BEUTH HOCHSCHULE
FÜR TECHNIK
BERLIN
University of Applied Sciences

3D-Indoor-Positionierung mit Low-cost-Messsensoren und Einplatinen-Computern

Beuth Hochschule für Technik Berlin
Fachbereich III - Geoinformationswesen

Masterarbeit

im Studiengang Geoinformation

vorgelegt von

Cornel Remer
Hans-Otto-Straße 3, 10407 Berlin
Matrikel- Nr. 863945

zur Erlangung des akademischen Grades

Master of Science
- M.Sc. -

Gutachter:

Prof. Dr. Stempfhuber
Beuth Hochschule für Technik Berlin

Prof. Dr. Korth
Beuth Hochschule für Technik Berlin

Berlin, 14. November 2018

Zusammenfassung

Die Entwicklung eines 3D-*Indoor*-Positionierungssystems mit *Low-cost*-Messsensoren und einem Einplatinen-Computer ist das Ziel dieser Masterarbeit. Für die Entwicklung wurden bereits existierende Methode zur *Indoor*-Positionierung sowie deren Vor- und Nachteile vorgestellt und diskutiert. Aus einem Beschleunigungssensor, einem Gyroskop und einem Magnetometer wurde durch eine Sensorfusion mit dem Kalman-Filter eine IMU (*Inertial Measurement Unit*) entwickelt. Eine weitere Sensorfusion aus den Werten der IMU und eines Barometers ergaben die Höhe des Systems. Für die Positionsbestimmung in der xy-Ebene wurde die Trilateration aus *Bluetooth*-Signalstärken, welche von *Bluetooth Low Energy Beacon* ausgehen, verwendet. Zur Untersuchung des Zusammenhangs zwischen der empfangenen Signalstärke und der Entfernung zur Signalquelle wurde eine polynomiale und eine logarithmische Regression durchgeführt. Für die endgültige Position wurden die Ergebnisse der Trilateration mit den Messungen der IMU auf einem Raspberry Pi mit Hilfe des Kalman-Filters fusioniert. Die gesamte Umsetzung in Programmiercode wurde dabei detailliert festgehalten.

In einer Testanwendung wurde das System auf einem autonom fahrenden *Arduino Robot* montiert. Die Tests in einem möblierten Zimmer ergaben einen Messfehler zwischen 10 *cm* und 150 *cm*. Für die Überprüfung der Höhe wurde das System nacheinander auf mehrere Ebenen mit unterschiedlichen Höhen gebracht. Der sich ergebende Messfehler liegt bei etwa 10 *cm*.

Die gemessenen Höhenwerte sind stabil, wenn sich der Luftdruck im Raum nicht verändert. Die Position in der xy-Ebene wird durch *Multipath*-Effekte beeinflusst, weshalb der Messfehler stark variiert. Da der Einfluss der Umgebung und der Position der *Beacon* nicht untersucht wurde, lassen sich die Ergebnisse nicht ohne weitere Versuche auf andere *Indoor*-Bereiche übertragen. Um die Genauigkeit des entwickelten *Indoor*-Positionierungssystem zu steigern, ist es erforderlich, den Einfluss und die Entstehung der *Multipath*-Effekte genauer zu untersuchen und weitere Messeinheiten zu verwenden.

Abstract

The aim of this master thesis is to develop a 3D indoor positioning system with low cost measurement sensors and a single-board computer. Therefore, existing methods for indoor positioning were presented and their advantages as well as disadvantages were discussed. The IMU (Inertial Measurement Unit) was developed with an accelerometer, a gyroscope, a magnetometer and the help of a sensor fusion with the Kalman-filter. Another sensor fusion of the IMU-values and a barometer yielded the height of the system. To determine the position in the xy-plane, trilateration from *Bluetooth* signal strengths, which were emitted from *Bluetooth Low Energy Beacon*, were used. Using polynomial and logarithmic regressions, the relationship between received signal strengths and the distance to the signal source were examined. For the final position, the results of the trilateration and the IMU-values were fused with the Kalman-filter on a Raspberry Pi. The used programming code for the entire implementation is recorded in detail.

The developed system was installed on an autonomously driving *Arduino Robot* to test it. The test, which was conducted in a furnished room, showed a measurement error between 10 cm and 150 cm. Afterwards, the system was successively carried on several planes with differing heights to examine the height. The resulting measurement error is 10 cm.

The measured value of the heights are stable as long as the air pressure remains constant in the room. *Multipath* effects influence the position in the xy-plane and hence, the measurement error varies heavily. Since the influence of the environment and the position of the Beacon were not investigated, more experiments are needed to transfer the results to other indoor areas. To enhance the accuracy of the developed indoor positioning system, the influence and emergence of the *multipath* effects need to be considered more precisely and the extension with additional measuring modules is essential.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

API	-	Application Programming Interface
BLE	-	Bluetooth Low Energy
CPU	-	Central Processing Unit
EID	-	Ephemeral Identifier
FMCW	-	Frequency-Modulated Continuous Wave
GNSS	-	Global Navigation Setellite System
GPIO	-	General Purpose Input/Output
GPS	-	Global Positioning System
HTTP	-	Hypertext Transfer Protocol
I2C	-	Inter-Integrated Circuit
IDE	-	Integrated Development Environmen
IEEE	-	Institute of Electrical and Electronics Engineers
IMU	-	Inertial Measurement Unit
IoT	-	Internet of Things
IPS	-	Indoor Positioning System
ISM	-	Industrial, Scientific and Medical
JSON	-	JavaScript Object Notation
LSB	-	Least Significant Bit
MAC	-	Media Access Control
MEMS	-	Microelectromechanical Systems
PWM	-	Pulse-Width Modulation
REST	-	Representational State Transfer
RF	-	Radio Frequency
RFID	-	Radio Frequency Identification
RSSI	-	Received Signal Strength Indicator
SCL	-	Serial Clock
SDA	-	Serial Data
TDOA	-	Time Difference Of Arrival
TLM	-	Telemetry
UDI	-	Unique Device Identification
URL	-	Uniform Resource Locator
UWB	-	Ultra-Wideband
Wifi	-	Synonym für WLAN
WLAN	-	Wireless Local Area Network

1 Einleitung

„Alexa, schalte das Licht ein.“ Diesen Satz hat vermutlich jeder bereits gehört, sei es in der Werbung oder im eigenen Zuhause. Ebenfalls ist es möglich zu sagen: „Alexa, schalte das Licht im Flur ein.“ (Sprachassistenten News, o.J.) Unter der Voraussetzung, dass der Nutzer die entsprechende Hardware besitzt, wird Alexa, Amazons virtueller Assistent, das Licht im Flur einschalten. Es wäre jedoch noch praktischer sagen zu können: „Alexa, schalte das Licht bei mir ein.“ Dazu ist es jedoch notwendig, die Position des Anwenders im Gebäude zu kennen.

Durch das *Internet of Things* (IoT) sind die verschiedensten Geräte und Maschinen, wie beispielsweise Alexa, in der Lage miteinander zu kommunizieren. Der aktuelle Trend verbindet IoT mit *Indoor-Positionierungssystemen* (IPS) und unterstützt so die Entwicklung kleinerer Geräte mit einer größeren Vielfalt an Services und besserer Energieeffizienz. Diese Verbindung ist jedoch nicht neu. Die *Radio Frequency Identification* (RFID)-Technologie wird bereits seit einigen Jahren im *Supply Chain Management* und in der Logistik eingesetzt, um beispielsweise Ware auf ihrem Produktionsweg zu verfolgen.(Gaudlitz, 2016).

Die Nachfrage nach neuen Technologien steigt nicht nur in der Industrie, sondern auch im Verbraucherbereich. So können Autos, Drohnen oder Roboter bereits autonom fahren beziehungsweise fliegen und sind über das IoT miteinander, mit anderen Objekten und mit ihrer Umgebung verbunden. Sie kommunizieren miteinander und können eigenständig Entscheidungen treffen und Aktionen ausführen (ITWissen, 2018). Für viele Aufgaben ist es notwendig die Position des Systems zu kennen. Die GNSS (*Global Navigation Satellite System*) Positionierung ist das Standardsystem im *Outdoor*-Bereich. Für den *Indoor*-Bereich müssen Alternativsysteme zur Positionierung verwendet werden, da in Gebäuden die benötigte Sichtverbindung zu den Satelliten nicht besteht. Für verschiedenste Anwendungsfälle haben sich spezielle IPS entwickelt, wobei einige Technologien und Methoden stärker vertreten sind als andere. Da sich autonome Fahrzeuge, Drohnen und Roboter nicht ausschließlich draußen bewegen, sind sie auch auf ein zuverlässiges 3D-Positionierungssystem für den Innenbereich angewiesen. Ein autonom fahrendes Auto soll zum Beispiel in einem mehrstöckigen Parkhaus zu einem freien Parkplatz fahren und Drohnen sowie Roboter müssen in Lagerhallen oder Einkaufszentren in der Lage sein, ihre Position zu bestimmen um bestimmte Aktionen auszuführen.

Die 3D-Positionierung in Gebäuden ist Gegenstand dieser Masterarbeit. Existierende IPS sind meist speziell für einen Anwendungsfall nutzbar und kostenintensiv. Ziel dieser Arbeit ist es ein System zu entwickeln, mit welchem diverse autonome Systeme in der Lage sind, sich im *Indoor*-Bereich zu lokalisieren. Mit dem Raspberry Pi als Einplatinen-Computer in

Verbindung mit einem Arduino Uno als Mikrocontroller und geeigneter Messtechnik entsteht ein 3D-*Low-cost-Indoor*-Positionierungssystem. Mit diesem System ist ein Objekt in der Lage, sich mit kostengünstigen Komponenten und geringem Energieaufwand zu lokalisieren. Viele verschiedene Einsatzgebiete mit unterschiedlichen Infrastrukturen fordern eine hohe Flexibilität und Erweiterbarkeit, welche das entwickelte System erfüllt. Ein mögliches Anwendungsszenario stellen autonome Fahrzeuge dar. Diese verfügen häufig über mehrere Kamerasysteme und Messtechnik, welche zur Erkennung von Hindernisse und Gefahren eingesetzt werden. Die unmittelbare Umgebung des Fahrzeugs kann auf diese Weise präzise erkannt werden. Dadurch ist es dem System möglich, frühzeitig auf eine Situation zu reagieren. Es ist somit nicht das Ziel ein System zu entwickeln, welches die Position eines Objekts mit den Standards industrieller oder vermessungstechnischer Genauigkeit bestimmt. Stattdessen werden vorhandene Signalquellen der Infrastruktur genutzt und mit Messungen von Sensoren kombiniert, um eine möglichst genaue Position zu erhalten. Es wird eine inertiale Messeinheit entwickelt, welche die Bewegungen und die Lage des Systems im Raum, sowie dessen Orientierung misst. Um die Genauigkeit dieser Messeinheit zu optimieren, werden eigenen Entwicklungen mit bestehenden Bibliotheken kombiniert.

Um ein System mit den genannten Anforderungen zu entwickeln und eine Übersicht über die aktuelle Entwicklung zu geben, werden im direkt folgenden Teil der Arbeit bereits existierende Systeme zur *Indoor*-Positionierung vorgestellt. Diesbezüglich wird der Aufbau und die Funktion der Systeme, sowie durchgeführte Experimente und deren Ergebnisse erläutert.

Im anschließenden Abschnitt werden die Hauptkomponenten des hier entwickelten Systems beschrieben. Dazu gehören die technischen Eigenschaften, die Funktion und der Aufbau der MEMS (*Micro Electro Mechanical Systems*)-Bauteile, sowie des Arduino Uno und des Raspberry Pi. Ebenfalls werden die *Eddystone Bluetooth Low Energy* (BLE) *Beacon* Protokolle von Google vorgestellt.

Die zur Entwicklung des System notwendigen mathematischen Grundlagen und Methoden werden im darauf folgenden Abschnitt erläutert. Dazu gehören unter anderem die Bestimmung einer Position durch Trilateration sowie die Filterung und Fusion von Sensordaten mit Hilfe des Kalman-Filters.

Nach den Grundlagen und Methoden wird im dritten Abschnitt des zweiten Kapitels das IPS entwickelt. Dazu ist es notwendig, sich mit den vorhandenen Positionierungssystemen im *Indoor*-Bereich kritisch auseinander zu setzen. Es werden Vor- und Nachteile verschiedener Technologien genannt und bewertet. Dadurch entsteht ein System, welches die oben genannten Anforderungen erfüllt. Das System wird konzeptioniert und geplant, um die verwendeten

Technologien optimal einsetzten zu können und um Probleme und technische Schwierigkeiten bereits im Voraus zu erkennen.

Es folgt die schrittweise Umsetzung des Konzepts. Dabei werden zunächst Sensoren und fertige Bibliotheken getestet, um deren Einsatzmöglichkeiten und Störanfälligkeit zu untersuchen. Anschließend werden alle Daten zusammengeführt und die Position des Systems berechnet. Die Überführung von mathematischen und technischen Methoden zur Positionsbestimmung und die Aufarbeitung von Sensordaten in Programmcode wird genau dokumentiert. Für die Programmierung der Sensoren wird die vereinfachte Form der Programmiersprache C von Arduino verwendet. Die Zusammenführung und Berechnung der Position wird in Python entwickelt.

Um das System nach seiner Fertigstellung zu testen, wird eine Testanwendung auf einem Bewegungssimulator entwickelt. Ein *Arduino Robot*, ausgestattet mit dem entwickelten Positionierungssystem, wird in einem Test geortet. Der Roboter folgt einer zuvor eingemessenen Route, wodurch eine anschließende Bewertung des Systems möglich ist.

Anschließend werden alle Ergebnisse der Entwicklung und des Tests des fertigen Systems aufgeführt. Diese werden im anschließenden Fazit, nach einer Zusammenfassung der Arbeit, diskutiert und bewertet.

2 Theoretische Grundlagen

In den folgenden Abschnitten werden die theoretischen Grundlagen erläutert, die zur Entwicklung des *Indoor*-Positionierungssystems erforderlich sind. Zunächst werden verschiedene Methoden zur Positionierung im *Indoor*-Bereich vorgestellt. Anschließend wird die Funktion und der Aufbau aller Systemkomponenten beschrieben. Im letzten Abschnitt dieses Kapitels werden die wichtigsten mathematische Grundlagen erläutert, die für die Umsetzung des Systems verwendet werden.

2.1 Methoden zur Positionsbestimmung im Indoor-Bereich

Bei der Navigation und Positionsbestimmung in Gebäuden besteht keine Sichtverbindung zu Satelliten, weshalb das GNSS nicht oder nur sehr eingeschränkt verwendet werden kann. (Kaergaard et al., 2010) Zur Lösung dieses Problems wurden bereits zahlreiche Techniken und Methoden entwickelt. Wesentliche Methoden und Systeme sind unter anderem Infrarot, Ultraschall, Radiosignale und visuelle Positionierungssysteme. Dabei weißt jede Technologie Stärken und Schwächen auf. (Retscher & Kistenich, 2006) In diesem Kapitel werden Methoden und wissenschaftliche Publikationen zur Positionierung mit Signalstärken, Laufzeitberechnungen und der inertialen Navigation vorgestellt.

2.1.1 Signalstärken

Positionierungsverfahren, welche zur Positionierung Signalstärken verwenden, beinhalten die Lateration (2.1) mit Distanzen abgeleitet aus RSSI-Messungen (*Received Signal Strength Indication*). Ziel ist es, die Position eines *Clients* über die Distanz zu bekannten Ankerpunkten zu bestimmen. Diese Ankerpunkte sind Signalquellen wie zum Beispiel Wlan (*Wire Local Area Network*)-*Router* oder Bluetooth *Beacon*.

$$r_i = \sqrt{(x_i - x_c)^2 + (y_i - y_c)^2}, \tag{2.1}$$

wobei (x_c, y_c) die Position des *Clients* ist, r_i die gemessene Entfernung und (x_i, y_i) für $i \in \{1, 2, 3\}$ die Koordinaten der Bezugspunkte sind (Er Rida, Liu, Jadi, Ali Abdullah Algawhari & Askourih, 2015).

Ein Wlan-*Router* erzeugt ein drahtloses, lokales Funknetz, das als Wlan bezeichnet wird. Diese Verbindung von Computern ist in vielen Gebäuden bereits installiert, wodurch ein Nachrüsten nicht oder nur in geringem Umfang notwendig ist. Wlan arbeitet mit Radiosignalen, welche zu den elektromagnetischen Wellen gehören und sich mit Lichtgeschwindigkeit ausbreiten. Es basiert auf den IEEE (*Institute of Electrical and Electronic Engineers*) 802.11 Standards und nutzt das ISM (*Industrial, Scientific and Medical*)-Frequenzband zwischen 2,4 und 2,48 GHz.

Unter idealen Bedingungen findet die Ausbreitung und Fortpflanzung der Radiowellen kugelförmig um die Signalquelle statt. Vor allem in Gebäuden kommt es durch Wände, Möbel und Personen zur Mehrwegsausbreitung (*Multipath*). Dies beeinträchtigt das Wlan-Signal und kann zu größeren Schwankungen der Signalstärke führen (Retscher & Moser, 2007).

Um aus den RSSI-Werten Distanzen zu bestimmen, muss zunächst der Zusammenhang aus den empfangen RSSI-Werten und der Distanz abgeleitet werden. Für die Modellierung der Signalstärke in Abhängigkeit der Distanz zwischen Sender und Empfänger existieren mehrere Modelle. Entsprechend der Umgebung ist es unter Umständen notwendig, ein Modell anzupassen, um Störfaktoren zu beseitigen. Großen Einfluss auf die Funkwellenausbreitung hat das sogenannte *Fading* (Schwund). Durch Interferenz, Abschattung, Mehrwegsausbreitung und Doppler-Effekt kommt es zu Schwankungen der Empfangenen Signalstärken.

Als einfaches empirisches Modell dient das *One-Slope-Modell*, welches auf dem Prinzip der Freiraumdämpfung aufbaut. Dieses Modell beschreibt den Zusammenhang zwischen Entfernung und Dämpfung der Signalstärke logarithmisch:

$$P(d) = P_0 + 10\gamma \cdot log_{10}(d), \tag{2.2}$$

wobei P die empfangene Signalstärke, P_0 die Referenzsignalstärke bei $1\ m$ Entfernung, γ der Dämpfungsfaktor und d die Distanz zwischen Signalquelle und Empfänger darstellt.

Ein weiteres Modell ist das *Multi-Wall-Modell*, welches als Ausbreitungsmodell für den *Indoor*-Bereich entwickelt wurde. Im Gegensatz zum *One-Slope-Modell* (2.2) wird in dieser Modellierung eine Summe aus einzelnen Signaldämpfungen von Wänden hinzugezogen:

$$P(d) = P_0 + 10\gamma \cdot log_{10}(d) + \sum_{i=1}^{n} D_i, \tag{2.3}$$

wobei D_i den Dämpfungswert der i-ten Wand darstellt.

In Gebäuden mit vielen Wänden aus verschiedenen Materialien, Einrichtungsgegenständen, schmalen oder verwinkelten Räumen geht der Zusammenhang zwischen Intensität und Distanz zunehmend verloren. Für diese Fälle existieren zwei weitere Methoden zur Wiederherstellung dieses Zusammenhangs.

Zum einen die *Fingerprint*-Methode, bei der in einem Gebäude Signalstärkenmessungen durchgeführt werden. Anschließend werden die Messergebnisse als Histogramme mit den dazugehörigen Koordinaten in einer Datenbank abgelegt. Bei der Positionsbestimmung eines *Clients* wird die gemessene Signalstärke mit der Datenbank verglichen und somit die wahrscheinlichste Position ausgegeben (Retscher & Tatschl,2017). Das Erstellen der Datenbank ist sehr arbeitsaufwändig und muss für jeden neuen *Access Point* (beispielsweise Wlan *Router*) oder bei anderen Änderungen der Infrastruktur neu erstellt werden. Die Genauigkeit dieser Methode liegt je nach Anbieter in einem Bereich von ein bis drei Metern (Retscher & Moser, 2007).

Bei der zweiten Methode, der Signalmodellierungsmethode, wird ein möglichst genaues Abbild der realen Ausbreitung der Wellen erzeugt, um somit die Signalstärke am Empfängerort abzuleiten. Je genauer die Positionierung erfolgen soll, desto besser muss das Modell der Realität entsprechen. Dadurch steigt auch die Komplexität des Modells, weshalb bei der Modellierung häufig auf Möbelstücke und andere mobile Gegenstände verzichtet wird. Aus einem zumeist rasterförmigen Messgebiet, den *Access Points* und der Distanz zu den Wänden wird ein Modell erstellt, mit welchem sich Vorhersagen zu den Signalstärken von jedem Punk im Messgebiet treffen lassen. Ähnlich wie bei der *Fingerprint*-Methode werden die vorhergesagten Werte mit den dazugehörigen Punkten in einer Datenbank gespeichert, um bei der Positionierung den

Eintrag mit der besten Übereinstimmung auswählen zu können.

Ein Vorteil der Signalmodellierungsmethode besteht darin, dass die Datenbank schneller erstellt werden kann als bei der *Fingerprint*-Methode. Ebenfalls ist die Integration neuer *Access Points* einfacher. Dennoch ist diese Methode ungenauer als die *Fingerprint*-Methode (Retscher & Moser, 2007).

Neben Wlan ist es auch möglich, andere *Radio frequencies* (RF) zu verwenden, zum Beispiel BLE. Dies hat gegenüber Wlan, UWB (*Ultra Wide Band*), Ultraschall und dem *Bluetooth 3.0* den Vorteil, dass es einen geringen Energie- und Kostenaufwand aufweist (Er Rida et al., 2015).

Rida et al. der *Tongji University* stellten 2015 ein System zur *Indoor*-Positionierung mittels BLE Signalstärkenmessungen vor. In diesem System wurden *Smart Devices* mit *Andoid* 4.1 über Signalstärkenmessungen und Triangulation in einem Testgebiet geortet. Die *Access Points* oder *Nodes* sendeten in kurzen periodischen Abständen Hochfrequenzsignale aus. Dabei wurden die *Nodes* so angeordet, dass sich jeder *Client* in der Reichweite von mindestens drei von ihnen befand. Jeder *Node* hatte eine Reichweite von 15 Metern, weshalb alle sechs Meter ein solcher installiert werden musste. Zur Untersuchung des Zusammenhangs zwischen Empfangenem RF-Signal und der Entfernung wurden alle 100 Zentimeter Messungen vorgenommen und mehrmals wiederholt. Mit der sich ergebenen Exponentialfunktion war es möglich, Tests in der *Tongji University* durchzuführen, wobei sich der Positionierungsfehler auf 0 bis 2 Meter belief (Er Rida et al., 2015).

2.1.2 Laufzeitdifferenzmessung

Die Laufzeitdifferenzmessung oder *Time Difference of Arrival* (TDOA) ähnelt dem Verfahren der Laufzeitmessung. Der Unterschied besteht darin, dass bei der TDOA Methode ein Sender ein Signal aussendet, welches von zwei Basisstationen empfangen wird. Betrachtet wird nicht die absolute Laufzeit, sondern die Differenz der Laufzeiten, welche durch die Empfängerstationen gemessen wurden (Retscher & Kistenich, 2006). Dies hat den Vorteil, dass ein *Offset* (Versatz) von nicht synchronisierten Empfängeruhren durch die Subtraktion der Laufzeiten eliminiert wird. Für den Empfängern muss somit die absolute Zeit der Übertragung nicht bekannt sein. Bei zwei Empfängerstationen mit bekannter Position kann die Position des Senders auf eine Hyperbel begrenzt werden (Mautz, 2012). Da sich elektromagnetische Signale mit Lichtgeschwindigkeit bewegen, ist die zu messende Laufzeit und Laufzeitdifferenz sehr gering. Diese Methode lässt sich jedoch auch auf andere Signale, wie zum Beispiel Ultraschall, anwenden, die eine viel geringere Geschwindigkeit aufweist (Retscher & Kistenich, 2006).

Zhang, Kuhn, Merkl, Fathy und Mahfouz (2006) stellten ein System zur Positionierung in Gebäuden vor, welches die Laufzeitdifferenzmessung von UWB Funksignalen verwendet. Vor der Entwicklung ihres Systems verglichen sie das *Frequency Modulated Continuous Wave* (FMCW) Radar mit den UWB-Signalen. FMCW ist in Positionierungssystemen sehr verbreitet. So erzielten beispielsweise Stelzer, Pourvoyeur und Fischer (2004) für ein *Outdoor*-Positionierungssytem eine Genauigkeit von unter 10 cm. Im *Indoor*-Bereich wird diese Genauigkeit durch *Multipath*-Effekte gemindert. UWB-Systeme dagegen verringern die *Multipath*-Effekte und ermöglichen eine Anpassung der Frequenz, um verschiedene Materialien besser durchdringen zu können (Zhang et al., 2006).

Zhang et al. (2006) untersuchten daraufhin die Genauigkeit ihres Systems in zwei Experimenten. Zunächst wurde ein eindimensionaler Versuch durchgeführt. Ein Empfänger wurde in drei Metern Entfernung zum Sender platziert. Anschließend wurde der Empfänger um zwei Zentimeter vom Sender weg verschoben. Mit dem System wurde eine Verschiebung von $1.92\,cm$ gemessen, was ein Differenz von $0.8\,mm$ zum wirklichen Wert von $2\,cm$ darstellt.

In einem zweidimensionalen Experiment wurden drei Empfänger mit bekannten Positionen platziert. Die Position des Senders war $(1m, 0m)$ und wurde ebenfalls durch das System aus der *line of sight*-Entfernung zu jedem Empfänger und durch Triangulation ermittelt. Das Ergebnis der Positionsbestimmung war $(1.0111m, 0.0168m)$, wodurch ein Messfehler von $1.68cm$ erreicht wurde.

2.1.3 Inertiale Navigation

Die inertiale Navigation ist ein Verfahren des *Dead Reckoning* oder auf deutsch der Koppelnavigation. Dies ist eine Methode zur Positionsbestimmung, bei dem aus der bekannten Bewegungsrichtung und der Bewegungsgeschwindigkeit eines Objekts dessen Position näherungsweise bestimmt werden kann. Zu Zeiten, in denen es noch keine Navigation mit GNSS gab, wurde unter anderem dieses Verfahren verwendet, um beispielsweise Schiffe zu navigieren. Die Bewegungsrichtung (der Kurs) wurde mit dem Kompass ermittelt und die Geschwindigkeit mit einem Log. Bei konstanter Geschwindigkeit ist es möglich, die zurückgelegte Strecke über ein bekanntes Zeitintervall zu errechnen. Die neu bestimmte Position wird zur letzten bekannten hinzu addiert.

Bei der inertialen Navigation wird der Kurs und die Geschwindigkeit mit Inertialsensoren, wie zum Beispiel dem Beschleunigungsmesser und dem Drehratenmesser, ermittelt. Es werden jeweils drei Beschleunigungs- und Drehratenmesser benötigt, welche orthogonal in x,y und z-Richtung angeordnet sind. Die Sensoren sind fest mit einem Objekt, beispielsweise einem Fahrzeug, verbunden und können dadurch die Lage und die Geschwindigkeit des Objekts und deren Änderung erfassen. Ein solches System wird *Inertial Measurement Unit* (IMU) genannt. Durch die Integration der Drehraten wird die Lage des Objekts bestimmt, um anschließend die gemessenen körperfesten Beschleunigungen in ein Koordinatensystem mit raumfesten Koordinatenrichtungen umzurechnen. Die Position wird nun durch Integration der Geschwindigkeit bestimmt. Durch die Integration der mit Fehlern behafteten Drehraten und Geschwindigkeiten wächst auch der Navigationsfehler mit der Zeit (Wendel, 2007).

Ksentini, Elhadi und Lasa stellten 2014 ein System zur *Indoor*-Positionierung vor („Indoor Dalil“). Das Ziel war es eine Positionierung ohne GPS (*Global Positioning System*), Wifi oder andere externe Positionierungssysteme zu ermöglichen. Des Weiteren wurde darauf geachtet, den Energieverbrauch so gering wie möglich zu halten. Als günstige, effektive und energiesparende Technik wurden Interialsensoren verwendet. Ksentini et al. evaluierten dieses System und die Genauigkeit eines Beschleunigungssensors, sowie dessen Nutzen für die Positionierung.

Die Evaluierung des Systems geschah in zwei Szenarien. Bei dem ersten Szenario wurde die Position durch die Erkennung des Bewegungsstatus bei konstanter Geschwindigkeit bestimmt. Eine Person bewegte sich mit konstanter Geschwindigkeit („Gehgeschwindigkeit“) in eine bekannte Richtung. Mit dem Beschleunigungssensor wurde festgestellt, ob sich das System in „Bewegung“ oder in „Ruhe“ befand. Wird das System über eine gewisse Zeit mit bekannter Geschwindigkeit und Richtung bewegt, kann dessen Position bestimmt werden. Kleinere Bewegungen der Hand verursachen den Status „Bewegung“, obwohl das System in „Ruhe“ ist, wodurch eine Kalibrierung notwendig ist. Tabelle 2.1 zeigt die Ergebnisse des ersten Szenarios:

Real distance (m)	11	22	33
Estimated distance (m)	10.92	21.87	31.72
Error (m)	0.08	0.13	1.28

Tabelle 2.1: Indoor Dalil, Ergebnisse erstes Szenario
(Ksentini, Elhadi & Lasla, 2014)

Das zweite Szenario dient der Positionsbestimmung bei unbekannter Geschwindigkeit. Dafür wurden zwei Experimente durchgeführt. Im ersten Experiment wurde das System auf einer zwei Meter langen Eisenschiene bewegt, um die Erschütterungen der Hand und die des Laufens zu eliminieren. Im zweiten Experiment hielt eine Person das System in der Hand und bewegte sich in Richtung der y-Achse des Systems. Zunächst lief die Person vier Meter, stoppte dann und lief erneut vier Meter. Die Positionierung geschah währenddessen in drei Schritten. Der Beschleunigungssensor misst die Beschleunigung und die Messwerte des Sensors werden integriert, um die momentane Geschwindigkeit zu erhalten. Anschließend ergibt die Integration der Geschwindigkeit die momentane Position.

Die Messwerte des zweiten Szenarios zeigen, dass mit der Zeit die Distanz weiter anwächst, auch wenn sich das System in „Ruhe“ befindet. Grund dafür ist unter anderem Messrauschen, welches eine scheinbare Bewegung des Systems während des Stillstandes verursacht. Um das Rauschen zu verringern, wurde ein *High Pass*- und ein *Infinite Impulsive Response*(IIR)-Filter verwendet. Die Ergebnisse sind in Abbildung 2.1 als Graph dargestellt:

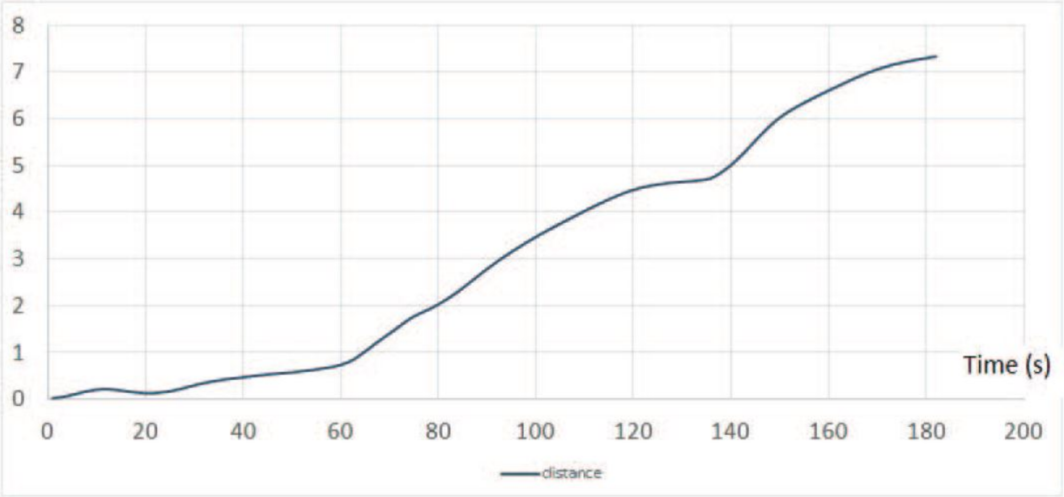


Abbildung 2.1: Indoor Dalil, Distanz des zweiten Szenarios mit Filtern (Ksentini, Elhadi & Lasla, 2014)

Nach Ksentini et al. (2014) sind die Ergebnisse des Experiments in Bezug auf Distanz und Geschwindigkeit vielversprechend und kommen der Realität sehr nahe. Deshalb konnte eine App mit akzeptabler Genauigkeit zur Positionierung mit Android entwickelt werden.

2.2 Systemkomponenten

Für die Umsetzung des Systems sind mehrere Systemkomponenten notwendig. Dazu zählen auch die MEMS, welche elektromechanische Bauteile im Mikrometerbereich sind. Die Aufgabe der MEMS ist es, physikalische Größen wie beispielsweise wirkende Kräfte oder Magnetfeldstärken zu messen. Mit Hilfe der gemessenen physikalischen Größen lassen sich Systemzustände bestimmen, zum Beispiel die Lage im Raum und die Beschleunigung.

MEMS werden meist aus Silizium hergestellt, haben einen geringen Energiebedarf, sind klein, kostengünstig und in Massen herstellbar. Aus diesem Grund sind sie unverzichtbar für moderne und innovative Elektrotechnik (Elektro Kompendium, o.J.).

Im folgenden Kapitel wird die Kommunikation zwischen MEMS und Mikrocontroller mit dem I2C-Bus erklärt. Anschließend werden Aufbau und Funktion der für dieses *Indoor*-Positionierungssystem verwendeten MEMS beschrieben. Des Weiteren wird auf die technischen Grundlagen des Einplatinencomputers (Raspberry Pi), des Mikrocontrollers

(Arduino Uno) und der BLE *Beacon* eingegangen.

2.2.1 I2C

I2C oder IIC steht für *Inter-Integrated Circuit*. Es wurde 1980 von dem Unternehmen Philips für die Kommunikation zwischen elektrischen Bauelementen entwickelt, die sich auf einer Platine befinden. Ebenso kann es für eine *Master-Slave* Kommunikation von Komponenten verwendet werden, die sich nicht auf einer Platine befinden. Dazu werden zwei Verbindungen benötigt. Die *Clock Line* oder *Serial Clock* (SCL), welche die zu übertragenden Datenbits synchronisiert, und die *Data Line* oder *Serial Data* (SDA), auf welcher Bits übertragen werden. Abbildung 2.2 zeigt ein Beispiel einer solchen Verbindung (I2C Bus, o.J.):

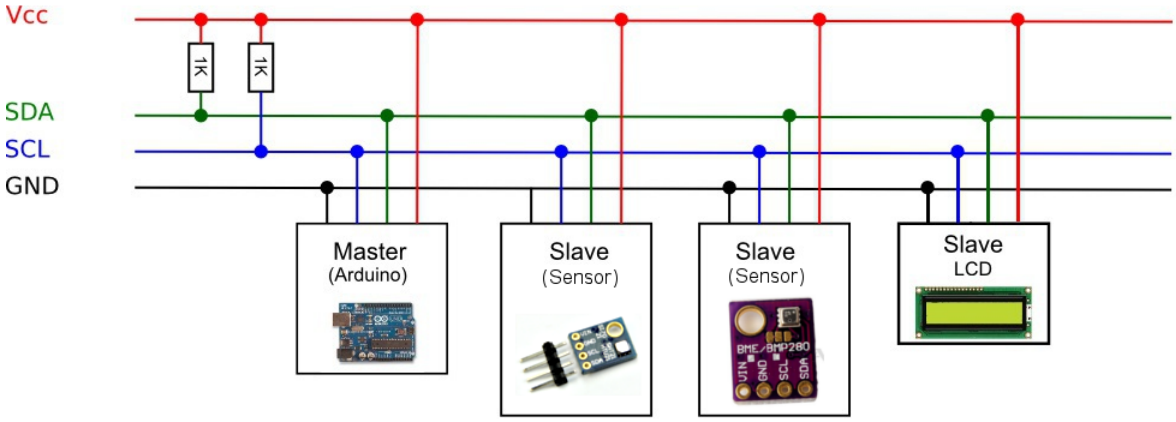


Abbildung 2.2: Beispiel einer I2C Verbindung
(Bac STI 2D, o.J.)

Im Falle einer Kommunikation wird zunächst ein Startbit übertragen. Anschließend folgt das Steuerbyte oder auch die *Device Adress*, um ein bestimmtes *Device* anzusprechen. Nun werden dem *Device* die ersten Daten geschickt, beispielsweise die Addresse eines Registers. Ein weiteres Bit folgt, welches entweder eine 1 für die Leseoperation oder eine 0 für die Schreiboperation darstellt. Dadurch wird dem *Device* mitgeteilt, ob Daten gelesen oder geschrieben werden sollen. Das *Device* bestätigt dies mit einer 0 und sendet anschließend die Daten oder lehnt es mit einer 1 ab (Buchman, o.J.). Zu sehen ist dies in Abbildung 2.3:

2.2.2 Barometer

Das Barometer ist ein Druckmessgerät, welches zu den Manometern gehört. Mit ihm wird der Luftdruck gemessen. Luftdruck bezeichnet den Druck, der durch das Gewicht der Atmosphäre auf die Erdoberfläche entsteht. Schwerpunktanwendungen für die Messung des Luftdrucks sind unteranderem die Messung des absoluten Druckwertes für meteorologische Messungen und die

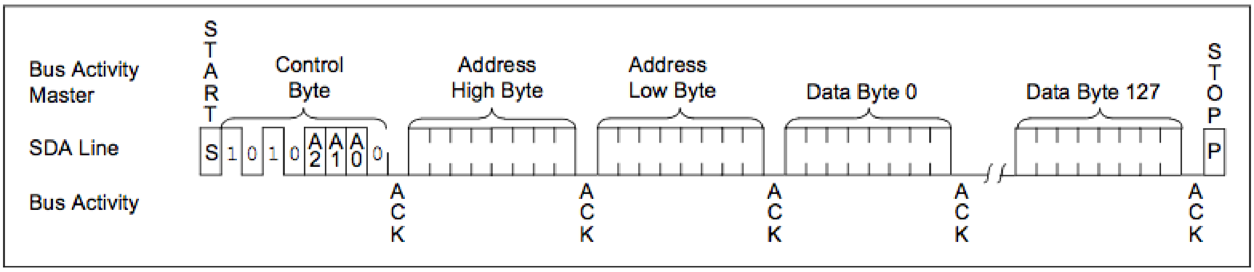


Abbildung 2.3: Write Byte einer I2C Verbindung
(TotalPhase, o.J.)

Referenzwertmessungen für Pegel- Druckmessungen.

Außerdem kann mit Hilfe des gemessenen Luftdrucks die Höhe berechnet werden. Für diese Berechnung wird in Bereichen der Troposphäre (bis zu einer Höhe von elf Kilometern) die internationale Höhenformel verwendet:

$$p = p_0 (1 - \frac{6,5 \cdot h}{288})^{5,256}, \tag{2.4}$$

wobei p der Druck in der Höhe h, p_0 der Druck auf der Höhe 0 und h die Höhe über 0 in Kilometern darstellen. Auf Höhe des Meerespiegels beträgt der durchschnittliche Luftdruck 101.325 Pa, wobei dieser Mittelwert über den Tag schwankt. Die Schwankungen folgen der Temperatur, da diese Einfluss auf die Dichte der Luft hat. Im mitteleuropäischen Raum können diese Schwankungen bis zu 100 Pa betragen. Größere Auswirkungen haben meteorologische Einflüsse wie Hoch- und Tiefdruckgebiete, welche Schwankungen von bis zu 10.000 Pa verursachen können.

Bewegt man sich vom Meeresspiegel senkrecht nach oben, verringert sich die verbleibende Luftsäule, wodruch der Druck abnimmt. Diese Abnahme des Drucks geschieht exponentiell, wobei die mit der Höhe sinkende Temperatur ebenfalls Einfluss hat. Die Temperaturabnahme wird in (2.4) durch den Wert $6,5$ Kelvin miteinbezogen. Der Jahresmittelwert der Lufttemperatur am Boden beträgt 288 Kelvin.

Für das Messen von Druck wird meinstens die Verformung einer Membran durch die Druckdifferenz zwischen Vorder- und Rückseite bestimmt. Im Falle einer absoluten Druckmessung des Luftdrucks befindet sich an der Membranrückseite ein abgeschlossenes Vakuum mit definiertem Druck (Hering & Schönfelder, 2018).

Der Sensor $BMP280$ von Adafruit, welcher für dieses Projekt zur Luftdruckmessung verwendet wird, misst den absoluten Druck durch die Referenz mit der Materialstruktur und nicht mit dem Vergleich eines anderen Drucks. Das Messverfahren beruht auf dem piezoelektrischen Effekt. Erfährt der Piezokristall eine Deformation, kommt es zu einer elektrischen Polarisation, wobei das auftretende elektrische Moment proportional zur Deformation ist. Die Messung des elektrischen Moments wird genutzt, um auf den Druck zu schließen (Spektrum, o.J.).

2.2.3 Beschleunigungssensor

Zur Positionsbestimmung, aber auch zur Stabilisierung des Gyroskops ist ein Beschleunigungsmesser (Akzelerometer) unabdingbar. Um die Beschleunigung, also die Änderung der Geschwindigkeit, eines Objekts zu bestimmen, wird das zweite Newtonsche Gesetz verwendet (2.5). Jeder Körper der Masse m, welcher einer Kraft $\overrightarrow{F}$ ausgesetzt wird, erfährt eine Beschleunigung $\overrightarrow{a}$. Dadurch werden positive und negative Beschleunigungen bestimmt.

$$\overrightarrow{F} = m \cdot \overrightarrow{a} \tag{2.5}$$

Eine einfache Möglichkeit eine Beschleunigung durch eine Weg-Zeit-Messung zu bestimmen, is eine federnd aufgehängte Masse, die durch eine auf sie wirkende Kraft beschleunigt und ausgelenkt wird. Diese Auslenkung kann nun durch kapazitive Systeme erfasst werden (Hering & Schönfelder, 2018). Das Prinzip eines solchen kapazitiven Systems ist in Abbildung 2.4 dargestellt. Ein Kondensatorpaar besteht aus zwei festen Elektroden, die symmetrisch zur Bewegungsrichtung der Masse ausgerichtet sind. Die bewegliche Masse (seismische Masse) dient als dritte Elektrode und befindet sich zwischen dem Kondensatorpaar. Wird die seismische Masse durch eine einwirkende Kraft aus ihrer Ruhelage bewegt, ändert sich die Kapazität des Kondensators. Die Änderung der Kapazität ist proportional zur Auslenkung der seismischen Masse und ein Maß für die Beschleunigung (Gevatter, 2006).

Für einen Beschleunigungssensor im dreidimensionalen Raum werden drei dieser kapazitiven Systeme jeweils so angeordnet, dass eines immer senkrecht zu den anderen beiden steht. In dem hier entwickeltem Positionierungssystem wird das *LSM303 DLHC* Modul verwendet. Dieses beinhaltet neben dem dreidimensionalen Akzelerometer auch ein Magnetometer, auf welches im Kapitel 2.2.5 näher eingegangen wird.

