Bachelor Thesis

A Web-based Framework for Sensor Networks - Case Study of a Temperature Network for Precise Metrology

Department of Geodesy and Geoinformation
Research Division Engineering Geodesy
Vienna University of Technology

Author: David Rejchrt
Matriculation number: 01428817

Supervisors: Univ. Prof. Dr. Ing. Hans-Berndt Neuner
Dipl.-Ing. Tomas Thalmann

Vienna, January 2020

Abstract

Recent development in the field of electronics has achieved a significant decrease of price while increasing performance of the components. This leads to a development of a multi sensor systems which includes sensor networks. In this thesis a framework for multi purpose sensor networks is conceptualized and subsequently implemented. The framework uses Robot Operating System (ROS) in implementation of the network itself and a Django Web Application as a user interface for the framework. The functionality of the framework is demonstrated on deployment of a permanent temperature sensor network in measurement lab of our research group. The temperature sensors are prototypes that were developed in our research group and had to be calibrated before installation in the measurement lab of our institute. A calibration experiment was conducted in order to show specific use case. There were also two hypotheses tested. First hypotheses states, that it is possible to replace one component in a sensor without having to conduct a new calibration and the second hypotheses states, that the ambient temperature itself has no effect on the measurement. In the end, both of the hypotheses were rejected.

Contents

1 Introduction **5**

2 Sensor Networks **5**

2.1 Sensor nodes 7

2.2 Central Computer and User Interface 9

2.3 Network 10

3 ROS **10**

3.1 Nodes and Topics 11

3.2 Parameters and Launch files 12

3.3 Services 13

4 Implementation of the framework **13**

4.1 Igors ROS Package 14

4.1.1 Data acquisition node 15

4.1.2 Messages 15

4.1.3 Data storage node 17

4.1.4 Time synchronization 17

4.2 Django Web Application 18

4.2.1 Django 19

4.2.2 Design/bootstrap 21

4.2.3 Home View 22

4.2.4 Job View 23

4.3 Database 25

4.3.1 Values 26

4.3.2 Jobs . . . 26

4.3.3 Machines . . . 27

4.3.4 Sensors . . . 27

4.3.5 Sensor Type . . . 27

4.3.6 Quantity . . . 28

4.3.7 SensorTypeQuantity . . . 28

5 Electrical Sensor and Calibration **28**

5.1 Signal conversion . . . 28

5.2 Signal processing . . . 29

5.3 Calibration . . . 30

6 Calibration Experiment **31**

6.1 Instruments . . . 31

6.1.1 Sensors . . . 31

6.1.2 Climatic chamber . . . 35

6.2 Experiment . . . 35

6.3 Evaluation . . . 37

6.3.1 Evaluation module . . . 37

6.3.2 Series . . . 37

6.3.3 Calibrator . . . 39

6.3.4 Data formatting . . . 40

6.3.5 Raw data . . . 40

6.3.6 Filtering . . . 42

6.3.7 Classification and equalisation . . . 45

6.3.8 Line fitting . . . 47

6.4 Results of the experiment 50

6.4.1 Interpretation 51

6.4.2 Hypotheses testing 53

6.4.3 Deployment of another A/D Converter 54

6.4.4 Effect of the ambient temperature on performance of the sensors 54

7 Summary **56**

7.1 Implementation of the framework 56

7.2 Calibration experiment 57

8 Outlook **57**

1 Introduction

A key competence of engineering geodesy is planning, organization and execution of measurement campaigns, generally speaking the data acquisition process in a specific application. This provides the foundation for further in-depth analysis and evaluation to generate the required information. This information is necessary to conclude the right decisions depending on the specific goal or task of the application. In most applications, multiple sensors are necessary due to the growing level of complexity. From an application point of view, a single sensor is at the retreat. [1]

Due to the technological development over the last years, sensors and affiliated hardware has become cheaper, smaller and less power consuming. A development, also driven by hot topics like Industry 4.0 and Internet of Things (IoT). This also leads to an increasing number of sensors involved in data acquisition tasks in engineering geodesy. Furthermore, permanently mounted sensor networks required for real-time or near-real-time applications become more feasible and economical. [1]

There are already several software solutions for separate applications which are mostly developed for one specific task. The range of hardware compatible with these solutions is not very wide and is not easy to use outside of the manufacturer's software. This represents another downside of current products, as the software is often proprietary and closed source, which limits the community driven development.

This invokes a question, whether it is possible, to create a framework for sensor networks, that would be hardware and task independent. In this thesis, such framework is conceptualized and implemented. As a case study, the framework will be deployed as a temperature sensor network inside the measurement lab of our institute which can increase the accuracy of laser distance measurement, as the atmosphere, through which a laser beam propagates will be more accurately described.

2 Sensor Networks

A sensor network is a set of sensors capable of autonomous operation connected together in a network in order to communicate and exchange data with each other. It is used to acquire state of an examined object by measuring physical (paragraph Physical quantities, page 6) quantities simultaneously on spatially distributed points. The data acquisition, depending on sensors, occurs at relatively high frequency and can provide near-real-time or even real-time data. If the spatial information about the individual sensor nodes is provided, one can

inspect the behaviour of examined object in both temporal as well as spatial dimension. [2]

A sensor network typically consists of following components:

- Sensor nodes, which provide measurement values
- Central computer, which manages the sensor nodes and stores the measured values
- Communication Network, through which the sensor nodes communicate with the central computer and with each other
- An user interface software that allows the user to process the data

In following sections the individual components and their functions are described in detail. Figure 1 shows a schematic representation of a sensor network, its components and functions.

The range of applications in which sensor networks are deployed is very wide. For example in environmental monitoring systems, which monitor air pollution or water quality. Sensor networks can be as well used as a detection systems, that help to prevent or detect imminent danger of a natural disaster thus potentially reducing causalities and damage costs. Other good example of usage case can be found in meteorology. In engineering geodesy, sensor networks can be deployed in monitoring tasks. A total station can be considered as sensor node as well, provided that the communication with the central computer can be established.

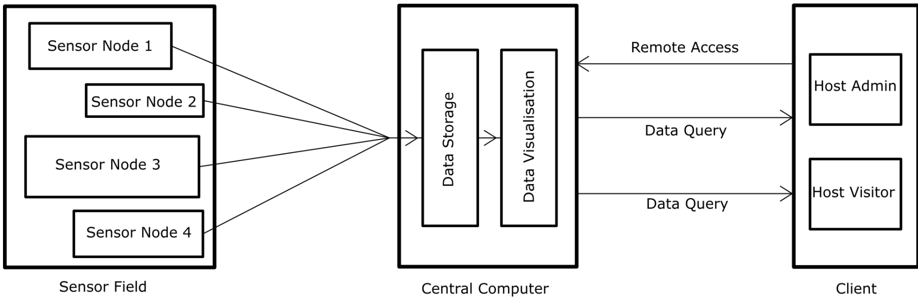


Figure 1: Components of the sensor network

Physical quantities

Physical quantity is used in this thesis as a hypernym for all different types physical quantities. These are for example:

- Mechanical (force, acceleration)
- Geometrical (length, angle)
- Thermodynamical (temeperature)
- ...

Physical quantity in context of this thesis does not contain electrical quantities like voltage, current or resistance. The reason for this is, that sensors generally convert physical quantities (other than electrical) into electrical quantities (section 5.1). This way, it should not come to any confusion while reading this thesis.

2.1 Sensor nodes

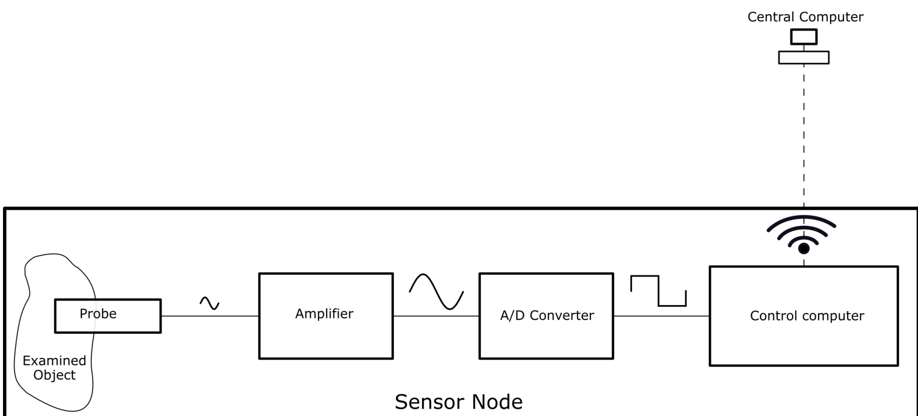


Figure 2: Schematic representation of an sensor node and its components

A sensor node is a device, that acquires a measurement value of some physical quantity and assigns a time stamp to that measurement. The sensor node consists typically out of four components:

- Probe
- Amplifier
- A/D converter
- Control computer

Figure 2 shows a schematic representation of a sensor node and functions of its components.

A sensor is a device that converts an analogue physical signal that into an analogue electrical signal. The process of acquiring a discrete measurement values is described in detail in section 5.1. The sensors can be implemented either as a tiny single circuit board with all the components on one board, or every component separately connected in a circuit, in which case it would be wise to protect the non-probe parts of the sensor node in some sort of casing.

The advantage of the single board sensors is its compactness. These sensors can be very small don't have any cables or wires sticking out, which makes them easy to deploy. The disadvantage is, that even if the probe were waterproof, one can not submerge the sensor into a liquid, because of the open wirings. This might also cause a failure when measuring in a cold environments and bringing the sensor to a warmer environment, as the water droplets from the air would condensate on the surface and cause corrosion of the wires. Advantage of the multiple board design is, that one can easily tweak the sensor node's performance by combining different parts together. The downside is, that an extra effort is needed, since such designs wouldn't obviously work out-of-the-box. The separate parts might be also much more susceptible to physical damage. However with a suitable protection, the range of applications of such sensor increases significantly.

The signal generated by the probes is very weak and has to be amplified. The amplified signal is then processed by an analogue to digital converter (A/D converter) which outputs a computer comprehensible digital signal. More on this in section 5.2

The control computer receives the digital signal from A/D converter. Depending on whether the sensor node is connected to the network or not, it can either store the data on its own file system or even in its own database, or it sends the data over to the central computer. A user should not be interacting with these computers directly. Therefore there is no need for external peripheries or displays. Single board computers are the most suitable computers for this purpose, as they are small and mobile yet still can be treated as a standard computer, which simplifies the configuration.

The sensor nodes should to be portable which introduces the problem of the power supply. To ensure mobility of the device, the power consumption of the components is minimized, so that the sensor node can be powered by batteries for extended periods of time, or even by the means of energy harvesting, such as solar panels.

2.2 Central Computer and User Interface

The central computer is the most important computer in a sensor network. It can be a common workstation or a portable laptop. Unlike sensor node control computers, this computer has a standard (graphical) user interface with peripheral devices such as keyboard, mouse and a display. It is connected to the sensor network, and can be also used as a internet gateway, which can be used for establishing a remote access or for time synchronization of the sensor nodes (section 4.1.4). Main functions of the central computer in a sensor network are:

- Management of stored measurements
- Management of the sensor nodes and evtl. network configuration
- User Interface

The measurement data are sent from the sensor node to the central computer, where the data are saved either into a file or a database. The database can also contain data about the network configuration.

The central computer also provides a user interface for operating the sensor network. An operator can start or stop the measurement sessions, or configure the network for a different task, or process the measured data, as the central computer might provide task-specific evaluation software.

One can set up multiple user accounts on the computer and assign them access rights. The users can be also sorted into groups like visitors, operators and administrators. If internet connection is available, the users can establish remote access to the central computer. Visitors might be able to view the data, operators might be able to start or stop their own measurement sessions and manipulate the data from these sessions, and administrators would have rights to change the configuration of the whole network and system, including the ability to add or remove users and change their rights inside the system.

The control computer can also act as a web server. On such web server, there could run a web application which provide the access to software for data management and evaluation. It is also possible to set up a user based access system on the web server. There are two advantages of a web server. Firstly, the users remotely access the server and not the whole computer, so there is a lower risk of the computer being misused. Secondly, the users access the computer through their browser, which provides a simpler, more user-friendly remote access, than e.g. *ssh* (Secure Shell) or Remote Desktop.

2.3 Network

The main function of the network is to enable the sensor nodes to communicate with the central computer and eventually with each other. The components of the network can be connected by an ethernet cable, or wirelessly, using different communications technologies, like WiFi, Bluetooth, or similar, depending distance the signals need to travel. The network can be set up in two modes: ad-hoc or industrial. The most important aspect of the network is, that every device on the network must have an unique identifier, such as IP address.

If the network is run in ad-hoc mode, the devices on that network communicate directly with each other on peer to peer basis. It is easier to set the network up in this mode but the more the devices on the network, the slower the communication is. Another downside of ad-hoc networks is, that they use up more resources, as it communicates with all device on the network. With growing number of devices there is also an issue of interference.

The other possibility is to set the network up in an industrial mode, where the computers communicate with each other through a central router. The set up can be more tedious than ad-hoc network, but it provides a faster, more stable and less power consuming communication. The interference effects are also reduced, as the devices communicate over one device only - the router.

3 ROS

Robot Operating System or *ROS* is an open source framework for programming robotic networks. It provides an elegant, easy and robust solution for connecting many separate computing units together in a neatly structured network, thus enabling all machines on the network to communicate with each other. This framework, as the name suggests, comes from the field of robotics, but because of its flexibility, it is possible to find applications in many other fields, engineering geodesy including.

One of the key concepts of ROS is its modularity as the whole system consists of many different packages, which makes reusability of the code simple. There are already many packages available to use. In 2019, 11403 unique packages are available in the free to use repositories [3]. In addition, many manufacturers are aware of the impact of ROS, thus already providing ROS packages (drivers) for their hardware. These packages are able to interact smoothly with each other because ROS defines the underlying interfaces and protocols.

3.1 Nodes and Topics

A ROS package consists of one or more ROS *nodes*, which are essential pieces of code that solve a small part of the whole task. For example, a driver node encapsulates some piece of hardware. On the other hand, an algorithm node implements a methodical processing step. For example, a total station driver node handles the measurement process and streams spherical coordinates to a subsequent algorithm node, which converts these to cartesian coordinates. Subsequently, another node transforms these coordinates to the project coordinate system. A productive data stream is created by chaining multiple nodes, which allows solving complex tasks using divide and conquer design paradigm. A graphical representation (called ROS graph) of all ROS *nodes* is shown in figure 3.

