

# 1 Software-Entwicklung und Experimente

Aus den anfänglichen Ideen für die Umsetzung des Lighthouse Keeper entstehen in diesem Kapitel die ersten Module für dessen Verwirklichung. Zum Anfang wird die verwendete Hardware vorgestellt und in Verbindung dazu, die Aufgaben der genutzten Software-Komponenten näher erläutert. Durch die geeignete Wahl aller Bestandteile, lassen sich viele Aufgaben auf vorgefertigten Module übertragen. Beispielsweise werden statt einer eigenen Entwicklung, ein bestehendes Kommunikations-Framework und eine bereits existierende Middleware für den Roboter genutzt. Mit der Nutzung dieser fertigen Lösungen wird im großen Maße Arbeitsaufwand eingespart und ermöglicht einen stärkeren Fokus auf das eigentliche Thema. Da dies im Hinblick auf die Smartphone-Applikation nicht immer möglich war, wird deren Implementierung in diesem Kapitel exemplarisch einmal dargestellt.

Die durchgeführten Experimente beziehen sich in diesem Teil der Arbeit auf die Messung der Signalstärke der Beacons in Abhängigkeit zu deren Entfernung. Diese Phase in der Prozessplanung legt den Grundstein für die Modellbildung, welche anschließend im nächsten Kapitel vorgenommen wird. Ferner werden einzelne Einflüsse auf die Messungen näher betrachtet und eine Analyse zu der Batterilaufzeit eines Beacons in Abhängigkeit zu seinen Einstellungen aufgestellt.

## 1.1 Verwendete Hardware

Da die Beacon-Technologie vorrangig zur Indoor-Lokalisierung von Personen eingesetzt wird und als Peripheriegerät meistens ein Smartphone Verwendung findet, wird auch ein solches für die gesamte Testdauer als Messgerät genutzt. Die Wahl fiel dabei auf ein Android-Smartphone mit dem Namen Motorola Moto G der gleichnamigen Firma Motorola Inc. Es wurde ausgewählt, weil es alle Hardware- und Software-Anforderungen zum Empfang von BLE-Signalen erfüllt und als ein Standard-Smartphone gilt, sodass sich mit den ihm erzielten Ergebnisse auch auf andere Produkte übertragen lässt. Des Weiteren werden zwei Roboter in den Experimenten genutzt, um die Messungen reproduzierbar und standardisiert durchzuführen. Für die reinen Distanz-Signalstärke-Messungen wird der Roboters bzw. lediglich sein Arm namens „Youbot“ der Kuka AG und später für die Validierung einer Beacon-Konfiguration der Roboter „Scitos G5“ der MetraLabs GmbH verwendet.

### 1.1.1 Motorola Moto G

Das Moto G dient als Empfangsstation der BLE-Signale, dessen grundlegende Spezifikationen ein 1,2 GHz Snapdragon 400 Prozessor mit 1 GB RAM und ein WCN3620 BT/FM/WLAN RF Modul aussmachen [? ]. Seine Abmaße betragen 129.9 mm × 65.9 mm × 11.6 mm bei einem Gewicht von 143 g. Desweiteren verwendet es standardmäßig ein Android 4.3 als Betriebssystem, welches jedoch für die Experimente auf die Version 4.4 geupdated wurde. Neben der technischen Ausstattung und Software sind für die späteren Messungen die Antennen und deren Charakteristiken von hoher Bedeutung, denn deren Eigenschaften wirken sich direkt auf den Empfang der Signale aus. Um die Einflüsse besser zu verstehen, sind in den Abbildungen 1.1 und 1.2 die Anordnung der Antennen einmal skizziert. Dabei fällt es auf, dass sich WLAN- und Bluetooth-Modul die selben Antennen teilen. Dies führt zu der Frage, ob es zu Konflikten in der Funktionsweise des Smartphones kommt, wenn gleichzeitig auf beide Module zugegriffen wird. Jedoch dazu mehr im Abschnitt der App-Entwicklung. Die zweite Frage die sich daraus stellt, ist die Veränderung der Empfangs- und Sendequalität des Moto G in verschiedenen Positionen. Eine Antenne hat je nach Bauform und Funktionsweise Bereiche, in der sie mit voller Leistung sendet und empfängt, aber auch Bereiche in der Funk-Signale sie weder verlassen noch erreichen können. Das hat verschiedene physikalische Gründe, jedoch sind diesen komplexen nichtlinearen Eigenschaften der Antennen des Moto G zumaldest nicht öffentlich bekannt und können auch nicht einfach bestimmt werden. Es sei nur anzumerken, dass bei den Messungen auch darauf geachtet werden muss, wie und an welchen Halte-Punkten das Moto G am besten befestigt wird, ohne deren Transceiver-Fähigkeiten negativ zu beeinflussen. Eine gute Annahme dabei ist es das Smartphone so auszurichten, als ob es flach in der Hand eines Menschen liegen würde. Der Hersteller wird schließlich darauf bedacht sein, sein Produkt für einen normalen Betrieb auszulegen und somit auch die



Abb. 1.1: Vorderseite des Motorola Moto G [? ]

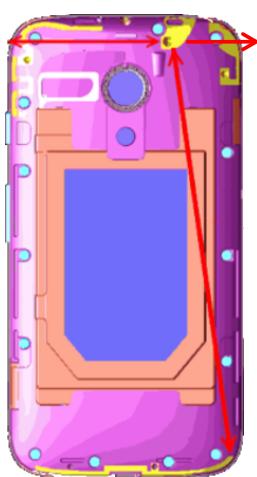


Abb. 1.2: Rückseite vom Moto G mit Antennen-Gerüst [? ]

Konstruktion und Bau der Antennen danach optimieren. Diese Annahme muss jedoch noch anhand von Messungen verifiziert werden.