Mit dem *LSM303 DLHC* werden lineare Beschleunigungen für verschiedene Sensitivitäten gemessen, siehe Tabelle 2.2. Der serielle I2C-Bus überträgt die gemessenen Werte mit einer Rate von $1, 10, 25, 50, 100, 200$ oder 400 Hz. Das Modul ist somit in der Lage, geringe Beschleunigungen zu messen und diese mit einer hohen Updaterate an den Mikrocontroller

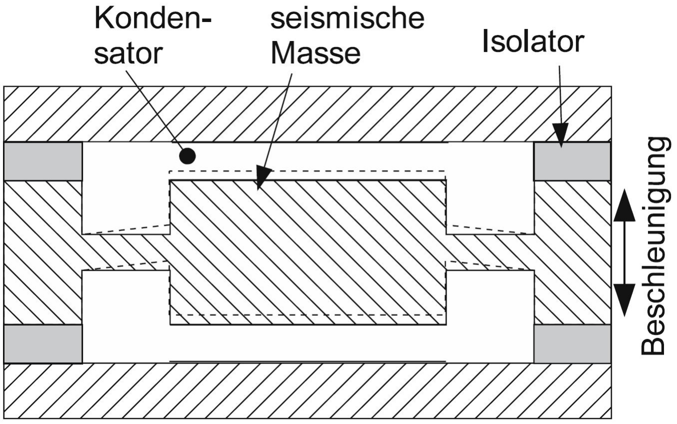


Abbildung 2.4: Prizip eines kapazitiven Beschleunigungssensor
(Gevatter, 2006)

weiterzugeben. Um aus den übermittelten Werten die Beschleunigungen a_i in m/s^2 zu erhalten, wird der Output $rawOutAcc_i$ mit dem Umrechnungsfaktor $mg/LSB/1000$ (siehe Tabelle 2.2) und der Konstanten der Erdbeschleunigung $g = 9.81\,m/s^2$ für alle Raumrichtungen x,y und z multipiziert (STlife.augmented, 2013b).

$$a_i = rawOutAcc_i \cdot mg/LSB \cdot 0.001 \cdot g \qquad wobei\ i \in \{x, y, z\} \tag{2.6}$$

Acc range [g]	**Factor** [mg/LSB]
±2.0	1
±4.0	2
±8.0	4
±16.0	12

LSB = *least significant bit*

Tabelle 2.2: Mögliche Sensitivitäten des *LSM303 DLHC* mit Umrechnungsfaktoren
(Nach STlife.augmented, 2013b)

2.2.4 Gyroskop

Ein Gyroskop oder auch Kreiselinstrument besteht aus beweglich gelagerten Rädern, die sich in einem stabilen Rahmen bewegen. Befinden sich die Räder in Rotation, behält das Instrument seine Ausrichtung im Raum, auch wenn die Aufhängung um die Räder rotiert. Neben dem rotatorischen Gyroskop existieren in der Mikrotechnik auch schwingende Gyro-Chips (Hesse & Schnell, 2014).

Um mit einem mikromechanischen Sensor dessen Lage im Raum zu bestimmen, wird im Falle des schwingenden Gyro-Chips die Corioliskraft $\overrightarrow{F_C}$ als Messeffekt verwendet. Diese Kraft tritt auf, wenn sich in einem rotierenden Bezugssystem eine Masse m mit der Radialgeschwindigkeit $\overrightarrow{v_R}$ relativ zu diesem Bezugssystem bewegt. Bestimmen lässt sich diese Trägheitskraft wie folgt (Wefel & Rost, 2016):

$$\overrightarrow{F_C} = -2 \cdot m(\overrightarrow{v_R} \times \overrightarrow{\omega})$$

Ein solcher Gyro-Chip oder Drehratensensor nutzt Schwingungen, um die Winkelgeschwindigkeit zu bestimmen. Dazu wird ein Aktor mit sich periodisch ändernder Geschwindigkeit angeregt und in Schwingung gebracht (Primärschwingung), siehe Abbildung 2.5. Die durch die Corioliskraft verursachte Sekundärbewegung wird durch einen Sensor detektiert. Die Sekundärbewegung wird mittels mikromechanischer, oszillierender Biegebalken in eine horizontale Sekundärbewegung transformiert und kann nun, wie bei einem Beschleunigungssensor in Kapitel 2.2.3 beschrieben, kapazitiv ausgewertet werden (Mescheder, 2000).

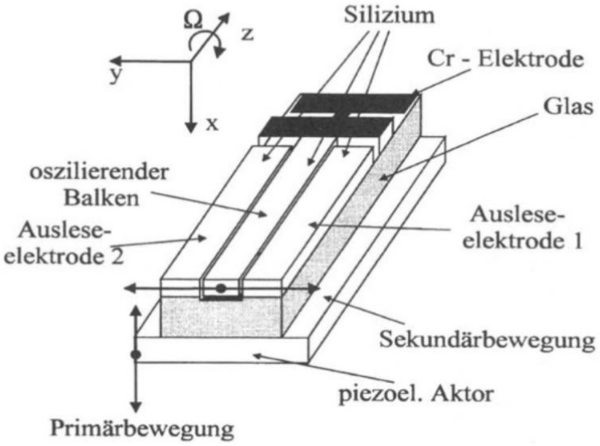


Abbildung 2.5: Schema eines mikrotechnischen Drehratensensors
(Mescheder, 2000)

Für dieses Projekt wir das $L3GD20$ Modul verwendet, welches die Winkelgeschwindigkeit in x, y und z-Richtung erfassen kann. Die mechanischen Charakteristika des Sensors sind in Tabelle 2.3 aufgeführt.

Das Modul überträgt die Messwerte diskret mittels I2C-Bus mit $95, 190, 380$ oder $760 Hz$. Um aus den gemessenen Winkelgeschwindigkeiten $\overrightarrow{\omega_i}$ die Lage des Sensors im Raum zu ermitteln und durch die Eulerwinkel (*pitch*, *roll*, *yaw*) zu beschreiben, werden diese zeitdiskret beziehungsweise numerisch über dem Intervall $[0; n]$ integriert (2.8). Dazu werden zunächst die vom Sensor übermittelten Bytes $rawOutGyro_i$ mit dem Umrechnungsfaktor $mDPS/digit \cdot 0.001$ (siehe

Tabelle 2.3) multipliziert und durch die eingestellte Updaterate ν dividiert (2.7). So ergibt sich die Winkelgeschwindigkeit $\overrightarrow{\omega_i}$ in Grad pro Sekunde (*DPS, degree per second*) (STlife.augmented, 2013a).

$$\overrightarrow{\omega_i} = \frac{rawOutGyro_i \cdot 0.001 \cdot mDPS/digit}{\nu} \tag{2.7}$$

$$\alpha_i = \int_0^n \overrightarrow{\omega_i}(t)dt$$

$$\alpha_i = \sum_{j=0}^{n} \overrightarrow{\omega_i} \tag{2.8}$$

wobei $n :=$ Anzahl der Messungen, $i \in \{pitch,\ roll,\ yaw\}$.

Sensitivity [DPS]	**Factor** [mDPS/digit]
±250.0	8.75
±500.0	17.5
±2000.0	70.0

Tabelle 2.3: Mögliche Sensitivitäten des $L3GD20$ mit Umrechnungsfaktoren (Nach STlife.augmented, 2013a)

2.2.5 Magnetometer

Zu den Magnetometern gehören viele verschiedene Sensortypen, welche alle verschiedene physikalische Effekte nutzen. Das im Kapitel 2.2.3 verwendete *LSM303 DLHC* Modul enthält neben einem Akzelerometer auch ein Magnetometer, mit welchem die magnetische Flussdichte ermittelt wird. Daraus lässt sich ein digitaler Kompass entwickeln, mit welchem die Richtung des magnetischen Nordpols der Erde und bei bekannter Deklination auch die Richtung des geografischen Nordpols bestimmt wird.

Das Modul nutzt dafür den Hall-Effekt. Dieser gehört zu den galvanomagnetischen Effekten, bei denen sich ein stromdurchflossener Leiter oder Halbleiter in einem Magnetfeld befindet, welches senkrecht zur Fließrichtung des Stroms steht. Fließt in dem Leiter ein Strom I_x senkrecht zum Magnetfeld B_z werden die Ladungsträger durch die Lorentzkraft F_L in y-Richtung verschoben (Abbildung 2.6).

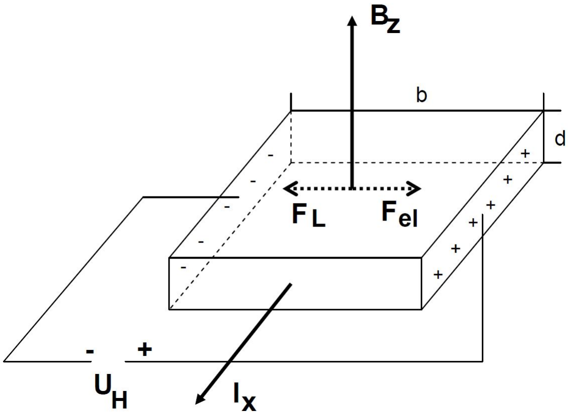


Abbildung 2.6: Prinzip des Hall-Effekts
(Hering & Schönfelder, 2018)

Durch die Verschiebung der Ladungsträger entsteht ein Elektronenüberschuss (siehe Abbildung 2.6 links, - Ladung) und ein Elektronenmangel (siehe Abbildung 2.6 rechts, + Ladung), welche wiederum ein elektrisches Feld erzeugen, das entgegen der Lorentzkraft in Richtung F_{el} wirkt. Die Hall-Spannung U_H welche in y-Richtung entsteht wird nun gemessen. Zusammen mit den bekannten Größen d als Dicke des Leiters, I_x als Stromstärke und A_H als Hall-Koeffizient, welcher spezifisch für das Material des verwenden Leiter ist, lässt sich die magnetische Flussdichte B_z wie folgt bestimmen (Hering & Schönfelder, 2018):

$$B_z = \frac{U_H \cdot d}{A_H \cdot I_x}$$

Das verwendete *LSM303 DLHC* Modul misst die magnetische Flussdichte und schickt die gemessenen Werte jeder Richtung (x, y, z) mit dem I2C-Bus als Bytefolge an den Mikrocontroller. Die magnetische Flussdichte B_i in Gauß berechnet sich aus dem Output $rawOutMag_i$ dividiert durch den Umrechnungsfaktor $gain_i$ in x-, y- und z-Richtung, welcher je nach eingestellter Sensitivität variiert und dem Datenblatt entnommen werden kann. Eine Übersicht der möglichen Sensitivität und dazugehörende Umrechnungsfaktoren kann der Tabelle 2.4 entnommen werden.

$$B_i = \frac{rawOutMag_i}{gain_i} \qquad wobei\ i \in \{x, y, z\} \tag{2.9}$$

Field range [Gauss]	**Gain X,Y** [LSB/Gauss]	**Gain Z** [LSB/Gauss]	**Output range**
±1.3	1100	980	$[-2048; 2047]$
±1.9	855	760	$[-2048; 2047]$
±2.5	670	600	$[-2048; 2047]$
±4.0	450	400	$[-2048; 2047]$
±4.7	400	355	$[-2048; 2047]$
±5.6	330	295	$[-2048; 2047]$
±8.1	230	205	$[-2048; 2047]$

Tabelle 2.4: Mögliche Sensitivitäten des *LSM303 DLHC* mit Umrechnungsfaktoren (Nach Stlife.augmented, 2013b)

Mögliche Updateraten des Moduls sind je nach Einstellung: 0.75, 1.5, 3.0, 7.5, 15.0, 30.0, 75.0 oder 220.0 Hz (Nach Stlife.augmented, 2013b).

2.2.6 Raspberry Pi 3

Das Raspberry Pi ist ein *low cost* Computer in Kreditkartengröße, der alles kann, was auch ein normaler Desktop Computer kann. Zusätzlich ist das Raspberry in der Lage, mit der Außenwelt zu interagieren. Dazu besitzt er 40 GPIO (*General Purpose Input/Output*) Pins.

Entwickelt wurde das Raspberry Pi von der *Raspberry Pi Foundation*, eine Stiftung aus Großbritannien, welche es sich zum Ziel gesetzt hat, die Bildung von Kindern und Erwachsenen im Bereich der Computerwissenschaften zu verbessern (Raspberry Pi Foundation, o.J.b).

Für dieses Projekt wird das *Raspberry Pi 3 Model B* verwendet, welches 2016 das *Raspberry Pi 2 Model B* ersetzte. Das Gerät ist für etwa 35 Euro[1] erhältlich und verwendet einen *Quad Core* 1.2 *GHz Broadcom BCM2837* 64 *bit* Prozessor mit 1 GB RAM. Es wird mit 5 V (±5%) und 2.5 A betrieben. Weitere, für dieses Projekt wichtige, Besonderheiten des *Raspberry Pi 3 Model B* sind *wireless LAN (BCM43438)*, BLE, Ethernet und vier USB 2.0 Ports (Raspberry Pi Foundation, o.J.a).

2.2.7 Arduino Uno

Der Arduino Uno ist ein *Microcontroller Board*, dessen Hauptkomponente der ATmega328P, ein 8 *Bit Microcontroller* mit einem *Flash*-Speicher von 32 KB, ist. Er besitzt 14 digitale *In-* und *Output*-Pins, von denen sechs als PWM (Pulsweitenmodulation) Pins genutzt werden können. Zusätzlich besitzt er sechs analoge *Input* Pins, wobei die Pins $A4$ (SDA) und $A5$ (SCL) für die Kommunikation mittels I2C-Bus gedacht sind. Er läuft mit 16 mHz und bietet durch einen Einkaufspreis von 20 Euro[2] und einer vereinfachten Form der C-Programmierung einen guten

---

[1] https://www.amazon.de/Raspberry-Pi-Model-ARM-Cortex-A53-Bluetooth/dp/B01CD5VC92 [Stand 11.11.2018]

[2] https://store.arduino.cc/arduino-uno-rev3 [Stand 11.11.2018]

Start in die Programmierung und Elektrotechnik. Die Betriebsspannung liegt bei 5 V, wobei laut Hersteller die empfohlene Spannung zwischen 7 und 12 V liegt.

Programmiert wird der Arduino mit der Arduino Software IDE (*Integrated Development Environment*). Dazu muss im Menü der IDE unter *Tools* und *Board* lediglich der Arduino ausgewählt und der USB Port, über welchen der Mikrocontroller an dem PC angeschlossen ist, festgelegt werden (Arduino, o.J.).

Der folgende Code-Ausschnitt (1) zeigt das Gerüst eines Arduino-Skripts. Grob kann das Skript in drei Bereiche untergliedert werden. Oberhalb der Funktion *setup()* können Bibliotheken, beispielsweise *Wire.h*, eingebunden und Variablen festgelegt und initiiert werden. In der darauf folgenden *setup()*-Funktion wird der Code nur ein einziges Mal ausgeführt. Dieser Bereich dient häufig dazu, Verbindungen, wie beispielsweise eine serielle Verbindung zum Computer, herzustellen, Pin-Belegungen festzulegen oder um Sensoren zu initialisieren. Darauf folgt der Hauptteil des Programms, die *loop()*-Funktion, welche solange wiederholt wird, bis die Stromversorgung getrennt wird. In dieser Schleife werden alle Funktionen und Berechnungen durchgeführt, die mehrmals wiederholt werden können.

Code-Ausschnitt 1: Gerüst eines Arduino-Skripts

```
/* Eigene Programmierung, enthält Code-Snippets des Standard Arduino Frame */
#include <Wire.h> // eingebundene Bibliotheken
int a = 5; // Variablen definieren
float b, c;
void setup() {
  // hier geschriebener Code wird nur einmal ausgeführt
}
void loop() {
  // hier geschriebener Code wird permanent wiederholt
}
```

2.2.8 BLE Beacons

Beacon, zu deutsch „Signalfeuer“, sind kleine Geräte, die Radiowellen über die BLE-Technologie senden. Für das hier entwickelte Positionierungssystem werden drei *iBKS 105 Beacons* von *Accent Systems* verwendet. Diese übertragen in regelmäßigen Abständen geringe Mengen an Daten an mobile Geräte in der Umgebung. Dadurch ist es unter anderem möglich URLs (*Uniform Resource Locators*) an beispielsweise Smartphones zu senden oder Objekte, Haustiere oder Gepäck zu verfolgen (Accent Systems, o.J.).

Das *iBKS 105* verwendet das *Bluetooth Low Energy - Radio Protocol*. Die Hauptbestandteile des *Beacons* sind eine Knopfzelle (CR2477) mit 3 V und 1000 mAh und ein *Nordic nRF51822 - Chip* in einem weißen Gehäuse. Die Sendereichweite hängt von der Sendeleistung ab und wird in der Abbildung (2.7) verdeutlicht. Ebenfalls von der Sendeleistung abhängig ist die

Batterielebensdauer, zusätzlich sind auch die Anzahl der Sendeplätze mit entsprechendem *Beacon*-Protokoll, die Sendeperiode und *Beacon mode* (*connectable* oder *non-connectable*) entscheidend. Je nach Konfiguration beträgt die Laufzeit eines *iBKS 105 Beacon* mit einer 1000 mAh Batterie zwischen 16 und 48 Monate.

Das *iBKS 105* unterstützt das *iBeacon - Protocol* und das *Eddystone UDI, URL, TLM & EID - Protocol* (Accent Systems, 2016). Das *Eddystone - Protocol* ist ein offenes *Beacon* Format, das von Google entwickelt wurde. Es ermöglicht das übermitteln von Daten via BLE an mobile Geräte (Google Beacon Platform, o.J.). Zum *Eddystone* Format gehören:

- *Eddystone-UID*, welche eine eindeutige, statische 16 Byte ID darstellt
- *Eddystone-URL*, eine komprimierte URL, die vom *Beacon* übertragen wird
- *Eddysteone-TLM*, welche den Status des *Beacon* überträgt (Google Beacon Platform, 2018).

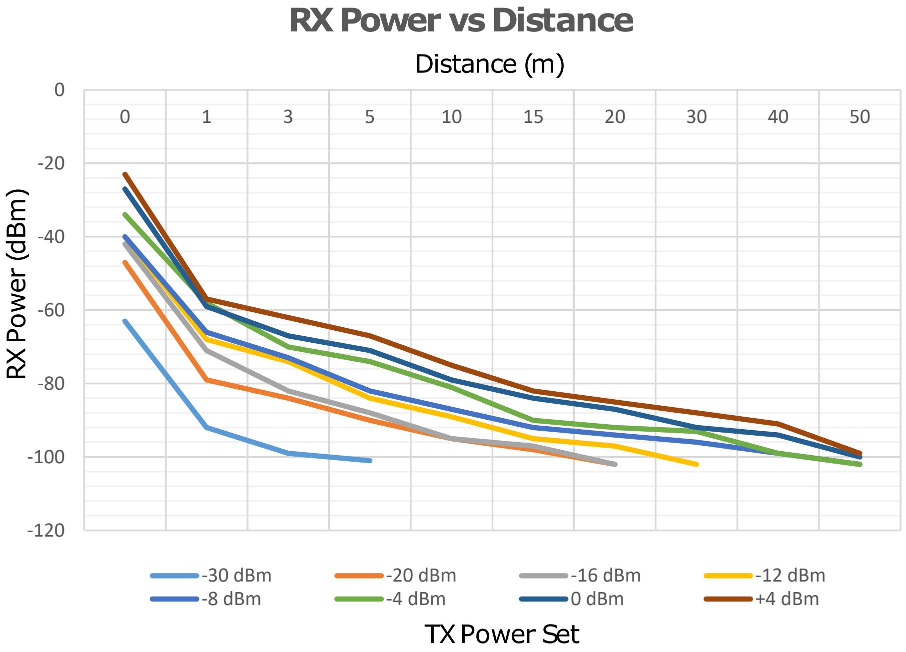


Abbildung 2.7: Abhängigkeit Sendereichweite und -leistung des *iBKS 105* (Accent Systems, 2016)

2.3 Mathematische Grundlagen

In den folgenden Abschnitten werden die wichtigsten mathematischen Grundlagen erklärt, die für die Entwicklung des *Indoor*-Positionierungssystems erforderlich sind. Dazu gehört die Trilateration, grundlegende Bewegungsgleichungen der Kinematik, Drehungen mit der Hilfe von Eulerwinkeln, sowie nichtlineare Regressionen und das Kalman-Filter.

2.3.1 Trilateration

Lateration ist ein Verfahren zur Positionsbestimmung aus den Distanzen zwischen Punkten mit bekannter Position (Ankerpunkte) und einem *Device* oder Nutzer mit unbekannter Position. Bei der Trilateration wird die Position eines *Devices* in einer zweidimensionalen Ebene mit drei Ankerpunkten bestimmt. Die Entfernung von einem *Device* zu einem Ankerpunkt kann beispielsweise aus der empfangenen Signalstärke errechnet werden. Als Voraussetzung dafür muss der Zusammenhang zwischen empfangener Signalstärke und der Entfernung bekannt sein. Für einen Ankerpunkt liegt die Position des *Devices* auf einem Kreis mit dem Abstand zum Ankerpunkt als Radius und dem Ankerpunkt mit bekannten Koordinaten als Mittelpunkt. Bei drei vorhandenen Ankerpunkten existieren drei Kreise, die sich unter idealen Umständen in einem Punkt schneiden. Dieser Schnittpunkt stellt die Position des *Devices* dar (Retscher, 2016). In der Praxis kommt es unter anderem durch Störungen und Messrauschen zu mehreren oder sogar zu keinen Schnittpunkten. Abbildung (2.8) zeigt mögliche Schnittpunkte zwischen Kreisen, in deren Mittelpunkt sich Ankerpunkte befinden.

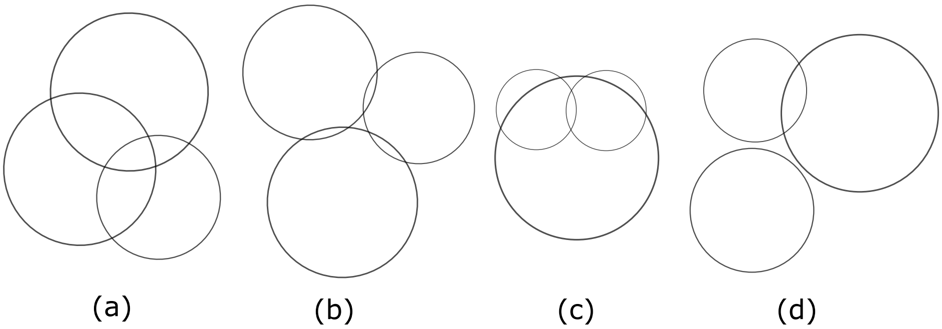


Abbildung 2.8: Schnittmöglichkeiten von Ankerpunktkreisen unter Einfluss von Rauschen (Nach Li, Yue, Chen & Deng, 2017)

Unterschieden wird zwischen Schnittpunkten, von denen sich einer im dritten Kreis befindet (a), Schnittpunkten, bei denen keiner im dritten Kreis liegt (b), Schnittpunkten, bei denen beide im dritten Kreis liegen (c) und Schnittpunkten zwischen nur zwei Kreisen (d). Diese Unterscheidung ist dann wichtig, wenn drei Schnittpunkte ausgewählt werden sollen, mit Hilfe derer die Position des *Devices* bestimmt wird. Existieren keine Schnittpunkte (d), können die Radien vergrößert werden, bis es zu einem oder mehreren Schnittpunkten kommt. Dies wird als *distance compensation* bezeichnet (Li et al., 2017).

Für die Berechnung der Schnittpunkte zweier Kreise A und B wird zunächst ein neues Koordinatensystem definiert, in dessen Ursprung sich der Mittelpunkt von Kreis A befindet. Dieser Mittelpunkt sei der Ankerpunkt P_i mit bekannter Position (x_i, y_i). Der zweite Kreis

B liegt mit seinem Zentrum, Ankerpunkt P_j mit bekannter Position (x_j, y_j), auf der x-Achse. Aus dem Radius a des Kreises A, dem Radius b des Kreises B und dem Abstand c der beiden Ankerpunkte P_i und P_j lassen sich nun die Schnittpunkte (x_{ij}, y_{ij}) und (x'_{ij}, y'_{ij}) wie folgt berechnen:

$$c = \mid \overrightarrow{B} - \overrightarrow{A} \mid = \sqrt{(P_{2x} - P_{1x})^2 + (P_{2y} - P_{1y})^2} \tag{2.10}$$

$$x_{ij} = x'_{ij} = \frac{a^2 + c^2 - b^2}{2 \cdot c} \tag{2.11}$$

$$y_{ij}, y'_{ij} = \pm\sqrt{a^2 - x^2} \tag{2.12}$$

(2.11) ist nur definiert, wenn die beiden Ankerpunkte nicht dieselben Koordinaten haben, also wenn $c \neq 0$. (2.12) ist nur definiert, wenn $a^2 \geq x^2$. Ist der Wert unter der Wurzel größer 0, so ergeben sich zwei y-Werte. Ist der Wert unter Wurzel genau 0, so berühren sich die beiden Kreise in einem Punkt. Andernfalls schneiden sich die beiden Kreise nicht.

Nachdem die Schnittpunkte der Kreise in einem Hilfskoordinatensystem bestimmt wurden, müssen diese nun wieder in das ursprüngliche Koordinatensystem transformiert werden. Dies geschieht mit Hilfe der Einheitsvektoren $\overrightarrow{e_1}$ und $\overrightarrow{e_2}$ des Hilfskoordinatensystems und einer Vektoraddition:

$$\begin{aligned}
\overrightarrow{e_1} &= \begin{pmatrix} e_{1,x} \\ e_{1,y} \end{pmatrix} = \frac{\overrightarrow{B} - \overrightarrow{A}}{\mid \overrightarrow{B} - \overrightarrow{A} \mid} = \frac{1}{c} \begin{pmatrix} B_x - A_x \\ B_y - A_y \end{pmatrix} \\
e_{1,x} &= \frac{B_x - A_x}{c} \quad \text{und} \quad e_{1,y} = \frac{B_y - A_y}{c} \\
\overrightarrow{e_2} &= \begin{pmatrix} e_{2,x} \\ e_{2,y} \end{pmatrix} = \begin{pmatrix} -e_{1,y} \\ e_{1,x} \end{pmatrix} \\
e_{2,x} &= -\frac{B_y - A_y}{c} \quad \text{und} \quad e_{2,y} = \frac{B_x - A_x}{c}
\end{aligned}$$

Die Schnittpunkte $Q_{i,j}$ und $Q'_{i,j}$ der Kreise A und B im ursprünglichen Koordinatensystem ergeben sich somit aus (Bislin, 2017):

$$\overrightarrow{Q}_{i,j}, \overrightarrow{Q'}_{i,j} = \overrightarrow{A} + x \cdot \overrightarrow{e}_1 \pm y \cdot \overrightarrow{e}_2 \tag{2.13}$$

Für die Trilateration existiert ein Kriterium, mit dem aus zwei Schnittpunkten zweier Kreise der ausgewählt wird, welcher dichter am Zentrum des dritten Kreises liegt. Dieses Kriterium wird folgendermaßen beschrieben:

$$(x_{ij} - x_k)^2 + (y_{ij} - y_k)^2 < (x'_{ij} - x_k) + (y'_{ij} - Y_k)^2, \tag{2.14}$$

wobei $(x_{ij,y_{ij}})$ und (x'_{ij}, y'_{ij}) die Schnittpunkte zweier Kreise darstellen und (x_k, y_k) den Mittelpunkt des dritten Kreises. Mit Hilfe von (2.14) können die optimalen Schnittpunkte gefunden werden, mit welchen anschließend die gesuchte Position bestimmt wird. Diese Methode kann jedoch nicht auf alle Schnittmöglichkeiten angewendet werden, wie Abbildung (2.9) zeigt (Li et al., 2017).

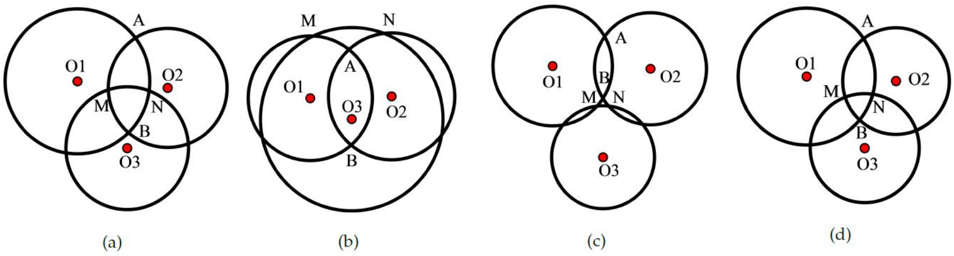


Abbildung 2.9: Schnittmöglichkeiten und Auswahl des optimalen Schnittpunkts (Li et al., 2017)

Nach Kriterium (2.14) ist in Abbildung (2.9) (a) bis (d) immer der Punkt B der optimale Schnittpunkt zur Berechnung der Position, obwohl die Wahrscheinlichkeit für die tatsächliche Position am Kreis des dritten Ankerpunktes höher ist als an dessen Mittelpunkt. Somit befindet sich nach Kriterium (2.14) für die Figuren (a) und (b) die gesuchte Position im Dreieck $\triangle BNM$, obwohl die Wahrscheinlichkeit, dass sich die gesuchte Position im Dreieck $\triangle AMN$ befindet, größer ist. Für die drei Ankerpunkte (x_i, y_i), (x_j, y_j), (x_k, y_k) und deren Radien d_i, d_j, d_k ergeben sich die Schnittpunkge (x_{ij}, y_{ij}) und (x'_{ij}, y'_{ij}). Um nun den neuen optimalen Schnittpunkt zu bestimmten, veränderten Li et al. das Kriterium (2.14) dahingehend, dass nun nicht mehr die Distanz zum Mittelpunkte des dritten Kreises ausschlaggebend für die Bestimmung des optimalen Schnittpunktes ist, sondern der Abstand zum dritten Kreis. Der neue optimale Schnittpunkt kann somit wie folgt bestimmt werden:

$$(d_k - \sqrt{(x_{ij} - x_k)^2 + (y_{ij} - y_k)^2})^2 < (d_k - \sqrt{(x'_{ij} - x_k)^2 + (y'_{ij} - y_k)^2})^2 \tag{2.15}$$

Ist das Kriterium (2.15) erfüllt, so ist (x_{ij}, y_{ij}) der neue optimale Schnittpunkt, andernfalls ist es (x'_{ij}, y'_{ij}) (Li et al., 2017).

Bei drei Kreisen, die sich nicht einen gemeinsamen Schnittpunkte teilen, sondern sechs Schnittpunkte haben, werden mit dem Kriterium (2.15) drei optimale Schnittpunkte bestimmt. Diese Punkte bilden ein Dreieck $\triangle BNM$, in dessen Zentrum sich die gesuchte Position befindet. Mit der folgenden Formel wird der Mittelpunkte $M(x, y)$ bestimmt (Xu, Tang & Li, 2017):

$$M(x,y) = \left(\frac{x_B + x_N + x_M}{3}, \frac{y_B + y_N + y_M}{3}\right) \tag{2.16}$$

2.3.2 Kinematik

Die Kinematik dient der Beschreibung der Bewegung eines Massepunkts. In diesem Projekt werden mittels eines Akzelerometers Beschleunigungen gemessen. Diese ergeben sich aus der Änderung der Geschwindigkeit pro Zeitintervall (Heintze, 2014):

$$a(t) = \frac{dv}{dt} = \frac{d^2x}{dt^2} = f''(t) \tag{2.17}$$

Bei bekannter Beschleunigung $a(t)$ als Funktion der Zeit, kann die Geschwindigkeit $v(t)$ und die Bahngleichung $x(t)$ durch Integration ermittelt werden. Daruch ergeben sich folgende Gleichungen, wobei v_0 und x_0 die Anfangsgeschwindigekeit und Anfangslage des Massepunktes darstellen:

$$a(t) = const \tag{2.18}$$

$$v(t) = v_0 + a \cdot t \tag{2.19}$$

$$x(t) = x_0 + v_0 \cdot t + \frac{a}{2} \cdot t^2 \tag{2.20}$$

Die gemessene Beschleunigung $a(t)$ wird für das diskrete Zeitintervall dt als konstant angesehen (Heintze, 2014).

2.3.3 Eulersche Winkel

Um die Lage eines Körpers im Raum zu beschreiben, hat Leonhard Euler die nach ihm benannten Eulerschen Winkel eingeführt. Sie dienen dazu, ein Bezugssystem S, welches durch die Einheitsvektoren $\hat{e}_i$ definiert werden, durch eine Drehmatrix $R(t)$ in das Bezugssystem S^*, definiert durch $\hat{e}_i^*$, zu überführen (Bartelmann et al., 2018). Euler stellte fest, dass jede

Drehmatrix als Produkt dreier einfacher Drehmatrizen dargestellt werden kann. Eine Drehung um die x-Achse mit dem Winkel α (*roll*-Winkel) wird wie folgt beschrieben:

$$R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos\alpha & -sin\alpha \\ 0 & sin\alpha & cos\alpha \end{pmatrix}$$

Die Drehung um die y-Achse mit dem Winkel β (*pitch*-Winkel):

$$R_y(\beta) = \begin{pmatrix} cos\beta & 0 & -sin\beta \\ 0 & 1 & 0 \\ sin\beta & 0 & cos\beta \end{pmatrix}$$

Drehung um die z-Achse mit dem Winkel γ (*yaw*-Winkel):

$$R_z(\gamma) = \begin{pmatrix} cos\gamma & -sin\gamma & 0 \\ sin\gamma & cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Die drei Einzeldrehungen lassen sich mit $R = R_x(\alpha) \cdot R_y(\beta) \cdot R_z\gamma$ zu einer einzigen Drehmatrix zusammenfassen. Dabei sind die Drehungen nicht kommutativ (Burg, Haf & Wille, 1987).

2.3.4 Nichtlineare Regression

Die nichtlineare Regression dient wie die lineare Regression der Untersuchung des Zusammenhangs zwischen einer Zielgröße Y und einer oder mehrerer Ausgangsgrößen x^j. Der Unterschied zur linearen Regression besteht in der geeigneten Funktion h, die von den Ausgangsgrößen und Parametern abhängt sowie nicht linear ist. Das allgemeine Modell einer linearen Regression lautet (Ruckstuhl & Werner, 2008):

$$Y_i = h\langle x_i^1, x_i^2, ..., x_i^m; \theta_1, \theta_2, ..., \theta_p\rangle + E_i \tag{2.21}$$

2.3.5 Kalman-Filter

Der amerikanische Mathematiker mit ungarischen Wurzeln Rudolf E. Kalman entwickelte 1960 ein spezielles Filter für zeitdiskrete und lineare Systeme. Das nach Rudolf E. Kalman benannte Kalman-Filter ermöglicht es, aus verrauschten und redundanten Messungen die Zustände und Parameter eines Systems zu schätzen. Besonders gut geeignet ist das Filter für Echtzeitanwendungen, da es einen iterativen Aufbau besitzt (Marchthaler & Dingler, 2017).

In Abbildung 2.10 sind die Beschleunigungswerte (in X-Richtung) des Beschleunigungssensors *LSM303* ungefiltert (in gelb) und als Schätzwerte des Kalman-Filters (in rot) dargestellt:

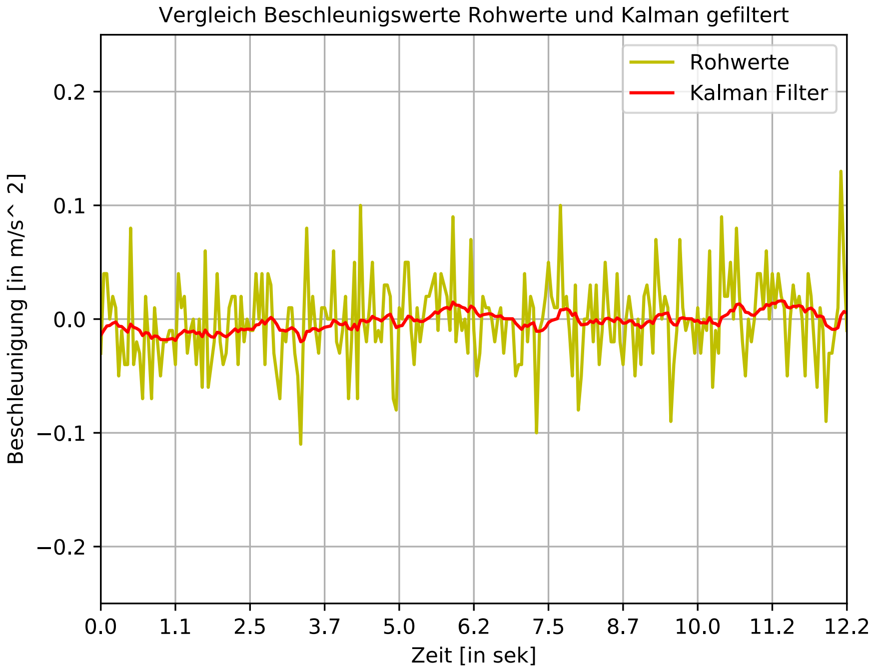


Abbildung 2.10: Messwertrauschen mit und ohne Anwendung des Kalman-Filters
Eigene Darstellung

1969 betrat Neil Armstrong als erster Mensch den Mond. Um nach der Ankunft der Apollo im Mondorbit mit der Mondlandefähre (*Lunar Module Eagle*) auf der Oberfläche des Mondes sicher landen zu können, war es notwendig den Ort und die Geschwindigkeit der Fähre präzise zu schätzen. Verwendet wurde dazu das Kalman-Filter, welches statistische und systematische Messfehler weitgehend eliminiert und damit ideal für diese Aufgabe war.

Die Höhe der Fähre über der Mondoberfläche wurde mit Hilfe von Radaranalgen bestimmt, welche sich auf der Erde befanden. Aufgrund der großen Entfernung zwischen Erde und Mond waren die Messungen jedoch nicht genau genug um sich allein darauf zu verlassen. Beschleunigungssensoren an Bord lieferten die Beschleunigung der Mondlandefähre senkrecht zur Mondoberfläche. Dadurch war es möglich die zunehmende Geschwindigkeit der Mondfähre beim Landeanflug durch rechtzeitiges Zünden der Bremsraketen zu verringern. Durch die Kombination verschiedener Sensoren und Messungen war es möglich die Astronauten sicher auf der Oberfläche des Mondes abzusetzen (Marchthaler & Dingler, 2017).

2015 veröffentlichte Tim Babb auf seinem Blog *BZARG* den Artikel „How a Kalman filter works, in pictures“ über die Funktion des Kalman-Filters. Dieses Filter findet in vielen Bereichen Verwendung, auch in der Positionsbestimmung. Fährt ein Fahrzeug, dessen Position über GPS erfasst wird, in einen Tunnel, so existiert keine Sichtverbindung zu den notwendigen Satelliten und das GPS-Signal bricht ab. Modernen Navigationssystemen ist es dennoch möglich die Position des Fahrzeugs zu bestimmen. Angenommen, die letzte Position $\overrightarrow{p}$ des Fahrzeuges befindet sich vor der Tunneleinfahrt und die aktuelle Geschwindigkeit $\overrightarrow{v}$ des Fahrzeuges wird durch Sensoren gemessen. Dann kann der Zustand des Systems x (des Fahrzeugs) zum Zeitpunkt k folgendermaßen beschrieben werden:

$$\overrightarrow{x_k} = (\overrightarrow{p}, \overrightarrow{v})$$

Die Anzahl der systembeschreibenden Parameter kann beliebig erweitert werden, beispielsweise durch Beschleunigung, Temperatur oder Höhenmessungen. Dabei ist es nicht möglich ein System so zu beschreiben, dass es der Wirklichkeit entspricht. Äußere Einwirkungen wie zum Beispiel Reibung, Luftwiderstand, Seitenwind, Unebenheiten der Straße und viele weitere können nicht oder nur schwer gemessen und dem System hinzugefügt werden. Zudem liefert jeder Sensor Messergebnisse, welche auch Messrauschen, Messungenauigkeiten oder redundante Messungen beinhalten.

Für die Schätzung der neuen Position wird davon ausgegangen, dass die Position und Geschwindigkeit normal verteilt sind und durch die Gauß-Verteilung beschrieben werden können:

$$N(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{2.22}$$

μ beschreibt den Mittelwert und somit das Zentrum der Verteilung der Messwerte. σ^2 ist die Varianz und wird im weiteren Verlauf als Unsicherheit der gemessenen Werte bezeichnet.

Wird die neue Position durch die alte Position und die Geschwindigkeit des Fahrzeugs bestimmt, so korreliert die Position mit der Geschwindigkeit. Sie sind somit nicht mehr unabhängig voneinander. Dieser Zusammenhang wird durch die Kovarianzmatrix Σ_{ij} beschrieben. Jedes Element dieser Matrix beschreibt den Grad der Korrelation der $i$ten und $j$ten Variablen.

Somit lässt sich nun die Position $\hat{x}_k$ des Systems zum Zeitpunkt k durch eine Normalverteilung zweier Variablen (Position p und Geschwindigkeit υ) beschreiben. $\hat{x}_k$ ist die Schätzung der Position (vorher als Mittelwert μ bezeichnet) und P_k die Kovarianzmatrix:

$$\begin{aligned} \hat{x}_k &= \begin{bmatrix} p \\ \upsilon \end{bmatrix} \\ P_k &= \begin{bmatrix} \Sigma_{pp} & \Sigma_{p\upsilon} \\ \Sigma_{\upsilon p} & \Sigma_{upsilonv} \end{bmatrix} \end{aligned}$$

Damit ist das System zum aktuellen Zeitpunkt $k-1$ beschrieben. Um nun eine Voraussage über den Zustand des Systems zum Zeitpunkt k treffen zu können, muss das System verschoben werden. Der Zustandsvektor zum aktuellen Zeitpunkt $\hat{x}_{k-1}$ wird durch die Matrix F_k (*prediction matrix*) zum Zustandsvektor des Systems zum Zeitpunkt k (Vorhersage):

$$\hat{x}_k = F_k \hat{x}_{k-1}$$

Im Beispiel des Fahrzeugs, welches in einen Tunnel einfährt, ergibt sich die neue Position aus der vorherigen addiert mit dem in der Zeit Δt zurück gelegten Weg. Vorausgesetzt die Geschwindigkeit des Fahrzeugs bleibt konstant.

$$\begin{aligned} p_k &= p_{k-1} + \Delta t \upsilon_{k-1} \\ \upsilon_k &= \upsilon_{k-1} \end{aligned}$$

$$\begin{aligned} \hat{x}_k &= \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \hat{x}_{k-1} \\ &= F_k \hat{x}_{k-1} \end{aligned}$$

Mit der *prediction matrix* F_k ergibt sich nun auch die Kovarinazmatrix des Systems zum Zeitpunkt k:

$$Cov(A \cdot (x, y)) = A \cdot Cov(x, y) \cdot A^T$$
$$P_k = F_k P_{k-1} F_k^T$$

Damit ist das System als solches zum Zeitpunkt k und $k - 1$ beschrieben. Jedoch ist beispielsweise ein Fahrzeug in der Lage zu beschleunigen (a) und somit einen externen Einfluss auf das System auszuüben. Dieser externe Eingang dient dazu, das System zu steuern und zu korrigieren. Dementsprechend wird er auch als Steuer- oder Eingangsvektor $\overrightarrow{u_k}$ bezeichnet.

Im Beispiel des Fahrzeugs im Tunnel ergibt sich die neue Position und Geschwindigkeit dann wie folgt:

$$\begin{aligned} p_k &= p_{k-1} + \Delta t \upsilon_{k-1} + \frac{1}{2} a \Delta t^2 \\ \upsilon_k &= \upsilon_{k-1} + a \Delta t \end{aligned}$$

$$\begin{aligned} \hat{x}_k &= F_k \hat{x}_{k-1} + \begin{bmatrix} \frac{\Delta t^2}{2} \\ \Delta t \end{bmatrix} \hat{x}_{k-1} a \\ &= F_k \hat{x}_{k-1} + B_k \overrightarrow{u_k}, \end{aligned}$$

wobei B_k als Steuermatrix bezeichnet wird.

Die neue Position des Fahrzeugs im Tunnel wird nun aus seiner alten Position, seiner Geschwindigkeit und der Beschleunigung geschätzt, mit der Fahrer die Geschwindigkeit des Fahrzeugs ändert. Wie jedoch bereits oben erwähnt, haben auch Kräfte Einfluss auf das System, deren genauen Betrag oder deren Richtung nicht ohne Weiteres bestimmt werden können, zum Beispiel die Reibung zwischen den Reifen und der Fahrbahn oder der Strömungswiderstand.

