DHBW
Duale Hochschule
Baden-Württemberg

Sensor-Integration und Indoor-Navigation

Studienarbeit T3100

im Studiengang TEN-13

an der DHBW Ravensburg
Campus Friedrichshafen

von

Plabst Ulrich

Abgabedatum
17.06.16

Matrikelnummer	8235805
Betreuer (DH)	Prof. Dr.-Ing Thomas Mannchen

Erklärung

gemäß § 5 (3) der „Studien- und Prüfungsordnung DHBW Technik“ vom 22. September 2011.
Hiermit erkläre ich, dass ich die vorliegende Arbeit mit dem Titel

Sensor-Integration und Indoor-Navigation

selbständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt, keine anderen als die angegebenen Hilfsmittel benutzt und wörtliche sowie sinngemäße Zitate als solche gekennzeichnet habe.

Ravensburg, den 17.06.16

_____ Plabst Ulrich TEN 13

Inhaltsverzeichnis

Abkürzungsverzeichnis

APL	Application Layer
BBB	Beagle Bone Black
CL	Compatibility Layer
DOP	Dilution Of Precision
FL	Filter Layer
GPS	Global Positioning System
HAL	Hardware Abstraction Layer
I^2C	Inter Integrated Circuit
MAC	Media Access Control
OSL	Operating System Layer
RSSI	Receive Signal Strength Indicator
UAV	Unmanned Aerical Vehicle
USB	Universal Serial Bus

Abbildungsverzeichnis

1. Einführung

2. Zielsetzung

Im Rahmen der Projektarbeit soll eine Software zur Bestimmung der Position entwickelt werden. Die errechneten Positionswerte sollen an den Flugführungsrechner eines Unmanned Aerical Vehicle (UAV) übergeben werden. Die Software soll für die eingebettete Plattform Beagle Bone Black (BBB) erstellt und getestet werden. Anschließend soll eine kurze Evaluation Aussage über die Güte der entstandenen Positionslösung geben. Ein Flugtest, in dem das UAV mit Positionslösungen des *indoorGps* versorgt wird, soll als Systemtest Aussage über die Funktionsfähigkeit treffen.

3. Anforderungen an die Navigationssoftware

Die Anforderungen an die Navigationssoftware, die innerhalb dieses Projekts entwickelt bzw. integriert werden soll, werden im folgenden aufgelistet. Für den Erfolg des Projekts sind sowohl funktionale als auch nicht-funktionale Anforderungen zu erfüllen.

3.1. Funktionale Anforderungen

Die funktionalen Anforderungen an die Software können direkt aus der Zielsetzung des Projekts abgeleitet werden.

Ortung
Die Navigationssoftware soll Sensoren auslesen und die erhaltenen Sensordaten dazu nutzen, seine Position im Raum zu bestimmen. Eine minimale Güte der Positionslösung wird in den Anforderungen nicht festgelegt, da zunächst unterschiedliche Ansätze experimentell ausprobiert werden sollen. Anschließend kann die beste Methode vollständig implementiert und auf dem UAV implementiert werden. Positionslösungen sollen mindestens mit einer Frequenz von 1 Hz errechnet werden können, da dies die Frequenz ist, mit der das aktuelle Global Positioning System (GPS)-Modul seine Positionslösung in den Flugrechner einspeist.

Treibermodule
Von der Software sollen Treibermodule zum Auslesen der für die Ortung relevanten Parameter bereitgestellt werden. Jeder Parameter soll mit einem Zeitstempel

versehen werden, über den andere Softwaremodule nachvollziehen können, wann der Sensorwert entgegengenommen wurde.

Plattform
Da es Ziel der Software ist, eine Positionslösung für den Flugregler des UAV zu generieren, soll sie für den Payload-Controller des UAV entwickelt werden. Der bereits auf dem Flugobjekt integrierte Controller ist der BBB, auf dem eine embedded Version der Linux Distribution *Debian* läuft. Bei der Implementierung der Software soll insbesondere auf die Kompatibilität eventueller Frameworks und Bibliotheken mit der eingebetteten Plattform geachtet werden.