### 1.1.2 Youbot

In vorigen Abschnitt wurde schon angesprochen, dass die Lageposition des Messinstrumentes zu seiner Empfangsleistung überprüft werden muss. Zudem soll das Smartphone so gehalten werden, als wenn es sich in einer flachen Hand befindet und so auch die Experimente durchgeführt werden. Um all dies zu erreichen und auch unter der Anforderung an einen automatisierten und reproduzierbaren Prozess, empfiehlt es sich einen Roboterarm als Mess-Plattform zu benutzen. Aufgrund der Verfügbarkeit wurde das Modell „Youbot“ der Firma „Kuka“ gewählt und für die Messungen mit einer Halterung aus einem 3D-Drucker regänzt (siehe Abbildung 1.3). Die Vorteile des Systems sind zum einen der montierte hochpräzise Roboterarm mit Greifer auf dem Youbot und zum anderen, dass ein vollwertiger Rechner mit einem Linux Betriebssystem und eine drahtlose WLAN-Schnittstelle im System verbaut sind und so die Kommunikation zum Roboter sehr einfach aufgebaut werden kann. Der künstliche Arm kann sich dabei um seine fünf Achsen drehen (siehe Abbildung 1.4) und bietet somit genug Möglichkeiten, die Lageposition vom Moto G zu verändern.



Abb. 1.3: Youbot mit Halterung (gelber Aufsatz)

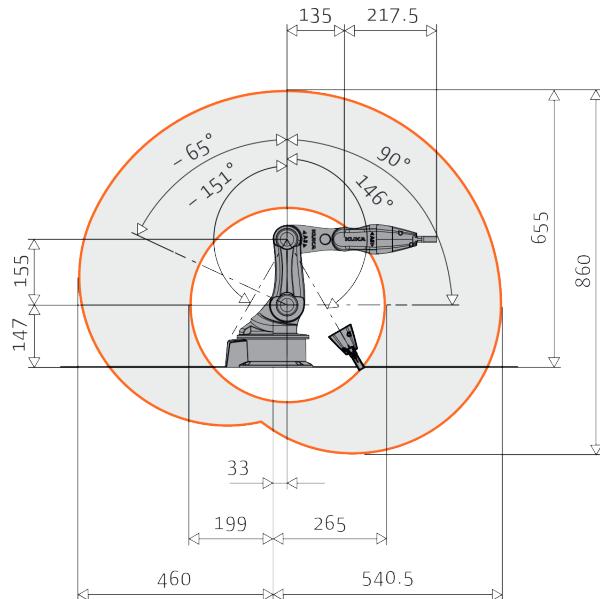


Abb. 1.4: Zeichnung eines Youbot-Arms und seiner fünf Rotationsachsen [? ]

### 1.1.3 Scitos G5

Die Durchführung der Evaluation einer Beacon-Konfiguration wäre grundsätzlich auch mit dem modifizierten Youbot aus Abschnitt 1.1.2 möglich. Jedoch wurde es in der

Aufgabenstellung gefordert, den Roboter „Scitos G5“ von der MetraLabs GmbH zu verwenden. Der Scitos G5 ist mit den Maßen  $55\text{ cm} \times 60\text{ cm} \times 60\text{ cm}$  etwas größer als der Youbot und durch seinen Dreirad-Lenkung weniger wendig (vgl. Abbildung 1.5). Die Vorteile gegenüber dem Youbot bestechen jedoch durch seine bessere Ausstattung und den größeren Software-Umfang, weswegen es keiner zusätzlichen Entwicklungen bedarf und somit den Arbeitsaufwand für das Lighthouse Keeper-Konzept verringert wird. Der Scitos G5 verfügt über Schrittmotoren, einem Laserscanner und Inertialsensoren zur Navigation und er besitzt zudem einen leistungsfähigen Intel Core i7-Prozessor, WLAN und ein Linux-Betriebssystem, sodass auch hier genug Ressourcen für eine Kommunikation vorhanden sind. Die Machbarkeit des gesamten Kontroll-Prozesses von Beacon-Konfigurationen beruht dabei auf der Leistungsfähigkeit von diesem Roboter, denn er muss in der Lage sein, seine Position unabhängig von der Triangulation von Beacon-Signalen zu bestimmen und diese als Referenzquelle zur Verfügung zu stellen. Für seine Positionsbestimmung verwendet er die Software-Umgebung „Miracenter“ (siehe 1.2.2 im nächsten Absatz), welche die Sensordaten aus der Wahrnehmung (Motoren, IMU, Laserscanner) so verwertet, dass sie eine Karte der Umgebung erstellen kann und somit im Abgleich von Sensorinformationen und Karte der Roboter stets seine Position kennt. Der Hersteller wirbt dabei mit einer Nutzlast von bis zu 50 kg und einer Maximalgeschwindigkeit von  $1,4\frac{\text{m}}{\text{s}}$  bei einer batteriebetriebenen Laufzeit von ca. 20 Stunden [?], sodass ein künstlicher Roboterarm auf dem Scitos G5 ebenfalls denkbar wäre und somit auch die Aufgaben vom Youbot zukünftig übernimmt.



Abb. 1.5: Scitos G5 von der MetraLabs GmbH

## 1.2 Verwendete Software

Um die Hardware zu nutzen und das geplante Konzept umzusetzen, benötigt es einer Kommunikation zwischen den Geräten und weiterer Werkzeuge zur Aufnahme von Messungen und deren Verarbeitung. Bei der Umsetzung wurde besonders auf Konfirmität der verschiedenen Systeme und deren reibungslosen Zusammenspiels geachtet.