Data and information between those nodes are exchanged through so-called ROS *topics*. A ROS *topic* is a subject oriented abstract space or data stream. A topic is strongly typed, which means that meta-information like structure and type of the data is strictly defined in a ROS *message*. A ROS *message* is a packet of structured information through which data is exchanged in a ROS network. The structure of this message is fully customizable (section 4.1.2). ROS provides several pre-defined message types.

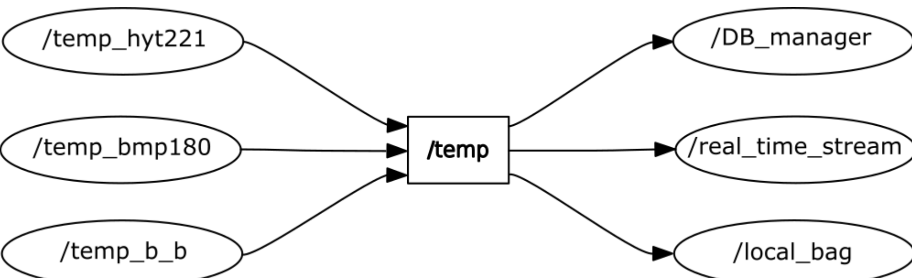


Figure 3: Example of ROS Graph. An ellipse represents a node and topic is shown as a rectangle.

Whereas a *message* defines the kind of data, a ROS *topic* is the data stream through which *nodes* share data and information. It is possible for two or more *nodes* to publish to the same *topic* as well as an arbitrary number of *nodes* is able to subscribe to a *topic* and receive all published *messages*. Once a *message* arrives at a subscriber *node*, the *message* gets decomposed and the data is further processed according to the purpose of this specific *node*. ROS is completely responsible for the underlying communication, so that the developer can focus on the actual problem.

Every topic is visible in the whole ROS network. This is achieved by a

central part of ROS, called the ROS *Master* that serves as 'broker' who keeps track about topics, subscriptions and publishers [4].

3.2 Parameters and Launch files

A ROS *node* should be developed as a reusable piece of code for a recurring task or problem. No matter if it abstracts a hardware device or encapsulates a specific processing step, it is good practice to define some configuration switches or settings. For instance, one might design the measurement rate, thresholds or processing parameters like constants or filter parameters. With such an approach, it is possible to enable the program to an even broader range of application and therefore improve its reusability.

These settings are generally called ROS *parameters* in the ROS terminology. These *parameters* are stored and managed centrally, so that every *node* can access its own private *parameters* as well as global or local *parameters*. At execution time, the *parameters* are parsed by a *parameter server*. A *parameter server* is a shared, multi-variate dictionary. *Nodes* use this server to store and retrieve *parameters* at runtime. It is meant to be globally viewable so that tools or *nodes* can easily inspect the configuration state of the system and modify it if necessary.

This makes the key requirement for geodetic sensor networks of online administration and configuration very easy, since changes at this central parameter server are automatically propagated to the affected parts of the sensor network.

As mentioned above, one of the key concepts of ROS is its modularity. As a consequence, a number of *nodes* and *topics* will be necessary to tackle a specific task or to solve a specific problem. For different applications one might need different sets of *nodes* that interact in specific ways with each other. To represent/reproduce these variable applications (or use cases) ROS introduces a helpful concept called a launch file, which is a XML file where it is specified, which *nodes* should run, with which *parameters* and on which machine. An example of such file is shown in listing 1. Before launching individual *nodes*, it makes sure that all prerequisites (such as *master* and *parameter server* are running) have been met.

Listing 1: Example of a launch file

```
<launch>
  <machine address="10.11.12.13" env-loader="~/ws/
     devel/env.sh" name="igros"/>
  <machine address="10.11.12.20" env-loader="~ros/ws/
     devel/env.sh" name="drone-pi" />
  <machine address="10.11.12.21" env-loader="~ros/ws/
     devel/env.sh" name="tps-pi" />
```

```
  <node machine="igros" name="db" pkg="igros" type="
     dbNode.py">
    <param name="bulkSize" value="100" />
   </node>
   <node machine="drone" name="imu" pkg="igros" type="
      imuNode.py">
     <param name="rate" value="200" />
    </node>
   <node machine="drone" name="gnss" pkg="igros" type=
      "gnssNode.py">
     <param name="rate" value="0.5" />
   </node>
   <node machine="tps" name="tps" pkg="leica_ros" type
      ="tpsNode.py">
     <param name="type" value="MS50" />
     <param name="reflector" value="GRZ4" />
     <param name="rate" value="5" />
   </node>
   <node machine="tps" name="temp" pkg="igros" type="
      meteoSensorNode.py"></node>
</launch>
```

3.3 Services

ROS *Service* is a *node* that communicates with other *nodes* on a request/respond basis. Any *node* can send a request to the service *node*. The requests may have any number of input arguments for the service node. The service *node* processes this request and returns processed values to the requesting *node*. It is also important to mention, that the *services*, just like *topics* are strongly typed, but differ in the communication paradigm. Whereas ROS *topic* is used for data streaming, ROS *service* works on request/response basis.

4 Implementation of the framework

The whole framework has been developed as a single software package called IGROS which is a abbreviation of **I**ngenieur**g**eodsie **R**obot **O**perating **S**ystem. The package consists out of three separate components:

- igros ROS Package, which is responsible for the data acquisition of the sensor network

- igros Web Application, which provides the User Interface to the sensor network
- database, that connects the previous two components together

In this section there is a detailed description of every component.

4.1 Igors ROS Package

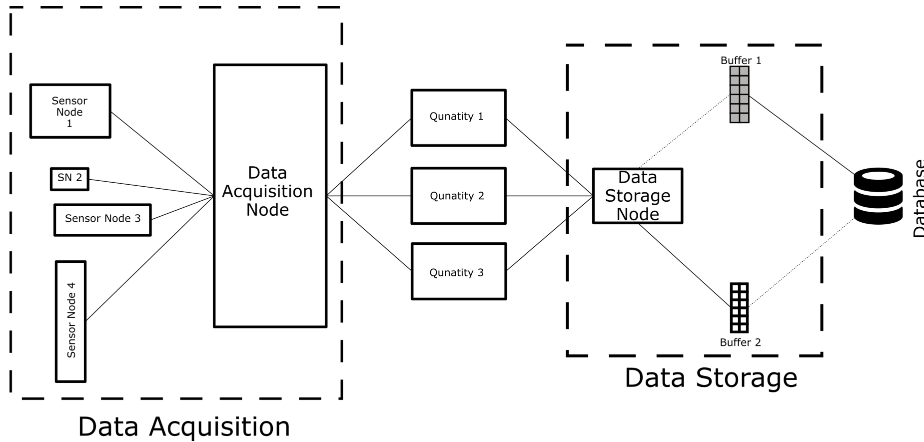


Figure 4: Data flow in igros ROS package

The aim of the ROS package was to standardise the data flow (fig 4). There are two separate steps in the data flow, first the actual data acquisition, which happens in the sensor field and the data storage which happens in the central computer. Since the variety of the sensors commonly used in engineering geodesy is fairly wide (meteorological sensors, GNSS antennas, total stations, ultra sound sensors etc.), the variety of the possible output formats is correspondingly wide. This also implies a separate database for the every combination of the sensors. Igros ROS Package abstracts the possible outputs into one common format. In the data acquisition step, output must be accordingly reformatted, which requires some programming effort, whereas the data storage step remains the same, as does the database.

In ROS, the two steps of the data flow is realised by two ROS nodes which exchange the data via topics (section 3.1). One node acquires the measurements from the sensors and the other node stores it. The topics, and therefore the messages are physical-quantity-based.

4.1.1 Data acquisition node

The function of this node is to run the code for reading the values from the sensors, assign a time stamp to the value and compact the data into a ROS Message and publish the message in a corresponding ROS topic.

The code that is used to communicate with the sensor has to simply return the measurement value. This code can be written by anyone: the manufacturer of the sensor, or internet community member as well as the user, that wants to use the sensor inside the igros framework. It only has to output the value.

The data acquisition node is affected by several ROS parameters. Two of them specify how one single value is produced: measurement rate f and measurement time t. The measurement rate describes, how often is a value produced and measurement time describes, how long should the data acquiring code be run. The data acquisition node runs the code for reading the sensors every $1/f$ seconds for t seconds. This produces an array of values from which the average represents the resulting value that is then going to be processed further. The f and t must be selected that $\frac{1}{f} > t$.

The data acquisition node, when collecting the values for the array also records the time stamps. The resulting time stamp assigned to the measurement value is the average of these time stamps. It is important to mention, that the in order for this to work, the control computers of the sensor nodes have to be synchronised, which is often not an easy tasks. More on the time synchronisation in section 4.1.4

The data acquisition node also keeps record about the number of measurement taken to produce one value and the standard deviation of the values, which indicate, whether there were some outliers that might affect the resulting value. These values indicate the quality of that measurement value.

When the value, number of measurements, standard deviation of the measurements and time stamps are assembled, the data acquisition node compacts the message into a custom defined ROS Message and publishes it into a corresponding topic.

4.1.2 Messages

As mentioned before, the structure of the message is fully customizable. The message definition is stored in the *.msg* file, which is a simple text file that contains the structure of a message type. For example the definition of the temperature message is shown in listing 2

As we can see, there are two fields: *header* which is is of type *igrosHeader*

Listing 2: Defining .msg file for temperature message

```
igrosHeader header
float64 val
```

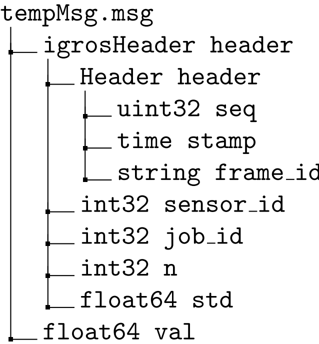
and *val*, which is a 64 bit float that holds the actual measurment value produced by the sensor. The type *igrosHeader* is a custom type, which is defined in its own *.msg* file (listing 3). The *header* is a whole message encapsulated in another message. The messages for the other quantities are designed analogously.

Listing 3: Defining .msg file for igrosHeader message

```
Header header
int32 sensor_id
int32 job_id
int32 n
float64 std
```

The igrosHeader consists of five fields: from *sensor_id* which holds information about the sensor that made the measurement, from *job_id* which holds the information about the job (section 4.3.2) for which the measurement was made and from *header*, which is of pre-defined type from ROS Package *std_msgs*. *n* and *std* are the parameters of the quality of measurement and describe the number of measurements from which this value was computed and the standard deviation of those measurements.

The *std_msgs/Header* message consisting from *timestamp* which is of type *time* which is a two-integer that consist of seconds and nanoseconds. The other two fields of the Header type are *frame_id* which is a string identifies spatial reference system and *seq* which is a unsigned 32 bit number representing consecutively increasing ID of a message. Following schematics represents the whole message structure:



4.1.3 Data storage node

This node basically subscribes to all topics and stores the data into a database However, storing each value separately as it comes, can cause problems at higher measurement rates, since the connection to the database has to be built up from scratch every time. We have implemented a class *DataBuffer* which consists of two buffers. The incoming messages are stored into one of the buffers until it's full. When the buffer is full, the incoming data get stored into the other buffer meanwhile the messages in the full buffer get decomposed and are stored in one bulk operation into the database. This way the node doesn't have to connect to the database so often. The length of the buffers can be set via parameter to the constructor. There is a *DataBuffer* object for each topic. The *DataBuffer* objects are initialized at start of the *DataStorageNode* and the length of the buffer can be set by the user via program arguments resp. via parameter server.

4.1.4 Time synchronization

Since the controlling computers of the sensor nodes do not posses hardware clock, the system time is into a file on the hard drive at shut down and picked up again at boot up. This can lead to severe time desynchronization between the computers, which poses a problem, because these computers assign the time stamp to the measurements. Therefore the central computer is used as a NTP (Network Time Protocol) server for the sensor node computers, which synchronize their time with the time of the central computer. The central

computer might or might not be connected to internet, which does not matter, because the central computer should have hardware clock that keep running even if the computer is turned off.

4.2 Django Web Application

Assuming that the user has already correctly set up database, the sensor network can be operated with the ROS package only without any programming skills. However, that would be a very tedious work. The user has to make a lot of decisions in a variety of processes on different components, for example, setting measuring rates of sensors must be done on the control computers. A control interface provides a way to neatly set all these parameters in a graphical user interface in one place. Figure 7 graphically represents the usage levels of ROS and relevant skills (yellow and green boxes). The blue box represents the usage level of the IGROS and required skills to use it.

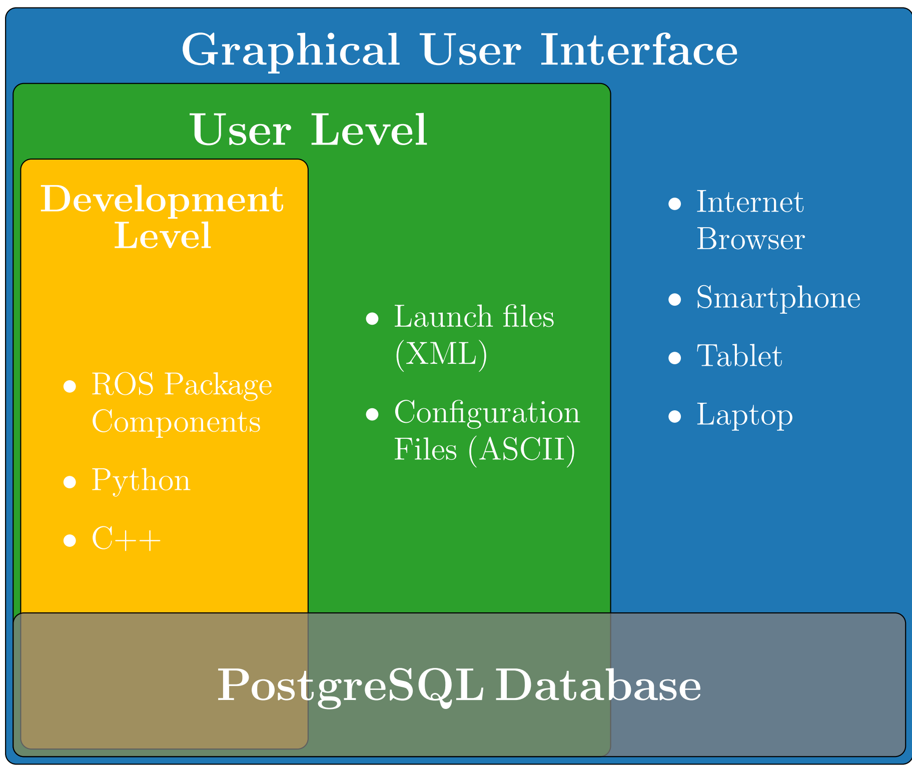


Figure 7: Usage levels of ROS resp. IGROS

The control interface could optionally be accessible over the internet so that

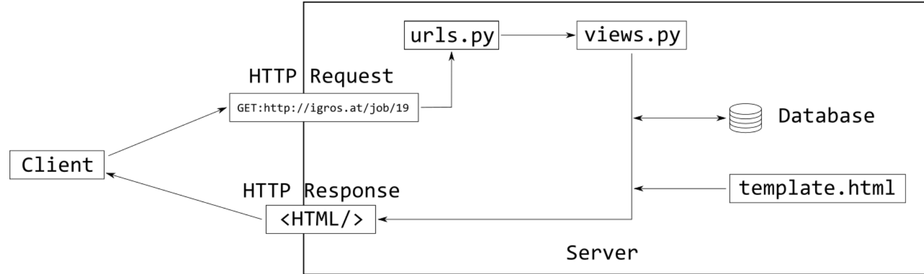


Figure 8: Handling of a HTTP Request in Django

a user does not have to be physically present at the computer that is running the control interface which is the central computer of the sensor network. The easiest way to make a graphical control interface remotely accessible is by creating some web application that could be accessed from any device on any operating system in the internet browser.