Diese zusätzlichen Unsicherheiten werden nun dem System mit einer weiteren Kovarianzmatrix Q_k hinzugefügt. Die Vorhersage (*prediction*) des Systems sieht nun wie folgt aus:

$$\hat{x}_k = F_k \hat{x}_{k-1} + B_k \overrightarrow{u_k}$$
$$P_k = F_k P_{k-1} F_k^T + Q_k$$

Im Fahrzeug verbaute Sensoren liefern nach einer gewissen Zeit neue Messwerte, welche nun in das System einfließen, um es zu aktualisieren. Dies geschieht im *update*-Schritt. Dabei werden zunächst die Skalen der Messwerte und die gesuchten Größen mit der Matrix H_k angepasst, da es möglich ist, dass Sensoren Werte in anderen Einheiten liefern. Somit ergeben sich der Mittelwert $\overrightarrow{\mu}$ und die Kovarianz Σ der Verteilung der möglichen Position zum Zeitpunkt k folgendermaßen:

$$
\begin{aligned}
\overrightarrow{\mu} &= H_k \hat{x}_k \\
\Sigma &= H_k P_k H_k^T
\end{aligned}
$$

Die eingehenden Messwerte der Sensoren werden ebenfalls als normalverteilt angesehen, wobei der eigentliche Messwert $\overrightarrow{z_k}$ den Mittelwert und die Kovarianzmatrix R_k die Unsicherheit der Messwerte angibt.

Es existieren nun zwei Schätzungen des Systems, welche durch die Gaußsche Normalverteilung (2.22) beschrieben werden. $N(x, \mu_0, \sigma_0)$ als vorhergesagte Messung und $N(x, \mu_1, \sigma_1)$ als beobachtete Messung werden nun zusammengeführt:

$$
N(x, \mu_0, \sigma_0) \cdot N(x, \mu_1, \sigma_1) = N(x, \mu', \sigma') \tag{2.23}
$$

Substituiert man nun (2.22) in (2.23) so ergibt sich:

$$
\begin{aligned}
\mu' &= \mu_0 + \frac{\sigma_0^2(\mu_1 - \mu_0)}{\sigma_0^2 + \sigma_1^2} \\
\sigma'^2 &= \sigma_0^2 - \frac{\sigma_0^4}{\sigma_0^2 + \sigma_1^2}
\end{aligned} \tag{2.24}
$$

Aus (2.24) ist es nun möglich die Kalman-Verstärkung k zu erhalten, auf dessen Einfluss auf die Schätzung des Systems im späteren Verlauf dieses Kapitels noch näher eingegangen wird.

$$
\begin{aligned}
k &= \frac{\sigma_0^2}{\sigma_0^2 + \sigma_1^2} \\
\mu' &= \mu_0 + k(\mu_1 - \mu_0) \\
\sigma'^2 &= \sigma_0^2 - k\sigma_0^2
\end{aligned} \tag{2.25}
$$

(2.25) trifft nur auf das Zusammenführen eindimensionaler Gauß-Verteilungen zu. Um dies auf Matrizen anwenden zu können, wird aus der Kalman-Verstärkung k die Matrix der Kalman-Verstärkung K und aus der Varianz σ^2 wird die Kovarianzmatrix Σ'. μ stellt die Mittelwerte der hinzugezogenen Variablen dar. Daraus folgt für $(\mu_0, \Sigma_0) = (H_k\hat{x}_k, H_k P_k H_k^T)$ und für $(\mu_1, \Sigma_1) = (\overrightarrow{z_k}, R_k)$.

$$\begin{aligned} H_k\hat{x}'_k &= H_k\hat{x}_k + K(\overrightarrow{z_k} - H_k\hat{x}_k) \\ H_k P'_k H_k^T &= H_k P_k H_k^T - K H_k P_k H_k^T \\ K &= H_k P_k H_k^T (H_k P_k H_k^T + R_k)^{-1} \end{aligned} \tag{2.26}$$

Durch das Kürzen beider Seiten durch H_k erhält man die endgültige Form des klassischen Kalman-Filters, welcher sich in zwei Funktionen aufteilt:

Prediction:

$$\begin{aligned} \hat{x}_k &= F_k\hat{x}_{k-1} + B_k\overrightarrow{u_k} \\ P_k &= F_k P_{k-1} F_k^T + Q_k \end{aligned} \tag{2.27}$$

Update (messen):

$$\begin{aligned} \hat{x}'_k &= \hat{x}_k + K'(\overrightarrow{z_k} - H_k\hat{x}_k) \\ P'_k &= P_k - K' H_k P_k \end{aligned} \tag{2.28}$$

$$K' = P_k H_k^T (H_k P_k H_k^T + R_k)^{-1} \tag{2.29}$$

Nun ist $\hat{x}'_k$ die neue beste Schätzung des Systems und P'_k ist die neue Kovarianzmatrix (Babb, 2015).

Die zwei Formeln der *prediction* (2.27) und die drei Formeln des *updates* (2.28 und 2.29) beschreiben das Kalman-Filter. Dieses ist iterativ, wodurch die *prediction* und das *update* unter Hinzunahme von aktuellen Beobachtungen des Systems (Messungen) ständig neu berechnet werden.

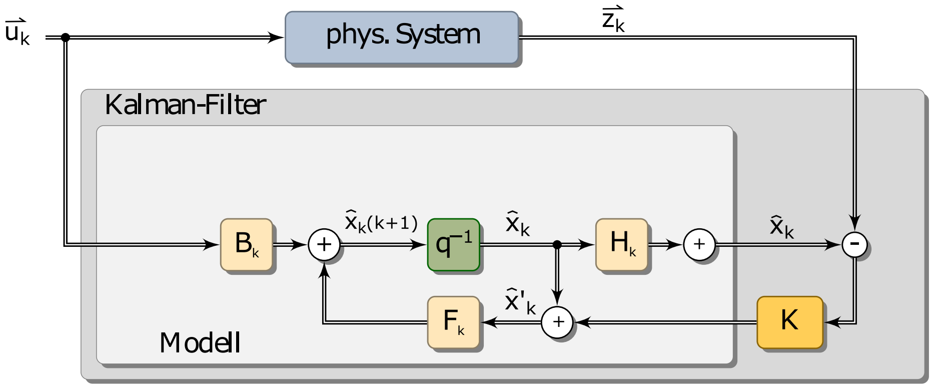


Abbildung 2.11: Struktur des Kalman-Filters
(Nach Marchthaler & Dingler, 2017)

Die in Abbildung 2.11 dargestellte Struktur verwendet ein lineares Modell zur Schätzung des Zustandsvektors $\hat{x}_k$. Diese geschätzte Ausgangsgröße wird mit der gemessenen Ausgangsgröße $\overrightarrow{z_k}$ (Messung) verglichen. Gewichtet durch die Kalman-Verstärkung K wird die Differenz der Ausgangsgrößen zur Korrektur des geschätzten Zustandsvektors $\hat{x}'_k$ verwendet. Dieser wird im darauf folgenden iterativen Schritt zur neuen geschätzten Ausgangsgröße, welche wiederum mit der neuen gemessenen Ausgangsgröße verglichen wird (Marchthaler & Dingler, 2017).

Wie bereits erwähnt, hat die Gewichtung durch die Kalman-Verstärkung erhebliche Auswirkungen auf den neuen Schätzwert. Wie in (2.29) zu sehen, ergibt sich die Kalman-Verstärkung aus der Unsicherheit der gemessenen Werte und der Unsicherheit der Schätzung. Im Falle sehr geringer Unsicherheiten in den gemessenen Werten, wird die Kalman-Verstärkung gegen eins gehen. Ein Wert von glatt eins ist unrealistisch, da dies bedeuten würde, dass die Messwerte perfekt die Realität darstellen, was wiederum nur mit einem perfekten Sensor funktioniert. Im Falle sehr großer Unsicherheiten der Messungen nimmt die Kalman-Verstärkung Werte gegen null an.

Unter der Annahme, es würde ein perfekter Sensor verwendet und die Kalman-Verstärkung erreiche den Wert eins, so ergäbe sich in (2.28) der neue Schätzwert $\hat{x}'_k$ nur aus dem neuen Messwert $\vec{z}_k$. Nimmt die Kalman-Verstärkung jedoch den Wert null an, so ergäbe sich der neue Schätzwert aus dem alten $\hat{x}_k$. Die Kalman-Verstärkung hat somit großen Einfluss auf die neue Schätzung des Systems (o.V., 2004).

3 System zur Indoor-Positionierung

In diesem Kapitel werden zunächst einige Vor- und Nachteile von Techniken zur Indoor-Positionierung aus Kapitel 2.1 genannt. Unter verschiedenen Aspekten, wie beispielsweise Kosten, Genauigkeit und Verfügbarkeit, werden diese Systeme verglichen. Anschließend wird erläutert, welche Technologien für die Entwicklung des hier vorgestellten Systems zu Indoor-Positionierung verwendet werden.

Im darauf folgenden Abschnitt wird das Konzept des Systems mit den dazugehörenden Komponenten und dem gesamten Aufbau beschrieben. Anschließend folgt die Umsetzung des Konzepts, einschließlich der Programmierung aller Sensoren, sowie der Sensorfusion und Berechnung der dreidimensionalen Position.

Die gesamte Anwendung wird auf einem Bewegungssimulator, einem *Arduino Robot*, getestet. Der Testaufbau wird im letzten Abschnitt dieses Kapitels beschrieben.

3.1 Verwendete Technologien

Wie in Kapitel 2.1 beschrieben, existieren zahlreiche Methoden und Technologien zur Bestimmung der Position eines Geräts oder einer Person in Gebäuden. Zusätzlich werden Neue entworfen und Vorhandene stetig weiter entwickelt. Positionierungssysteme werden in vielen verschiedenen Gebäuden verwendet. Ebenso verschieden ist deren Einsatz. So können beispielsweise Gerätschaften wie Baumaschinen, aber auch medizinische Apparaturen geortet werden, Gäste navigieren sich auf großen Veranstaltungen oder in Einkaufszentren und Besucher[3] in Museen finden schnell zu dem gesuchten Ausstellungsstück. Die an das System gestellten Anforderungen hängen stark von der eigentlichen Anwendung ab. Zusätzlich gibt es technische und wirtschaftliche Anforderungen, welche abhängig vom Einsatzgebiet, erfüllt werden müssen. Es gibt somit kein System zur Indoor-Positionierung, welches allen anderen überlegen ist. Wichtig ist, dass das entwickelte System den jeweils gestellten Anforderungen entspricht (Retscher & Kistenich, 2006). Dementsprechend werden zunächst die Anforderungen und der Einsatzort definiert.

Das IoT verbindet verschiedene Geräte wie *Smart Objects*, Maschinen und viele weitere miteinander. Objekte, die über IoT miteinander kommunizieren, können eigenständig Entscheidungen treffen und Aktionen auslösen. Dafür ist es mitunter notwendig, andere Objekte zu lokalisieren. Unterschieden werden muss dazu zwischen Konzepten für die Industrie und für den Verbraucher (ITWissen, 2018). Die hier entwickelte Methode zur Indoor-Positionierung ist

---

[3] Auf geschlechtsneutrale Formulierungen wurde aus Gründen der Lesbarkeit verzichtet. Im Text sind immer alle Geschlechter gemeint. Trotzdem wurde, ohne Anspruch auf Vollständigkeit, auf geschlechtsneutrale Formulierungen geachtet.

primär für den Verbraucher bestimmt. Autonom fahrende Systeme wie Autos, Drohnen aber auch Roboter unterstützen den Menschen im Alltag und sind im IoT miteinander verbunden. Selbst ein Parkplatz kann mit Sensoren ausgestattet werden, um zu überprüfen ob er belegt ist. Ein freier Parkplatz ist somit in der Lage, einem autonom fahrenden Fahrzeug über dessen Status (belegt oder frei) Auskunft zu geben. Damit das Fahrzeug sich im Parkhaus navigieren kann, ist es von Vorteil, wenn es seine Position kennt. Ein weiteres mögliches Szenario stellt ein Putzroboter in einem Einkaufszentrum dar. Immer wieder kommt es vor, dass ein Besucher versehentlich Abfall fallen lässt. Ein Putzroboter sammelt den Müll automatisiert auf. Dazu ist es ebenfalls notwendig, die Position des Roboters zu kennen. Es gibt noch zahlreiche weitere Szenarien, für welches das hier entwickelte Positionierungssystem Anwendung findet. Eines haben jedoch alle Beispiele gemeinsam. Die zu lokalisierenden Objekte besitzen Sensoren, die genutzt werden können, um deren Bewegung im Raum nachzuvollziehen. Des Weiteren sind die Objekte in der Lage, ihre nähere Umgebung, beispielsweise die Entfernung zu anderen beweglichen Objekten, zu bestimmen. Es genügt somit, die Position des Systems hinreichend genau zu berechnen. Weiterhin ist es von Vorteil, die bereits existierenden Technologien und Infrastrukturen zu nutzen. Dies kommt dem wirtschaftlichen Aspekt zugute und senkt die Kosten für das System. Neben der bereits vorhandenen Infrastruktur, werden *Low-cost*-Komponenten verbaut und es wird darauf geachtet, eine hohe Energieeffizienz zu erzielen. Aufgrund der zahlreichen Einsatzgebiete mit vielen verschiedenen Sensoren und Technologien wird ein System entwickelt, welches flexibel ist und gut erweitert werden kann, um die Genauigkeit, Ausfallsicherheit und Störanfälligkeit auch noch nachträglich zu verbessern.

Zunächst muss geklärt werden, ob es sich bei dem zu entwickelnden Positionierungssystem um eine server- oder clientseitige Positionsbestimmung handelt. Beide Methoden weisen in Abhängigkeit des Einsatzgebiets Vor- und Nachteile auf. Bei der serverseitigen Ortung sendet ein Gerät ein Signal aus, welches von Empfängerstationen empfangen wird.

Das von Zhang et al. (2006) in Kapitel 2.1.2 vorgestellte System verwendet dazu die Laufzeitdifferenzmessung. Die Verarbeitung und Berechnung der Position findet dann auf einem Server statt und nicht bei dem Gerät, das geortet wird. Ein Vorteil besteht unter anderem darin, dass Server meist eine höhere Rechenkapazität besitzen, wodurch die Berechnung schneller ausgeführt wird. Weiterhin kann die Position des Objekts direkt auf einer Webseite angezeigt werden oder der Server führt aufgrund der Position bestimmte Aktionen aus. *Smart Objects*, die über das IoT kommunizieren, können dies jedoch ebenfalls und benötigen dazu keinen Server. Die Positionsbestimmung über einen Server ist somit eher für Maschinen und Systeme geeignet, die nicht über das IoT miteinander verbunden sind. Ein Nachteil ist, dass die benötigte

Rechenkapazität mit der Anzahl der Objekte, die geortet werden müssen, steigt. Bei steigender Zahl der Besucher auf Messen oder in Einkaufszentren und mehreren Fahrzeugen in Parkhäusern steigen auch die Anforderungen an den Server, was häufig mit höheren Kosten verbunden ist.

Bei clientseitigen Systemen wird die Position des zu ortenden Objekts von ihm selbst berechnet. Daraufhin ist dieses in der Lage, eigenständig Entscheidungen zu treffen oder Aktionen auszulösen. Vorteile bestehen darin, dass die verschiedenen internen Sensoren des Objekts zur Bestimmung der Position sehr einfach hinzugezogen werden können, wodurch eine höhere Genauigkeit erzielt wird. Des Weiteren bleibt die Information über die berechnete Position bei dem Objekt selbst und wird nur an andere Objekte übermittelt, wenn dies gewünscht ist. Die clientseitige Positionierung wird dann erschwert, wenn die vorhandene Infrastruktur keine nutzbaren Signalquellen besitzt, oder diese nicht nutzbar gemacht werden können. Ein Objekt, welches seine Position aus der Signalstärke von Wlan-Signalen ermittelt, kann dies offensichtlich nicht tun, wenn kein Wlan vorhanden ist. Eine Ortung über die Signalstärke von Wlan-Signalen ist jedoch ebenfalls nicht möglich, wenn die Position der *Router* unbekannt ist. Somit ist das benötigte Signal zwar vorhanden, kann aber nicht genutzt werden. Ein System, welches sich ohne externe Signale ortet, ist auf seine internen Sensoren angewiesen. Diese Form der Positionsbestimmung nennt sich inertiale Navigation und wurde in Kapitel 2.1.3 beschrieben.

Das hier entwickelte System kann Signalquellen der vorhandenen Infrastruktur nutzen, um diese mit internen Sensoren zu verbinden. Verwendet wird eine clientseitige Lokalisierung, wodurch zunächst nur das Objekt selbst seine Position kennt. Je nach Anwendungsszenario ist es möglich, die Koordinaten an einen Server oder an ein anderes Objekte zu übermitteln.

Es existieren viele verschiedene Signalarten, die sich für eine Ortung in Gebäuden eignen. Sie alle weisen bestimmte Eigenschaften auf, wodurch sie sich für spezielle Anforderungen eignen. Infrarotes Licht beispielsweise ist leicht erhältlich und relativ preiswert. Es wird in einem Raum an Objekten reflektiert, wodurch keine direkte Sichtlinie zwischen Sender und Empfänger notwendig ist. Das Signal kann Mauern nicht durchdringen, dafür jedoch eine Sendeidentifikation übertragen, um verschiedene Benutzer zu unterscheiden. Deren genaue Position ist jedoch nicht feststellbar (Retscher & Kistenich, 2006). Aus diesem Grund wird die Methode nicht weiter vertieft. Ebenso wird kein Ultraschall verwendet, obwohl es auf den ersten Blick als vielversprechende Technik zur Positionsbestimmung erscheint. Ultraschall ist erheblich langsamer als elektromagnetische Wellen und abhängig vom Medium, in dem es sich der Schall ausbreitet, und dessen Temperatur. Das Medium (Luft) ist bekannt und die Temperatur kann über andere Sensoren ermittelt werden. Die Entfernung zu einem Objekt ergibt sich aus der

Zeit, die der Schall benötigt, um vom Sender zum Objekt und wieder zurück zu gelangen. Mit der Schallgeschwindigkeit kann nun die Distanz sehr genau berechnet werden, wobei die vom Schall zurück gelegte Strecke noch durch zwei dividiert werden muss. Die Sensoren sind günstig und gut erhältlich (Retscher & Kistenich, 2006). Das *HC-SR04* ist bereits für unter drei Euro[4] erhältlich. Es besitzt jedoch nur eine Reichweite von etwa $3,5\,m$. Für die Anwendung in Tiefgaragen oder Einkaufszentren müssten somit sehr viele Sensoren verbaut werden, um eine flächendeckende Positionserfassung zu gewährleisten. Der Installationsaufwand wäre sehr groß, da Ultraschall in der Regel nicht in öffentlichen Gebäuden vorhanden ist. Aus diesem Grund wird Ultraschall in dieser Arbeit nicht zur Lokalisierung eingesetzt.

Funksignale des UWB werden, wie in Kapitel 2.1.2 beschrieben, ebenfalls zur *Indoor*-Positionierung eingesetzt. Mit ihnen werden sehr hohe Genauigkeiten erzielt und die gesendeten Signale sind fähig bestimmte Materialien zu durchdringen. Wie bereits erwähnt, verwendeten Zhang et al. (2006) eine Laufzeitdifferenzmessung für UWB, bei der das gesuchte Objekt als Sender diente. Dies ist eine serverseitige Lösung, welche aus den oben genannten Gründen für das hier entwickelte System nicht in Frage kommt. Ein clientseitiges Verfahren für UWB ist die *Time of arrival* Methode. Ähnlich wie beim Ultraschall wird die Zeit gemessen, welche das Signal benötigt, um die Entfernung zu überwinden (Retscher & Kistenich, 2006). Der Unterschied besteht darin, dass für UWB Laufzeitmessungen der Sender und Empfänger synchronisiert werden müssen. Dies ist für eine Anwendung mit vielen Clients nur schwer möglich. Die Kosten für einen Sensor wie das *ScenSor-Modul DWM1000* von *Decawace* mit etwa 25 Euro[5] sind erschwinglich. Dennoch wird diese Technik hier nicht verwendet, da, wie bereits erwähnt, vorhandene Infrastrukturen genutzt werden sollen, um den Installationsaufwand und somit die Kosten zu senken.

Signale, die bereits jetzt sehr weit verbreitet sind und die sich für eine Positionierung in Gebäuden eignen, sind Bluetooth und Wlan. Zudem besitzen viele Geräte bereits verbaute Sensoren und Antennen, um diese Signale empfangen zu können. In Kapitel 2.1.1 wurde ein System von Rida et al. (2015) vorgestellt, welches Wlan und Bluetooth nutzt, um ein Objekt zu orten. In einem Experiment erreichten sie eine Genauigkeit von 0 bis 2 Metern, was für das hier entwickelte System als hinreichend genau definiert wird. Wlan findet sich an jedem Wifi-*Hotspot* und auch Bluetooth findet immer mehr Verwendung, besonders im Zusammenhang mit IoT und *Smarthome*-Technologien.

Mit der Entwicklung von BLE wurde die Energieeffizient von Bluetooth vergrößert, zusätzlich ist es günstiger als Wlan (Er Rida et al., 2015). Mit beiden Signalen lassen

---

[4] https://www.ebay.de/i/272450296960?chn=ps&var=571456608064 [Stand 11.11.2018]

[5] https://www.semiconductorstore.com/cart/pc/viewPrd.asp?idproduct=50013 [Stand 11.11.2018]

sich Informationen übertragen. Ein BLE-*Beacon* kann, wie in Kapitel 2.2.8 beschrieben, Informationen an einen *Client* übermitteln, ohne sich mit ihm zu verbinden. Dies ist ein erheblicher Vorteil, wenn es darum geht mehrere Geräte in einem Raum zu lokalisieren.

Der Installationsaufwand für Bluetooth und Wlan ist geringer als der von UWB und Ultraschall, da diese Technologien bereits häufig vorhanden sind. Eines muss jedoch bei allen Signalarten, unabhängig der Verfügbarkeit, getan werden. Für jede Methode muss die Position der Sender, zum Beispiel der Wlan-*Router* oder der BLE-*Beacon*, oder des Empfängers, die Basisstationen bei einer serverseiten UWB-Lokalisierung, bekannt sein. Je nach Art der Anwendung genügt es, die Position in einem eigenen Koordinatensystem des Gebäudes zu kennen, oder in geografischen Koordinaten. Zusätzlich muss das zu ortende Objekt diese Positionen kennen.

Eine Methode zur Positionierung, bei der keine externen Signale notwendig sind, ist die inertiale Navigation. Diese wurde in Kapitel 2.1.3 genauer beschrieben. Die für diese Art der Lokalisierung notwendigen Sensoren sind häufig in den zu ortenden Geräten verbaut. Daher werden sie für das hier entwickelte System genutzt. Diese mikromechanischen Systeme (MEMS, siehe Kapitel 2.2) sind günstig und benötigen nur wenig Energie. Ein großer Nachteil ist jedoch, dass sich Messfehler bei einer inertialen Navigation, die nur auf MEMS beruht, fortpflanzen. Mit zunehmender Messdauer weicht die ermittelte Position somit immer mehr von der wirklichen ab.

Die Anforderungen und möglichen Einsatzgebiete des hier entwickelten Systems zur Positionierung in Gebäuden wurden bereits am Anfang des Kapitels beschrieben. Es folgten einige Technologien, die sich aufgrund ihrer Eigenschaften mehr oder weniger für diese Arbeit eignen. Die ausgewählte Technologie für die Positionierung in der zweidimensionalen xy-Ebene ist die Messungen der Signalstärke von Bluetooth-Signalen. Mindestens drei eingemessene BLE-*Beacon* senden in regelmäßigen Abständen Informationen über ihren genauen Standort. Das System empfängt diese und berechnet daraus dessen Position. Interne mikromechanische Systeme sollen helfen, den Einfluss von Messrauschen und *Multipath*-Effekten zu verringern und somit die Genauigkeit der Position zu erhöhen. Für die Berechnung der Höhe des Systems werden ebenfalls MEM-Systeme verwendet. Es ergibt sich somit ein *low-cost* Positionierungssystem, welches energiesparende BLE- und MEMS-Technologien verwendet. Die Genauigkeit der BLE-Signalstärkemessungen ist hinreichend genau und wird durch weitere Sensoren verbessert. Es entsteht ein clientseitiges und flexibles Positionierungssystem für den Verbraucher, welches je nach Anforderungsgebiet erweitert werden kann.

3.2 Konzept

Dieses Kapitel dient dazu, die Idee und das Konzept des hier entwickelten Systems zur *Indoor*-Positionierung zu erläutern. Dabei wird zuerst auf die Planung des zeitlichen Ablaufs eingegangen. Außerdem liegt ein Augenmerk auf den technischen Problemen, die im Laufe der Entwicklung gelöst werden müssen. Anschließend werden der Aufbau und das Zusammenspiel aller einzelnen Komponenten erläutert.

Für die zeitliche Aufteilung und Planung der Arbeit wird das Projekt in sechs Phasen eingeteilt. Mit dem Ende jeder Phase wird ein Meilenstein erreicht. Eine Übersicht über den genauen Ablauf, alle Phasen und Meilensteine befindet sich im Anhang (siehe Abbildung zu Kapitel 3.2). In der ersten Phase geht es um die Themenfindung und Festlegung der Fragestellung, welche mit dem Meilenstein „Start der Masterarbeit“ abgeschlossen wird. Es folgt die Phase der Programmierung aller Sensoren und die Zusammenführung aller Komponenten auf dem Raspberry Pi. Während und am Ende dieser Phase werden Tests durchgeführt, um die entwickelten Systemkomponenten zu verbessern. Ab der Mitte der zweiten Phase beginnt die dritte Phase, in der erste Ergebnisse und Methoden in einer schriftlichen Rohfassung der Arbeit festgehalten werden. Mit den Enden der Phasen zwei und drei beginnt die vierte Phase, die Schreibphase. In dieser Phase werden Änderungen der Rohfassung vorgenommen, gegebenenfalls die Struktur angepasst und die noch fehlenden Teile der Arbeit verfasst. Ab der Hälfte der vierten Phase beginnt die fünfte Phase, in der bereits geschriebene Abschnitt auf Vollständigkeit und logischen Zusammenhang überprüft werden. Mit dem Ende der Korrektur folgt die letzte Phase, die Abschlussphase, in der die Arbeit zu einer druckfertigen Datei zusammengestellt wird.

Das hier entwickelte System dient der dreidimensionalen Positionsbestimmung von Objekten im *Indoor*-Bereich. Es basiert auf *Low-cost*-Komponenten und arbeitet zuverlässig und energieeffizient. Der Einsatzort liegt bei autonom fahrenden Systemen. Das System muss flexibel und erweiterbar sein, damit es den Anforderungen entsprechend angepasst werden kann. Ziel des hier entwickelten System ist nicht die Position eines Objekts mit der Genauigkeit industrieller oder vermessungsstechnischer Standards zu bestimmen. Vielmehr soll es einem autonomen System möglich sein, sich in einer neuen Umgebung ohne GNSS navigieren zu können. Dazu nutzt es eigene Sensoren sowie Bluetooth Signale, die von BLE-*Beacon* gesendet werden. Vorhandenen interne Sensoren des Objekts in Verbindung mit dem Bluetooth-Signal erhöhen die Genauigkeit. Die BLE-*Beacon* übertragen eine URL über das Bluetooth-Signal. Mit einem *Request* an diese URL erhält das Objekt alle notwendigen Informationen über den *Beacon*, wie zum Beispiel dessen genaue Position. Mit mehreren *Beacon* ist das Objekt nun in der Lage, seine Position zu bestimmen. Neben der Postion des *Beacon* können über die URL auch beliebig viele weitere Informationen übertragen werden, wie zum Beispiel die Ausgangsposition oder Kartenmaterial der Umgebung. Das Objekt muss sich somit nicht erst mit einem Server der Umgebung verbinden, um Informationen zu erhalten oder um sich mit Sender- / Empfängerstationen zu synchronisieren. Somit hat niemand, außer dem autonomen System selbst, Kenntnis von dessen Position.

Das zu entwickelnde System besteht grundsätzlich aus vier Sensoren, zwei Einheiten um Sensordaten zu verarbeiten und aus drei BLE-*Beacon* in Verbindung mit einem Server. Die zwei Einheiten zur Datenverarbeitung sind das Arduino Uno und das Raspberry Pi 3. Der Aufbau und die Funktion aller technischen Komponenten des Systems wurden in Kapitel 2.2 beschrieben. Den Hauptteil der Datenverarbeitung übernimmt das Raspberry Pi. Das Arduino verarbeitet lediglich die Daten der an ihm angeschlossenen Sensoren. Dazu gehören ein Beschleunigungssensor, ein Gyroskop und ein Magnetometer. Abbildung 3.1 zeigt die schematische Verarbeitung der Sensordaten auf dem Arduino. Daten werden in elliptischen Feldern dargestellt, Prozesse und Berechnungen in rechteckigen. Die Darstellung des gesamten Systems befindet sich im Anhang (siehe Abbildung zu Kapitel 3.2).

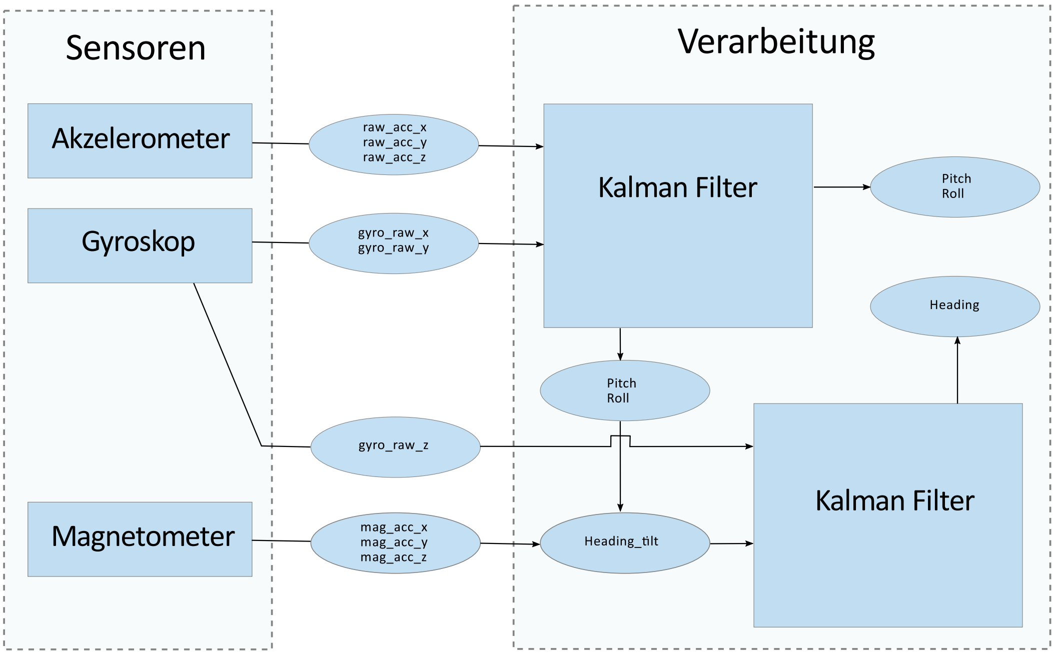


Abbildung 3.1: Konzept der Datenverarbeitung auf dem Arduino Uno
Eigene Darstellung

Die drei Sensoren liefern rohe Messwerte. Um aus den Daten zuverlässige Informationen über den aktuellen Systemzustand zu gewinnen, werden diese miteinander fusioniert und gefiltert. Vor diesem Schritt ist es notwendig, die Sensoren zu kalibrieren, da jeder gemessenen Wert ein unbestimmtes *Offset* beinhaltet. Dadurch misst ein Sensor bei jeder Messung etwas zu viel oder zu wenig. Dies kann die späteren Ergebnisse verfälschen. Die Kalibrierung der einzelnen Sensoren wird in Kapitel 3.3.1 beschrieben.

Für die Sensorfusion und Filterung der Messwerte wird das Kalman-Filter verwendet. Aus den Daten des Akzelerometers lässt sich der Winkel zwischen dem System und dem Gravitationsvektor berechnen. Dadurch erhält man den *pitch*- und *roll*-Winkel des Systems aus den Beschleunigungswerten des Akzelerometers. Diese allein sind jedoch sehr störanfällig und werden durch äußere Kräfte, die auf das System einwirken, beeinflusst. Die Sensorfusion der beiden Winkel zusammen mit den Drehraten des Gyroskops in x- und y-Richtung ergeben stabile und weniger störanfällige Werte für die *pitch*- und *roll*-Winkel des Systems.

Mit Hilfe des Magnetometers wird ein Kompass entwickelt, der den aktuelle Kurswinkel des Systems misst. Dieser Sensor wird von ferromagnetischen Metallen wie Eisen und von stromdurchflossenen Leitern beeinflusst, die ein Magnetfeld erzeugen. Er muss daher ebenfalls kalibriert werden. Zusätzlich sind die gemessenen Werte in x-, y- und z-Richtung abhängig von der Lage des Sensors im Raum. Es ist somit notwendig, die Messwerte durch eine Neigungskompensation zu korrigieren. Die daraus resultierenden *Heading_tilt*-Werte werden mit den Drehraten des Gyroskops um die z-Achse in einem Kalman-Filter fusioniert und gefiltert.

Über eine serielle Verbindung gibt das Arduino die Messwerte des kalibrierten Beschleunigungssensors, den Kurswinkel (*Heading*), den *pitch*- und *roll*-Winkel an das Raspberry Pi weiter. Dort werden die Daten weiter verarbeitet und mit anderen Sensorwerten kombiniert. Weitere Daten, die auf dem Raspberry Pi empfangen und verarbeitet werden, sind die Bluetooth-Signale der BLE-*Beacon*. Abbildung 3.2 zeigt ein Schema der Verarbeitung der empfangenen Bluetooth Signale auf dem Raspberry Pi.

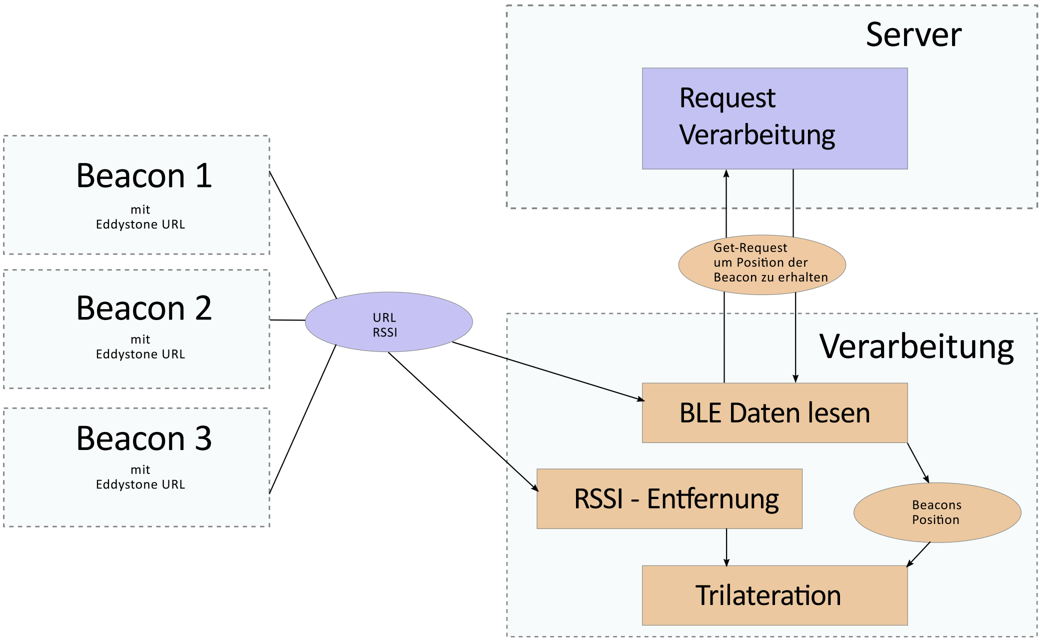


Abbildung 3.2: Konzept der Datenverarbeitung von Bluetooth Signalen auf dem Raspberry Pi 3

Eigene Darstellung

In Abbildung 3.2 werden die Daten und Prozesse in zwei Farben dargestellt. Orange steht dabei für das Raspberry Pi und lila für Daten und Prozesse, die sich auf Bluetooth und einen Server beziehen. Die Bedeutungen der Formen sind dieselben wie in Abbildung 3.1.

Jeder BLE-*Beacon* sendet in vorher festgelegten Intervallen ein Signal aus. Dieses enthält unter anderem eine URL, unter der Informationen des jeweiligen *Beacon* zu finden sind. Nachdem das Raspberry Pi beim Scanvorgang ein Bluetooth-Signal empfangen hat, wird an die übermittelte URL ein *get-Request* gesendet. Ein Server dient dazu, die *Requests* zu verarbeiten und die geforderten Informationen als Antwort zurückzusenden. Neben den gesendeten Informationen wird auch die Signalstärke jedes *Beacon* gespeichert. Aus den Signalstärken ist es möglich, die Entfernung zu jedem *Beacon* zu bestimmen. Dazu ist es notwendig, zuvor den Zusammenhang zwischen Signalstärke und Distanz zu untersuchen. Mit der Entfernung der Signalstärken und den Positionen der Sender wird nun eine Trilateration durchgeführt. Die sich ergebende Position in (x, y) wird stark schwanken, da Objekte und Wände die Bluetooth-Signale reflektieren. Daher werden diese Informationen weiter verarbeitet und mit anderen Daten kombiniert, um die Genauigkeit zu erhöhen.

Abbildung 3.3 zeigt die letzten Schritte der Verarbeitung aller Daten. Dies geschieht auf dem Raspberry Pi, auf dem alle Daten zusammenlaufen. Da es sich bei den vom Arduino übermittelten Beschleunigungswerten um Rohwerte handelt, beinhalten diese nicht ausschließlich die Beschleunigungen des Systems. Zusätzlich wird auch die Erdbeschleunigung gemessen. Abhängig von der Lage des Sensors beeinflussen die Beschleunigungswerte ein, zwei oder alle drei Achsen des Akzelerometers. Eine Gravitationskompensation wird genutzt, um den Anteil der Erdbeschleunigung von jeder Achse zu subtrahieren. Mit der Beschleunigung in z-Richtung ohne die Erdbeschleunigung ist es nun möglich, die Höhe des Systems zu bestimmen. Ein Kalman-Filter fusioniert Höhenwerte des Barometers mit den Beschleunigungen. Die Höhenwerte ergeben sich aus dem Luftdruck. Da dieser jedoch wetterabhängig ist und schwankt, wird versucht die Genauigkeit durch das Akzelerometer zu erhöhen.

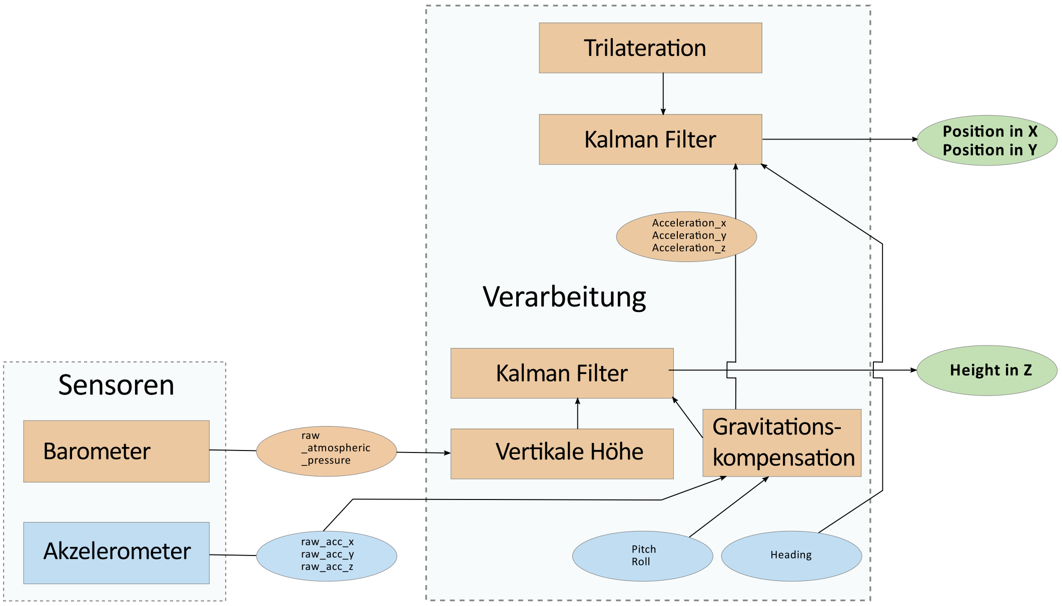


Abbildung 3.3: Konzept der Datenverarbeitung auf dem Raspberry Pi 3
Eigene Darstellung

Um die Position in der zweidimensionalen Ebene zu bestimmen, wird, wie oben beschrieben, eine Trilateration aus Signalstärken verwendet. Für zusätzliche Genauigkeit sorgt ebenfalls der Beschleunigungssensor. Nach der Gravitationskompensation werden dessen Werte, der Kurswinkel des Magnetometers und das Ergebnis der Trilateration in einem weiteren Kalman-Filter fusioniert. Der Kurswinkel dient dazu, das Koordinatensystem des Sensors zu drehen, damit es mit dem der BLE-*Beacon* übereinstimmt. Dadurch wird erreicht, dass der Sensor Beschleunigungen nicht in x- und y-Richtung misst, sondern nach Osten und Norden.

Schon vor der Umsetzung des Konzepts konnten bereits einige Herausforderungen identifiziert werden, welche bei der Umsetzung beachtet werden müssen. Zum einen müssen für die Berechnung der Position die Beschleunigungswerte zweifach integriert werden. Jede Messunsicherheit und jeder Messfehler wird dadurch mit integriert. Je länger eine Messung dauert, desto größer wird die Abweichung der berechneten Position von der wirklichen. Zum anderen besitzt jeder Sensor andere Updateraten, die berücksichtigt werden müssen. Für alle

Sensoren existieren fertige Bibliotheken, mit deren Hilfe die Daten aus den Registern gelesen werden. Voraussichtlich bieten einige auch die Möglichkeit der Sensordatenfusion. Diese müssen getestet werden, um anschließend zu entscheiden, ob sie den Anforderungen des hier entwickelten Systems gerecht werden. Beispielsweise müssen die *pitch*- und *roll*-Winkel trotz Erschütterungen und Vibrationen relativ konstante Werte liefern. Die ist notwendig, weil das fertige System auf einem *Arduino Robot* montiert und getestet wird.

Alle Verbindungen zwischen den Sensoren und dem Arduino Uno sowie dem Raspberry Pi 3 sind im Anhang (siehe Abbildung zu Kapitel 3.2) in einem Schema dargestellt, welches mit der *fritzing*-Software der *Friends-of-Fritzing Foundation* erstellt wurde. Die roten und schwarzen Verbindungen stellen die Stromversorgung der Sensoren dar. Die weißen und grünen Verbindungen dienen der Kommunikation mittels I2C-Bus. Das komplette System wird von der *PowerCore 13000*-Powerbank von Anker mit Strom versorgt und auf einem platzsparenden Regal aus Plexiglas als Stack montiert. Die folgende Abbildung 3.4 zeigt ein Foto des Systems:

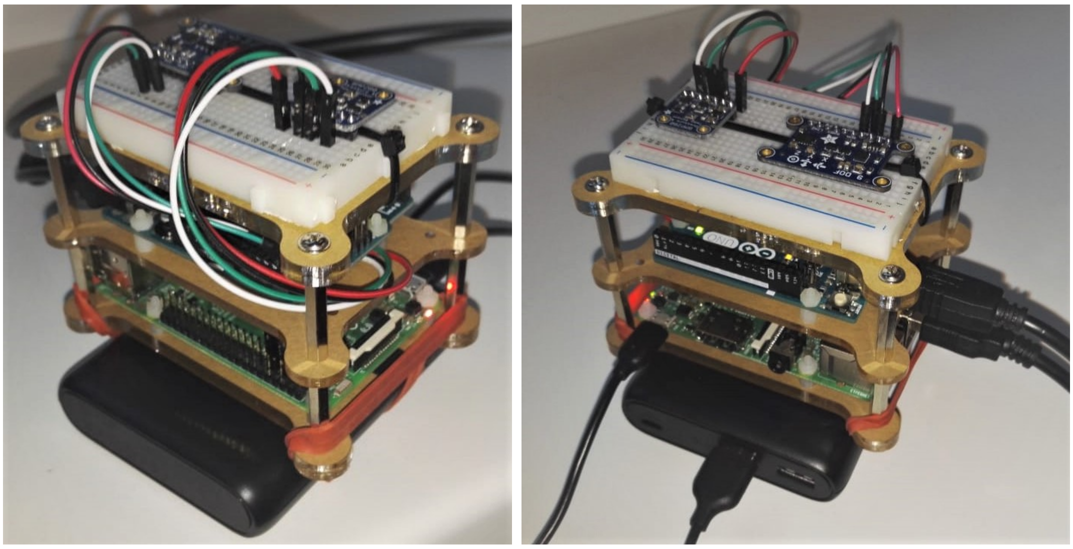


Abbildung 3.4: Foto des fertigen Systems als Stack
Eigene Darstellung

3.3 Umsetzung

In den folgenden Abschnitten wird die Umsetzung des Konzepts zum *Indoor*-Positionierungssystem beschrieben. Die verwendeten Sensoren werden untersucht und fusioniert, um eine möglichst genaue IMU zu erhalten. Des Weiteren werden die verwendeten Einstellungen der Sensoren aufgelistet und erklärt, wie die Rohdaten ausgelesen und weiter verarbeitet werden. Die zu beschreibenden Sensoren sind der Beschleunigungssensor, das Gyroskop, der Kompass und das Barometer. Anschließend wird auf die verwendete

Bluetoothtechnik eingegangen und die Zusammenführung aller Daten und Techniken auf dem Raspberry Pi dargelegt.