Um eine gemeinsame Basis zu schaffen, wurde das Software-Framework „Robots Operating System“(ROS) verwendet. Mit einem gemeinsamen Standard lassen sich die Messungen besser vergleichen, wodurch ihre Qualität und Aussagekraft zunimmt. Die Messungen müssen dabei auf der Smartphone-Plattform und den Roboter-Plattformen aufgenommen und diese synchronisiert werden. Während ROS die übergeordnete Schnittstelle darstellt, müssen auf den einzelnen Hardware-Elementen die Messungen eigenständig durchgeführt werden. Die Messung auf dem Scitos-Roboter entfällt dabei auf die Software „Miracenter“ und funktioniert ohne weiteres. Die Software für die Messungen auf dem Smartphone ist hingegen nicht vorgefertigt und muss mithilfe eines Editors für Android-Applikationen und einer speziellen Bibliothek für die Kommunikation Smartphone ↔ Estimote Beacon entwickelt werden.

### 1.2.1 Robot Operating System – ROS

Das „Robot Operating System“ oder kurz ROS, ist ein Projekt zur Schaffung eines flexiblen Frameworks für die Entwicklung von Software für Roboter. Daraus entstand eine der mächtigsten Sammlungen aus Werkzeugen, Bibliotheken und Normen, um komplexes Verhalten zwischen Robotern über verschiedenste Robotik-Plattformen robust zu gestalten [? ]. Der Anwendungsbereich für ROS ist demzufolge sehr umfangreich, jedoch werden für diese Arbeit nur folgende Eigenschaften [? ] des Projektes benötigt und hier näher betrachtet:

- „Peer to Peer“ (P2P)-Verbindungen
- Modularer Aufbau
- Unterstützung mehrerer Programmiersprachen
- freier Nutzung und Open-Source

Das Kommunikation-Framework des Lighthouse Keeper-Konzeptes, welches auf ROS aufbaut, besteht dabei aus mehreren miteinander verbundenen Rechnern (sog. „Hosts“) die zur Laufzeit über P2P miteinander kommunizieren. Bei der P2P-Verbindung können alle Teilnehmer ihre Dienste bzw. ihre Informationen gleichermaßen einander anbieten und nutzen, indem sie Daten im Netzwerk gleichzeitig empfangen und senden können. Dabei läuft auf einem zentralen Server der eigentliche Kern des Frameworks, über den der sämtliche Datenverkehr geleitet wird. Und auf den Host-Systemen laufen die eigentlichen modularen Anwendungen („Nodes“), die über Schnittstellen („Sockets“) des Betriebssystems ihre Daten auf das Netzwerk und schließlich an ROS verteilen. Dadurch können die Nodes in verschiedenen Sprachen programmiert werden, die lediglich auf die entsprechende Schnittstelle zugreifen. Ausgehend von den Nodes werden den gesendeten Nachrichten gesonderte Bezeichnungen („Topics“) zugeordnet und mit einem Datentyp versehen. Diese Informationen sind allen Teilnehmer des P2P-Netzwerkes bekannt und können auch von ihnen angefordert werden. Der Kern organisiert dabei eine einheitliche Uhrzeit, die dadurch für alle Nachrichten bzw. deren Zeitstempel in den verteilten Systemen konsistent bleibt

und so eine Synchronisierung der Messgeräte nicht mehr nötig wird. Somit ermöglicht die Verwendung von ROS den Aufbau der gesamten Kommunikation, wie es in der Konzept-Planung in ?? gefordert wurde. Zudem erleichtert ROS durch sein großen Funktionsumfang nachfolgende Erweiterungen und bietet somit Freiheiten für zukünftige Aufgaben.

### 1.2.2 Miracenter

Das Paket „CogniDrive“, als Bestandteil der Software „Miracenter“ von der Firma MetraLabs GmbH, dient als Navigator des Scitos G5 und ermöglicht es mit ihm den Grundriss eines Raumes zu erstellen (siehe Abbildung 1.6) und zusätzlich die Lokalisierung des Roboters in diesem durchzuführen. Während beispielsweise der Roboter entlang der Wände fährt, messen seine Schrittmotoren den zurückgelegten Weg, die Inertialsenso- ren verfeinern die Informationen und erweitern sie um die Ausrichtung des Roboters. Dabei misst zusätzlich der Laserscanner die Distanz zu den Wänden und allen anderen festen Objekten in Reichweite. Aus der Fusion aller Messwerte lässt sich so der Grundriss des vermessenden Raumes generieren. Im nebenstehendem Bild ist eine solche Karte als Beispiel aufgeführt. Die Daten die Miracenter dem Nutzer zur Verfügung stellt, bestehen dabei aus der Position des Roboters und seiner Ausrichtung in der Karte. Die Karte liegt dabei als „Portable Network Graphics“ (PNG)-Datei vor, wobei ein Pixel einer konstanten metrischen Länge entspricht, deren Verhältnis in einer „Extensible Markup Language“ (XML)-Datei definiert ist. Die Farbwerte der Bildpunkte beschreiben zudem, ob an einer Stelle ein Hinderniss oder ein frei befahrbarer Raum vorliegt. In der Abbildung ist zu erkennen, dass die erstellte Karte teilweise verrauscht ist, Wände nicht gerade verlaufen, oder offene Türen und herumlaufende Personen die Messungen verfälschen. Die Karte kann dafür nach der Erstellung von jedem beliebigen Bildbearbeitungsprogramm geöffnet und bearbeitet werden. Dabei ist darauf zu achten die Auflösung nicht zu verändern, da ansonsten die Dimensionen nicht mehr übereinstimmen oder gegebenenfalls die XML-Datei angepasst werden. Nach der Bearbeitung der Bilddatei kann sie anschließend im Miracenter geladen werden. Mittels Mausklick in die Karte wird die ungefähre Position des Roboters bestimmt. Danach orientiert sich der Scitos G5 automatisch und lokalisiert sich im