Schnittstelle zum Flugregler
Die Flugregelung des UAV wird von der Copter Version der *ArduPilot*-Software, die auf dem dafür ausgelegten Controller *APM 2.5* ausgeführt wird, gehandhabt. Die *ArduPilot*-Software ist zwar open-source aber sehr wenig dokumentiert. Um eine Integration des indoor Navigationssystems bei bestehender Flugregelung so einfach wie möglich zu machen, soll die von der Navigationssoftware errechnete Positionslösung mit dem gleichen Format, wie die GPS-Nachrichten und über die gleiche Hardware-Schnittstelle, die aktuell das GPS verwendet, übertragen werden.

3.2. Nicht-funktionale Anforderungen

Die nicht-funktionalen Anforderungen nehmen keinen Einfluss auf die Funktionalitäten, die Navigationssoftware bereitstellt. Im Hinblick darauf, dass die Software jedoch in weiteren Studienarbeiten nächster Semester weiterverwendet und auch erweitert werden soll, sind die folgenden Anforderungen von entscheidender Bedeutung.

Modularität und Erweiterbarkeit
Die gesamte Software soll im Hinblick auf Modularität und Erweiterbarkeit entwickelt werden. Dies bedeutet, dass die Funktionalitäten sinnvoll in eigenen Softwaremodulen gekapselt werden. Ferner sollen Abstraktionen zwischen Softwaremodulen das leichte Ausbauen der Software ermöglichen, um zukünftig auch das effiziente Hinzufügen weiterer Sensortreiber für unterschiedlichste Sensoren zu erlauben.

Robustheit
Da die Positionslösung ein kritischer Bestandteil der Flugregelung ist, soll die Soft-

ware auch robust gestaltet werden. Im zeitlichen Rahmen des Projekts ist es jedoch nicht möglich eine ausreichende Testabdeckung zur Evaluation der Robustheit zu erreichen. Es soll daher für dieses Projekt ausreichen ein geeignetes Konzept zur Fehlerhandhabung zu implementieren. Dafür sollen aus kritischen Funktionen Antwortcodes zurückgegeben werden, die über den Erfolg der aufgerufenen Funktion Aufschluss geben. Die aufrufende Funktion muss den Fehler dann handhaben oder seinerseits einen Fehlercode generieren. Dabei sollen aufschlussreiche Fehlermeldungen ausgegeben werden, die Funktionsname, Klasse und eine Beschreibung des Fehlers beinhalten.

Dokumentation
Die gesamte Software soll ausreichend dokumentiert und kommentiert werden, um in zukünftigen Semester weiterentwickelt werden zu können.

4. Software Design

Im folgenden Kapitel soll geforderte Funktionalität in eigene Softwaremodule unterteilt werden und das Zusammenspiel der Module untereinander erläutert werden.
Wie in den vorhergehenden Kapiteln beschrieben, soll die Ortung über die Messung der Signalstärke von Bluetooth und Wifi Beacons mit bekannter Position geschehen. Als Sensoren werden handelsübliche Bluetooth und Wifi Universal Serial Bus (USB)-Sticks verwendet. Aus der Signalstärke der verschiedenen Beacons kann ein Abstand berechnet werden. Hat man mindestens drei Abstände berechnet, kann die lokale Position gesucht werden. Theoretisch liegt diese im Schnittpunkt der von den Radien aufgespannten Kugeln.

4.1. Modularchitektur

Abbildung 1 zeigt die Anordnung der Verschiedenen Software- und Hardwaremodule, die am Aufbau beteiligt sind und über welche Schnittstellen diese miteinander interagieren.

4.2. Software Architektur

Zunächst sollen die in Abschnitt 3 geforderten Funktionalitäten in einzelne Softwaremodule gekapselt werden.