3.3.1 Sensoren

Für das System werden mehrere Sensoren verwendet und fusioniert, um die Lage und Bewegung möglichst genau ermitteln zu können. Wie in Kapitel 2.2.3 beschrieben, dient der Beschleunigungssensor neben der Positionsbestimmung auch der Stabilisierung des Gyroskops. Mit den Formeln der Kinematik (2.17) ist es möglich, die Geschwindigkeit und die zurückgelegte Strecke zu integrieren. Durch die Bestimmung des Gravitationsvektors wird die Lage in *pitch* und *roll* ermittelt und mit dem Gyroskop fusioniert. Es gibt mehrere Möglichkeiten der Sensorfusion, zum Beispiel ein Komplementär-Filter oder ein Kalman-Filter. Manche Filter eignen sich besser als andere, sind jedoch aufwendiger zu integrieren.

Die Notwendigkeit eines Filters zeigt Abbildung 2.10. Beschleunigungssensoren sind sehr empfindlich und liefern unter Umständen sehr stark schwankende Messwerte, obwohl sich ein System in Ruhe befindet. Der Unterschied zwischen dem Kalman-Filter und dem Komplementär-Filter wurde unter anderem am Gyroskop getestet. Abbildung 3.5 zeigt den Neigungswinkel des Systems über eine Zeitspanne von zehn Minuten. In gelb wird der Neigungswinkel durch den Winkel des Gravitationsvektors (*Acceleration* Winkel) bestimmt. In grün wird der Neigungswinkel dargestellt, welcher sich durch die Sensorfusion aus dem Beschleunigungssensor und dem Gyroskops mittels Kalman-Filters ergibt. Die rote Linie stellt wie die grüne Linie die fusionierten Werte des Beschleunigungssensors und des Gyroskops dar, jedoch wurde hier ein Komplementär-Filter verwendet.

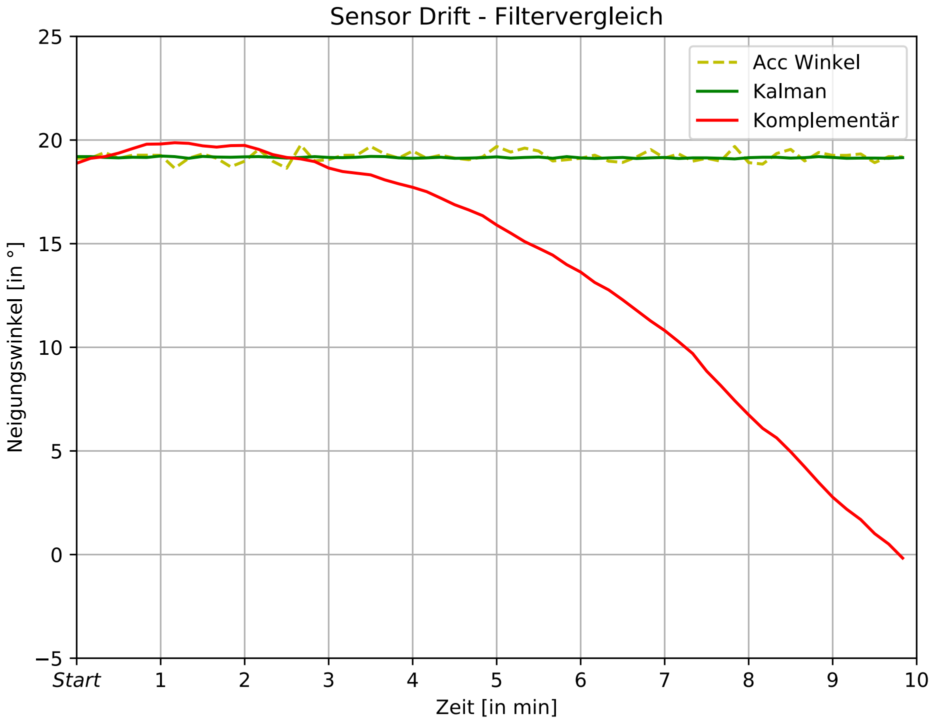


Abbildung 3.5: Vergleich Kalman- und Komplementär-Filter für den Neigungswinkel über Zeit
Eigene Darstellung

Das Komplementär-Filter ist einfacher zu implementieren als das Kalman-Filter, jedoch wird deutlich, dass seine Ergebnisse des Neigungswinkels mit der Zeit stark abdriften. Der Drift liegt bei etwa $0.03°/s$. Es ist möglich, das Komplementär-Filter so anzupassen, dass der Drift verringert wird, jedoch wird dann das gesamte Filter träge und die errechneten Neigungswinkel hängen den wahren Winkeln zeitlich hinterher. Die genau Implementierung des Kalman-Filters wird im Folgenden noch näher beschrieben.

Da für die Sensorfusion neben dem Gyroskop auch ein Beschleunigungssensor verwendet wurde und dieser sehr empfindlich auf Vibrationen reagiert, wurde das Verhalten der Filter ebenfalls unter Vibration untersucht. Abbildung 3.6 zeigt drei Plotts derselben Messung. Hierbei wurde der Sensor auf eine ebene Fläche gelegt und anschließend zwei Mal geneigt. Die erste Neigung findet bei etwa 0.7 bis 1.6 Sekunden statt und wird nicht durch Vibrationen beeinflusst. Die Ausschläge des Winkels, der aus den Beschleunigungsdaten errechnet wird, sind der hohen Empfindlichkeit des Sensors zu zuschreiben. Ab Sekunde 2.4 wird das System durch die Vibrationen eines *Brushless*-Motors beeinflusst. Diese Motoren werden unter anderem für Drohnen verwendet und erzeugen aufgrund ihrer hohen Drehzahl starke Vibrationen im System.

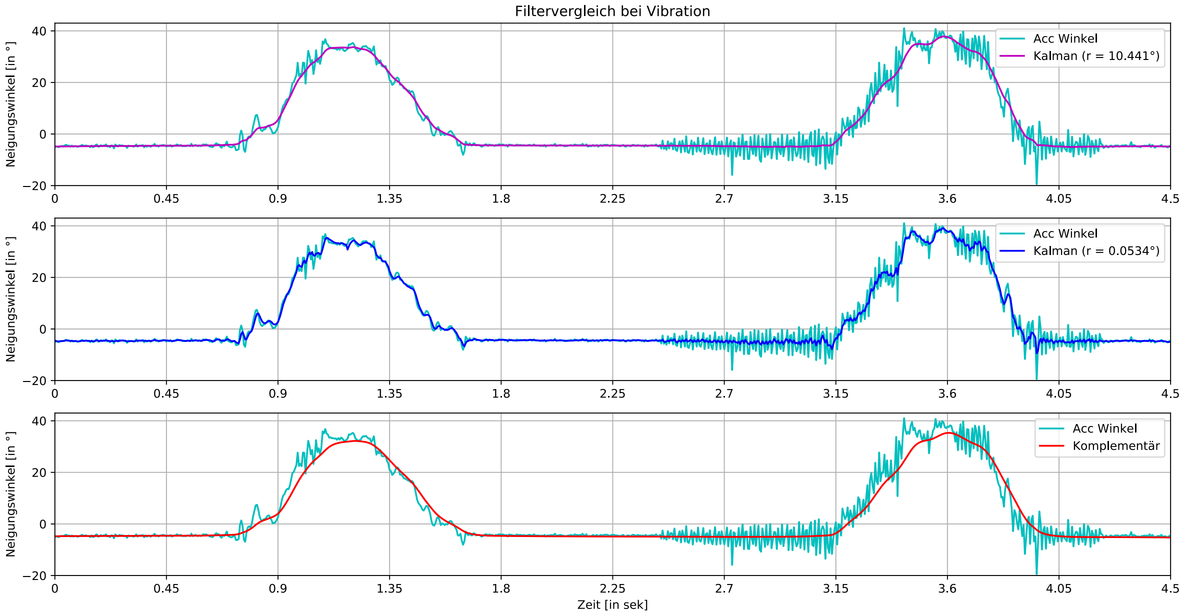


Abbildung 3.6: Vergleich Kalman- und Komplementär-Filter für verschiedene Neigungswinkel bei Vibrationen

Eigene Darstellung

Im Ruhezustand weisen die Messwerte des Neigungswinkels, berechnet aus dem Gravitationsvektor, eine Varianz von $\sigma^2 \approx 0.0534°$ auf. Mit Vibrationen steigt die Varianz auf $\sigma^2 \approx 10.441°$. Für das Kalman-Filter ist es möglich, die Kovarianzmatrix R_k der Unsicherheit der Messwerte entsprechend anzupassen.

Im obersten Plot ist der Neigungswinkel des Gravitationsvektors (cyan) und der Neigungswinkel der Sensorfusion (magenta) für eine Varianz der Messwerte von $\sigma^2 \approx 10.441°$ dargestellt. Im mittleren Plot sind ebenfalls der Neigungswinkel des Gravitationsvektors (cyan) und der Neigungswinkel der Sensorfusion (blau) dargestellt, jedoch wurde hier die Varianz der Messwerte eines ruhenden Systems ohne externe Einflüsse verwendet, $\sigma^2 \approx 0.0534°$. Im untersten Plot wird neben dem Neigungswinkel des Gravitationsvektors (cyan) der Neigungswinkel der Sensorfusion mittels Komplementär-Filter (rot) dargestellt.

Das Komplementär-Filter (unterster Plot, rot) zeigt eine starke Glättung der Messwerte, jedoch reagiert er zeitlich verzögert und hängt dem System hinterher. Der zeitliche Versatz hängt davon ab, wie die Parameter des Filters festgelegt wurden. Aus den Untersuchungen geht hervor, dass ein geringerer zeitlicher Versatz zu einem stärkeren Drift der Messwerte führt und umgekehrt. Das Kalman-Filter (oberster Plot, magenta und mittlerer Plot, blau) schätzt den nächsten Zustand des Systems, wodurch die Ergebnisse nicht zeitlich verzögert sind. Die Glättung der Werte kann durch die Kovarianzmatrix der Unsicherheit der Messwerte angepasst werden (siehe oberster und mittlerer Plot). Aus diesen Gründen wird für die Umsetzung des Konzeptes der Kalman-Filter verwendet. Der genaue Aufbau und die Umsetzung des Filters in *Python* oder *C* wird in den jeweiligen Abschnitten genauer erläutert.

Um die Richtung (Kurs) zu bestimmen, in der sich das System bewegt, wird ein Kompass verwendet. Der Hersteller des *Adafruit 9-DOF IMU Breakout - L3GD20H + LSM303* stellt mehrere Bibliothek zur Bestimmung verschiedener Messgrößen kostenlos bereit, darunter auch mehrere zur Bestimmung des Kurses. Die *Library Adafruit_Sensor* bietet beispielsweise die Möglichkeit, die Richtung des magnetischen Nordpols zu bestimmen. Jedoch wird keine Neigungskompensation vorgenommen, wie im Quellcode ersichtlich. Falls das System, zum Beispiel ein Roboter, durch einen schiefen Untergrund geneigt wird, ändern sich die vom Kompass zurückgegebenen Werte, obwohl keine Änderung des Kurses stattgefunden hat. *Adafruit_9DOF*, ebenfalls eine *Library*, enthält einen Algorithmus zur Bestimmung des Kurses mittels Neigungskompensation, ebenfalls im Quellcode ersichtlich. Um die gemessene magnetische Flussdichte durch die Lage des Systems zu korrigieren, wird in der genannten *Library* der Gravitationsvektor verwendet. Dieser Vektor wurde für das hier entwickelte System bereits für die Sensorfusion des Gyroskops verwendet und zeigt immer in Richtung des Gravitationszentrums. Er ermöglicht es, den Neigungs- und Rollwinkel des ruhenden oder sich geradlinig gleichförmig bewegenden Systems zu bestimmen. Erfährt das System jedoch eine Beschleunigung, unabhängig der Richtung, so wird die Neigungskompensation erheblich gestört. Um dem entgegen zu wirken, wird für das hier entwickelte System die Lage im Raum durch den Neigungs- und Rollwinkel bestimmt, welche zuvor durch die Sensorfusion mittels Kalman-Filter gewonnen wurden. Der Einfluss externer Kräfte wird somit verringert. Zusätzlich wird die Deklination miteinbezogen, um die Genauigkeit des Kurses noch zu steigern. Abbildung 3.7 zeigt die gemessenen Kurswinkel des geneigten Systems:

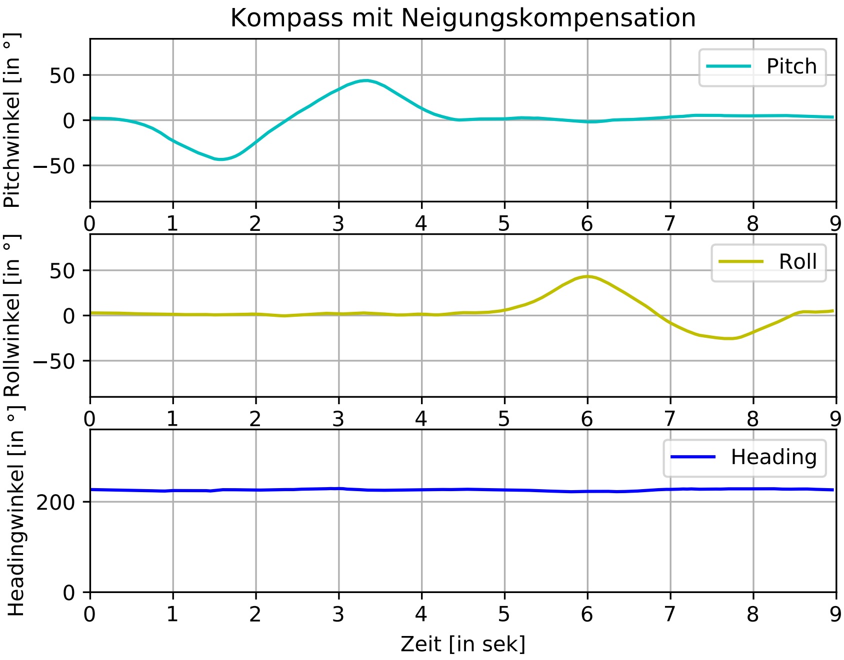


Abbildung 3.7: Kompass mit Neigungskompensation
Eigene Darstellung

Für Abbildung 3.7 wurde das System zunächst geneigt (*pitch*) und anschließend gerollt (*roll*). Dabei wurde darauf geachtet, dass es zu keiner Änderung der Gierachse (*heading*) kam. Der Algorithmus zur Berechnung des Kurswinkels mit Neigungskompensation und die Umsetzung im Programmcode wird im Folgenden genauer erläutert.

Beschleuninungssensor

Wie in Kapitel 2.2.3 bereits beschrieben, wird für dieses Projekt der Beschleunigungssensor *LSM303 DLHC* verwendet. Alle Register, die für die Initialisierung und das Auslesen der Daten notwendig sind, lassen sich aus dem zum Sensor gehörenden Datenblatt entnehmen.

Im Code-Ausschnitt (2) wird zunächst die *Wire.h* Bibliothek von Nicholas Zambetti eingebunden, um eine Kommunikation zwischen dem Arduino und dem Beschleunigungssensor mittels I2C zu ermöglichen. In der Funktion *initialize()* wird der Sensor über das *CTRL_REG1_A* Register ($0x20$ in hexadezimal) initialisiert und die Übertragungsrate der Daten auf 200 Hz ($0x67$) festgelegt. Der Hexadezimalwert $0x67$ ergibt sich aus dem Binärcode $0b01100111$ und wird der Registerbeschreibung des Sensors entnommen. In Zeile 15 des Code-Ausschnitts (2) wird die Verbindung zum Sensor erneut hergestellt, um über das Register *CTRL_REG4_A* ($0x23$) die Einstellung für die Messskala auf $\pm 2\ g$ ($0x00$) festzulegen.

Code-Ausschnitt 2: I2C Kommunikation und Sensoreinstellungen

```
/* Eigene Programmierung, enthält Code-Snippets der Adafruit_LSM303 library
von Kevin Townsend, Adafruit Industries */
#include <Wire.h>
...
void initialize(){
  ...
  // ===== INIT ACCEL =====
  // ======================
  // WAKE UP AND SET DATA RATE CONFIGURATION
  Wire.beginTransmission(LSM303_address);
  Wire.write(0x20);      // CTRL_REG1_A (0x20)
  Wire.write(0x67);      // Data rate configuration
  Wire.endTransmission(true);

  // SET FULL SCALE SELECTION
  Wire.beginTransmission(LSM303_address);
  Wire.write(0x23);      // CTRL_REG4_A (0x23)
  Wire.write(0x00);      // Full-scale selection
  Wire.endTransmission(true);
}
```

Der Code-Ausschnitt (3) definiert die Funktion *getData()*, welche das Register $OUT\_X\_L\_A$ ($0x28$) anspricht, um die Messwerte der Register zu empfangen. Dafür wird in Zeile acht ein *Request* an den Sensor geschickt, um die sechs *Output*-Register auszulesen.

Code-Ausschnitt 3: Auslesen der Beschleunigungswerte

```
 1 /* Eigene Programmierung, enthält Code-Snippets der Adafruit_LSM303 library
 2 von Kevin Townsend, Adafruit Industries */
 3 void getData(){
 4   ...
 5   // ===== DATA ACCEL =====
 6   // ======================
 7   Wire.beginTransmission(LSM303_address);
 8   Wire.write(0x28 | 0x80);
 9   Wire.endTransmission(true);
10   Wire.requestFrom(LSM303_address,6);
11   while(Wire.available()<6);
12   acc_x_raw = (int16_t)((uint16_t)Wire.read()|((uint16_t)Wire.read()<< 8))>> 4;
13   acc_y_raw = (int16_t)((uint16_t)Wire.read()|((uint16_t)Wire.read()<< 8))>> 4;
14   acc_z_raw = (int16_t)((uint16_t)Wire.read()|((uint16_t)Wire.read()<< 8))>> 4;
15 }
```

Als erstes werden die Messwerte der x-Achse (Register $0x28$ und $0x29$), anschließend der y-Achse (Register $0x2A$ und $0x2B$) und zu Letzt der z-Achse (Register $0x2C$ und $0x2D$) übertragen. Dabei beinhaltet das erste Register der jeweiligen Achse das *Low-Byte* und das zweite das *High-Byte*. Die acht Bit des *High-Bytes* werden in einer 16 Bit *unsigned* Variablen vom Typ *Integer* gespeichert und zunächst bitweise um acht Bit nach links verschoben. Mit dem *bitwise inclusive OR* Operator wird nun das *Low-Byte* hinter das verschobene *High-Byte* gehängt. Um sicher zu stellen, dass keine Daten verloren gehen, werden alle Werte zunächst in *unsigned Integer* gespeichert und erst nach der bitweisen Manipulation in 16 Bit *Integer* umgewandelt.

Für weitere Berechnungen ist es wichtig, das *Offset* des Sensor zur bestimmen und herauszurechnen. Dafür ist eine Kalibrierung des Sensors notwendig, andernfalls werden mit jeder Messung höhere oder niedrigere Beschleunigungen ausgegeben. Da die Erdbeschleunigung konstant ist, wird jede Sensorachse mit der Hilfe der Erdbeschleunigung kalibriert. Der Wert von $g = 9.81 \frac{m}{s^2}$ entspricht etwa $981\,Gal$. Mit der Software *Magneto* (v1.2), welche im Abschnitt *Kompass* genauer beschrieben ist, wird eine Korrekturmatrix erzeugt und das *Offset* jeder Achse berechnet. Als *Input* benötigt die Software eine Messreihe, bei welcher der Sensor um alle drei Achsen rotiert werden muss. Es ist dabei sehr wichtig, dass das System keine Erschütterungen erfährt, da dies die Messungen beeinflusst. Zur Kalibrierung wurden zwei Halbkugel aus Polystyrol verwendet, in welchen das System fest verankert wurde. Anschließend wurde die Kugel für mehrere Minuten langsam in alle Richtungen gerollt. Die folgenden drei *Offsets* und folgende Korrekturmatrix sind das Ergebnis der Kalibrierung:

$$\begin{aligned}
accX_{off} &= 32.853954\,Gal \\
accY_{off} &= 11.928418\,Gal \\
accZ_{off} &= -18.129956\,Gal \\
Acc_{correct} &= \begin{pmatrix} 0.966771 & 0.000634 & -0.000244 \\ 0.000634 & 0.938452 & -0.000220 \\ -0.000244 & -0.000220 & 0.960528 \end{pmatrix}
\end{aligned}$$

Abbildung 3.8 zeigt zwei Mal die Messreihe der Kalibrierung. In rot sind die nicht korrigierten Werte in *Gal* dargestellt und in blau die korrigierten Beschleunigungen ebenfalls in *Gal* und ohne *Offset*. Mit der Korrekturmatrix und den bekannten *Offsets* werden nun alle gemessenen Beschleunigungen auf dem Arduino korrigiert. Dies geschieht wie in Code-Ausschnitt (4).

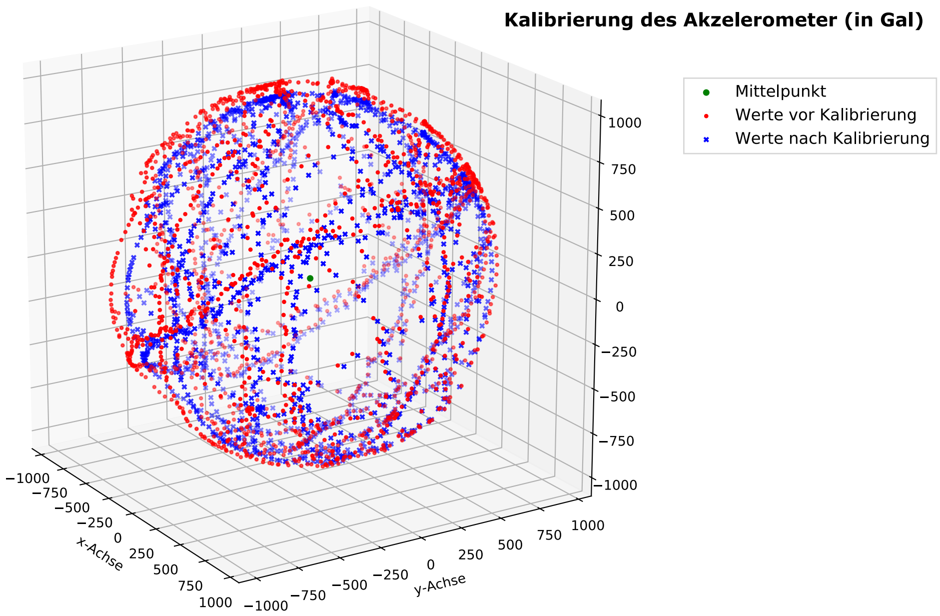


Abbildung 3.8: Messwerte des Beschleunigungssensors vor und nach der Korrektur
Eigene Darstellung

Code-Ausschnitt 4: Umrechnung der Beschleunigungen

```
/* Eigene Programmierung, enthält Code-Snippets von Chris Holm Jul 25, 2018
Abgerufen von https://forums.adafruit.com/viewtopic.php?f=8&t=136357&p=685932)*/
// ===== Acceleration calculations =====
accx_bias = acc_x_raw + 32.853954;
accy_bias = acc_y_raw + 11.928418;
accz_bias = acc_z_raw - 18.129956;

acc_x=( 0.966771 * accx_bias + 0.000634 * accy_bias - 0.000244 * accz_bias)/100;
acc_y=( 0.000634 * accx_bias + 0.938452 * accy_bias - 0.000220 * accz_bias)/100;
acc_z=(-0.000244 * accx_bias - 0.000220 * accy_bias + 0.960528 * accz_bias)/100;
```

Gyroskop

Im vorherigen Abschnitt wurde die Initialisierung, das Auslesen und die Umrechnung der Messwerte des Beschleunigungssensors beschrieben. Mit Hilfe dieser Werte wird der Gravitationsvektor bestimmt, welcher immer in die Richtung des Gravitationszentrums zeigt. Daraus ergeben sich der Neigungs- und Rollwinkel des Systems. Dieses muss sich dazu in Ruhe befinden, da externe Kräfte Beschleunigungen verursachen und somit den Gravitationsvektor verschieben. Ein Beschleunigungssensor kann Winkel sehr genau messen, ist aber sehr störanfällig.

Das Gyroskop $L3GD20$ misst Winkelgeschwindigkeiten $\overrightarrow{\omega_i}$ in *degree per second*. Um mit den Winkelgeschwindigkeiten die Lage des Systems im Raum in Grad zu bestimmen, können diese integriert werden. Da jeder Sensor Messunsicherheiten und Messfehler aufweist, werden diese

Fehler mit integriert, wodurch die gemessenen Winkel driften, auch wenn sich das System in Ruhe befindet.

Die Sensorfusion verbindet die Vorteile beider Sensoren, welche die Messgenauigkeit des Beschleunigungssensors sowie die weniger starke Störanfälligkeit des Gyroskops sind, und vermindert die Nachteile, welche die Störanfälligkeit und der Drift der Sensoren sind. Eine solche Sensorfusion wird durch das Kalman-Filter ermöglicht.

Im Code-Ausschnitt (5) wird der Sensor zunächst initialisiert. Die Initialisierung gleicht der des Beschleunigungssensors, benötigt ebenfalls die *Wire.h* Bibliothek und wird durch die Funktion *initialize()* ausgeführt. Die anzusprechenden Register sind aus dem Datenblatt des Sensors zu entnehmen. Zunächst wird der Sensor über seine *device*-Adresse, dem Register *CTRL_REG1* ($0x20$) und dem Wert $0x00$ aus dem *sleep-mode* in den *normal-mode* versetzt. Anschließend werden über das selbe Register mit dem Wert $0x4F$ die x, y und z Kanäle aktiviert und die *output rate* auf 190 Hz festgelegt.

Das *CTRL_REG4* Register ($0x23$) dient dazu, den Messbereich einzustellen. Mit dem Wert $0x10$ ist der Sensor in der Lage 500 DPS zu messen.

Code-Ausschnitt 5: Initialisierung des Gyroskops

```
/* Eigene Programmierung, enthält Code-Snippets der Adafruit_L3GD20 library
von Kevin Townsend, Adafruit Industries */
  // ===== INIT GYRO =====
  // ====================
  // WAKE UP
  Wire.beginTransmission(L3GD20_address);
  Wire.write(0x20);     // CTRL_REG1 (0x20)
  Wire.write(0x00);     // reset register to normal mode
  Wire.endTransmission(true);
  // SET DATA OUTPUT RATE
  Wire.beginTransmission(L3GD20_address);
  Wire.write(0x20);     // CTRL_REG1 (0x20)
  Wire.write(0x4F);     // enable normal mode, data output rate x,y,z channel
  Wire.endTransmission(true);
  // SET DATA RANGE CONFIGURATION
  Wire.beginTransmission(L3GD20_address);
  Wire.write(0x23);     // CTRL_REG4 (0x23)
  Wire.write(0x10);     // Data range configuration
  Wire.endTransmission(true);
```

Das Auslesen der Daten erfolgt ebenfalls auf die gleiche Weise wie beim Beschleunigungssensor, mit dem Unterschied, dass auf einen *unsigned Integer* mit 16 Bit verzichtet werden kann.

Vor der Umwandlung der übertragenen Messwerte wird das *Offset* des Sensors für jede Achse bestimmt. Im Code-Ausschnitt (6) werden für jede Achse 1000 Messungen addiert und anschließend durch 1000 dividiert.

Code-Ausschnitt 6: Offset-Bestimmung des Gyroskops

```
/* Eigene Programmierung, enthält Code-Snippets von Joop Brokking
abgerufen von http://www.brokking.net/imu.html */
void calibrate(){
  for(cal_i = 0; cal_i < 1000; cal_i++){
    getData();
    gyro_x_offset += gyro_x_raw;
    gyro_y_offset += gyro_y_raw;
    gyro_z_offset += gyro_z_raw;
    delay(500);
    Serial.println(cal_i);
  }
  gyro_x_offset /= 1000;
  gyro_y_offset /= 1000;
  gyro_z_offset /= 1000;
}
```

Diese Mittelwerte werden als *Offset* von jeder weiteren Messung subtrahiert. Tabelle (2.3) beinhaltet Umrechnungsfaktoren, um die vom Sensor übertragenen Bytes in *mDPS* umzurechnen. Daraus ergibt sich folgende Formel zur Berechnung der Winkelgeschwindigkeit in *DPS* (STlife.augmented, 2013a):

$$angle\_gyro\_rate = gyro\_raw \cdot \frac{17.5\,mDPS}{1000}$$

Code-Ausschnitt 7: Winkelgeschwindigkeit in DPS ohne Offset

```
/* Eigene Programmierung, enthält Code-Snippets von Joop Brokking
abgerufen von http://www.brokking.net/imu.html */
// ===== Gyro Rate calculations =====
  gyro_x_raw -= gyro_x_offset;
  gyro_y_raw -= gyro_y_offset;
  gyro_z_raw -= gyro_z_offset;

  roll_gyro_rate   = gyro_x_raw  * (0.0175);
  pitch_gyro_rate  = -gyro_y_raw * (0.0175);
  yaw_gyro_rate    = gyro_y_raw  * (0.0175);
```

Wie erwähnt, wird für die Sensorfusion in diesem System das Kalman-Filter verwendet. In Kapitel 2.3.5 ist das Kalman-Filter in seiner Matrixform dargestellt. Für die Fusion der Sensorwerte ist eine eindimensionale Form jedoch ausreichend (Babb, 2015).

Prediction:

$$\hat{x}_k = a \cdot \hat{x}_{k-1} + bu_k$$

$$p = a \cdot p_{k-1} \cdot a + q$$

Update (messen):

$$\hat{x}'_k = \hat{x}_k + k'(z_k - \hat{x}_k)$$
$$p'_k = (1 - k') \cdot p_k$$

$$k' = p_k \cdot (p_k + r)^{-1}$$

Der aktuelle *pitch*- oder *roll*-Winkel ergibt sich aus dem vorherigen addiert mit dem in der Zeit Δt gedrehten Winkel. Aufgrund der Eindimensionalität des Systems wird die *prediction matrix* F_k durch den Faktor a ersetzt, wobei $a = 1$. Die Steuermatrix B_k ist durch die vergangene Zeit $b = \Delta t$ seit der letzten Berechnung gegeben und der Steuervektor $\overrightarrow{u_k}$ durch die Variable u_k, welche die Winkelgeschwindigkeit ω_k darstellt. $\overrightarrow{z_k}$ ist der aktuelle Messwert des zu berechnenden Winkels. Dieser wird mit Hilfe des Gravitationsvektor $\overrightarrow{g}$ bestimmt (Pedley, 2013):

$$|\overrightarrow{g}| = \sqrt{acc_x^2 + acc_y^2 + acc_z^2}$$
$$\alpha_{pitch} = sin^{-1}(\frac{acc_y}{|\overrightarrow{g}|})$$
$$\alpha_{roll} = sin^{-1}(\frac{acc_x}{|\overrightarrow{g}|})$$

Für die Umsetzung auf den Arduino (Code-Ausschnitt (8)) ist zu beachten, dass die Funktion $asin()$ den Winkel als Radiant zurückgibt. Das Ergebnis muss somit mit $\frac{180}{\pi}$ multipliziert werden.

Code-Ausschnitt 8: Berechnung des Neigungs- und Rollwinkels aus Beschleunigungswerten

```
/* Eigene Programmierung, enthält Code-Snippets von Joop Brokking
abgerufen von http://www.brokking.net/imu.html */
  // ===== Accelerometer angle calculations =====
  vector_acc_total = sqrt(pow(acc_x_raw,2)+pow(acc_y_raw,2)+pow(acc_z_raw,2));
  roll_acc = asin(acc_y_raw / vector_acc_total) * 180 / PI;
  pitch_acc = asin(acc_x_raw / vector_acc_total) * -180 / PI;
```

Als System- und Messunsicherheit werden die Werte $q = 0.02$ und $r = 0.7148$ festgelegt. Im folgenden Code-Ausschnitt (9) wird die Sensorfusion für den *pitch*-Winkel dargestellt. Der *roll*-Winkel ergibt sich auf die gleiche Weise.

Code-Ausschnitt 9: Sensorfusion des Gyroskops

```
/* Eigene Programmierung */
// ========== KALMAN CALCULATION ==========
float kalmanCalculate_pitch(float newAngle, float newRate,int looptime){
    dt = float(looptime) * 0.000001;
    // prediction
    x_hat_pitch = a * x_hat_pitch + dt * newRate;
```

```
    p_pitch = p_pitch + q;
    // update
    k_pitch = p_pitch * (1.0 / (p_pitch + r));
    x_hat_pitch = x_hat_pitch + (k_pitch * (newAngle - x_hat_pitch));
    p_pitch = (1 - k_pitch) * p_pitch;
    return x_hat_pitch;
}
```

Ist die Rate, mit der die Register eines Sensors ausgelesen werden, größer als die für den Sensor eingestellte Updaterate, werden Messwerte doppelt ausgegeben. Um dem entgegen zu wirken, wird die Zeit jedes Schleifendurchlaufs gemessen und anschließend solange gewartet, bis die Register des Sensors mit der langsamsten Updaterate wieder neue Messwerte beinhalten.

Code-Ausschnitt 10: Schleife zum einhalten der Updateraten

```
/* Eigene Programmierung */
  while(micros() - loop_timer < 5300);    // 190 Hz ~ 5300
  time_last_loop = micros() - loop_timer;
  loop_timer = micros();
```

Kompass

Die Hauptaufgabe des Kompass besteht darin, die Richtung der gemessenen Beschleunigungen von einem Bezugssystem in ein anderes zu überführen. Dadurch wird die Bewegungen des Systems vom Sensorkoordinatensystem in ein geografisches Koordinatensystem umgewandelt.

Für die Entwicklung des Kompass wird das Magnetometer *LSM303 DLHC* des *9DOF Breakout Boards* verwendet (siehe Kapitel 2.2.5). Um aus der gemessenen magnetischen Flussdichte in x-, y- und z-Richtung den magnetischen Nordpol zu bestimmen, wird folgende Gleichung verwendet (Kostiainen & Bhaumik, 2018)):

$$orientation = arctan2(-mag_y, mag_x) \tag{3.1}$$

Wird ausschließlich Gleichung (3.1) zur Berechnung der Orientierung verwendet, so ergibt sich ein Problem, wenn das System geneigt ist. Der Sensor misst nun nicht mehr dieselbe magnetische Flussdichte, da das Magnetfeld der Erde konstant bleibt, wenn das System geneigt ist. Mithilfe des Gyroskops lässt sich am Magnetometer eine Neigungskompensation durchführen. Abbildung (3.7) zeigt das Ergebnis der Neigungskompensation des Kompass. Zusätzlich ist es erforderlich, eine Kalibrierung des Sensors vorzunehmen, da dieser sehr durch andere Magnetfelder beeinflusst wird.

Zunächst wird das Magnetometer initialisiert und grundlegende Konfigurationen festgelegt. Dies erfolgt ähnlich wie bei dem Beschleunigungssensor und dem Gyroskop über die I2C-Kommunikation und der *Wire.h* Bibliothek. Als erstes wird mit Hilfe der Sensoradresse ($0x3C >> 1$) eine Verbindung zum Modul aufgebaut und dieses über das *MR_REG_M*-Register ($0x02$) mit dem Wert $0x00$ aus dem *Sleep-mode* in den *Wake-up-mode* überführt. Dargestellt ist dies im Code-Ausschnitt (11).

Nach der Aktivierung wird der Messbereich über das *CRB_REG_M*-Register ($0x01$) auf den Wert $\pm 1.3\,Gauss$ ($0x20$) festgelegt. Die entsprechenden Werte werden dem Datenblatt des Sensors entnommen. Bei einem Messbereich von $\pm 1.3\,Gauss$ gibt das Datenblatt einen Umrechnungsfaktor von $1100\,\frac{LSB}{Gauss}$ für die Umrechnung empfangener Bytes der x- und y-Achse an. Die Werte der z-Achse werden mit dem Faktor $980\,\frac{LSB}{Gauss}$ umgerechnet.

Das Festlegen der Updaterate ist ebenfalls im Code-Ausschnitt (11) dargestellt. Dazu wird an das *CRA_REG_M*-Register ($0x00$) der Wert $0x1C$ übermittelt. Dies entspricht einer Updaterate von $220\,Hz$.

Code-Ausschnitt 11: Initialisierung des Magnetometers

```
/* Eigene Programmierung, enthält Code-Snippets der Adafruit_LSM303 library
von Kevin Townsend, Adafruit Industries */
  // ===== INIT MAG ======
  // ====================
  Wire.beginTransmission(LSM303_mag_address);
  Wire.write(0x02); // Register MR_REG_M
  Wire.write(0x00); // wake up and continuous-conversation mode
  Wire.endTransmission(true);
  // set magnetic gain
  Wire.beginTransmission(LSM303_mag_address);
  Wire.write(0x01); // Register CRB_REG_M
  Wire.write(0x20); // set magnetic gain to +/-1.3 Gauss
  Wire.endTransmission(true);
  // set magnetic update rate
  Wire.beginTransmission(LSM303_mag_address);
  Wire.write(0x00); // Register CRA_REG_M
  Wire.write(0x1C); // set magnetic update rate to 220 Hz
  Wire.endTransmission(true);
```

Das Auslesen der Daten aus den Registern ähnelt dem des Beschleunigungssensors und des Gyroskops und ist im Code-Ausschnitt (12) umgesetzt. Ein wichtiger Unterschied besteht in der Reihenfolge, in der die Register ausgelesen werden und in der Reihenfolge, in der *High-* und *Low-Bytes* übertragen werden. Bei dem Magnetometer wird zuerst das Register der x-Achse ausgelesen und anschließend das der z- und der y-Achse. Außerdem wird im Gegensatz zum Beschleunigungssensor und Gyroskop zuerst das *High-Byte* und danach das *Low-Byte* übertragen.

Code-Ausschnitt 12: Auslesen der Register des Magnetometers

```
/* Eigene Programmierung, enthält Code-Snippets der Adafruit_LSM303 library
von Kevin Townsend, Adafruit Industries */
  // ===== DATA MAG  =====
  // ====================
  Wire.beginTransmission(LSM303_mag_address);
  Wire.write(0x03);
  Wire.endTransmission(true);
  Wire.requestFrom(LSM303_mag_address, 6);
  while(Wire.available() < 6);
  // reading data, high befor low
  mag_x_raw = int16_t(Wire.read()<<8 | Wire.read());
  mag_z_raw = int16_t(Wire.read()<<8 | Wire.read());
  mag_y_raw = int16_t(Wire.read()<<8 | Wire.read());
```

Für die *Hard-* und *Soft-Iron* Kalibrierung, sowie der Bestimmung des *Offset* des Sensors wird die Kalibrierungssoftware *Magneto* (v1.2) von *Sailboat Instruments* verwendet. Diese *Software* ist eine erweiterte Version des *MagCal*, ein *Software tool* der *PLAN Research Group* der Universität von Calgary aus Alberta in Canada. *Magneto* (v1.2) berechnet einen Ellipsoiden, der alle Rohmesswerte des Magnetometers möglichst genau abbildet. Dazu wird der *Least Squares Ellipsoid Specific Fitting*-Algorithmus von Li und Griffiths (2004) verwendet (Merlin, 2011).

Für die Kalibrierung wird der Sensor langsam für mehrere Minuten in alle Richtungen rotiert. Anschließend wird eine Textdatei mit allen Messwerten in x-, y- und z-Richtung in die Software geladen. Zusätzlich wird die vollständige magnetische Flussdichte des Ortes angegeben, an dem die Messungen durchgeführt wurden. Dieser Wert wurde dem *National Oceanic and Atmospheric Administraion - Magnetic Field Calculator* entnommen und beträgt 49663.3 nT.

Als Ergebnis gibt die Kalibrierungssoftware das *Offset* jeder Achse und eine Korrekturmatrix aus, mit der die Messwerte des Ellipsoiden in eine sphärische Form transformiert werden. Diese Werte sind sensor- und systemspezifisch und müssen unter Umständen erneut bestimmt werden, wenn sich am Aufbau des Systems etwas ändert, oder wenn ein neues Magnetometer eingebaut wird. Die von der Kalibrierungssoftware ausgegebenen *Offsets* und die Korrekturmatrix sieht wie folgt aus:

$$
\begin{aligned}
magX_{off} &= 4702.561\, nT \\
magY_{off} &= 3067.99\, nT \\
magZ_{off} &= 13619.51\, nT \\
A_{correct} &= \begin{pmatrix} 0.996058 & 0.008493 & -0.016802 \\ 0.008493 & 1.122115 & 0.001001 \\ -0.016802 & 0.001001 & 0.883519 \end{pmatrix}
\end{aligned}
$$

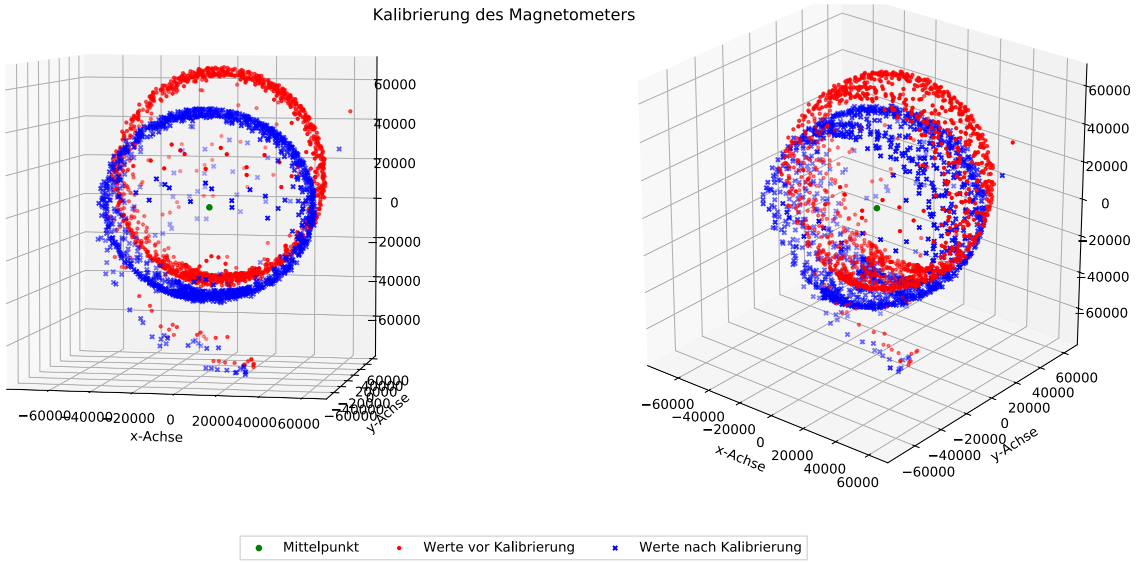


Abbildung 3.9: Messwerte des Magnetometers vor und nach der Kalibrierung
Eigene Darstellung

Abbildung (3.9) zeigt zwei verschiedene Ansichten derselben Messreihe. Der grüne Punkt stellt den Koordinatenursprung und gleichzeitig die Position des Sensors dar. In rot sind die rohen Messwerte (ohne Kalibrierung) dargestellt, welche in positiver z-Richtung verschoben sind (gut zu sehen in der linken Abbildung). Die Verformung entsteht durch den Systemaufbau, welcher in Kapitel 3.2 genauer beschrieben wurde. Der MiKrocontroller, das Raspberry Pi und die Stromversorgung befinden sich beinahe direkt unter dem Magnetometer, wodurch die Messwerte in deren Richtung verzerrt werden und der Ellipsoid entsteht. In der Abbildung (3.9) wird die Verzerrung in Richtung der positiven z-Achse dargestellt, da die positive und negative Seite der z-Achse des Magnetometers vertauscht sind. Die blauen Werte stellen dieselbe Messreihe da, mit dem Unterschied, dass diese mit der Korrekturmatrix korrigiert wurden. In der rechten Abbildung ist gut zu erkennen, wie sich die korrigierten Messwerte in einem kreisförmigen Band um den Mittelpunkt verteilen.

