UNIVERSITY OF BERN

BACHELOR THESIS

Attitude Heading Reference System and Step Recognition for Cloud-Based Indoor Positioning – Android Client

handed in by
Lucien MADL

Supervisor

PROFESSOR DR. TORSTEN BRAUN

Communication and Distributed Systems
Institute of Computer Science

September 16, 2018

UNIVERSITY OF BERN

Faculty of Science
Institute of Computer Science

Bachelor of Science in Computer Science

Attitude Heading Reference System and Step Recognition for Cloud-Based Indoor Positioning – Android Client

by Lucien MADL

Abstract

Due to the rising popularity of mobile devices and their performance increase over the last few years, a whole new generation of location based services were created. This work is one part of three different theses which together propose a Cloud-Based Indoor Positioning System, providing three different clients for data gathering, a service able to compute the current position running on a Tornado server and a web application to visualize the computed position. Being completely platform independent, the proposed service can be accessed by any device providing the required data (accelerometer, gyroscope, magnetometer and signal strength from access points) and the ability to connect to a WebSocket. Within the scope of these theses projects we built three different types of clients (Android, iOS and ESP32) that gather data and provide it to the server. The service provides real time indoor positioning to connected clients by implementing a *Particle Filter* identifying the current position, which is supported by an Attitude Heading Reference System (AHRS) called *Rotations*, a *Step Detection* function as well as the *Ranging Model*, which creates a regression model by means of the measured signal strength to define the distance to access points.

Acknowledgements

I would like to thank Torsten Braun, Zhongliang Zhao and Jose Carrera for supervising this thesis and supporting it on various occasions during this project.

Further thanks goes to my project partners, Stefan Serena and Patrick Hodel. Sometimes we were struggling, but we learned a lot while always staying positive and being supportive.

Contents

List of Figures

List of Tables

List of Abbreviations

AHRS	**A**ttitude **H**eading **R**eference **S**ystem
ANOVA	**An**alyis **o**f **Va**riance
BSSID	**B**asic **S**ervice **S**et **ID**entifier
CDS	**C**ommunication and **D**istributed **S**ystems
Df	**D**egrees of **f**reedom
GPS	**G**lobal **P**ositioning **S**ystem
IMU	**I**nertial **M**easurement **U**nit
MVVM	**M**odel **V**iew **V**iew**M**odel
RSSI	**R**eceived **S**ignal **S**trength **I**ndicator

Chapter 1

Introduction

1.1 Overview

Due to the rising popularity of mobile devices (mostly smart phones) and their increasing performance during the last few years, a whole new generation of location based services were created [31]. These services in general use the Global Positioning System (GPS) to localize a device, which generally is not suitable for indoor localization, because GPS signals will be attenuated and scattered by roofs, walls and other objects [35].

For this reason, there is a need for alternative methods to determine the position inside a building, since todays location services provide various opportunities which require location information. Context information could be the exact position in a room, the current time, the heart rate of a user or even the current weather situation. All this information can then be used to interact with a mobile device and its user. These new services, which interact with mobile devices, offer a lot of opportunities like exact indoor positioning, context aware advertisement or even the possibility to create social graphs without using any social network data.

This thesis is part of a collaboration project which involves two other students, with the goal to create a cloud-based indoor positioning system which could provide indoor positioning services for multiple mobile devices. The location is computed by a centralized server called service that is able to process data gathered by a variety of mobile devices connected over WebSocket to calculate the current position. Three different client applications are within the scope of this thesis, namely for Android, iOS and micro controllers (ESP32 [11]).

Figure 1.1 shows the architecture of the proposed cloud-based indoor positioning system, which includes the following components:

- A ***Particle Filter*** providing Real-time position based on:
  - Pedestrian dead reckoning (PDR)
  - WiFi ranging
- The ***Rotations*** computing the orientation of a device relative to the earths frame providing an Attitude Heading Reference System (AHRS) using an implementation of the Madgwidge algorithm.
- A ***Step Detection*** able to notify the system when a step is detected.
- The ***Ranging***-models offering the possibility to determine the distance from a client to an access points.

- A ***web client*** showing a visual representation of the current position in a web browser.

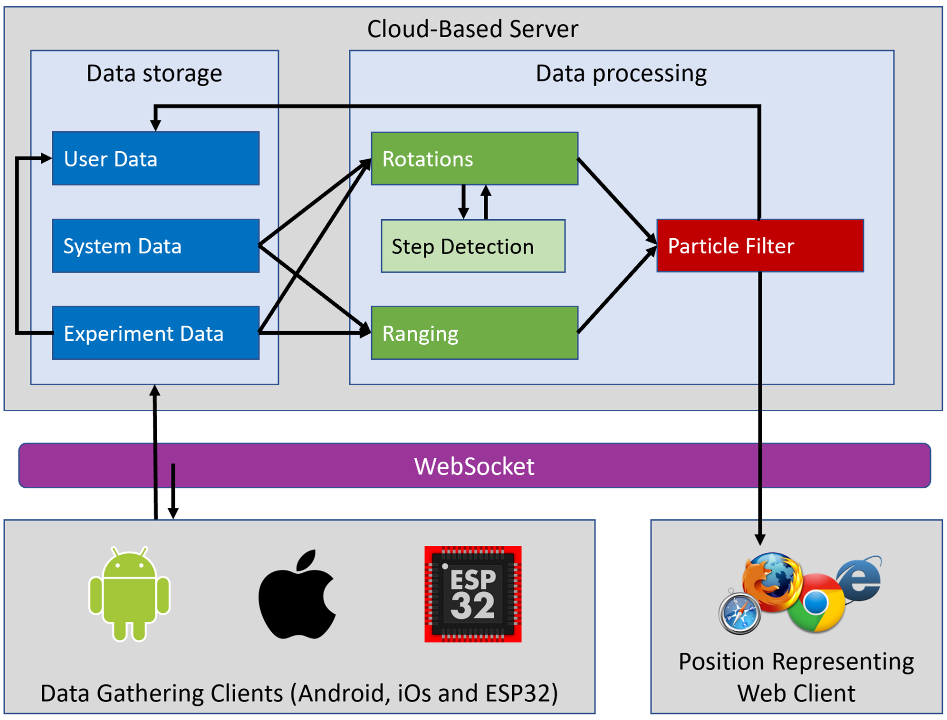


FIGURE 1.1: Architecture of the cloud-based indoor positioning system

The *particle filter* is at the heart of the service. It determines the current location by merging information from pedestrian dead reckoning (PDR) and WiFi trilateration using the strengths of both methods, while trying to overcome their weaknesses.

The PDR uses the sensor data of an accelerometer, a magnetometer and a gyroscope to calculate the current location with respect to the previous position. Due to the inaccuracies of the sensors, this process entails a small error. This error might accumulate by continually using the previous positions to calculate the current ones. That is why we get a composite error that grows over time. We eliminate this error and correct the current position periodically using the position computed by WiFi trilateration [41]. WiFi trilateration strongly relies on the WiFi *Ranging Model*, which evaluates the distance of a mobile device relative to multiple access points with known positions by using the Receiving Signal Strength Indicator (RSSI).

In theory, these two methods supplement each other quite well, but in reality the WiFi trilateration is not always reliable. The measurement of WiFi RSSI needs time, the measured results are strongly influenced by external circumstances of the environment and individual measurements cannot be made at the same time.

The purpose of the other parts of the service is to support the *particle filter* by improving its accuracy. It accomplishes this by taking advantage of *ranging models*, which provide the distance to access points, as well as additional services like *rotations* and *step detection*.

The main focus of this thesis is on *rotations*, the *step detector* as well as the *Android client*.

1.2 Motivation

Due to the increasing computational performance of modern mobile devices, service providers and application developers are tempted to run the logic of services directly on mobile devices, which are already gathering data. But there are some downsides to computing complex algorithms on mobile devices. First of all, the energy resources of mobile devices are severely limited, which should never be ignored when developing a service. Furthermore, when running logic directly on a mobile device, we strongly depend on the operating system and its framework. This also makes us dependent on powerful companies like Apple or Google, which makes the operation of an independent service difficult.

To offload these resource-intensive computations from the mobile device to the cloud brings with it the advantage of strict separation between data gathering and computation, which allows for an improved system architecture and provides a platform independent service to a much bigger customer base. However, by offloading the computational aspects to the cloud, we have to consider other challenges like the communication between the server and mobile devices.

To determine the orientation of a device and to recognize steps, timing is key, since both algorithms depend on the sampling rate of the gathered data. The algorithms, therefore, benefit from a continuous input stream of sensory data, which can be a challenge when separating the computation from the data gathering. This work will present a solution to this problem.

1.3 Contributions

This work is part of a cloud-based indoor positioning project. The whole project provides a centralized server as a service, which serves different needs of Android, iOS and micro controllers (ESP32) clients.

The main contributions of this thesis are summarized as follows:

- We present an architecture able to offload CPU intensive algorithm to a cloud-based server.
- We demonstrate how to identify the orientation of a device relative to gravity and the earth's magnetic field, by implementing the logic of the Madgwick algorithm in our service.
- We provide a simple but functional observer to our service that recognizes steps by considering the accelerometer measurements.
- We developed an Android client based on the modern Android Architecture Components, which is a framework that allows us to create applications providing life-cycle aware objects.

1.4 Structure

This thesis is structured as follows: The theoretical background is reviewed in chapter 2. The architecture of the different system parts is introduced in chapter 3, while the important aspects of the implementation from the client up to the server are explained in chapter 4.

The evaluation follows in chapter 5. It explains the experiment and examines the data obtained. The final chapter 6 summarizes the work and concludes the evaluation results.

Chapter 2

Theoretical Background

This chapter describes the theoretical foundations of the main aspects of this thesis, namely *rotations*, the *step detection*, and the *Android Client*. Furthermore, it explains the underlying concept of the indoor localization approach used.