The control interface must have access to measured data. The most general way to ensure this is by creating a database. The control interface should be able to manage the database.

4.2.1 Django

Django is a free, open-source high-level python web framework that encourages rapid development. [5]. It is based on the model-template-view paradigm. It also provides excellent options to manage the database. In following sections the main components of Django are discussed.

Views
A *view* in Django terminology is a piece of code that generates the HTTP (HyperText Transfer Protocol) response (Figure 8). In its simple form it is a python function, in more complex and general form it is a python class. This class is a subclass of base *View* class that has a method *as_view()* which, when called, generates a HTML (HyperText Markup Language) file. The view, (either as function or the *as_view()* method) gets called when the server receives a HTTP request, requesting sepcific URL (Uniform Resource Locator). The server looks into the django's *urls.py* file which contains the URL patterns and resolves the URL to a view that gets subsequently called. The view executes its code, that generates a HTTP response and either returns the response to the client or redirects the HTTP request to another view.

Templates
If the view returns a HTML file, it is created from a *template*. A template is a HTML file that contains the structure of a page without specific content with place holders for the actual content. The *view* gets the content from elsewhere, most usually from the database. There is a special Django template language that adds more functionality to the content place holders like *for loops*, *while loops*, *if-statements* etc. For example: a user profile page contains a photo, name and date of birth. In the *template* there would be HTML code that contains all the required HTML structures, like the <*img*> tag, a table for nice display of the name and date of birth. The *src* attribute of the <*img*> tag, which specifies the path to the profile picture on the server file system, would contain a place holder, that gets filled in by the *view*.

Listing 2: Example Django Template Language

```
<img src="{{profile.img_path}}">
```

The listing 2 shows an example of the django template language in a HTML file. The double curly braces denote, that the value of the variable *profile.img_path* should be filled in between the braces. The content is parsed to the *template* by the *view* as a dictionary. In the template, one can access the values of variables through this dictionary. The values can be of any type, dictionary included. So in the example, in the content dictionary there is a variable *profile*, which is a dictionary itself, and contains key *img_path*. The value of this key will get printed into the HTML file that will be parsed by the server to the browser.

The *templates* don't necessarily have to be whole pages. It can be only parts of a complete web page. One can link the templates together so there could be templates for separate components, like menus, panels, tables etc. that will then make up a whole page. This makes the code extremely easy to maintain.

Models
When generating a HTML document, the *view* must fill in the gaps in the content. The content comes usually from the database. Django obtains data from database via *models*. Every table in the database has a django *model*, which is a python class, that abstracts a table entry. The class consists from fields that correspond to table attributes and the instances of this *model class* have the values of their fields filled by the values from the database entry. So, essentially we are creating a *model class* that can represent a database table. *Django* reverses this line of thinking, because it creates a database table based on a class. This way, *Django* can operate with many different DBMSs (Database Management System), because the model always stays the same. The fields of the *model classes* are predefined objects, (*IntegerField, FloatField, CharField, Foreign key* etc.) so that each binding to a specific DBMS can create correct structures in the database.

When the database is queried by the *view* for *template* content, *Django* returns a *QuerySet* object. This is a very useful data structure that consists of the instances of the model classes. This data structures allows operations like filtering, slicing, sorting etc.

Listing 4: Example of a model class for measured value

```
class Values(models.Model):
  timestamp = models.DateTimeField(primary_key = True)
  val = models.FloatField()
  property = models.ForeignKey(Property, on_delete=models.
      CASCADE)
  job = models.ForeignKey(Job, on_delete=models.CASCADE)
  sensor = models.ForeignKey(Sensor, on_delete=models.
      CASCADE)
```

4.2.2 Design/bootstrap

Since the control interface should be accessible from any device, the web pages have to be designed responsively. This poses a great challenge but luckily there are frameworks for responsive designs. One of the most commonly used framework is *Bootstrap*. *Bootstrap* is the most popular HTML, CSS (Cascade Styling Sheet), and JavaScript front-end framework for developing responsive, mobile-first websites. [6] It is free to use and is open-source.

Front-end components

Bootstrap provides a large number of pre-programmed components, like buttons, menus, user input dialogues, navigation bars or paginations etc. These can be easily styled using CSS. The functionality of the elements is provided by the library but can be easily extended with JavaScript. All the components will also responsively transform into different design depending on the screen size. For example menus look different on a cell phone than on a Ultra HD screen, even though the developer has declared it in code only once. o

Bootstrap templates

The bootstrap's modularity allows the developer community to encapsulate complete web designs into so called templates (not to confuse with Django templates). There is countless number of templates available today, some of the templates are free to use. We have chosen the SB-Admin[1] template for our control interface as it contains pre-programmed elements suitable for the needs of a control interface of a sensor network.

[1] Live preview: https://startbootstrap.com/previews/sb-admin/

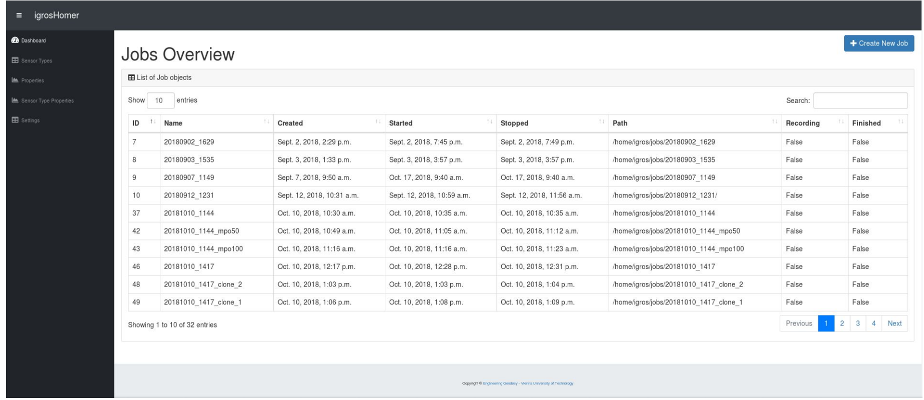


Figure 9: Home view user interface

4.2.3 Home View

This is the first page (figure 9) that user will see after loading the interface. There are two main elements on the page: the side panel which contains links to the long-term settings and a list of jobs. There is also a header panel and a footer.

Job selection
The job table is a list of all jobs in the database. If a job is currently acquiring measurements, the job is highlighted. One can sort the jobs by any attribute, per default they are sorted by their ID. In the search field, the jobs can be filtered by entering text. The search occurs in all attributes so it is not necessary to specify, for which attribute we are looking for.

There are also listing options. On top of the table there is a input field where user can specify how many entries are displayed on one page. The pages can be selected in the bottom-right corner of the table. Above the table section there is a button to create a new job.

The job table is part of the bootstrap template and is therefore fully implemented in JavaScript, so it is not necessary to send any requests on the server when modifying the table content by searching or sorting, so there are no page reloads necessary.

Side panel
Under long-term settings one can understand configuration of the *physical quantities* and *sensor types* (section 4.3.5) and their relations. The user can edit or delete currently available entries or add new ones. There are also settings that

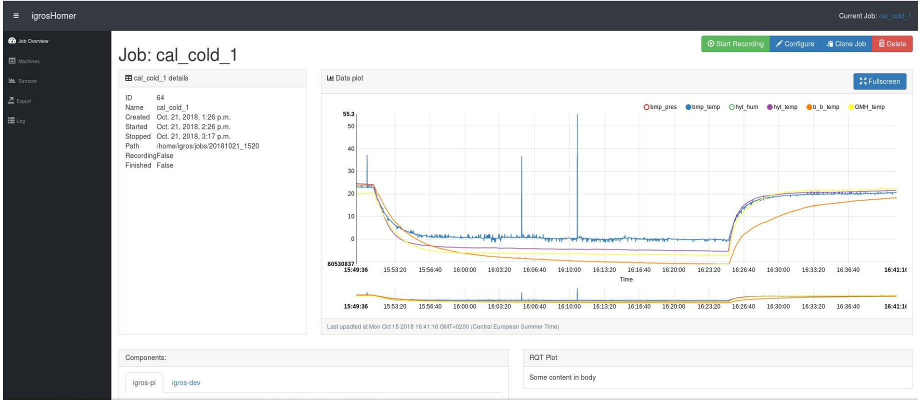


Figure 10: Job view user interface

control the data buffer length (section 4.1.1) and displaying parameters for the chart in Job view (section 4.2.4)

4.2.4 Job View

This page gets loaded when the user selects a job. There is again the side panel that contains job related settings, and the main area, where that consists of four *cards*. In the top right corner there is an operation panel that consists of buttons for managing the job. There is also the header panel and a copyright footer.

Data viewer

The data chart is created by a JavaScript library *nv3d* [2]. It consists of two set of axes. The bottom axes are used to select a specific time interval that the user wants to inspect closer. Defaultly the whole timespan is shown. The user can make the selection either by drag and drop or by moving the sliders to desired locations.

The top axes display selected data. There is a time series for each sensor and quantity. Unfortunately stackable y-axes (e.g. for each quantity) were not implemented yet, but it is planned in the future. This causes a problem when the multiple quantities have been recorded. For example, when the user has measured temperature in range from 10°C to 20°C and atmospheric pressure typically in range around 1000 millibars, the temperature series seems completely flat. To enable a user to inspect the temperature series as well one can turn the pressure series off. The chart gets re-drawn with the optimal scaling

[2]Live preview:http://nvd3.org/examples/lineWithFocus.html

of the y-axis for the temperature series. One can turn any series on or off by clicking the corresponding legend entry in top-right corner. The time series are per default all turned on.

The chart has also a inspection cursor so that user can hover over the chart to inspect the exact values of a series to specific time stamp.

There is also a option to switch the chart into full screen mode, which is not very well implemented yet, as it simply stretches the chart area over the whole viewport, although the interactivity of the chart is still functional.

The data for the chart are queried from the database. Django returns the data in form of a QuerySet. This QuerySet gets serialized to JSON and gets printed into the HTML file that the server generates and parses to the browser. However, if the there is too much of data requested from the database, it can lead to long load times of the page. Therefore there is a user-settable limit for the maximal length of query set.

Job info

Taking a look at figure 10, in the top left card, there are the general data about the job displayed. It displays the values of the attributes of the job in the database.

Component info

In the bottom left part there are tabular information about the associated computers and sensors. It is based on bootstrap *nav-tabs*. There is a tab for every computer associated with the job. Under the tab selection there are computer's attributes from the database tabularly listed. Underneath this table there is a list of sensors associated with that computer and their attributes. In future development is planned, that clicking the entries in these tables will take the user to configuration of these components.

RQT Graph

In the bottom right is a empty card. This card should contain the RQT Graph of the ROS network. There is a possibility to export these graphs to SVG in the RQT GUI but the automatization of this process has not been implemented yet. Further development should make the graph interactive so that the user could see the messages upon hovering over a topic in the graph.

Sidepanel

On the side panel there are links that take the user to the configuration of the components, that are associated with the job.

There is also a link to the export page. The user can export the job data for further processing. There are some options the user can filter the exported data, e.g. specific time range, or only some sensors or quantities etc. The user can also specify in which format the data should be exported. Currently implemented export format is CSV with various delimiters, but there are more formats planned e.g. MS Excel *..xlsx*, *.txt*, *.pandas* dataframes, *numpy* arrays, *.mat* for MATLAB scripts.

Job control
On the right side above the data chart card, there is a panel consisting of four buttons.

The first green button starts the measuring session. After the button gets clicked the server collects the information about job and gets associated machines and sensors. With these informations a ROS launch file is generated. Then the server starts a *roslaunch* sub-process that launches the generated launch file. When the data acquisition is running, the button changes and is used for stopping the process. When clicked, the *roslaunch* sub-process gets simply killed.

The next button takes user to the job configuration, where the job attributes can be edited.

When having a job with a lot of computers and sensors, setting it up can be a tedious work. We have therefore implemented a *job cloning* which by simply clicking the clone button creates a new job with different name but with same component configuration.

Last button on the panel deletes the job. This deletes not only the job but also the associated components in the database.

4.3 Database

The two main functions of the database is to store the measured values and the sensor network configuration. Since the control interface is a web application based on *Django*, which is a python package, we were looking for relational database management system that works well with Django/Python. We have decided for PostgreSQL as it meets all of the requirements and the author had already previous experience with PostgreSQL. Figure 11 shows the implemented tables and their relations.

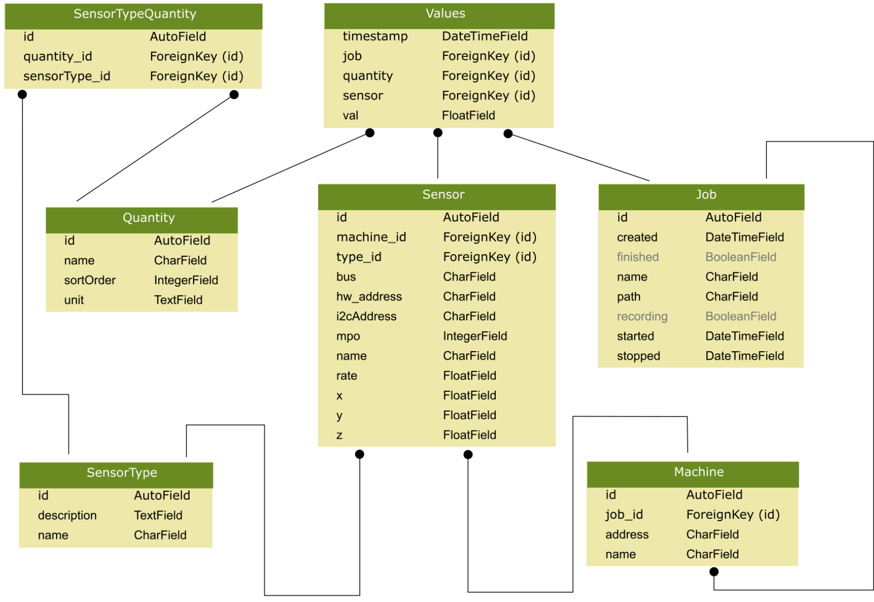


Figure 11: Implemented database model

4.3.1 Values

This table is the most important as it contains the measured values. The primary key of this table is the *timestamp*. Every value is then associated with a *job*, physical *quantity* and *sensor* that made the measurement. The measured value is available in the *val* attribute with two quality parameters n which describes, how many measurements were made by the sensor to provide this value. The *std* attribute is the standard deviation of the set of measurements from which the value was computed from.

