HTBLuVA Wiener Neustadt
Höhere Lehranstalt für Informatik

DIPLOMARBEIT

GRAMOC - Gradienten Magnetometer Online Controller

Ausgeführt im Schuljahr 2017/18 von:

Entwicklung Server, Netzwerkprotokoll
Nico Kratky 5CHIF-13

Entwicklung App, Visualisierung
Nico Leidenfrost 5CHIF-14

Betreuer / Betreuerin:

Dr. Michael Stifter

Wiener Neustadt, am 4. April 2018

Abgabevermerk:

Übernommen von:

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche erkenntlich gemacht habe.

Wiener Neustadt, am 4. April 2018

Verfasser / Verfasserinnen:

Nico Kratky

Nico Leidenfrost

Contents

htl bildung mit zukunft	
	Fachrichtung: Informatik

Namen der Verfasser/innen	Nico KRATKY Nico LEIDENFROST
Jahrgang Schuljahr	5CHIF 2017 / 18
Thema der Diplomarbeit	Gradienten Magnetometer Online Controller
Kooperationspartner	F-WuTS

Aufgabenstellung	Für den industrielle Einsatz eines hochsensitiven MEMS Gradienten Magnetometer müssen dessen Sensorwerte möglichst echtzeitnah ausgelesen und in geeigneter Form dargestellt werden.

Realisierung	C++ für alle serverseitigen Applikationen HTML, CSS, JavaScript, Vue.js, Plot.ly für die mobile Applikation

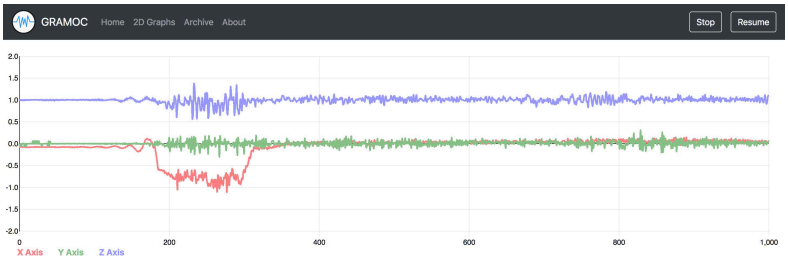
Ergebnisse	Server zur Aufbereitung und Verteilung von Sensordaten Applikation zur Speicherung von Sensordaten REST-API zur Bereitstellung historischer Daten Web App zur Visualisierung von Sensordaten

Authors	Nico KRATKY Nico LEIDENFROST
Form Academic Year	5CHIF 2017 / 18
Topic	Gradient Magnetometer Online Controller
Co-operation partners	F-WuTS

Assignment of tasks	For industrial use of a highly sensitive MEMS gradient magnetometer, its sensor values must be read out as close to real-time as possible and displayed in a suitable form.

Realization	C ++ for all server-side applications HTML, CSS, JavaScript, Vue.js, Plot.ly for the mobile application

Results	Server for processing and distribution of sensor data Application for storing sensor data REST-API for providing historical data Web App for the visualisation of sensor data

Illustrative graph, photo (incl. explanation)	Screenshot of the mobile app with data of an accelerometer during emergency braking 

Participation in competitions, Awards	

Accessibility of diploma thesis	HTBLuVA Wiener Neustadt Dr.-Eckener-Gasse 2 A 2700 Wiener Neustadt

Approval	Examiner	Head of Department
(Date, Sign)	MMag. Dr. Michael Stifter	AV Dipl.-Ing. Felix Schwab

Kurzfassung

Diese Diplomarbeit stellt GRAMOC vor. GRAMOC ist ein System, dass es ermöglicht Stahlbänder, ohne die Produktion stoppen zu müssen, effektiv zu charakterisieren. Dies wird durch die Erfindung eines hochsensitiven MEMS Gradienten Magnetometer ermöglicht.
Diese Prozedur steigert nicht nur die Produktivität, sondern ist auch material- und kosteneffizient.
Die ersten Versuche diese Problemstellung zu lösen wurden mit einem TCP-basiertem Server und einer nativen Android Applikation unternommen. Schnell stellte sich heraus, dass dies nicht die geforderten Echtzeitkriterien erfüllen kann. Diese Erfahrung resultierte in einem Neustart des Projektes mit geänderten Anforderungen. Die Android App wurde durch eine Webanwendung mit responsiven Design ersetzt. Dies erweitert nicht nur die Anzahl der unterstützen Endgeräte, sondern bringt auch Vorteile in punkto Drittanbieter Visualisierungsbibliotheken.
GRAMOC besteht letztendlich aus zwei Kommandozeilenprogrammen und einer Web Applikation. Das erste Kommandozeilenprogramm ist der Server, der die vom Sensor empfangenen Daten vorverarbeitet. Dieses Programm führt auch eine dynamische Datenanalyse durch. Das zweite Kommandozeilenprogramm kümmert sich um die Datenspeicherung, da alle Sensordaten in HDF5 Dateien gespeichert werden müssen, um eine nachträgliche Inspektion der Daten zu ermöglichen. Beide Programme laufen auf einem Raspberry Pi 3 Model B. Das Client Programm ist eine Web Anwendung die dazu dient, Sensordaten visuell darzustellen. Auch wird ein Formular zur Verfügung gestellt, um historische Sensordaten aus den HDF5 Datein abfragen zu können. Die kabellose Übertragung von Daten erfolgt über ein Wireless LAN Netzwerk, unterstützt durch ein eigens entwickeltes UDP-basiertes Netzwerkprotokoll.
Die empfangenen Sensordaten werden mittels multipler Linear Regression analysiert. Dies ermöglicht es, mechanische Parameter des Stahlbandes von den magnetischen Daten abzuleiten.

Abstract

This diploma thesis introduces GRAMOC, a system that can help to effectively characterise steel belts without halting the production lines. This is made possible by using a recently invented MEMS gradient magnetometer to measure the magnetic field of produced steel belts.
This new procedure not only increases productivity, but is also a lot more material and cost efficient.
The first approaches to solving this problem were made using a TCP-based server and a native Android application. It quickly turned out that this solution would not satisfy the requested real-time criteria. This experience resulted in a project restart with changes requirements. The Android application was replaced with a webapplication with responsive design. This does not only extend the amount of supported devices, but also has some advantages with regards to third-party plotting libraries.
GRAMOC ultimately consists of two command line programs and one web application. One command line program is the server that processes the data that is received from the sensor. This program also performs dynamic data analysis using this data. The second command line program handles data storage, as all sensor data has to be saved to HDF5 files to allow further inspection. These two programs run on a Raspberry Pi 3 Model B. The client program is a web application that visualises the received sensor data. It also provides a form to the users to request historical sensor data from the HDF5 files. The data is transmitted wirelessly via a Wireless LAN network supported by a specially developed UDP-based networking protocol.
The received data is analysed using multiple linear regression. This analysis method allows to predict mechanical parameters of the steel belt from magnetic data that is received from the sensor.

Chapter 1

Introduction

Industry is ever-changing. Especially people working in the information technology branch know that, since these are the people that have to upgrade the current systems using latest technology. The latest industry-changing milestone was the rise of the so-called Industry 4.0, which combines regular mechanical processes with modern information and communication technology.
Industry 4.0 is a term that was coined by the German government [30]. It describes the fourth industrial revolution. As explained in *Im Wandel der Zeit: Von Industrie 1.0 bis 4.0*, the first industrial revolution took place around 1800 with the rise of steam and water-powered machines [58]. One century later electricity heralded the start of the second industrial revolution, production lines being one of the biggest milestones. Also division of labour was first practiced. The third industrial revolution occurred with the invention of computers, robots and computer automation. The fourth and final one basically just refines the third revolution. This revolution includes the term *cyber-physical systems*, which are systems that are controlled by computers, algorithms and sensors. This also means that there has to be some kind of communication between these systems which happens mostly over the internet. Figure 1.1 depicts this sequence of revolutions.

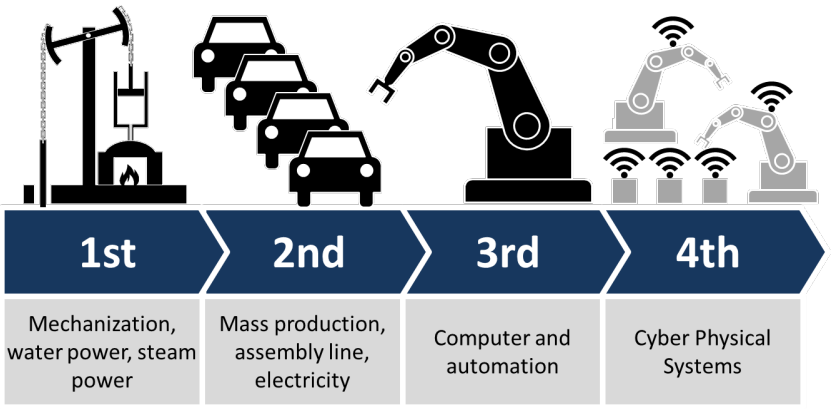


Figure 1.1: The four industrial revolutions that took place over the last centuries [42]

This drastic change means that many companies have to adapt to keep up with the competing companies that already have these technologies. Steel belt production companies are no exception. With the invention of a gradient magnetometer that can effectively characterise steel belts, the foundation for this diploma thesis was laid.

1.1 Task

The task of this diploma thesis is to develop a system to read sensor data, process it and visualise the results. Sensor data is continuously read from a highly sensitive MEMS gradient magnetometer. This data is structured as raw binary data and has to be processed by the system. The processed data will also undergo statistical analysis to predict parameters on the basis of this data. After this processing step the data has to be sent wirelessly to a mobile app. This mobile app acts as the client-side of the system. The app visualises the sensor data and its predicted parameters. The app should also offer a way of browsing through historical data that was saved prior.

1.1.1 Requirements of GRAMOC

Server side requirements:

- Read data from a sensor
- Save sensor data for further inspection
- Predict mechanical parameters from sensor data
- Send data to clients
- Provide historical data to clients

Client side requirements:

- Visualise sensor data
- Provide a form for requesting historical data
- Visualise historical data

1.2 Existing Solutions

1.2.1 Steel Belt Quality Inspection

Currently there are no solutions for dynamic steel belt characterisation. All these measurements have to be made manually.
The current procedure to inspect the quality of a steel belt is as follows: The first thing that has to be done is to produce a roll of steel belt. To get the quality level of this product, a sample has to be taken from it. There are two samples taken from each steel belt roll, one from the start and one from the end. These two samples can now undergo quality inspection procedures. The results from these tests can be used to assess the produced steel belt. According to these tests, the parameters of the production machines can be adjusted.
This procedure has a few major disadvantages. Firstly, if the product does not pass the quality tests, the whole steel belt has to be discarded. Time and personnel are also two big disadvantages. These quality tests are not only time consuming but they also require special trained staff for conducting these inspections.

1.2.2 Handling Sensor Data

Currently there are a lot of solutions available that can plot sensor data. The majority of these are even free. The one constraint that most of these solutions share is that the sensor has to be directly connected to the computer. As the sensor that is used for this project sends its data over the network, almost all solutions are considered irrelevant. Also

some custom features are wanted that these programs do not offer. For example visualizing historical data.

1.2.3 Plotting Real Time Data

Author: Nico Leidenfrost

As already mentioned there are a lot of solutions out there that can be used to plot sensor data. But the one thing that these solutions mostly can not provide is real time plotting. Static plots can be achieved in many different ways, with big amounts of data or just small amounts, many plots combined or divided in separate plots and many more variations are within the bounds of possibility. A lot of these solutions promote themselves with *dynamic data updates* or *streaming data*. That just means that the data can be changed at runtime and therefore some could say the data is displayed in real time. But real time can be defined very differently. As one would say real time applications can update their data once every second, others consider that the data must be updated within less than 20 milliseconds to achieve a high framerate. Most of the solutions available can handle the former definition of real-time but nearly none of them can provide enough performance for the latter. Another important point is the amount of data that one wants to depict, because most of the already existing programs that can handle real-time are just powerful enough to handle small amounts of data.

1.3 Outline

This diploma thesis is structured into two big parts. These parts can be seen as two phases of implementation. Each phase is completely separate. The first phase is a more experimental one as both authors were unfamiliar with these types of projects, so some experience had to be made. At the end of this phase there was a big cut and the project was restarted from the beginning. The second phase discusses the different approaches and decisions that were made starting from this cut. The second phase was not only better planned, but the decisions that were made, were mostly made out of experience from the first phase.

Chapter 2

Real Time Systems

Author: Nico Kratky

2.1 Definition

Real-time systems (RTS) have one big constraint that normal computer programs don't have, time. Contrary to normal applications, where the correctness of data only depends on the made computations, RTS also depend on the timing of these computations. This time limit, that has to be adhered to, is also often called deadline. Types of RTS are mostly differentiated between what happens when the deadline is not met. There are three basic types of real-time systems.

2.1.1 Hard Real-time Systems

In hard real-time systems an overrun in response time will lead to failure [26]. This can mean big financial losses of even danger to life. An example for a hard RTS would be the ECU (Electronic Control Unit) of a modern car. If the timing of the fuel injection or ignition is not correct, the engine could fail and lead to a crash. Another example would be the control unit for airbags. This unit has to constantly monitor the cars crash sensors and decide whether to trigger the airbags or not. This system would have to fulfill the hard criteria because if the airbags trigger to late, then human safety can not be guaranteed.

2.1.2 Soft Real-time systems

A soft real-time system still does have a deadline but it is not that big of a deal if it is not met [26]. There are some consequences to not meeting the time limit but they are tolerable. A video-stream is a example for a soft real-time system. If some frames are not delivered in time the video will stutter, but the content will still be delivered.

2.1.3 Firm Real-time systems

In firm real-time systems the data will be useless if the time limit is overrun [26]. The data will then be discarded. This is the type of RTS that GRAMOC can be associated with.
Author: Nico Leidenfrost

2.2 Programming Language

In the early days of computer programming, there were only a few programming languages available. Today there is a broad variety of them ready to be used. The popularity of a language can reach from only a few users to worldwide professional use. To create applications that are able to process streaming data in real-time, only a minority of languages are considered useful. Reasons why some programming languages are used more often than others are [10]:

- The high performance of the native implementation
- The ease of use
- The popularity and community support

2.2.1 Java

Java was a long time a major language for developing real-time web applications because of their "Write once Run everywhere" principle. That is possible because of the Java Virtual Machine(JVM). All the code written in Java is compiled to run inside this virtual machine, therefore every system that can run the JVM, can also run the same Java code as all the other systems. The client side web development was early replaced by Adobe's Flash project, since then Java is disabled per default in most of the web browsers.
Java did however find its place at the server side, the so called back-end, especially because the Java Database Connectivity(JDBC) was developed in the early stages of Java and enabled an easy way to interact with databases. One of the most important points why somebody would use Java was because it is easy to integrate third-party packages as a result of many available package management systems. Also the deployment of the finished application was easier than the deployment of an application from their main competitor C++.

Scala and Clojure

A variety of languages were designed to also run inside the JVM, two of the more popular ones in real-time programming are Scala and Clojure. Both these languages can use Java packages as they run in the same environment. Scala is mostly used for academic projects, but as of their rich standard library it is also used in high performance server applications. The distinction to Java is that Scala utilises features from functional programming languages although it is declared as a object-oriented language. Clojure is a dialect of Lisp that can also make use of Java packages.

2.2.2 JavaScript

JavaScript is the most popular programming language in terms of web development, it is supported by every browser and during the development every browser developer wanted to have the fastest JavaScript engine. Thanks to that JavaScript is now incredibly fast and capable of implementing web applications on its own. The only similarities between JavaScript and Java are the name, which was a marketing gag, and the syntax, because they both inherit some parts of it from C/C++. Any other aspects of these two languages are distinct from each other. JavaScript is a functional programming language which means functions are treated the same way as data. In JavaScript a function can be assigned to a variable and be passed to another function. A lot of JavaScript frameworks and libraries rely on that feature. Since JavaScript quickly gained a lot popularity in the front-end

development it is also capable of running in the back-end, most of the time as a Node.js server (see section 13.4 on page 51).

2.2.3 C/C++

These two languages are known for their efficiency and therefore are often considered to be used within real-time projects. C is widely used in embedded system programming while C++ is used in all kinds of programming. C++ is a superset of C, regardless of that fact C is still used in low-level system programming because of the simplicity. C++ is more complex than C but it also offers features from object-oriented programming. C as well as C++ were first introduced way before the other languages mentioned here, therefore the developers had enough time to optimise the compilers for these languages, which resulted in very efficient code at runtime. Since these languages are considered as low-level programming languages, a developer can gain control over system resources more easily and use them efficiently. This can also be seen as a huge problem when used by people that do not have the required skills to use it correctly or people who exploit this feature on purpose. High performance applications like video games, as well as real-time applications rely heavily on C++ because of its performance.

2.2.4 Go

Go is a language developed at Google based on the C language but with mechanisms included that provide concurrency. This language is still under development and therefore the variety of available libraries and community support is not as great as with the other languages. Nevertheless are the benchmark performance on web server development still very good.

2.3 Data Transfer

In order to build an application that is capable of displaying sensor data in real-time, like GRAMOC it is crucial to find the optimal way to transfer data from the server to the client. There are many solutions to this problem, but two of them are especially popular when it comes to real-time communication:

- Sever Sent Events
- WebSockets

2.3.1 Server Sent Events

Server Sent Events(SSE) were introduced in 2006 as a protocol to transfer data from a server to a client [57]. Before SSE was introduced many client server communications relied on polling, a technique where the server was asked for new information by the client in a constant interval. This technique obviously created an enormous amount of overhead, because every time a request for new data was sent there had to be a new HTTP connection established and afterwards destroyed. SSE was build to be efficient, it only creates one HTTP connection where all the data from the server is pushed and then received by the client through events. The biggest advantage of SSE is of course the long lived connection between the server and the client, but there are also a few downsides of this protocol. The main problem is that the communication is unidirectional, which means the client can receive data from the server, but can not send any data back. Since GRAMOC relies on bidirectional communication between client and server this method is unqualified to be

used in this project. A few use cases of SSE would be notifications, status updates or the streaming of stock tickers.