2.1 Basic Idea - Indoor Localization Approach

Let us start this section with a fitting definition of localization:

> “Localization refers to the task of determining the location of a traveler in a specified coordinate system, which is subject to topological constraints, using a mobile device carried by the traveler.” [9]

In our case, we try to determine the indoor location of a traveler in a known building. As already implied in the first chapter, there is a substantial difference between indoor and outdoor localization. While both scenarios typically require one or multiple transmitting devices and one receiving device, we have to use different approaches.

Due to the distraction of buildings, long-range transmitting devices used for outdoor localization like satellites or antennas are not working properly indoors [35]. Therefore, we need to substitute them with alternative transmitting devices like WiFi access points or iBeacons [16].

One indoor transmitting device such as a WiFi access point enables us to define the proximity location of a receiving device, which can be graphically interpreted as a circle around the transmitting device. When receiving data from two transmitting devices, we can determine a line as location indicator. Even more precision can be achieved by using three or more transmitters, since geometrical algorithms based on trilateration allow us to specify a specific point[41].

2.2 Rotations

One of the main goals of this thesis is to extend the cloud-based indoor positioning service with the possibility to determine the orientation of a device relative to the direction of gravity and the earth’s magnetic field. Due to this extension, it should be possible to neglect the orientation of a device relative to space and still be able to provide the positioning functionality.

2.2.1 Attitude and Heading Reference System (AHRS)

The initial assignment of this project was to deliver a Inertial Measurement Unit (IMU), but after some research we noticed that an IMU is only able to measure the orientation relative to the direction of gravity. What we really need for our service is an AHRS, since they are able to provide the complete measurement of orientation relative to the direction of gravity and the earth's magnetic field. [19] Let us start with the sensors of the clients and then work our way up to the server.

2.2.2 Sensors

In order for the orientations to be calculated, the clients have to continuously send data from three different sensors (accelerometer, gyroscope and magnetometer). In this work, we use the following sensors that define how the algorithm must work:

The **gyroscope** delivers the angular velocity and is the most dynamic sensor of the three because it provides real time information about rotation changes immediately. The downside of the gyroscope is that the measured changes are not relative to the earths coordinate system.

The **accelerometer** measures acceleration which is the rate of change of velocity of a body in its own instantaneous frame of rest [36]. This means that we can determine the orientation relative to the direction of gravity in relation to the frame of the earth, at least when moving slowly. The main weakness of the accelerometer is that it can be very noisy.

The **magnetometer** measures magnetism and points to north in relation to the earth's coordinate system. The magnetometer is prone to inaccurate data because of its vulnerability to noise from the environment. This is why we need to calibrate the magnetometer before use. There are various ways to calibrate a magnetometer. In this work we implemented hard and soft iron calibration [28]. During each measurement phase, the hard iron calibration takes n data samples and calculates the offsets b_x , b_y and b_z for all the three axes x, y and z. It does so by calculating the average of the maximum and minimum of the sensor data for every axis $s_{x,y,z}$ and then subtracting the offsets from the raw magnetometer data [28].

Offset b for the three axes:

$$b_{x,y,z} = \frac{(\max_{1 \le j \le n}(s_{x,y,z}) + \min_{1 \le j \le n}(s_{x,y,z}))}{2}$$

The soft iron calibration multiplies the soft iron correction estimate with the raw sensor data. This process can lead to worse results than using the hard iron calibration.

2.2.3 Representation of the Rotation

Since three dimensional rotations are complex movements, we need to find an ideal way to represent them in our system. We also have to take into account that we need to continuously feed state updates to the *rotations* algorithm which includes the AHRS.

First, we define a locked, root coordinate system x, y and z and a rotating coordinate system X, Y and Z, which is relative to the device. At first, these two coordinate systems are overlapping. After every rotation of the device and its changing coordinate system, we need to express the difference between the root and the rotating coordinate system.

Euler angles are often used in aviation or car manufacturing. They describe the rotations around the axis using the angles yaw (or heading) φ, pitch ψ and roll θ. Yaw φ represents the rotation around the z-axis, pitch ψ around the y-axis and roll θ around the x-axis.

One problem of the Euler angles is that they allow for a gimbal lock, which leads to inaccurate results due to losing one degree of freedom. For example, when pitch ψ is at 90 degrees, a device is pointing straight up and lies exactly on the z-axis. In this situation, yaw φ and roll θ rotations describe the exact same movement [18].

Many use cases like cars do not allow the heading to point upward, so they do not need to take this problem into account. However, chances that a mobile device is pointing upward are clearly very high, so we need to find an alternative way to represent rotations.

One solution to this problem could be to represent the rotation using three rotation matrices for the three degrees of freedom [39]. There is, however, an even more elegant way to represent rotations and provide a less computationally intensive solution at the same time: Quaternions [37].

2.2.4 Quaternions

A quaternion q is a four dimensional vector that contains a scalar q_0 and a vector part $q_1\mathbf{i}$, $q_2\mathbf{j}$ and $q_3\mathbf{k}$, where q_0, q_1, q_2 and q_3 are real numbers and $\mathbf{i}$, $\mathbf{j}$ and $\mathbf{k}$ are quaternion units [37].

Expresions of a quaternion:

$$q = q_0 + q_1\mathbf{i} + q_2\mathbf{j} + q_3\mathbf{k} \tag{2.1}$$

$$q = (q_0, q_1, q_2, q_3) \tag{2.2}$$

We need additional definitions to be able to compute the orientation using quaternions.

Six quaternion axioms:

$$i^2 = j^2 = k^2 = -1 \tag{2.3}$$

$$ij = k \tag{2.4}$$

$$jk = i \tag{2.5}$$

$$ki = j \tag{2.6}$$

Product of quaternions:

$$pq = (p_0 + \vec{p})(q_0 + \vec{q}) \tag{2.7}$$

$$= (p_0q_0 + p_0\vec{q} + q_0\vec{p} + \vec{p}\vec{q}) \tag{2.8}$$

$$= (p_0q_0 - \vec{p} \cdot \vec{q} + p_0\vec{q} + q_0\vec{p} + \vec{p} \times \vec{q}) \tag{2.9}$$

Addition of quaternions:

$$p + q = (p_0 + \vec{p}) + (q_0 + \vec{q}) \tag{2.10}$$

Quaternion length:

$$|q| = \sqrt{qq^*} = \sqrt{q_0^2 + q_1^2 + q_2^2 + q_3^2} \tag{2.11}$$

Quaternion conjugate:

$$q^* = q_0 - q_1 i - q_2 j - q_3 k \tag{2.12}$$

Note that every quaternion other than the additive identity 0 has an inverse:

$$q^{-1} = \frac{q^*}{|q|^2} \tag{2.13}$$

The beauty of rotating a three dimensional vector v by a quaternion q is that it is just a cross product been the quaternion and the vector and another cross product by the conjugation of the quaternion. This leads to low computational cost since we only have to compute this two cross products leading to great performance compared to the rotation matrix [38].

Rotation of ${}^{A}v$ by the quaternion ${}^{A}_{B}q$:

$${}^{B}v =^{A}_{B} q \times^{A} v \times^{A}_{B} q^* \tag{2.14}$$

From quaternion to Euler angles [19]:

$$\theta = \arctan 2(2q_2q_3 - 2q_1q_4, 2q_1^2 + 2q_2^2 - 1) \tag{2.15}$$

$$\psi = -\sin^{-1}(2q_2q_4 + 2q_1q_3) \tag{2.16}$$

$$\varphi = \arctan 2(2q_3q_4 - 2q_1q_2, 2q_1^2 + 2q_4^2 - 1) \tag{2.17}$$

Now that we have laid the mathematical basis for quaternions, we will use them in our algorithm. Chapter 3 shows why they are a perfect solution to our problem.

2.2.5 Rotation Algorithm

There are various algorithms which could be used to calculate the orientation based on the already mentioned sensors. For our system however, the primary requirements are:

- The ability to work with low sampling rates (around 10 Hz)
- To be as efficient as possible, which implies low computational intensity
- Delivering a real time AHRS

A prominent algorithm commonly used to compute orientation is the "Kalman filter", which is accurate as well as effective. But as the Kalman filter requires sampling rates between 512 Hz and 30 kHz and is computationally intense, this approach would not cover all our requirements [13, 19]. Another widespread algorithm is the Mahony algorithm, which is able to handle low sampling rates. However, it is even more computationally demanding than the Madgwick algorithm [2]. Unfortunately, the Mahony algorithm only works for IMUs. Since we need an AHRS, it can not cover all our requirements [20, 19].

2.3 Step Detection

2.3.1 Traveled distance

In theory, the traveled distance could be easily computed by integrating the acceleration twice. The real-world problem with this approach is that sensor data contains noise. By integrating this noise twice, we get an enormous error that can not be overlooked [14].

For this reason, we have decided to implement a step detection service to provide real-time updates, which is lightweight and can handle the input of various device types and their sensors.

2.3.2 Scalar Product of the Accelerometer Data

One common way to implement step detection is to use the accelerometer data and try to recognize steps by analyzing the behavior of the data stream. In the following Figure 2.1 you can see the scalar product of the raw accelerometer data stream while walking very intensely. The data represented in this Figure were measured during the testing of the *step detection* in the building of the Institute of Computer Science at the Neubrückstrasse 10 in Bern. As a client device we used a Samsung Galaxy S7 Edge (introduced in chapter 5), the data was recorded on the cloud-based server.