4.3.2 Jobs

A job organises related data into a directory-like structures and makes adding meta-information possible. Every job in the database has its own numerical unique id. The *name* attribute allows the user to assign the job a name so that the user doesn't have to remember the job id. The *path* attribute describes, where on the file system the job related files (data exports, launch files etc.) are stored. The *DateTimeFields: created, started, stopped* store dates and times of creation, begin of measurement session and its end. The *finished* attribute marks whether the job has been finished, meaning that these won't be no more measurement sessions carried out, but the job is not ready to be deleted yet.

4.3.3 Machines

This table represents the computers in the network. The primary key is an integer *id*. Every machine is associated with a *job* over the job id. The other two attributes store the IP *address* of the computer and the *name* of the computer. It is important to mention here, that the *machine* is here a bit more abstract concept than one would intuitively assume. Even though entries in this table describe a computer, one has to realize, that a *machine* is associated with a *job*, which means that there could be many entries in this table describing the same physical computer. However these entries will differ in the *job_id*. Every job has its own set of machines, even though they describe the same physical computers.

4.3.4 Sensors

The entries in this table describe sensors and their configuration. Since the same physical sensors can be used in multiple jobs, the level of abstraction here is same as by the computers in the *machines* table. Every entry in this table is therefore associated with one machine and the machine is then associated with the job. This way is the *job* - *sensor* relationship established. The*machine_i*d attribute associates a sensor with a computer.

The physical connection of the sensor to the computer must be described in the database as well. For this purpose there are attributes *bus*, *hw_address* and *i2cAddress*. The *bus* attribute has two possible values: i^2c or *USB* since these two types of the interfaces have been used so far. If the sensor is connected over i^2c bus, then the i^2c address has to be additionally specified as well. The *hw_address* attribute describes the system address of the bus interface which is typically something like */dev/ttyUSB0* or */dev/i2c-1*.

Every entry in this table is assigned a unique number *id* and user can specify a *name* of the sensor, which is easier to remember. To allow spatial referencing of the data, the sensor can be provided with 3D coordinates triplet in projects spatial reference frame.

Every sensor is also provided with its sensor type (section 4.3.5).

4.3.5 Sensor Type

This table describes the properties of the physical sensors that do not vary with job or its purpose in a project. It also establishes link between the physical quantities the sensor measures.

The entries have their unique *id*. The user can also provide a *name* for the sensor type and more detailed *description* of the sensor type like manufacturer, type, product number etc.

4.3.6 Quantity

This table describes quantities provided by sensors. The entries use unique *id* as primary key and allows user to set the quantity *name* (to give the user the freedom to use full name of the quantity or to use its abbreviation etc.) and its *unit*. The attribute *sortOrder* describes listing or displaying priority in the control interface. Lower the number, higher the priority.

4.3.7 SensorTypeQuantity

This table is fully abstract one as it's purpose is to link the sensor type with the quantities it measures.

5 Electrical Sensor and Calibration

An electrical sensor is a device that converts a general physical signal into an electrical signal. This electrical signal can be further processed and digitalized, so that the signal becomes computer comprehensible. However, the digital signal loses the physical context. The physical context of the digital signal is recovered by calibration. Calibration is a process of finding a characteristic curve of the sensor, which describes the functional relationship between variation of the input physical quantity and sensor output. [8] Figure 12 shows the the workflow of an electrical sensor.

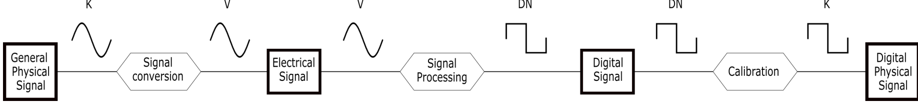


Figure 12: Process of data acquisition and calibration

5.1 Signal conversion

The examined object provides a general physical signal. This signal is a continuous function of time and has units of some physical quantity, e.g. meter, kelvin etc. A sensor converts this signal to a continuous electrical signal. The

electrical signal has a unit of electrical quantity, such as volt, ampere, or hertz etc. The output physical quantity depends on the construction of the sensor as well as on the purpose for which it was constructed. There are three main types of electrical sensors [7]

- Resistive sensors
- Inductive sensors
- Capacitative sensors

which produce the electrical signal based on variation of the respective electrical quantities. This does not mean, that the sensor outputs these quantities. In most cases the sensors output related voltage change.

Resistive sensors produce their output signal based on a variation of the resistance. The variation of the resistance is caused by the variance of the input physical signal. An example of such sensor is a resistive thermometer.

Central component of an inductive sensor is a coil or coils with variable inductance. The resulting signal is produced either by variation of relative permeability of the coil core, which can be moved along the coils axis, or, if using two coils, by variation of the distance between the two coils. These sensors are used for example for measurement of angles.

Capacitative sensors produce their signal based on variation of capacitance of a capacitor. The variations can be caused by moving the capacitor plates away from each other, by moving the plates in their own planes (the vertical distance between the plates stays the same, but planes are misaligned), or by changing the dielectric between the two plates by introducing an object in between the plates. An example of an capacitative sensors is a touch displays of smartphones.

5.2 Signal processing

A sensor provides a continuous electrical signal. This analogous signal is usually very week and has to be amplified to a wider voltage range. However the signal is typically afflicted by a measurement noise. As the signal gets amplified, the noise is amplified as well. The amplification is described by an amplification function. The parameters of the amplification function are either fixed by the hardware, or can set by the users by, for example, setting trimmer potentiometers.

The signal can be filtered by methods of signal processing. For example, typical high frequency noise can be filtered out by low pass filters like moving average or moving median.

Once the signal is sufficiently amplified, it is converted to a digital signal. The conversion happens in an analogue to digital converter (A/D converter) which samples the values of the analogous signal in discrete timestamps (Figure 13).

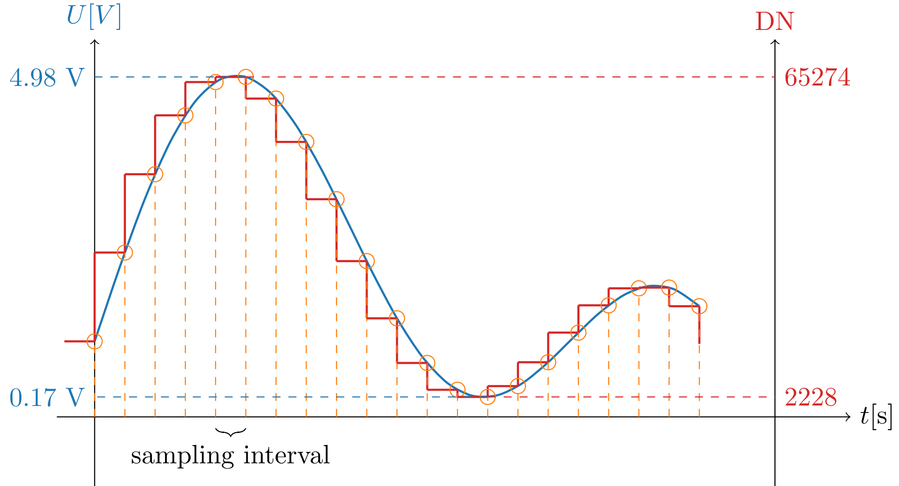


Figure 13: Conversion of analogous signal to digital

A/D converters have maximal sampling frequency which describes how often can the analogous signal be sampled. The A/D converter maps the value of the signal at a discrete time stamp to a digital number. It has a given digital number range, typically from zero to some power of 2 minus one, e.g. 0 - $2^{16} - 1 = 65535$ = 16 bit range. The digital number range defines, how fine can the codomain of the analogous signal be sampled.

The digital number itself doesn't have any physical context. The physical context must be established by specifying the actual voltage range of the analogue signal, which is then linearly mapped to the digital number range. For example 0 - 5 Volts range is mapped to digital number range of 16 bits, meaning, that 0 V maps to 0 DN and 2.5 V to $\frac{2^{16}-1}{2} = 32767$ DN and 5 V to $2^{16} - 1 = 65535$ DN giving us resolution of $\frac{5}{2^{16}-1} = 76.293\mu V$. This way a DN can be considered as having a physical context.

5.3 Calibration

Now, when the physical context of the digital number has been restored, if only in context of a electrical quantity, it is time to find the relationship between the digital numbers and the originally measured physical quantity. This rela-

tionship is described by a characteristic curve of the sensor which is a function. This function takes the electrical quantity provided by the sensor as input and outputs the desired physical quantity. Since the characteristic curve is usually modelled as a linear function, it has following form:

$$p(E) = kE + d \tag{1}$$

where p is the physical quantity, E the electric quantity and k and d the linear parameters. [8]

The characteristic curve of a sensor is sensor-specific. Since the sensor is composed of many components, replacing one of the components should result in different characteristic curve.

The parameters of the line are determined in a calibration measurement, where a sensor with unknown characteristic line measures in same conditions as a sensor with known characteristic line. Such calibration measurement was done for this thesis and is described in section 6

6 Calibration Experiment

6.1 Instruments

6.1.1 Sensors

We have built three sensor node prototypes. In this section are described the components of the node as well as their configuration.

The probe (fig. 17(a)) is a Pt1000 resistance thermometer. It consists of a platinum wire wrapped around a ceramic core covered in a protective steel housing. Since platinum is very expensive, the wire is very fine. The reason one uses a fine platinum wire in resistance temperature probes is, that the resistance-temperature relationship is very accurately known.[9] Pt1000 means, that at 0°C, the probe provides resistance of 1000 Ω [10]. There are two wires going into the probe that are compacted into one, about 150 cm long cable. The ends are stripped of the protective isolation so that the probe can be connected or soldered to a sensor circuit.

In order to amplify the signals from the probes, which are very weak, there were *Pollin Electornic GmbH PT1000*[3] amplifiers used. (fig. 14) These amplifiers were set to provide analogue voltage output from 0 V to 5 V. There are two trimmer potentiometers on the amplifier that control the parameters of the

[3] Data sheet: https://www.pollin.de/productdownloads/D810144B.PDF

amplifying function. Since the function is linear, the potentiometers control the slope of the line and its offset. The voltage span of the amplifier was mapped to temperature span from approximately -20°C to +70°C [4]

Figure 14: Used amplifier Pollin Pt1000 with output from 0-5 V

In subsequent digitalisation of the signal were deployed the *Texas Instruments ADS1115*[5] A/D converters, (figure 15), which have i^2c interface and produce 15 bit digital numbers. The digital number range is mapped to voltage range of the amplifier, which has the maximal output voltage of 5 Volts.

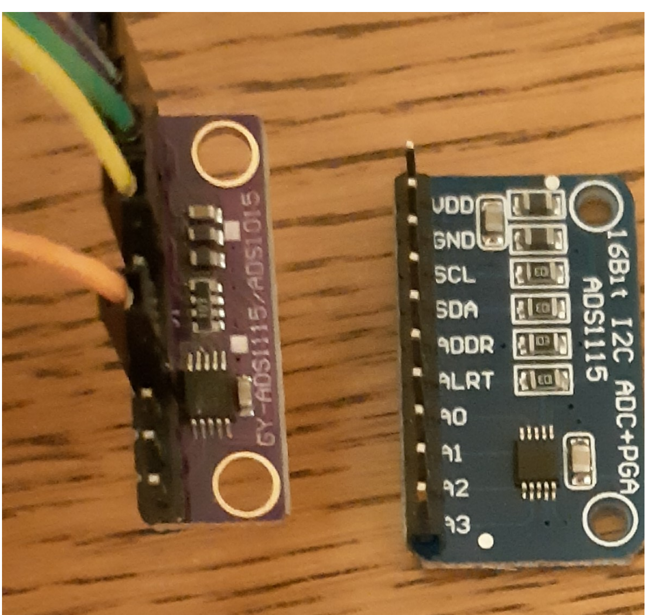


Figure 15: Used A/D converter TI ADS1115 with i^2c interface and 15 bit output

[4]Configuration on bottom of page 6: https://www.neuhold-elektronik.at/datenblatt/N7975.pdf

[5]Data sheet:http://www.ti.com/lit/ds/symlink/ads1115.pdf

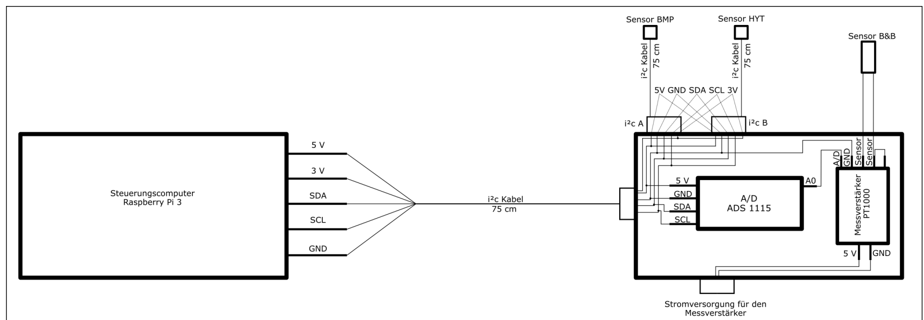


Figure 16: Circuit diagram of the protective box housing the A/D converters and amplifiers

Some of the prototypes were furnished with a small protective box, providing protection against physical damage. Figure 16 shows schematically how the components are connected together inside the protective housing and the boxes' inputs and outputs, whereas figure 17 show the realisation of the schema. The probe cable goes into the box which makes the placement of the probe easier and provides a cover for the components that are susceptible to physical damage. The protective box also provides longer wiring for other meteorological sensors. Each prototype was connected to a separate control computer. The numerical suffix in the prototypes name designates, which of the control computers was the prototype connected to.