2.3.2 WebSockets

In 2011 the WebSocket protocol was standardised by the RFC6455 [11]. WebSockets offer instead of unidirectional communication like server sent events, bidirectional communication between server and client (see subsection 2.3.1 on the preceding page). WebSockets communicate per default over port 80, therefore there are no problems with the firewall. Unlike protocols before WebSockets perform a handshake on connection to upgrade the connection. The WebSocket protocol is way more complicated than the SSE protocol, but at the time of writing most browsers offer solid native WebSocket support and many libraries that implement WebSockets are existing and well maintained. The probably most popular WebSocket library is called socket.io, this library is also used within GRAMOC to communicate between the server and the client (see section 13.6 on page 52). Other use cases of WebSockets would be applications that rely on bidirectional real-time updates like games or chat applications.

Part I

Implementation Phase 1

Chapter 3

Networking

Author: Nico Kratky

As sensor data is received over a network connection and should also be delivered to clients wirelessly, a common way of communication had to be developed. This development process resulted in GSDEP, GRAMOCs networking protocol. It is a TCP-based networking protocol that is used for sending large amounts of sensor data [36].

3.1 Data Flow

Figure 3.1 depicts the handshake performed by GSDEP that is based on TCP's three-way handshake. The client sends a synchronize (SYN) message to the server to let it know that it wants to connect. If the server can accept new clients it returns an acknowledgment message (ACK). The client then also returns this acknowledgment message to inform the server that it is indeed connected. The connection now is established and data can be transmitted.

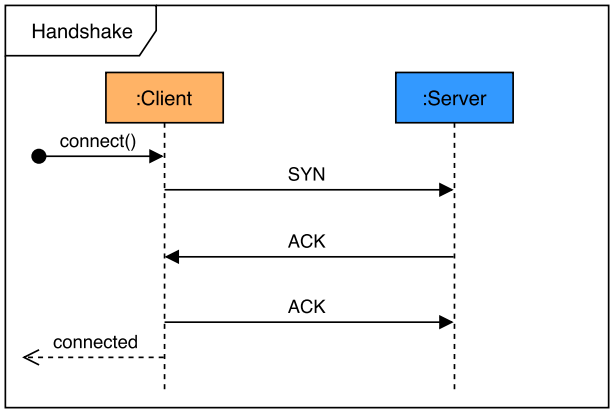


Figure 3.1: TCP-like three way handshake performed on client connect

If a client wants to disconnect from the server it will send a disconnect message (FIN) to the server. Before it actually disconnects, it has to wait for the server to finish cleaning up and return the FIN packet. After the client has received this message, it can close the connection and shut down. This procedure is shown in figure 3.2 on the next page.

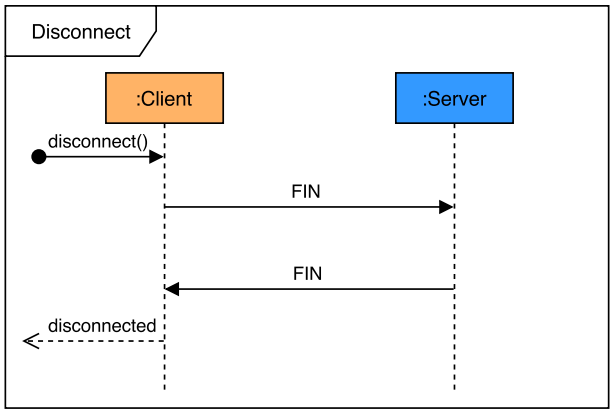


Figure 3.2: Two way handshake performed on client disconnect

3.2 Data Interchange Format

Messages have to be brought to a common format to be understood by all communication partners. Therefore every transmitted message is prefixed with a header. This header includes additional information that is used by the receiving end to determine the size of the payload (see section 3.5 on the following page), to differentiate between different kinds of messages (see section 3.4) and to rebuild the message data to its correct data type. The header consists of 8 bytes, 4 bytes to store the payload length, and 2 bytes each for data type and channel. A example packet is illustrated in figure 3.3.

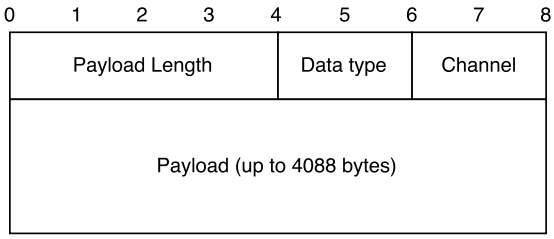


Figure 3.3: Structure and field sizes of a packed message sent with GSDEP

3.3 Commands

Commands are special message that are used to prompt the other end to do something. These commands are used for two purposes. On the one hand they are used during the connection establishment and termination phases, and on the other hand they are used to request data or to stop data transmission. These commands are listed in table 3.1 on the next page.

3.4 Channels

In the case of GRAMOC, where large amounts of data are received in short periods of time, it is crucial to differentiate between communication data and sensor data in split seconds. To accomplish this, 2 bytes are included in the message header. This field simply

Command	Used by	Meaning
SYN	client	Tells the server that a new client is waiting for the connection procedure
ACK	server & client	Tells the other end that it acknowledges the previous command
FIN	server & client	Tells the other end that it will disconnect
STD	client	Tells the server that a client requests data
SPD	client	Tells the server that a client does not want any more data

Table 3.1: Commands sent by one of the connection partners and what they do

contains numbers that represent different channels (see table 3.2). This information can then be used by the client to tell apart these two types of data, without even analysing the payload.

Channel	Value
Communication	1
Data	2

Table 3.2: Channels used to distinguish between message types

3.5 Message framing

A common mistake that many developers make is to assume that TCP operates with messages and that TCP can tell apart these messages [6, 46]. Sadly this is not true as TCP operates with continous streams of data. Therefore the differentiation of messages has to be done by developers themselfs. This can be achieved in two ways. These procedures are taken from the blog posts *Message Framing* and *Understanding The Internet: How Messages Flow Through TCP Sockets* [6, 46].

3.5.1 Delimiters

Sending

Using delimiters is probably the simplest solution. This can be done by sending a special character between each message. This character can either be a character that does not show up in actual messages (e.g. a Null character), or a character that is present in a message. If the second approach is used, every message has to be run through an escaping process which replaces these characters in the messages.

Receiving

Receiving delimited messages is relatively straightforward. The program knows that a message has been fully read when it encounters a delimiter character. This message then has to be passed to an unescaping function when a delimiter character is chosen that can exist in messages.

3.5.2 Length Prefixing

Another method of message framing is to prefix each message with its length. When doing so the format of this prefix has to be stated explicitly. In the case of GSDEP that is a 4 byte unsigned integer.

Sending

First, the message has to be encoded into its binary representation. To send this message, the length followed by the binary encoded message simply has to be sent.

Receiving

Receiving one message is done by first reading into a buffer with the length of the length prefix (in this instance the buffer would be 4 bytes long). Then the payload is read into a second buffer with the just read length. When this buffer is full, one message has been read.

3.5.3 Security Concerns

Whichever solution is chosen, each solution has to provide code regarding Denial of Service (DoS) attacks. Wether a very big message length or large amounts of data without a delimiter are received, both can result in *Out of Memory Exceptions*.

Chapter 4

Server

Author: Nico Kratky

A server is a computer program that supplies clients with services. The term *server* often refers to the machine on which the program is running on. In this project the server has to accomplish several tasks. It has to read data from the sensor that is used, and distribute this data to clients. To do this it also has to manage clients and incoming connection requests.

4.1 Raspberry Pi 3 Model B

The Raspberry Pi is a small single-board computer, developed by the Raspberry Pi Foundation [38]. Originally it was created to teach children how to use computers and more importantly, how to program them. The biggest advantage of these mini-computers is the variety of extension possibilities. These extensions are so-called HAT's (Hardware Attached on Top) or Shields (which is a term that is more often used when talking about Arduinos). They are connected by using the on-board General-purpose input/output (GPIO) pins. They mostly provide additional hardware to extend the application possibilities and to achieve the desired goal. Further advantages are its relatively small footprint, low cost and wide-variety of available Linux distributions. Having these advantages was the decisive factor for choosing the Raspberry Pi. In this project the latest available version, which is the Raspberry Pi 3 Model B, was used. Specifications of this computer are listed in table 4.1.

SoC	Broadcom BCM2837
CPU	4x ARM Cortex-A53, 1.2GHz
GPU	Broadcom VideoCore IV
RAM	1GB LPDDR2 (900MHz)
Networking	10/100 Ethernet, 2.4GHz 802.11n wireless
Bluetooth	Bluetooth 4.1 Classic, Bluetooth LE
Storage	microSD
GPIO	40-pin header, populated
Ports	HDMI, 3.5 analogue audio-video jack, 4x USB 2.0, Ethernet, Camera Serial Interface (CSI), Display Serial Interface (DSI)

Table 4.1: Raspberry Pi 3 Model B specifications

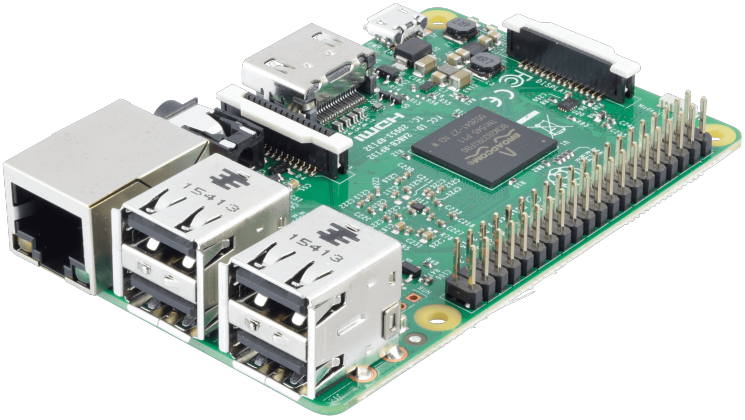


Figure 4.1: Raspberry Pi 3 Model B [40]

4.2 Raspberry Pi SenseHAT

The SenseHAT was made especially for the Astro Pi mission, where students could create and code projects, which were then run on the International Space Station by astronaut Tim Peake [1]. This board was chosen because it offers a wide variety of sensors and therefore offers many possibilities in terms of testing GRAMOC. Figure 4.2 on the following page shows a SenseHAT that is not attached to a Raspberry Pi.

The Raspberry Pi Sense HAT includes following sensors and inputs/outputs [39]:

- ST LSM9DS1 Inertial measurement sensor
  - 3D accelerometer
  - 3D gyroscope
  - 3D magnetometer
- ST LPS25H barometric pressure and temperature sensor
- ST HTS221 relative humidity and temperature sensor
- Alps SKRHABE010 5-button mini-joystick
- 8x8 RGB LED matrix

Although GRAMOCs main task is to charactarise steel belts using a magnetometer, a Raspberry Pi SenseHAT add-on board was used to get sensor data. It was dicided to use the now available accelerometer to perform measurements because it is easier to control the sensor data than by using the magnetometer. Different sensor values can be generated by simply moving around the Raspberry Pi with the attached SenseHAT.

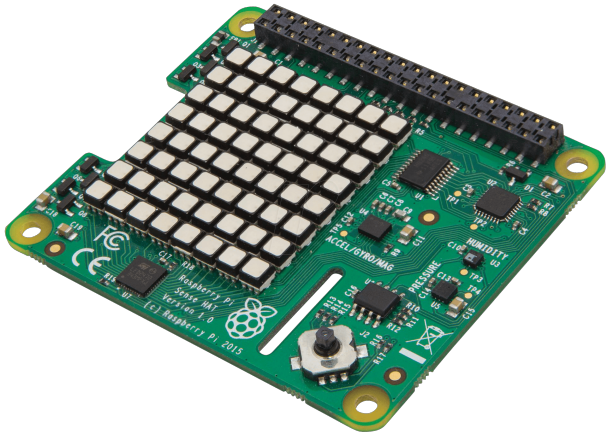


Figure 4.2: Raspberry Pi SenseHAT [41]

4.3 Implementation

The server program of GRAMOC is completely written in Python. This allows for great compatibility as Python comes preinstalled on many systems.

4.3.1 Programming Language

Python is a simple yet powerful modern programming language and supports both procedure-oriented as well as object-oriented programming. It was developed by Guido van Rossum at Centrum Wiskunde & Informatica (CWI) in the Netherlands in 1989 and first released in 1991 [53]. It was meant to be a successor to the ABC programming language. Python is a high-level language and therefore includes features such as automatic memory management.
Python is currently available in version 3.6.2. Nevertheless version 2.7 is still available as the Python Software Foundation announced that it will be supportet until 2020, effectivly making it an Long-term support version. Despite that, Python 3.6.2 was chosen for GRAMOC as the Foundation also encourages users to use the newest version if possible. Another reason for choosing the newer version is that GRAMOC does not have to be backwards compatible to any existing software.

4.4 Program Flow

As depicted in figure 4.3 on the next page, the server starts accepting new connections right after is has started. It then performs the handshake that is required by GSDEP (further explained in 3.1 on page 9). If this handshake is performed without errors, the server starts listening for data from this now connected client on a separate thread. While this thread is running it receives one message and checks if it is a command (see 3.3 on page 10). If it is, the message is interpreted and the appropriate function is executed. This thread is kept alive until the client disconnects or the server is shutdown by the user.

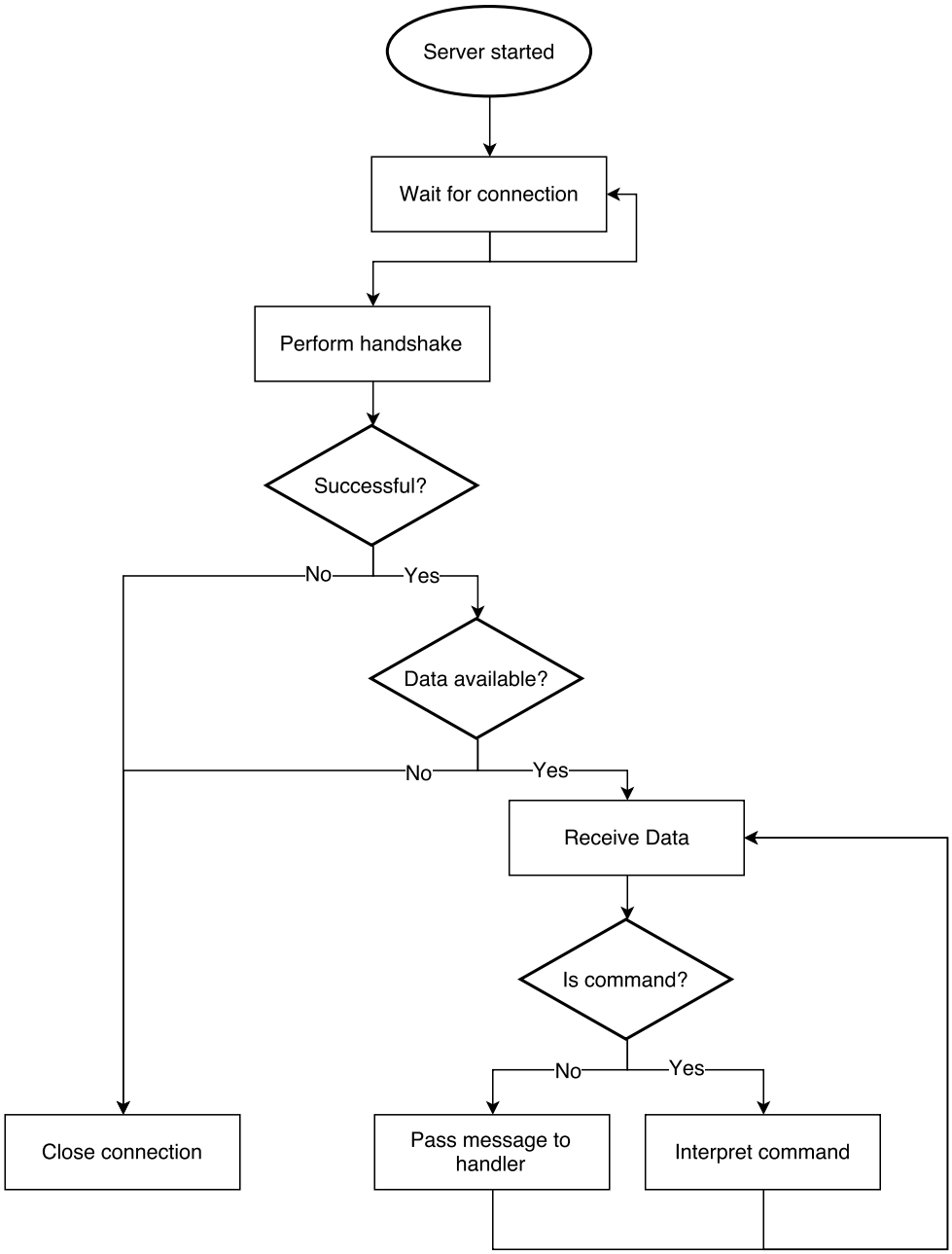


Figure 4.3: Flowchart of server program showing the procedure

Chapter 5

Android

Author: Nico Leidenfrost

Android is a linux distribution and is currently developed by the software giant Google. Android is an operating system with primary focus on mobile devices with a built-in touchscreen. The most popular examples, in which Android is used, would be smartphones and tablets. Since Android is an open source project, developers all over the world can contribute to it and even build their own Android system. Android programs are called *apps* which is the short version of application, these applications extend the basic functionality of an Android device.