Scalar product of the raw accelerometer data:

$$|a| = \sqrt{a_x^2 + a_y^2 + a_z^2} \tag{2.18}$$

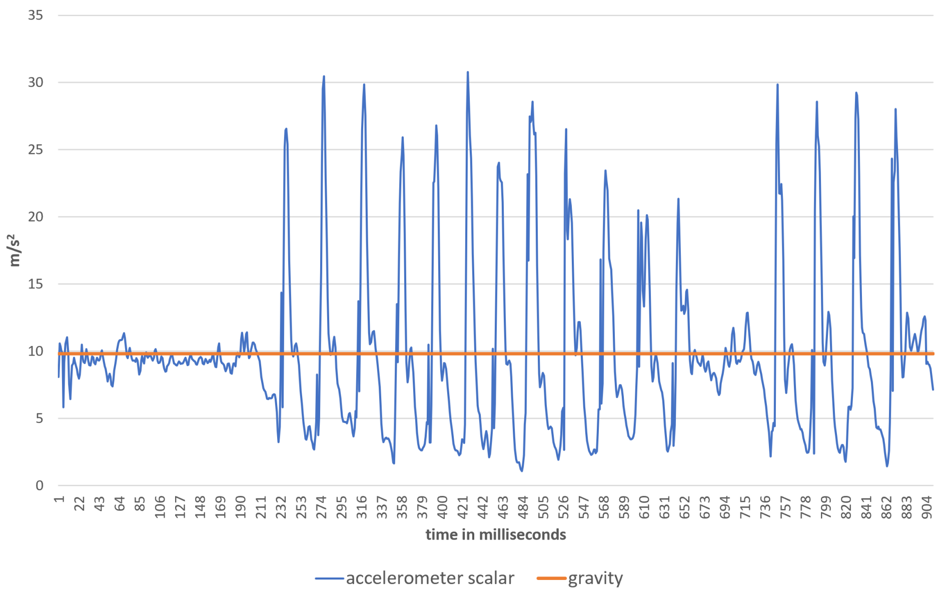


FIGURE 2.1: Line chart of the accelerometer scalar product while walking very intensely and the constant gravity at 9.81 m/s^2

The basic idea behind step detection using accelerometer data is really simple. We have a neutral value at 9.81 m/s^2 which represents gravity. When a device is still, the scalar product of the accelerometer data $|a|$ would be close to this value. Once a subject starts moving, $|a|$ would start oscillating since the device would be accelerated up and down during each step.

To recognize steps from the accelerometer scalar product, we need to define a sequence of accelerometer scalar products to be recognized as a step. Let us therefore define a *jerk* shown in Figure 2.2. It starts from the neutral position, goes up to a maximum, drops back to the neutral position before going down to a minimum and rising back up again to the neutral position. The data represented in Figure 2.2 are just a short extract of the data used in Figure 2.1.

Still lacking is a definition of when precisely a jerk is a step. This will be explained in chapter 3 section 3.3.

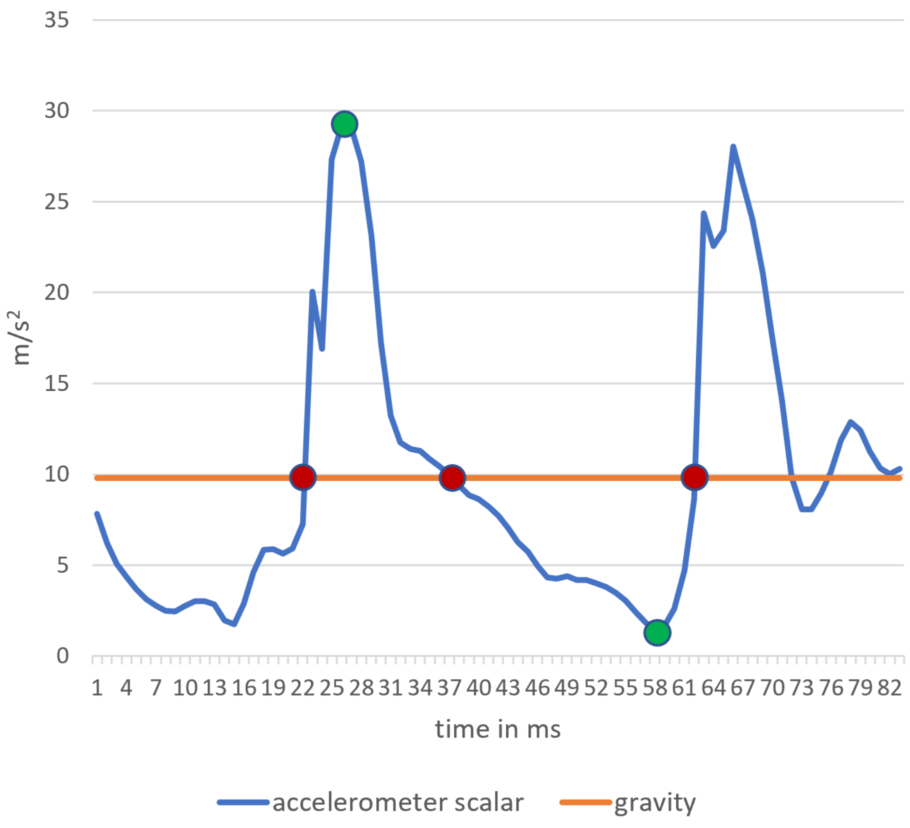


FIGURE 2.2: A Sequence defined as jerk

Chapter 3

System Architecture

In chapter 2 we introduced the theoretical and mathematical foundations which will now be applied. We will start off by introducing the architecture of the *Android client*.

Next, we will focus on the central themes of this thesis and dive deeper into the architecture and functionality of the Madgwick algorithm implemented in *rotations* and the *step detection* algorithms.

3.1 Implementation Architecture

3.1.1 Android Client

The architecture of the Android client is based on Android architecture components, which are a collection of libraries to design testable and easily expandable applications using a Model View ViewModel (MVVM) pattern [4]. Furthermore they provide:

- Livecycle-aware components that simplify the life cycle of an activity
- LiveData objects that can notify views when database entries change
- ViewModel which stores UI data that survives app rotations
- "Room", which is a SQLite library to reduce boilerplate code and convert between tables and Java objects

Due to this architectural decision, we could design the *Android client* as shown in Figure 3.1. By initializing a ViewModel, "Activities" provides a user interface and defines live cycles for the classes below. The logic is coordinated in the *AndroidClientRepository*, which starts a foreground service called *DataService* and saves data to the databases using the *AccessPointDao* and *LocationDao*. The *DataService* manages the gathering of data using the helper classes *AccessPointGathering* and *SensorDataGathering*. When receiving data from the gathering classes, the classes above are notified automatically because of the LiveData objects, while the *JSONDataSender* class sends the data to the server using a *Socket*.

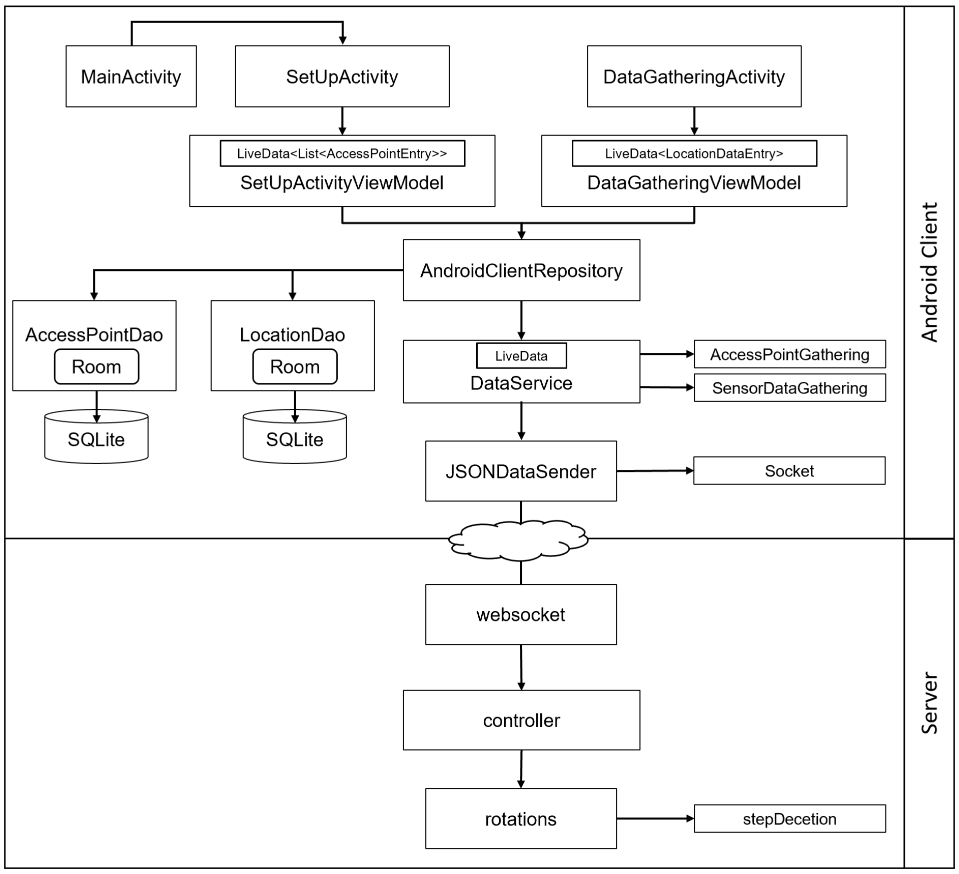


FIGURE 3.1: System architecture of the relevant parts

3.1.2 Server

The server receives the data in a *websocket* class and initializes the *controller*, which sends the received data to the whole service. This includes *rotations*, which in turn supply the AHRS by implementing the Madgwick algorithm, explained in great detail in subsection 3.2.

In addition, initializing and observing an instance of the *step detection* introduced in subsection 3.3 will allow *rotations* to be alerted if required.

3.2 Madgwick Algorithm

The Madgwick algorithm is a sensor fusing algorithm providing an AHRS by combining accelerometer, gyroscope and magnetometer data such that the resulting orientation has less uncertainty than would be possible when these sensors were used individually. This is achieved by taking advantage of the strengths of each of these sensors and minimizing their weaknesses as described in chapter 2 subsection 2.2.2. The Madgwick algorithm is implemented in the class *rotations* (shown in Figure 3.1) running on the server.