Die Umsetzung auf dem Arduino wird im Code-Ausschnitt (13) beschrieben. Zu Beginn werden die Messwerte des Magnetometers ohne *Offset* in *nano Tesla* berechnet. Anschließend erfolgt die Korrektur der Messwerte mit der vorher bestimmten Korrekturmatrix der Software *Magneto* (v1.2). Bevor die eigentliche Neigungskompensation stattfinden kann, werden die Magnetfeldvektoren normalisiert und die Richtung der y- und z-Achse vertauscht. Anschließend kann die x- und y-Komponente in Abhängigkeit der Lage des Sensors im Raum wie folgt berechnet werden:

$$magX_{tilt} = mag_x \cdot cos(\beta) + mag_z \cdot sin(\beta)$$
$$magY_{tilt} = mag_y \cdot cos(\alpha) - mag_z \cdot sin(\alpha),$$

wobei β den Neigungswinkel (*pitch*) und α den Rollwinkel (*roll*) des Systems darstellt (Pololu, o.J.). *Pitch* und *roll* müssen für die Umsetzung auf dem Arduino als Radianten angegeben werden, weshalb sie mit $\frac{\pi}{180} = 0.001745$ multipliziert werden. Zusätzlich ergeben sich die beiden Lagewinkel des Systems nicht aus dem Beschleunigungssensor, sondern aus der Sensorfusion von Bescheunligungssensor und Gyroskop, da diese Werte weniger störanfällig sind.

Nach der Neigungskompensation wird die Orientierung mit der Gleichung (3.1) bestimmt. Zu der sich ergebenden Orientierung wird die Deklination des Ortes, an dem sich das System befindet, hinzu addiert, da der hier entwickelte Kompass in Richtung des magnetischen Nordpols deutet und nicht zum geografischen. In den letzten Zeilen des Codes wird darauf geachtet, dass sich der berechnete Winkel zwischen 0 und 2π befindet. Anschließend wird der Kurs (*heading*) in Grad umgerechnet.

Code-Ausschnitt 13: Kalibrierung und Neigungskompensation des Magnetfeldsensors

```
/* Eigene Programmierung, enthält Code-Snippets von Chris Holm,
  gepostet am 25. Juli 2018
 (https://forums.adafruit.com/viewtopic.php?f=8&t=136357&p=685932)
 und Ergebnisse der Calibration Software Magneto v1.2
 (https://sites.google.com/site/sailboatinstruments1/home)*/
void calcHeading(){
 // CALIBRATION
 // Offset (Werte in nT)
 mag_x_offset = mag_x_raw*(100000.0 / 1100.0)  -  4702.561101;
 mag_y_offset = mag_y_raw*(100000.0 / 1100.0)  -  3067.987428;
 mag_z_offset = mag_z_raw*(100000.0 /  980.0)  - 13619.506938;
 // Korrekturmatrix
 mag[0] =  0.996058*mag_x_offset + 0.008493*mag_y_offset -0.016802*mag_z_offset;
 mag[1] =  0.008493*mag_x_offset + 1.122115*mag_y_offset +0.001001*mag_z_offset;
 mag[2] = -0.016802*mag_x_offset + 0.001001*mag_y_offset +0.883519*mag_z_offset;
 // Normalisieren der Magnetfeldvektoren
 mag_norm = sqrt(pow(mag[0],2) + pow(mag[1],2) + pow(mag[2],2));
 mag[0] = mag[0] / mag_norm;
```

```
  mag[1] = mag[1] / mag_norm;
  mag[2] = mag[2] / mag_norm;
  // Änderung der y- and z-Achse
  mag[1] = -1.0 * mag[1];
  mag[2] = -1.0 * mag[2];
  // Neigungskompensation
  mag_x = mag[0] * cos(pitch * 0.01745) + mag[2] * sin(pitch * 0.01745);
  mag_y = mag[1] * cos(roll  * 0.01745) - mag[2] * sin(roll  * 0.01745);

  orientation = atan2(-mag_y, mag_x); // tilt orientation
  // Deklination von: https://www.ngdc.noaa.gov/geomag-web/#igrfwmm
  declination = 0.06736;
  orientation += declination;
  if(orientation <0) orientation += 2*PI;
  if(orientation > 2*PI)orientation -= 2*PI;
  // Kurswinkel in Grad
  headingDegrees = orientation * 180/PI;
}
```

Trotz der Kalibrierung ist der Kompass weiterhin störanfällig gegenüber äußeren Magnetfeldern, die durch ferromagnetische Metalle oder stromdurchflossene Leiter entstehen. Eine Möglichkeit diese Störanfälligkeit zu vermindern, ist eine Sensorfusion. Dafür werden im Code-Ausschnitt (14) die *yaw*-Rate (Gier-Rate) und der Kurswinkel mit dem Kalman-Filter fusioniert. Das System zum Berechnen des Kurswinkels lässt sich wie folgt beschreiben:

$$heading_k = heading_{k-1} + \Delta t \cdot yaw_{rate}$$

Der neue Kurswinkel ergibt sich aus der Summe des alten Kurswinkels und der *yaw*-Rate multipliziert mit der Zeitdifferenz (Δt) der letzten Messung. Als Korrekturwert wird der vom Kompass ausgegebene Kurswinkel verwendet. Ein Problem besteht darin, dass der Kurswinkel nur zwischen 0° und 360° liegen kann. Durch die Addition der *yaw*-Rate kann es nun vorkommen, dass ein Winkel größer 360° oder kleiner 0° berechnet wird. Dies wird gelöst, indem eine *if*-Abfrage direkt hinter der ersten Zeile der *Prediction* den Winkel im Falle eines Über- oder Unterschreiten der Grenze korrigiert. Weiterhin kommt es zu erheblichen Schwankungen des Kurswinkels, wenn das System genau auf Norden ausgerichtet ist. Die Messwerte des Kompass schwanken dann um 0° und 360° und das Filter versucht diese Werte zu glätten. Zur Lösung dieses Problems wurde ebenfalls eine *if*-Abfrage verwendet. Dabei wird das Kalman-Filter nur verwendet, wenn der Kurswinkel $> 3°$ oder $< 357°$ ist. Dadurch kann ein Filtern der Werte verhindert werden, wenn das System genau nach Norden ausgerichtet ist.

Code-Ausschnitt 14: Sensorfusion des Kompass und der *yaw*-Rate

```
/* Eigene Programmierung */
float kalmanCalculate_head(float newAngle, float newRate,int looptime){
    if((newAngle <= 3) || (newAngle >= 357)) x_hat_head = newAngle;
    else{
    dt = float(looptime) * 0.000001; // dt = 1/Hz
    // prediction
    x_hat_head = a * x_hat_head + dt * newRate;
    if(x_hat_head > 360) x_hat_head -= 360;
    if(x_hat_head < 0)   x_hat_head += 360;
    p_head = p_head + q_head;
    // update
    k_head = p_head * (1.0 / (p_head + r_head));
    x_hat_head = x_hat_head + (k_head * (newAngle - x_hat_head));
    p_head = (1 - k_head) * p_head;
    }
    return x_hat_head;
}
```

Barometer

Wie in Kapitel 2.2.2 bereits beschrieben, dient das Barometer der Luftdruckmessung. Mit Hilfe der internationalen Höhenformel (2.4) lässt sich die Höhe über dem Meeresspiegel (in Meter) bestimmen. Verwendet wird das $BMP280$ von Adafruit, dem die fertige Bibliothek *Adafruit_BME280* für Arduino und Raspberry Pi mitgeliefert wird. Diese Bibliothek enthält mehrere Methoden zum Auslesen von Sensormesswerten und zur Einstellung des Sensors. Da jedoch nur wenige der Standardfunktionen verwendet werden und die Bibliothek keine Funktion für *Multithreading* enthält, wird eine eigene Klasse implementiert.

Code-Ausschnitt (15) zeigt den Konstruktor der *BMPSensor*-Klasse, in dem zunächst ein Sensorobjekt der *BME280*-Klasse aus der Bibliothek erzeugt wird. Sollte der Sensor nicht funktionieren, oder nicht korrekt angeschlossen sein, wird ein *RuntimeError* ausgelöst und das Programm sofort beendet. Anschließend werden Variablen initiiert, die für weitere Methoden der Klasse notwendig sind.

Code-Ausschnitt 15: Eigene BMPSensor Klasse

```
""" Eigene Programmierung """
from Adafruit_BME280 import *
import time
import threading
import queue
class BMPSensor:
  """ Erzeugt eine BMPklasse zum Messen des Luftdrucks """
  def __init__(self):
      try:
          self.sensor = BME280(t_mode=BME280_OSAMPLE_8, p_mode=BME280_OSAMPLE_8,
          h_mode=BME280_OSAMPLE_8)
      except Exception:
          print("Fehler beim Initiieren des BMPsensors.
          Prüfe Verbindung mit $ i2cdetect -y 1")
          raise RuntimeError
```

```python
        self.height = 0
        self.mean_height = 0
        self.bmp_scanner_on = True

    def calcMeanHeight(self):
        """ Berechnet die mittlere Höhe (Dauert: 5 sek) """
        for i in range (10):
            self.mean_height = self.mean_height + self.getHeightBMP()
            time.sleep(0.5)
        self.mean_height = self.mean_height / 10
        return self.mean_height

    def getHeightBMP(self):
        """ Berechnet die aktuelle Höhe (absolut) """
        self.degrees = self.sensor.read_temperature()
        self.pascals = self.sensor.read_pressure()
        self.hectopascals = self.pascals / 100
        self.height = (288.15/0.0065)*(1-((self.hectopascals/1013.25)**(1/5.255)))
        return self.height
```

Die Methode *calcMeanHeight()* dient dazu, die Höhe des Systems beim Starten zu bestimmen. Da der Luftdruck auch in Gebäuden von der aktuellen Wetterlage beeinflusst wird, kann mit dieser Methode die Ausgangshöhe des Systems gemessen werden. Von allen weiteren Höhenmessungen wird diese Ausgangshöhe subtrahiert, um somit die momentane Höhe im Raum zu bestimmen. Die Messung der aktuellen Höhe geschieht in *getHeightBMP()*. Mit der internationalen Höhenformel (2.4) wird aus dem Druck die Höhe des Systems (in Meter) über Normalnull ausgegeben.

Das $BMP280$ liefert neue Messwerte mit etwa $3Hz$. Um das doppelte Auslesen von Registern zu vermeiden, muss das Programm für etwa 0.3 Sekunden gestoppt werden. Da dies jedoch ebenfalls andere Sensoren und die Berechnungen anderer Prozesse stoppt, wird ein neuer *thread* erzeugt. Eine ausführliche Beschreibung des *Multithreadings* in Python befindet sich in Kapitel 3.3.3.

Code-Ausschnitt (16) beinhaltet drei weitere Methoden der *BMPSensor*-Klasse. Deren Hauptaufgabe besteht darin, die vom Sensor gelesenen Daten in einem neuen *thread* für eine weitere Bearbeitung bereitzustellen.

Code-Ausschnitt 16: Weitere Methoden der *BMPSensor*-Klasse

```python
""" Eigene Programmierung """
def streamHeight(self, bmp_queue):
    while self.bmp_scanner_on:
        try:
            bmp_queue.put(self.getHeightBMP())
        except Exception:
            print("Fehler beim Berechnen der Höhe")
            raise RuntimeError
        else:
            time.sleep(0.3)

def startMeasuringHeight(self, bmp_queue):
    """ Startet neuen Thread und streamt BMP-Daten """
    self.bmp_thread = threading.Thread(target=self.streamHeight,
```

```
    args=(bmp_queue,))
    self.bmp_thread.daemon = True
    self.bmp_thread.start()

def stopMeasuringHeight(self):
    """ Beendet den Thread aus startMeasuringHeight() """
    self.bmp_scanner_on = False
    while self.bmp_thread.isAlive():
        pass
    print("BMP Measuring-thread wurde beendet")
```

Die Methode *startMeasuringHeight()* erzeugt einen neuen *thead* für die vom Sensor ausgegebenen Daten. In dem neuen *thead* wird die Methode *streamHeight()* so lange ausgeführt, bis die dritte Methode *stopMeasuringHeight()* den Wert für *self.bmp_scanner_on* auf *False* setzt. In der Zeit, in der diese Variable den Wert *True* besitzt, übergibt *streamHeight()* die gemessenen Höhenwerte in dem parallel laufenden *thread self.bmp_thread* an eine *queue*. Dabei wird der *thread* nach jedem erfolgreichen Lesen von Höhenwerten für 0.3 Sekunden pausiert, um das doppelte Auslesen von Registern zu vermeiden. Andere Prozesse können ungehindert weiterlaufen. Eine genauere Beschreibung der Kommunikation zwischen *threads* befindet sich in Kapitel 3.3.3.

3.3.2 Bluetooth

Das hier entwickelte System zur *Indoor*-Positionierung verwendet, ähnlich wie in Rida et al. (2015), die RSSI Methode mit BLE Signalen. Mit BLE ist es möglich, Bluetooth-Signale ohne *pairing* zu senden und zu empfangen. Um dem System die Anzahl der *Beacon* im Raum, deren Position und deren MAC (*Media-Access-Control*) Adressen zu übermitteln, wird eine einfache *Client-Server*-Architektur eingesetzt.

Für die Berechnung der Entfernung vom System zu einem beliebigen *Beacon* wird zunächst der Zusammenhang zwischen empfangener Signalstärke und Distanz untersucht. Dafür wird in einem Experiment, ähnlich wie bei Rida et al. (2015), über eine Strecke von 5 Metern Signalstärken gemessen. Alle 20 Zentimeter werden 60 Messungen durchgeführt und aufgezeichnet. Die Messergebnisse sind von Ausreißern und Fehlmessungen zu bereinigen, ehe ein Mittelwert gebildet wird. Die 25 Mittelwerte ergeben einen Datensatz, der anschließend durch eine Regression beschrieben werden kann. Für die Modellierung der Messwerte werden zwei Regressionen durchgeführt. Die erste beruht auf dem *One-Slope-Modell* und wird durch die Gleichung (2.2) beschrieben. Dabei nimmt die Signalstärke logarithmisch mit zunehmender Entfernung ab. Die Gleichung (2.2) kann wie folgt in eine Matrixdarstellung überführt werden:

$$\begin{bmatrix} y_1 \\ y_2 \\ ... \\ y_{n-1} \\ y_n \end{bmatrix} \approx \begin{bmatrix} 1 & log(x_1) \cdot 10 \\ 1 & log(x_2) \cdot 10 \\ ... & ... \\ 1 & log(x_{n-1}) \cdot 10 \\ 1 & log(x_n) \cdot 10 \end{bmatrix} \cdot \begin{bmatrix} P_0 \\ \gamma \end{bmatrix}$$

Wie in Kapitel 2.3.4 stellen die Werte $(y_1, y_2, ..., y_{n-1}, y_n)$ die Zielgröße beziehungsweise die gemessenen Signalstärken dar. Die dazugehörenden Entfernungen (Ausgangsgrößen) werden mit $(x_1, x_2, ..., x_{n-1}, x_n)$ beschrieben. P_0 stellt die Signalstärke bei einem Meter Entfernung dar und γ den Dämpfungskoeffizient mit dem das Signal mit zunehmender Entfernung abnimmt. Zur Berechnung der Konstanten $\omega = (P_0, \gamma)$ werden die empfangenen Signalstärken in der Matrix Y und die logarithmischen Entfernungen in der Matrix X zusammengefasst und wie folgt umgestellt:

$$\begin{aligned} Y &\approx X \cdot \omega \\ \omega &\approx (X^T \cdot X)^{-1} \cdot Y \cdot X^T \end{aligned} \tag{3.2}$$

Mit (2.3.4) ergibt sich für die Signalstärke $P_0 = -57.686\, dBm$ und für die Dämpfung $\gamma = -2.168 \frac{dBm}{m}$.

Für dieselbe Messreihe wird anschließend eine Polynomial Regression durchgeführt. Dafür kann die geeignete Funktion h aus (2.21) zur Berechnung der Zielgröße beziehungsweise der empfangenen Signalstärke aus der Ausgangsgröße (Entfernung) wie folgt beschrieben werden:

$$y \approx c_0 \cdot x^0 + c_1 \cdot x^1 + c_2 \cdot x^2 + c_3 \cdot c^3 \tag{3.3}$$

Um (3.3) in eine Matrixform zu bringen, mit der die Parameter (c_0, c_1, c_2, c_3) bestimmt werden können, werden die Messwerte wie folgt eingesetzt:

$$\begin{bmatrix} y_1 \\ y_2 \\ ... \\ y_{n-1} \\ y_n \end{bmatrix} \approx \begin{bmatrix} x_1^0 & x_1^1 & x_1^2 & x_1^3 \\ x_2^0 & x_2^1 & x_2^2 & x_2^3 \\ ... & ... & ... & ... \\ x_{n-1}^0 & x_{n-1}^1 & x_{n-1}^2 & x_{n-1}^3 \\ x_n^0 & x_n^1 & x_n^2 & x_n^3 \end{bmatrix} \cdot \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

Diese Matrixform kann erneut mit (3.2) vereinfacht und anschließend aufgelöst werden. Für die Parameter (c_0, c_1, c_2, c_3) ergeben sich dann folgende Werte in $\frac{dBm}{m}$:

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} \approx \begin{bmatrix} -33.45 \\ -40.29 \\ 17.861 \\ -2.732 \end{bmatrix}$$

Die Ergebnisse beider Regressionen sind in Abbildung (3.10) dargestellt. Die grünen Kreuze stellen die Messwerte der empfangenen Signalstärken dar. In blau ist das Ergebnis der Polynomial Regression und in rot das Ergebnis der logarithmischen Regression dargestellt. Da der Logarithmus für null nicht definiert ist und die Messungen der Signalstärken in einem Abstand von $0.2\,m$ durchgeführt wurden, liegen keine Werte für das Intervall $[0, 0.2)$ vor. Für den Bereich $0.2 \leq x \leq 3.2$ werden die gemessenen Signalstärken von beiden Regressionen gut modelliert. Bei einer Entfernung von zwei Metern ist die Steigung der Polynomial Regression etwa null, weshalb Signalstärke in diesem Abschnitt nicht eindeutig modelliert sind. Für $x > 3.2$ fällt die Polynomial Regression stark ab und entfernt sich dadurch zunehmend von den realen Werten. Aus diesem Grund und wegen der nicht Eindeutigkeit der Werte im Bereich um $x = 2$, wird für die Modellierung des Zusammenhangs zwischen empfangener Signalstärke und Entfernung die logarithmische Regression verwendet.

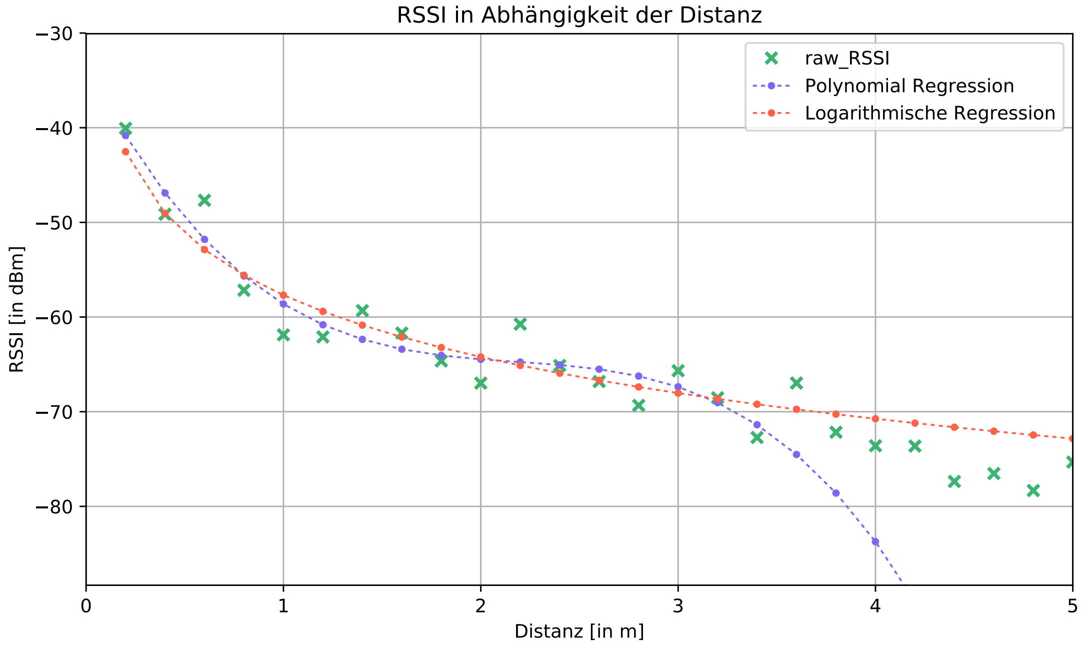


Abbildung 3.10: Vergleich der logarithmischen und Polynomial Regression
eigene Darstellung

Die Auswertung der Messergebnisse zeigte zwei Bereiche von empfangen Signalstärken bei einigen Datensätzen. So wurden zum Beispiel bei einem Abstand von 2.4 m zum *Beacon* Signalstärken zwischen -49 dBm und -52 dBm sowie zwischen -64 dBm und -68 dBm gemessen. Dieser Effekt wird als *Multipath* bezeichnet und tritt dann auf, wenn ein Signal an einem Objekt, wie beispielsweise einer Wand, reflektiert wird. Durch den längeren Weg, den das Signal zurücklegen muss ist die Signalstärke schwächer, was die Modellierung des Zusammenhangs aus Signalstärke und Distanz beeinflusst (Mautz, 2012). Abbildung 3.11 zeigt die Messwerte für eine Entfernung von 2.4 m zur Signalquelle

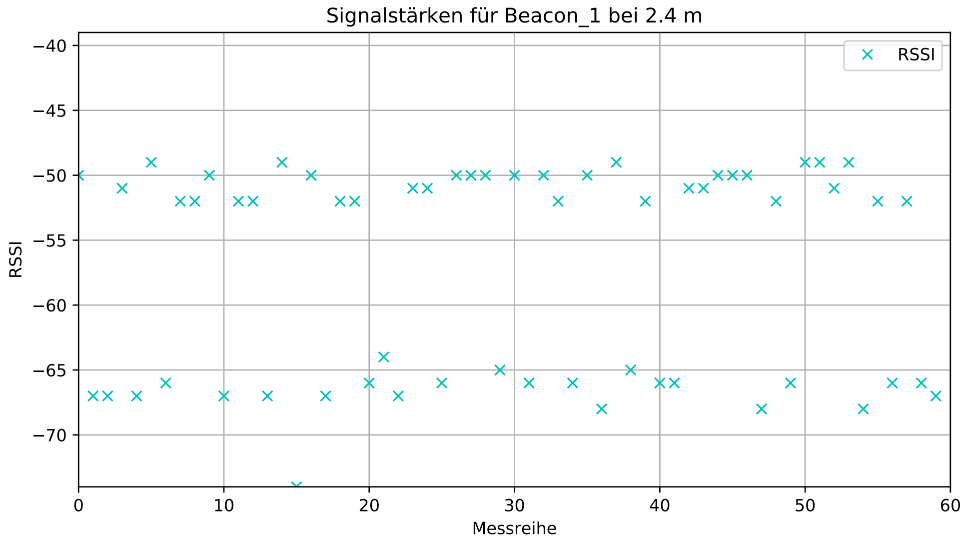


Abbildung 3.11: *Multipath*-Effekt bei gemessenen Signalstärken
Eigene Darstellung

In Python wird die Klasse *Beacon* (Code-Ausschnitt (17)) angelegt, welche die Signalstärke P_0 und die Dämpfung γ für die logarithmische Regression als Klassenattribute beinhaltet. Zusätzlich wird eine *queue* nach dem *Last in, first out* (Lifo) Prinzip zur späteren Kommunikation zwischen *threads*, eine Liste mit registrierten *Beacons* und eine Liste mit *Beacons*, die beim Scannen gefunden wurden, initiiert.

Code-Ausschnitt 17: *Beacon*-Klasse mit Konstruktor

```python
""" Eigene Programmierung, enthält Code-Snippets der beacontools 1.3.0
Project description, abgerufen von https://pypi.org/project/beacontools/ """
import time
import numpy as np
import queue
from beacontools import BeaconScanner, EddystoneFilter, EddystoneURLFrame
from Kalman import RSSIKalman

class Beacon:
  """ Erstellt Beacon-Objekte mit der MAC-Adresse, RSSI und der URL """
  p_0    = -57.686  # RSSI für 1 m, für logarithmische Regresseion
  w      =  -2.168  # Dämpfung des BLE-Signals
  beacon_list = [] # Liste mit bekannten Beacons
  beacons = []     # Liste mit gefundenen Beacons
  ble_queue = queue.LifoQueue()

  def __init__(self,mac_adr=' ',url=' ',rssi=0,pos_lon=.0,pos_lat=.0,name=' '):
      """ Konstruktor """
      self.mac_adr  = mac_adr
      self.url      = url
      self.pos_lon  = pos_lon
      self.pos_lat  = pos_lat
      self.name     = name
      self.distance = 0.0
      self.filter   = RSSIKalman(rssi)
      self.rssi = rssi
      self.rssi_f = rssi
```

Neben dem Konstruktor enthält die *Beacon*-Klasse noch weitere Methoden. Diese dienen dazu einen Scanvorgang zu starten oder zu stoppen, Objekte gefundener *Beacons* oder Attribute der *Beacons* zu speichern.

Zur Kommunikation und Übertragung von Daten zwischen dem System und den *Beacons* wird das im Kapitel 2.2.8 beschriebene *Eddystone-URL* Protokoll verwendet. Dazu werden aus der Bibliothek *beacontools* die Klassen *BeaconScanner, EddysteoneFilter* und *EddystoneURLFrame* importiert (Code-Ausschnitt (17), Zeile 6). Um den Scanvorgang zu starten, muss die in Code-Ausschnitt (18), Zeile 32 definierte Funktion *initBeaconScanner()* ausgeführt werden. Diese Methode erzeugt ein Objekt *scanner* der vorher importierten *BeaconScanner*-Klasse. Bei der Instantiierung des Objekts wird an die Klasse die Methode *callback()* und das Bluetooth-Protokoll, nach dem gescannt werden soll, übergeben. Mit der Methode *startBleScanner()*, im Code-Ausschnitt (18), Zeile 16, und dem Objekt *scanner*, welches die Funktion benötigt, wird der Scannvorgang gestartet. Während des Scannens wird für jeden gefundenen *Beacon* die im Code-Ausschnitt (18), Zeile 3 definierte Methode *callback()* ausgeführt. Diese Methode überprüft, ob der gefundene *Beacon* zu den registrierten *Beacon* der *Beacon.beacon_list*-Liste gehört. Trifft dies zu, wird der gefundene *Beacon* mit den bisher gefundenen *Beacon* der *Beacon.beacons*-Liste verglichen. Gibt es dabei eine Übereinstimmung, so wird die gemessene Signalstärke des vorhandenen *Beacons* aktualisiert. Sollte der gefundenen *Beacon* nicht in der Liste existieren, wird ein neues Objekt der Klasse *Beacon* aus Code-Ausschnitt (17) instantiiert. Dabei wird der Konstruktor der Klasse *Beacon* aktiviert und alle Attribute werden im erzeugten Objekt gespeichert. Mit der Methode *stopBleScanner()* aus Code-Ausschnitt (18), Zeile 25 wird der Scanvorgang gestoppt.

Code-Ausschnitt 18: Scan-Methoden der *Beacon*-Klasse

```
 1 """ Eigene Programmierung, enthält Code-Snippets der beacontools 1.3.0
 2 Project description, abgerufen von https://pypi.org/project/beacontools/ """
 3     def callback(bt_addr, rssi, packet, additional_info):
 4         """ Fügt gefundene Beacon der Liste Beacons hinzu;
 5         übergibt Parameter an Warteschlange """
 6         if bt_addr in Beacon.beacon_list:
 7             unique = True
 8             for beacon in Beacon.beacons:
 9                 if beacon.mac_adr == bt_addr:
10                     unique = False
11                     data = [bt_addr, rssi]
12                     Beacon.ble_queue.put(data) # gibt Daten an Warteschlange
13             if unique == True:
14                 # fügt neuen Beacon der Beaconliste hinzu
15                 Beacon.beacons.append(Beacon(bt_addr, packet.url))
16     def startBleScanner(scanner):
17         """ Startet den Scanvorgang """
18         try:
19             scanner.start()
20         except Exception:
21             print("Start des BLE-Scannvorgangs fehlgeschlagen.")
```

```
        else:
            while len(Beacon.beacons) <= 2:
                pass
    def stopBleScanner(scanner):
        try:
            scanner.stop()
        except Exception:
            print("Stoppen des BLE-Scannvorgangs fehlgeschlagen.")
        else:
            print("BLE-Scannvorgang wurde gestoppt.")
    def initBeaconScanner():
        """ Erstellt neuen EddystoneURL-Scanner """
        try:
            scanner = BeaconScanner(Beacon.callback,
            packet_filter=[EddystoneURLFrame])
        except Exception:
            print("Fehler beim initiieren des BLE Beacon
            Scanners in BLE_Beacon.py")
        else:
            return scanner
```

Jedes Objekt der Klasse *Beacon* enthält das Attribut *self.distance*, welches die Entfernung zwischen dem System und dem ausgewählten *Beacon* enthält. Im Code-Ausschnitt (19), Zeile 2 wird eine Methode definiert, welche aus der gemessenen Signalstärke die Entfernung berechnet. Dazu wird der vorher modellierte logarithmische Zusammenhang zwischen empfangener Signalstärke und Entfernung verwendet. Falls die gemessene Signalstärke größer als $-15\,dBm$ ist, wird eine Distanz von $0.0\,m$ ausgegeben, da andernfalls negative Entfernungen berechnet würden. Dieser Grenzwert entspricht einer Distanz von annähernd null.

Code-Ausschnitt 19: Methoden der *Beacon*-Klasse zur Berechnung der Distanz

```
 1 """ Eigene Programmierung """
 2   def getDistance(self, rssi):
 3       """ Berechnet Distanz (m) aus Signalstärke """
 4       self.rssi = rssi
 5       self.rssi_f = self.filter.filterRSSI(rssi)
 6       if (self.rssi_f > -15):
 7           self.distance = 0.0
 8       else:
 9           self.distance = 10**((self.rssi_f - Beacon.p_0) / (10 * Beacon.w))
10
11   def setPosition(self, pos_lon, pos_lat):
12       """ Legt Position (lon,lat) des Beacons fest """
13       self.pos_lon = pos_lon
14       self.pos_lat = pos_lat
15
16   def setName(self, name):
17       """ Legt den Namen des Beacon fest """
18       self.name = name
19
20   def getRequirement(self):
21       """ Gibt Werte zurück, für Trilateration """
22       return self.pos_lon, self.pos_lat
```

Die Messungen der Signalstärke bei einer Entfernung von zwei Metern in Abbildung (3.12) zeigen, dass die gemessenen Werte zwischen $-45\ dBm$ und $-51\ dBm$ schwanken. Zusätzlich befinden sich im Datensatz Signalstärken von $-60 dBm$ bis $-66 m$, welche auf *Multipath*-Effekte zurück zuführen sind. Der Betrag, um den die Messwerte schwanken, wächst mit zunehmender Distanz.

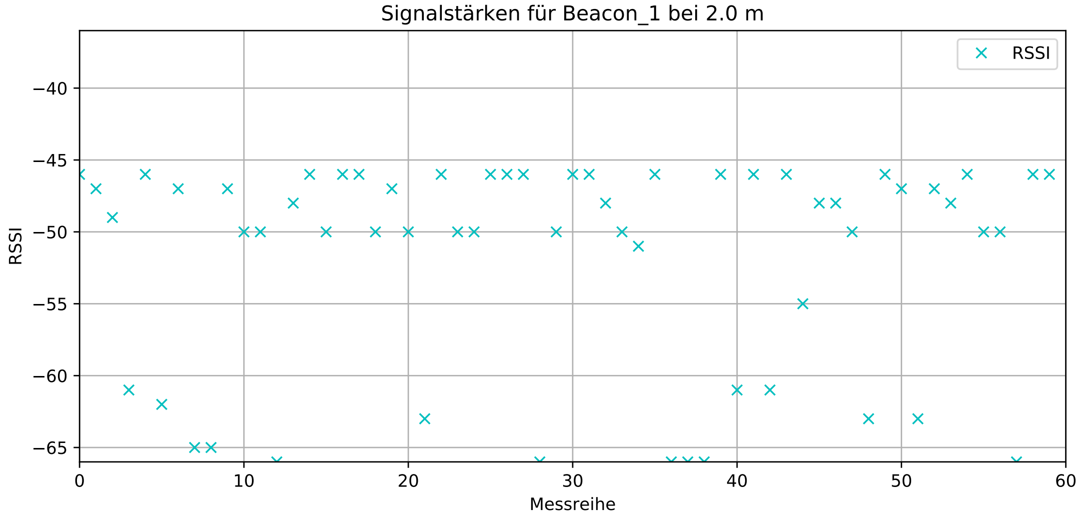


Abbildung 3.12: Messungen der Signalstärke bei 8 Metern Entfernung
Eigene Darstellung

Um die Schwankungen zu verringern, werden die RSSI-Werte vor der eigentlichen Berechnung der Entfernung im Code-Ausschnitt (19), Zeile 5 mit einem eindimensionalen Kalman-Filter geglättet. Jedes Objekt der *Beacon*-Klasse erhält im Konstruktor (Code-Ausschnitt (17), Zeile 25) seinen eigenen Filter.

Der erste Teil des Kalman-Filters (*Prediction*) wird wie folgt beschrieben:

$$\hat{x}_k = a \cdot \hat{x}_{k-1}$$

$$p_k = p_{k-1} + q,$$

wobei $a = 1$ und $q = 0.8$ beträgt. Der Update-Schritt des Filters ergibt sich wie in (2.28). Die gemessene Signalstärke stellt den Messwert $z_k = rssi$ dar, der zur Berechnung der neuen besten Schätzung des Systems notwendig ist. Für die Bestimmung der Kalman-Verstärkung wird eine Messunsicherheit von $r = 8.7$ festgelegt.

Für die Umsetzung in Python wird zur Filterung der RSSI-Werte im Code-Ausschnitt (20) die Klasse *RSSIKalman* definiert. Der Konstruktor legt unter anderem den Startwert von $\hat{x}$ und die System- und Messunsicherheit fest.

Die Klasse enthält nur eine Methode *filterRSSI()*, welche die an sie übergebenen RSSI-Werte glättet. Anschließend werden die neuen RSSI-Werte an das Programm zurückgegeben.

Code-Ausschnitt 20: *RSSIKalman*-Klasse zum Glätten der gemessenen Signalstärken

```
""" Eigene Programmierung """
class RSSIKalman:
    def __init__(self, rssi):
        self.q = 0.8
        self.r = 8.7
        self.a = 1
        self.x_hat = rssi
        self.p = 0
    def filterRSSI(self, rssi):
        # prediction
        self.x_hat = self.a * self.x_hat
        self.p = self.p + self.q
        # update
        self.k = self.p * (1.0 / (self.p + self.r))
        self.x_hat = self.x_hat + (self.k * (rssi - self.x_hat))
        self.p = (1 - self.k) * self.p
        return self.x_hat
```

Server Side

Ein potentielles Anwendungsbeispiel für ein Positionierungsverfahren in Gebäuden ist ein Museum. In ihm können viele *Beacon* mit verschiedenen Aufgaben verwendet werden. Einige haben eventuell die Aufgabe *Audioguides* zu aktivieren, wenn sich ein Besucher in der unmittelbaren Umgebung eines Ausstellungsstücks befindet. Andere dienen dazu, Informationen weiterzugeben oder Messungen der Temperatur durchzuführen, und wieder andere dienen dazu, Besucher im Raum zu orten, damit diese sich im Museum besser navigieren können. Zur besseren Verwaltung aller *Beacon* ist es von Vorteil, wenn diese auf einem Server in einer Datenbank hinterlegt sind. Die Idee für das hier entwickelte System besteht darin, dass das Positionierungssystem einen *Request* an einen Server sendet und eine Liste mit allen *Beacon*, die zur Positionierung gedacht sind, als Antwort erhält. Diese Liste enthält unter anderem die MAC-Adresse und den Namen der *Beacon*.

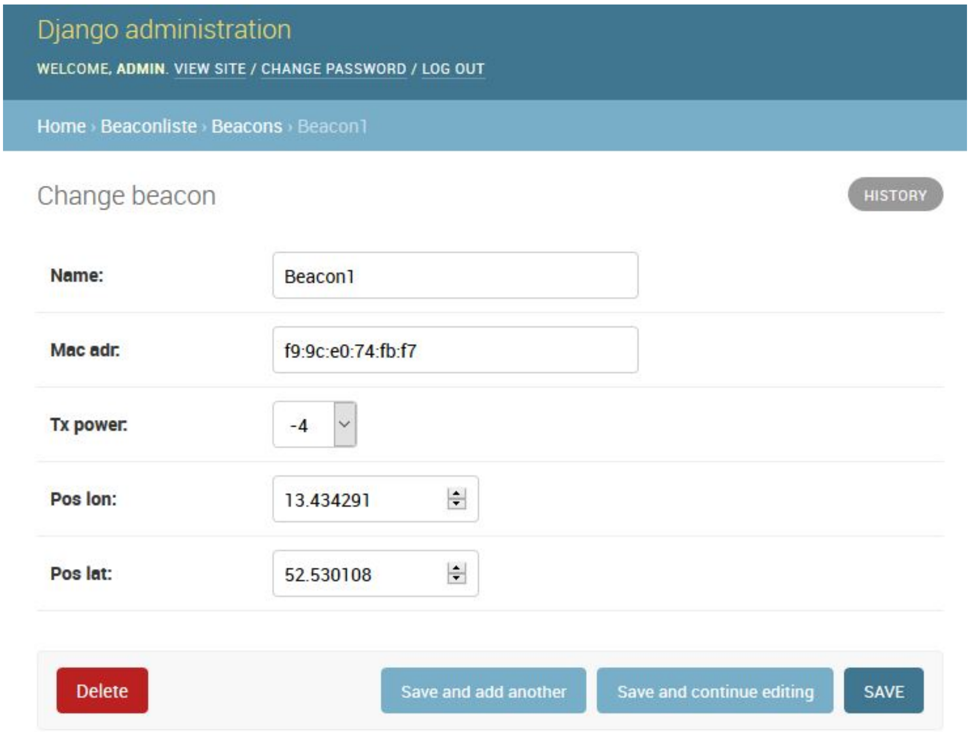
Für die Serverseite wird das quelloffene Python *Webframework Django* (Version 2.0.7) verwendet. In der Datei *models.py* einer neuen App wird die Klasse *Beacon()* erzeugt (Code-Ausschnitt (21)).

Code-Ausschnitt 21: Serverside Django Model

```
""" Eigene Programmierung, enthält Code-Snippets der django Model documentation,
abgerufen von https://docs.djangoproject.com/en/2.1/topics/db/models/ """
from django.db import models
class Beacon(models.Model):
    TX = {
        (-57, '+4'),
        (-59, '0'),
        (-58, '-4'),
```

```
        (-66, ’-8’),
        (-68, ’-12’),
        (-71, ’-16’),
        (-79, ’-20’),
        (-92, ’-30’),
    }
    name = models.CharField(max_length = 50)
    mac_adr = models.CharField(max_length = 50)
    tx_power = models.IntegerField(default = 0, choices=TX)
    pos_lon = models.FloatField()
    pos_lat = models.FloatField()
    def __str__(self):
        return self.name
```

Der vorgefertigte Admin-Bereich des *Django-Frameworks* ist in Abbildung (3.13) dargestellt. Mit der eingeführten Klasse können Objekte manuell erzeugt und in einer Datenbank abgelegt werden.



Change beacon

Name: Beacon1
Mac adr: f9:9c:e0:74:fb:f7
Tx power: -4
Pos lon: 13.434291
Pos lat: 52.530108

Abbildung 3.13: Admin-Bereich zur Eingabe eines Beacon
Eigene Darstellung

Damit der Server auf ein *get-Request* des Positionierungssystems reagieren kann und um neue *Beacon*, zum Beispiel über eine App, hinzuzufügen, ist eine REST (*Representational State Transfer*) API (*Application Programming Interface*) notwendig. Passend zum *Django-Framework* existiert das *djangoRESTframework* (Version 3.8.2), welches die erstellten Modelle (*Beacon()*) serialisiert und im Falle eines *Requests* mit einer Rückgabe von Daten reagiert. Die Daten werden als Antwort auf einen *get-Request* als JSON (*JavaScript Object Notation*) versendet.

Beacon

In Kapitel 2.2.8 wurden die Funktionen von BLE *Beacon* und das verwendete Model (*iBKS 105*) beschrieben. Diese *Beacon* werden mit dem *iBKS Config Tool* des Herstellers konfiguriert. Insgesamt werden drei *Beacon* mit dem *Eddystone URL*-Protokoll zu Positionierung verwendet. Dieses Protokoll erlaubt es, einem *Beacon* in regelmäßigen Abständen eine URL zu senden. Alle Geräte, die sich in Reichweite des Signals befinden, können dieses entschlüsseln und auf die URL zugreifen. Da mit BLE nur wenig Daten übermittelt werden können, darf die URL nicht länger als 17 *characters* sein.

Für das hier entwickelte System sendet jeder *Beacon* eine eigene URL. Das System ist nach dem Empfangen der Daten in der Lage, einen *get-Request* an diese URL zusenden. Die Antwort erhält das System als JSON-Format, in dem Information zu den Koordinaten, der *Transmit-Power* und andere Parameter des *Beacon* enthalten sind.

Client Side

Die *Client Side*, das Raspberry Pi, hat zunächst die Aufgabe einen *get-Request* an einen Server zu senden. Die Antwort erfolgt in Form einer JSON, in der alle *Beacon* zur Positionierung enthalten sind. Anschließend scannt der *Client* nach eben diesen *Beacon*, wobei jedes in regelmäßigen Abständen eine URL sendet. Mit dieser URL ist es möglich, nähere Informationen, zum Beispiel die Koordinaten des Senders, zu erhalten.

Für Python existiert die Bibliothek *requests*, welche den Umgang mit http-Funktionalitäten sehr vereinfacht. Im Code-Ausschnitt (22) wird zunächst die Klasse *Request* erstellt. Bei der Instanziierung muss eine URL angegeben werden. An diese URL sendet der Konstruktor direkt einen *Request* und sichert die Antwort im Attribut *self.res_json*. Die Methode *getBeaconList()* erzeugt aus der *self.res_json* eine Liste mit allen registrierten *Beacon* und deren MAC-Adressen. Anschließend gibt die Funktion diese an das Programm zurück.