5.1 History of Android

Android started as a startup Company under the name *Android Inc.*, founded in October 2003, in the US city Palo Alto, California. At first its purpose was to serve as an operating system for digital cameras, that would be more advanced than the standard in 2003. One Year later in 2004 they changed their goals to focus on mobile phones instead of cameras because the market declined their approach. Google became aware of this company and acquired it in July 2005 along with its founding members. The first working prototype of an Android smartphone looked quite similar to a BlackBerry phone of the time, because it had the BlackBerry typical QWERTY keyboard. They made two versions of this prototype, both without a touchscreen. In 2007, Apple introduced the iPhone which already featured a touchscreen. Since then Google also focused on mobile devices that included a touchscreen, but also stated that a touchscreen could never fully replace physical buttons. The first officially sold Android smartphone that was the HTC Dream, launched in 2008. Google continued to maintain the Android project and launched many updates which introduced new features or just fixed existing bugs. The developers of Android did choose a quite funny naming scheme for their major releases, namely the names of desserts. Each version starting with ongoing letters from the alphabet starting with *Cupcake* as the name for version 1.5. After that came version 1.6 called *Donut* up to 7.0 as *Nougat* and the latest version 8.0 as *Oreo*. Google explained this naming scheme with the statement, “Since these devices make our lives so sweet, each Android version is named after a dessert”.

5.2 Design

Material Design is Google’s visual design language that was first introduced in 2014. The goal was to develop a single underlying system that allows for a unified experience across all

kinds of devices. It tries to support visual elements with the characteristics of real materials, hence Material Design. These guidelines help the users to interact and quickly understand different kinds of User Interface (UI) elements by using familiar tactile attributes.
GRAMOC's Android app uses these design principles for the user interface as shown in figure 5.1

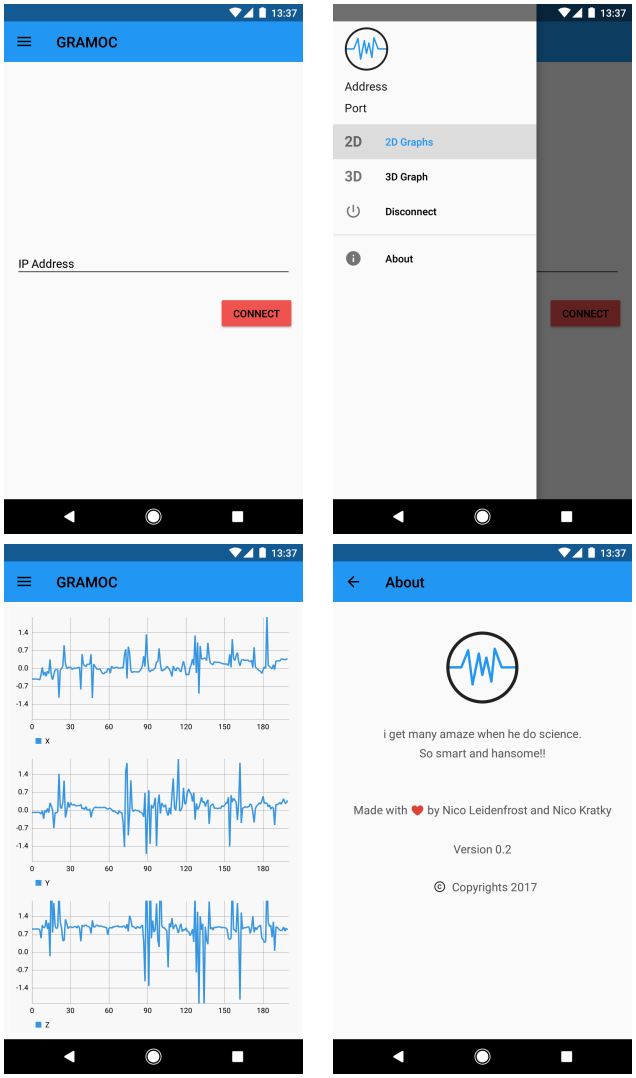


Figure 5.1: Screenshots of App

5.3 Overview of Android Application Development

Applications are often abbreviated as *apps*. To write Android apps one must use the Android software development kit (SDK). The variety of programming languages that can be used is not very broad, so a developer must choose one of a few options to build a native Android app.

5.3.1 Java

Java is the most commonly used programming language to develop Android applications. The majority of apps and libraries are written in Java. These apps are compiled to bytecode which then will be translated to native instruction by the Android Runtime (ART). ART is an application runtime environment that replaced its predecessor Dalvik, a process virtual machine developed to run Android applications. Java was chosen to be the programming language to build the Android application in this project because of the broad variety of third-party libraries and support available.

5.3.2 C/C++

With C or C++ code and the Android native development kit (NDK), a native library for Android, applications can get much better results in terms of performance. The reason behind this is that the C/C++ code does not need a virtual machine to be executed (i.e. the code runs natively), therefore the performance of an application that uses C or C++ code can be much higher than the performance of an app written only in Java. Important is to mention that an Android application should not be written entirely in C/C++ because all the UI still needs to be handled by the Android framework and that is only available in Java. Since the Java native interface (JNI) handles the interoperability of the two languages, which adds a lot of complexity to the application, it would be best to only write functions that require a high CPU performance in C or C++ code.

5.3.3 Go

"The Go programming language is an open source project developed by a team at Google and many contributors from the open source community" [51]. This programming language is supported although there are limitations to the application programming interfaces (API), therefore it was not considered a reasonable option for GRAMOC.

5.3.4 Kotlin

Kotlin is a modern and powerful language, which is officially supported since May 2017 and solved various issues addressed with Java (e.g. Null references). Kotlin is interoperable with Java which means an Android application can contain both Kotlin and Java code. Kotlin was considered to be used in this project, but the fact that the official support was only recently introduced leads to less available third-party libraries as in Java. This led to the decision that Java was the programming language of choice.

5.3.5 Runtime

A runtime is needed to convert high level code written in languages like Java, into CPU readable byte code. Compiled Java code can not run on any machine because the code is compiled to Java byte code, which a CPU can not interpret. To run this Java byte code, the Java virtual machine (JVM) is needed, because it translates the Java byte code into CPU readable byte code. In Android however the Java code is compiled to Java byte code, then compiled again to Dalvik byte code and then given to the runtime. Android implemented a runtime called *Dalvik*, which was later replaced by the *Android Runtime Environment* [20].

Dalvik

The Dalvik Virtual Machine (DVM) was the first runtime used in Android. The DVM was chosen instead of the JVM to be used in the early days of Android because it could perform better when running multiple apps at once. Both virtual machines are quite similar to each other, except the matter that the JVM is stack-based and the DVM is register based, which means the DVM needs less instructions, but these must be more complex. At first Android devices only had a small amount of memory available, therefore the just-in-time (JIT) compilation of the DVM was a perfect concept because it resulted in a small memory footprint. This was achieved by only translating and caching the chunks of byte code that were needed to execute the next few steps of an application. So instead of compiling the whole code of an application, only the parts needed were compiled.

Android Runtime Environment

The problem of having to few memory available was solved by the fact that the hardware improved over the years. The primary focus of application developers changed from most efficient way to use memory to improve performance and simultaneously decrease battery usage. With that in mind the Android Runtime Environment (ART), which now uses Ahead-of-Time (AOT) compilation, was created. First introduced in Android 4.4 and later replaced Dalvik completely in version 5.0, ART increased the performance of Android application by compiling the whole code at once at the time of the installation of the app. This method improved startup time, battery consumption and overall performance, because now the code does not need to be compiled during runtime.

5.4 Components

In order to build this Android application following Android components were used:

- Intent
- Toolbar
- Activity
- Service
- NavigationDrawer
- Threads

5.4.1 Intent

"An intent is an abstract description of an operation to be performed" [17]. It handles the execution of a specific action that it takes along with data to operate on. It is most used when launching a new activity.

5.4.2 Toolbar

This component is a widget from the Android *appcompat support library* and is persistent throughout the whole application. Most of the time it is referred to as app bar or action bar. Since this element is persistent, it will be used to perform important actions, like searching or navigating, but also to create space for identification of an application.

5.4.3 Activity

"An activity is a single, focused thing that the user can do." [16] Each application starts with launching an activity, therefore an activity handles the creation of a new window and loads all the User Interface (UI) elements. Activities are usually shown as a full-screen window, but also as a floating window or even be embedded inside of another activity by implementing an *ActivityGroup*. Inside the Android-system, activities are handled as a stack, this means every time a new activity is launched, the Android system puts that activity on the top of the stack and this activity becomes the running activity. The other activities in the stack are placed below the active activity and therefore remain inactive. An activity's lifecycle can be understood as depicted in figure 5.2.

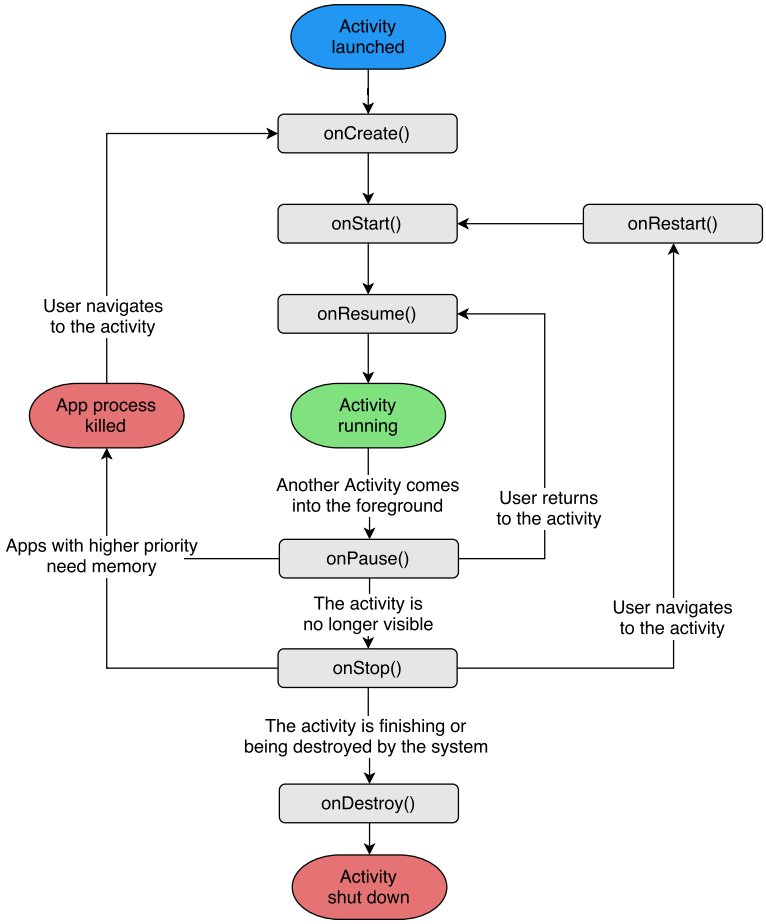


Figure 5.2: Flowchart showing the lifecycle of an Android-activity

5.4.4 Service

A service in Android can be compared to a daemon. It is a process that runs in the background and therefore there is no need for a visual interface. Once started, a service can persist in the background and is therefore not interrupted by switching applications. To use a service within another component, it must bind the service to enable interprocess communication (IPC). The most common application of a service is, handling network connections. The lifecycle is defined as depicted in figure 5.3 on the following page.

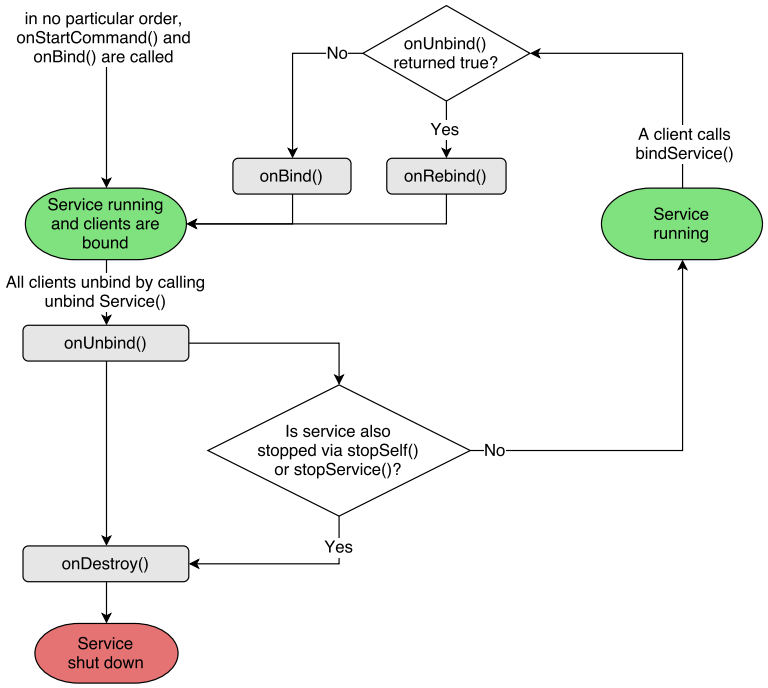


Figure 5.3: Flowchart showing the lifecycle of an Android-service

5.4.5 NavigationDrawer

To navigate between the activities and views a navigation drawer was implemented. A navigation drawer is a panel which is pulled from the left border of the screen to approximately 3 quarters of the screen width. It contains a header where general information is displayed and a body which contains various navigation items. The navigation drawer is included in the material design pattern, which is often used in Android application development, so most Android apps provide a navigation drawer.

5.4.6 Threads

When an Android application component is launched and it is the first component of an application, Android will start a new Linux process. If this application is already running inside a process and a new component is launched or an action is preformed, Android will execute this task in the main-thread of the application. Unless it is explicitly stated to execute the operation in a new thread within this process. When working with threads in Android two essential rules must be followed [18] :

1. Do not block the UI thread
2. Do not access the Android UI toolkit from outside the UI thread

The reasons behind this two rules are quite simple. The point why the thread that contains the user interface should never be blocked is simply because then, no events could be dispatched, including events that update the UI itself. This would mean that the application could not give any information to the user unless the operation which is blocking the thread has finished. This is really bad, because the user could think that the application stopped working. Accessing the UI toolkit from a thread other than the UI thread is also a bad idea, since the UI toolkit is not thread-safe. This means if multiple threads would access UI elements, race conditions could happen and therefore cause errors within the application. To

avoid such errors Android implemented different ways how to execute tasks asynchronously in Android:

Extended Threads

The first way is to implement a subclass of the Java *Thread* class. If this solution is chosen a developer must override the *run* method of the superclass. If the way of how a thread handles certain situations needs to be changed or new functionalities needs to be added, a developer should choose this method, otherwise the Runnable interface should be implemented.

Runnable interface

Another method to accomplish asynchronous behaviour would be to implement the *Runnable* interface when creating a class. To execute the tasks, an instance of this class needs to be given to the thread which should execute the tasks. This way is preferred to use when running tasks which does not need modified thread behaviour. When tasks from a runnable class are executed there is no need to launch a new thread for each task, instead they can be executed on various threads.

AsyncTasks

AsyncTasks are implemented to move work to the background and then update the UI accordingly. An AsyncTask is defined to execute blocking operations, therefore there will be only one active AsyncTask at the same time. In order to perform an AsyncTask a cycle of four tasks is executed, with following steps:

1. **onPreExecute**: executed on the UI thread before the task is executed.
2. **doInBackground**: executed on the background thread, here the background tasks are executed.
3. **onProgressUpdate**: executed on the UI thread every time when *publishProgress* is called in the background thread.
4. **onPostExecute**: executed on the UI thread when the background tasks are finished.

5.4.7 Libraries

The Android client was implemented using a small number of libraries:

- **Android SDK**

  The standard libraries included in the Android platform itself [19].
- **GramocAlgorithm-client**

  The Java implementation of the GSDEP client developed along with this project [27].
- **MPAndroidChart**

  An easy to use but also powerful open source 2 dimensional chart library for Android [24].
- **android-about-page**

  This library allows to simply create an about page for your Android application [43].

5.5 Implementation

The entry point of this Android application is called the *MainActivity*. When this Activity starts a background service is additionally started, which is basically a wrapper for the GSDEP client, therefore it handles all the networking related tasks within the app. The service will be bound to the active activity, so every time another activity is launched the service will be unbound by the current activity and newly bound by the starting one. The *MainActivity's* goal is to give the user an easy way to connect to the server. Once the application successfully connected to the server, a new activity responsible for plotting the received sensor data will be launched, whether the 2-dimensional or the 3- dimensional plotting activity is launched depends on the selection made in the *NavigationDrawer*, by default the 2-dimensional plotting activity will be launched. When the 2-dimensional plotting activity is launched the networking service will be bound and three *LineCharts* contained within the library *MPAndroidChart* will be created and properly set up. After these tasks are finished and the activity is ready to receive data, the server will be notified. Now each data set received will be added to the data buffer of the respective chart. If the buffer of a chart is full, the values at the end will be truncated until there is enough space to add the new values. The 3-dimensional plotting activity however was not implemented at all, since the Android application was discontinued because of problems that appeared during the development of the 2-dimensional activity (see chapter 6 on page 26).

Part II

Lessons Learned

Chapter 6

Problems

Author: Nico Leidenfrost

After extensive testing as described in section 6.5 on the following page, it was decided that this approach will not lead to a successful project outcome. This decision was made while taking several factors into consideration.

6.1 Android

Android is a great platform to create simple and even complex application systems that does not rely on heavy performance. Since the key element in this project is the ability to display the sensor data in real-time, the Android development was discontinued due to the performance issues that come with it.

6.2 Software limitations

Android is designed to render the UI with 60 frames per second (fps), which results in redrawing frames every 16 milliseconds at best. The task of drawing frames will be executed by the main thread along with many other operations like system events, input events, application callbacks and so on. The system tries to update the screen every 16 milliseconds, if however other operations than the redrawing of the screen are pulled from the work queue when trying to update the screen, these frames will simply be dropped and users will experience lacks of smoothness while using the application. To be sure about how much milliseconds the rendering of one packet takes the time was stopped. The results showed that it took up to about 50 to 60 milliseconds to render one update. These measurements were the prime factor that led to the decision to discontinue the work on Android.