Should an elevator pitch about the Madgwick algorithm [19] applied in this work ever be needed, the author of this thesis would propose something like this:

"We take the angular velocity provided by the raw gyroscope date and integrate it to get the angle. By doing that we remove noise but add some drift to the result. Now we react to this drift by keeping the orientation in relation to the gravity and north by infusing the accelerometer and magnetometer data."

In mathematical terms, this means we set the gyroscope data as vector part of a quaternion ω, calculate the rate of change q', and compute the orientation $q_{\omega,t}$ by integrating. Please keep in mind that q' and $q_{\omega,t}$ are defined in the earth's frame, relative to the sensor frame.

Gyroscpe data:

$$\omega = (0, \omega_x, \omega_y, \omega_z) \tag{3.1}$$

Rate of change:

$$q' = \frac{1}{2} q \times \omega \tag{3.2}$$

Orientation:

$$q_{\omega_t} = q_{est,t-1} + q'_{\omega,t} \Delta t \tag{3.3}$$

As previously mentioned, the next step of the algorithm is to infuse the accelerometer and magnetometer data to get the orientation relative to the gravity and north.

According to the paper *An efficient orientation filter for IMUs and MARG sensor arrays* [19], this is best achieved by using a gradient descent algorithm, because it is a simple and fast algorithm. By multiplying the Jacobian and the objective function, we get the gradient, which is then used for the gradient algorithm [19].

Gradient:

$$\bigtriangledown f({}^S_E q_k, {}^E d, {}^S s) = J^T({}^S_E q_k, {}^E d) f({}^S_E q_k, {}^E d, {}^S s) \tag{3.4}$$

Gradient descent algorithm:

$$q_{k+q} = q_k - \mu \frac{\bigtriangledown f({}^S_E q_k, {}^E d, {}^S s)}{|| \bigtriangledown f({}^S_E q_k, {}^E d, {}^S s)||}, k = 0, 1, 2, 3, ..., n \tag{3.5}$$

Since gravity in earth's frame has only one vector component in the z direction and the magnetometer data is split up into the two directions b_x and b_z relative to the earth frame due to the orbital inclination, we can simplify the equation.

Gravity:

$$^E g = (0 \quad 0 \quad 0 \quad 1) \tag{3.6}$$

Accelerometer:

$$^S a = (0 \quad a_x \quad a_y \quad a_z) \tag{3.7}$$

Gradient descent algorithm inserted ${}^{E}g$ and ${}^{S}a$:

$$\nabla f({}^{S}_{E}q, {}^{S}a) = \begin{pmatrix} -2q_3 & 2q_4 & -2q_1 & 2q_2 \\ 2q_2 & 2q_1 & 2q_4 & 2q_3 \\ 0 & -4q_2 & -4q_3 & 0 \end{pmatrix} \begin{pmatrix} 2(q_2q_4 - q_1q_3) - a_x \\ 2(q_1q_2 + q_3q_4) - a_y \\ 2(\frac{1}{2} - q_2^2 - q_3^2) - a_z \end{pmatrix} \tag{3.8}$$

Magnetic field:

$${}^{E}b = (0 \quad b_x \quad 0 \quad b_z) \tag{3.9}$$

Magnetometer:

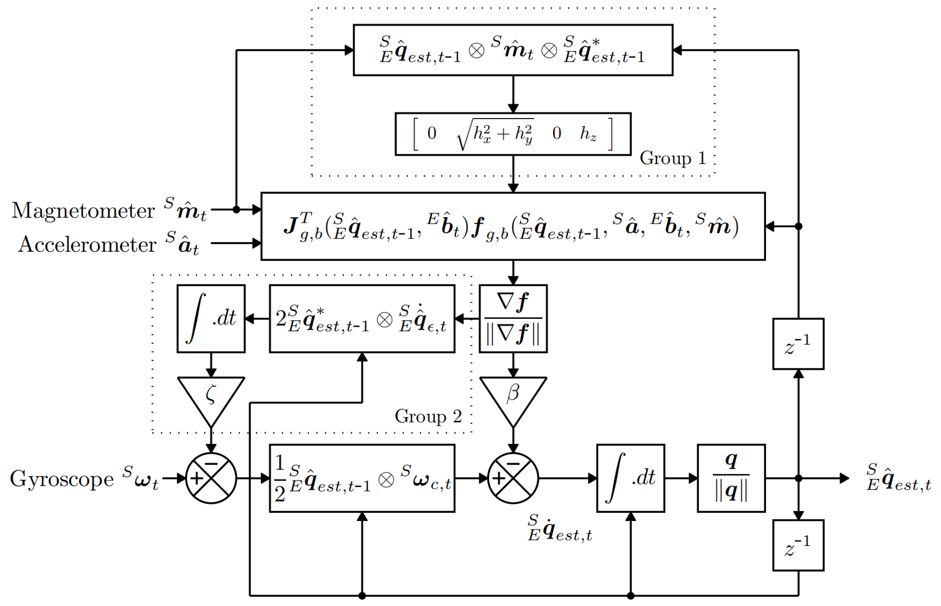
$${}^{S}m = (0 \quad m_x \quad m_y \quad m_z) \tag{3.10}$$

Gradient descent algorithm inserted ${}^{E}b$ and ${}^{S}m$:

$$\nabla f({}^{S}_{E}q, {}^{E}b, {}^{S}m) = \begin{pmatrix} -2b_zq_3 & -2b_xq_4 + 2b_zq_2 & 2b_xq_3 \\ 2b_zq_4 & 2b_xq_3 + 2b_xq_1 & 2b_xq_4 - 4b_zq_2 \\ -4b_xq_3 - 2b_zq_1 & 2b_xq_2 + 2b_zq_4 & 2b_xq_1 - 4b_zq_3 \\ -4b_xq_4 + 2b_zq_2 & -2b_xq_1 + 2b_zq_3 & 2b_xq_2 \end{pmatrix}^T \begin{pmatrix} 2b_x(\frac{1}{2} - q_3^2 - q_4^2) + 2b_z(q_2q_4 - q_1q_3) - m_x \\ 2b_x(q_2q_3 - q_1q_4) + 2b_z(q_1q_2 + q_3q_4) - m_y \\ 2b_x(q_1q_3 + q_2q_4) + 2b_z(\frac{1}{2} - q_2^2 - q_3^2) - m_z \end{pmatrix} \tag{3.11}$$

Having gathered all the parts required for the Madgwick algorithm to work, we are ready to fuse them to create our AHRS. As shown in Figure 3.2, we are now rotating the current sensor data of the accelerometer ${}^{S}a_t$ and magnetometer ${}^{S}m_t$ from the sensor frame to the earth frame using equation (2.2.4) and the last state of the quaternion ${}^{S}_{E}q_{est,t-1}$. In group 1, we then proceed to fix the effect of an erroneous inclination of the measured direction of the earth's magnetic field. This can be achieved by computing ${}^{E}b_t$ from ${}^{E}h_t$ which only has x and z components by taking the scalar product of the x and y components of ${}^{E}h_t$. In group 2 the gyroscope drift is compensated.

The accelerometer and magnetometer data are now relative to the earth frame and can be inserted to the gradient descent algorithm to compute the drift of the gyroscope. Having computed this drift, we are then able to correct the integrated gyroscope data and subtract the drift.



Magnetometer ${}^S\hat{\boldsymbol{m}}_t$
Accelerometer ${}^S\hat{\boldsymbol{a}}_t$
Gyroscope ${}^S\boldsymbol{\omega}_t$
${}^S_E\hat{\boldsymbol{q}}_{est,t\text{-}1} \otimes {}^S\hat{\boldsymbol{m}}_t \otimes {}^S_E\hat{\boldsymbol{q}}^*_{est,t\text{-}1}$
$\begin{bmatrix} 0 & \sqrt{h_x^2+h_y^2} & 0 & h_z \end{bmatrix}$
Group 1
$\boldsymbol{J}^T_{g,b}({}^S_E\hat{\boldsymbol{q}}_{est,t\text{-}1}, {}^E\hat{\boldsymbol{b}}_t)\boldsymbol{f}_{g,b}({}^S_E\hat{\boldsymbol{q}}_{est,t\text{-}1}, {}^S\hat{\boldsymbol{a}}, {}^E\hat{\boldsymbol{b}}_t, {}^S\hat{\boldsymbol{m}})$
$\int .dt$
$2{}^S_E\hat{\boldsymbol{q}}^*_{est,t\text{-}1} \otimes {}^S_E\dot{\hat{\boldsymbol{q}}}_{\epsilon,t}$
$\frac{\nabla \boldsymbol{f}}{\|\nabla \boldsymbol{f}\|}$
ζ
β
Group 2
z^{-1}
$\frac{1}{2}{}^S_E\hat{\boldsymbol{q}}_{est,t\text{-}1} \otimes {}^S\boldsymbol{\omega}_{c,t}$
${}^S_E\dot{\boldsymbol{q}}_{est,t}$
$\frac{\boldsymbol{q}}{\|\boldsymbol{q}\|}$
${}^S_E\hat{\boldsymbol{q}}_{est,t}$
z^{-1}

FIGURE 3.2: Block diagram representing the AHRS [19]

3.3 Step Detection

3.3.1 Jerk Average Threshold

Step detection subdivides the incoming accelerometer stream into multiple jerks (defined in subsection 2.3.2) by calculating its scalar product and checking if a jerk value is bigger than 65% of the step jerk average threshold. If this is the case, the jerk will be treated as a step and be normalized, multiplied by the current average $s_{avg}(i-1)$ and added to the current step jerk sum $s_{sum}(i-1)$ [3].

Normalized step j_i:

$$\hat{j}_i = 1.6 - \frac{1.6}{j_i + 1} \tag{3.12}$$

Step jerk sum:

$$s_{sum}(i) = s_{sum}(i-1) + \hat{j}_i s_{avg}(i-1) \tag{3.13}$$

Step jerk average threshold:

$$s_{avg}(i) = \frac{s_{sum}(i)}{i} \tag{3.14}$$

The advantage of calculating a jerk average threshold is that the algorithm reacts dynamically to various walking styles. In addition, devices can deliver more or less reactive accelerometer data streams.