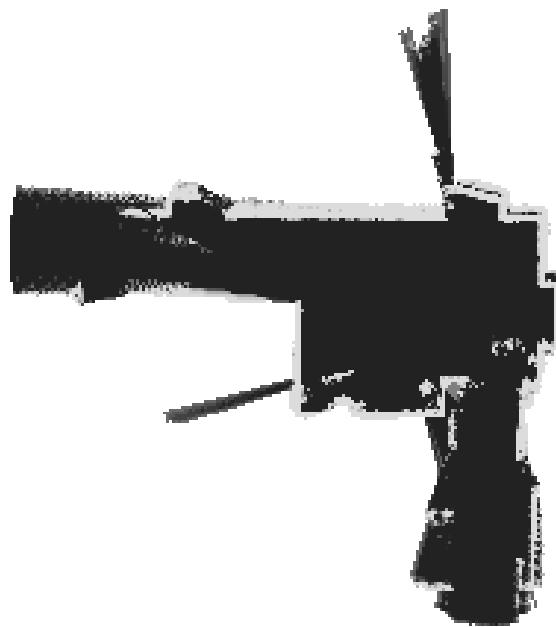


Abb. 1.6: Erstellter Grundriss eines Flures in Gebäude 29 der OvGU, Stockwerk 3

weiteren Verlauf selbst anhand seiner Sensordaten. Nun können ihm Positionen und Ausrichtung vorgegeben werden und durch seiner internen Pfadplanung steuert er sie autonom an. Durch ein Miracenter-ROS-Interface kann dabei die interne Lokalisierung und die Vorgabe von Position und Ausrichtung vom Scitos extern übermittelt werden. Die gesamte Software ist jedoch proprietär, d.h. nicht quell offen, sodass ein tieferer Blick in die Funktionsweise der Software hier verwehrt wird.

### 1.2.3 Android-Studio, Estimote SDK und ROSjava

Als letzten Baustein in der Software-Architektur fehlt die App für das Moto G, deren Entwicklung auf dem Zusammenwirken dreier Grundpfeiler aufgebaut sein wird:

1. Das Android-Studio [?] ist eine Entwicklungsumgebung für Programme speziell von Android-Smartphones. Es bietet die Mittel eine Applikation, basierend auf der Programmiersprache Java, auf ein Android-Gerät zu portieren und dort auch zu testen. Aufgrund der Implementierung der „Integrated Development Environment“ (IDE) in Java, lässt es sich auf beinahe jedem Betriebssystem verwenden und bietet dadurch eine hohe Flexibilität in der Anwendung.
2. Damit die Applikation die Beacon-Signale verarbeiten kann, müssen die gesendeten Nachrichtenpakete und das verwendete BLE-Protokoll in das Programm eingebettet werden. Vom Hersteller der Beacons wird dafür eigens eine Bibliothek unter dem Namen „Estimote Software Development Kit (SDK)“ [?] bereitgestellt, die sich problemslos integrieren lässt und einfach zu bedienen ist.
3. Um die empfangenen Signale an den Server via P2P zu senden, muss eigens eine Schnittstelle von der Java-basierten Android-App zum in C/C++ gehaltenen ROS implementiert werden. Dafür wird das Paket „Rosjava“ [?] benötigt, um die Unterstützung von ROS-Funktionen für Java-Programme zu realisieren. Dies erlaubt es ROS-Pakete in eine Android-App zu integrieren und dadurch die zu übermittelnden Nachrichten-Formate darin auch zu verwenden.

## 1.3 App-Entwicklung

Um das geplante Programm auf dem Smartphone zu verwirklichen, werden die aus 1.2.3 beschriebenen Hilfen und zudem eigener Programmcode benötigt. In diesem Abschnitt werden die einzelnen Elemente der App erläutert und ihr Zusammenspiel in der fertigen Applikation anschließend betrachtet.

### 1.3.1 Beacon-Detektierung

Der kritischste, aber der auch am essentiell wichtigste Programmteil ist die Aufnahme von Beacon-Signalen. Hierfür muss die SDK von Estimote in die App über das Android-Studio als Bibliothek importiert werden. Daraufhin stehen neue Klassen und Objekte

der IDE zur Verfügung, die speziell auf die Verwendung von Beacons zugeschnitten sind. Zum einen muss dabei die App bei jedem Start für die Nutzung mit Beacons mit Abfragen neu initialisiert werden, das wären zum Beispiel die Überprüfung, ob Bluetooth angeschalten ist oder ob BLE überhaupt auf dem Gerät unterstützt wird. Des Weiteren muss der sogenannte Beacon-Manager, welcher im Hintergrund der App läuft, mit Zeiten für die Pause zwischen zwei Abtastungen und der Länge eines Scans, eingestellt werden. Die Problematik der Abstastraten wird hierbei später im Programmablaufplan noch erörtert. Zum anderen muss eine Art Empfangs-Funktion geschrieben werden, die bis zur Beendigung des Programms auf Meldungen der Leuchtfeuer wartet und diese für die Verarbeitung im Programm aufbereitet. Die Anwendung von Filtern oder dergleichen entfällt hier, da schließlich das Verhalten der Signalausbreitung studiert werden soll und zusätzliche Einflüsse die Ergebnisse verfälschen könnten. Interessanter wird es bei der Frage, wie häufig die Abfragen der Eingänge nach Beacon-Signalen stattfinden sollen. Hierbei stellt sich noch eine viel wichtigere Frage und zwar der nach der Funktionsweise der SDK. Also wie die Datenströme, die von den Beacons in Intervallen gesendet, darauf vom Beacon-Manager aufgenommen werden. Denn die Sende-Intervalle der Beacons können variieren, während die Applikation mit festen Werten für alle Beacons initialisiert werden muss. Die Antwort nach den Abläufen im Beacon-Manager kann leider nicht beantwortet werden, da die Entwickler keinen Einblick in ihren Quellcode gewähren und auch sonst keine Angaben oder Dokumentationen veröffentlichen. Eine bessere Veranschaulichung dieser Problematik findet sich im Abschnitt über den Programmablauf.