6.3 Plotting Libraries

There are a lot of freely available plotting and charting libraries to use in Android development. Unfortunately most of them do not meet the requirements to be used in this project. There are many good libraries to plot 2 dimensional charts like pie charts or bar charts, but there is a lack of libraries that can display scientific plots (e.g. surface plots). The libraries that would meet all the requirements however are not originally designed to be used in Android development and therefore work only in specific versions of Android or do not work at all because they rely on components that are not available in Android.

6.4 Networking

The first approach of transmitting sensor data to the plotting application was the GSDEP protocol (see chapter 3 on page 9). The goal of this protocol was to send sensor data reliable to clients such as the GRAMOC Android application. As it was meant to send data reliable, the TCP/IP stack was used as underlying technology [35, 36]. This approach worked well with small amounts of data, but since GRAMOC requires to send an enormous amount of data this approach failed, because the data was stuck in a buffer and could not be displayed in real-time.

6.5 Tests

The two main test factors that lead to discontinuing the Android development in conjunction with the GSDEP protocol are:

- The time of one chart update
- The time to empty the data buffer

6.5.1 Update Test

This test was used to determine the redraw or update performance of the Android application. To measure the time that one full update took, the timestamps of the start and end of an update circle were taken. The start time was then subtracted form the end time and the result was the time of one update cycle. These durations were measured in milliseconds. Results showed that one update took 50 to 60 milliseconds, that corresponds to 16 to 20 frames per second (fps). The goal was to achieve a smooth experience with about 50 to 60 fps and thats far away from what the test results revealed.

6.5.2 Buffer Test

During the testing phase it occurred that the displayed data values did not match the given data input at the time. Also the application kept displaying new data after the sensor stopped transmitting. These anomalies indicated that the huge amounts of data could not be plotted in real-time and ended up in a buffer. To measure the time it took to fully empty the buffer of the not drawn data, the sensor was constantly sending data for about five minutes and then stopped. The time between the sensor stopped transmitting data and the last chart update was taken and evaluated. The results showed that the application stopped updating the chart roughly about two minutes after the sensor stopped sending new data. Based on this test the whole networking stack was rebuild on top of UDP to achieve the desired speed.

Chapter 7

Résumé

Author: Nico Kratky

After researching alternatives that still fit the purpose of GRAMOC, a meeting with the client was arranged to discuss these alternatives. This meeting resulted in new goals and expected results. This new project specification now includes a web application instead of the mobile Android application.

7.1 Advantages

The switch to developing a web application still offers a few advantages that were not existing while focusing on a Android application. This includes the flexibility that a web application can run on basically any device the end user wishes. As of today many devices support network connections and can run a web browser. Another advantage is that JavaScript offers a tremendous amount of third party libraries, especially plotting libraries. A few of these libraries even support scientific plotting, like for example VTK, the visualisation toolkit that is used by ParaView [25].

7.2 Disadvantages

The change of specifications also brings some disadvantages with it. For example the whole networking stack has to be rewritten because raw TCP streams are not supported in web environments. They were replaced be the WebSocket technology [11]. Also a new third-party plotting library has to be chosen and read up on.

Part III

Implementation Phase 2

Chapter 8

Software Architecture

Author: Nico Kratky

After studying lots of literature about real-time systems, a new fundamental software architecture was developed. The main principle of this is to seperate different tasks into seperate processes. This makes use of the fact that the processed data is sent to the client over the internet anyways. Therefore the process that handles data storage also acts as a client. This leads to increased protability, and more important, increased performance. Another perfomance increasing change is the use of C++ instead of Python as programming language. This sofware stack is depicted in figure 8.1 and its components are further introduced and discussed in the following chapters.

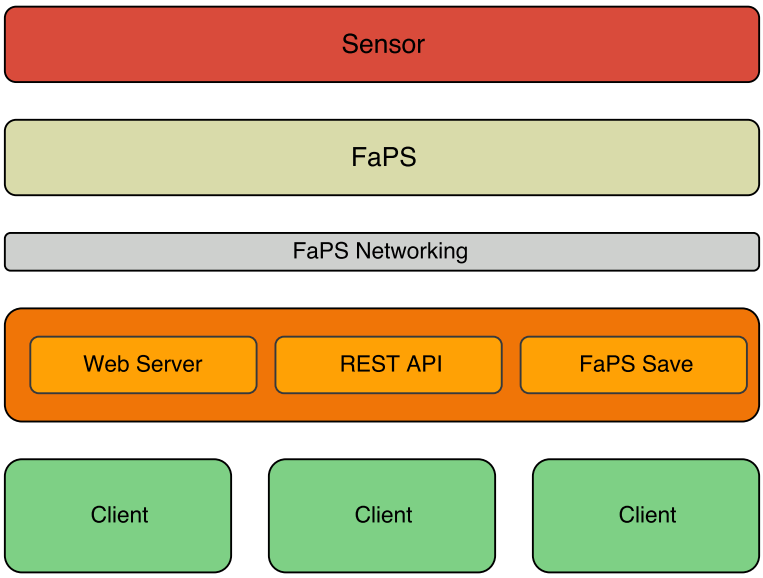


Figure 8.1: GRAMOC Software Architecture Diagram

Chapter 9

FaPS Networking

Author: Nico Kratky

FaPS Networking is a custom UDP-based library that is mainly used for communication between FaPS (see chapter 10 on page 35) and different kinds of client, eg. FaPS-save (see chapter 11 on page 38) and the Node.js server (see section 13 on page 50).

9.1 TCP vs. UDP in Real Time Environments

Both TCP and UDP are members of the transport layer of the internet protocol suite, commonly known as the TCP/IP stack [4].

9.1.1 Connection-Oriented and Connectionless Protocols

There are two groups of protocols. The ones that require setting up a logical connection before data can be exchanged and the ones that don't require a link between the two communication partners. They are also called *connection-oriented transport services* (COTS) and *connectionless transport services* (CLTS) [49]. The advantages and disadvantes of both are important to know when choosing one of these types of protocols. The main feature of COTS is that it is reliable, meaning that the protocol will ensure that sent messages are received reliably and in order. As discussed in chapter 2 on page 4 it is not that big of a deal for real-time applications if packets are dropped, therefor a CLTS was chosen.

9.1.2 Perfomance

As perfomance is a critical component of real-time applications, some research had to be conducted to get the best possible result.
This research included the comparison of the packet headers. As seen in figures 9.1 on the next page and 9.2 on the following page it is clearly visible that the TCP header is much larger the the UDP header. In fact, TCP requires 20 bytes and UDP requires only 8 bytes for the header information.

Bit #	0 7	8 15	16 23	24 31
0	source port		destination port	
32	sequence number			
64	acknowledgment number			
96	offset	res	flags	window size
128	checksum		urgent pointer	
160 ...	option (0 - 40 bytes)			

Figure 9.1: Header found in a TCP packet

Bit #	0 7	8 15	16 23	24 31
0	source port		destination port	
32	length		checksum	

Figure 9.2: Header found in a UDP datagram

TCP also has a built-in feedback mechanism which checks if all packets are received by the communication partner and if they are received in order. This mechanisms not only produces a lot of overhead, but also is the data most likely outdated when it is resent. Therefore it is not a problem if such packets are dropped.
Considering all these factors the decision to use UDP over TCP was made. TCP is a great protocol for example sending large files where it is necessary that all bytes come in order and reliably. If one byte is missing then the whole file would be corrupt. UDP however is a lot better for transferring time critical information as it produces less overhead [7].

9.2 Handling Connections

As UDP is a connectionless protocols, neither does it know if the other end of the communacation is ready to receive data nor if it is even existing. Therefor a way of handling connections using UDP had to implemented.
The core of this implementation is a map. A map is a associative container that is available through C++'s STL (Standard Template Library). This maps contains all clients as keys, and the associated timestamps of the last received keepalive message as values.
After starting the server, two threads are started. The first one handles all incoming messages. If a received message is a keepalive message the timestamp of the client that sent this message is update to the current time. The second thread monitors these timestamps. If the difference between the current timestamp and the stored timestamp of a client exceeds 1.5 seconds, the client is declared disconnected and removed from the list.
Also when a client is instantiaed a thread is started to handle the keepalive messages. The only task of this thread is to send a keepalive message to the server and then wait one second. All of this is done in a loop that only finishes when the clients deconstructor is called, thus disconnects.

9.3 Handshake

The handshake procedure is a very important part of connecting. This makes sure that the server is notified whenever a client is waiting to connect.

When the server receives a connection request (see table 9.1 on the next page), it has to check if it can accept further clients. This limit is set as a static constant in the `Server` class. The default value is 8. If this check is successful the server sends an acknowledgement message to the client. If the check fails, the client will receive a connection refused message. In this implementation the clients connect call will block until it is connected. This is done by a loop that will be exited once the server sends an acknowledgement. In between the connection attempts one second is waited.

These two procedures can be both seen in the code listings (9.1 and 9.2) and the sequence diagram (9.3 on the following page) below.

```
1  void Server::shake_hands(boost::asio::ip::udp::endpoint& remote) {
2      if (endpoints_.size() < MAX_CLIENTS_) {
3          // client limit not reached, new client can connect
4
5          send(control_messages["ACKNOWLEDGE"], remote);
6          // set the keepalive timestamp to now
7          endpoints_[remote] = std::chrono::system_clock::now();
8      }
9      else {
10         // too many clients connected, send connection refused message
11
12         send(control_messages["CONNECTION_REFUSED"], remote);
13     }
14 }
```

Listing 9.1: Server handshake method

```
1  void Client::connect() {
2      while (!connected) {
3          // retry connecting, until connection is accepted by server
4
5          send(control_messages["CONNECTION_REQUEST"]);
6
7          std::string reply;
8          receive(reply);
9
10         if (reply.compare(control_messages["ACKNOWLEDGE"]) == 0) {
11             // connection established, start sending keepalive messages
12
13             connected = true;
14
15             std::thread t_keepalive{&Client::keepalive, this};
16             t_keepalive.detach();
17         }
18         else {
19             // connection refused, try again in TIMOUT_ seconds
20             std::this_thread::sleep_for(TIMEOUT_);
21         }
22     }
23 }
```

Listing 9.2: Client handshake method

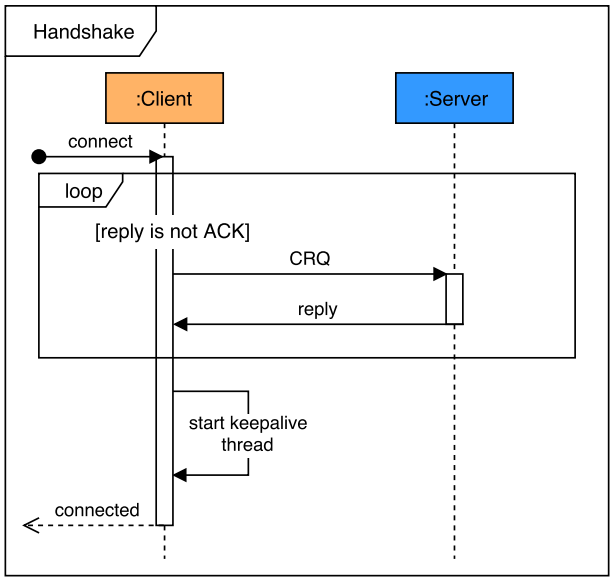


Figure 9.3: Handshake performed when a client tries to connect to server

9.4 Control Messages

Control messages are a special type of message that require further action to be taken. These messages are especially important during the handshake. The supported messages and their meaning and depicted in table 9.1.

Message	Sent to	Meaning
CRQ	server	Tells the server that a new client is waiting for the connection procedure
ACK	client	Tells the client that the connection is acknowledged
CRF	client	Tells the client that the server can not accept the connection
KAV	server	Tells the server that the client is still alive and wants to stay connected

Table 9.1: Commands sent by one of the connection partners and what they do

Chapter 10

Filtering and Preprocessing System

Author: Nico Kratky

FaPS, which stands for Filtering and Preprocessing System, is the main application of the GRAMOC backend. It is an application that reads digital sensor output, preprocesses it and forwards it to another process to distribute it to the final clients. It can also start a seperate process to save the sensor data to HDF5 files [52]. The data is also dynamically analysed using regression analysis.
The preprocessing consists of regression analysis (see 14 on page 55 for further details.)

10.1 Command Line Interface

As FaPS is a command line program, arguments that are passed to it have to be parsed. This is done by utilising Boosts Program_options [37]. This module allows easy parsing and exception handling.
The arguments that FaPS can handle are depicted in table 10.1 on the next page.

10.2 Data Storage

To be able to offer a possibility for further data inspection all received data is saved to HDF5 files. This is done when both -s and -f parameters are supplied. If either of those is supplied solely, a error is printed and FaPS exits. The data is saved by a seperate process that acts as a client to FaPS. This seperate program is described in chapter 11 on page 38.

10.3 Data Processing

As the sensor data is received in a raw binary format, it has to be processed in order to use it in a simple way.
One datagram that is received from the sensor is 1200 bytes long and consists of 600 shorts. This data is in network byte order. By convention, network byte order is always big-endian, which means that the most significant bit is placed first [21]. The conversion of two bytes in network byte order to a short is shown in code listing 10.1.

```
1 short s = (((short) bytes[i]) << 8) | bytes[i+1];
```

Listing 10.1: Conversion of two bytes to a short

Flag	Argument	Default value	Description
-h, --help			Outputs the usage information
--version			Prints version information
-l, --loglevel	*loglevel*	info	Information granularity during runtime
--ip	*ip_ adress*	127.0.0.1	IP address to which FaPS will connect to read sensor data
--in-port	*port number*	9760	Port to which FaPS will connect to read sensor data
--out-port	*port number*	1337	Port for the started UDP Server
-s, --save	*path to faps-save*		Path to the faps-save executable
-f, --filename	*filename*		Filename to which the data will be saved
-p, --predict			Enter predict mode
-c, --config	*filename*		Regression config file (only needed when -p is supplied)
-d, --driver	*Driver ID*	1	Only needed when -p is not supplied

Table 10.1: Flags that can be set, which arguments they take, their default values and what they do or change

10.4 Data Serialisation

In order to send data so that the other end can interpret the message it has to be packed into a common format. To do this JSON is used [5]. It is a language- and platform-independent data serialisation format originally specified by Douglas Crockford in the early 2000s. JSON uses text a human-readable form and stores data in key-value pairs. Originally it was derived from JavaScript but as of today many programming languages include methods and functions to en- and decode JSON. This integration in JavaScript is the biggest advantage for GRAMOC and ultimately led to the decision to use it in this project.

10.5 Data Distribution

Once preprocessing and serialization is finished, the received sensor data has to be distributed to connected clients. This is a typical use case for inter-process communication.

10.5.1 Unix Domain Sockets

Unix domain sockets are a way of communicating between a client and a server that are on the same host. There are two types of sockets available: stream sockets (compareable to TCP) and datagram sockets (compareable to UDP). Although this solutions seams ideal for this, it was quickly decided against it. The reason for this decisions is that Node.js does not support datagram unix domain sockets anymore and real time application usually make use of datagrams. Therefore regular UDP was used.

10.5.2 Solution

This problem led to the decision to use a custom UDP protocol for data transfer. This custom protocol is discussed in chapter 9 on page 31.

10.6 Storing Data Between Measurements

In order to continue the regression analysis after FaPS has been stopped and restarted, a way of storing the regression analysis data had to be developed. This was achieved by writing the data of the `MLR` class to a file. This process includes serialising the M matrix and V vector (explained in section 14.1.3 on page 58) to arrays. The matrix is serialised by creating a two dimensional array where the inner arrays represent the matrix rows. So a $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ matrix would become `[[1, 2, 3], [4, 5, 6], [7, 8, 9]]`.

Chapter 11

Saving Sensor Data

Author: Nico Kratky

In order to be able to look up recent sensor readouts, all data that is received from the sensor has to be saved in a persistent way. This chapter introduces the program that was implemented to solve this task.

11.1 File Type

The Hierarchical Data Format (also called HDF) is a open source data format that allows storing large amounts of heterogenous data. Heterogenous in this context means that each entry in a dataset can itself be a complex type. This format is often used in scientific fields because it is a very flexible format.
There are two very important terms that are used when dealing with HDF files. Groups and Datasets.

11.1.1 Groups

A group is a container that can contain other groups and/or datasets. Datasets are often stored in a group, but that does not mean that this has to be.

11.1.2 Datasets

A dataset is a the actual data that is contained. Datasets can contain multidimensional, complex and heterogenous data.

11.1.3 Metadata

It is possible to associate every file, group or dataset with metadata. This makes HDF self-describing.

11.1.4 HDF and HDF5

Although these two formats share a similar name, they are two completely different file formats.
The biggest difference is that only HDF5 uses a true hierarchical structure similar to the UNIX file system. Every object has to belong to one group. Only one group does not have a parent group, the so-called root group. HDF uses a pseudo-flat structure using Vgroups. Objects do not necessarily have to belong to a group and there is also no root group.

11.2 Structure

In this project the structure of the HDF file are kept rather simple.
Every file represents a measuring process. Files do not have any groups. Datasets are identified by a number that represents a timestamp as microseconds since 1.1.1970 (also known as Unix Time or epoch).

11.2.1 Example

A dataset recorded on 1.1.2018 00:00:00 would have the identifier 1514764800000000. This dataset would contain a 6 dimensional integer array where each one would contain 100 values.

11.3 C++ Library

Although the HDF Group, which are the maintainers of the HDF project, offers a C++ API, it was decided against this library as it is quite complex. Instead a library developed by the Blue Brain Project, called HighFive was used [50]. This library allows for easy creation and modification of HDF5 files.

11.4 Implementation

To save the sensor data to HDF5 files a seperate command-line program was developed. This application acts as another client to FaPS, which transmitts the sensor data over the network anyways. This approach yields two major advantages: Perfomance and hidden complexity.
Perfomance is increased as FaPS can finish one iteration of its main loop faster as it does not have to save the data and can carry on receiving data from the sensor.