3.3.2 Pace Buffer

Aside from the dynamically calculated jerk average threshold, we are using a pace buffer that works in similar ways: it calculates the average time between the last 20 steps and makes sure that no jerk, which is recognized as a step, is unrealistically close to the previously detected step [3].

Chapter 4

Implementation

While chapter 3 focused on the overall architecture and the design decisions, this chapter will introduce the implementation of the *Android Client*, the *rotations* and *step detections*.

We will start by introducing the technology stack used as well as the proposed connection between cloud and clients.

4.1 Technology Stack

4.1.1 Server

Tornado Server Version 4.5.2 provides both, a WebSocket- as well as a regular web server. This project requires both. For the database we used Peewee 3.0.19 which supports SQLite [32, 22].

Python Version 3.6 has been used to implement the logic of the cloud-based service [26].

4.1.2 Client

Android clients run a version of Android and feature the following specifications:

- **Motorola Moto X Style**
  - **OS:** Android 6.0
  - **Processor:** Hexa-core 4x1.4 GHz Cortex-A53 & 2x1.8 GHz Cortex-A57
  - **Internal sensors:** Accelerometer, gyroscope, proximity sensor, compass
  - **Memory:** 3 GB RAM
- **Samsung Galaxy S7 Edge**
  - **OS:** Android 7.0
  - **Processor:** Octa-core 4x2.3 GHz Mongoose & 4x1.6 GHz Cortex-A53
  - **Internal sensors:** Fingerprint (front-mounted), accelerometer, gyroscope, proximity sensor, compass, barometer, heart rate, SpO2
  - **Memory:** 4 GB RAM

The Android clients were predominantly used to gather the WiFi signal strength and the sensor data form the accelerometer, magnetometer and the gyroscope. In terms of performance, no difference was identified. Both phones were equally used during development.

4.1.3 Software

While developing and evaluating the cloud-based indoor positioning system, we used the following tools:

- **PyCharm** to write the server side code [23]
- **Android Studio** for the client side code [7]
- **R Studio** for the statistical analysis and data processing [27]
- **Bitbucket** as version control repository hosting service [8]
- A **Spreadsheet** to double check the statistical analysis done with R Studio and to create some of the diagrams [12]

4.2 Connection

The communication between the server and the client is provided by a WebSocket connection sending JSON data. The data exchange is initiated by the client, which uploads an initial message indicating:

- The **surrounding access points** by sending the BSSID.
- The **capabilities** that are provided by the client, such as WiFi or Bluetooth, used for ranging.

The server responds to this message by informing the client about:

- The **access points of interest** by sending the BSSID of the access points that should be observed by the client.

After the initialization process, the clients begins transmission of:

- **Sensor data** gathered form the accelerometer, gyroscope and magnetometer.
- **RSSI values of the access points**. This information will be appended to the JSON data only periodically, since it is not gathered as fast as the sensor data. Full scans require around 2 seconds, depending on the amount of available access points.

  Furthermore every JSON package contains:
- A **random integer** number between 1 and 99 which will be returned by the server after receiving the package. This helps to detect missing packages. Currently, the Android client ignores these responses, because there was never an issue with data loss.

When communication is interrupted, the Android client will immediately take notice and automatically reconnect to the server by sending another initialization message.

4.3 Implementation Details

This section describes the flow of the relevant aspects of this implementation and points out some noteworthy details in regard to the architecture introduced in chapter 3. Moreover the described flow is visualized in Figure 4.2 as a sequence diagram. Each point of the following list is drawn into this sequence diagram with the corresponding number and a red circle around it.

1. Upon booting up, an Android application first checks its **application permissions**. Since Android version 6.0, this needs to be done during runtime. Due to the fact that the Android client needs eight different permissions, a small method was created to check permissions anytime a corresponding Activity is launched. An Activity is a crucial component of an Android app, and the way activities are launched and put together is a fundamental part of the platform's application model. Unlike programming paradigms in which apps are launched with a main() method, the Android system initiates code in an Activity instance by invoking specific callback methods that correspond to specific stages of its lifecycle [17].

2. Because of the **Android Architecture Components**, there is only little logic in the Activity itself. However, the ClientRepository will be notified to start the foreground service through the ActivityViewModel.

3. The **foreground service** is needed to protect our logic from the Android operating system which constantly tries to minimize energy consumption by killing all long-running, unprotected processes.

4. The foreground service starts the **sensor** and **access point gathering**. The resulting data is formatted as JSON and sent to the server over WebSocket, implemented in the **JSONDataSender** class.

5. The server passes the JSON packages to ***rotations*** where a **dynamic sample rate** for the Madgwick algorithm is computed on the fly using timestamps. This is required because due to data being sent in JSON packages, our system can not provide the algorithm with a continuous data stream at a constant sampling rate. While the interval between measurements is constant, time between transmissions is not always predictable. Therefore, it is essential to compute the sample rate dynamically.

6. Once the ***rotations*** receive the first data package, the hard iron **magnetometer calibration** method will be executed to improve the AHRS results.

7. Afterwards, and while the Madgwick algorithm continues to be fed with sensor data, the accelerometer data is passed to the ***step detection*** which determines steps by observing the fluctuations of the accelerometer data stream. Both is implemented as described in chapter 3.

8. To notify the *rotations* about a step detected by the *step detection*, the system is using a simple **observer pattern** allowing the *step detection* to push data anytime to the subscribing *rotations* object.

9. As soon as the *rotations* are triggered by the observer, a two dimensional **step vector** (x, y) is created with x referring to east and y to the earth's magnetic field. This step vector is calculated by applying trigonometry using the computed heading and a predefined standard step length of 65 cm [40].

10. This vector is now passed to the particle filter which calculates the current position and forwards this information to the web client for visualization. Both are described in greater detail in another thesis of this project.

11. Another **visualization** that is part of this work and can be enabled when needed is the one representing the current state of the AHRS. In a pop up window created with the pygame library we display six differently colored rectangles that together form a cuboid and are implemented with functions provided by the OpenGL library. This cuboid will then permanently be adjusted to the current orientation computed by the *rotations* by rotating the whole cuboid using the OpenGL function glRotatef [25, 24]. Even though it does not increase the performance of the system, it was very convenient to measure progress, which otherwise would have been barely noticeable. An example of this visualization is shown in Figure 4.1.

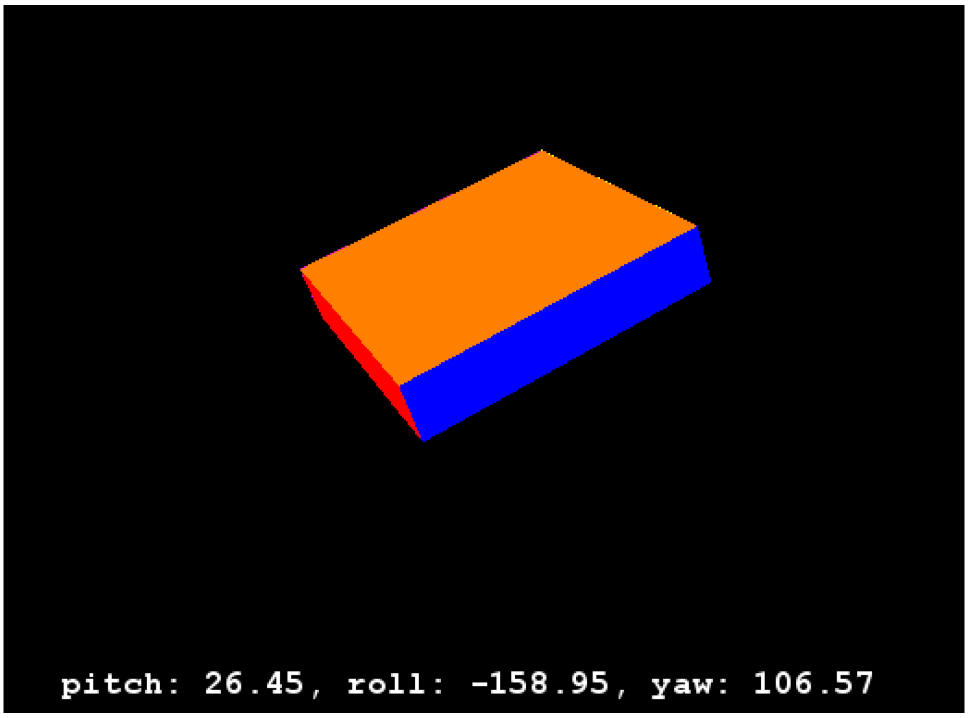


FIGURE 4.1: Visualization of the current AHRS state

4.3.1 Training and indoor positioning

The entire cloud-based indoor positioning system can be used to train and create the required regression models for a particular room. This regression model, as well as other information about the environment such as floor plan or access point positions, will then be used for the indoor positioning, which is of course the main use case.

The whole environment can be configured over a web interface currently running on port 8101 of the server.