### 1.3.2 Lagemessung

Parallel zur Beacon-Detektierung soll gleichzeitig die Lage des Smartphones gemessen werden, um die Antennen-Charakteristiken besser zu verstehen und ihre Einflüsse auf die Empfangsqualität zu untersuchen. Für diese Aufgabe müssen die Inertialsensoren oder auch englisch „Inertial Measurement Unit“ (IMU) aus Gyroskop, Beschleunigungssensor und Magnetometer vom Smartphone ausgelesen werden, was jedoch einfach umzusetzen ist, da die IDE nativ dafür Bibliotheken bereitstellt.

Bei der Messung stellt sich hierbei die Frage nach der Häufigkeit der Sensordatenabfragen, denn letztendlich senden die Beacon – zwar asynchron – ihre Signale alle 2 s bis 50 ms, jedoch können die Inertialsensoren in Intervallen von mehreren hundert Hertz ausgelesen werden. Aufgrund der höheren Verfügbarkeit der IMU-Daten, wäre es jedoch nicht zweckmäßig diese permanent über das Netzwerk zu senden und somit das Datenvolumen unnötigerweise künstlich aufzublähen. Es reicht daher völlig zu jedem Zeitpunkt an dem ein Messung nach Beacon-Signalen definiert ist, eine Lageposition vom Smartphone vorrätig zu haben. Jedoch lässt sich dabei durch die schnelleren Messungen die Qualität der Informationen mit einem Filter erhöhen. Im Hinblick auf noch genügend freie Rechenkapazitäten auf dem Smartphone (bisher lediglich Abfragen der Sensoren), wurde ein Komplementärfilter auch im

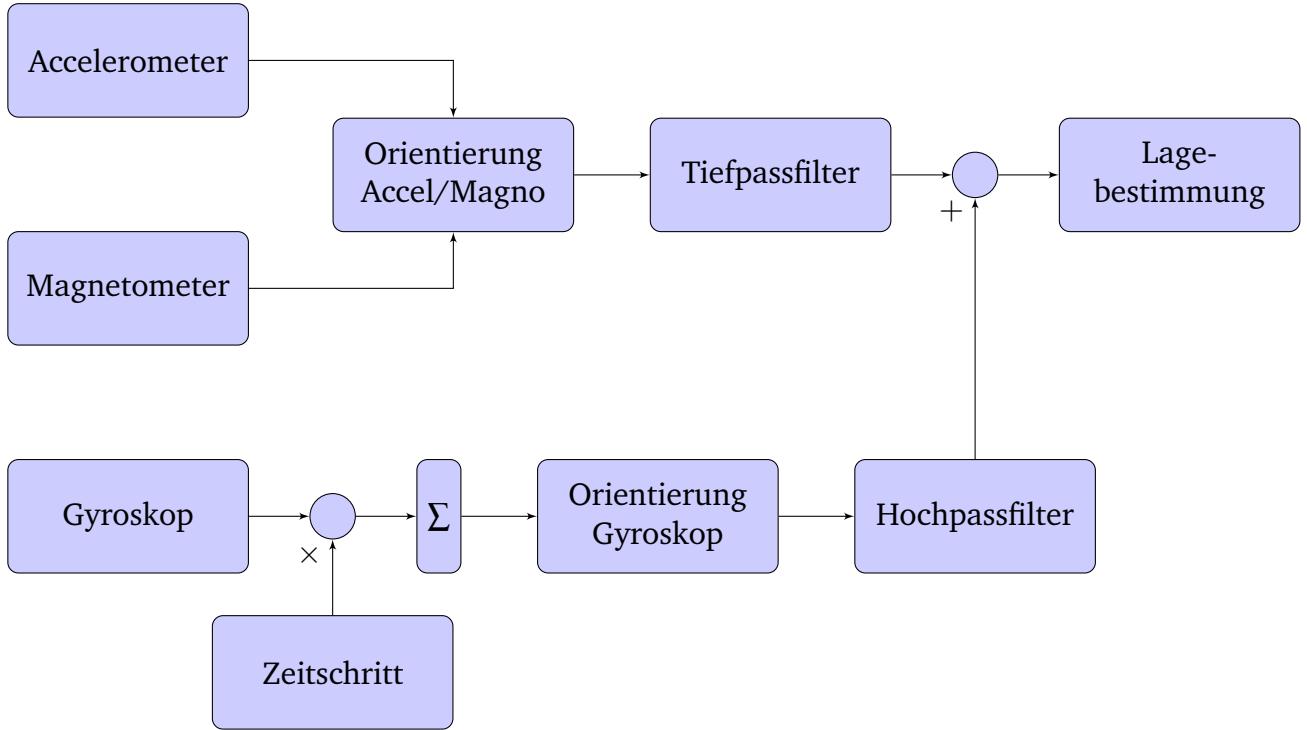


Abb. 1.7: Aufbau des Komplementärfilters, in Anlehnung an [? ]