11.4.1 Command Line Interface

As described in chapter 10 on page 35, Boosts Program_options were used in this project to parse command line arguments. This program has only a few parameters that are depicted in table 11.1.

Flag	**Argument**	**Default value**	**Description**
-h, --help			Outputs the usage information
--version			Prints version information
-l, --loglevel	*loglevel*	info	Information granularity during runtime
--ip	*ip_adress*	127.0.0.1	IP address of FaPS
--port	*port number*	9760	Port of FaPS
-f, --filename	*filename*		Filename to which the data will be saved
-i, --interval	*seconds*	1	Compression level

Table 11.1: Flags that can be set, which arguments they take, their default values and what they do or change

11.4.2 Compression

As GRAMOC handles a lot of sensor data a possibility to compress the datasets had to be implemented. This is necessary because when measurements are perfomed over an extended period of time, an enormous amount of datasets will be recorded. Therefore readouts in a specified time frame are consolidated and stored in a mutual dataset. This dataset's ID is set to the timestamp of the first sensor readout that is stored in this dataset. The timespan in which the data will be consolidated into one dataset can be set via the `--interval` command line flag.

Chapter 12

Webapp

Author: Nico Leidenfrost

After the conclusion that an Android application would not satisfy all the requirements of GRAMOC, the decision to build a Web application was made. A so called *Webapp* is an application that runs inside a web browser (e.g. Google Chrome) and is usually provided by a web server. After a user connects to the web server, the user will get application files and associated data. This enables the ability of a Webapp to be platform independent.

12.1 Framework

To create a modern Webapp a developer should choose a framework to build the web application. A software framework can be classified as a huge software library. It provides basic functionality like rendering content or routing between views in the context of a web framework. The biggest benefit of a web framework is that the developer does not have to reinvent the wheel, because a framework already implements the basic functionalities. Also the majority of frameworks out there are open source, which means thousands of people can help to enhance the project and also resolve issues. Therefore the user gains a solid code base which is efficient, secure and usually well documented.

12.2 Vue.js

In the case of GRAMOC, a framework called *Vue.js* was used because of the convenience compared to other big frameworks and the "simplicity and ease of use", as stated in a blog post published by the Frontend DC Lead of GitLab [15, 44, 59]. Another factor in choosing Vue.js as the Framework for the GRAMOC web application was the performance compared to other big Frameworks. A Benchmark application to measure the performance of the Frameworks was created by Stefan Krause and is available on GitHub [28]. The results are depicted below in figures 12.1 on the following page and 12.2 on the next page.

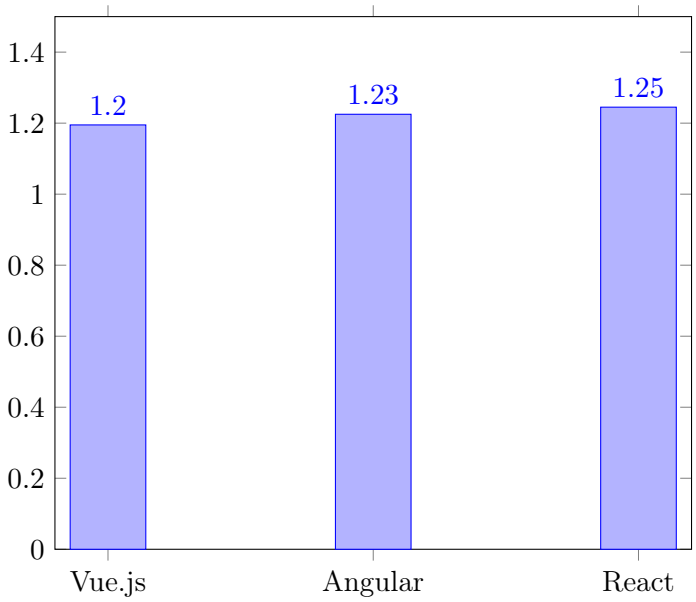


Figure 12.1: Benchmark results: average slowdown in milliseconds

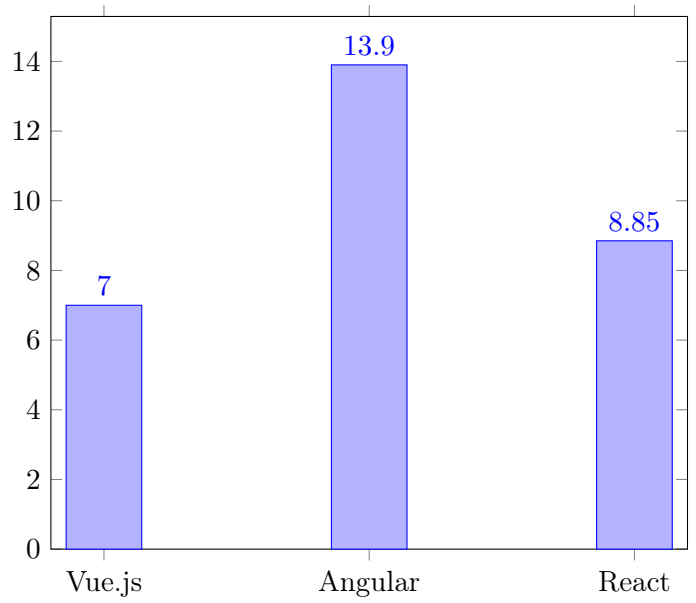


Figure 12.2: Benchmark results: average memory usage when running in MB

Although the performance of all three competitors is almost equal, Vue.js is slightly ahead of the others. All these results lead to the decision that Vue.js will be used as the Web Framework in GRAMOC.
In order to use Vue.js it is recommended by the developers to use *webpack* as module bundler and *Babel* as JavaScript compiler, this can be done by using the vue-cli tool [2, 54, 56]. A guide on how to create new Vue.js applications with this tool is available on the GitHub page of the vue-cli tool [54].

12.2.1 webpack

webpack is a module bundler for modern JavaScript applications, that builds a dependence graph which includes every module needed to run the application. It packages all the needed modules into several bundles which will be commonly served as static asserts.

12.2.2 Babel

Babel is a JavaScript compiler that is capable of converting up to date JavaScript code into correct JavaScript code of a prior version. This is especially useful when a developer needs to work in an environment where the most recent version of JavaScript is not supported, but still wants to be able to write up to date JavaScript code.

12.2.3 Vue Instance

Every Vue.js application begins with the initialisation of a Vue instance, this is done by calling the Vue function. In most of the cases the Vue instance is bound to an element within the DOM, which usually is a div element with the id *app*. Since this part needs to be done in Javascript, most of the time there is also a *App* component imported, which will be the so to say *main component* of the application. This can be done by writing following code:

```
1 new Vue({
2     el: '#app',
3     template: '<App/>',
4     components: { App }
5 })
```

Listing 12.1: Creating a Vue instance

12.2.4 Components

Components in Vue.js are very important and powerful because with this feature it is possible to create custom elements that can be reused within the application. These components contain three sections, first the template, which is basically the HTML part of a component, second the script section, where all the JavaScript code is written and at last the style section, to add custom styling to the component. These components are then used like ordinary HTML elements in another template section or in the HTML code itself. There are two ways to implement components, either the *Vue.component* function has to be called to create a new component object, or all the components are separated into distinct *.vue* files. The latter method is preferred, especially in larger projects like GRAMOC, because the code is much easier to maintain and it also solves some problems. For example the scoped CSS styling is only possible when using single file components. In order to use components, a build tool like Webpack or Browserify has to be used. The two ways of using components are shown below.

```
<div id="app">
  <hello-comp></hello-comp>
</div>

new Vue({
  el: '#app'
})

Vue.component('hello-comp', {
  template: '<div>{{msg}}</div>',
  data: {
    msg: 'Hello World'
  }
})
```

Listing 12.2: Creating a Vue instance and adding a component to it

```
<template>
  <div> <p>{{msg}}</p> </div>
</template>

<script>
  export default {
    name: 'name',
    data () {
      return { msg: 'Hello World' }
    }
  }
</script>

<style scoped>
  p { color: red; }
</style>
```

Listing 12.3: Example for a simple single file component

12.2.5 Router

Vue.js itself only supports single-page applications, but the Vue.js team is maintaining a few core libraries that work in direct correlation to the base core system [55]. This library enables the creation of multi-page applications, through binding Vue.js components to the individual routes. This is quite beneficial to this project, since GRAMOC supports a few distinct core features, that are best displayed within a multi-page application. A router can be created as shown in listing 12.4.

```
import Vue from 'vue'
import Router from 'vue-router'
import Home from '@/components/Home'

Vue.use(Router)

export default new Router({
    mode: 'history',
    routes: [
        {
            path: '/',
            name: 'Home',
            component: Home
        }
    ]
})
```

Listing 12.4: Creating a router instance with one *Home* route

12.2.6 WebSockets

WebSockets are used to communicate and rapidly sending data between the Webapp and the web server. In GRAMOC a library called socket.io was chosen because of their focus on reliable real-time communication (see section 13.6 on page 52). In order to use socket.io within a Vue.js application the npm package Vue-Socket.io can be used [45]. With this library, a socket object can be created and attached to the Vue instance that needs to use the socket connection.

12.3 Plotly

To visualise the data received from the sensor a graphing library called *Plotly*, more specific the open source JavaScript library plotly.js is used [33, 34]. Plotly is build on top of state of the art JavaScript libraries like *D3.js* and *stackgl* [3, 48]. The library offers a broad variety of two and three dimensional charts in the categories statistical, financial, scientific and more. In GRAMOC one of the chosen graphing libraries is Plotly, because of the capability to easily create custom and dynamic charts.

12.3.1 Line Chart

To visualise the received sensor data in 2 dimensions, the line chart provided by Plotly was implemented. This particular type of line chart was used to depict the saved sensor data. For the real-time visualisation a D3.js line chart was implemented instead of a Plotly line chart (see below subsection 12.4.1 on the following page). If the data does not need to be depicted in real-time, Plotly has some advantages over D3.js. Plotly provides a rich set of options to configure the behaviour and style of a chart. Plotly also provides some events, to give the user the ability to interact with the chart. These events cover interactions like clicking, dragging, zooming, scrolling and more. An example is shown in figure 12.3

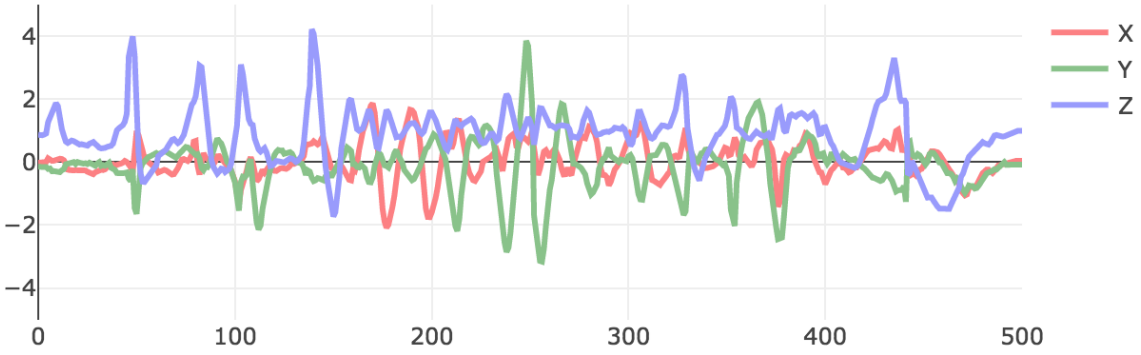


Figure 12.3: line chart used to visualise sensor data in 2D and provide an interactive way to analyse the data

In GRAMOC the Plotly line chart was configured to hold three traces, one for each axis as shown in figure 12.3. The main focus of the Plotly library is to give the user a convenient way to create interactive charts and not to provide high performance real-time charts. These aspects fit best into the archive page of GRAMOC, where the source of the data are static files and not real-time streams of data.

12.4 D3.js

The name *D3* is an abbreviation of Data-Driven Documents, that is a very precise description of what this framework has to offer, namely the manipulation of documents based on data. The goal of D3.js is not to visualise data on documents and at the same time be able to handle all the things around the objects as well as implementing every imaginable feature, it is build to be perfect at one thing: “efficient manipulation of documents based on data”, as stated on their website [3]. Since one of the key features of GRAMOC is real-time representation of the data that is provided by a sensor, this framework was chosen to be used within the real-time display. D3.js has advantages as well as a few disadvantages.

Probably the biggest advantage is that this framework is very lightweight. This means there is only a minimal overhead and therefore it is very fast compared to other frameworks or libraries like Plotly (see section 12.3 on the previous page). A disadvantage of D3.js would definitely be the lack of convenient high-level functions to create or modify objects. This lack of high-level functions leads to bigger development cost, because to implement simple features it is necessary to write a lot of code compared to high-level solutions. In most high-level frameworks a developer just needs to call one function to create a chart and another one to add data to it. High-level functions are great to begin with, but to squeeze every last bit of performance out of the code, low-level functions are much better. Also to understand what is happening behind the code, low-level functions would be superior, because the developer has to do nearly every step on his own and not just call a magic function that does a lot of processing on its own. Therefore the lack of high-level functions could be seen as an advantage, because programmers that use low-level functions instead of high-level functions often have more knowledge about how the system works and that is clearly a good thing.

12.4.1 Line Chart

In GRAMOC D3.js was used to create a simple line chart to be able to visualise scientific sensor data in real-time. The chart is based on the line chart provided by Plotly, but with the distinction that the D3.js chart can render the given data faster, and therefore sustain the real-time support of the application. The design should be similar to the Plotly line chart to maintain a uniform design within the application. The chart with example data is shown below in figure 12.4.

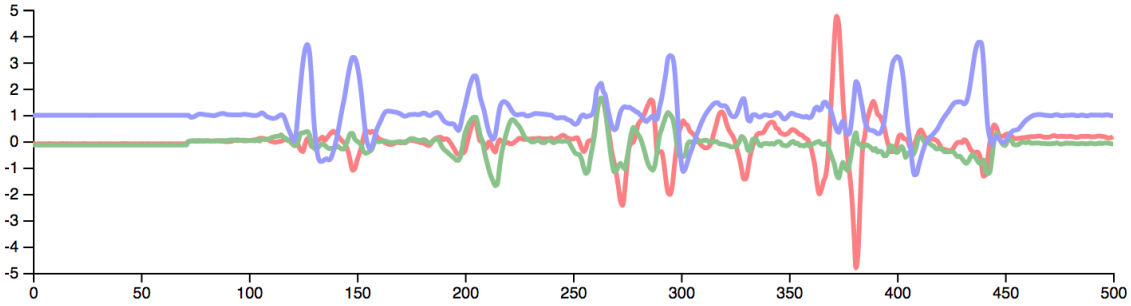


Figure 12.4: line chart used to visualise sensor data in 2D and be able to render in real-time

12.5 Implementation

As shown in figure 12.7 on page 49 the server asynchronously tries to connect with the UDP socket and starts listening for incoming connections on a specified port. The server keeps retrying to connect to the UDP socket until a connection is established. Without this connection no live data from the sensor can be forwarded to the web application. If a client connects on the before specified port the connection will be immediately upgraded to a socket connection and the web application will be served to the user. When the Webapp launches it will display the Home page, then the user can navigate to the 2D, the Archive or the About page, through the navigation bar at the top.

12.5.1 2D Page

If the user navigates to the 2D page, a line chart created with D3.js will be loaded. This chart consists of 3 traces, one for each axis of the sensor. The second chart displayed is a density chart created with HTML5 Canvas. This chart is represented by an ellipse, which is bent or stretched according to the received sensor data. Both these charts will be initialised and then the client emits a message to the server to start receiving the sensor data. This data will be used to update the charts accordingly. This page is responsible for visualising the sensor data in real time and therefore its components are optimised to provide the necessary performance. The exact procedure is shown in figure 12.5.

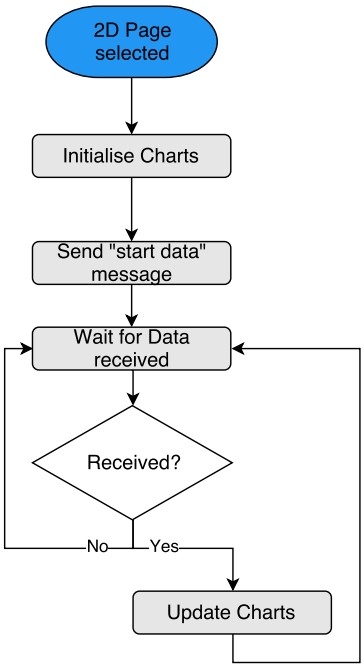


Figure 12.5: Flowchart of the procedure when 2D page is selected

12.5.2 Archive Page

The Archive page displays a line chart similar to the line chart on the 2D page, but with the focus on convenience rather than real-time performance. Therefore, Plotly is chosen to be used within this component. Along with the chart, a form will be available which is responsible for requesting the already recorded sensor data selected by the user. The execution flow of this component is depicted in figure 12.6 on the next page.

12.5.3 About Page

The About page is simply a static page that displays a few informations about the project.