Sensor	unit	range min		range max		covered
GMH 3750	°C	-20		70		yes
B+B 40	V (DN)	0 V	0 DN	5 V	2^{15} DN	yes
B+B 50	V (DN)	0 V	0 DN	5 V	2^{15} DN	yes
B+B 60	V (DN)	0 V	0 DN	5 V	2^{15} DN	no

Table 1: Used sensors and their parameters

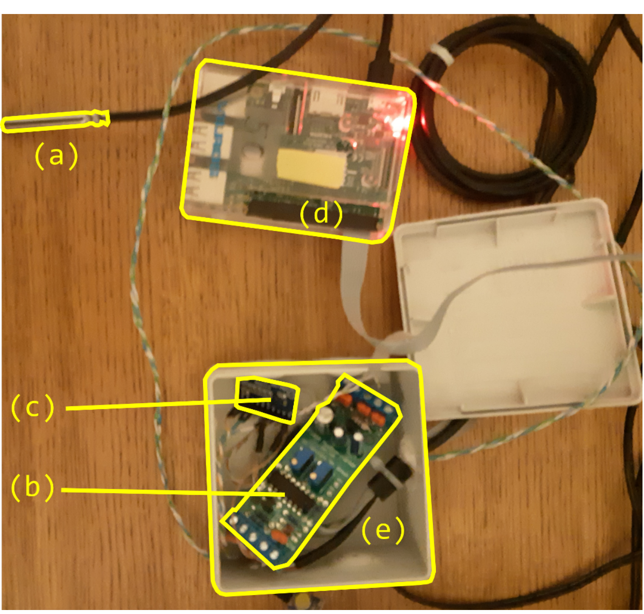


Figure 17: The complete B+B sensor prototype, where (a) is the probe, (b) the amplifier,(c) A/D converter, (d) control computer and (e) protective box

For the controlling computers we have used Raspberry Pi Model 3, [6] as our institute posses a few of them. The three computers were labelled 40, 50 and 60. Since every prototype was connected to separate computer, the prototypes were assigned names in accordance with their controlling computes.

As reference sensor we used a Greisinger GMH 3750 sensor with a Pt100 probe with range from -20°C to 70°C with accuracy better than 0.07 °C [7] (figure 18)

[6]Home page: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/
[7]https://www.greisinger.de/files/upload/en/produkte/kat/5.pdf

Figure 18: Reference Sensor Greisinger GMH 3750

6.1.2 Climatic chamber

The reason why we needed a climatic chamber is, that the reference sensor probe and the prototype probes have different masses, which means, that the probes have different heat capacities. This causes the probes to warm up and cool down at different rates. That means that reference values do not correspond to the measured values during temperature change and making data from these intervals unusable for the calibration. A climatic chamber enables us to create a stable environment for long enough periods of time for the probes to acclimatise. The other considerable advantage is, that one can set temperatures above, as well as bellow freezing point which could not be achieved in any day-to-day appliance, such as fridge or oven (which aren't capable of creating a stable enough environments for such experiment anyway). This way, we can acquire all needed data without having to move the probes between two appliances.

For the experiment we used a *Weiss WK3*[8] climatic chamber (fig. 19) that is in possession of the Federal Office of Metrology and Surveying (BEV) which kindly allowed us to use their climatic chamber for our experiment.

6.2 Experiment

Since the sensors are to be deployed in the measurement lab of our institute, where the atmospheric conditions are fairly stable we have decided not to measure on the whole measurement range of the reference sensor but only the range from -10 to 40 °C with 6 stabilized measurement points (every 10 degrees). The measurements outside this range should be considered less accurate because of the extrapolation, since the characteristic line is defined inside the interval -10 to 40 °C. As mentioned before, the prototypes are to be deployed in a very

[8] http://www.weissfr.com/fr/download/Enceinte-climatique-simulation-environnement-Weiss-WK3-0-anglais.pdf

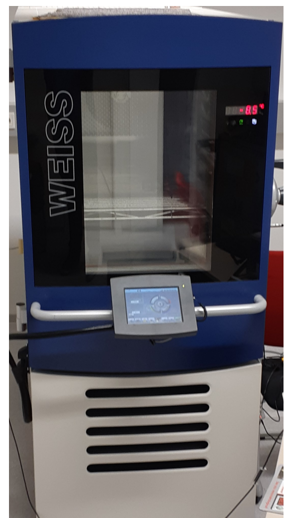


Figure 19: Climatic Chamber Weiss WK3 in BEV

stable environment with temperature around 20 degrees which lies well inside the calibration range.

This means, that after setting the temperature in the climatic chamber, it takes time for the chamber to heat-up or cool down to a given temperature and stabilize. Afterwards the sensors have to acclimatise on the temperature as well. Each measurement point has taken between 30-40 minutes to acquire with at least 10 minutes of stable data. The measurement rate of all sensors was set to 1 Hz.

The goals of the experiment were:

1. Find parameters of the characteristic lines of the sensors
2. Test, whether we can use the same calibration parameters for two different amplifiers and A/D converters of the same type.
3. Examine the effect of the temperature on the hardware parts of the prototypes.

Because of the second goal, we needed to carry out the experiment twice. The two measurements were stored to separate jobs. The data of the first measurement is contained in job 124 and the second in job 126. In job 124, sensor B+B 60 has been deployed with different A/D converter, than in job 126. Moreover, to address the third goal, in job 124 the A/D converters and amplifiers of sensors B+B 40 and B+B 50 were placed inside the chamber, whereas in job 126 only the probes were kept inside the chamber.

The experiment (fig. 20) took place on December 19 2019 approx. from 8:30 to 15:00 in the building of the Federal Office of Metrology and Surveying (BEV) in Arltgasse 35 A-1160 Vienna.

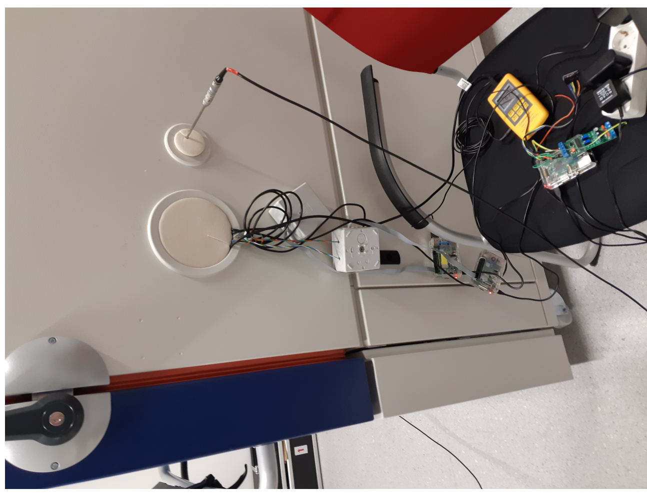


Figure 20: Ongoing experiment. The climatic chamber has several inlets, filled by cylindrical foam lids, that allow the probes to be inserted.

6.3 Evaluation

6.3.1 Evaluation module

We have written a separate *python* module for evaluation of the measured data. This module contains data-structures and methods for fitting, plotting and filtering. It also contains methods for performing various statistical operations. In this module there are two basic data-structures:

- Series
- Calibrator

as well as several helper functions.

6.3.2 Series

The *Series* represent a time series of a quantity. The elements of a time series are data points, which consist of a timestamp and a value. The data in the

series object are chronologically ordered. It proved advantageous to implement this data structure as two lists:

- Timestamps, list of *datetime objects*
- Values, list of *floats*

rather than as single list of objects with two fields. This implementation simplifies the access to values by timestamps via the *__getitem__()* method. We have implemented the method ourselves, so that it is possible to access values between two timestamps. If the exact time stamp that is being requested does not exist in the series, the method locates the nearest two data points and returns the linearly interpolated value to the given time stamp.

The data structure contains methods for dumping the data into a *csv* file on the file system, that preserves the chronological sorting. These files can be then very quickly read from the file system without the sorting procedure, which might take some time and therefore slow down the evaluation.

When a new value is introduced into the series, it is placed into the correct position in the series, so that the object stays chronologically consistent.

The *Sereis* objects have properties that can be accessed as attributes, such as:

- number of values
- maximum value
- minimum value
- mean
- standard deviation
- variance
- autocorrelation

These properties can be easily printed to the screen by single statement by calling the *print_stat* function. There is also a function for computing the histogram. *Series* objects also have several filtering functions like:

- median filter with settable size of an uniform kernel
- moving average with settable size of an uniform kernel

- filter which selects values by user defined logical condition

The filtering functions can be chained into one statement. There is also a function that selects values based on time stamps. This differs from the __getitem__() method, since the __getitem__() method can take only a single time stamp and interpolates the values if needed, which the other function does not.

One can also easily compute the temporal derivative by calling the *first_derivative* function. There is also a function that multiples all values of the series by a polynomial. This is useful e.g. when applying the calculated calibration parameters to the uncalibrated series.

There are also two plotting functions. The *plot* function simply plots the series. The user can also specify, which axes object (figure) should the *Series* object be plotted into. The other plotting function plots a histogram of the values of the series.

6.3.3 Calibrator

The calibrator class is the class in which the fitting takes place. Basically it consists of two Series object, one contains the reference values and the other the observed values. The constructor of this class calls a function, that computes a fitting polynomial and stores the computed parameters as well as quality parameters as attributes of the object. The order of the fitting polynomial is based on an optional argument, whose default value is one (a line). Other attributes of this class are specifying additional information about the two series. For example, one can set the name and colour of the two series, which is helpful while plotting, as the plotting functions automatically create legend entries, titles etc..

Apart from plotting functions there is the function *poly_fit* that computes the fitting polynomial. This function is called in the constructor, when the object is being initialized. The fitting procedure is described in detail in section 6.3.8. The method contains two encapsulated methods that are used only inside the *poly_fit*. One method computes the initial estimates for the parameters and the other performs a test and returns a boolean value whether one more iteration is needed or not. (Section 6.3.8)

All the other methods of the class are plotting methods. All of these methods have an argument that specifies, into which figure should be plotted. The methods are:

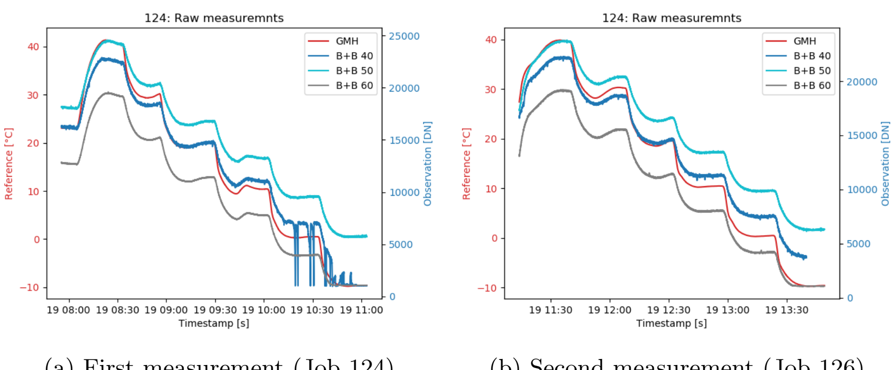
- *plot_series*, which plots the two series into one figure. The user can determine, whether the two series are plotted to scale, or whether both series should have separate ordinate with different scaling.

- *plot_ref_vs_obs*, which plots the values on a plot with reference on the abscissa and observations on the ordinate, since it should represent the relationship between the two. In this dimension takes the fitting place.
- *plot_fit*, which plots the same as the *plot_ref_vs_obs* and additionally the fitting polynomial
- *plot_residuals*, which plots the residuals series of the fit
- *plot_residuals_histogram*, which plots the histogram of the residuals.
- *histogram*, which plots the histogram of the both series into one plot.

6.3.4 Data formatting

Although the *igrosHomer* WebApp has an exporting function, it exports the data only to formats that aren't suitable for the evaluation module, since the module is not a part of the *igros* software package. One can use the *igrosHomer* to export the data into a *csv* file, which is then fairly simple to reformat to make the data usable for the evaluation module. The evaluation module also provides a function that reformats a simple tabular output from the *psql* command, which can be used to export the data directly from the database, to the format that is compatible with the evaluation module. Once the data have been successfully read into *Series* object, it can be saved on the file-system into a file, that is then easily read by the evaluation module and it's structures.

6.3.5 Raw data



(a) First measurement (Job 124) (b) Second measurement (Job 126)

Figure 21: Raw measurements. The blue-ish lines are the observed values and are related to the right axis, The red line represents the reference values and is related to the axis to the left.

Figure 21 display the raw measurements recorded during the experiment. Table 2 lists the variance parameters of the raw measurements and figure 22 represent the variance parameters graphically. At the first look it is clear, that in the first measurement (Job 124 , fig. 21a) the B+B 40 sensor has failed as the temperature approached zero degrees. One can suspect, that the failure was caused by the low temperature, because in the first measurement, all the B+B 40 sensor components (apart from the control computer) were placed inside the chamber. The temperature variation causes expanding and shrinking of the materials, so maybe one of the circuit board components might malfunction due to the rapid shrinking, not to mention, that this behaviour of the prototype has been observed on several occasions before. This prototype is the oldest one and has been transported several times in unsuitable cases during the development process before being furnished with a protective box. The other prototypes are much younger and have not been deployed as extensively as the original one. Nevertheless the data up to 10 degrees seem usable.

The age of the first prototype (B+B 40) is visible in the amount of variation of the series, which is caused by signal noise. To be able to compare the amount of noise, one has to compute a moving average of the series(kernel size 51) and subtract it from the original series (fig. 22). From the resulting series one can inspect the variation parameters of the series. Since the B+B 40 sensor exhibits obvious outliers at the end of the series, these values are filtered out as well, so that the resulting parameters are not distorted by them. The resulting values are listed in the table 2.

	Job 124			Job 126		
	B+B 40	B+B 50	B+B 60	B+B 40	B+B 50	B+B 60
number of observations	9975	10649	10667	8219	8736	8809
standard deviation [DN]	51.5809	30.8352	10.6180	53.0864	29.9175	25.0124

Table 2: Variance parameters of the series

From table 2 it is apparent that the variance parameters of the raw measurements are not significantly affected by the ambient temperature. The standard deviations of the sensors B+B 40 and B+B 50 are in both jobs very similar. for B+B 60, there is a significant change in the standard deviation between both measurement sessions. This is presumably caused by deployment of another A/D converter whereas the two other prototypes remained the same.

Interestingly, although all of the prototypes consist of components of the same type and are configured the same way, the amount of noise introduced by the hardware varies (compared to each other) significantly. When comparing the variances between the three sensors, one can see, that the B+B 40 introduces the most noise to the signal in both measurements. This might be caused by the inappropriate treatment of the hardware to which the sensor was subjected to before it was furnished with a protective housing.