Hinblick auf die Sensordatenfusion von den drei IMU-Sensoren gewählt. Der dazu nötige Quellcode wurde aus [?] entnommen, der im Grunde den Drift des Gyroskops und das Signalrauschen des Beschleunigungssensors und des Magnetometers entfernt. Der Aufbau ist diesbezüglich in Abbildung 1.7 einmal skizziert. Die gesamte Funktionsweise beruht auf dem Prinzip der Fusion mehrerer Daten, um die Qualität der eigentlichen Information zu erhöhen. Es würde für den Zweck der Orientierungserfassung des Smartphones ausreichen, die Information vom Schwerkraft-Vektors des Beschleunigungssensors mit der des Magnetometers zusammenzufassen und lediglich diese auszugeben. Jedoch sind besonders die Daten des Magnetometers sehr verrauscht, sodass ein Tiefpass oder anders ausgedrückt ein Mittelwert aus den Daten gebildet wird. Da das Moto G zusätzlich über eine Gyroskop verfügt, welches die Bewegung des Smartphones viel genauer messen kann, wird auch diese Informationsquelle benötigt. Aber auch hier existiert es ein Nachteil in dessen Anwendung. Und zwar der Drift, sprich das Aufsummieren/Integrieren der Fehler oder des Messrauschen über die Zeit. Um auch diesem Problem zu begegnen, werden die Daten nur über kleine Zeitabschnitte mithilfe eines Hochpasses gesammelt. Dadurch werden anhand der Daten aus Accelerometer und Magnetometer als Stützinformationen und die Daten vom Gyroskop als Informationen über schnelle Änderungen verwendet.

### 1.3.3 ROS-Anbindung

Die Realisierung der Datenübertragung über eine ROS-Schnittstelle in der App wurde so umgesetzt, dass nach dem Start der Applikation ein Feld zur manuellen Eingabe der Adresse des Servers bzw. des ROS-Masters (z.B. die Internetprotokoll (IP)-Adresse) eingegeben werden muss. Dies ist nötig, um einen Zielort für die Nachrichten zu bestimmen. Nach der manuellen Eingabe wartet der sogenannte „Publisher“ – also der Programmteil, der die Nachrichten aufbereitet und an das Netzwerk übermittelt – auf die Messdaten. Dabei muss für den Publisher eine Bibliothek bereitstehen, die die Anzahl der zu übertragenden Informationen und deren Datentypen definiert und in Java integrieren kann, um sie später in der App zu verwenden. Dazu muss, wie anfänglich erwähnt die ROSjava-Erweiterung installiert sein und anschließend das Message-Format als Java-Paket damit erstellt werden. Da die versendeten Daten schon in vorigen Abschnitten erläutert wurden, wird in Abbildung 1.8 nur noch eine Zusammenfassung in Form der Implementierung präsentiert. Das dargestellte Message-Format enthält dabei die UUID empfangener Beacons als Zeichenkette oder String-Wert, sowie der empfangenen Signalstärke RSSI und der eingestellten Sendeleistung (bezeichnet als Power), als 32 Bit Integer und der daraus errechneten Distanz als 64 Bit Fließkommazahl (ein zusätzliches Feature der Estimote SDK). Des Weiteren beinhaltet sie die errechnete Orientierung vom Moto G und über allem den Zeitstempel der Zusammensetzung der Nachricht. Natürlich existiert zwischen Aufnahme der Messungen und dem Sendezeitpunkt eine zeitliche Differenz (kleiner als 1 ms Bearbeitungszeit), welche jedoch für die langsame Dynamik des Systems in den ersten Experimenten zu vernachlässigen ist.

```
string UUID
int32[] RSSI
int32[] Power
float64[] Distance
float64[] Orientation
float64[] Time
```

Abb. 1.8: Quellcode-Beispiel eines Message-Paketes für eine ROSjava-Implementierung

### 1.3.4 Programmablauf

Nachdem die einzelnen Module fertig gestellt wurden, musste aus ihnen ein gesamtes Programm zusammengefügt werden. Bei der Vereinigung der einzelnen Programmteile kristallisierten sich dabei verschiedene Probleme, die unter anderem auch auf den verstärkten Einsatz von proprietärer Software zurückzuführen sind. Denn bei der notwendigen Verwendung von nicht quelloffener Software, können Fehlfunktionen der App, die eigentlich auf dem Versagen der genutzten Software beruht, lediglich umgangen oder zumindest mit Einschränkungen im Ablauf der Applikation behoben wer-

den. Im konkreten Fall verursachte die gleichzeitige Verwendung von WLAN und Bluetooth, nach anscheinend willkürlichen Nutzungszeiten der App, Abstürze derjenigen. Das Problem scheint hierbei primär bei Android zu liegen, weil der Sachverhalt schon seit zwei Jahren in diversen Online-Foren bekannt ist, jedoch der Fehler weiterhin auch in den neuesten Versionen des Betriebssystems auftritt [?]. Der günstigste Fall den Konflikt zu lösen, wäre es einfach die Nutzung von Bluetooth durch die Estimote SDK und dem Gebrauch der WLAN-Verbindung besser zu organisieren. Jedoch bietet die SDK keine Einstellmöglichkeiten und keinerlei Dokumentation über die Verwendung des Bluetooth-Adapters, sodass auch nicht auf Seiten der WLAN-Übertragung die Anpassung stattfinden kann. Da die Zeit fehlte, sich weder mit dem Entwicklerteam der Firma Estimote auseinander zu setzen oder sich in das Android-Betriebssystem einzurbeiten, war die Lösung eher primitiv als elegant. In Abbildung 1.9 ist die gesammelte App als Ablaufplan mit Blockschaltbildern dargestellt. Während die eigentliche Aufgabe der Applikation schon im Vorfeld erklärt wurde, fällt hier eine zusätzliche Funktion, hier als Kontroll-Funktion bezeichnet, auf. Diese ist im Grunde eine Sicherheitsabfrage, wo vor der Messung und der Sendung der Nachricht über den Publisher kontrolliert wird, ob eine Verbindung zum P2P-Netzwerk existiert. Es fiehl in den unzähligen Testläufen des Programmes auf, dass sobald die WLAN-Verbindung wegbricht, der Publisher kein Ziel für seine Nachrichten ausfindig machen kann und dadurch es logischerweise zum Veragen der Funktionen führt. Durch Zuhilfenahme der Debug-Funktion vom Android-Studio konnte auch die Zeile ausfindig gemacht werden, die zum Absturz des Programmes führt. Und zwar versucht der Publisher nach der missglückten Transmission ständig auf das WLAN-Modul zuzugreifen und wenn dazu noch der Beacon-Manager versucht, den Bluetooth-Adapter bei der Detektierung zu aktivieren, geschieht der Zusammenstoß und das Programm wird außerplanmäßig beendet. Nachdem mit der Abfrage schon die Messung bei fehlender WLAN-Verbindung unterbunden wird und die Systeme erst deaktiviert und dann einzeln reaktiviert werden, lief die App durchgehend 24 Stunden ohne Vorkommnisse.