Code-Ausschnitt 22: Requests für Beacon-Liste

```python
""" Eigene Programmierung, enthält Code-Snippets von Requests: HTTP for Humans,
abgerufen von http://docs.python-requests.org/en/master/ """
import requests
class Request:
    """ Erzeugt ein Request-Objekt welches
    die JSON-Daten der gewünschten URL enthält """
    def __init__(self, url=''):
        self.url = url
        try:
            self.response = requests.get(url)
        except Exception:
            print("Beacon-Server (%s) ist nicht erreichbar." % (self.url))
            self.res_json = []
        else:
            if self.response.status_code == requests.codes.ok:
                self.res_json = self.response.json()
```

```python
      else:
        self.res_json = []
    self.tag_data = ''
    self.beacon_list = []
  def getBeaconList(self):
    """ Erzeugt eine Liste mit registrierten Beacon MAC-Adressen """
    for item in self.res_json:
      self.beacon_list.append(item['mac_adr'])
    return self.beacon_list
```

Nach dem ein *Beacon* durch den Scanner erkannt wurde, und dieser seine URL an das System übertragen hat, wird mit einer neuen Instanziierung der Klasse *Requet* ein *get-Request* an die empfangene URL gesendet. Die Antwort wird ebenfalls in *self.res_json* gespeichert. Mit *getBeaconData()* (Code-Ausschnitt (23)) ist es nun möglich, gezielt Informationen eines registrierten *Beacon* abzufragen. Dazu muss der Methode der gewünschte *tag* mitgegeben werden. Beispielsweise kann mit *self.getBeaconData('pos_lat')* die latitudinale Koordinate des *Beacon* abgefragt werden.

Code-Ausschnitt 23: Request an Beacon-URL

```python
""" Eigene Programmierung, enthält Code-Snippets von Requests: HTTP for Humans,
abgerufen von http://docs.python-requests.org/en/master/ """
 def getBeaconData(self, tag=''):
   """ Gibt Daten einer Beacon-URL wieder """
   if tag == 'mac_adr':
     for item in self.res_json:
       if item == tag:
         self.tag_data = self.res_json['mac_adr']
     return self.tag_data
   elif tag == 'name':
     for item in self.res_json:
       if item == tag:
         self.tag_data = self.res_json['name']
     return self.tag_data
   elif tag == 'pos_lon':
     for item in self.res_json:
       if item == tag:
         self.tag_data = self.res_json['pos_lon']
     return self.tag_data
   elif tag == 'pos_lat':
     for item in self.res_json:
       if item == tag:
         self.tag_data = self.res_json['pos_lat']
     return self.tag_data
   elif tag == 'tx_power':
     for item in self.res_json:
       if item == tag:
         self.tag_data = self.res_json['tx_power']
     return self.tag_data
   else:
     print("Falscher \'tag\'. Versuche \'mac_adr\',
            \'name\', \'pos_lon\',\'pos_lat\', \'tx_power\'")
```

3.3.3 Zusammenführung

Für das *Indoor*-Positionierungssystem spielt das Raspberry Pi eine wichtige Rolle, denn auf ihm laufen alle Sensormessungen zusammen und werden anschließend verarbeitet. Ein Problem stellten die unterschiedlichen Updateraten der verschiedenen Sensoren dar. Der Beschleunigungssensor, das Gyroskop und der Magnetfeldsensor liefern nach ihrer Konfiguration mit etwa 190 Hz neue Messwerte. Das Barometer erreicht eine Updaterate von 3 Hz und das Bluetooth-Modul liefert etwa einmal pro Sekunde (1 Hz) neue Messwerte, was von der Konfiguration der BLE *Beacon* abhängt. Werden die Register eines Sensors mit einer höheren Frequenz abgelesen als mit der eingestellten Updaterate, so werden dieselben Messwerte erneut ausgegeben. Dies wiederholt sich so oft, bis neue Messwerte vom Sensor gemessen und in den entsprechenden Registern abgelegt wurden. Das Ergebnis einer Messung, bei der die Register zu häufig ausgelesen wurden, ist in Abbildung 3.14 dargestellt:

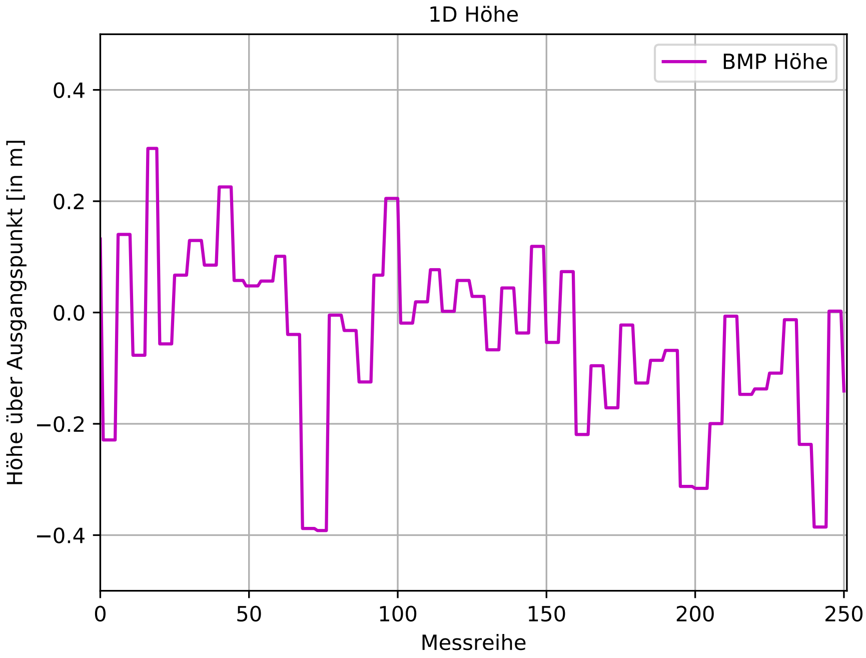


Abbildung 3.14: Beispiel einer Messung mit zu hoher Ausleserate eines Registers
Eigene Darstellung

Bei zu hoher Ausleserate entstehen Stufen, da Register mehrmals dieselben Messwerte wiedergeben. Dies hat auch Einfluss auf die Filterung der Daten. Ein Lösungsansatz ist es, das Programm solange zu pausieren, bis die Register mit neuen Messwerten gefüllt sind. Allerdings liefern andere Sensoren in dieser Zeit weitere Messwerte, welche somit verloren gehen würden. Ein weiterer Lösungsansatz, welcher auch in diesem Projekt verwendet wird, ist das *Multithreading*.

Ein *thread*, zu deutsch Faden oder Faser, bezeichnet eine Aufgabe oder eine Funktion, die zur selben Zeit bearbeitet oder ausgeführt werden soll. *Threading* erlaubt es Python in der Zeit, in der ein anderer Prozess warten muss, Code auszuführen. Dies macht das Programm nicht zwingend schneller. Falls die CPU (*Central Processing Unit*) bereits zu 100 % ausgelastet ist, gibt es keine weiteren Kapazitäten und die *threads* werden nacheinander ausgeführt (Wikibooks, 2018).

Sensordatenübertragung

Für das Empfangen der Sensordaten des Arduino wird auf dem Raspberry Pi mit Python eine Klasse *Sensor()* erstellt. Bei der Instanziierung eines Objekts der Klasse versucht der Konstruktor eine serielle Verbindung zum Arduino herzustellen. Baudrate und *port* sind per *default* auf 115200 und *'/dev/ttyACM0'* festgelegt, können bei der Instanziierung eines Objektes jedoch angepasst werden. Zu sehen ist dies im folgenden Code-Ausschnitt (24).

Code-Ausschnitt 24: Empfangene Sensordaten auf dem Raspberry Pi

```python
""" Eigene Programmierung, enthält Code-Snippets pyserial short introduction,
abgerufen von https://pyserial.readthedocs.io/en/latest/shortintro.html """
 class Sensor():
  """ Erzeugt eine Sensorklasse um Daten in einem neuen Thread zu lesen """
  def __init__(self, port = '/dev/ttyACM0', baudrate = 115200):
    """ Initiiert den Sensor und öffnet eine Serielleverbindung """
    self.port = port
    self.baudrate = baudrate
    try:
      self.ser = serial.Serial(self.port, self.baudrate)
    except Exception:
      print("Fehler: Serielleverbindung ohne Erfolg
      (port: %s, baudrate: %s)" % (self.port, self.baudrate))
      raise RuntimeError
    else:
      print("Serielleverbindung erfolgreich hergestellt.")
      self.scanner_on = True
      self.ser.flushInput()
```

Die *Sensor()* Klasse beinhaltet neben dem Konstruktor noch vier weitere Methoden, welche in Code-Ausschnitt (25) zu sehen sind. *startReadingData()* erzeugt einen neuen *thread*, welcher parallel zum *Mainthread* arbeitet. Dieser führt die Methode *readData()* aus und übergibt dazu eine *queue* , welcher zur Kommunikation zwischen dem neuen *thread* und dem *Mainthread* dient. Weiterhin liest die Methode *readData()* Daten, die über die serielle Verbindung übertragen werden, ein, decodiert diese und teilt den empfangenen *String* an jedem Komma. Die Daten werden an die *queue* übergeben und können parallel im *Mainthread* verarbeitet werden. Um das Lesen der Daten zu stoppen, wird die Methode *stopReadingData()* verwendet. Am Endes eines Programms, oder bei Fehlern, die zu Programmabstürzen führen, ist es wichtig die serielle Verbindung mit der letzten Methode *stopConnection()* zu trennen.

Code-Ausschnitt 25: Methoden der Klasse *Sensor()*

```python
""" Eigene Programmierung, enthält Code-Snippets von Python Advanced Course
Topics, abgerufen von https://www.python-course.eu/threads.php """
  def readData(self,my_queue):
      """ Liest hereinkommenden Datenstring und decodiert
      diesen und übergibt sie an ein Queue """
      while self.scanner_on:
          while self.ser.inWaiting() == 0:
              pass
          try:
              self.dataString = self.ser.readline()[:-2].decode()
          except Exception:
              print("Fehler: ser.readline()[:-2].decode")
          try:
              my_queue.put(self.dataString.split(’,’))
          except Exception:
              print("Fehler: dataString.split()")
  def startReadingData(self, my_queue):
      """ Startet einen neuen Thread um die Daten vom Arduino zu streamen """
      self.newthread = threading.Thread(target=self.readData, args=(my_queue,))
      self.newthread.daemon = True
      self.newthread.start()
  def stopReadingData(self):
      """ Beendet den Thread aus startReadingData() """
      self.scanner_on = False
  def stopConnection(self):
      """  Beendet den Thread aus startReadingData()
      und trennt serielle Verbindung """
      self.scanner_on = False
      while self.newthread.isAlive():
          pass
      if not self.newthread.isAlive():
          print("Thread beendet")
      self.ser.close()
      print("Serielleverbindung getrennt.")
```

Die vom Arduino übertragenen Sensorwerte für die Beschleunigungen in x-, y- und z-Richtung, die Lage des Systems im Raum (*pitch* und *roll*) und die Orientierung (*heading*) werden somit in einem neuen *thread* gesammelt. Für weitere Prozesse und Berechnungen werden diese Daten über eine *queue* an den Hauptprozess übergeben.

Sensordatenfusion zur Berechnung der Höhe

Eine Komponente der dreidimensionale Position des Systems ist die Höhe. Um diese zu bestimmen wird im Folgenden eine Sensordatenfusion aus den Beschleunigungswerten in z-Richtung und den Höhenwerten des Barometers durchgeführt. Dafür wird das Kalman-Filter angewandt. Ziel ist es von den Vorteilen beider Sensoren zu profitieren und die Nachteile zu minimieren.

Messungen mit dem Barometer $BMP280$ in einem geschlossenen Raum zeigten, dass die aus dem Luftdruck errechnete Höhe um knapp einen Meter schwankt. Dadurch ist das Messen von kleinen Höhenänderungen nur schwer möglich. Über längere Zeit bleibt die durchschnittliche Höhe jedoch gleich.

Die mit dem Beschleunigungssensor $LSM303\ DLHC$ gemessenen Beschleunigungen schwanken ebenfalls. Für die Bestimmung des zurückgelegten Wegs mit Hilfe des Beschleunigungssensors werden dessen Werte zunächst über die Zeit dt integriert. Daraus ergibt sich die Geschwindigkeit des Systems (2.19). Eine zweite Integration ergibt den im Zeitintervall dt zurück gelegten Weg (2.20). Durch die doppelte Integration der Beschleunigungswerte werden auch die Messfehler und Messunsicherheiten integriert, wodurch ein *driften* der Position entsteht. Über kurze Zeit können Bewegungen des Systems zuverlässig bestimmt werden. Mit zunehmender Messdauer wächst jedoch auch der Fehler der bestimmten Position.

Für die Berechnung der Höhe aus Beschleunigungs- und Höhenwerten werden die Formeln aus (2.27) zur Schätzung des Systems und zur Berechnung der Kovarianzmatrix verwendet. Somit ergibt sich für die Schätzung des Systems $\hat{x}_k$ folgendes:

$$\hat{x}_k = \begin{pmatrix} pos_z \\ v_z \end{pmatrix}_k = \begin{pmatrix} 1 & dt \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} pos_z \\ v_z \end{pmatrix}_{k-1} + \begin{pmatrix} 0.5 \cdot dt^2 & 0 \\ 0 & dt \end{pmatrix} \cdot \begin{pmatrix} a_z \\ a_z \end{pmatrix}_{tilt}$$

Von den gemessenen Beschleunigungswerten a_z des Eingangsvektors u_k wird der Betrag des Erdbeschleunigungsvektors $\mid \overrightarrow{g} \mid = 9.81\,\frac{m}{s^2}$ subtrahiert. Dieser erhält zuvor durch den *pitch*- und *roll*-Winkel eine Neigungskompensation. Somit ergibt sich eine Beschleunigung in z-Richtung ohne den Einfluss der Gravitation:

$$a_{z,tilt} = a_{z,raw} - cos(pitch) \cdot cos(roll) \cdot 9.81\,\frac{m}{s^2}$$

Für die Kovarianzmatrix P_k wird zunächst die Unsicherheit des Systems in der Kovarinazmatrix Q beschrieben. Auf der Diagonalen dieser Matrix befinden sich die Varianzen der Position in z-Richtung $Var(pos_z)$ und die Geschwindigkeit in z-Richtung $Var(v_z)$. Auf den beiden Seiten der Diagonalen liegen die Kovarianzen $Cov(pos_z, v_z)$ und $Cov(v_z, pos_z)$, welche mit Null festgelegt werden. Die Standardabweichungen σ_{pos} und σ_v hängen von der Standardabweichung der Messwerte des Beschleunigungssensors σ_a ab. Somit ergibt sich für Q und P_k folgendes:

$$\sigma_a = 0.7$$
$$\sigma_v = \sigma_a \cdot dt$$
$$\sigma_{pos} = \frac{1}{2} \cdot \sigma_a \cdot dt^2$$

$$Q = \begin{pmatrix} \sigma_{pos}^2 & 0 \\ 0 & \sigma_v^2 \end{pmatrix} \tag{3.4}$$

$$P_k = \begin{pmatrix} 1 & dt \\ 0 & 1 \end{pmatrix} \cdot P_{k-1} \cdot \begin{pmatrix} 1 & dt \\ 0 & 1 \end{pmatrix}^T + \begin{pmatrix} \sigma_{pos}^2 & 0 \\ 0 & \sigma_v^2 \end{pmatrix}$$

Für den Update-Schritt des Kalman-Filters wird zunächst die Kalman-Verstärkung (2.29) bestimmt. Da keine Anpassung der Skalen zwischen den gemessenen Werten und den gesuchten Werten besteht, wird H_k als Einheitsmatrix beschrieben. Die Unsicherheit der Messwerte R_k ergibt sich aus σ_{bmp}. Da das BMP 280 mit 3 Hz eine geringere Updaterate besitzt als der Beschleunigungssensor und das Gyroskop, wird die Unsicherheit der Messwerte in Abhängigkeit der Zeit eines Schleifendurchlaufs gesetzt. Dadurch wird erreicht, dass nur alle 0.3 Sekunden neue Messwerte des Barometers Einfluss auf den Filter haben.

$$\sigma_{bmp} = \begin{cases} 1000.0, & \text{für } looptime < 0.3\, sek \\ 1.39, & \text{für } looptime \geq 0.3\, sek \end{cases}$$

Die Kalman-Verstärkung zur Berechnung der Höhe ergibt sich nach (2.29), wobei R_k und H_k folgendermaßen beschrieben werden:

$$R_k = \begin{pmatrix} \sigma_{bmp}^2 & 0 \\ 0 & \sigma_{bmp}^2 \end{pmatrix} \qquad H_k = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Die neue beste Schätzung des Systems $\hat{x}'_k$ und die neue Kovarianzmatrix P'_k ergeben sich nach (2.28). Der dafür notwendige Vektor mit den Messwerten z_k besitzt im ersten Eintrag den gemessenen Höhenwert des Barometers und im zweiten Eintrag den Wert Null, da kein Sensor zur Messung der Geschwindigkeit im System integriert ist.

Für die Umsetzung in Python wird eine neue Klasse *HeightKalman* angelegt, welche im Konstruktor zunächst alle wichtigen Matrizen, Vektoren und Parameter festlegt (Code-Ausschnitt (26))

Code-Ausschnitt 26: *HeightKalman*-Klasse zur Sensorfusion und Berechnung der Höhe

```
""" Eigene Programmierung """
class HeightKalman:
    def __init__(self):
        self.dt = 0.05
        self.acc_z_tilt = 0
        self.x_hat = np.matrix([[0],
                                [0]])
        self.P = np.matrix([[0, 0],
                            [0, 0]])
        self.F = np.matrix([[1, self.dt],
                            [0, 1]])
        self.B = np.matrix([[0.5*self.dt**2, 0],
                            [0, self.dt]])
        self.u = np.matrix([[self.acc_z_tilt],
                            [self.acc_z_tilt]])
        # Varianzen berechnen
        self.sigma_acc = 0.7
        self.sigma_v   = self.sigma_acc * self.dt
        self.sigma_pos = (self.sigma_acc * self.dt**2)/2
        self.sigma_bmp = 1.39
        # Matrix Q für Systemrauschen
        self.Q = np.matrix([[self.sigma_pos**2, 0],
                            [0, self.sigma_v**2]])
        # Matrix H für Anpassung der Skalen
        self.H = np.matrix([[1, 0],
                            [0, 1]])
        # Matrix R für Messrauschen
        self.R = np.matrix([[self.sigma_bmp**2, 0],
                            [0, self.sigma_bmp**2]])
        # Messvektor z
        self.z = np.matrix([[0],
                            [0]])
```

Mit der Methode *self.calcHeight()* wird die Sensordatenfusion durchgeführt und die Höhe des Systems berechnet. Dazu müssen der Methode die folgenden Werte übergeben werden: $a_{z,raw}$, *pitch*, *roll*, $height_{bmp}$, *looptime*, *dt*. Die *looptime* dient dazu, doppeltes Auslesen der Register des Barometers zu vermeiden. Sie misst die Zeit, die vergangen ist, seitdem die Register des *BMP 280* das letzte Mal ausgelesen wurden. Das Zeitintervall *dt* misst die vergangene Zeit zwischen den übertragenden Beschleunigungswerten, um aus diesen die Geschwindigkeit und Position des Systems zu integrieren. Dargestellt ist dies in Code-Ausschnitt (27). Die Methoden *setR() und setDt()* berechnen alle Matrizen und Werte neu, die von *dt* abhängig sind. *tiltAcc()* und *setU()* führen die Neigungs- und Gravitationskompensation der Beschleunigungswerte durch.

Code-Ausschnitt 27: Methoden der*HeightKalman*-Klasse zur Berechnung der Höhe

```
""" Eigene Programmierung """
  def tiltAcc(self, acc_z, pitch, roll):
      """ Neigungskompensation für z-Beschleunigung """
      self.acc_z_tilt = acc_z - np.cos(pitch * np.pi /180) * np.cos(roll *
      np.pi / 180) * 9.81  -0.3242
```

```
    def setR(self):
        """ Aktualisiert Messrausch-Matrix R """
        self.R = np.matrix([[self.sigma_bmp**2, 0],
                            [0, self.sigma_bmp**2]])
    def setU(self):
        """ Aktualisiert Messvektor u """
        self.u = np.matrix([[self.acc_z_tilt],
                            [self.acc_z_tilt]])
    def setDt(self):
        """ Aktualisiert alle von dt abhängingen Parameter """
        self.F = np.matrix([[1, self.dt],
                            [0, 1]])
        self.B = np.matrix([[0.5*self.dt**2, 0],
                            [0, self.dt]])
        self.sigma_v   = self.sigma_acc * self.dt
        self.sigma_pos = (self.sigma_acc * self.dt**2)/2
        self.Q = np.matrix([[self.sigma_pos**2, self.sigma_pos * self.sigma_v],
                            [0, self.sigma_v**2]])
    def calcHeight(self, acc_z, pitch, roll, bmp_height, looptime,dt = 0.05):
        """ Kalman-Filter Berechnung """
        self.tiltAcc(acc_z, pitch, roll) # Neigungskompensation
        self.dt = dt
        self.setU()  # aktualisiert den Messvektor
        self.setDt() # aktuailisiert alle von dt abhängigne Parameter
        if looptime < 0.3:
            self.sigma_bmp = 10000
            self.loop_reset = False
        else:
            self.sigma_bmp = 1.39
            self.loop_reset = True
        self.setR() # aktualisiert Messrausch-Matrix
        self.z = np.matrix([[bmp_height], # festlegen des Messvektors
                            [0]])
        # PREDICTION
        self.x_hat = self.F * self.x_hat + self.B * self.u
        self.P = self.F * self.P * np.transpose(self.F) + self.Q
        # UPDATE
        self.S = inv(self.H * self.P * np.transpose(self.H) + self.R)
        self.K = self.P * np.transpose(self.H) * self.S
        self.x_hat = self.x_hat + self.K * (self.z - self.H * self.x_hat)
        self.P = self.P - self.K * self.H * self.P
        return self.x_hat[0], self.loop_reset
```

Trilateration

Die Berechnung der Höhe im vorherigen Abschnitt ist nur eine Dimension der dreidimensionalen Positionsbestimmung. Die übrigen beiden werden durch eine Kombination aus IMU-Werten und empfangenen Bluetooth-Signalstärken ermittelt. Um aus den RSSI-Werten eine Position in der xy-Ebene zu erhalten, wird die Trilateration verwendet. Die Grundlagen dieser Methode wurden in Kapitel 2.3.1 erläutert. Die Umsetzung erfolgt auf dem Raspberry Pi mit Python und wird in diesem Abschnitt beschrieben.

Für die Berechnung der Position in der xy-Ebene wird zunächst eine neue Klasse definiert. Diese heißt *Trilateration* und besitzt zwei Klassenattribute. Die Attribute sind

Listen, welche für alle Objekte der Klasse existieren. In der Liste *intersection_list* werden alle berechneten Schnittpunkte der Kreise gespeichert. Die Liste *best_intersection* beinhaltet die besten Schnittpunkte zur Bestimmung der Postion. Sie enthält entweder einen, zwei oder drei Schnittpunkte. Zu sehen ist dies im folgenden Code-Ausschnitt (28).

Code-Ausschnitt 28: Klasse *Trilateration* mit zwei Klassenattributen

```python
""" Eigene Programmierung """
class Trilateration:
    """ Berechnung der Position durch Trilateration"""
    intersection_list = [] # Liste aller Schnittpunkte
    best_intersection = [] # Liste der Schnittpunkte zur berechnung der Position
```

Nachdem die Klasse instanziiert wurde, gibt die Methode *calcLateration()* die berechnete Position als Tupel an das Programm zurück. Dies geschieht in mehreren Schritten. Zunächst legt die Methode *calcLateration()* die Reihenfolge fest, in welcher die drei Kreise auf Schnittpunkte untersucht werden. Untersucht werden drei Kombinationsmöglichkeiten der Kreise. Für jede Kombination wird die Funktion *getIntersection()* ausgeführt. An sie werden drei Ankerpunkte übergeben, wobei nur von den ersten beiden die Schnittpunkte bestimmt werden. Der dritte Ankerpunkte dient dazu, in einem späteren Schritt den besseren Schnittpunkt zu bestimmen, falls sich die Kreise in zwei Punkten schneiden. Jeder übergebene Ankerpunkt ist ein Objekt der *Beacon*-Klasse aus Kapitel 3.3.2 (Code-Ausschnitt (17)). Jedes dieser Objekte enthält die Position des jeweiligen *Beacon*, sowie die aus der empfangenen Signalstärke berechneten Distanz. Die Methode *getIntersection()* speichert zunächst die Entfernungen zu den Ankerpunkten als Radien des Objekts ab. Zu sehen ist dies in Code-Ausschnitt (29). Mit einer *if*-Abfrage wird überprüft, wie viele Schnittpunkte zwei Kreise haben. Existiert einer oder zwei Punkte, so werden diese berechnet und anschließend der bessere Schnittpunkt bestimmt. Alle Punkte werden in den jeweiligen Listen der Klassenattribute gespeichert.

Existiert kein Schnittpunkt, so wird der Radius des größeren Kreises vergrößert oder verkleinert, bis sich die Kreise schneiden. Durch den Zusammenhang zwischen Signalstärke und Entfernung aus Abbildung (3.10), wird deutlich, dass die Genauigkeit der berechneten Distanz mit zunehmender Entfernung geringer wird. Zusätzlich verursachen *Multipath*-Effekte mit zunehmender Entfernung stärkere Schwankungen der Signalstärke. Aus dieses Gründen wird die Annahme getroffen, dass die Genauigkeit eines Kreises mit geringerem Radius höher ist, als die eines mit größerem Radius. Abhängig davon, ob ein Kreis im zweiten Kreis liegt, muss der Radius vergrößert oder verkleinert werden. In Zeile 26 bis 37 des Code-Ausschnitts (29) wird untersucht, wie die Kreise zueinander liegen. Ist der Radius eines Kreises größer als die Summe des zweiten Radius und dem Abstand beider Kreise, so muss dieser Radius verkleinert werden, da sich die Kreise sonst nicht schneiden. Ist das Gegenteil der Fall, so muss der Radius vergrößert werden. Sollten beide Radien exakt gleich groß sein, werden beide vergrößert, bis sie sich schneiden.

Code-Ausschnitt 29: Methoden der Klasse *Trilateration*

```
""" Eigene Programmierung """
def calcLateration(self):
    """ Führt Lateration durch """
    self.getIntersections(Beacon.beacons[0],Beacon.beacons[1],Beacon.beacons[2])
    self.getIntersections(Beacon.beacons[0],Beacon.beacons[2],Beacon.beacons[1])
    self.getIntersections(Beacon.beacons[1],Beacon.beacons[2],Beacon.beacons[0])
    self.calcPosition()
    return self.position

def getIntersections(self, anchor1, anchor2, anchor3):
    """ Fügt berechnete Schnittpunkte zu Listen hinzu """
    self.radius1 = anchor1.distance
    self.radius2 = anchor2.distance
    self.radius3 = anchor3.distance
    if (self.checkIntersection(anchor1, anchor2) == 2):
        self.calcIntersections()
        self.chooseBetterOne(anchor3)
        Trilateration.intersection_list.append((self.s1_x, self.s1_y))
        Trilateration.intersection_list.append((self.s2_x, self.s2_y))
    elif (self.checkIntersection(anchor1, anchor2) == 1):
        self.calcIntersections()
        self.chooseBetterOne(anchor3)
        Trilateration.intersection_list.append((self.s1_x, self.s1_y))
    else:
        while(self.checkIntersection(anchor1, anchor2) == 0):
            if (self.d + self. radius1) < self.radius2:
                self.radius2 = self.radius2 - 0.01
            elif (self.d + self.radius2) < self.radius1:
                self.radius1 = self.radius1 - 0.01
            else:
                if self.radius1 < self.radius2:
                    self.radius2 = self.radius2 + 0.01
                elif self.radius1 > self.radius2:
                    self.radius1 = self.radius1 + 0.01
                else:
                    self.radius1 = self.radius1 + 0.01
                    self.radius2 = self.radius2 + 0.01
        self.calcIntersections()
        self.chooseBetterOne(anchor3)
        Trilateration.intersection_list.append((self.s1_x, self.s1_y))
```

Die Methode *checkIntersection()* bestimmt die Anzahl der Schnittpunkte zwischen zwei Kreisen. Der entsprechende Code ist in Code-Ausschnitt (30) dargestellt. In einer ersten *if*-Abfrage wird untersucht, wie die beiden Kreise A und B zueinander liegen. Anschließend wird die Anzahl der Schnittpunkte mit einer zweiten *if*-Abfrage bestimmt. Die Radien der Kreise A und B werden mit r_A und r_B bezeichnet. Der Abstand zwischen den beiden Mittelpunkten ist d. Die Schnittmöglichkeiten, die untersucht werden, können wie folgt vereinfacht dargestellt werden:

- Mittelpunkt von A liegt im Kreis B
  - für $(d + r_A < r_B)$ oder $(d + r_B < r_A) \Rightarrow 0$ Schnittpunkte
  - für $(d + r_A = r_B)$ oder $(d + r_B = r_A) \Rightarrow 1$ Schnittpunkt

  - sonst existieren 2 Schnittpunkte

- Mittelpunk von A liegt auf dem Kreis B
  - für $(r_A = 0)$ oder $(r_B = 0) \Rightarrow 1$ Schnittpunkt
  - sonst existieren 2 Schnittpunkte

- Beide Mittelpunkte liegen außerhalb des anderen Kreises
  - für $(d > r_A + r_B) \Rightarrow 0$ Schnittpunkte
  - für $(d = r_A + r_B) \Rightarrow 1$ Schnittpunkt
  - sonst existieren 2 Schnittpunkte

Code-Ausschnitt 30: Anzahl der Schnittpunkte bestimmen

```python
""" Eigene Programmierung, enthält Code-Snippets von Walter Bislin,
abgerufen von http://walter.bislins.ch/blog/index.asp?page=Schnittpunkte
+zweier+Kreise+berechnen+%28JavaScript%29 """
def checkIntersection(self, anchor1 , anchor2):
    """ Funktion gibt die Anzahl der Schnittpunkte zweier Kreise wieder """
    self.x1, self.y1 = anchor1.getRequirement()
    self.x2, self.y2 = anchor2.getRequirement()
    self.r1 = self.radius1
    self.r2 = self.radius2
    self.d = abs(np.sqrt((self.x2 - self.x1)**2 +(self.y2 - self.y1)**2))

    if (self.d < self.r1) or (self.d < self.r2):
        # Mittelpunkt liegt im Kreis
        if (self.d + self.r1 < self.r2) or (self.d + self.r2 < self.r1):
            return 0 # Kein Schnittpunkt
        elif (self.d + self.r1 == self.r2) or (self.d + self.r2 == self.r1):
            return 1 # Ein Schnittpunkt
        else:
            return 2 # Zwei Schnittpunkte

    elif (self.d == self.r1) or (self.d == self.r2):
        # Mittelpunkt liegt auf dem Kreis
        if (self.r1 == 0) or (self.r2 == 0):
            return 1 # Ein Schnittpunkt
        else:
            return 2 # Zwei Schnittpunkte
    else:
        # Mittelpunkte liegen außerhalb der Kreise
        if (self.d > self.r1 + self.r2):
            return 0 # Kein Schnittpunkt
        elif (self.d == self.r1 + self.r2):
            return 1 # Ein Schnittpunk
        else:
            return 2 # Zwei Schnittpunkte

def calcIntersections(self):
    """ Berechnet zwei Schnittpunkte """
    self.c = np.sqrt((self.x2 - self.x1)**2 + (self.y2 - self.y1)**2)
    self.e1_x = (self.x2 - self.x1) / self.c
    self.e1_y = (self.y2 - self.y1) / self.c
    self.e2_x = -self.e1_y
    self.e2_y = self.e1_x
    self.Sx = (self.r1**2 + self.c**2 - self.r2**2) / (2*self.c)
    self.Sy = np.sqrt(abs(self.r1**2 - self.Sx**2))
    self.s1_x = self.x1+self.Sx*((self.x2-self.x1)/self.c)-self.Sy*
        ((self.y2-self.y1)/self.c)
    self.s1_y = self.y1+self.Sx*((self.y2-self.y1)/self.c)+self.Sy*
```

```
        ((self.x2-self.x1)/self.c)
    self.s2_x = self.x1+self.Sx*((self.x2-self.x1)/self.c)+self.Sy*
        ((self.y2-self.y1)/self.c)
    self.s2_y = self.y1+self.Sx*((self.y2-self.y1)/self.c)-self.Sy*
        ((self.x2-self.x1)/self.c)
```

Nachdem die Anzahl der Schnittpunkte zwischen den zwei Kreisen bekannt ist, werden diese durch die Funktion *calcIntersection()* berechnet. Dazu wird die von Bislin (2017) in Kapitel 2.3.1 vorgestellte Methode verwendet. Die Umsetzung ist ebenfalls in Code-Ausschnitt(30) dargestellt und enthält Code-Snippets von Bislin (2017).

Im vorletzten Schritt werden die besten Schnittpunkte nach dem Kriterium (2.15) von Li et al. (2017) ausgewählt und in der Liste *best_intersection* gespeichert. Zuletzt berechnet die Methode *calcPosition()* aus den besten Schnittpunkten die Position, welche anschließend ausgegeben wird. Die Berechnung für drei Schnittpunkte erfolgt nach (2.16) und bestimmt den Mittelpunkt des entstehenden Dreiecks. Für zwei Schnittpunkte wird die Mitte beider Punkte als Position ausgegeben und für einen einzelnen Schnittpunkt wird dieser als Position ausgegeben (siehe Code-Ausschnitt (31)).

Code-Ausschnitt 31: Berechnung der Schnittpunkte und Bestimmung der besten Schnittpunkte

```
""" Eigene Programmierung """
def chooseBetterOne(self,anchor3):
    """ Bestimmt Schnittpunkt für Positionsberechnung """
    self.left  = (anchor3.distance - np.sqrt((self.s1_x - anchor3.pos_lon)**2 +
        (self.s1_y - anchor3.pos_lat)**2))**2
    self.right = (anchor3.distance - np.sqrt((self.s2_x - anchor3.pos_lon)**2 +
        (self.s2_y - anchor3.pos_lat)**2))**2
    if self.left < self.right:
        Trilateration.best_intersection.append((self.s1_x, self.s1_y))
    else:
        Trilateration.best_intersection.append((self.s2_x, self.s2_y))
def calcPosition(self):
    """ Berechnet Position aus drei Schnittpunkten """
    self.position_x = 0
    self.position_y = 0
    if len(Trilateration.best_intersection) == 3:
        for point in Trilateration.best_intersection:
            self.position_x = self.position_x + point[0]
            self.position_y = self.position_y + point[1]
            self.position = ((self.position_x)/3,(self.position_y)/3)
    elif len(Trilateration.best_intersection) == 2:
        for point in Trilateration.best_intersection:
            self.position_x = self.position_x + point[0]
            self.position_y = self.position_y + point[1]
            self.position = ((self.position_x)/2,(self.position_y)/2)
    elif len(Trilateration.best_intersection) == 1:
        self.position = Trilateration.best_intersection[0]
def clearIntersectionLists(self):
    """ Löscht alle Schnittpunkte """
    Trilateration.intersection_list = []
    Trilateration.best_intersection = []
```

Nach jeder Triangulation ist es wichtig, mit *clearIntersectionLists()* alle Schnittpunkte aus den Listen zu löschen, da diese sonst die nächste Berechnung beeinflussen.

Sensorfusion zur Positionsbestimmung

Im finalen Schritt werden alle bisherigen Daten und Berechnungen in der Klasse *Main* gebündelt. Diese erbt Attribute und Methode der *Sensor-, BMPSensor-, Request-*, und *Beacon*-Klassen. Code-Ausschnitt (32) zeigt den Konstruktor von *Main*, der alle Basisklassen initiiert. Anschließend wird der *Request* ausgelöst, welcher die Liste aller vorhandenen *Beacon* wiedergibt, nach denen gescannt werden kann. Nach der Berechnung der Ausgangshöhe und der Instantiierung des BLE-Scanners starten ab Zeile 18 die verschiedenen Prozesse zum Messen und Übertragen von Sensordaten. Anschließend wird für jeden erkannten BLE *Beacon* ein *Request* an die von ihm übertragene URL geschickt. Die Antwort beinhaltet die Position des jeweiligen *Beacon*, sowie seinen Namen (siehe Zeile 23 bis 29).

Die Funktion *stopSensorAll()* ist die einzige Methode der *Main*-Klasse. Sie ist sehr wichtig, wenn das Programm zur Positionierung beendet werden soll. Da die Daten der verschiedenen Sensoren in eigenen *Threads* gesammelt werden und diese unabhängig vom *Main-Thread* existieren, müssen sie vor dem Schließen des Hauptprogramms beendet werden. Diese Aufgabe übernimmt die genannte Methode. Wird diese Funktion bei Beenden des Programms nicht ausgeführt, so arbeiten alle Sensoren und deren *Threads* weiter.

Code-Ausschnitt 32: Aufbau der *Main*-Klasse

```
 1 """ Eigene Programmierung, enthält Code-Snippets der Adafruit BMP280 Library
 2 abgerufen von https://circuitpython.readthedocs.io/projects/bmp280/en/latest/"""
 3 class Main(Sensor, BMPSensor, Beacon, Request):
 4     """ Main Klasse führt alle Sensorklassen zusammen """
 5     def __init__(self, sensor_queue, bmp_queue, url=' '):
 6         """ Konstrukor initiiert alle Sensoren """
 7         self.sensor_queue = sensor_queue # Queue für Arduino-Sensoren
 8         self.bmp_queue    = bmp_queue    # Queue für BMP-Sensor
 9 
10         Sensor.__init__(self)    # Aktiviert Konstruktor der Sensorklasse
11         BMPSensor.__init__(self) # Aktiviert Konstruktor der BMPklasse
12         request = Request(url)   # Aktiviert Konstruktor der Requestklasse
13         request.getBeaconList(Beacon.beacon_list)  # erzeugt Liste mit Beacon
14 
15         self.mean_height = BMPSensor.calcMeanHeight(self) # mittlere Höhe
16         self.ble_scanner = Beacon.initBeaconScanner()     # erzeugt Scanner
17 
18         ''' Startet Sensor-Readings '''
19         self.startReadingData(self.sensor_queue) # IMU
20         self.startMeasuringHeight(self.bmp_queue)# BMP
21         Beacon.startBleScanner(self.ble_scanner) # BLE
22 
23         while len(Beacon.beacons) <= 2:
24             pass
25         for beacon in Beacon.beacons:
26             b_request = Request(url + beacon.url[-1] + '/')
27             beacon.setPosition(b_request.getBeaconData('pos_lon'),
28                                b_request.getBeaconData('pos_lat'))
29             beacon.setName(b_request.getBeaconData('name'))
30 
31     def stopSensorAll(self):
32         """ Stopt alle Threads und trennt Verbindungen """
```

```
        self.stopMeasuringHeight()              # stop BMP
        self.stopReadingData()                  # stop IMU
        self.stopConnection()                   # stop serielle Verbindung
        Beacon.stopBleScanner(self.ble_scanner) # stop BLE-Scan
```

Nachdem alle Sensordaten und die Positionen der Trilateration vorliegen, wird erneut mit Hilfe des Kalman-Filters eine Sensordatenfusion durchgeführt. Ähnlich wie bei der Berechnung der Höhe, werden die gemessenen Beschleunigungen zweifach integriert und mit den Positionen der Trilateration fusioniert. Der Zustand des Systems wird somit wie folgt beschrieben:

$$\hat{x} = \begin{pmatrix} pos_x \\ pos_y \\ v_x \\ v_y \end{pmatrix}$$

Mit der Übergangsmatrix (*prediction matrix*) F_k, der Steuermatrix B_k und dem Eingangsvektor $\overrightarrow{u}_k$ wird der nächste Zustand des Systems mit (2.27) geschätzt. Der für die *Update*-Phase notwendige Messvektor $\overrightarrow{z}_k$ und die genannten Matrizen werden folgendermaßen definiert:

$$F_k = \begin{pmatrix} 1 & 0 & dt & 0 \\ 0 & 1 & 0 & dt \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, B_k = \begin{pmatrix} \frac{dt^2}{2} & 0 & 0 & 0 \\ 0 & \frac{dt^2}{2} & 0 & 0 \\ dt & 0 & 0 & 0 \\ 0 & dt & 0 & 0 \end{pmatrix}, \overrightarrow{u}_k = \begin{pmatrix} acc_x \\ acc_y \\ 0 \\ 0 \end{pmatrix}, \overrightarrow{z}_k = \begin{pmatrix} pos_x \\ pos_y \\ 0 \\ 0 \end{pmatrix}$$

Die Systemunsicherheit Q wird durch eine 4×4 Matrix beschrieben, deren Einträge sich wie in (3.4) ergeben. Die Umsetzung erfolgt im Code-Ausschnitt (33) in Zeile 38. Da Q ebenso wie F_k und B_k (Code-Ausschnitt (33), Zeile 27) von dt abhängen, müssen diese für jede neue Berechnung neu bestimmt werden. Dies geschieht über die Methoden *setQ()* und *setFB()*. Die übrigen Matrizen H_k, R und die Initialisierung des Zustandsvektors des Systems $\hat{x}_k$ sowie dessen Kovarianzmatrix P_k werden im Konstruktor definiert.

Ziel ist es, die Bewegungen des Systems im Raum in Zusammenhang mit den Ergebnissen der Trilateration zu bringen. Alle Sensordaten beziehen sich auf das Koordinatensystem des jeweiligen Sensors. Wird demnach eine Beschleunigung in x-Richtung gemessen, entspricht dies der x-Richtung des Sensors. Je nach Lage und Orientierung des Systems stimmt dies nicht mit dem Koordinatensystem der *Beacon* überein. Die Positionen der *Beacon* können absolut oder relativ angegeben werden. Wichtig ist, dass das Koordinatensystem nach Norden ausgerichtet ist und dass die Einheiten mit denen des Systems über einstimmen. Das System misst Beschleunigungen in $\frac{m}{s^2}$. Die mit Hilfe der IMU bestimmten Bewegungen und zurückgelegten Strecken sind demnach in Metern angegeben. Mit (3.5) wird die x- und y- Achse des Sensorkoordinatensystems in mathematisch negativer Richtung um den Kurswinkel α_{head} gedreht. Der Kurswinkel wird durch den zuvor entwickelten Kompass bestimmt und ergibt mit den Beschleunigungen acc_x und acc_y die neuen Beschleunigungen acc_x^* und acc_y^* in Bezug auf das Koordinatensystem der *Beacon*:

$$\begin{aligned} acc_x^* &= cos(-\alpha_{head}) \cdot acc_x - sin(-\alpha_{head}) \cdot acc_y \\ acc_y^* &= sin(-\alpha_{head}) \cdot acc_x + cos(-\alpha_{head}) \cdot acc_y \end{aligned} \tag{3.5}$$

Nachdem alle Komponenten definiert wurden, führt die Methode *calcPosition()* die Berechnung der *Prediction*- und *Update*-Phase des Kalman-Filters durch (Code-Ausschnitt (33), Zeile 54). Anschließend wird das Tupel (x, y) der berechneten Position zurück gegeben.