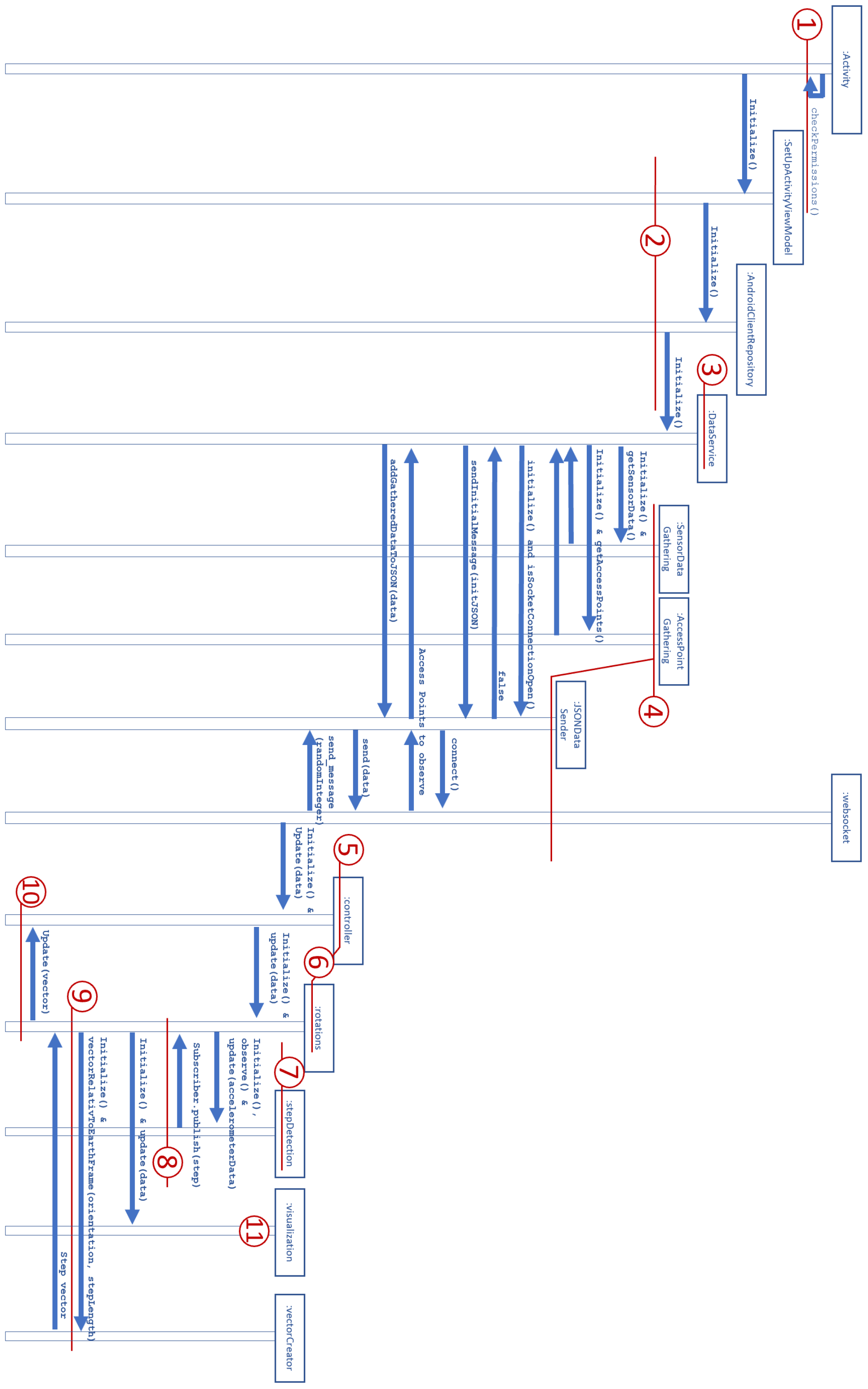

:Activity
:SetUpActivityViewModel
:AndroidClientRepository
:DataService
:SensorData Gathering
:AccessPoint Gathering
:JSONData Sender
:websocket
:controller
:rotations
:stepDetection
:visualization
:vectorCreator
checkPermissions()
Initialize()
Initialize()
Initialize()
Initialize() & getSensorData()
Initialize() & getAccessPoints()
initialize() and isSocketConnectionOpen()
false
sendInitialMessage(initJSON)
Access Points to observe
addGatheredDataToJSON(data)
connect()
send(data)
send_message (randomInteger)
Initialize() & Update(data)
Initialize() & update(data)
Initialize(), observe() & update(accelerometerData)
Subscriber.publish(step)
Initialize() & update(data)
Initialize() & vectorRelativeToEarthFrame(orientation, stepLength)
Step vector
Update(vector)
1
2
3
4
5
6
7
8
9
10
11

FIGURE 4.2: Sequence diagram showing the system flow

Chapter 5

Evaluation

In this chapter we evaluate the performance of the *rotations* and the *step detection*.

5.1 Evaluation Methods

5.1.1 Introduction

To evaluate the accuracy of an AHRS is quite challenging. The performance of an AHRS algorithm depends strongly on the strategies to reject perturbations like sudden accelerations or deformations of the earth's magnetic field. To be able to estimate the biases of the gyroscopes accurately, we depend on an environment that is free from interference [21]. These requirements cannot be met in most buildings.

Another challenge is to define the aspects to be examined in relation to the AHRS. Options include a sequence of a continuous data stream or some snapshots to predefined time points. The investigation of a continuous data stream or gyroscopic data would require some kind of measuring arm capable of performing different predefined movements multiple times. Therefore, this approach was refused. Instead, we attempted to test both parts at the same time, partly because they should work concurrently anyway.

As already explained in chapter 4, *rotations* are running steadily and continuously while the particle filter will be notified about the current orientation only when the *step detection* detects a step. To satisfy this logic in the evaluation, it was decided to define a path with 22 steps, each step 65 cm in length, containing three turns as shown in Figure 5.1.

The path chosen had to be rather short in order for the individual steps from different solutions (introduced in 5.1.2) to be compared with each other. This would not be possible for longer paths, as missed steps would accumulate and therefore preclude a precise comparison between solutions.

To get additional data, the path was walked five times, every time in a slightly different walking style while keeping the same walking rhythms, and always providing a five seconds calibration phase at the beginning of a walk.

5.1.2 Comparison with Android Framework Solution

To be able to compare the results of the *rotations* and the *step detection*, we made use of the Android framework to implement evaluations on the clients. Since the Android framework documentation mentions two different ways to determine the orientation, we decided to implement both solutions.

The experiments have been divided into three different groups that will be referred to as follows:

- And1 - This implementation is using the Android sensor of the type rotation vector, which returns quaternions. These quaternions are then converted to an Euler Angle relative to the earth's frame as introduced in chapter 2 and 3 using my own code [6].
- And2 - This implementation returns a rotation matrix, that is converted to an Euler Angle relative to the earth's frame by an Android Framework function [5].
- Rot - This part is running on the server of the cloud-based indoor positioning system, as mentioned in chapter 4.

Since the same step detection is used by both client-side solutions (And1 and And2), we will refer to the Android step detection as And2.

5.1.3 The predefined walking path for the experiment

To prepare the walking path for the 22 steps, the 23 marks, 65 cm apart, were measured by hand and sticked to the ground as visualized in Figure 5.1. The heading of the room relative to the earth's magnetic field was measured with the Google Earth Pro desktop application using the function "ruler", because results gained this way were much more accurate compared to the hand compass "Suunto A-30", which was first used [15, 30]. By defining this path, we were able to verify the accuracy of the three AHRS orientations by comparing them to the actual one of the room.

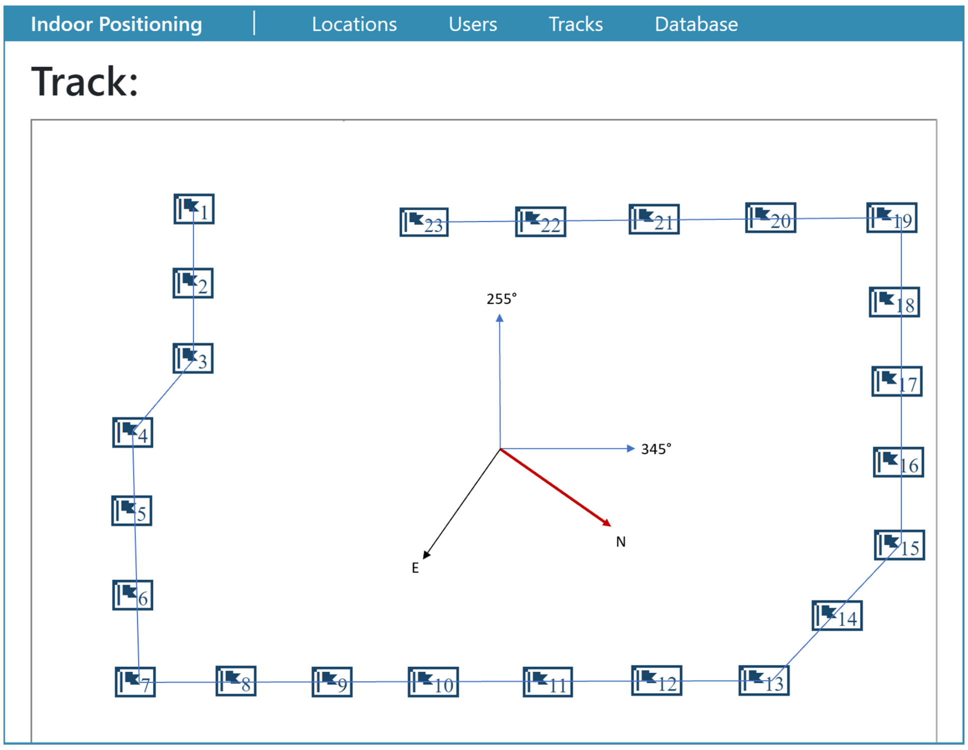


FIGURE 5.1: Visualization of the predefined walking path

5.1.4 Experimental Data

Since the starting and ending process of the experiment strongly and negatively affected the experimental data, we decided to cut off the last samples of the test generously. Therefore, only the data generated while walking and holding the phone in the hands was used for comparisons. Furthermore, the time differences between steps that were recognized by the two step detection implementations "And2" and "Rot" were normalized so they could be compared. Note that delays of the step detection would differ, even if both methods delivered reasonable and comparable results. We assume this is due to Rot losing time while transmitting the data to the server.

Since the chosen walking rhythms were always steady, the unrecognized steps could be easily estimated by checking the normalized time differences $t_{i,norm}$ between the detected steps. If the normalized difference between two steps $t_{i,norm}$ was larger than the average t_{avg}, multiplied by 1.6, a step was missing. After adding the missing row, the average was subtracted from the normalized difference to check if a second step was missing.