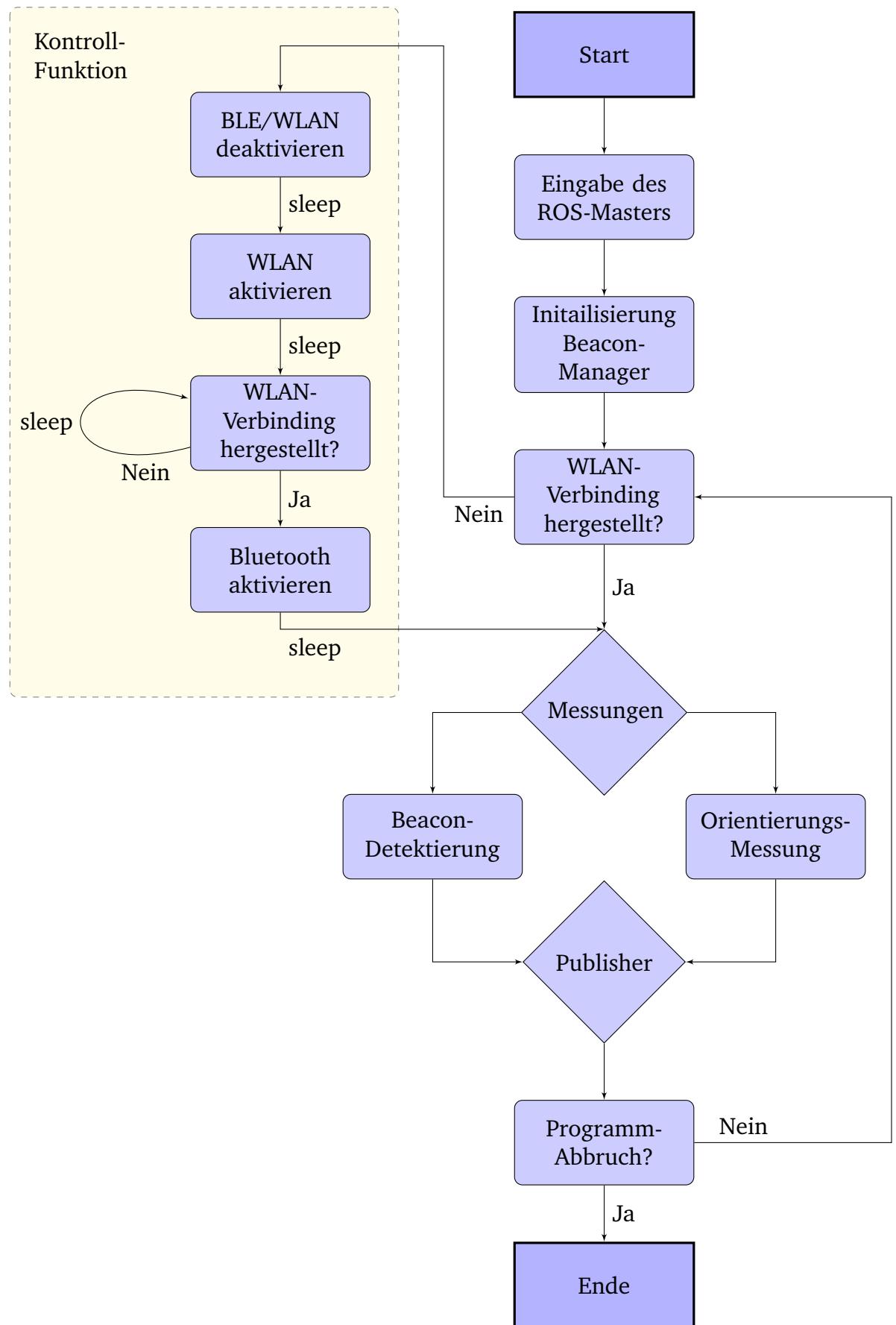


Abb. 1.9: Programmablaufplan der Lighthouse Keeper Applikation für Android-Smartphones

## 1.4 Experimente

Die hier vorgenommenen Versuche betrachten zum einen die Ausbreitung der Signale bei einer fest justierten, für die Arbeit vorher definierten Einstellung der Beacons. Zum anderen werden auch Aspekte der Beacon-Technologie betrachtet, die für spätere Forschungen von Interesse wären, aber auch die Notwendigkeit dieser Arbeit unterstreichen sollen. Denn es macht gerade den Reiz der Beacons aus, dass sie relativ einfach an ihre Bestimmung angepasst werden können. Schließlich ergab sich nach der Aufnahme von über 30 Stunden Sensordaten-Material die Erkenntnis, dass das gesamte Thema weit umfangreicher ist und die möglichen Modi genauere Untersuchungen verdienen. Gerade deswegen soll dies nach der ursprünglichen Aufgabe der Aufnahme von Messwerten für die Modellbildung nachträglich mit einigen Abschnitten über das Verhalten der Beacons gewürdigt werden.