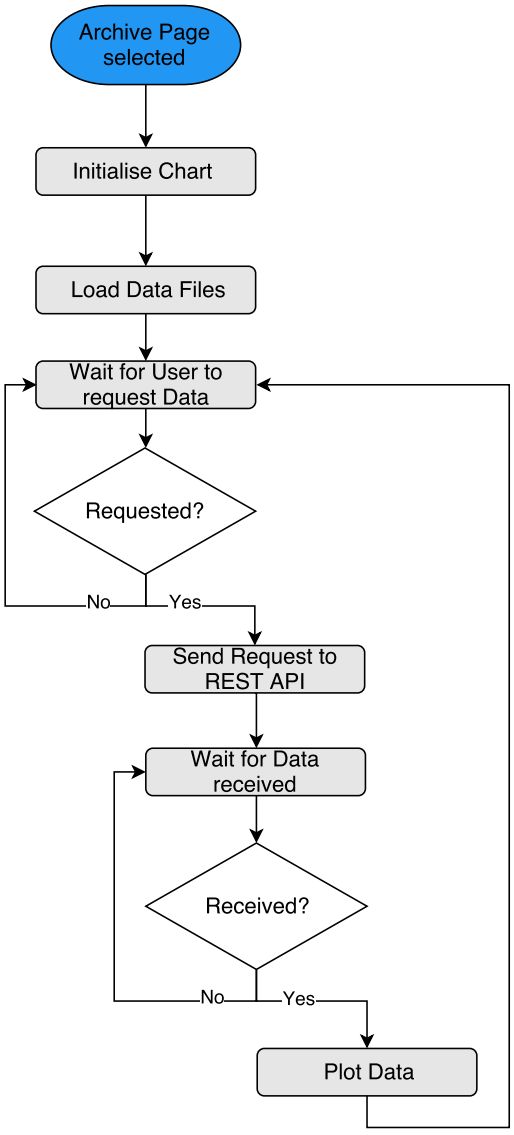


Figure 12.6: Flowchart of the procedure when Archive page is selected

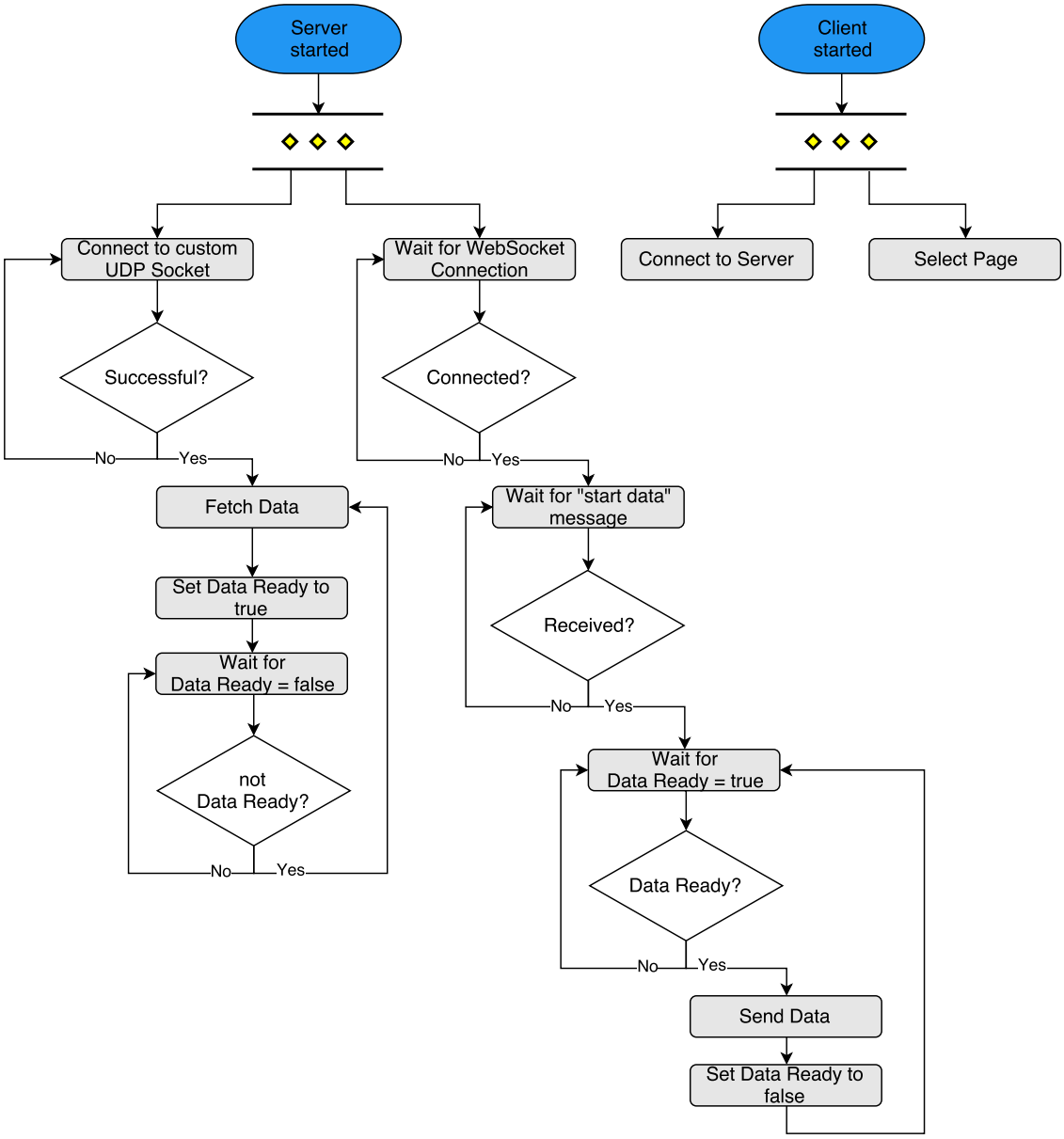


Figure 12.7: Flowchart of web server and client program showing the procedure

Chapter 13

Web Server

Author: Nico Leidenfrost

To make a web application accessible to a user, there needs to be a web server which serves the files to the web browser of the user. In the case of GRAMOC, NGINX was chosen to be used as a web server [32].
A web server is a server dedicated to provide information, web sites or web applications. To achieve this goal the web server first transmits the files needed to run the web application and then sends information whenever it is requested.

13.1 Apache

Apache HTTP Server was first launched in 1995 [13]. It gained a lot of popularity very quickly and since April 1996 it is the most used web server of active websites. Apache's approach on how to handle incoming connections is quite simple: for each connection there is one thread. This can lead to problems when a lot of clients are trying to connect to the server at the same time. Apache consists of a core module and many dynamically loadable modules. These modules provide various features like:

- Support for various programming languages
- URL rewrite
- Proxy functionality
- SSL support
- Authentication utilities

This module system gives Apache a lot of power in terms of flexibility, because there is no need to use other systems as Apache itself is most often able to provide these features.

13.2 NGINX

NGINX, pronounced *engine x*, is a lightweight web server that was initially created to solve the *c10K* problem. The goal of this challenge was to create a web server that is able to handle ten thousand concurrent client connections at once. To achieve this goal, the main difference between NGINX and its competitors is that NGINX handles client connections asynchronously instead of synchronously. NGINX is also event driven and single threaded. This means that there is only one thread running and not one thread per connection. In GRAMOC, NGINX is used to provide only the core features of a web server, namely providing static web application content and basic routing between the web server and

the Node.js server. Any other functionality will be passed to components that can handle the specific tasks. NGINX was chosen over Apache because of its lightweight design and speed. This is mainly achieved through passing on tasks to other programs and not trying to handle everything on its own like Apache does.

13.3 Apache vs NGINX

According to Upguard, a cyber security company, NGINX is about 4.2 times faster than Apache [22]. The main reason for this is the single threaded, asynchronous approach of NGINX. Apache however contains a much larger feature set and better support, because of the fact that Apache is a lot older than NGINX. The smaller feature set of NGINX can be seen as a disadvantage, but also as an advantage. Less features are usually a clear disadvantage, but on the other hand if there are less features available, the system is much more lightweight. That is one of the main reasons why NGINX was chosen in GRAMOC. Due to this lightweight design, NGINX can operate also on systems with less computing power, like the Raspberry Pi that is used in GRAMOC.
In terms of popularity Apache is at the time of writing twice as much used in web server development as NGINX. These figures originate from the *February 2018 Web Server Survey* conducted by Netcraft and are shown in figure 13.1 [29].

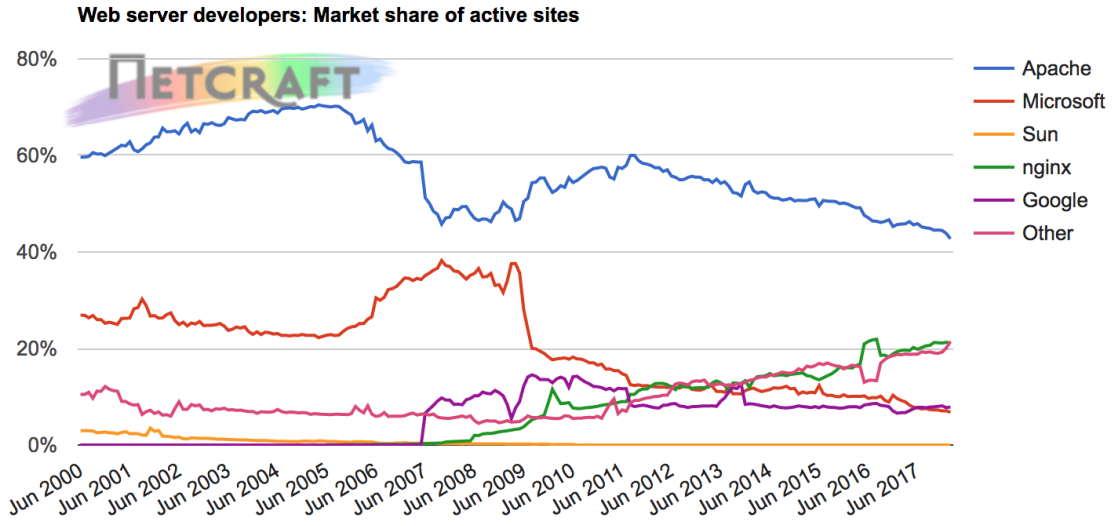


Figure 13.1: Survey results about most used web servers in currently active websites

These results show that Apache is still the number one web server with 42.72 percent market share. Number two with 21.13 percent is NGINX. The trend shows that NGINX is gaining more and more popularity as Apache slowly looses its market share. Below these two main competitors there is Google's in house developed web server, which they use for their own services, and Microsoft's IIS.

13.4 Node.js

Node.js is an asynchronous, event driven JavaScript runtime, designed to build network applications [14]. The built-in HTTP module can be used to create a web server based on

Node.js. This module is used to utilise communication over the HTTP protocol. In terms of GRAMOC the sensor data is received via a UDP connection and then transmitted over HTTP to the web application. Therefore, the main task of the Node.js server is to receive data which is emitted by the Filtering and Preprocessing Layer (FAPS) and forward it, to the web application via WebSockets implemented through the *socket.io* library [47]. In order to use socket.io the HTTP module in combination with the express framework must be used.
To implement an API (Application Programming Interface) to retrieve historical data, also Node.js was used. The actual framework that was used to implement the API is called *Express* (see below section 13.5).

13.5 Express

Express is a minimal and flexible web application framework to be used with Node.js. It is build on top of the HTTP module provided by Node.js. Handling basic routing tasks and creating powerful APIs (Application Programming Interfaces) are the main tasks of the Express framework. Sometimes Express is mistaken to be a web server on itself, but it is just a layer on top of a web server. Express was used to create the REST API used in GRAMOC (see below section 13.7).

13.6 socket.io

socket.io is a JavsScript library that implements real-time communication via WebSockets. This library was selected to be used within this project, because it aims to make real-time applications possible in every browser. Socket.io is available for both Node.js and Vue.js. The socket.io framework works well within GRAMOC, because GRAMOC also aims to deliver sensor data in real-time.

13.7 REST API

GRAMOC also saves the incoming sensor data beside plotting it. This data is especially useful to create statistics or if a user wants to inspect the sensor data of a certain point of time. How the data is saved is further explained in chapter 11 on page 38. To make the stored sensor data available to users, a REST API was implemented. REST means *Representational State Transfer*, which is an architectural style. To call an API RESTful it needs to satisfy a number of constraints [12]:

- Client-Server
- Stateless
- Cache
- Uniform Interface
- Layered System
- Code-On-Demand

13.7.1 Client-Server

REST features a client-server model, like many other web based architectures. The main concept of this model is that two clients can not talk directly to each other. Every client needs to communicate with a server. If network units need to communicate with each other,

then every unit must implement a client and a server. This separation of client and server can lead to better scalability of each component, as they are developed independently.

13.7.2 Stateless

A stateless system can not store data from clients to use this data in future requests. Each request must contain enough information, so that the server is able to send an appropriate response. This constraint leads also to improved performance as the server does not need to store any contextual data about clients and is therefore able to process multiple requests faster.

13.7.3 Cache

A response must be explicitly marked as cacheable or non-cacheable. If a response is marked as cachable, it can be reused for future equivalent requests. This is useful if such cached information can be reused at least one time, because then the client can simply use the cached data instead of sending a new request to the server every time.

13.7.4 Uniform Interface

A REST API should have a standardised uniform interface in order to maintain the simplicity of interactions. A resource in a system should only have one logical URI.

13.7.5 Layered System

In a layered system, every component can access only the layer next to it. This removes a lot of complexity from the system, as a component just needs to interact with its neighbours.

13.7.6 Code-On-Demand

The last constraint within the REST architecture is the Code-On-Demand constraint which is only optional. It allows an API to send executable code to the clients to extend their set of features. This is used to create simple clients which can be dynamically extended after deployment.

13.7.7 Implementation

In the GRAMOC REST API are only GET routes, as there is no need to change or modify data. The available routes are listed in table 13.1.

Route	**Response**
/files	returns a list of files within the data directory
/files/:file/data	returns the data of each dataset inside a file
/files/:file/datasets	returns a list of datasets stored inside a file
/files/:file/datasets/:id	returns the data of a specific dataset

Table 13.1: GET request routes of the REST API used in GRAMOC

The files are all stored in a specific directory from which the API can read. Inside these files there are datasets, which represent time points. With this structure it is possible to request the data from a specific timespan.

Pagination

Due to the large amounts of data, one request took longer to process than the timeout of the web browser allowed. This problem was solved by using pagination. Pagination is a technique to split the whole data from one big request into more smaller requests. These requests can be processed much faster and therefore a response can be send in time. To use pagination a query string with the parameters *limit* and *page* must be provided. The limit indicates the maximum amount of data which can be transmitted by one response. The page parameter specifies the offset of the data which has to be send. To check if another page is available, every response contains a field *has_more*.

Chapter 14

Data Analysis

Author: Nico Kratky

14.1 Regression Analysis

Regression analysis is a statistical method to determine relationships between a response variable y and one or more predictor variables x_i, where $i = 1, 2, ..., p$. Linear regression analysis assumes that these predictor variables are related linear to the response variable.

14.1.1 Simple Linear Regression

Simple Linear Regression is a regression model that can only build a relationship between one predictor variable and one response variable. To find the best fit for this linear model the *Ordinary Least Squares* method is used. The linear regression model builds a linear function

$$y = k * x + d \tag{1}$$

that represents the predicted values. This function can also be denoted as

$$y = \beta_0 + \beta_1 * x, \tag{2}$$

where β_0 is the intercept and β_1 is the slope of the line.
The two regression parameters,

$$\beta_1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x}) * (y_i - \bar{y}))}{\sum_{i=1}^{n}(x_i - \bar{x})^2} \tag{3}$$

$$\beta_0 = \bar{y} - \beta_1 * \bar{x}, \tag{4}$$

that are used in this regression model are calculated using the least squares method. This method tries to minimize the sum of squared residuals.

Example

A good example for linear regression analysis is the ringsize of women. This example was taken from the website http://www.crashkurs-statistik.de [8, 9].

If somebody wants to know the ringsize of his girlfriend, but does not want to ask her, it is possible to predict the size. To be able to do this a data basis has to be formed. A decisive factor for someones ringsize is for example the body height.
The data, which is depicted in table 14.1 can be used to calculate the regression coefficients using the formulae discussed in section 14.1.1 on the preceding page. These calculations result in the two regression coefficients

$$\beta_1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x}) * (y_i - \bar{y}))}{\sum_{i=1}^{n}(x_i - \bar{x})^2} = 0.2838 \tag{5}$$

$$\beta_0 = \bar{y} - \beta_1 * \bar{x} = 2.8457, \tag{6}$$

which are slope and intercept, respectively.

Person i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Ringsize y	47.1	46.8	49.3	53.2	47.7	49.0	50.6	47.1	51.7	47.8
Height x	156.3	158.9	160.8	179.6	156.6	165.1	165.9	156.7	167.8	160.8

Table 14.1: Ringsizes of example persons and their body heights

After these calculations, the regression line can be plotted as shown in figure 14.1.

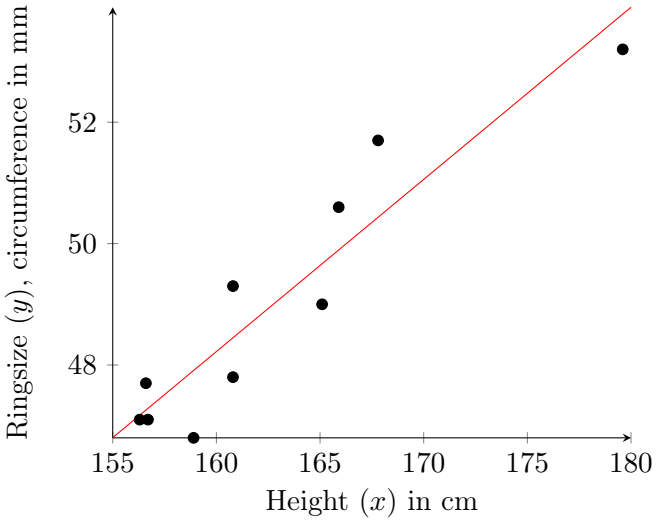


Figure 14.1: Ringsizes of women. The single data points represents the gathered data. The red line depicts the regression line.

To predict a persons ringsize, two options are available. The first option, which is not as accurate as the second one, is to simple read the predicted value from the regression line. If, for example, a person is 165cm tall, the proper ringsize would be a little bit less than 50. To be more precise, the second option can be used, which is to calculate the size using the regression coefficients. This can be done by applying the regression line to the height

$$y = 2.8457 + 0.2836 * 165, \tag{7}$$

which yields $y = 49.64$.