Since And1, And2 and Rot return the computed values in an interval of -180 to 180 degrees, they were transformed to an interval of 0 to 360 degrees. After preprocessing, 83 positions were available to compare rotations, and 88 to compare the *step detection*.

5.2 Rotation Evaluation

With this data set, we could compare three different implementations of the AHRS in 83 situations.

The first hypothesis H_0 evaluated whether or not the three implementations are significantly different from each other. Since three groups have been compared, a standard t-test is not applicable. To evaluate this, the choice fell on an analysis of variance (ANOVA) test, as ANOVA is useful to in regard to statistical significance [34, 33].

To get to the results of the test shown in Table 5.1, we had to do the following calculations:

The sums of square of the three groups (SS_{Rot}), (SS_{And1}) and (SS_{And2}) where $y_{j,i}$ are the experimental data and $\hat{y}_j$ the the means of the three groups:

$$SS_j = \sum_{i=1}^{n} (y_{j,i} - \hat{y}_j)^2 \tag{5.1}$$

$$j = \{Rot, And1, And2\} \tag{5.2}$$

The sum of square within groups (SSW) containing the three sums of square (SS):

$$SSW = SS_{Rot} + SS_{And1} + SS_{And2} \tag{5.3}$$

Then we joined all three groups into one group and calculate the total sum of squares (SST).

The total sum of squares (SST) where y_i are all the joined experimental data from the three groups and $\hat{y}$ is the mean:

$$SST = \sum_{i=1}^{n}(y_i - \hat{y})^2 \tag{5.4}$$

The sum of squares between groups (SSB):

$$SSB = SST - SSW \tag{5.5}$$

The degree of freedom (Df) between groups:

$$Df_{SSB} = groups - 1 \tag{5.6}$$

The degree of freedom (Df) within groups:

$$Df_{SSW} = (groups * observations) - groups \tag{5.7}$$

The mean of squares (mean sq) between groups:

$$Meansq_{SSB} = \frac{SSB}{Df_{SSB}} \tag{5.8}$$

The mean of squares (mean sq) within groups:

$$Meansq_{SSW} = \frac{SSW}{Df_{SSW}} \tag{5.9}$$

The F-value:

$$F = \frac{Meansq_{SSB}}{Meansq_{SSW}} \tag{5.10}$$

5.2.1 ANOVA

	Df	**Sum sq**	**Mean sq**	F	$F_{0.05;2,250}$
between groups	2	45857.40	22928.70	11.28	3.03
within groups	246	50040.50	2032.68		

TABLE 5.1: ANOVA test

If we define the level of significance which is the probability of rejecting the null hypothesis when it is true to be at 0.05, we get a critical value $F_{0.05;2,250}$ = 3,03, where 0.05 describes the level of significants, 2 the Df between and 250 the Df within groups (rounded up from 246).

Now we only need to compare our calculated F-value with the one we looked up in the F distribution table [33]. As 11.28 > 3.03 we got to reject the $H0$-hypothesis, which means the three different implementations are statistically significantly different.

5.2.2 Tukey's Honestly Significant Difference (HSD) Test

To find out which of the implementations are statistically significantly different from each other, we decided to run a HSD called "Tukey's Range Test" [42]. The main idea of this test is to compute the significant difference between two means using a statistical distribution defined by the q distribution. This distribution provides the exact sampling distribution of the largest difference between a set of means. All pairwise differences are evaluated using the same sampling distribution used for the largest difference. This why the HSD approach is quite conservative and works well to check our hypotheses [1].

To run the HSD test shown in Table 5.2, we need to know n, which is the number of observations, the *Df numerator* which is the number of groups that were compared, as well as the *Df denominator*, which is the degree of freedom within groups Df_{SSW} we already used for the ANOVA test in Equation 5.5.

Next we defined the level of significants α and looked up the Q-Value for a Df numerator of 3 and a level of significants α of 0.05 [29].

We then calculated the average of the within group variances means we define as v_{avg}, the *Absolute Differences* δ and the *Critical Range* θ.

Absolute Difference δ:

$$\delta = |mean_i - mean_j| \tag{5.11}$$

$$where: 1 <= i < j <= n \tag{5.12}$$

Critical Range θ:

$$\theta = Q * \sqrt{\frac{v_{avg}}{n}} \tag{5.13}$$

$$\tag{5.14}$$

To determine if a result is significant, we compare the *absolute difference* with the *critical range*. If the *absolute difference* is smaller than the *critical range*, the difference between the two compared groups is not significant.

Setup			
n	83	α	0.05
Df numerator	3	Q-value [29]	3.314
Df denominator	246	v_{avg}	2008.19
Comparison	**Absolute Difference**	**Critical Range**	**Result**
Rot to And1	25.59	16.30	significant
Rot to And2	5.57	16.30	not significant
And1 to And2	31.17	16.30	significant

TABLE 5.2: Tukey's Honestly Significant Difference (HSD) test

The results show that the accuracies achieved by the Rot, which is our implementation of the Madgwick algorithm called *rotations* running on the cloud-based server, and the And2 calculated on the client do not differ significantly. The client side implementation And1, however, performs significantly worse compared to the other two implementations Rot and And2.

It is therefore evident that *rotations* with its AHRS does not outperform And2 offered by the Android Framework. Nonetheless, in our test, the difference was not significant.

Figure 5.2 shows a graphic comparison of the two solutions Rot and And2, where the y-axis represents the deviation in degrees of the actual orientation and the two computed orientations from the AHRS systems. The x-axis shows all the steps we took into account during the experiment described in subsection 5.1.3.

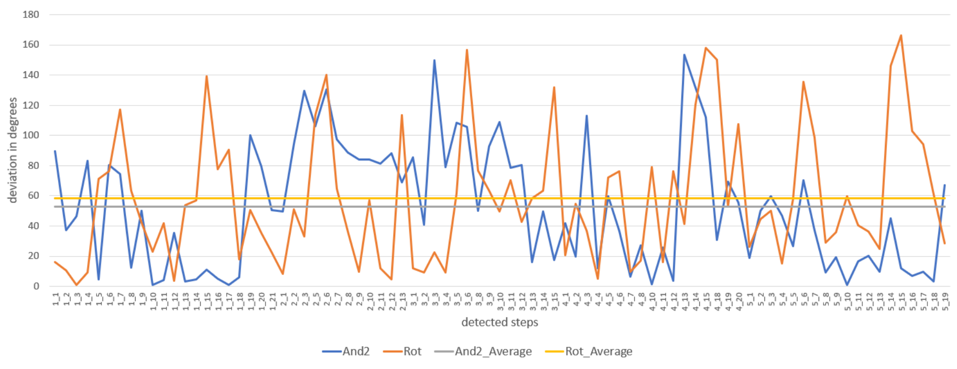


FIGURE 5.2: Diagram showing the difference between the computed orientations and the actual orientation, as well as their averages.

5.3 Step Detection Evaluation

After preprocessing, 88 steps were left which could be compared. Out of these, the client side step detection using the Android framework And2 recognized 87, and the server side code provided by the proposed Rot recognized 84 steps. Out of 100 steps, the Android solution would recognize 98 and our solution 95 steps. To test whether or not step detection of the Android framework And2 works significantly better than the proposed solution, a Pearson χ^2-square test was conducted [34, 10]. This test is used to determine whether there is a significant difference between the *expected frequency* and the observed frequency, while the frequencies stand for the rate at which the steps were detected or not during the experiment.

5.3.1 Pearson χ^2-square Test

To run a Pearson χ^2-square test we setup the *start table* as shown in Table 5.3 with all the cell values called Value(cell), the total of the row called $total_{row}$, the total of the columns called $total_{column}$ and the overall total ($total_{overall}$). Using this information, we calculated the *expected frequency* $f_{exp}(cell)$ for each cell, the *Test statistic* t_{sta} for each cell and finally the χ^2-*value*.

Expected frequency f_{exp} for each cell:

$$f_{exp}(cell) = \frac{total_{row}(cell) * total_{column}(cell)}{total_{overall}} \tag{5.15}$$

Test statistic t_{sta} for each cell:

$$t_{sta}(cell) = \frac{(Value(cell) - f_{exp}(cell))^2}{f_{exp}(cell)} \tag{5.16}$$

χ^2*-value*:

$$\chi^2 = \sum^{cells} t_{sta}(cell) \tag{5.17}$$

Start table $(Value(cell))$	**Recognized**	**Not recognized**	$\boldsymbol{total_{column}}$
Rot	87	1	88
And2	84	4	88
$\boldsymbol{total_{row}}$	171	5	176
Expected frequency $f_{exp}(cell)$			
Rot	85.5	2.5	
And2	85.5	2.5	
Test statistic $t_{sta}(cell)$			
Rot	0.026315789	0.9	
And2	0.026315789	0.9	
$\boldsymbol{\chi^2}$**-value**	1.853		
$\boldsymbol{\chi^2_{0.05;1}}$**-value**	3.842		

TABLE 5.3: Pearson χ^2-square test comparing the step detections form Rot and And2

Now we can compare the calculated χ^2-value with the critical value $\chi^2_{0.05;1}$ = 3.84 for a level of significants of 0.05 and a degree of freedom between groups of 1 (as described in equation 5.4) [10].

Since 1.852 < 3.84, the hypothesis that the solution of the Android framework is significantly better than the one proposed by this work is to be rejected.

5.4 Evaluation Summary

There are no significant differences between the Rot and the And2 solutions. But they are both not precise enough to predict reasonably accurate positions after being triggered by the step detection solutions. Due to the quick consecutive turns defined in the experiment and the rather deep sampling rate provided to the AHRS algorithms, they are not able to recover after a direction change, leading to inaccurate indications of direction.

As it can be inferred from Figures 5.2 and 5.1, the AHRS algorithms can not react fast enough and determine the new orientation after a direction change. This is because after changing direction the algorithms need a certain number of accelerometer and magnetometer measurements to be able to correct the gyroscope drift.