### 1.4.1 Messung der BLE-Signalausbreitung

Für die Versuche wurde zunächst ein großer Raum benötigt, um auch für reale Anwendungen die nötigen Erkenntnisse zu gewinnen. Anschließend bestand das Equipment für das Experiment aus dem Youbot, drei Beacons, dem Moto G und einem Laptop mit installiertem Ubuntu Betriebssystem. In den Bildern ... im Anhang sind einige Fotos mit dem Aufbau hinterlegt. Die Messungen wurden dabei für jeden der drei Beacons für eine Distanz für jeweils zehn Minuten durchgeführt. Aufgrund der Masse an Daten werden hier nur drei Messreihen exemplarisch in den Abbildungen ... bis ... gezeigt. Es fällt dabei auf, dass die Abweichungen bzw. das Rauschen mit größerer Distanz zunimmt. Für die ersten fünf Distanz-Meter ist die über alle Messreihen die Regel, danach wird es sehr unbestimmt und nimmt sogar ab dem zehnten Meter bis zum fünfzehnten zwischenzeitlich ab. Diese können die „Verschmutzung“ des Raumes durch andere Signale im 2,4 GHz Band sein oder auch durch Reflexionen der Wände, der Decke oder des Fußbodens hervorgerufen werden. Aufgrund der schon erwähnten Asynchronität der Beacon-Signale, der nicht einsehbaren Messzyklendauer der Estimote SDK und/oder auch der parallelen Nutzung von Bluetooth- und WLAN-Modul des Smartphones, könnten die stärksten Signale von den Beacons manchmal nicht aufgenommen und nur deren „Echo“ wahrgenommen werden, weil ihnen zu dem Zeitpunkt, als die Signale die Antennen vom Smartphone passierten, niemand zuhörte. Dieser Verdacht sieht sich jedoch nicht bestätigt, wenn nur Abbildung ... betrachtet wird. Hier ist der Abstand zwischen Smartphone und Beacon sehr gering, nichtsdestotrotz werden keine reflektierten Signale aus großerer Entfernung aufgezeichnet, weder in dieser Messung noch in weiteren. Also muss die Verstärkung des Rauschens auf die physikalischen, statt den programmiertechnischen Erklärungen zurückzuführen sein. In dieser ersten Betrachtung des Verhaltens des Systems kann schon von vornherein vermutet werden, dass das Rauschverhalten bzw. das Auftreten von Störungen raumabhängig und somit nur schwer vorhersehbar ist. Bezogen auf den Einsatz einen späteren Modells für andersartige Situationen, lässt zunächst der

Versuch einer Modellbildung anzweifeln. Jedoch zeigen gerade die genannten Probleme die Notwendigkeit eines Modells, da es als Mensch unmöglich ist alle genannten Eventualitäten in Betracht zu ziehen. Um den Anfang in einer Theoreifindung zu machen, müssen die gesammelten Daten genauer betrachtet werden und die Ausbreitung des BLE-Signals der Beacons ohne Störungen herausgefiltert werden. In Abbildung ... sind einmal alle Messreihen in einem Diagramm zusammengefasst. Dabei wurde von jeder Messung eines jeden Beacons der Mittelwert der empfangenen Empfangsstärke über der Distanz zwischen Smartphone und Leuchtfeuer aufgetragen. Auf den ersten Blick fällt auf, dass lediglich bis fünf Meter dadurch eine genaue Aussage über die Entfernung getroffen werden kann. Die Unterschiede bzw. das Fehlen eines Unterschiedes macht es für größere Entfernungen schwer, anhand dessen eine qualitative Berechnung durchzuführen. Deswegen wird der Anteil an Informationen in Abbildung ... erhöht, indem nicht der Mittelwert aller Messungen, sondern die Verteilung der empfangenen Signalstärke farblich über die Entfernungen markiert wurde. Hier erscheint verstärkt das Phänomen, dass das Rauschen nichts mit der Distanz korreliert, sondern ortabhängig ist. Zumaldest erhöht sich mit einem größeren Luftweg die Wahrscheinlichkeit, dass dies auftritt. Werden statt aller Messwerte einer Entfernung nur die 100 stärksten zur Mittelwertbildung herangezogen, wie es in Abbildung ... gezeigt wird, sieht die Entwicklung der Abnahme der Signalstärke deutlich vorhersehbarer aus und wird dadurch fassbar.

(3 Bilder - 1, 5 und 10 Meter mit Mittelwert) (Bild Mittelwert aller Messungen über Distanz) (Bild der Verteilung) (Bild Mittelwert 100 bester Messungen über Distanz)

In dem Abschnitt über ROSjava wurde in Bild 1.8 eine von der SDK berechneten Distanz erwähnt. Dies ist jedoch nur eingeschränkt nutzbar, wie es Abbildung ... zeigt. Hier wurden die tatsächlichen Distanzen mit den geschätzten der SDK gegenübergestellt und das Ergebnis verdeutlicht noch einmal die Notwendigkeit einer eigenen Theorie, um die Qualität der Berechnung zu erhöhen.

(Bild reale Distanz über berechneter)

### **Einfluss der Intervalllänge**

### **Einfluss der Signalstärke**

#### **1.4.2 Auswirkung der Smartphone-Orientierung**

#### **1.4.3 Einfluss anderer Signalquellen**

#### **1.4.4 Energieverbrauch eines Beacon**