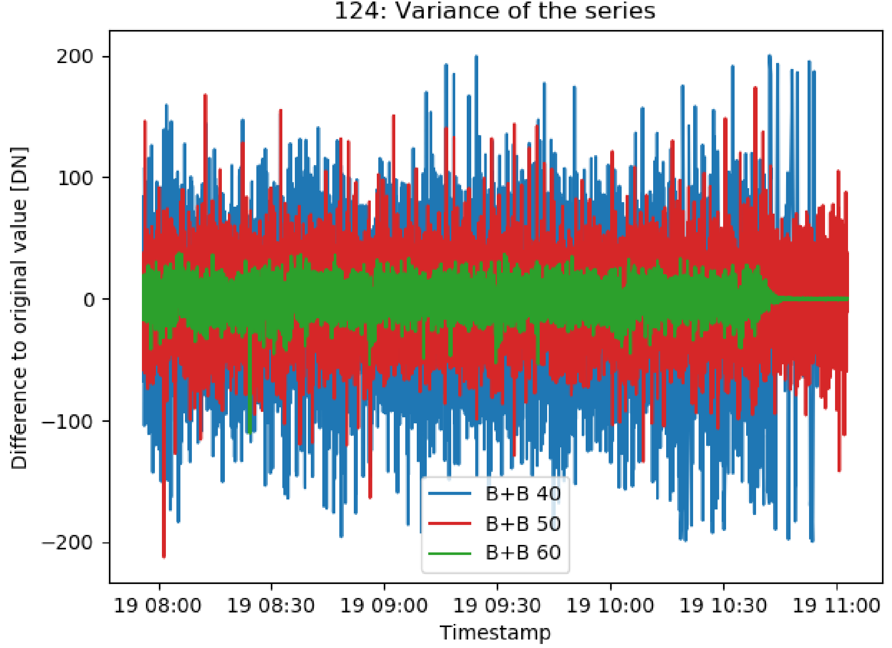
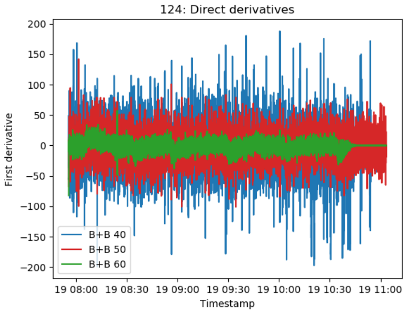


Figure 22: Job 124: Difference between the series that were filtered by moving average and the original series. Same procedure was used in job 126, where the resulting graph looks very similar

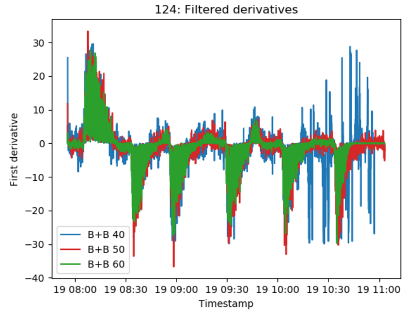
6.3.6 Filtering

The main idea of filtering is to extract those data points where the temperature was stable. The data from the dynamic parts of the measurements (from unstable intervals) deviate from a linear model that is formed by the data from the stable intervals.

The stable intervals can be found by examining the first temporal derivative. The first derivative represents the rate of change of a certain function. Based on the first derivative one can declare some threshold value, that separates the data set into stable intervals and unstable intervals. The values of the stable intervals are then used for the fitting.



(a) Derivatives of the unfiltered series



(b) Derivatives of the filtered series

Figure 23: Whereas it is hard to distinguish any low frequency changes in (a), where the derivative was taken from unfiltered series, deriving the series after low pass filtering (median filter with kernel size 51) in (b) we can see clear spikes marking the unstable intervals

However, the measured data were afflicted by high frequency measurement noise, which made direct computation of the first derivative useless. (fig. 23a). Therefore it was necessary to apply a low pass filter to the series to smooth it first, before computing the derivative. This was done by median filter with kernel size of 51.

Finding optimal kernel size of the low pass filter

Median filter is a low pass filter, that is well suited for eliminating high frequency noise and works similarly to the above mentioned moving average filter. Both filters work data-point-wise and compute either average or median of the surrounding values. The kernel size defines, how many neighbouring values participate on determining the new value for given data point. Is the kernel size too large, it 'rounds off' the series and on the other hand if the kernel size is too small it might not remove the noise completely. The optimal kernel size can be determined by displaying it's impact on the series. This can be done with the function *argplot* which plots the same function with different argument values into one plot (fig. 24). After analysing figure 24 it seemed, that the optimal result can be achieved with kernel size of 51. The advantage of the median filter is, that it is not as susceptible to outliers as the average filter.

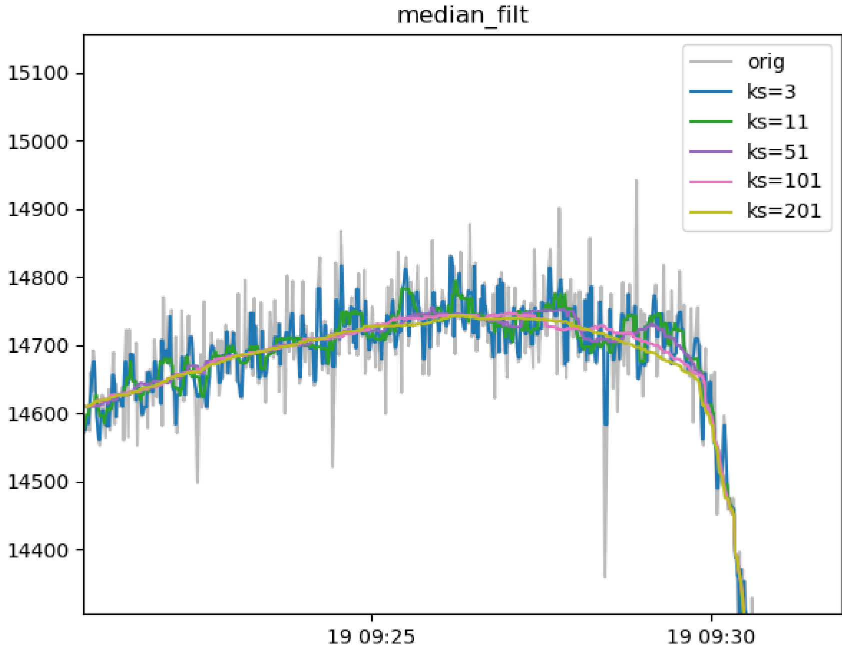


Figure 24: Median filter with different kernel sizes. Is the kernel size to small, the high frequent noise is not fully filtered out and if the kernel is too large, the series gets 'abraded'. The optimal kernel size this case is 51 (magenta line)

Once the series are smoothed, one can calculate the first derivative and perform the threshold filtering, based on the threshold values determined by the inspection of the first derivative (fig. 23b). The resulting series provides the timestamps of the stable intervals. Data from the original, raw, unfiltered series are then selected based on the stable-interval-timestamps and compacted to a new series, that is used in the fitting procedure (fig. 25).

After analysing the graphs of derivatives of the filtered series (23b), we have declared the intervals, where the first derivative was less than 0.1, as stable. Unfortunately for the B+B 40 sensor in job 124 this filtering was insufficient as the filtered values contained obviously unstable data at the end of the series, where the sensor has failed. To exclude these values, we have simply eliminated values under 8000 from the series. The filtered series with marked values in stable intervals can be seen in figure 25. These series were used in the fitting (fig. 25).

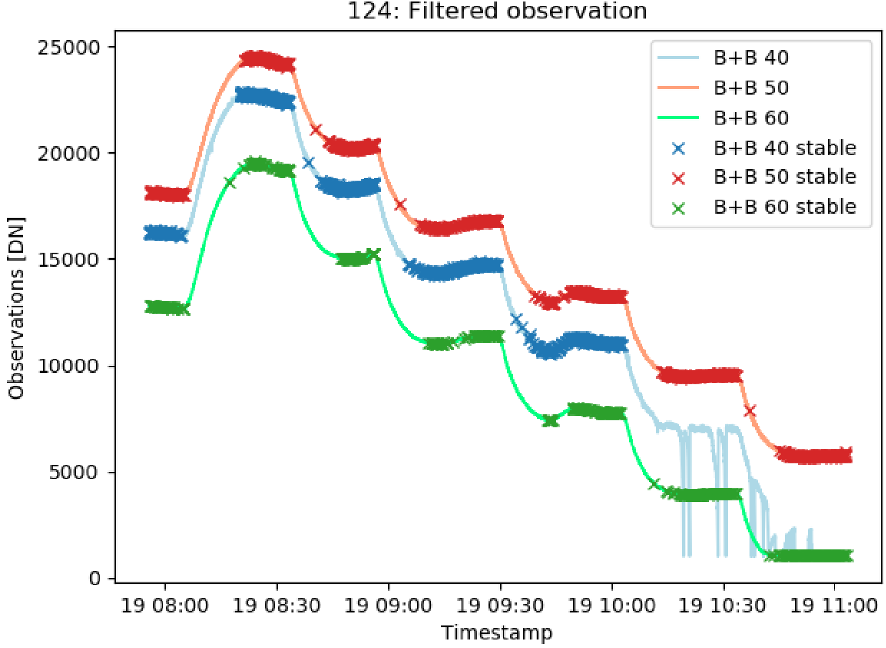
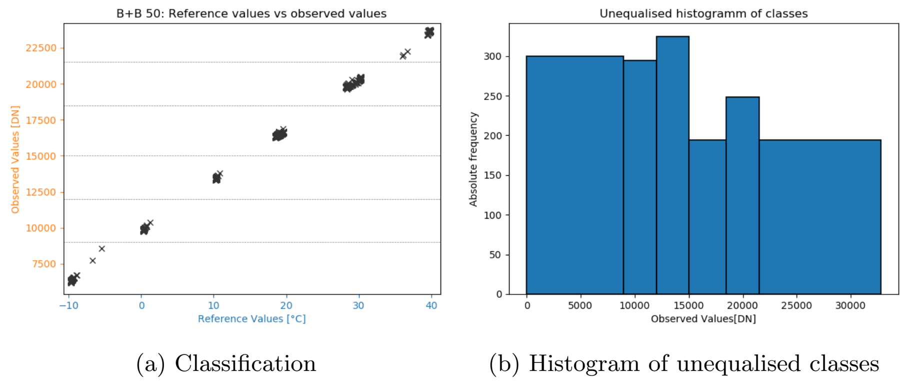


Figure 25: Filtered series with highlighted stable intervals

6.3.7 Classification and equalisation



(a) Classification (b) Histogram of unequalised classes

Figure 26: Classification and equalisation of the B+B 50 second measurement. In (a) we see the distinct classes and in (b) that the classes contain different number of measurements

The experiment design as well as the before mentioned filtering have caused,

	Job 124 [DN]			Job 126 [DN]		
MP	B+B 40	B+B 50	B+B 60	B+B 40	B+B 50	B+B 60
MP -10°C	-	0	0	0	0	0
MP 0°C	-	8000	2500	6000	9000	2500
MP 10°C	9000	11500	6000	10000	12000	6500
MP 20°C	13000	16000	10000	13500	15000	10000
MP 30°C	17000	19000	14000	16000	18500	13500
MP 40°C	21000	22500	17500	18500	21500	16500

Table 3: Manually set edges of the classes. The numbers denote the begin of the range of a measurement point (MP) in corresponding job and sensor.

that there are distinct groups of measurements as figure 26 shows. These groups have understandably formed around the six measurement points. However, the groups do not contain the same number of measurements that act as a weight of the measurements, which, if unaccounted for, can have distorting effect on the fitting. Therefore the data set was classified into six classes (four classes in case of the first measurement of the B+B 40) by dividing the codomains of each series. The edges of the classes were set manually. The values are listed in table 3. When the data set is classified, one has to equalise the classes, so that there is the same number of measurements in every class i.e. the classes have the same weight. This is done by removing random values from classes of the series so, that all classes have the same amount of measurements, as the class with the least values of each series. This way, every measurement point has the same weight.

A drawback of the classification is the random data point selection. Since there is a lot of possible combinations to select a subset of points from the class, the evaluation script provides slightly different values. To address this problem, it was established, that the effect of random selection is insignificant and was continued with only one randomly selected data point set.

In order to quantify how the particular data point selection affects the results, the fitting has been run with different classification 10000 times for each job. This is of course not even close to the number of all possible combinations. However, the graphical representation of the whole data sets suggests no unexpected behaviour, as all the points lie relatively close to each other and therefore it should not matter, which of the points are selected. The established variance of the parameters from these 10000 samples can be seen in table 4. The values in the table confirm, that the effect is not significant. The k parameter express, how many DNs are equal to one °C. The standard deviation of the d (which denotes the variance of the offset) is negligible in comparison to mean of k. The standard deviation of the offset is equal to $\frac{4.59[DN]}{379.465[DN/°C]} = 0.012°C$ which is well below the best achieved accuracy.

		Job 124			Job 126		
		B+B 40	B+B 50	B+B 60	B+B 40	B+B 50	B+B 60
k	mean	379.46584121	368.21738237	367.79012857	371.09669509	351.62824391	369.45434297
	std	0.10635212	0.05588487	0.09165659	0.09653273	0.04700436	0.04994227
d	mean	7132.69574298	9375.71746904	4157.76623035	7388.73496091	9739.56197789	4301.83836801
	std	4.59214606	1.50416662	2.63424516	1.78281203	1.23893339	1.37569851

Table 4: Variance of the parameters regarding the random point selection. The values were calculated from 10000 combinations for each job

6.3.8 Line fitting

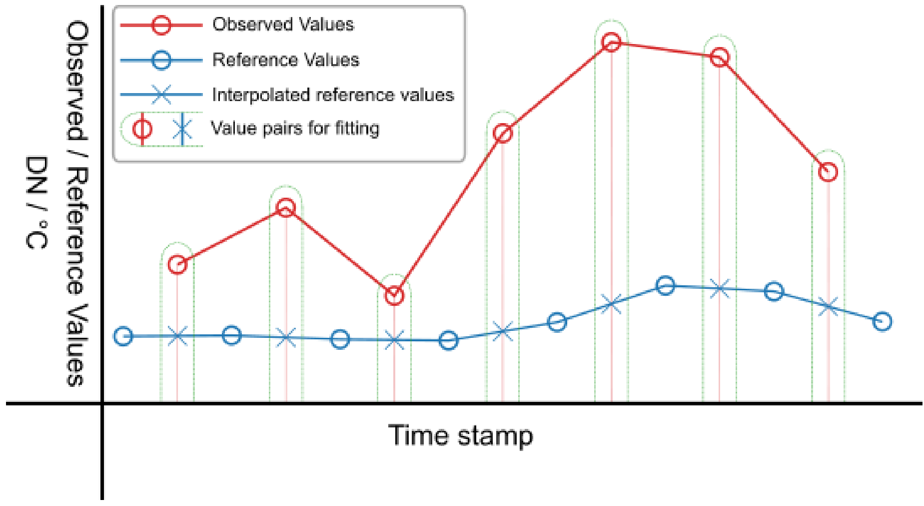


Figure 27: Interpolating values for fitting. The round points designate true measurements whereas the crosses denote an interpolated reference value. The reference measurements are considered variance free, therefore are more suitable for interpolation;

The fitting takes place in the voltage-temperature space. To eliminate the temporal dimension of the both series, they had to be aligned to a single set of timestamps. This can be done by choosing time stamps of one of the series and interpolate the values of the other series to timestamps of the first series. Since the values from the reference sensor are very stable (i.e. the values vary under the resolution of the sensor) this series is much more suited for interpolation than the observations series. So, to every time stamp of the observation series there is a observation value from the sensor prototype as well as an interpolated reference value from the reference sensor (fig. 27) This way the temporal dimension can be neglected and therefore plotted in 2D graph.