14.1.2 Multiple Linear Regression

When the dependent variable depends on not just one variable, multiple linear regression analysis is used. This method uses two or more independent variables to describe the dependent variable. To calculate the regression coefficients the predictor variables have to be put into a $n \times p$ matrix,

$$X_{n,p} = \begin{bmatrix} 1 & x_{1,1} & x_{1,2} & \cdots & x_{1,p} \\ 1 & x_{2,1} & x_{2,2} & \cdots & x_{2,p} \\ 1 & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n,1} & x_{n,2} & \cdots & x_{n,p} \end{bmatrix} \tag{8}$$

where n is the amount of datasets and p is the amount of predictor variables $+1$ because the intercept also has to be calculated. Also, a vector of all response variables

$$y_n = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \tag{9}$$

has to be formed. These two matrices can be used to describe the basic multiple linear regression model

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_p \end{bmatrix} = \begin{bmatrix} 1 & x_{1,1} & x_{1,2} & \cdots & x_{1,p} \\ 1 & x_{2,1} & x_{2,2} & \cdots & x_{2,p} \\ 1 & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n,1} & x_{n,2} & \cdots & x_{n,p} \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{bmatrix}, \tag{10}$$

where β are the regression coeffecients. Or shorter

$$y = X\beta. \tag{11}$$

The problem with this equation is that it is possible that this equation does not have a solution. Therefore the y and X matrices are multiplied with the transpose of X.

$$\hat{\beta} = (X^T X)^{-1} X^T y \tag{12}$$

This equation is always solvable, though not always exactly.

Example

With this in mind, the example from section 14.1.1 on page 55 can be extended by more body parameters like weight and age, as they may also have an impact on the accuracy of the regression model. Figure 14.2 depicts the ringsizes and body heights from the previous example, plus the weight and age of these people.

Person i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Ringsize y	47.1	46.8	49.3	53.2	47.7	49.0	50.6	47.1	51.7	47.8
Height x_1	156.3	158.9	160.8	179.6	156.6	165.1	165.9	156.7	167.8	160.8
Weight x_2	62	52	83	69	74	52	77	65	79	51
Age x_3	24	34	26	51	43	33	22	21	19	34

Table 14.2: Ringsizes of example persons and the appropriate body parameters

This dataset can now be used to form the previously explained matrices.

$$X_{10,3} = \begin{bmatrix} 1 & 156.3 & 62 & 24 \\ 1 & 158.9 & 52 & 34 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & 160.8 & 51 & 34 \end{bmatrix} \tag{13}$$

$$y_{10} = \begin{bmatrix} 47.1 \\ 46.8 \\ \vdots \\ 47.8 \end{bmatrix} \tag{14}$$

Using the regression parameter formula (see equation 12 on the preceding page) we get the regression parameters

$$\hat{\beta} = (X^T X)^{-1} X^T y = \begin{bmatrix} 0.66 \\ 0.28 \\ 0.06 \\ -0.02 \end{bmatrix}. \tag{15}$$

Using these parameters it is now possible to form the regression function

$$y = 0.66 + 0.28 * x_1 + 0.06 * x_2 - 0.02 * x_3, \tag{16}$$

which allows for data prediction.
In multiple linear regression analysis the predicted value can no longer be read off a graph, as the line is multi-dimensional. Lets assume that a woman is 170cm tall, weighs 68kg and is 29 years old. Inserting these values into the calculated model

$$y = 0.66 + 0.28 * 170 + 0.06 * 68 - 0.02 * 29 \tag{17}$$

gives us the predicted ringsize $y = 51.76$.

14.1.3 Working with Streaming Data

As GRAMOC uses streaming data a few adjustements had to be made. The ordinary multiple linear regression model assumes that all observations are available when calculating the regression coefficients. Therefore a way of calculating these regression coefficients in a streaming way had to be found. To accomplish this the regression parameters have to be calculated incrementally. This means that the two parts of the regression parameter calculation, $X^T X$ and $X^T y$ have to be recalculated every time a new observation is made. These two matrices are then added up, since matrix addition between two $n \times n$ matrices also result in a $n \times n$ matrix. The sum of these matrices are named M and V. This is possible as $X^T X$ always returns a $p \times p$ matrix and $X^T y$ always returns a p-dimensional vector.
When the regression coefficients are calculated,

$$\hat{\beta_k} = (M + {X_k}^T X_k)^{-1} (V + {X_k}^T y_k), \tag{18}$$

the sum of previous observation data is added to the current data, and then the procedure described in 14.1.2 on the previous page can be applied.
This procedure was introduced in a regression calculation programm called StreamFitter, which also works with streaming data [31].

14.1.4 r^2 - Coefficient of Determination

r^2 is the quotient of the explained variation

$$ESS = \sum_{i=1}^{n} (\hat{y}_i - \bar{y})^2 \tag{19}$$

and the total variation

$$TSS = \sum_{i=1}^{n} (y_i - \bar{y})^2, \tag{20}$$

$$r^2 = \frac{ESS}{TSS} \tag{21}$$

It is a measure for how much variation of the data can be explained with this regression model. r^2 values are ranged between 0 and 1, where 1 is considered a perfect fit and 0 says that no data can be explained using this regressino model.

14.2 Implementation

These procedures were implemented as a C++ class. The two most important methods from this class are `void push(std::vector<double> X, double y)` and `double predict(std ::vector<double> X)`. The first method adds new observations to the model, and the second one predicts the response variable from the given predictor variables using the current linear regression model.
Code listing 14.1 on the following page shows the `push` method. Additionally to computing the M and V values, this method adds the variables to a queue-like data structure. This is necessary as the coefficient of determination needs the average of the response variable $\bar{y}$ and the last few datasets to test this fit (see 14.1.4). The size of this structure is set to 100. This means that if the structure contains 100 datasets, the oldest one is removed before a new one is inserted.

```cpp
void MLR::push(std::vector<double> X, double y) {
    if (X.size() != p_) {
        // not the correct amount of predictor variables
        return;
    }

    // add observation to queue-like structure
    // used for computing the coefficient of determination
    data_.add(y, X);

    // create a 1x(p+1) matrix
    Eigen::Matrix<double, 1, Eigen::Dynamic> X2{p_ + 1};

    // insert values into matrix
    X.insert(X.begin(), 1);
    for (std::size_t i{0}; i < X.size(); ++i) {
        X2(0, i) = X[i];
    }

    auto X2T = X2.transpose();

    M_ = M_ + X2T * X2;
    V_ = V_ + X2T * y;

    // inserting new observations invalidates the coefficients
    coefficients_.clear();
}
```

Listing 14.1: C++ method to add new observations to the regression model

Code listing 14.2 show the `predict` method. This method initializes the result variable with the line intercept and then adds up the products of the predictor variable and the corresponding regression coefficient.

```cpp
double MLR::predict(std::vector<double> X) {
    if (X.size() != p_) {
        // not the correct amount of predictor variables
        return 0;
    }

    // start prediction with the intercept
    double prediction = coefficients_[0];

    // add each predictor variable with its coefficient
    for (std::size_t i{0}; i < p_; ++i) {
        prediction += coefficients_[i + 1] * X[i];
    }

    return prediction;
}
```

Listing 14.2: C++ method to predict the response variable using the passed predictor variables

14.2.1 Coefficient of Determination

To calculation of the coefficient of determination in FaPS a queue-like structure had to be implemented as the calculation formula needs the response variable, the predictor variables and the predicted y value. As GRAMOC works with streaming data, not all data can be stored for this calculation. Therefore only the last 100 values are stored.

The implemented method is depicted in code listing 14.3.

```
1 double MLR::r_squared() {
2     // initialise variables
3     double ess{0};
4     double tss{0};
5
6     // get last 100 observations
7     auto data{data_.data()};
8     // calculate average y value from the last 100 observations
9     double avg_y{data_.average_y()};
10
11    for (std::size_t i{0}; i < data.first.size(); ++i) {
12        // calculate explained sum of squares
13        ess += std::pow(predict(data.second[i]) - avg_y, 2);
14
15        // calculate total sum of squares
16        tss += std::pow(data.first[i] - avg_y, 2);
17    }
18
19    // return coefficient of determination
20    return ess / tss;
21 }
```

Listing 14.3: C++ method to calculate the coefficient of determination

14.2.2 Matrix Calculations

As the mathematical side of regression analysis requires a lot of calculations to be made using matrices and C++ does not have an equivalent to matrices, a third party library had to be chosen to compensate this. After comparing several linear algebra libraries that offer matrix calculations, it was decided to use Eigen.

14.2.3 Eigen

Eigen is a header-only C++ library, that was designed for doing basic linear algebra. This include matrix operations, vector calculations and numerical solvers. Its benefits include a very clean API which is fairly easy to use and a low memory overhead.
Perfomance-wise, Eigen is also better than its competitors.

Benchmarks

Figures 14.2 on the following page and 14.3 on the next page depict the computational performance of Eigen. These two tests were done by computing the products of a matrix and its transpose and a matrix and a vector. When multiplying a matrix with its transponse, the GOTO BLAS library has a slight advantage over all matrix sizes but when multiplying matrices with vectors even the older version of eigen outperforms all competitors.

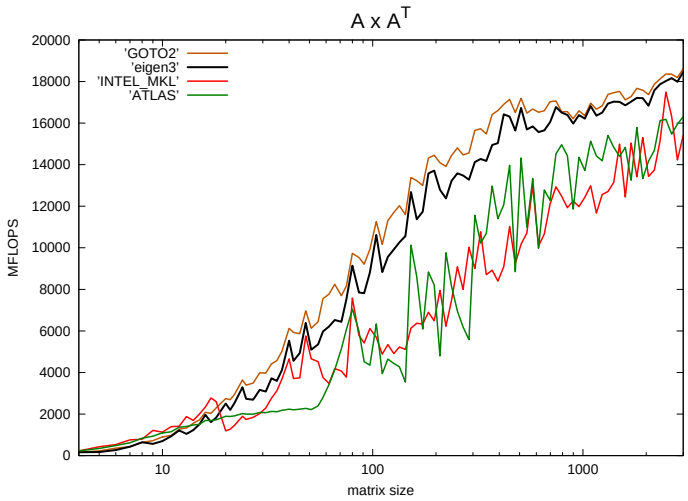


Figure 14.2: $A \times A^T$ Eigen Benchmark [23]

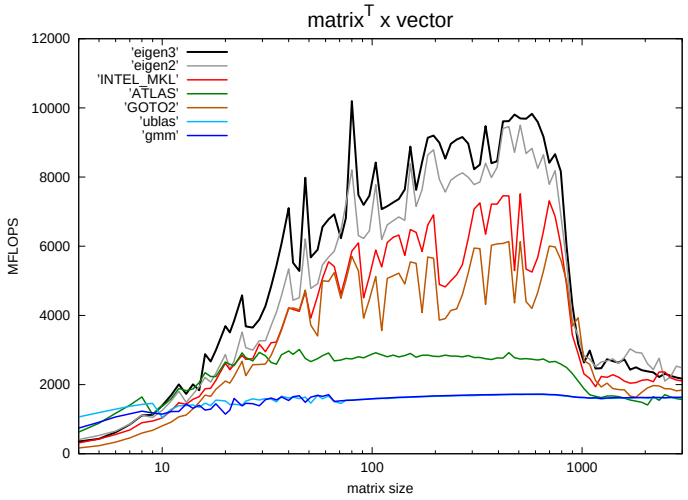


Figure 14.3: $matrix^T \times vector$ Eigen Benchmark [23]

Chapter 15

Measurement Results

Author: Nico Leidenfrost

In order to test GRAMOC, a test scenario had to be created. As an abstraction to the magnetometer, an accelerometer was used. The mathematical model to process the accelerometer sensor data is the same as the one form the magnetometer data which is ultimately used. The reason that a alternative had to be used is that the development of the gradient magnetometer was delayed during the writing of this thesis. Also acceleration is easier to influence without using scientific items.
To get example sensor readouts from the accelerometer, the sensor was mounted inside a car.

15.1 Car Mount

To mount the GRAMOC setup in a car, the 12V DC on-board power supply had to be inverted to 230V AC. This was necessary to power an access point as GRAMOC works over an internet connection. The two Raspberry Pis were powered by a powerbank. The Raspberry Pi that acted as sensor was taped to the arm rest of the car. This setup is depicted in figure 15.1 on the following page.

15.2 Test Scenarios

To demonstrate the real-time plotting capabilities, a few scenarios were chosen in which it is easy to understand the recorded data. These scenarios are:

- Shifting gears
- Driving in a roundabout
- Emergency braking
- Oversteering

As the used sensor is an accelerometer, the measured sensor data represent g-forces. The x axis represents the forward acceleration of the car, the y axis represents the lateral acceleration and the z axis represents the vertical acceleration. In the way the sensor was positioned, positive values on the x axis represent forward acceleration and negative values represent backward acceleration. Since the y axis displays the lateral acceleration, positive values represent a right turn and negative values represent left turns.

Figure 15.1: Car Mount of the Raspberry Pi accelerometer

15.2.1 Shifting Gears

The first demonstration scenario is shifting gears while driving. Every time a driver shifts to another gear, the acceleration is interrupted. This short interruption is documented by the red line, which represents the forward acceleration, in figure 15.2. It can be observed that every time the driver shifts to another gear the line drops back to zero. The example in figure 15.2 shows that the driver shifts 3 times from first to fourth gear.

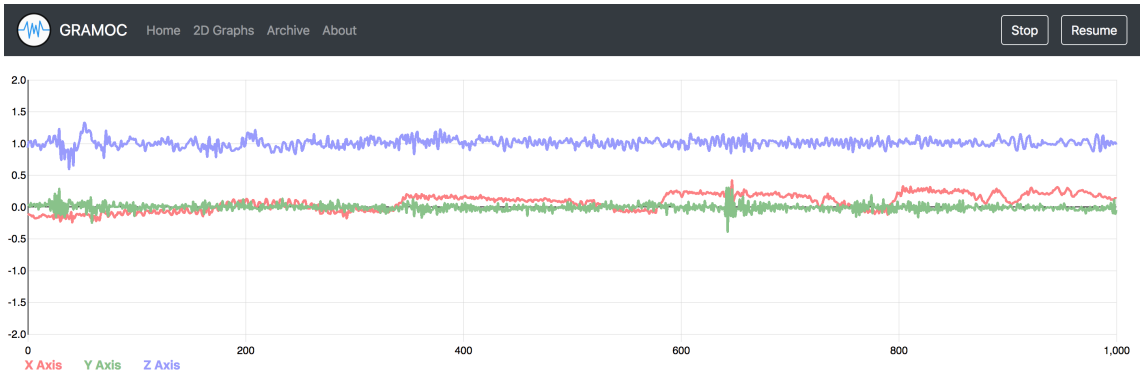


Figure 15.2: Measured sensor data when shifting gears

15.2.2 Driving in a Roundabout

In the second scenario, a driver is driving through a roundabout. This scenario was chosen to be used as an example, because it is a situation where the lateral acceleration is roughly the same all the time. As shown in figure 15.3 on the next page, the forward acceleration stays around zero the whole time and the lateral acceleration is located about -0.5g. The reason why the acceleration remains stable the whole time is simply because when driving through a roundabout, a driver should drive with constant speed and turn.

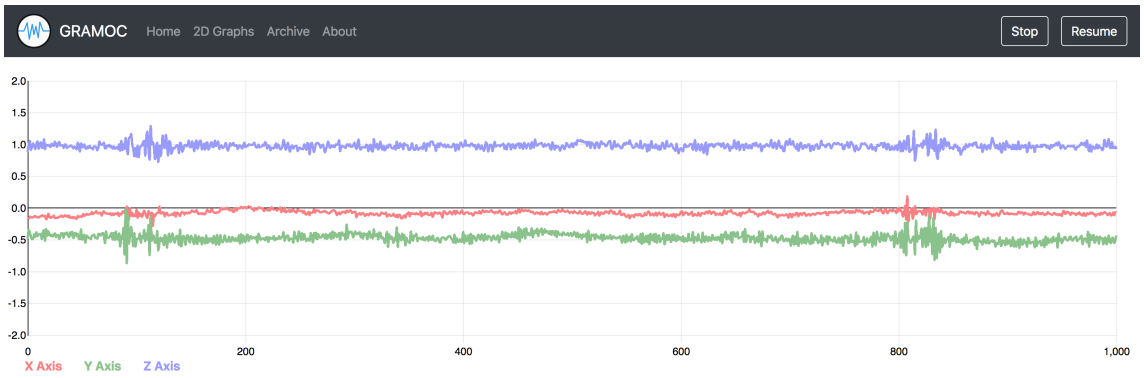


Figure 15.3: Measured sensor data when driving in a roundabout

15.2.3 Emergency Breaking

Another scenario where acceleration can be easily shown is the case of emergency braking. When a driver needs to stop his car immediately because there are for example people in front of the car, there needs to be a massive amount of negative acceleration, depending on the current velocity of the car. The acceleration force of such a maneuver is depicted in figure 15.4. The example below shows a driver who is driving at a steady speed of 30 km/h at first and then stops the car immediately. While braking the measured g-force reached a maximum of -1.1g.