The primary goal of the rotations is not to provide an exact location, but rather to provide a first estimate while being emended only afterwards by means of the particle filter. Therefore, the starting requirements are fulfilled nevertheless.

The *step detector* is a simpler algorithm, which provides more accurate results. It was, however, slightly surpassed by the Android Framework solution. During our tests, it was noticed that the walking style had a tremendous influence on the performance of both solutions.

As seen in the experiment, there are no significant differences between the step detections of algorithms Rot and And2 while walking and holding the phone in hands.

Chapter 6

Conclusion

During this thesis, we presented parts of a cloud-based indoor positioning system. The aim was to offload resource-intensive algorithms from various types of low-powered mobile devices to the cloud.

The two parts that were emphasized on and evaluated are:

- *Rotation* detection – an implementation of an AHRS
- *Step detection* to notify the service about a step

Rotations The implemented AHRS system, which makes use of the Madgwick algorithm, delivers not significantly worse results compared to the better one of the implemented client side solutions as shown in Table 5.2. This can be interpreted as a success, as the server side computation requires splitting up and then rejoining the sensor data stream, before passing them to the algorithm. The client-side solution, on the other hand, is able to process the continuous data stream without interruptions, but the CPU-intensive algorithms will empty the batteries of the mobile devices in a short period of time and are therefore not suitable for running on devices with limited energy resources.

Taking all this into account, a system which performs comparably but is much more flexible can be seen as progress.

Due to the slow sampling rate, the performance of the AHRS systems implemented does not fit into a highly dynamic use case, as the gyroscopic drift cannot be corrected immediately. A better use case for the AHRS algorithms would be to determine the current orientation to a mapping application that is used while standing still. However, since the current position of our cloud-based indoor positioning service is provided by the particle filter and will be corrected through WiFi trilateration, the proposed AHRS implementation can meet the requirements demanded, even though delivered directions are ambiguous.

Step Detection The implemented *step detection* delivered reliable results with an accuracy of 95% and can extend the functionality of the entire cloud-based indoor positioning service. This is considered as a success.

Further Thoughts Even though the implementation of the AHRS *rotations* is slightly less accurate and the *step detection* is only on the same level compared to the Android version, there are still some great advantages by offloading the logic and computations to a cloud-based service:

- Resource intensive computations will not stress the limited resources of low-powered devices.
- The service can be used by a wider range of devices, providing results derived from complex computations to simple devices like an ESP32 microcontroller.

Possible Future Work It would be interesting to evaluate whether or not performing computations on the server consumes less energy than client-side computations, despite all the traffic flowing through the WebSocket interface.

Moreover, future work might compare the entire cloud-based indoor positioning service to other positioning approaches in order to validate its effectiveness.

Bibliography

[1] Herve Abdi and Lynne Williams. “Tukey’s Honestly Significant Difference (HSD) Test”. In: (Aug. 2018).

[2] *adafruite Sensor Fusion Algorithms*. `https://learn.adafruit.com/ahrs-for-adafruits-9-dof-10-dof-breakout/sensor-fusion-algorithms`. Accessed: 2018-06-21.

[3] *Adaptiv An Adaptive Jerk Pace Buffer Step Detection Algorithm*. `https://github.com/danielmurray/adaptiv`. Accessed: 2018-08-02.

[4] *Android Developers Android Architecture Components*. `https://developer.android.com/topic/libraries/architecture/`, note = Accessed: 2018-04-02.

[5] *Android Developers SensorManager.getRotationMatrix @ONLINE*. July 2018. URL: `https://developer.android.com/reference/android/hardware/SensorManager`.

[6] *Android Developers Sensor.TYPE_ROTATION_VECTOR @ONLINE*. July 2018. URL: `https://developer.android.com/reference/android/hardware/SensorEvent`.

[7] *Android Studio @ONLINE*. Aug. 2018. URL: `https://developer.android.com/studio/`.

[8] *bitbucket Built for professional teams @ONLINE*. Aug. 2018. URL: `https://bitbucket.org/`.

[9] S. S. Chawathe. “Beacon Placement for Indoor Localization using Bluetooth”. In: *2008 11th International IEEE Conference on Intelligent Transportation Systems*. 2008, pp. 980–985. DOI: `10.1109/ITSC.2008.4732690`.

[10] *Chi-square table Chi-square table @ONLINE*. Aug. 2018. URL: `https://www.di-mgt.com.au/chisquare-table.html`.

[11] *Espressif ESP32 @ONLINE*. Aug. 2018. URL: `https://www.espressif.com/en/products/hardware/esp32/overview`.

[12] *Excel @ONLINE*. Aug. 2018. URL: `https://products.office.com/de-ch/excel`.

[13] Li J. Zhang X. Shen C. Bi Y. Zheng T. Liu J. Feng K. “A New Quaternion-Based Kalman Filter for Real-Time Attitude Estimation Using the Two-Step Geometrically-Intuitive Correction Algorithm.” In: *Sensors (Basel, Switzerland)* 17.9 (2017), p. 2146. ISSN: 1424-8220. DOI: `10.3390/s17092146`.

[14] *Freescale Semiconductor Implementing Positioning Algorithms Using Accelerometers*. `https://www.nxp.com/files-static/sensors/doc/app_note/AN3397.pdf`. Accessed: 2018-06-12.

[15] *Google Earth Pro @ONLINE*. Aug. 2018. URL: `https://www.google.com/earth/download/gep/agree.html`.

[16] *ibeacon.com What is iBeacon? A guide to beacons @ONLINE*. Aug. 2018. URL: http://www.ibeacon.com/what-is-ibeacon-a-guide-to-beacons/.

[17] *Introduction to Activities @ONLINE*. Sept. 2018. URL: https://developer.android.com/guide/components/activities/intro-activities.

[18] Eric M. Jones and Paul Fjeld. "Gimbal Angles, Gimbal Lock, and a Fourth Gimbal for Christmas". In: *apollo lunar surface journal* (2011).

[19] S.O.H. Madgwick, R. Vaidyanathan, and A.J.L. Harrison. *An Efficient Orientation Filter for Inertial Measurement Units (IMUs) and Magnetic Angular Rate and Gravity (MARG) Sensor Arrays*. Tech. rep. Department of Mechanical Engineering, 2010. URL: http://www.scribd.com/doc/29754518/A-Efficient-Orientation-Filter-for-IMUs-and-MARG-Sensor-Arrays.

[20] R. Mahony, T. Hamel, and J. Pflimlin. "Nonlinear Complementary Filters on the Special Orthogonal Group". In: *IEEE Transactions on Automatic Control* 53.5 (2008), pp. 1203–1218. ISSN: 0018-9286. DOI: 10.1109/TAC.2008.923738.

[21] Estefania Munoz Diaz et al. "Evaluation of AHRS algorithms for inertial personal localization in industrial environments". In: 2015 (June 2015), pp. 3412–3417.

[22] *Peewee 3.0.19 @ONLINE*. Aug. 2018. URL: https://github.com/coleifer/peewee.

[23] *PyCharme @ONLINE*. Aug. 2018. URL: https://www.jetbrains.com/pycharm/.

[24] *pygame @ONLINE*. Sept. 2018. URL: https://www.pygame.org.

[25] *PyOpenGL @ONLINE*. Sept. 2018. URL: http://pyopengl.sourceforge.net.

[26] *Python 3.6 @ONLINE*. Aug. 2018. URL: https://www.python.org/downloads/release/python-360/.

[27] *RStudio @ONLINE*. Aug. 2018. URL: https://www.rstudio.com/.

[28] *Sensors online Compensating for Tilt, Hard-Iron, and Soft-Iron Effects*. https://www.sensorsmag.com/components/compensating-for-tilt-hard-iron-and-soft-iron-effects. Accessed: 2018-08-07.

[29] *Studentized Range q Table Studentized Range q Table @ONLINE*. Aug. 2018. URL: http://www.real-statistics.com/statistics-tables/studentized-range-q-table.

[30] *Suunto A-30 @ONLINE*. Aug. 2018. URL: https://www.suunto.com/de-ch/Produkte/Kompasse/Suunto-A-30/Suunto-A-30-NH-USGS-Compass/.

[31] *Technology, Media and Telecommunications Predictions*. https://www2.deloitte.com/global/en/pages/technology-media-and-telecommunications/articles/tmt-predictions.html. Accessed: 2018-07-25.

[32] *Tornado What's new in Tornado 4.5.2 @ONLINE*. Aug. 2018. URL: http://www.tornadoweb.org/en/stable/releases/v4.5.2.html.

[33] *Universität Koeln F-Verteilung für (1-a)=0,95*. `http://eswf.uni-koeln.de/glossar/fvert3.htm`. Accessed: 2018-08-18.

[34] *Universität Zürich Methodenberatung*. `https://www.methodenberatung.uzh.ch/de.html`. Accessed: 2018-08-18.

[35] Wan Mohd Yaakob Wan Bejuri and Mohd Mohamad. "Wireless LAN/FM radio-based robust mobile indoor positioning: An initial outcome". In: 8 (Jan. 2014), pp. 313–324.

[36] *Wikipedia Accelerometer*. `https://en.wikipedia.org/wiki/Accelerometer`. Accessed: 2018-08-07.

[37] *Wikipedia Quaternion*. `https://en.wikipedia.org/wiki/Quaternion`. Accessed: 2018-07-24.

[38] *Wikipedia Quaternions and spatial rotation*. `https://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation`. Accessed: 2018-08-22.

[39] *Wikipedia Rotation matrix*. `https://en.wikipedia.org/wiki/Rotation_matrix`. Accessed: 2018-06-18.

[40] *Wikipedia Trigonometry*. `https://en.wikipedia.org/wiki/Trigonometry`. Accessed: 2018-08-12.

[41] *Wikipedia Trilateration*. `https://en.wikipedia.org/wiki/Trilateration`. Accessed: 2018-08-07.

[42] *Wikipedia Tukey's range test*. `https://en.wikipedia.org/wiki/Tukey's_range_test`. Accessed: 2018-08-04.