Code-Ausschnitt 33: Sensordatenfusion zur Berechnung der Position in X und Y

```
1  """ Eigene Programmierung """
2  class KFPosition:
3    """ Berechnung der Position mit Erweitertem Kalman-Filter """
4    def __init__(self,pos_x, pos_y, head):
5        """ Konstruktor initiiert Matrizen für KF """
6        # init Prediction
7        self.x_hat = np.matrix([[pos_x],[pos_y],[0],[0]])
8        self.P = np.matrix([[0,0,0,0],
9                            [0,0,0,0],
10                           [0,0,0,0],
11                           [0,0,0,0]])
12       # init Update
13       self.H = np.matrix([[1,0,0,0],
14                           [0,1,0,0],
15                           [0,0,1,0],
16                           [0,0,0,1]])
17       self.r_pos  = 2.8
18       self.R = np.matrix([[self.r_pos],[self.r_pos],[0],[0]])
19
20   def transKoord(self,ax,ay,head):
21       """ Transformation Sensor- in geograf.-Koordiantensystem """
22       head = -head * np.pi / 180
23       nord_ax = np.cos(head) * ax - np.sin(head) * ay
24       nord_ay = np.sin(head) * ax + np.cos(head) * ay
25       return nord_ax, nord_ay
26
27   def setFB(self,ax,ay, dt):
28       """ Berechnet Steuer- und Eingangs-Matrix """
29       self.F = np.matrix([[1, 0, dt,  0],
30                           [0, 1,  0, dt],
31                           [0, 0,  1,  0],
32                           [0, 0,  0,  1]])
33       self.B = np.matrix([[0.5*dt**2, 0 , 0 , 0],
34                           [0, 0.5*dt**2 , 0 , 0],
```

```
                         [dt, 0 , 0 , 0],
                         [0 , dt, 0 , 0]])

    def setQ(self, dt):
        """ Berechnet System-Unsicherheit """
        self.sigma_a = 0.7
        self.sigma_v = self.sigma_a * dt
        self.sigma_s = 0.5 * self.sigma_a * dt**2

        self.Q = np.matrix([[self.sigma_s**2, self.sigma_s**2,
                         self.sigma_s*self.sigma_v, self.sigma_s*self.sigma_v],
                         [self.sigma_s**2, self.sigma_s**2,
                         self.sigma_s*self.sigma_v, self.sigma_s*self.sigma_v],
                         [self.sigma_v*self.sigma_s,self.sigma_v*self.sigma_s,
                         self.sigma_v**2, self.sigma_v**2],
                         [self.sigma_v*self.sigma_s,self.sigma_v*self.sigma_s,
                         self.sigma_v**2, self.sigma_v**2]])


    def calcPosition(self, ax, ay, pos_x, pos_y, head, dt=0.05):
        """ Berechnet neue Position """
        ax_n, ay_n = self.transKoord(ax, ay, head)
        self.u = np.matrix([[ax_n],[ay_n],[0],[0]])
        self.setFB(ax_n, ay_n, dt)
        self.setQ(dt)
        self.z = np.matrix([[pos_x],[pos_y],[0],[0]])

        # Prediction
        self.x_hat = self.F * self.x_hat + self.B * self.u
        self.P = self.F * self.P * np.transpose(self.F) + self.Q
        # Update
        self.S = inv(self.H * self.P * np.transpose(self.H) + self.R)
        self.K = self.P * np.transpose(self.H) * self.S
        self.x_hat = self.x_hat + self.K * (self.z - self.H * self.x_hat)
        self.P = self.P - self.K * self.H * self.P
        return (float(self.x_hat[0]), float(self.x_hat[1]))
```

Ist das System geneigt, werden auf Grund der Gravitation Beschleunigungen gemessen, obwohl sich das System in Ruhe befindet. Es ist daher notwendig, wie schon bei der Berechnung der Höhe des Systems, eine Neigungskompensation durchzuführen:

$$acc_{x,tilt} = acc_x - sin(\alpha_{pitch}) \cdot 9.81 \frac{m}{s^2}$$
$$acc_{y,tilt} = acc_y - sin(\alpha_{roll}) \cdot 9.81 \frac{m}{s^2}$$

Für die Umsetzung in Python wird die Funktion *tiltCompensation()* (Code-Ausschnitt (34)) definiert. Für die Berechnung des Sinus in Python müssen die Winkel *pitch* und *roll* als Radiant angegeben werden (Zeile 4 und 5)

Code-Ausschnitt 34: Neigungskompensation acc_x und acc_y

```
1 """ Eigene Programmierung """
2 def tiltCompensation(ax, ay, pitch, roll):
3     """ Neigungskompensation für ax und ay """
4     pitch = pitch *  np.pi / 180
5     roll  = roll  *  np.pi / 180
```

```
    a_x_tilt = ax - np.sin(pitch) * 9.81
    a_y_tilt = ay - np.sin(roll)  * 9.81
    return round(a_x_tilt,2), round(a_y_tilt,2)
```

Nachdem alle Klassen und Funktionen zur Berechnung der Eigenschaften des Systems vorhanden sind, werden diese im Code-Ausschnitt (35) zusammengeführt. In den ersten Zeilen wird die URL mit allen registrierten *Beacon* festgelegt, die *queue* im *Last in, first out* (Lifo) Prinzip initiiert und Objekte der *Main*- und *Trilateration*-Klassen erzeugt. Anschließend werden in Zeile 10 bis 15 Startbedingungen festgelegt, dazu gehören unter anderem die Startzeit, die Messdauer und die Anfangsposition mit Ausgangshöhe.

Eine *while*-Schleife wiederholt sich so lange, wie es die angegebene Messdauer festlegt. Anschließend werden mit *main.stopSensorAll()* (Zeile 76) alle Verbindungen und Prozesse beendet. Während jedes Schleifendurchlaufs werden zunächst alle Sensordaten aus den einzelnen *Threads* ausgegeben (Zeile 21 bis 40). Dabei werden die Beschleunigungswerte von x und y direkt durch die Lage des Sensors korrigiert (Zeile 24). Ebenfalls wird aus der empfangenen Signalstärke jedes *Beacon* die Distanz bestimmt (Zeile 38 bis 40). Es folgt die Berechnung der Postion in der xy-Ebene aus der Trilateration und der Sensorfusion mit den gemessenen Beschleunigungswerten in x- und y-Richtung (Zeile 41 bis 53) und die Berechnung der Höhe aus den Werten des Barometers und den Beschleunigungen in z-Richtung (Zeile 55 bis 57). Anschließend wird die Zeit gemessen, die jeder Schleifendurchlauf benötigt hat und in *dt* gespeichert. Dieser Wert wird unter anderem für die Integration der Beschleunigungen der Kalman-Filter verwendet. Nachdem jede *queue* geleert ist und die dreidimensionale Position ausgegeben wurde, wiederholt sich die Schleife bis die angegebene Messzeit erreicht ist.

Code-Ausschnitt 35: Zusammenführen aller Komponenten

```
1  """ Eigene Programmierung """
2  if __name__ == '__main__':
3    main_url = 'http://192.168.1.101:8000/beacons/'
4    bmp_queue = queue.LifoQueue() # Warteschlange für BMP-Thread
5    sensor_queue  = queue.LifoQueue() # Warteschlange für Sensor-Thread
6    main = Main(sensor_queue, bmp_queue, main_url) # Erzeugt Main-Objekt
7    ble_queue = Beacon.ble_queue
8    lateration = Trilateration() # Erzeugt ein Objekt zur Trilateration
9
10   messzeit =  10# Legt Messzeit fest
11   startzeit = time.time() # Speichert Startzeit
12   loop_height_kalman = 0 # für Dauer der Höhenberechnung mit Kalman-Filter
13   dt = 0.05 # Initiierung des ersten dt
14   position = (0,0)
15   height_calc = HeightKalman() # Erzeugt Objekt für Kalman-Filter der Höhe
16
17   while time.time() < startzeit + messzeit: # Schleife für Messzeit
18       loop_start = time.time() # Start für Loopzeit Messung
19
20       # Ausgabe Arduino-Daten
21       arduino_data = sensor_queue.get()
22       sensor_queue.task_done()
23       ax_tilt, ay_tilt = tiltCompensation(arduino_data[0], arduino_data[1],
```

```
                                      arduino_data[3], arduino_data[4])
    # Korrektur für sehr kleine Beschleunigung
    if -0.15 < ax_tilt < 0.15: ax_tilt = 0
    if -0.15 < ay_tilt < 0.15: ay_tilt = 0

    # Ausgabe BMP-Daten
    bmp_height = main.mean_height - bmp_queue.get()
    bmp_queue.task_done()

    # Ausgabe BLE-Daten
    signals = ble_queue.get()
    ble_queue.task_done()

    for beacon in Beacon.beacons:
        if beacon.mac_adr == signals[0]:
            beacon.getDistance(signals[1])
    # Berechnung der Position
    if (Beacon.beacons[0].distance > 0.0) and
       (Beacon.beacons[1].distance > 0.0) and
       (Beacon.beacons[2].distance > 0.0):
        position = lateration.calcLateration()
        lateration.clearIntersectionLists() # Löscht alten Schnittpunkte

    if 'arduino_data' in locals() and len(arduino_data) == 6:
        try:
            final_position = kf_position.calcPosition(ax_tilt, ay_tilt,
                      position[0], position[1], arduino_data[5], dt)
        except Exception:
            kf_position = KFPosition(position[0],position[1],arduino_data[5])

    # Berechnung der Höhe
    height, loop_reset_height = height_calc.calcHeight(arduino_data[2],
      arduino_data[3], arduino_data[4], bmp_height, loop_height_kalman, dt)

    loop_end = time.time() - loop_start # Dauer eines Schleifendurchgangs
    dt = loop_end # dt = Dauer eines Schleifendurchgangs

    if loop_reset_height: # Schleifedurchlauf für Höhenberechnung
        loop_height_kalman = 0
    else:
        loop_height_kalman = loop_height_kalman + loop_end

    sensor_queue.queue.clear()
    bmp_queue.queue.clear()
    ble_queue.queue.clear()
    try:
        print("%.2f , %.2f , %.2f" % (final_position[0], final_position[1],
                              height))
    except Exception:
        pass

main.stopSensorAll()
```

3.3.4 Testanwendung auf einem Bewegungssimulator

Nach der Entwicklung und Umsetzung des in Kapitel 3.2 ausgearbeiteten Konzepts, wird das System zur Positionierung im *Indoor*-Bereich getestet. Untersucht werden neben der Genauigkeit der Postion auch einzelne Komponenten der entwickelten IMU, wie zum Beispiel das Gyroskop und der Kompass.

Von den Ergebnissen der Experimente mit dem entwickelten System werden nicht nur Informationen über die Genauigkeit erhofft. Vielmehr wird erwartet, eine Aussage über eventuelle Einsatzmöglichkeiten und Beschränkungen des Systems treffen zu können. Externe Einflüsse, die das System und deren Messungen beeinträchtigen, müssen identifiziert werden.

Ein erstes Experiment untersucht die Position in der zweidimensionalen xy-Ebene. Der Versuch wird unter realen Bedingungen in einem möblierten Zimmer durchgeführt. Dieses besitzt eine Grundfläche von etwa 15 m^2 und eine Deckenhöhe von etwa 2.90 m. Die drei BLE *Beacon* werden so verteilt, dass das entstehende Dreieck annähernd gleichseitig ist. Es besteht zu jeder Zeit während des Experiments eine direkte Sichtverbindung zwischen dem Positionierungssystem und jedem BLE *Beacon*. Dies ist notwendig, da der Zusammenhang zwischen gemessenen Signalstärken und der Entfernung aus Kapitel 3.3.2 für den Fall einer direkten Sichtverbindung mit Hilfe einer logarithmischen Regression modelliert wurde. Für ein Experiment, bei dem das Signal Wände oder andere Objekte durchdringt, kann der Zusammenhang mit (2.3), wie in Kapitel 2.1.1 beschrieben, modelliert werden. Für die Einmessung der *Beacon* werden für den ersten *Beacon* die Koordinaten $(0, 0)$ festgelegt. Von ihm ausgehend lassen sich die Koordinaten der übrigen beiden *Beacon* wie folgt berechnen:

$$x = -sin(360 - \alpha) * d$$

$$y = cos(360 - \alpha) * d$$

Die gemessene Distanz zwischen den *Beacon* ist durch d beschrieben. Der Winkel α stellt den Kurswinkel da und wird mit einem Kompass bestimmt. Tabelle 3.1 zeigt die Messungen und die sich ergebenen Koordinaten der drei Ankerpunkte:

Beacon	d	α	Lon	Lat
1	/	/	$0\,m$	$0\,m$
2	$2.55\,m$	$327.3°$	$-1.38\,m$	$2.14\,m$
3	$2.59\,m$	$255.8°$	$-2.51\,m$	$-0.63\,m$

Tabelle 3.1: Position der eingemessenen *Beacon*
Eigene Darstellung

Das Positionierungssystem wird auf einen Bewegungssimulator montiert (siehe Abbildung 3.16), welcher die Aufgabe hat, auf einer zuvor festgelegten Route zwischen den Ankerpunkten autonom zu fahren. Zu diesem Zweck wird der *Arduino Robot* verwendet. Er besitzt Sensoren, die es ihm ermöglichen einer Linie zu folgen. Insgesamt sieben Punkte werden für die

Route eingemessen und anschließend durch schwarzes Klebeband auf dem Boden miteinander verbunden. Die Positionen der einzelnen Stützpunkte der Route wird auf dieselbe Weise berechnet wie zuvor die Position der Ankerpunkte. Der erste *Beacon* mit den Koordinaten $(0, 0)$ dient als Ausgangspunkt, von dem jeder Stützpunkt aus berechnet wird. Tabelle 3.2 zeigt die Messungen und die sich ergebenden Koordinaten aller sieben Stützpunkte. Die Teststrecke sowie die drei *Beacon* sind in Abbildung 3.15 dargestellt.

Stützpunkt	d	α	Lon	Lat
1	$1.45\,m$	$25.6°$	$0.626\,m$	$1.31\,m$
2	$2.15\,m$	$330.2°$	$-1.1\,m$	$1.87\,m$
3	$2.10\,m$	$310.6°$	$-1.59\,m$	$1.37\,m$
4	$2.29\,m$	$268.4°$	$-2.28\,m$	$-0.06\,m$
5	$2.23\,m$	$261.3°$	$-2.2\,m$	$-0.34\,m$
6	$1.63\,m$	$254.1°$	$-1.56\,m$	$-0.45\,m$
7	$1.05\,m$	$253.3°$	$-1.0\,m$	$-0.32\,m$

Tabelle 3.2: Positionen der sieben Stützpunkte der Testroute
Eigene Darstellung

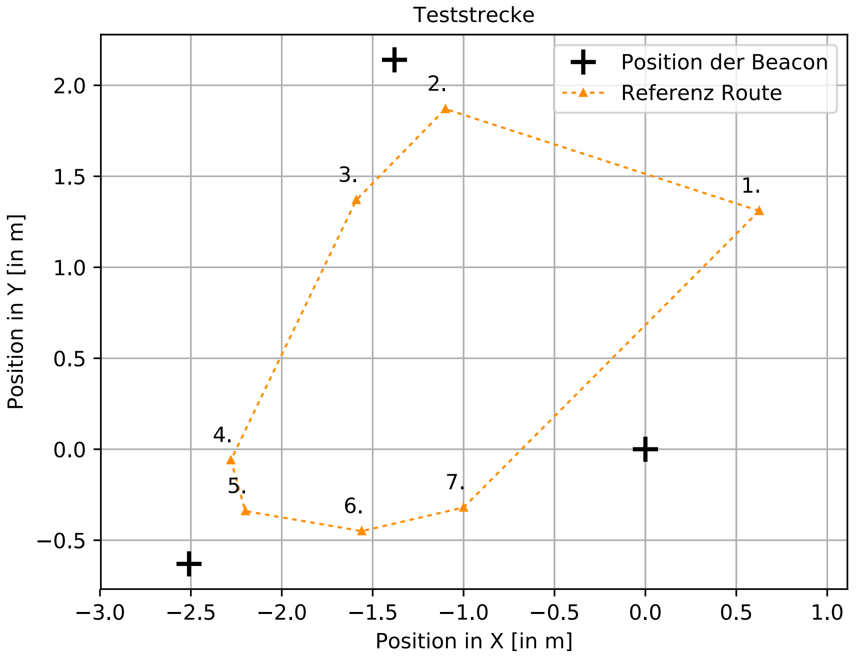


Abbildung 3.15: Teststrecke für den Versuchsaufbau mit bezifferten Stützpunkten und den *Beacon*

Eigene Darstellung

Für die Programmierung des *Arduino Robot* wird das fertiges Sketch-Beispiel *Line Following* des Herstellers verwendet. Dieses Skript lässt den Roboter den Boden nach einer schwarzen Linie absuchen. Hat er eine gefunden, so beginnt er damit dieser zu folgen. Der Code wird einmal ausgeführt und muss anschießend für jede Messung neu gestartet werden.

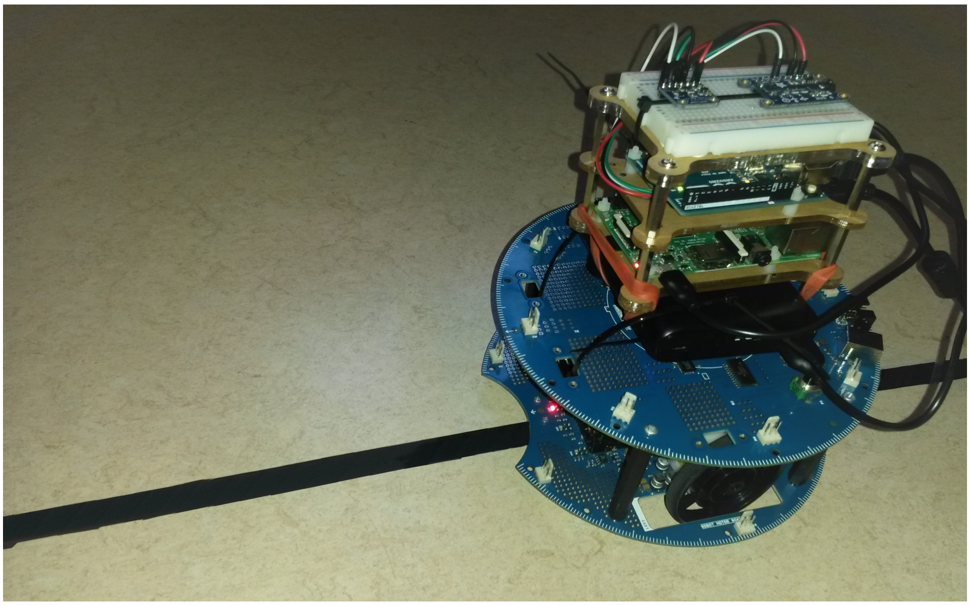


Abbildung 3.16: Das Entwicklte System zur Indoorpositionierung, montiert auf einem *Arduino Robot*
Eigene Darstellung

Da sich das System bei dem genannten Experiment nur auf dem ebenen Fußboden bewegt und der *Arduino Robot* somit seine Höhe nicht verändert, muss diese in einem separaten Versuch untersucht werden. Hierbei wird das System nacheinander auf verschiedene Höhen versetzt. Jede Höhe ist zuvor gemessen worden und bezieht sich auf die Ausgangshöhe. Für diese wird der Boden mit einer Höhe von Null definiert.

Für weitere Untersuchungen werden während des Testdurchlaufs zur Bestimmung der zweidimensionalen Postion Daten über den Kurswinkel und den Geschwindigkeiten in x und y gesammelt. Der Kurswinkel wird anschließend mit Referenzmessungen der einzelnen Streckenabschnitte verglichen.

Ebenfalls wird die Genauigkeit des entwickelten Gyroskops untersucht. Dazu wird das System zunächst in verschiedene Winkel geneigt. Anschließend wird der gleiche Versuch zur Überprüfung des Rollwinkels durchgeführt. Referenzmessungen werden durch einen digitalen Winkel- und Neigungsmesser aufgenommen.

4 Ergebnisse

Im Folgenden werden die Ergebnisse der in Kapitel 3.3.4 erläuterten Experimente mit dem hier entwickelten IPS und der IMU beschrieben. Dies beinhaltet die Ergebnisse der zweidimensionalen Positionsbestimmung des Systems auf dem *Arduino Robot* in der xy-Ebene und die Höhe als dritte Dimension der gesuchten Position. Während der Messungen der zweidimensionalen Positionsbestimmung wurden ebenfalls der Kurswinkel sowie die Geschwindigkeiten in x und y aufgezeichnet. Diese Ergebnisse und die Untersuchung des Gyroskops befinden sich ebenfalls in diesem Kapitel.

4.1 2D Position

Das Ergebnis der Testandwendung des entwickelten Positionierungssystems auf einem Bewegungssimulator ist in Abbildung 4.1 dargestellt. Die BLE *Beacon* werden durch ein schwarzes Plus gekennzeichnet. Die Position jedes einzelnen kann der Tabelle 3.1 entnommen werden. In orange ist die Route eingezeichnet, die das System abgefahren ist. Die Fahrtrichtung verläuft gegen den Uhrzeigersinn. Start- und Endpunkt des Versuchs liegt beim Referenzpunkt sieben. Die blaue Route stellt das Ergebnis der berechneten Position des Systems dar. Diese ergibt sich, wie in Kapitel 3.3.3 beschrieben, aus der Sensorfusion der gemessenen IMU-Daten und der Trilateration der empfangen Signalstärke der BLE *Beacon*.

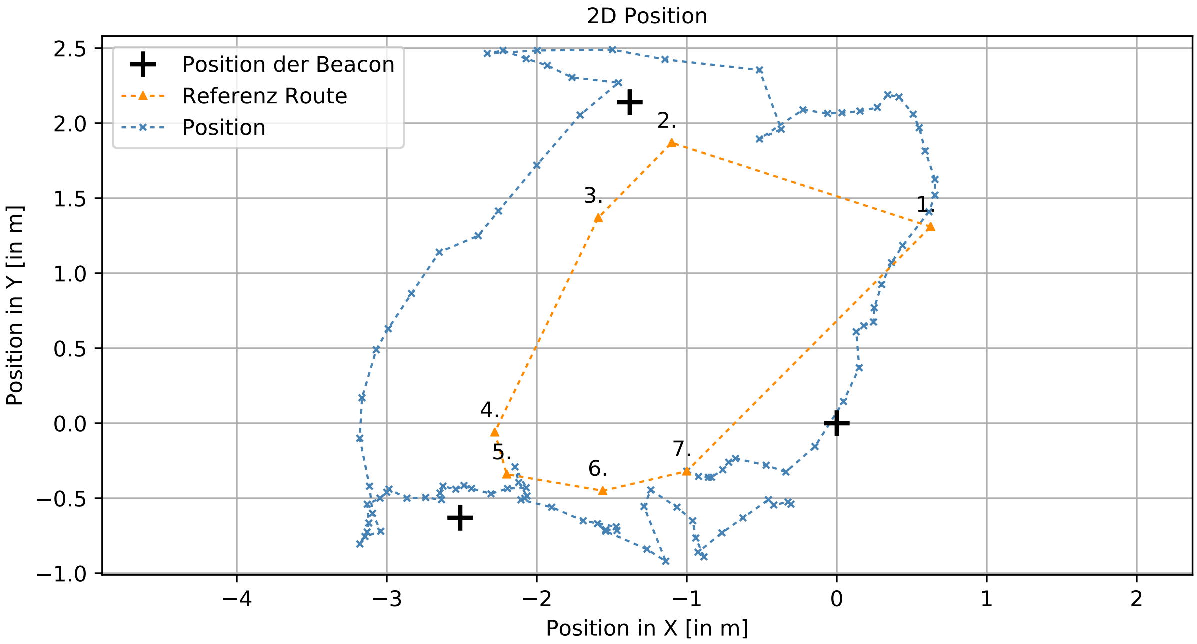


Abbildung 4.1: Ergebnis der Positionsbestimmung in der xy-Ebene
Eigene Darstellung

In der Abbildung ist zu erkennen, dass sich der Großteil der gemessenen Positionen außerhalb des Dreiecks befindet, welches durch die drei Ankerpunkte aufgespannt wird. Die Bereiche, in denen die gemessenen Positionen am dichtesten an der Referenzstrecke liegen, befinden sich an den Stützpunkten eins und fünf der befahrenen Route. Da der Versuch an Stützpunkt sieben begann, wurde dessen Position bei der Initialisierung des Filters als Ausgangsposition angegeben. Aus diesem Grund befindet sich die gemessenen Position dort sehr dicht an der realen. Die größten Abweichungen sind auf der Strecke zwischen den Stützpunkten eins bis vier zu erkennen.

Durch den Versuchsaufbau sind lediglich die befahrene Route und die berechneten Positionen des Systems bekannt. Es besteht jedoch keine exakte Zuordnung zwischen den gemessenen Positionen und den wirklichen. Es ist nicht möglich, den Messfehler der Position mathematisch zu bestimmen. Die Genauigkeit kann lediglich grob aus Abbildung 4.1 abgelesen werden. Demnach schwankt die Differenz der realen und der gemessenen Positionen zwischen 10 cm und 150 cm.

Um die Genauigkeit besser bestimmen zu können, wurden an drei Punkten der Teststrecke Messungen mit dem System durchgeführt. Die aufgezeichneten Daten sind in Abbildung 4.2 dargestellt. Für diese Untersuchung befand sich das System in Ruhe. Somit spiegeln die Ergebnisse nur die Position der gefilterten Trilateration wieder. Aus den gesammelten Daten werden die Entfernungen zwischen jedem Punkt und dem dazugehörendem Referenzpunkt mit (4.1) bestimmt. Das Ergebnis der Differenz der mittleren-, maximalen- und minimalen Distanz zum Referenzwert wird in Tabelle 4.1 dargelegt.

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{4.1}$$

Punktmessung	Δd_{min}	$\overline{\Delta d}$	Δd_{max}
1 (rot)	$0.09\,m$	$0.37\,m$	$0.67\,m$
2 (blau)	$0.03\,m$	$0.61\,m$	$1,10\,m$
3 (grün)	$0.01\,m$	$0.19\,m$	$0.54\,m$

Tabelle 4.1: Ergebnis der Punktmessungen
Eigene Darstellung

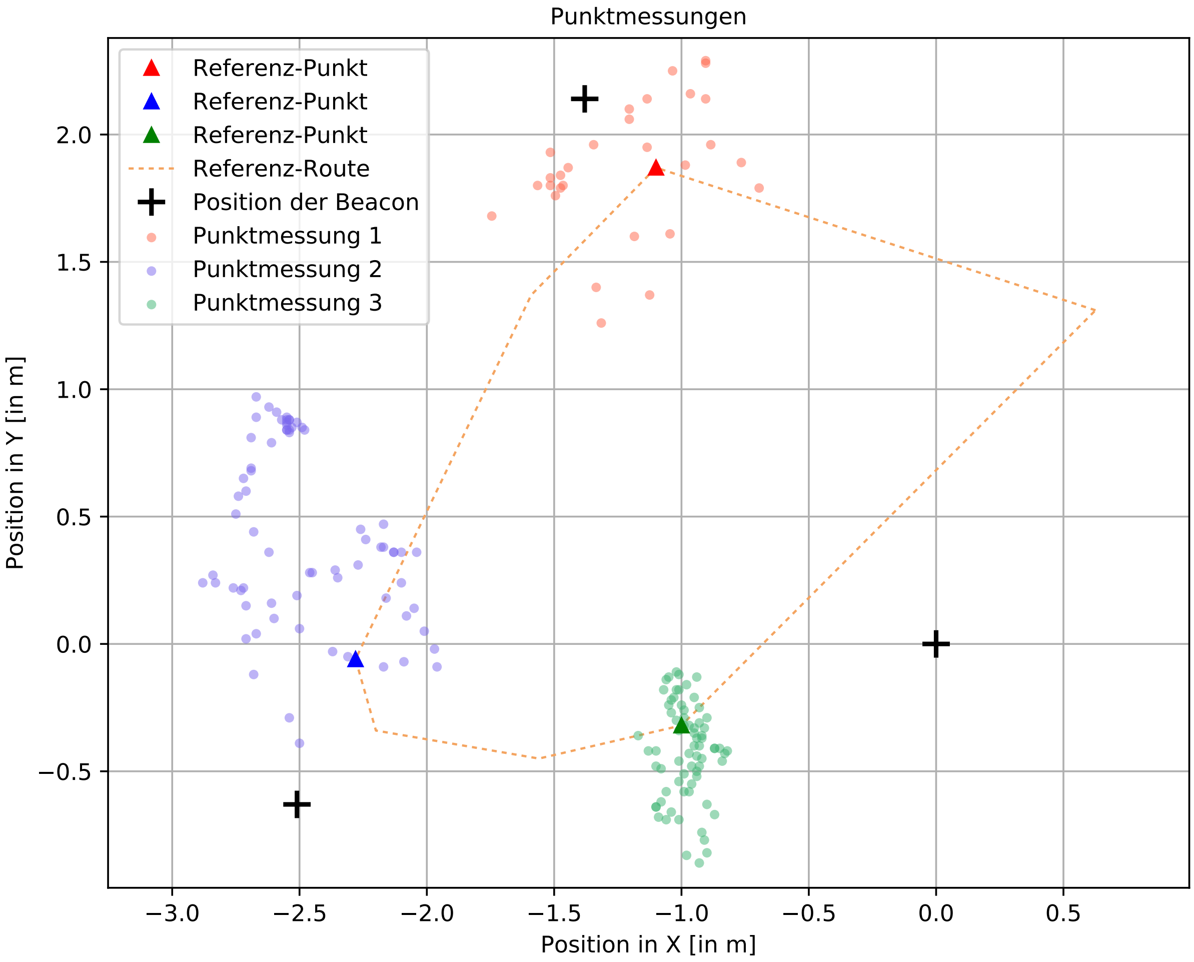


Abbildung 4.2: Punktmessungen während sich das System in Ruhe befindet
Eigene Darstellung

4.2 Höhe

Da sich der *Arduino Robot* nur auf einer Ebene bewegen kann, wurde für die zu untersuchende Höhe ein separater Versucht durchgeführt. Dieser wird in Kapitel 3.3.4 genauer beschrieben.

Abbildung 4.3 zeig das Ergebnis der Höhenmessungen mit dem entwickelten System. Mit der Ausgangshöhe wurden die Messungen auf insgesamt fünf verschiedenen Ebenen durchgeführt. In orange sind diese im Diagramm dargestellt. Die Höhenwerte wurden durch die Sensorfusion der IMU- und Barometerdaten, wie in Kapitel 3.3.3 beschrieben, berechnet. Sie sind im Diagramm blau dargestellt. Auf der Abszisse ist die Messzeit in Sekunden abgetragen. Der gesamte Versuch dauerte etwa 2.6 Minuten. Die Differenz der Höhen und der Referenzwerte der fünf Referenzebenen werden in Tabelle 4.2 dargelegt. Für die Berechnung der minimalen, maximalen und durchschnittlichen Höhe wurden die gemessen Werte, die sich zwischen den verschiedenen Ebenen befinden, nicht mit in Betracht gezogen.

Ebene	Referenz Höhe	Δh_{min}	Δh_{max}	$\overline{\Delta h}$
0	$0\,m$	$0.0\,m$	$0.16\,m$	$0.08\,m$
1	$0.39\,m$	$0.01\,m$	$0.11\,m$	$0.04\,m$
2	$0.61\,m$	$0.0\,m$	$0.29\,m$	$0.15\,m$
3	$0.85\,m$	$0.02\,m$	$0.25\,m$	$0.11\,m$
4	$1.68\,m$	$0.0\,m$	$0.18\,m$	$0.08\,m$

Tabelle 4.2: Ergebnis der Höhenmessungen
Eigene Darstellung

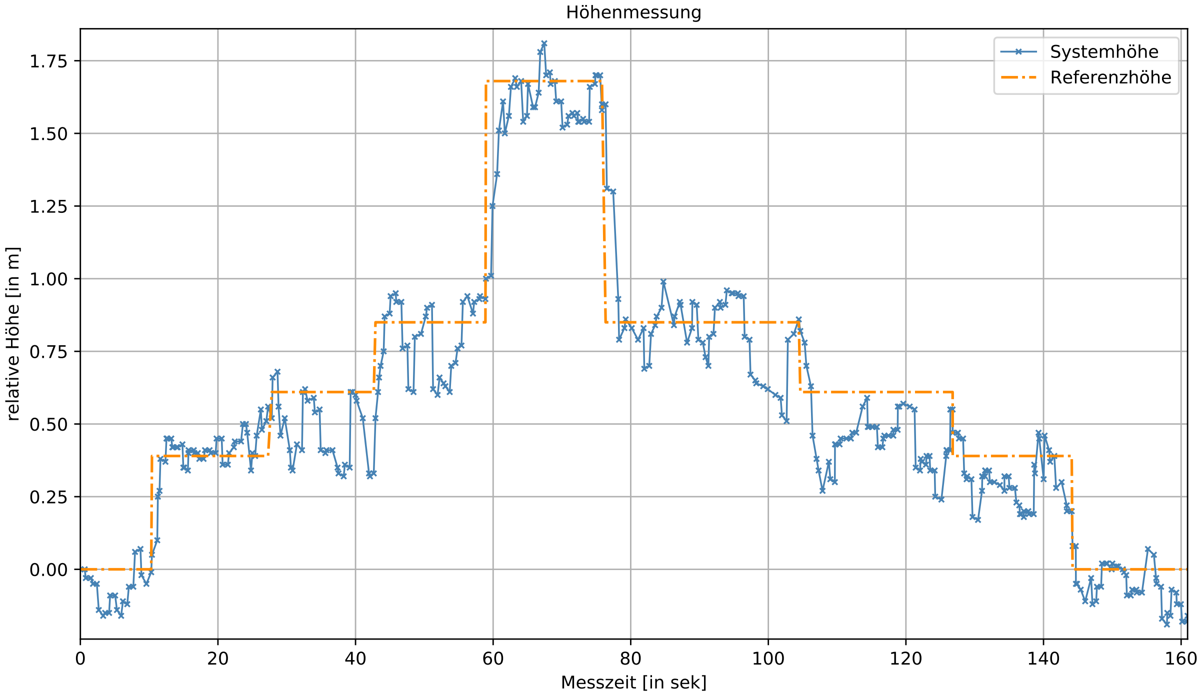


Abbildung 4.3: Ergebnis der Höhenmessungen
Eigene Darstellung

4.3 IMU

Während der Testanwendung des Systems auf dem Bewegungssimulator wurden neben den gemessen Positionen aus Kapitel 4.1 noch der Kurswinkel und die aus der Beschleunigung integrierte Geschwindigkeit aufgezeichnet. Diese Daten sind in Abbildung 4.4 dargestellt.

Das obere der beiden Diagramme zeigt den Kurswinkel des Systems während des Tests. In orange werden die Referenzwerte dargestellt. Sie beziehen sich auf die Teststrecke aus Abbildung 4.1. Der Versuch beginnt und endet auf der Strecke zwischen den Stützpunkten sieben und eins und verläuft in mathematisch negativer Richtung. Aus der absoluten Differenz des gemessenen Kurswinkels vom Referenzwinkel wurden die minimale, maximale und mittlere Abweichung berechnet. Die Messdaten, die direkt an den Stützpunkten aufgenommen wurden, fließen nicht in die Berechnung mit ein, da sich der *Arduino Robot* an diesen Stellen mehrfach dreht, um dem Verlauf der Linie folgen zu können. Die Differenzen der gemessen Kurswinkel und der Referenzwerte sind in Tabelle 4.3 dargestellt:

Streckenabschnitt	Referenzwinkel	$\Delta heading_{min}$	$\Delta heading_{max}$	$\overline{\Delta heading}$
7-1	49.7°	2.0°	25.6°	13.8°
1-2	290.3°	0.9°	44.9°	16.6°
2-3	242.8°	12.4°	13.7°	12.9°
3-4	220.6°	1.1°	14.0°	6.2°
4-5	167.1°	8.6°	22.3°	15.5°
5-6	136.3°	0.8°	12.0°	6.6°
6-7	121.9°	6.0°	12.1°	10.0°
7-1	61.7°	2.3°	21.0°	8.1°

Tabelle 4.3: Untersuchung des Kurswinkels aus der Testanwendung
Eigene Darstellung

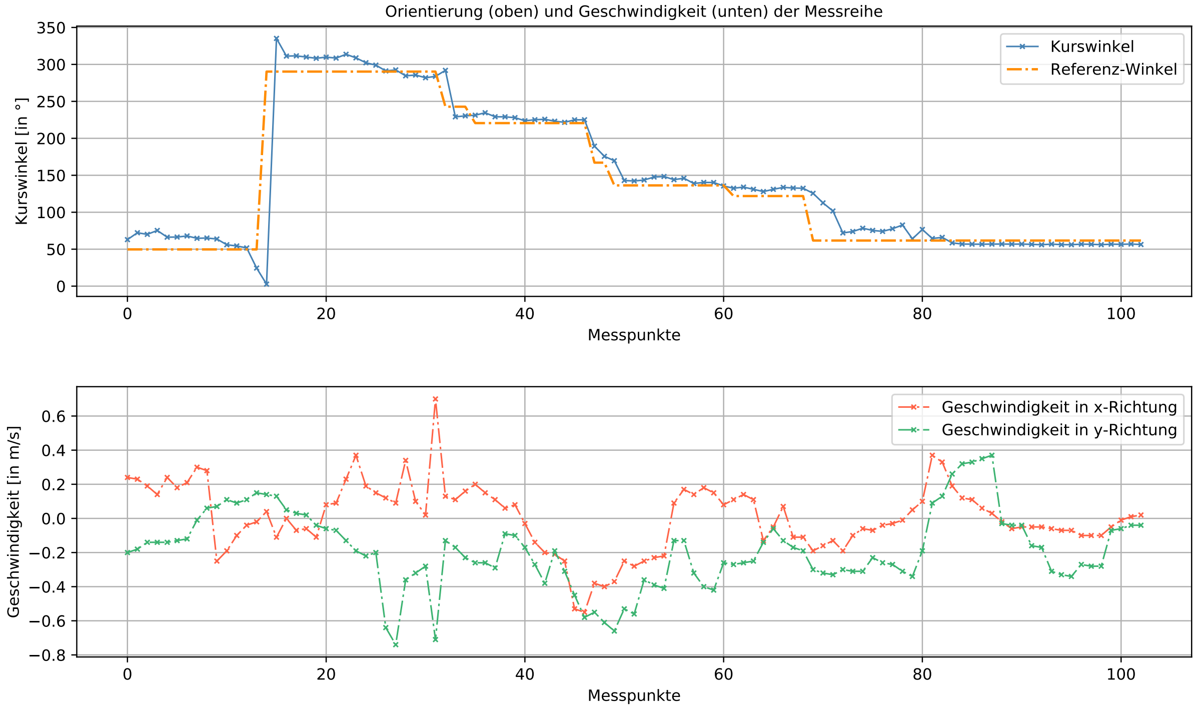


Abbildung 4.4: Kurswinkel und Geschwindigkeit der Testanwendung
Eigene Darstellung

Das Untere der beiden Diagramme zeigt die berechneten Geschwindigkeiten des Systems während der Testanwendung. In rot ist die Geschwindigkeit in x-Richtung dargestellt und in grün die in y-Richtung. Da die IMU die Beschleunigungen im Sensor eigenen Koordinatensystem misst, wurden diese mit (3.5) aus Kapitel 3.3.3, transformiert. Somit entspricht der rote Graph der Geschwindigkeit in Ost-Richtung und der grüne Graph der Geschwindigkeit in Nord-Richtung. Zu den Geschwindigkeiten des Systems existieren keine Referenzwerte.

In einem weiteren Test wurde die Genauigkeit des Gyroskops gemessen, welches für dieses

System entwickelt wurde. Es ermöglicht die Neigungskompensationen des Magnetometers und der Beschleunigungen in x-, y- und z-Richtung. Abbildung 4.5 zeigt zwei Diagramme. Im Oberen befindet sich das Messergebnis des Neigungswinkels und im Unteren das des Rollwinkels. Vor und nach dem Messen jedes Winkels wird das System auf eine Ebene ohne Neigung platziert. Für jeden Winkel wird in Tabelle 4.4 die minimale, maximale und mittlere Differenz aus dem gemessenen Winkel und dem Referenzwinkel aufgeführt. Die Werte zwischen den einzelnen Winkeln wurden für die Berechnung der Differenz nicht miteinbezogen.

Pitch	Referenzwert	$\Delta\alpha_{min}$	$\Delta\alpha_{max}$	$\overline{\Delta\alpha}$
Ebene	0.00 °	0.01 °	1.94 °	0.17 °
1.	7.40 °	0.00 °	0.20 °	0.09 °
2.	16.00 °	0.11 °	0.56 °	0.26 °
3.	24.50 °	0.05 °	0.23 °	0.13 °
4.	−7.40 °	0.00 °	0.29 °	0.15 °
5.	−16.00 °	0.01 °	1.77 °	0.36 °
6.	−24.50 °	1.22 °	1.49 °	1.35 °
Roll	-	-	-	-
Ebene	0.00 °	0.00 °	0.44 °	0.06 °
1.	−11.30 °	0.03 °	1.49 °	0.27 °
2.	−23.50 °	0.01 °	1.30 °	0.31 °
3.	−35.80 °	0.10 °	0.88 °	0.25 °
4.	11.30 °	0.40 °	2.45 °	1.62 °
5.	23.50 °	0.23 °	1.82 °	0.92 °
6.	35.80 °	0.06 °	3.68 °	0.81 °

Tabelle 4.4: Differenzen der gemessenen Werte und Referenzwerte des Gyroskops
Eigene Darstellung

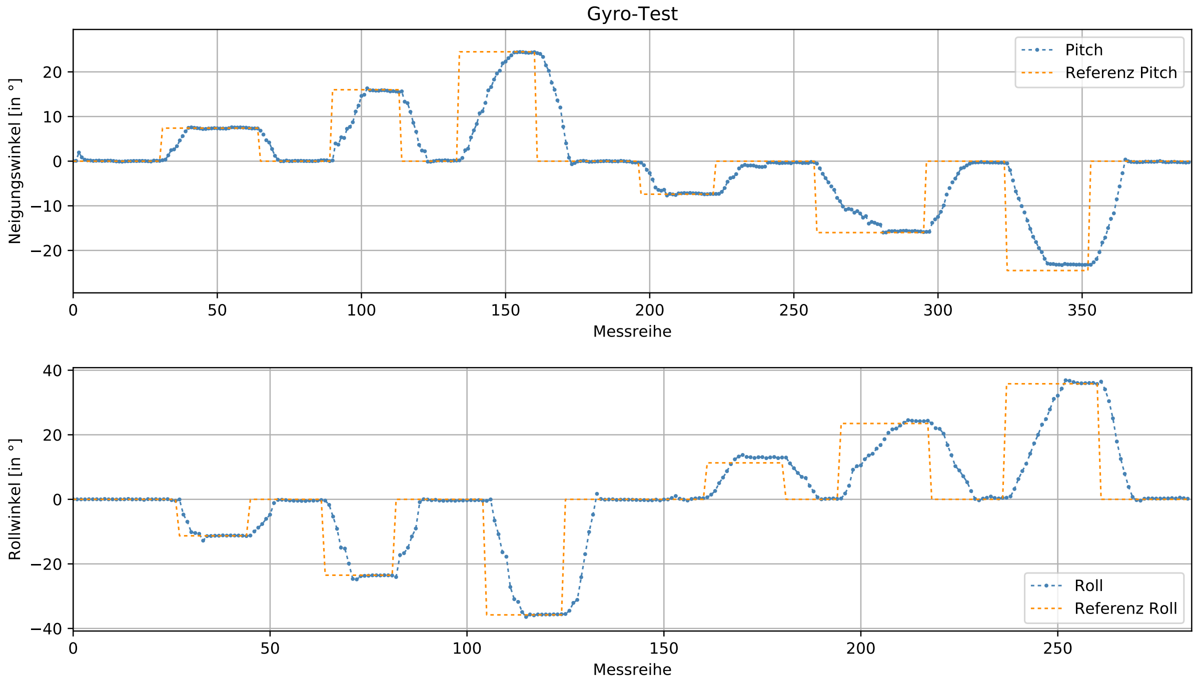


Abbildung 4.5: Test des entwickelten Gyroskops
Eigene Darstellung

5 Fazit und Ausblick

Im Folgenden wird ein grober Überblick über die Ergebnisse der einzelnen Kapitel gegeben. Anschließend erfolgt eine kritische Betrachtung der Ergebnisse aus Kapitel 4 sowie eine Beschreibung der Beschränkungen des Systems, sowie Fehlstellen und ein Ausblick auf mögliche Erweiterungen.