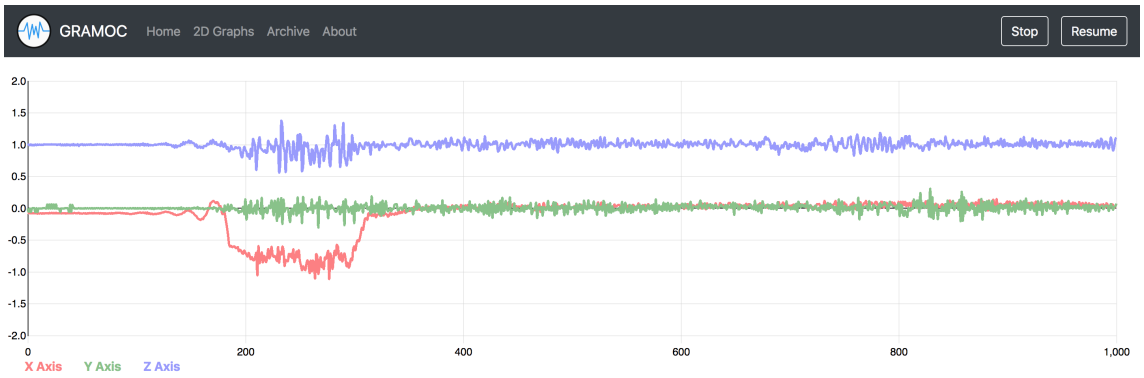


Figure 15.4: Measured sensor data when applying an emergency brake

15.2.4 Oversteering

The last scenario was chosen to show how a car behaves when it is temporarily out of control. This could happened during oversteer caused by a wet street. As depicted in figure 15.5 on the next page the forward acceleration is erratic, but above zero the whole time during the oversteer phase. The green line which represents the lateral acceleration fluctuates between 1.0g and -1.0g for a short duration, in the remaining time the line stays below zero, due to the fact that the maneuver was applied in a left turn.

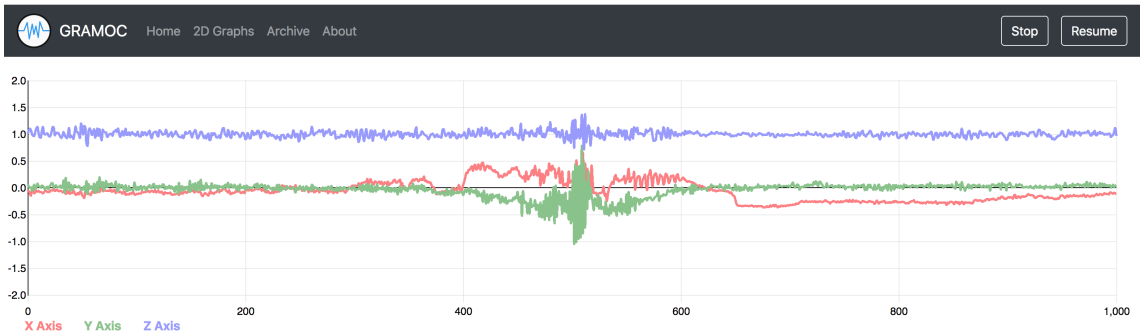


Figure 15.5: Measured sensor data when oversteering

15.3 Regression Results

To test the multiple linear regression capabilities of GRAMOC, another test scenario had to be created. During these tests, it was tried to predict the driver of the car using the acceleration data from the sensor. It was decided to only use forward and lateral acceleration, as the vertical acceleration mostly depends on parameters like road condition. To link the acceleration data to the driven route a timestamp was taken as the third variable.

15.3.1 Course 1

The first test used a predefined route that is depicted in figure 15.6. This route was driven by two sample drivers. One that drives rather quick and one slow driver. These drivers were numerically represented by 1 and 2.

Figure 15.6: Satellite view of the test route. Image taken from Google Maps

After completing the two runs, GRAMOC tried to predict the driver. As GRAMOC uses streaming data, the prediction can not simply be shown at the end of the program, as this would be too inaccurate. Therefore only the average prediction is shown after the program finishes. The prediction data itself is still getting calculated in real-time. Also the minimum, maximum and standard deviation of the average are calculated. The obtained results after testing this with the driving style of the two sample drivers are shown in table 15.1.

Driving Style	**Prediction**
1	1.4
2	1.8

Table 15.1: Prediction from the first regression testing phase

These predictions only allow to identify tendencies, but do not predict the driver precise enough.
After researching thoroughly, it was decided to use a course that does not have long straights as these were identified to mislead the regression model. This happens because on long straights the car mostly reaches the speed limit of the road and then does not have any acceleration. And no acceleration can not be matched with a driver.

15.3.2 Course 2

The second course had to be windy. The easiest way to get a lot of acceleration is to drive through a roundabout. In a roundabout there will always be at least some acceleration, so this was taken as the second course. For the sake of simplicity the roundabout that was used in the first course (see figure 15.6 on the preceding page) was used as this test-roundabout. The procedure was the same as in the first test, two drivers drive the roundabout at whichever speed they like. After this two runs one driver drives again and GRAMOC tries to predict him. This time, the results were much better as depicted in table 15.2.

Driving Style	**Prediction**	**Standard Deviation**
1	0.97	0.18
2	1.99	0.09

Table 15.2: Prediction from the second regression testing phase

Chapter 16

Conclusion

The key task of GRAMOC is to process and visualise sensor data. The visualisation happens in real-time, which means it is well suited for monitoring solutions. An important feature of GRAMOC is the prediction of certain parameters, based on the input sensor data. With this predicted values it is possible to use GRAMOC as a quality inspection tool or as a tool to verify the status of certain events.

16.1 Applications of GRAMOC

16.1.1 Steel Belt Quality Inspection

The initial and main use case of the GRAMOC system is to verify the quality of steel belts during production. With the gathered data from highly sensitive MEMS gradient magnetometers, GRAMOC is able to predict the quality of steel belts. If the quality of the steel belts can be predicted in real-time, it is no longer necessary to halt production periodically to inspect a part of the steel belt to determine the quality. Without the need to halt the production it is way more efficient than before.

16.1.2 Transport Driver Verification

Another possible use case of GRAMOC could be the verification of transport drivers. For companies that employ drivers to transport any kind of cargo, it could be of interest to know how the drivers are driving. Another concern for such companies is to know who is driving their cars. For example, if a taxi driver is sick but does not want to loose his income for the day, he could send a family member to do his work. To prevent such occurrences, GRAMOC could be used to determine the driving profile of a driver and report the current driver if the driving style does not match the given driving profile.

16.1.3 Sensor Monitoring

A very generic application of GRAMOC is the monitoring of multiple sensors at once. GRAMOC could be used to create a dashboard for a broad variety of different sensors. This system would best fit into an environment where a lot of sensors need to be monitored. Such environments could be production lines or power plants. Especially in the case of a power plant it is critical to know if every machine is working properly. Another benefit that GRAMOC would add is a uniform design and user experience. If every sensor brings it own monitoring system, it could be complicated for employees to operate every system

properly. GRAMOC could replace all of these systems and provide a uniform experience, which would simplify the tasks of employees.

16.2 Outlook

Many systems to gather sensor data that are used in the industry today feature different approaches for different sensors. The difference in how to control a specific sensor within one system can lead to many difficulties. To resolve that problem and to remove these difficulties, GRAMOC could be extended and implemented. Because with GRAMOC the controls of each sensor could be centralised and therefore it would be much easier to operate the whole system. But to use GRAMOC in such a large scale, it is necessary to implement additional features like:

- Additional Types of Sensors
- Additional Visualisation Methods
- Additional Features

Additional Types of Sensors

In the future GRAMOC should support a broad variety of sensor types. GRAMOC features abstraction layers, through which it is relatively easy to implement support for new sensor types. With the addition of new sensor types the field of application grows and GRAMOC could become a common solution to visualise sensor data.

Additional Visualisation Methods

If more sensors are included in the system, there will be at some point the need for more features or ways how to visualise data. Data from certain sensors can not be properly displayed in two dimensions. Therefore it will be necessary at some point to enable three or more dimensional data representation. This would be one of the most prominent features that need to be added in the future. Other important features than adding support for more dimensions, will be support for different representations. Because certain sensor data needs to be depicted in more than one representation to create relevant results.

Additional Features

Other features could be more advanced statistical analyses, or intuitive ways to interact or convert the gathered data. The implemented multiple linear regression is only a first step to fully automated statistical analyses, that could be realised within GRAMOC. With the addition of more sensors, data could be predicted based on multiple sources which could lead to greater accuracy. Also if users, most likely scientists, need to inspect the data or display different representations of the data, it is necessary to implement functions to transform the data.

Bibliography

[1] *Astro Pi Mission*. Aug. 2017. URL: https://astro-pi.org/about/mission/.

[2] Babel. *Babel*. Nov. 2017. URL: https://babeljs.io.

[3] Mike Bostock. *d3.js*. Nov. 2017. URL: https://d3js.org.

[4] Robert Braden. *Requirements for Internet Hosts - Communication Layers*. RFC 1122. RFC Editor, Oct. 1989, pp. 1–116. DOI: 10.17487/RFC1122. URL: https://www.rfc-editor.org/rfc/rfc1122.txt.

[5] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259. RFC Editor, Dec. 2017, pp. 1–16. DOI: 10.17487/RFC8259. URL: https://www.rfc-editor.org/rfc/rfc8259.txt.

[6] Stephen Cleary. *Message Framing*. (Accessed: 08.17). Apr. 2009. URL: https://blog.stephencleary.com/2009/04/message-framing.html.

[7] Matt Cook. *TCP vs. UDP for Real-Time Data Transfer*. (Accessed: 05.01.18). Oct. 2017. URL: https://www.lifesize.com/en/video-conferencing-blog/tcp-vs-udp.

[8] Crashkurs Statistik. *Einfache Lineare Regression*. (Accessed: 24.02.18). Jan. 2017. URL: http://www.crashkurs-statistik.de/einfache-lineare-regression.

[9] Crashkurs Statistik. *Multiple Lineare Regression*. (Accessed: 24.02.18). July 2017. URL: http://www.crashkurs-statistik.de/multiple-lineare-regression.

[10] Byron Ellis. *Real-Time Analytics: Techniques to Analyze and Visualize Streaming Data*. 1st ed. Wiley, July 2014. ISBN: 978-1-118-83791-7.

[11] Ian Fette and Alexey Melnikov. *The WebSocket Protocol*. RFC 6455. RFC Editor, Dec. 2011, pp. 1–71. DOI: 10.17487/RFC6455. URL: https://www.rfc-editor.org/rfc/rfc6455.txt.

[12] Roy Thomas Fielding. "Architectural Styles and the Design of Network-based Software Architectures". (Accessed: 22.03.18). PhD thesis. University of California, Irvine, 2000. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[13] Apache Software Foundation. *Apache HTTP Server*. (Accessed: 22.03.18). Mar. 2018. URL: https://httpd.apache.org.

[14] Node.js Foundation. *Node.js*. Nov. 2017. URL: https://nodejs.org.

[15] GitLab. *GitLab*. Nov. 2017. URL: https://about.gitlab.com.

[16] Google. *Android Activity*. Aug. 2017. URL: https://developer.android.com/reference/android/app/Activity.html.

[17] Google. *Android Intent*. Sept. 2017. URL: https://developer.android.com/reference/android/content/Intent.html.

[18] Google. *Android Processes and Threads*. Nov. 2017. URL: https://developer.android.com/guide/components/processes-and-threads.html.

[19] Google. *Android SDK*. Aug. 2017. URL: https://developer.android.com/reference/packages.html.

[20] Google. *ART and Dalvik*. (Accessed: 27.03.18). Nov. 2017. URL: https://source.android.com/devices/tech/dalvik.

[21] IBM. *Network byte order and host byte order*. (Accessed: 21.3.18). URL: https://www.ibm.com/support/knowledgecenter/en/SSB27U_6.4.0/com.ibm.zvm.v640.kiml0/asonetw.htm.

[22] Upguard Inc. *Apache vs Nginx*. (Accessed: 22.03.18). Mar. 2017. URL: https://www.upguard.com/articles/apache-vs-nginx.

[23] Benoît Jacob and Gaël Guennebaud. *Eigen - Benchmark*. (Accessed: 20.3.18). Dec. 2016. URL: http://eigen.tuxfamily.org/index.php?title=Benchmark.

[24] Philipp Jahoda. *A powerful & easy to use chart library*. Aug. 2017. URL: https://github.com/PhilJay/MPAndroidChart.

[25] Kitware. *Visualization Toolkit*. Sept. 2017. URL: https://www.vtk.org.

[26] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. 2nd ed. Real-Time Systems. Springer, Apr. 2011. ISBN: 978-1-4419-8236-0.

[27] Nico Kratky. *Java implementation of the GSDEP Client*. Aug. 2017. URL: https://github.com/nicokratky/GramocAlgorithm-client.

[28] Stefan Krause. *js-framework-benchmark*. Nov. 2017. URL: https://github.com/krausest/js-framework-benchmark.

[29] Netcraft Ltd. *February 2018 Web Server Survey*. (Accessed: 22.03.18). Mar. 2018. URL: https://news.netcraft.com/archives/2018/02/13/february-2018-web-server-survey.html.

[30] William MacDougall. *Industrie 4.0 - Smart Manufacturing for the Future*. Tech. rep. Germany Trade & Invest, 2014.

[31] Chandima Hewa Nadungodage et al. "StreamFitter: A Real Time Linear Regression Analysis System for Continuous Data Streams". In: Feb. 2011. DOI: 10.1007/978-3-642-20152-3_39.

[32] NGINX. *NGINX*. Feb. 2018. URL: https://www.nginx.com.

[33] Plotly. *Plotly*. Nov. 2017. URL: https://plot.ly.

[34] Plotly. *plotly.js*. Nov. 2017. URL: https://github.com/plotly/plotly.js.

[35] Jon Postel. *Internet Protocol*. RFC 791. RFC Editor, Sept. 1981, pp. 1–45. DOI: 10.17487/RFC0791. URL: https://www.rfc-editor.org/rfc/rfc791.txt.

[36] Jon Postel. *Transmission Control Protocol*. RFC 793. RFC Editor, Sept. 1981, pp. 1–85. DOI: 10.17487/RFC0793. URL: https://www.rfc-editor.org/rfc/rfc793.txt.

[37] Vladimir Prus. *Boost.Program_options*. Oct. 2017. URL: http://www.boost.org/doc/libs/1_65_1/doc/html/program_options.html.

[38] Raspberry Pi Foundation. *Raspberry Pi*. Aug. 2017. URL: https://raspberrpi.org.

[39] Raspberry Pi Foundation. *Raspberry Pi SenseHAT*. Aug. 2017. URL: https://www.raspberrypi.org/products/sense-hat/.

[40] reichelt elektronik. *Raspberry Pi 3 Model B*. URL: https://cdn-reichelt.de/bilder/web/xxl_ws/A300/RASP_03_01.png.

[41] reichelt elektronik. *Raspberry Pi Sense Hat*. URL: https://cdn-reichelt.de/bilder/web/xxl_ws/A300/RPI_SENSE_HAT_1.png.

[42] Christoph Roser. *Industry 4.0*. 2015. URL: http://www.allaboutlean.com/wp-content/uploads/2015/11/Industry-4.0.png.

[43] Mehdi Sakout. *Create an awesome About Page for your Android App in 2 minutes*. Aug. 2017. URL: https://github.com/medyo/android-about-page.

[44] Jacob Schatz. *Why We Chose Vue.js*. (Accessed: 05.11.17). Oct. 2016. URL: https://about.gitlab.com/2016/10/20/why-we-chose-vue.

[45] Metin Seylan. *Vue-Socket.io*. Jan. 2018. URL: https://github.com/MetinSeylan/Vue-Socket.io.

[46] Andrew Skotzko. *Understanding The Internet: How Messages Flow Through TCP Sockets*. (Accessed: 28.08.17). July 2015. URL: https://andrewskotzko.com/understanding-the-internet-how-messages-flow-through-tcp-sockets/.

[47] socket.io. *socket.io*. Nov. 2017. URL: https://socket.io.

[48] stackgl. *stackgl*. Nov. 2017. URL: http://stack.gl.

[49] Sun Microsystems, Inc. *Transport Interfaces Programming Guide*. Oct. 1998. URL: https://docs.oracle.com/cd/E19620-01/805-4041/805-4041.pdf.

[50] The Blue Brain Project. *HighFive - HDF5 header-only C++ Library*. Jan. 2018. URL: https://github.com/BlueBrain/HighFive.

[51] *The Go Project*. Aug. 2017. URL: https://golang.org/project/.

[52] The HDF Group. *HDF5*. Nov. 2017. URL: https://www.hdfgroup.org.

[53] L. V. Tulchak. *History of Python*. Tech. rep. Vinnytsia National Technical University, 2016.

[54] vuejs. *vue-cli*. Jan. 2018. URL: https://github.com/vuejs/vue-cli.

[55] vuejs. *vue-router*. Jan. 2018. URL: https://github.com/vuejs/vue-router.

[56] webpack. *webpack*. Nov. 2017. URL: https://webpack.js.org.

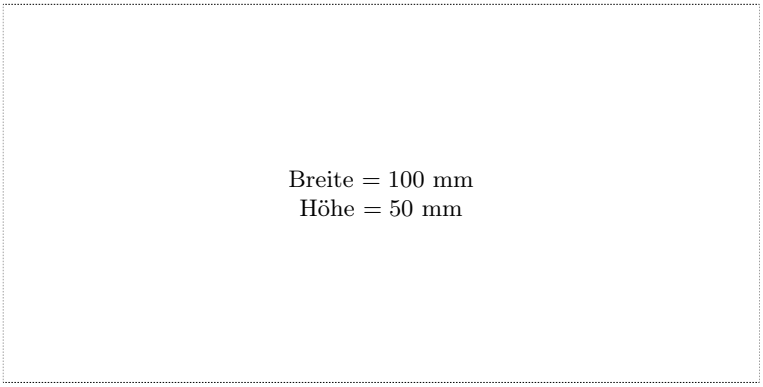
[57] WHATWG. *Server Sent Events*. Jan. 2018. URL: https://html.spec.whatwg.org/multipage/server-sent-events.html.

[58] Katharina Wuttke. *Im Wandel der Zeit: Von Industrie 1.0 bis 4.0*. (Accessed: 20.02.18). Sept. 2015. URL: https://www.lmis.de/im-wandel-der-zeit-von-industrie-1-0-bis-4-0/.

[59] Evan You. *Vue.js*. Nov. 2017. URL: https://vuejs.org.

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —