -    1 / (C_w *
R_w) * (T_w - T_h);

    %RETURN DIFFERENTIALS
    dxdt = zeros(2,1);
    dxdt(1) = dT_b;
    dxdt(2) = dT_w;

    y= zeros(2,1);
    y(1) = T_s;
    y(2) = T_h;
end

function [dxdt,y] = ModelR6C3(t,x,u,theta)
    % load parameters
    R_b         = theta(1);
    R_w         = theta(2);
    R_s         = theta(3);
    R_e         = theta(4);
    R_g         = theta(5);
    R_vent      = theta(6);
    C_b         = theta(7);
    C_w         = theta(8);
    C_s         = theta(9);

    %extract states
    T_b = x(1);
    T_w = x(2);
    T_s = x(3);

    %heat sources
    Qheater = u(1);
    Qpeople = u(2);

    Qappliences = u(3);
    Qsolar = u(4);
    Qextsolar = u(5);

    %outside weath condition parameters
    T_inf = u(6);

    %ventilation
    N = u(7);


    %sum up heat sources
    Q1 = Qheater + Qappliences;
    Q2 = Qsolar;
    Q3 = Qextsolar;

    %ventialtion equivalent resistance
    R_v = Ventilation(N,R_vent);

    %ALGEBRAIC NODE EQUATIONS (store for later return by Measurments)
    T_h = (R_e * R_w * Q3 + R_e * T_w + R_w * T_inf) / (R_e + R_w);
    %DIFFERENTIAL EQUATIONS
    dT_b = 1 / C_b * Q1       -       1 / (C_b * R_b) * (T_b - T_s)     -          1
/ (C_b * R_g) * (T_b - T_inf)     -       1 / (C_b * R_v) * (T_b - T_inf);
    dT_w =                    -       1 / (C_w * R_s) * (T_w - T_s)     -          1
/ (C_w * R_w) * (T_w - T_h);
    dT_s = 1 / C_s * Q2       -       1 / (C_s * R_b) * (T_s - T_b)     -          1
/ (C_s * R_s) * (T_s - T_w);

    %RETURN DIFFERENTIALS
    dxdt = zeros(3,1);
    dxdt(1) = dT_b;
    dxdt(2) = dT_w;
    dxdt(3) = dT_s;

    y= zeros(1,1);
    y(1) = T_h;
end

function [dxdt,y] = ModelR7C3(t,x,u,theta)
    % load parameters
    R_b         = theta(1);
    R_w1        = theta(2);
    R_w2        = theta(3);
    R_s         = theta(4);
    R_e         = theta(5);
    R_g         = theta(6);
    R_vent      = theta(7);
    C_b         = theta(8);
    C_w1        = theta(9);
    C_w2        = theta(10);

    %extract states
    T_b = x(1);
    T_w1 = x(2);    %middle of wall
    T_w2 = x(3);    %wall inside building

    %heat sources
    Qheater = u(1);
    Qpeople = u(2);

    Qappliences = u(3);
    Qsolar = u(4);
    Qextsolar = u(5);

    %outside weath condition parameters
    T_inf = u(6);

    %ventilation
    N = u(7);
    %sum up heat sources
    Q1 = Qheater + Qappliences;
    Q2 = Qsolar;
    Q3 = Qextsolar;

    %ventialtion equivalent resistance
    R_v = Ventilation(N,R_vent);

    %ALGEBRAIC NODE EQUATIONS (store for later return by Measurments)
    T_s = (R_b * R_s * Q2 + R_b * T_w2 + R_s * T_b) / (R_b + R_s);
    T_h = (R_e * R_w1 * Q3 + R_e * T_w1 + R_w1 * T_inf) / (R_e + R_w1);