5.1 Überblick

Das Ziel dieser Arbeit ist es, ein System zur *Indoor*-Positionierung zu entwickeln, um die Position von diversen autonomen Systemen zu bestimmen. Das System beinhaltet einen Raspberry Pi und einen Arduino Uno sowie *Low-cost* Messsensorik, die zu einer IMU kombiniert sind. Die IMU liefert Informationen über die aktuelle Lage im Raum sowie die aktuelle Orientierung und Beschleunigungen, die entstehen wenn externe Kräfte auf das System einwirken. Mit Hilfe des Kalman-Filters ergibt sich aus diesen Daten und *Bluetooth*-Signalenstärken von BLE *Beacon* die aktuelle dreidimensionale Position des zu ortenden Objekts. Für das System werden kostengünstige und energieeffiziente Technologien verwendet.

Für die Entwicklung des Systems wurden zu Beginn Recherchen zu bereits existierenden *Indoor*-Positionierungssystemen durchgeführt. Technologien, die für eine Anwendung in dieser Arbeit in Frage kommen, wurden näher beleuchtet und kritisch betrachtet. Die Auswahl der Technologien fiel auf eine Kombination aus IMU und Messung von empfangenen Signalstärkemessungen von BLE *Beacon*.

Um das genannte Ziel zu erreichen, wurde eine IMU entwickelt. Die zu der Messeinheit kombinierten Sensoren sind ein Beschleunigungssensor, ein Gyroskop und ein Magnetometer. Für jeden Sensor existieren vom Hersteller geschriebene Bibliotheken. Da eine Anwendung auf autonom fahrenden Systemen Vibrationen verursacht, müssen die Sensorwerte der fertigen Bibliotheken getestet werden, um die Nutzbarkeit zu gewährleisten. Außerdem geschieht die Ortung über einen längeren Zeitraum, was ein Driften der Sensorwerte verursachen kann. Dies muss ebenfalls für die zu gewährleistende Nutzbarkeit beachtet werden. Die resultierenden Werte waren für Vibrationen anfällig und drifteten außerdem mit zunehmender Messdauer ab. Aus diesem Grund wurde eine eigene IMU entwickelt, welche mit Hilfe von Sensorfusion und Filterung verlässliche Informationen über den aktuellen Systemzustand lieferte. Zu diesem Zweck wurden zunächst alle Sensoren kalibriert. Der Beschleunigungs- und Magnetfeldsensor wurde mit einem Referenzellipsoiden und der *Magneto* (v1.2) Software kalibriert. Für das Gyroskop wurden die *Offsets* jeder Achse durch Messungen bestimmt, bei denen der Sensor nicht rotiert oder bewegt wurde. Jeder Sensor erhielt über die entsprechenden Register Einstellungen bezüglich der Updaterate und der Messskala. Die übermittelten Rohwerte der einzelnen Sensoren wurden mit einem Kalman-Filter fusioniert und gefiltert. Die Sensorfusion des Beschleunigungssensors und des Gyroskops ergab eine zuverlässige Einheit zur Bestimmung des Neigungs- und Rollwinkels. Diese weisen keinen Drift auf und sind unempfindlicher gegenüber Vibrationen, die zum Beispiel durch Motoren entstehen. Die Kombination des neuen Gyroskops mit dem Magnetometer ergab einen neigungskompensierten Kompass. Durch eine Sensorfusion der Drehrate des Gyroskops um die z-Achse und dem Kurswinkel des Kompass wurde die Störanfälligkeit des Magnetometers gegenüber externen Magnetfeldern verringert. Das neue Gyroskop ermöglichte zudem eine Gravitationskompensation der gemessenen Beschleunigungswerte. Somit sind Messungen von Beschleunigungen in allen drei Raumrichtungen möglich. Die Erdbeschleunigung wird automatisch durch die vom Gyroskop bestimme Lage im Raum herausgerechnet.

Die zweite Komponente zur Bestimmung der Position stellt die BLE Technologie dar. Mit Hilfe der Trilateration wird die Position des Systems in der xy-Ebene bestimmt. Dafür sind drei BLE *Beacon* notwendig, wobei dem System die Position jedes Ankerpunkts bekannt sein muss. Das zu ortende Objekt misst die empfangene Signalstärke und ist dadurch in der Lage, die Entfernung zum jeweiligen *Beacon* zu berechnen. Aus diesem Grund wurde der Zusammenhang zwischen Entfernung und Signalstärke untersucht. Auf einer Teststrecke von fünf Metern wurden die Signalstärken in Abständen von 20 *cm* gemessen. Anschließend wurde eine polynomiale und logarithmische Regression zur Modellierung des Zusammenhangs durchgeführt. Die logarithmische Regression ergab ein besseres Modell, weshalb im weiteren Verlauf dieses zur Berechnung der Entfernung aus den Signalstärken verwendet wurde. Um dem System die Position jedes *Beacon* mitzuteilen, wurde eine einfache *Client-Server* Architektur entwickelt. Mit dem *Django Web framework* und der *Django REST API* wurde ein Webserver aufgesetzt, über dessen *Interface* die Koordinaten der *Beacon* eingetragen wurde. Beim Einschalten des Systems und Scannen nach *Beacon* sendet das System automatisch einen *Request* an die URL des genannten Servers .Jeder *Beacon* übermittelt eine eigene URL und erhält als Antwort die Position jedes Ankerpunkts.

Nachdem die bisherigen Technologien und Methoden umgesetzt wurden, war es möglich die endgültige Position in der xy-Ebene durch eine Sensorfusion aus den Messergebnissen der IMU und der Trilateraion zu bestimmen. Die zur dreidimensionalen Position fehlende Höhe wurde ebenfalls durch eine Sensorfusion berechnet. Neben den Messwerten der IMU wurde dafür ein Barometer verwendet. In einer anschließenden Testanwendung wurde das System auf einen *Arduino Robot* montiert. Der Roboter folgte im Test einer zuvor eingemessenen Linie und speicherte die berechneten Positionen. Diese wurden im Nachhinein in einer Abbildung dargestellt, um die gemessenen und die realen Positionen zu vergleichen. Zusätzlich zeichnete das System während des Testdurchlaufs Informationen der IMU über die aus der Beschleunigung integrierten Geschwindigkeit und dem Kurswinkel auf. Die Genauigkeit des Neigungs- und Rollwinkels wurde mit einem digitalen Winkel- und Neigungsmesser überprüft. Ein Test auf fünf Ebenen mit verschieden Höhen ergab Auskunft über die Genauigkeit der Höhenberechnung des Systems.

Für die gesamte Entwicklung des Systems wurde zuvor ein Konzept erstellt, in welchem sich auch Informationen über die Zeitplanung befinden. Die verschiedenen Phasen wurden während der Umsetzung grob eingehalten, lediglich die Zusammenführung aller Sensordaten auf dem Raspberry Pi nahm mehr Zeit in Anspruch als geplant. Durch den früheren Beginn der Schreibphase konnte wieder etwas Zeit gewonnen werden. Der Meilenstein der Fertigstellung der gesamten Masterarbeit wurde eine Woche später als ursprünglich geplant erreicht.

5.2 Diskussion der Ergebnisse

Die Testanwendung des Systems auf dem Bewegungssimulator ergab eine Genauigkeit von etwa 10 *cm* bis 150 *cm*. Eine genauere Angabe über die Exaktheit der Messergebnisse lässt der Versuchsaufbau aufgrund eines fehlenden Tachymeters nicht zu. Den aufgezeichneten

Positionen des Systems ließ sich kein genauer Ort auf der Teststrecke zuordnen. Eine Zuordnung ist lediglich grob möglich. Aus der Abbildung 4.1 wird deutlich, dass die Genauigkeit der berechneten Position auf den Strecken zwischen den Stützpunkten eins und fünf am geringsten ist. Eine mögliche Ursache dafür ist der *Multipath*-Effekt, welcher bereits bei der Untersuchung des Zusammenhangs zwischen Signalstärke und Distanz zur Signalquelle in Abbildung 3.11 verdeutlicht wurde. Bei dem Versuchsaufbau befanden sich die *Beacon* bei Stützpunkt fünf und zwei in etwa einem halben Meter Abstand zur Wand. Empfangene *Multipath*-Signale in diesen Bereichen legen dadurch eine Strecke zurück, die etwa einen Meter länger ist als Signale in den anderen Bereichen.

Rückschlüsse auf den Einfluss von *Multipath*-Effekten bei der Trilateration können durch Abbildung 4.2 gezogen werden. Am Stützpunkt sieben verteilen sich die Punktmessungen um ihren Referenzpunkt (Referenzpunkt drei, in grün). Die anderen beiden Punktmessungen zeigen eine breitere Verteilung der Messungen und der Referenzpunkt befindet sich nicht nahe des Zentrums der Verteilung. Dies wird vor allem bei der zweiten Punktemessung (in blau) deutlich. Der Algorithmus zur Berechnung der Trilateration geht davon aus, dass die Messung des *Beacon* mit der geringsten Entfernung am genausten ist. Dies beruht auf der RSSI-Entfernungsmodellierung. Werden diese Werte jedoch durch *Multipath*-Effekte beeinflusst, sinkt die Genauigkeit.

Während der Testanwendung wurden ebenfalls Daten zum Kurswinkel und der Geschwindigkeit aufgezeichnet (Abbildung 4.4). Der *Arduino Robot* besitzt fünf Sensoren um eine Linie zu erkennen und dieser folgen zu können. Er fährt die Linie jedoch nicht exakt gerade ab. Während der Fahrt pendelt er immer wieder von der einen zur anderen Seite. Diese Bewegungen sind in den Aufzeichnungen des Kurswinkels zu erkennen. Besonders deutlich wird dieses „Finden“ der Spur, wenn es zu einer Richtungsänderung kommt. Zu sehen ist dies unter anderem am Stützpunkt sieben. Dieser entspricht dem Bereich der Messreihe zwischen 60 und 80 in Abbildung 4.4. Bei jedem Richtungswechsel dreht sich der Roboter auf der Stelle. Seine Geschwindigkeit beträgt somit etwa Null. Im unteren Diagramm der Abbildung 4.4 ist dies jedoch nur sehr bedingt erkennbar. Da die Geschwindigkeit aus den Beschleunigungen integriert wird, wird auch jeder Fehler mit integriert. Durch das Fehlen eines Sensors zur Bestimmung der aktuellen Geschwindigkeit hat das Kalman-Filter keine Möglichkeit, den geschätzten Zustandswert zu korrigieren. Dennoch lassen sich grobe Zusammenhänge erkennen. So nähert sich die Geschwindigkeit in x und y kurz vor Messpunkt 20 Null an. Im oberen Diagramm ist eine Richtungsänderung zu erkennen, bei der die Geschwindigkeit des Roboters wie bereits erwähnt Null ist. Auch am Ende der Messreihe, als der Arduino sich nicht mehr bewegt, nähern sich die beiden Geschwindigkeiten der Null an.

Auf ähnliche Art und Weise wie die Position in xy- wurde die Position in z-Richtung bestimmt. Die Höhe ergibt sich aus der Sensorfusion der Daten der IMU und des Barometers. Die Ergebnisse des Tests, bei dem das System nacheinander auf verschiedene Ebenen mit bekannter Höhe gebracht wurde, sind in Abbildung 4.3 dargestellt. Die einzelnen Ebenen sind in den Daten erkennbar. Die Höhenwerte schwanken dennoch um durchschnittlich $9.2\,cm$. Der Versuch wurde in einem geschlossenen Raum mit konstantem Luftdruck durchgeführt. Das Öffnen einer Tür, eines Fensters oder eine Veränderung der Wetterlage hätten das Ergebnis beeinflussen können. Bei schwankendem Luftdruck variieren die Messungen stärker und mit zunehmender Messdauer würde auch die Messungenauigkeit steigen. Eine zusätzliche Referenzstation am Boden, welche Informationen über den Luftdruck auf Ausgangshöhe gibt, könnte die Messungenauigkeit bei Änderung des Luftdrucks verringern.

In einem letzten Experiment wurde die Genauigkeit des Neigungs- und Rollwinkels der IMU getestet. Die Ergebnisse werden in Abbildung 4.5 dargestellt. Die Differenz zwischen den Referenzwinkeln und den gemessenen betrug teilweise unter einem halben Grad. Einige der Messungenauigkeiten sind vermutlich auch auf eine nicht exakt ruhige Hand der Person, welche das System von Hand auf den entsprechenden Winkel gebracht hat, zurückzuführen.

5.3 Beschränkungen und Erweiterungen des Systems

Es wurde ein System entwickelt, mit dem die dreidimensionale Position eines Objekts im *Indoor*-Bereich bestimmt werden kann. Dieses System verwendet kostengünstige und energieeffiziente Komponenten, deren Genauigkeit durch Sensordatenfusion optimiert wurde. Die Genauigkeit des Systems befindet sich dabei im Meterbereich. Das Experiment zur Messung der Genauigkeit fand nicht unter Laborbedingungen statt, sondern wie bereits beschrieben unter realen Bedingungen in einem möblierten Zimmer. Um eine zuverlässigere Aussage über die Messgenauigkeit treffen zu können, muss der Versuch in verschiedenen Umgebungen durchgeführt werden. Bislang konnte nicht genau geklärt werden, welchen Einfluss die Umgebung auf das entwickelte Positionierungssystem hat. Kleine Räume könnten den *Multipath*-Effekt ebenso verstärken wie beispielsweise Möbel. Dabei spielt nicht nur die Grundfläche eine Rolle, sondern auch die Deckenhöhe. Dieses Problem ließe sich eventuell durch eine Anpassung der Sendeleistung der Bluetooth-Signale lösen. Ebenfalls ist der Einfluss der Verteilung der BLE *Beacon* unbekannt. Im Experiment wurde darauf geachtet, alle *Beacon* möglichst gleichmäßig im Raum zu verteilen. Aus den Resultaten des Versuchs kann der Einfluss der Konstellation der *Beacon* auf die Genauigkeit nicht geklärt werden. Dies gilt nicht nur für die Lage der *Beacon* zueinander, sondern auch für deren Position im Raum. So könnte es möglich sein, dass der *Multipath*-Effekt gemindert wird, wenn sich die *Beacon* eher in der Mitte des Raums befinden als direkt an den Wänden oder in den Ecken. Die durch das Experiment ermittelte Genauigkeit des Positionierungssystems bezieht sich demnach nur auf die Umgebung, in welcher der Versuch durchgeführt wurde. Zusätzlich kann keine verlässliche Aussage darüber getroffen werden, wie sich die Genauigkeit ändert, wenn die Sendeleistung der *Beacon* verändert wird. Aufgrund der vielen unbekannten Faktoren wird die Annahme getroffen, dass die Genauigkeit des hier entwickelten *Indoor*-Positionierungssystems noch gesteigert werden kann, wenn der Einfluss der Umgebung und der Position der *Beacon* näher untersucht wird.

Um die Position aus der Trilateration der Signalstärkemessungen durch Sensoren zu verbessern, wurde viel Wert auf die Genauigkeit der entwickelten IMU gelegt. Der Messfehler der endgültigen Position kann durch den Einbau zusätzlicher Messeinheiten, zum Beispiel zum Messen der Geschwindigkeit, verringert werden. Um die Flexibilität und die Erweiterbarkeit des Systems zu gewährleisten, wurde der Code in objektorientierter Programmierung mit Python geschrieben. Es ist demnach möglich, weitere Klassen zu implementieren und die verwendeten Kalman-Filter anzupassen. Ebenfalls können weitere *Beacon* über das *Interface* des *Django*-Adminbereichs hinzugefügt werden. Diesbezüglich muss der Algorithmus im Quellcode der Trilateration angepasst werden, um eine Multilateration zu ermöglichen. Demnach kann das hier entwickelte Positionierungssystem durch die genannten Möglichkeiten ohne großen Aufwand weiterentwickelt werden, um so die Genauigkeit zu erhöhen.

Anhang

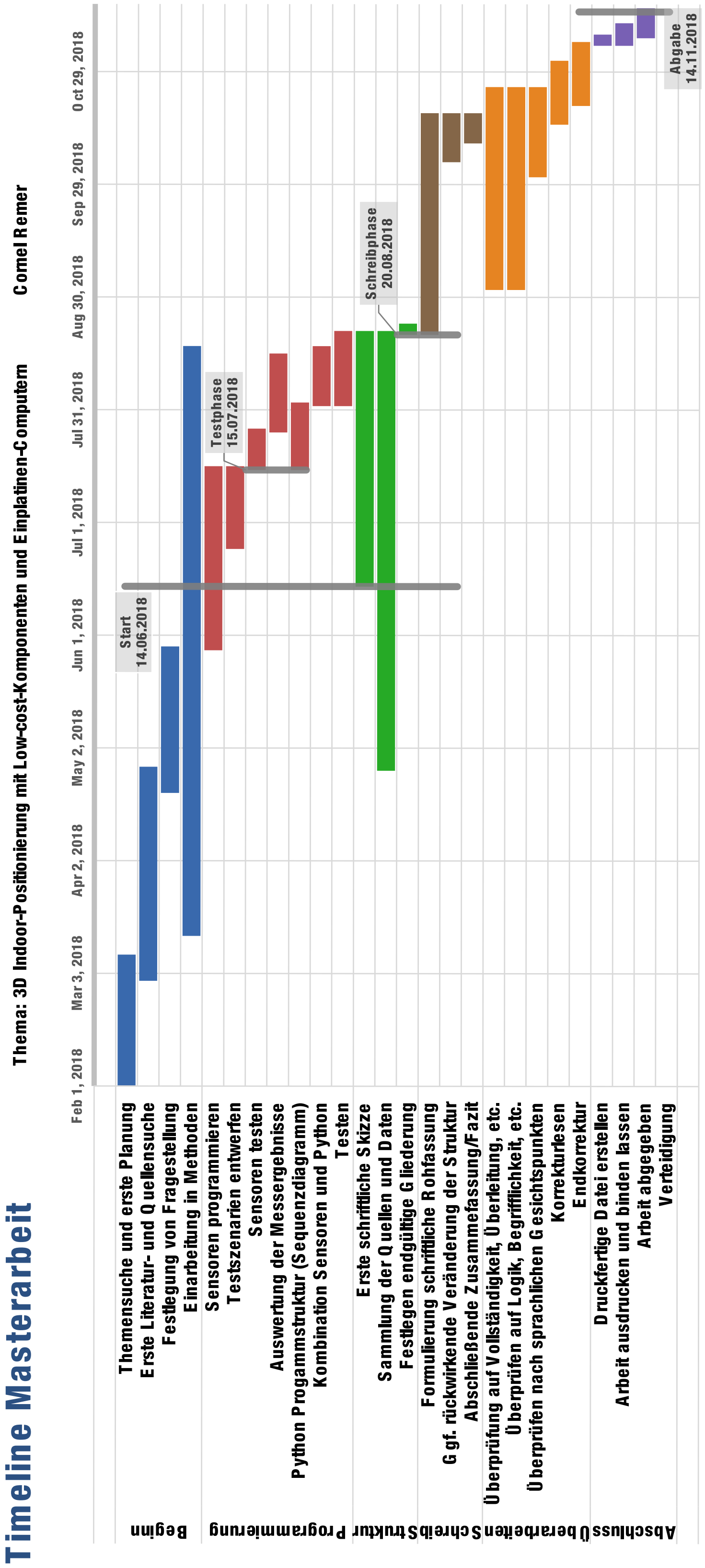
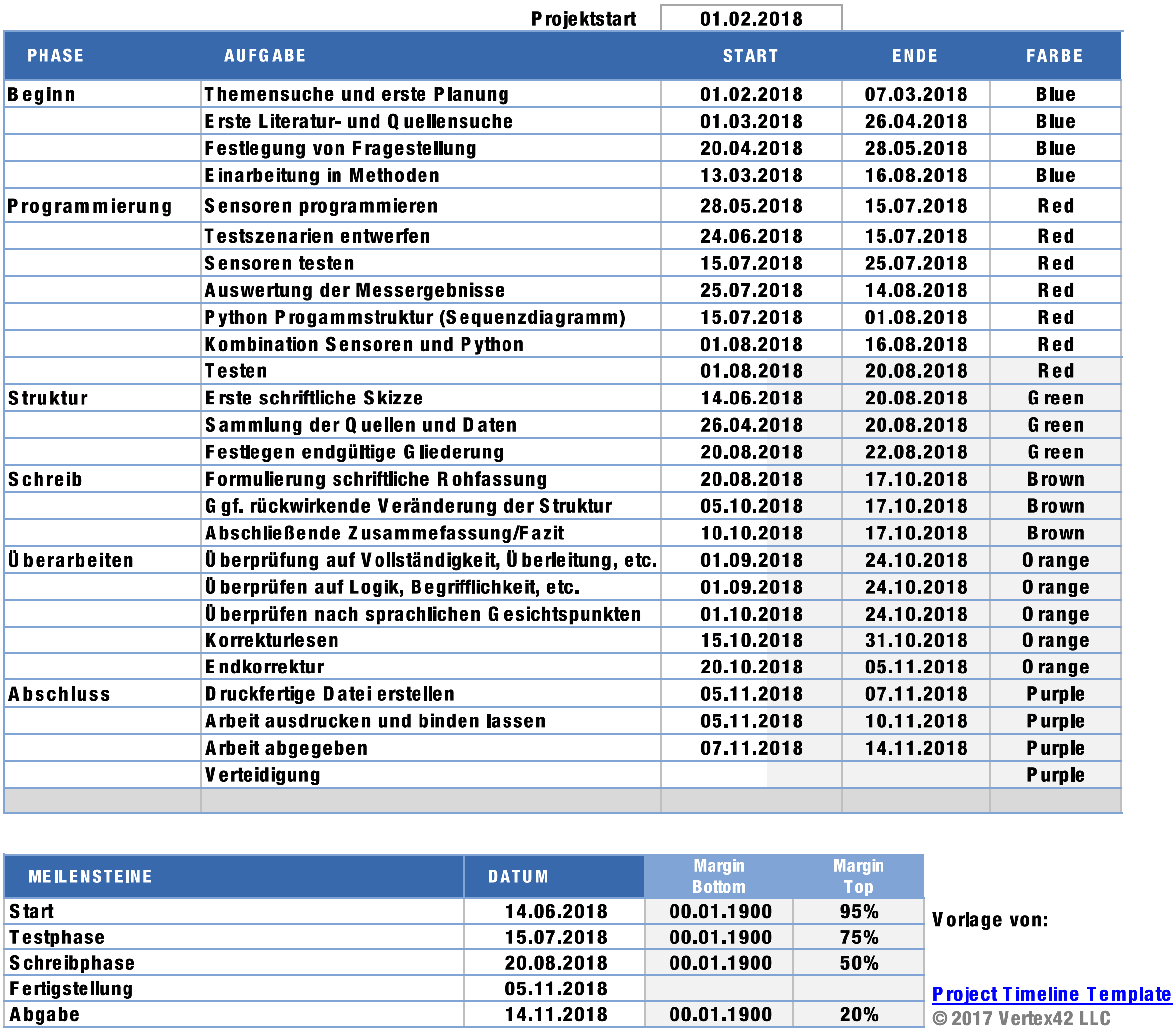


Abbildung zu Kapitel 3.2: Zeitplanung der Masterarbeit
Eigene Darstellung



Projektstart 01.02.2018

PHASE	AUFGABE	START	ENDE	FARBE
Beginn	Themensuche und erste Planung	01.02.2018	07.03.2018	Blue
	Erste Literatur- und Quellensuche	01.03.2018	26.04.2018	Blue
	Festlegung von Fragestellung	20.04.2018	28.05.2018	Blue
	Einarbeitung in Methoden	13.03.2018	16.08.2018	Blue
Programmierung	Sensoren programmieren	28.05.2018	15.07.2018	Red
	Testszenarien entwerfen	24.06.2018	15.07.2018	Red
	Sensoren testen	15.07.2018	25.07.2018	Red
	Auswertung der Messergebnisse	25.07.2018	14.08.2018	Red
	Python Progammstruktur (Sequenzdiagramm)	15.07.2018	01.08.2018	Red
	Kombination Sensoren und Python	01.08.2018	16.08.2018	Red
	Testen	01.08.2018	20.08.2018	Red
Struktur	Erste schriftliche Skizze	14.06.2018	20.08.2018	Green
	Sammlung der Quellen und Daten	26.04.2018	20.08.2018	Green
	Festlegen endgültige Gliederung	20.08.2018	22.08.2018	Green
Schreib	Formulierung schriftliche Rohfassung	20.08.2018	17.10.2018	Brown
	Ggf. rückwirkende Veränderung der Struktur	05.10.2018	17.10.2018	Brown
	Abschließende Zusammefassung/Fazit	10.10.2018	17.10.2018	Brown
Überarbeiten	Überprüfung auf Vollständigkeit, Überleitung, etc.	01.09.2018	24.10.2018	Orange
	Überprüfen auf Logik, Begrifflichkeit, etc.	01.09.2018	24.10.2018	Orange
	Überprüfen nach sprachlichen Gesichtspunkten	01.10.2018	24.10.2018	Orange
	Korrekturlesen	15.10.2018	31.10.2018	Orange
	Endkorrektur	20.10.2018	05.11.2018	Orange
Abschluss	Druckfertige Datei erstellen	05.11.2018	07.11.2018	Purple
	Arbeit ausdrucken und binden lassen	05.11.2018	10.11.2018	Purple
	Arbeit abgegeben	07.11.2018	14.11.2018	Purple
	Verteidigung			Purple

MEILENSTEINE	DATUM	Margin Bottom	Margin Top
Start	14.06.2018	00.01.1900	95%
Testphase	15.07.2018	00.01.1900	75%
Schreibphase	20.08.2018	00.01.1900	50%
Fertigstellung	05.11.2018		
Abgabe	14.11.2018	00.01.1900	20%

Vorlage von:

Project Timeline Template

Abbildung zu Kapitel 3.2: Erläuterung zur Zeitplanung der Masterarbeit
Eigene Darstellung

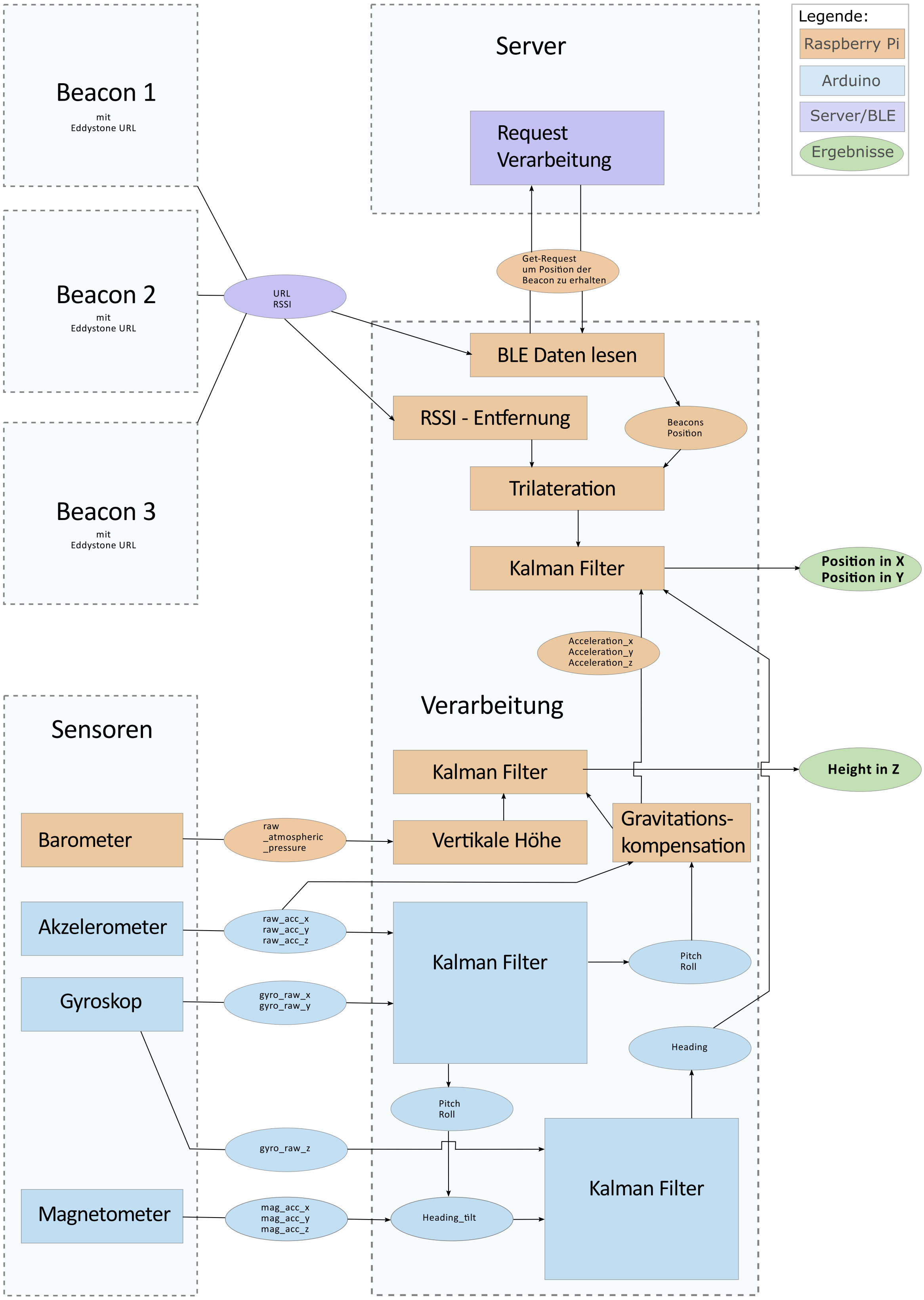


Abbildung zu Kapitel 3.2: Konzept-Grafik des gesamten Systems
Eigene Darstellung

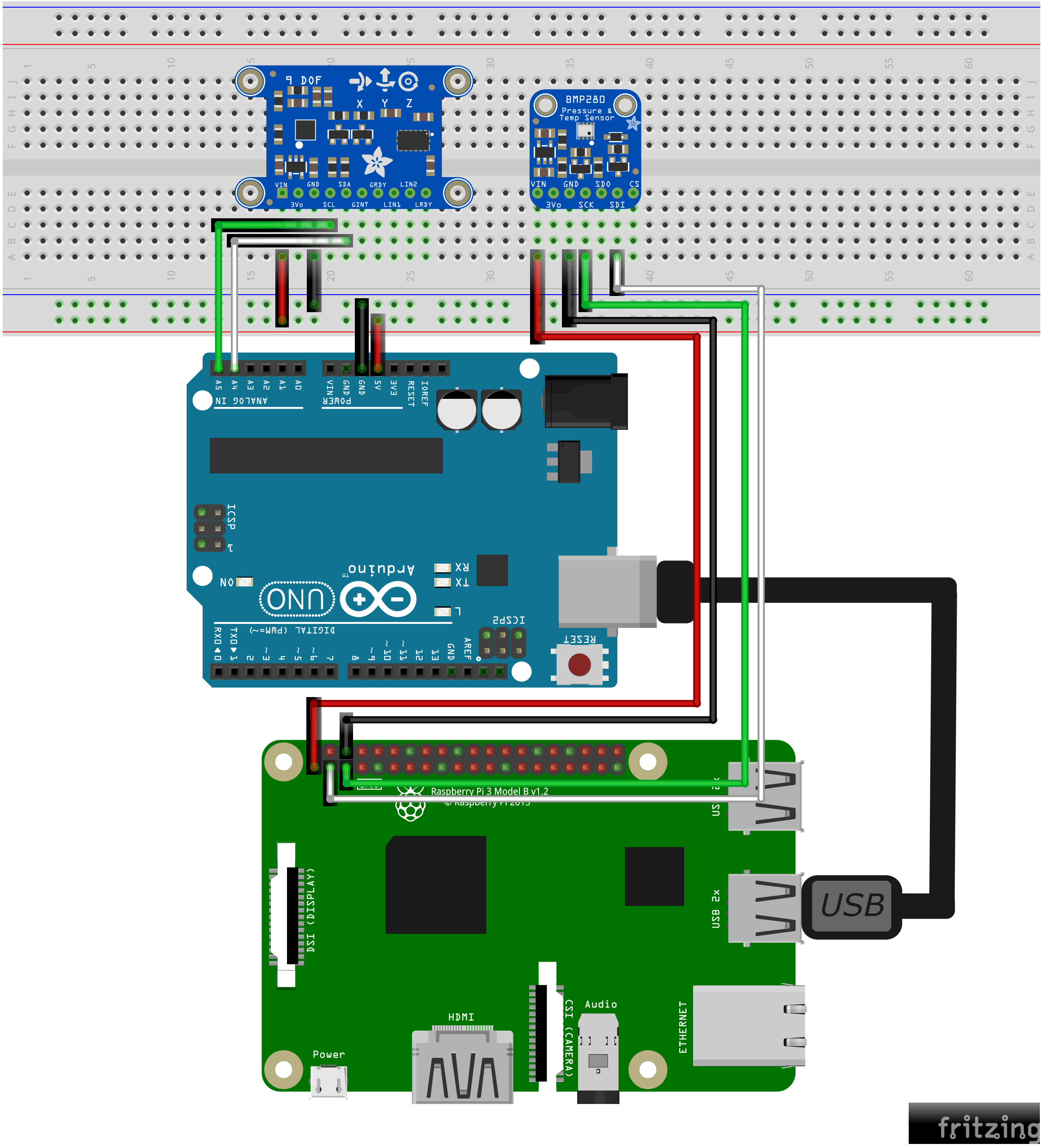


Abbildung zu Kapitel 3.2: Fritzing-Grafik aller Komponenten
Eigene Darstellung

Literaturverzeichnis

Accent Systems (o.J.). *What is a Beacon?*. Abgerufen von https://accent-systems.com/beacons [Stand 20.09.2018].

Accent Systems (2016). *iBKS105 Datasheet.* Abgerufen von http://www.accent-systems.com/wp-content/uploads/iBKS105_datasheet_rev2.pdf#%5B%7B%22num%22%3A46%2C%22gen%22%3A0%7D%2C%7B%22name%22%3A%22XYZ%22%7D%2C82%2C771%2C0%5D [Stand 20.09.2018].

Arduino (o.J.). *Arduino Uno REV3.* Abgerufen von https://store.arduino.cc/arduino-uno-rev3 [Stand: 28.05.2018].

Bac STI 2D (o.J.). *Communications informatiques: 3. Exemple N 2 : le bus I2C.* Abgerufen von https://sti2d.ecolelamache.org/iii_communications_informatiques___3_exemple_n2_le_bus_i2c.html [Stand: 03.05.2018].

Babb, T. (2015). *How a Kalman Filter works, in pictures.* Abgerufen von http://www.bzarg.com/p/how-a-kalman-filter-works-in-pictures/ [Stand 01.07.2018].

Bartelmann, M., Feuerbacher, B., Krueger, T., Luest, D., Rebhan, A., & Wipf, A. (2018) *Theoretische Physik 1 | Mechanik*, Berlin: Springer Spektrum.

Bislin, W. (2017). *Schnittpunkte zweier Kreise berechnen (JavaScript).* Abgerufen von http://walter.bislins.ch/blog/index.asp?page=Schnittpunkte+zweier+Kreise+berechnen+%28JavaScript%29 [Stand 13.10.2018].

Buchman, E. (o.J.). *I2C-Ansteuerung.* Abgerufen von https://www.ipd.kit.edu/mitarbeiter/buchmann/microcontroller/i2c.htm [Stand 05.05.2018].

Burg, K., Haf, H., & Wille, F. (1987). *Hoehere Mathematik fuer Ingenieure Band II Lineare Algebra*, Stuttgart: Vieweg + Teubner Verlag.

Ekbert, H., & Gert, S. (2018). *Sensoren in Wissenschaft und Technik: Funktionsweise und Einsatzgebiete*, Wiesbaden: Vieweg + Teubner Verlag.

Elektronik Kompendium (o.J.). *MEMS - Micro-Electro-Mechanical Systems.* Abgerufen von https://www.elektronik-kompendium.de/sites/bau/1503041.htm [Stand 02.06.2018]

Er Rida, M.,Liu, F., Jadi, Y., Ali Abdullah Algawhari, A., & Askourih, A. (2015). Indoor Location Position Based on Bluetooth Signal Strength. *2015 2nd International Conference on Information Science and Control Engineering*, ICISCE.2015, 177.

Gaudlitz, E. (2016): *Internet of things and Indoor Positioning* Abgerufen von https://www.infsoft.com/blog-en/articleid/135/internet-of-things-and-indoor-positioning [Stand 05.11.2018].

Gevatter, H. J. (2006). *Handbuch der Mess- und Automatisierungstechnik in der Produktion*, Berlin: Springer-Verlag Berlin Heidelberg.

Google Beacon Platform (2018.). *Eddystone format.* Abgerufen von https://developers.google.com/beacons/eddystone [Stand 25.09.2018].

Google Beacon Platform (o.J.). *Mark up the world using beacons.* Abgerufen von https://developers.google.com/beacons/ [Stand 25.09.2018].

Heintze, J. (2014). *Lehrbuch zur Experimentalphysik Band 1: Mechanik*, Berlin: Springer Spektrum.

Hering, E., & Schönfelder, G. (2018). *Sensoren in Wissenschaft und Technik: Funktionsweise und Einsatzgebiete*, Wiesbaden: Vieweg + Teubner Verlag

Hesse, S., & Schnell, G. (2014). *Sensoren für die Prozess- und Fabrikautomation: Funktion - Ausführung - Anwendung*, Wiesbaden: Springer Verlag.

I2C Bus (o.J.). *Was ist der I2C-Bus?*. Abgerufen von https://de.i2c-bus.org/ [Stand 03.05.2018].

ITWissen (2018). *Iot (Internet of things).* Abgerufen von https://www.itwissen.info/Internet-of-things-IoT-Internet-der-Dinge.html [Stand 20.09.2018].

Kaergaard, M. B., Blunk, H., Godsk, T., Toftkjaer, T., Christensen, D. L., & Grobnbaek, K. (2010). Indoor Positioning Using GPS Revisited. *Pervasive Computing*, Berlin, Heidelberg,:Springer Berlin Heidelberg, 38-56.

Kostiainen, A., & Bhaumik, R. (2018). *Magnetometer.* Abgerufen von https://www.w3.org/TR/magnetometer/ [Stand 19.06.2018]

Ksentini, D., Elhadi, A. R., & Lasla, N. (2014). Inertial Measurement Unit: Evaluation for Indoor Positioning. *2014 International Conference on Advanced Networking Distributed Systems and Applications*, 25-30.

Li, J., Yue, X., Chen, J., & Deng, F. (2017). A Novel Robust Trilateration Method Applied to Ultra-Wide Bandwidth Location Systems. *Sensors*, 17(4), 795.

Li, Q., & Griffiths, G. J. (2004). Least squares ellipsoid specific fitting. *Conference: Proceedings - Geometric Modeling and Processing 2004*. 2004. 335- 340.

Marchthaler, R., & Dingler, S. (2017). *Kalman-Filter Einführung in die Zustandsschätzung und ihre Anwendung für eingebettete Systeme.* Wiesbaden: Springer Verlag.

Mautz, R. (2012). Indoor Positioning Technologies, *ETH Zurich, Department of Civil, Environmental and Geomatic Engineering, Institute of Geodesy and Photogrammetry.*

Merlin, B. (2011). *Improved magnetometer calibration (Part 2).* Abgerufen von http://sailboatinstruments.blogspot.com/2011/09/improved-magnetometer-calibration-part.html [Stand 10.10.2018].

Mescheder, U. (2000). *Mikrosystemtechnik Konzepte und Anwendungen*, Leipzig:Vieweg + Teubner Verlag | Springer.

o.V. (2004). *1 The Discrete Kalman Filter.* Abgerufen von http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/WELCH/kalman.1.html [Stand 03.07.2018]

Pedley, M. (2013). *Tilt Sensing Using a Three-Axis Accelerometer* Abgerufen von https://cache.freescale.com/files/sensors/doc/app_note/AN3461.pdf [Stand 11.06.2018].

Pololu (o.J.). *Using LSM303DLH for a tilt compensated electronic compass* Abgerufen von https://www.pololu.com/file/0J434/LSM303DLH-compass-app-note.pdf [Stand 20.06.2018].

Raspberry Pi Foundation (o.J.a). *Raspberry Pi 3 Model B.* Abgerufen von https://www.raspberrypi.org/products/raspberry-pi-3-model-b/ [Stand 27.05.2018].

Raspberry Pi Foundation (o.J.b). *What is a Raspberry Pi?.* Abgerufen von https://www.raspberrypi.org/help/what-%20is-a-raspberry-pi/ [Stand 27.05.2018].

Retscher, G. (2016). Indoor Navigation. *Encyclopedia of Geodesy*, 1-7.

Retscher, G., & Kistenich, M. (2006). Vergleich von Systemen zur Positionsbestimmung und Navigation in Gebäuden. *zfv - Zeitschrift für Geodäsie, Geoinformation und Landmanagement*, 1/2006, 25.35.

Retscher, G., & Moser, E. (2007). Genauigkeits- und Leistungstest eines WLAN Indoor Positionierungssystems. *zfv - Zeitschrift für Geodäsie, Geoinformation und Landmanagement*, 1/2007, 4-10.

Retscher, G., & Tatschl, T. (2017). Positionierung in Gebäuden mit differenziellem WLAN. *zfv - Zeitschrift für Geodäsie, Geoinformation und Landmanagement*, 2/2017, 111-125.

Ruckstuhl, A., & Stahel, W. (2008). Nichtlineare Regression, *ETH Zuerich - Department of Mathematics.*

Spektrum (o.J.). *Piezoelektrischer Effekt.* Abgerufen von https://www.spektrum.de/lexikon/physik/piezoelektrischer-effekt/11270 [Stand: 20.05.2018].

Sprachassistenten News (o.J.). *Amazon Alexa - Umfassende Liste aller Sprachbefehle.* Abgerufen von https://sprachassistenten-news.de/amazon-alexa-liste-aller-sprachbefehle/

[Stand 05.11.2018].

Stelzer, A., Pourvoyeur, K., & Fischer, A. (2004). Concept and application of LPM - a novel 3-D local position measurement system. *IEEE Transactions on Microwave Theory and Techniques*, 52(12). 2664-2669.

STlife.augmented (2013a). *L3GD20-Datasheet.* Abgerufen von https://www.st.com/resource/en/datasheet/l3gd20.pdf [Stand 05.06.2018].

STlife.augmented (2013b). *LSM303DLHC-Datasheet.* Abgerufen von https://www.st.com/resource/en/datasheet/DM00027543.pdf [Stand 05.06.2018].

STlife TotalPhase (o.J.). *Programming an I2C EEPROM Using the Promira Serial Platform and the Control Center Serial Software.* Abgerufen von https://www.totalphase.com/support/articles/203818198-Programming-an-I2C-EEPROM-Using-the-Promira-Serial-Platform-and-the-Control-Center-Serial-Software [Stand 04.05.2018].

Wefel, S., & Rost, M. (2016). *Sensorik für Informatiker: Erfassung und rechnergestützte Verarbeitung nichtelektrischer Größen, Messsignalverarbeitung und Datenanalyse*, Oldenbourg: De Gruyter.

Wendel, J. (2007). *Integrierte Navigationssysteme: Sensordatenfusion, GPS und inertiale Navigation*, München: Oldenbourg Wissenschaftsverlag GmbH.

Wikibooks (2018). *Python Programming/Threading.* Abgerufen von https://en.wikibooks.org/wiki/Python_Programming/Threading [Stand 15.09.2018].

Xu, X., Tang, Y., & Li, S. (2017). Indoor localization based on hybrid Wi-Fi hotspots. *2017 International Conference on Indoor Positioning and Indoor Navigation (IPIN)*, 1-8.

Zhang, C., Kuhn, M., Merkl, B., Fathy, E. A., & Mahfouz, M. (2006). Accurate UWB indoor localization system utilizing time difference of arrival approach. *2006 IEEE Radio and Wireless Symposium*, 515-518.

Erklärung zur Urheberschaft

Hiermit erkläre ich, Cornel Remer, dass ich die vorliegende Arbeit noch nicht für andere Prüfungen eingereicht habe. Ich habe die Arbeit selbständig verfasst. Sämtliche Quellen einschließlich Internetquellen, die ich unverändert oder abgewandelt wiedergegeben habe, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, habe ich als solche kenntlich gemacht.

Ich bin mir darüber bewusst, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Cornel Remer
Berlin, 14. November 2018