The characteristic line is derived from equation 1 as follows:

$$T(v) = kv + d \tag{2}$$

where T is temperature in degrees Celsius, v is voltage in digital numbers, k and d the characteristic line parameters that are to be computed.

The temperature T comes from the measurements of the reference sensor and is considered as variance-free, whereas the voltages v come from the measurements of uncalibrated sensors. Each of the two series is compacted into its own vector $\vec{T}$ and $\vec{V}$. By plugging these vectors into the equation 2 we get an overdetermined system of linear equations

$$\vec{T} = m\vec{V} + b \tag{3}$$

with unknowns m and b. Such system is solved by the means of adjustment computation. However, in adjustment computation, the observations are function values of the unknowns which is in 3 not the case, as the reference $\vec{T}$ is (inversely) formulated as function value of the observations $\vec{V}$ Therefore the system has to be reformulated to

$$k\vec{T} + d = \vec{V} \tag{4}$$

which can be rewritten in matrix notation as:

$$\mathbf{A}\vec{x} = \vec{L} \tag{5}$$

with:

$$\mathbf{A} = \begin{pmatrix} \vdots & \vdots \\ \frac{\delta \vec{L}}{\delta k_{inv}} & \frac{\delta \vec{L}}{\delta d_{inv}} \\ \vdots & \vdots \end{pmatrix} \text{ and } \vec{x} = \begin{pmatrix} k \\ d \end{pmatrix}$$

where $\mathbf{A}$ is a design matrix, $\vec{x}$ vector of the unknowns (char. line parameters k and d) and $\vec{L}$ the observations vector, that contains the measured voltages ($\vec{L} = \vec{V}$) This is a common adjustment computation problem. The condition that has to be satisfied in adjustment computation is, that the sum of the squared residuals $\vec{v}$ (6) is minimal.

$$\vec{v}^T \vec{v} \rightarrow min \tag{6}$$

Residuals are the differences between the calibrated series and the reference series (9).

At first, one has to compute the intial estimates of the parameters. Since we have two parameters, we have to pick two data points from the data set to form a system of two equations with two variables that has a unique solution. The solution of this system yields the estimates k_0 and d_0 for the unknowns k

and d which are compacted into a vector $\vec{x_0} = (k_0, d_0)^T$. The two points can be any pair from the data set. We have simply used the first and the last value for the estimates.

With parameter estimate vector $\vec{x_0}$ we can compute the observations estimate vector $\vec{L_0}$ as follows: $\vec{L_0} = A\vec{x_0}$. The parameter adjustments $\vec{x}$ can now be computed as:

$$\vec{x} = \mathbf{N}^{-1}\mathbf{A}^T\vec{l} \tag{7}$$

with *normal equation matrix* $\mathbf{N} = \mathbf{A}^T\mathbf{A}$ and vector $\vec{l} = \vec{L} - \vec{L_0}$ The unknown parameters we are looking for can be computed by adding the parameter adjustments to the estimates:

$$(k, d)^T = \vec{x_f} = \vec{x_0} + \vec{x} \tag{8}$$

The residuals $\vec{v}$ are then computed as:

$$\vec{v} = \mathbf{A}\vec{x} - \vec{l} \tag{9}$$

To ensure, that no errors occurred in the computations, we can perform simple test by plugging in the adjusted parameters to the original functional model (4) and comparing it with the adjusted observations $\vec{L_f} = \vec{L} + \vec{v}$:

$$\mathbf{A}\vec{x_f} - \vec{L_f} = \vec{0} \tag{10}$$

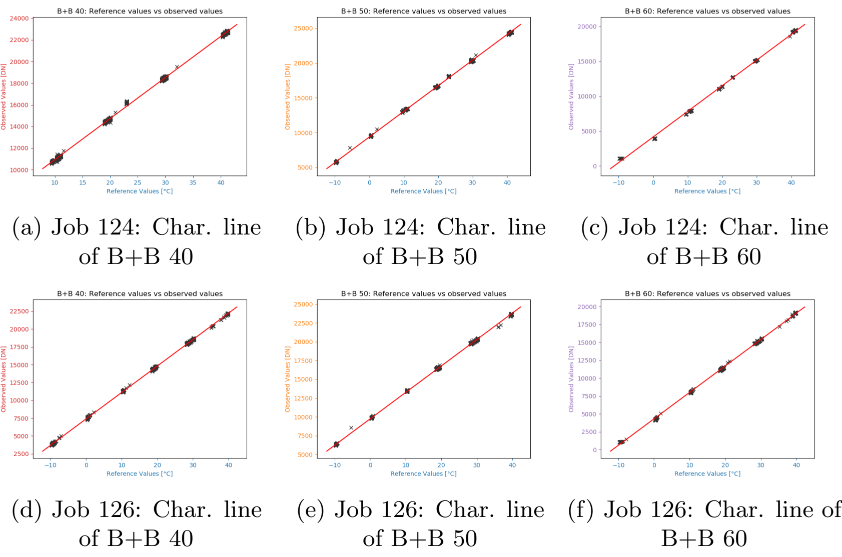
If there were no errors in the fitting procedure, the expression above (10) should be equal to zero vector. However, the values in the above mentioned expression might not be exactly zero due to the floating point arithmetic in which case these values should be considered as zero. In our case these values were smaller than 10^{-15}. Should we encounter greater numbers, it is possible, that the first estimate was not accurate enough, in which case the fitting can be repeated with the current adjusted parameters as the initial estimates. The fitting can be repeated as many times as needed until the desired accuracy is reached provided that the solution converges.

Residuals are essential for calculation of the variance of the unit weight a posteriori, which is a parameter that is used to describe the quality of the fitting. It is computed as follows:

$$s_0^2 = \frac{\vec{v}^T\vec{v}}{n - u} \tag{11}$$

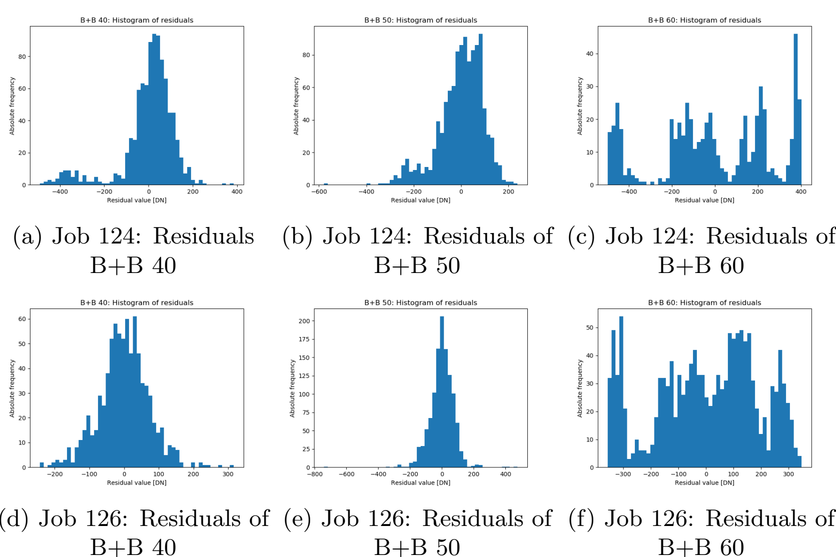
with n as number of observations and u as number of the parameters (in case of a line $u = 2$) The greater the s_0^2 the worse the fit.

The results of the fitting are displayed in figures 28 and 29.



(a) Job 124: Char. line of B+B 40
(b) Job 124: Char. line of B+B 50
(c) Job 124: Char. line of B+B 60
(d) Job 126: Char. line of B+B 40
(e) Job 126: Char. line of B+B 50
(f) Job 126: Char. line of B+B 60

Figure 28: Characteristic lines



(a) Job 124: Residuals B+B 40
(b) Job 124: Residuals of B+B 50
(c) Job 124: Residuals of B+B 60
(d) Job 126: Residuals of B+B 40
(e) Job 126: Residuals of B+B 50
(f) Job 126: Residuals of B+B 60

Figure 29: Residuals of the fitting

6.4 Results of the experiment

If the computation of the characteristic line was successful, the computed parameters can be applied to the original uncalibrated series. This is done by

Parameter		Job 124			Job 126		
		B+B 40	B+B 50	B+B 60	B+B 40	B+B 50	B+B 60
Fitting line	k $[\frac{DN}{°C}]$	379.412483	368.237310	367.747588	371.134370	351.653928	369.480376
	d $[DN]$	7135.547674	9374.440976	4157.484839	7386.632934	9739.557982	4300.846142
	σ_k $[\frac{DN}{°C}]$	7.681702	3.930066	11.088666	3.403352	3.337811	7.930835
	σ_d $[DN]$	213.308256	90.532302	255.850958	71.849100	74.598456	177.580140
	s_0^2 $[DN^2]$	14670.056473	9003.733903	71540.779053	5344.322013	5893.212241	35597.911901
Difference overall	n	7621	10699	10717	8305	8786	8860
	mean	-0.1743	-0.2347	-0.0425	-0.0942	-0.1624	-0.1236
	std	0.5431	0.8032	0.7923	0.6969	1.1930	0.7587
Difference stable	n	1488	2135	2523	1562	1555	2164
	mean	-0.0487	-0.0554	-0.2930	-0.0084	-0.0015	-0.1029
	std	0.3709	0.2805	0.8875	0.2026	0.2101	0.5814
Difference [19 - 21]°C	n	405	415	185	89	75	46
	mean	0.0096	-0.1039	0.3860	-0.0381	-0.1078	0.2667
	std	0.1708	0.1554	0.2445	0.2280	0.2745	0.1994

Table 5: Results of the fitting

solving (4) for $\vec{T}$:

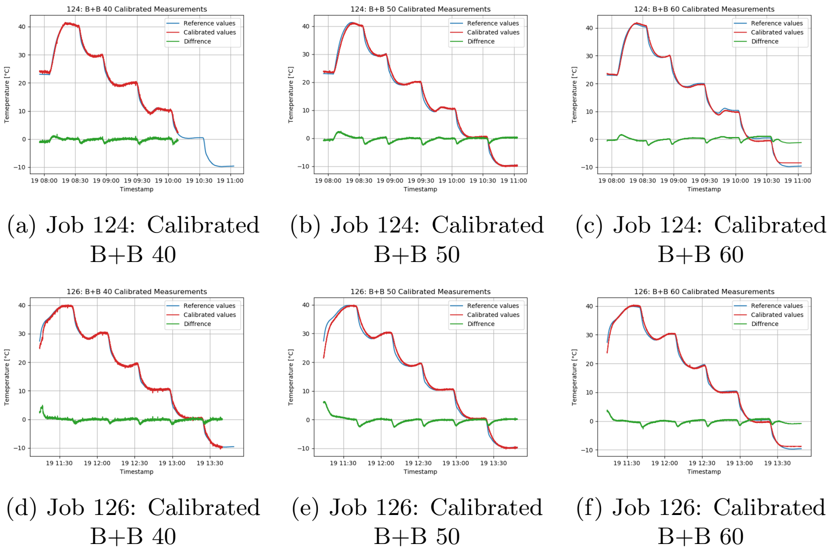
$$\vec{T} = \frac{\vec{V} - d}{k} \tag{12}$$

Subsequently the calibrated series is compared to the reference series by subtracting both series from each other. These series are essentially computed in the fitting as well. The calibrated series corresponds to the adjusted observations vector $\vec{L_f}$ and the difference corresponds to residuals $\vec{v}$. However, the vectors from the fitting contain only values from the stable intervals, whereas the difference between the reference series and the calibrated series contains overall differences (i.e. even in the unstable intervals). The dynamic parts are very easily observable in figures 30 (spikes of the green series) Table 5 contains results of the fitting. Since we are again interested in the non-dynamical part of the measurements, we have selected only those values that were in the stable intervals computed in section 6.3.6 (difference stable). As mentioned before, the prototypes are going to be deployed in the measurement lab with relatively constant temperature around 20 degrees, the table 5 contains information about the quality of the calibration especially in that interval (difference [19 - 21]°C).

6.4.1 Interpretation

B+B 40

This sensor has failed in the first measurement as the temperature approached 0 °C providing obviously incorrect data which had to be filtered out (fig. 21a blue



(a) Job 124: Calibrated B+B 40
(b) Job 124: Calibrated B+B 50
(c) Job 124: Calibrated B+B 60
(d) Job 126: Calibrated B+B 40
(e) Job 126: Calibrated B+B 50
(f) Job 126: Calibrated B+B 60

Figure 30: Calibrated sensors

line). After the filtering, the result is not as good as the result of B+B 50 but can be considered good compared to B+B 60. This result may be surprising given the significantly greater variance of the values produced by this sensor (table 2). In the second measurement the sensor has not failed and performed even better than B+B 50.

B+B 50

From the figures 28 and 30 the most appealing results have been achieved by the sensor B+B 50. This positive judgement is based on two facts: Unlike B+B 40, B+B 50 has not failed in any of the measurement sessions and in comparison to B+B 60, B+B 50 has a better fit than B+B 60. The consistence of solid results in both measurement sessions contributes the most to the positive judgement. The σ_0^2 values in table 5 suggest the same, even though B+B 40 did perform minisculely better in the second measurement.