    %DIFFERENTIAL EQUATIONS
    dT_b = 1 / C_b * Q1     -    1 / (C_b * R_b) * (T_b - T_s)       -    1 / (C_b *
R_g) * (T_b - T_inf)        -        1 / (C_b * R_v) * (T_b - T_inf);
    dT_w1 =                 -    1 / (C_w1 * R_w2) * (T_w1 - T_w2)   -    1 / (C_w1 *
R_w1) * (T_w1 - T_h);
    dT_w2 =                 -    1 / (C_w2 * R_s) * (T_w2 - T_s)     -    1 / (C_w2 *
R_w2) * (T_w2 - T_w1);

    %RETURN DIFFERENTIALS
    dxdt = zeros(3,1);
    dxdt(1) = dT_b;
    dxdt(2) = dT_w1;
    dxdt(3) = dT_w2;

    y= zeros(2,1);
    y(1) = T_s;
    y(2) = T_h;
end

function [dxdt,y] = ModelR5C3(t,x,u,theta)
    % load parameters
    %data;
    R_b         = theta(1);
    R_w         = theta(2);
    R_fur       = theta(3);
    R_g         = theta(4);
    R_vent      = theta(5);
    C_b         = theta(6);
    C_w         = theta(7);
    C_fur       = theta(8);

    %extract states
    T_b     = x(1);
    T_w     = x(2);
    T_fur   = x(3);

    %heat sources
    Qheater = u(1);
    Qpeople = u(2);
    Qappliences = u(3);
    Qsolar = u(4);
    Qextsolar = u(5);

    %outside weath condition parameters
    T_inf = u(6);

    %ventilation
    N = u(7);


    %sum up heat sources
    Q1 = Qheater + Qappliences;
    Q2 = Qsolar;

    %ventialtion equivalent resistance
    R_v = Ventilation(N,R_vent);

    %DIFFERENTIAL EQUATIONS
    dT_b    = 1 / C_b * Q1    -    1 / (C_b * R_b) * (T_b - T_w)    -    1 / (C_b *
R_fur) * (T_b - T_fur)   -    1 / (C_b * R_g) * (T_b - T_inf)    -    1 / (C_b *
R_v) * (T_b - T_inf);
    dT_w    = 1 / C_w * Q2    -    1 / (C_w * R_b) * (T_w - T_b)    -    1 / (C_w *
R_w) * (T_w - T_inf);
    dT_fur  = -    1 / (C_fur * R_fur) * (T_fur - T_b);

    %RETURN DIFFERENTIALS
    dxdt = zeros(3,1);
    dxdt(1) = dT_b;
    dxdt(2) = dT_w;
    dxdt(3) = dT_fur;

    y = [];
end
```

Appendix D - Summary sheet

Telemark University College

Faculty of Technology

M.Sc. Programme

MASTER'S THESIS, COURSE CODE FMH606

Student: **Ole Magnus Brastein (130482)**

Thesis title: **Grey-box models for estimation of heating times for buildings**

Signature: . . . .

Number of pages: 215

Keywords: Grey-box models

Heating time prediction

Thermal network models

Supervisor: Nils-Olav Skeie Sign.: . . . .

2nd supervisor: Carlos Pfeiffer Sign.: . . . .

Censor: Håkon Tjelland Sign.: . . . .

External partner: Sign.: . . . .

Availability: <Open >

Archive approval (supervisor signature)**:** Sign.: . . . . **Date :** . . . .

Abstract:

Energy-usage in buildings is responsible for a large part of the total demands on energy production. Models are required to estimate the heating and cooling times of buildings, thus allowing a control system to accurately maintain comfort temperature only when strictly needed, lowering the demand for energy used for heating.

Grey-box models based on Thermal Network Resistor-Capacitor equivalents are used to predict thermal behavior of buildings. Model structures are based on cognitive or intuitive understanding of thermodynamic behavior of buildings.

Working with models and data sets requires software tools both for treatment of data, simulation of models and identification of parameters. In this project software is developed both in c# and MATLAB as and when applicable.

Grey-box models are shown to accurately predict temperature over the prediction horizon, leading to accurate estimation of heating time, when compared to measurement data. Further, the proposed control strategy is shown to overcome some of the shortcomings of standard heater control systems.

Telemark University College accepts no responsibility for results and conclusions presented in this report.