However, values of overall differences suggest the contrary. Considering the extent of the spikes of the green lines 30 which protrude the most by B+B 50. This suggests, that the probe of the B+B 50 reacts to temperature variations at slowest compared to the other probes. This might not be caused only by some structural differences of the probes, but more likely by the placement inside the climatic chamber. Unfortunately it was not documented, where exactly the probe was positioned. If the unstable data are filtered out, the B+B 50 is the most consistent sensor with in average the smallest standard deviation of the residuals.

B+B 60
Despite exhibiting the smallest measurement noise, the data collected by this sensor aren't obviously optimal. This is apparent from the figures 29 and after taking a closer look from figure 28 and confirmed by the values in table 5 especially s_0^2. The hardware parts (apart from the probe) of this sensor were in both measurements kept outside of the chamber. This sensor has not been provided by protective housing and prolonged cables. However this enabled us to plug in another A/D converter. It seems, that the second converter performed better, nevertheless the s_0^2 was about ten times greater than the s_0^2 of the other sensors.

6.4.2 Hypotheses testing

Since the set up of the experiment was deliberately changed between the two measurement sessions, a hypothesis testing was conducted in order to decide, whether the changed set up of the experiment had some significant effect on the results. The testing is based on comparison of the fitting parameters of the sensors between the measurement sessions.

A null hypotheses H_0 have been formulated as follows:

$$H_0 : k_{126} - k_{124} = 0 \text{ and } H_0 : d_{126} - d_{124} = 0 \tag{13}$$

with alternative hypotheses

$$H_0 : k_{126} - k_{124} \neq 0 \text{ and } H_0 : d_{126} - d_{124} \neq 0 \tag{14}$$

For the test statistics t was used following relationship:

$$t = \frac{d_p}{\sqrt{\sigma_{d_p}^2}} \approx t_{n-4,0.975} \tag{15}$$

with

$$\begin{gathered} d_p = k_{126} - k_{124} \text{ resp. } d_p = d_{126} - d_{124} \\ \text{and} \\ \sigma_{d_p}^2 = \sigma_{k_{124}}^2 + \sigma_{k_{126}}^2 \text{ resp. } \sigma_{d_p}^2 = \sigma_{d_{124}}^2 + \sigma_{d_{126}}^2 \\ \text{and} \\ n = n_{124} + n_{126} \end{gathered} \tag{16}$$

where n_1 and n_2 are number of observations from both fits that participated on the fitting. The variances of the parameters (σ_k^2, σ_d^2) are located on the main diagonal of the variance-covariance matrix a posteriori C_{xx} which is computed during the fitting as part of the stochastic model. The computed test statistics t is then compared to $t_{n-4,0.975}$ (quantile of the t-distribution with $n-4$ degrees of

freedom and 5% probability of error). If $|t| > t_{n-4,0.975}$ then the null hypothesis H_0 is rejected in favour of alternate hypothesis H_A.

However, to be able to formulate the hypothesis test this way, a condition of $s^2_{0,124} = s^2_{0,126}$ (variances of unit weights are equal) must be satisfied. If that is not the case, a weight matrix P has to be formed so, that the condition is satisfied. This is done by multiplying the P matrix (per default an unit matrix) by the ratio of the variances of the unit weight $\frac{s^2_{0,124}}{s^2_{0,126}}$.

6.4.3 Deployment of another A/D Converter

To examine whether it is possible to deploy another A/D converter of the same type in the sensor circuit without recalibration, a hypothesis test (section 6.4.2) was conducted for the B+B 60 sensor, since this prototype was deployed with different A/D converters. The outcome of the test determines whether the separate fitting line parameters are significantly different or not. After satisfying the condition, the test statistics t_k and t_d (15) were calculated.

$$\begin{gathered} t_k = 2.352690022 \\ t_d = 8.835176443 \\ \text{and} \\ t_{n-4,0.975} = 1.960 \end{gathered} \tag{17}$$

Since t_k and t_d are greater then $t_{n-4,0.975}$, both null hypotheses are rejected in favour of H_A, therefore it is concluded the parameters are significantly different.

This means, that it is unfortunately not possible to replace an A/D converters (even of the same type) in the circuit without recalibrating the prototype. From figure 31 one can see, that the two lines are offset about a 0.5 °C, which is caused by greater difference in the d parameter. In comparison to the other prototypes (figures 32a and 32b), the difference in k parameter is not so severe and can not therefore be so easily observed in figure 31.

However, this outcome was not expected and is rather surprising. In order to provide a more conclusive evidence the experiment should be repeated several more times.

6.4.4 Effect of the ambient temperature on performance of the sensors

Same hypothesis test as in previous section (6.4.2) was conducted with the B+B 40 and B+B 50 prototypes. In the first measurement, there were not only the probes, but also other hardware parts (except for the controlling computer)

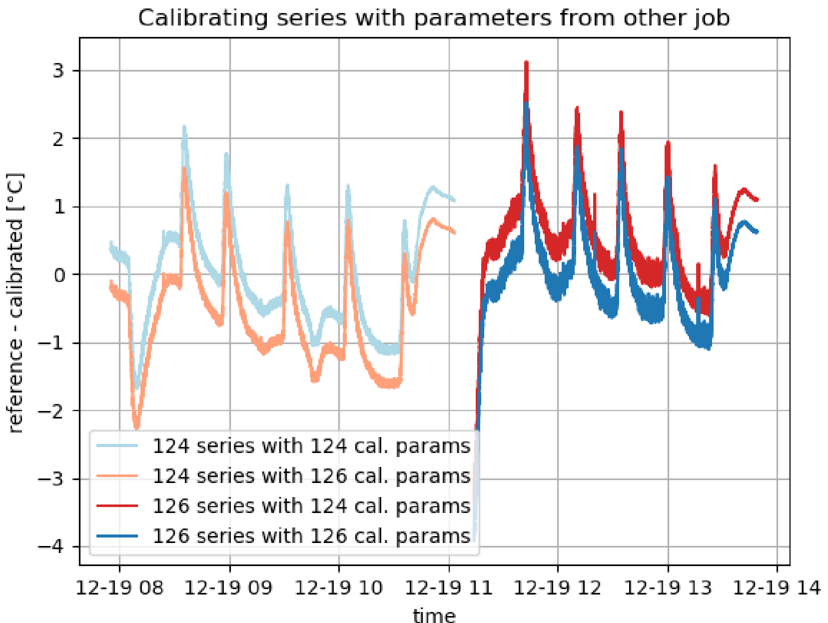
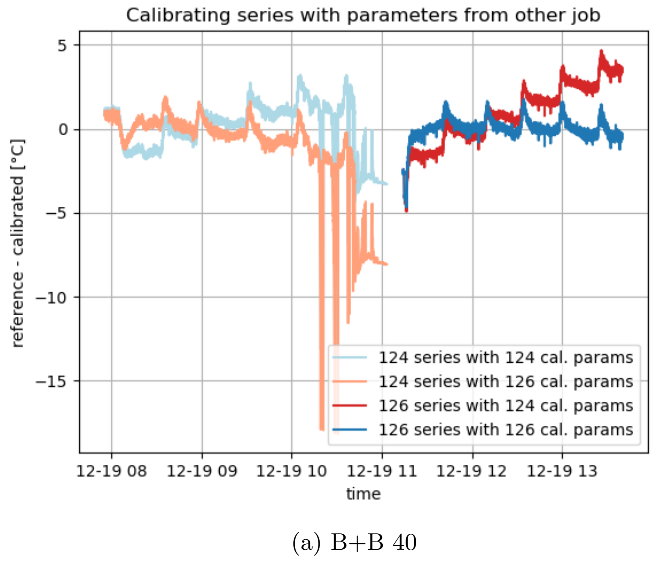


Figure 31: B+B 60

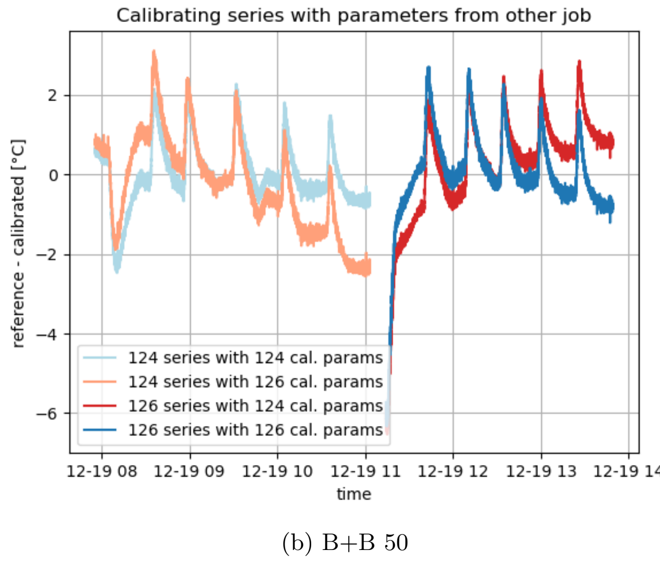
left inside the chamber, whereas in the second measurement only the probes were kept inside the chamber. The hypotheses test determines, whether the ambient temperature has an significant effect on the measurement or not. After satisfying the condition, the test statistics t_k and t_d (15) were computed:

$$\begin{gathered} t_{k,bb40} = -22.28837983 \text{ and } t_{k,bb50} = -16.637937926 \\ t_{d,bb40} = -75.85838160 \text{ and } t_{d_b b50} = 72.9792743397 \\ \text{and} \\ t_{n-4,0.975} = 1.960 \end{gathered} \tag{18}$$

These values provide a very strong evidence against all null hypotheses H_0 and therefore the alternative hypotheses H_A were accepted. The alternative hypotheses state, that the ambient temperature has indeed a significant effect on the measurement. This might be explained by the shrinking and expanding of the hardware parts. The severity of the effect can be easily spotted in figures 32a and 32b as the red and blue lines diverge from one another. However, analogously to the conclusions drawn in section 6.4.3, the experiment should be repeated to increase the validity of these conclusions.



(a) B+B 40



(b) B+B 50

Figure 32: Using calibration parameters from other jobs. The blue-ish lines represent the series calibrated with parameters calculated from their own measurements, whereas the red-ish lines show the calibration of the series with parameters that were gained from the other measurement series

7 Summary

7.1 Implementation of the framework

In this thesis, a concept of a framework for multi purpose sensor networks was presented and implemented.

For the data acquisition and communication between the network components was used Robot Operating System (ROS). The author has developed a ROS package that standardises the data acquisition process so that values from different kinds of sensors or *value providers* can be received by a single program. This simplifies the data storage as the program reformats incoming data into a common format.

The network is controlled by a web application which makes the usage of the sensor network even easier and user friendlier, as it can be accessed from any device and from anywhere, provided, that the sensor network is connected to the internet. The web application does not contain an evaluation tools. It is only used for data collection and raw data inspection and export.

To demonstrate the operation of the framework, the author has conducted a calibration of three temperature sensor prototypes built on our institute. Apart from finding the calibration parameters of the prototypes, author also tested two hypotheses. The summary of the experiment can be found in following section 7.2

7.2 Calibration experiment

The experiment set out with three goals:

1. Find parameters of the characteristic lines of the sensors
2. Test, whether we can use the same calibration line parameters for two different amplifiers and A/D converters of the same type.
3. Examine the effect of the temperature on the hardware parts of the prototypes.

We have managed to determine parameters of the characteristic lines of the sensors B+B 40 and B+B 50 with an accuracy of 0.2 °C. For the B+B 60 we have been able to determine the parameters only to 0.56 °C accuracy. In planned temperature range in which the sensors are going to be deployed, we have been able to reach even better accuracies up to 0.15 °C.

From the evaluated data we have concluded, that it is unfortunately not possible to change individual components of the sensor without having to recalibrate the sensor. Not even when the components are of the same type.

Our data have lead us to conclusion that the surrounding temperature probably has some effect on the measurement. However, one would have to conduct the experiment few more times to find out, whether it is the environment temperature or some other parameter causing the different outcomes.

8 Outlook

Although not all features of the framework have been implemented yet, the conducted experiment has shown, that the framework is operable. However, the claims of the author about a multi purpose functionality of the framework are yet still to be proven. The author is of that opinion, that a considerable amount of effort is still needed in order to finish the framework.

It would be also useful to integrate evaluation tools to the framework. This poses yet greater challenge, as the evaluation of the all kinds of data need

completely different tools. In order to truly incorporate these tools into the framework, one would have to program the tools so, that they can be modularly added to the framework without having to edit already existing code.

Since the *Igros* Web Application stores the number of measurements it has made to produce one value and the standard deviation of those values, it is planned, to include these a priori information about the measurements themselves in the fitting process. However, this will require significant changes reps. extension of the evaluation module.

The both repositories, in which the both igros packages and the evaluation module are located in are planned to be published in the departments Git Repository as soon as they are finished.

References

[1] Rejchrt, D., Thalmann, T., Ettlinger, A., Neuner, H.-B. (2019). Robot Operating System - A Modular and Flexible Framework for Geodetic Multi-Sensor-Systems. In *20. Internationale Geodtische Woche Obergurgl 2019* (pp. 177-188) Herbert Wichmann Verlag, VDE VERLAG GMBH Berlin Offenbach.

[2] Mser, M. Mller, G., Schlemmer, H. (Ed.), Heunecke, O., Kuhlmann, H., Welsch, W., Eichhorn, A., Neuner, H., Auswertung geodtischer berwachungsmessungen, 2. Aufl. Handbuch Ingenieurgeodsie, Herbert Wichmann Verlag, VDE VERLAG GMBH Berlin Offenbach.

[3] ROS Metrics (2020) Community Metrics Report July 2019. http://download.ros.org/downloads/metrics/metrics-report-2019-07.pdf (24.01.2019)

[4] ROS Wiki (2020). Robot operating system. Open Source Robotics Foundation. http://wiki.ros.org/Services (21.11.2018)

[5] Django Home Page (2020). Django Software Foundation https://www.djangoproject.com/

[6] Bootstrap Home Page (2020). https://getbootstrap.com/

[7] Neuner, H. (2014). Physikalische Prinzipien der elektronischen Messwertumwandlung

[8] Neuner, H. (2019). Elektrische Messwertgeber [Lecture slides] VO Ingenieurgeodsie Vertiefung 128.032, TU Wien

[9] Siemens, W. (1871). "On the Increase of Electrical Resistance in Conductors with Rise of Temperature, and Its Application to the Measure of Ordinary

and Furnace Temperatures; Also on a Simple Method of Measuring Electrical Resistances". The Bakerian Lecture. Royal Society. Retrieved May 14, 2014.

[10] What is a Pt1000 sensor? (n.d.). Retrieved January 11, 2020, from https://www.peaksensors.co.uk/what-is/pt1000-sensor/.