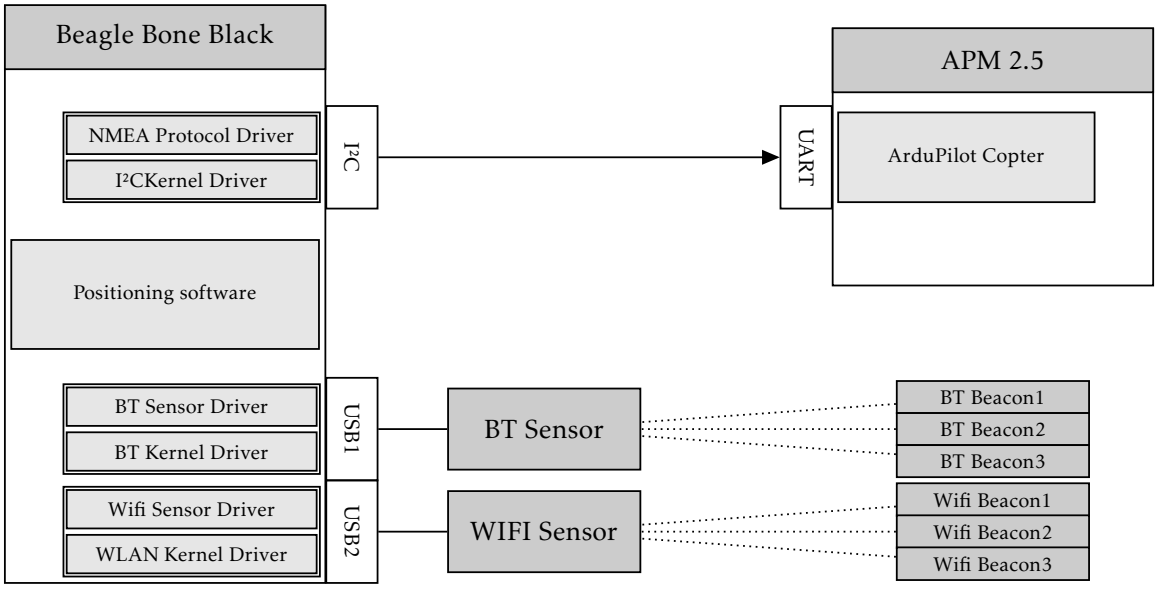

Abbildung 1: Anordnung der beteiligten Hardware- und Softwaremodule und deren Verbindungen

Triangulator
Der Triangulator ist das Modul, das auf Basis der gemessenen Radien die Position numerisch bestimmt und auch eine Bewertung der Lösung in Form einer Güteinformation generiert. Außerdem wird in diesem Modul der 1 Hz-Takt für die Ausgabe der Positionslösung generiert.

Filter
Eigene Filtermodule sollen eingesetzt werden, um eventuelles starkes Rauschen oder starken Jitter in den gemessenen Signalstärken zu unterdrücken.

Low level Sensor Treiber
Diese Treiber werden bereits vom Betriebssystem bereitgestellt. Sie interagieren mit der Sensorhardware und stellen die Signalstärke als gemessenen Parameter zur Verfügung.

High level Sensor Treiber
Die High level Sensor Treiber verwenden die von den Low level Sensor Treibern bereitgestellten Dienste zur Messung der Signalstärken und bringen diese in ein einheitliches Format. Anschließend geben sie diese Daten weiter.

Listener
Der Listener ist nötig, da wie in Abbildung 2 dargestellt, die High level Sensor Trei-

ber ihre Daten über eine POSIX Pipe an den Rest der Software übergeben. Dieses Modul dient dazu, die Daten von der Pipe entgegen zu nehmen, sie in C-Strukturen zu packen und mit einem Zeitstempel versehen an die Filter zu übergeben. Da die Ortung kontinuierlich jede Sekunde durchgeführt werden soll und gleichzeitig Sensordaten auf der Pipe kontinuierlich empfangen werden sollen, läuft der Listener in seinem eigenen Thread. Als Kommunikation zwischen dem Listener-Thread und dem Applikations-Thread wird die Signal/Slot Kommunikation, die von *Qt* bereitsgestellt wird, verwendet.
Die oben genanneten Module werden nun in einer nicht strikten Schichtenarchitektur angeordnet. Abbildung 2 zeigt die Module in den einzelnen Ebenen.

Hardware Abstraction Layer
Der Hardware Abstraction Layer (HAL) soll die verschiedene Hardware, die im Aufbau verwendet wird, für die darüber liegenden Schichten abstrahieren. In ihm sind die High level Sensor Treiber bzw. Protokoll Treiber für die Kommunikation mit der Flugregelung enthalten. Als Schnittstelle zur restlichen Software werden POSIX Pipes verwendet. So kann ein Treiber sowohl in C++, als auch in anderen Programmiersprachen wie beispielsweise im einfachsten Fall bash, geschrieben und mit dem Rest der Software verwendet werden. Jeder Sensortreiber im HAL läuft in seinem eigenen Ausführungsthread. Der Protokolltreiber dagegen, ist im Applikationsthread mit enthalten.

Compatibility Layer
Der Compatibility Layer enthält lediglich das Listener-Modul. Er fungiert als Schnittstelle zwischen den Sensortreibern, die ihre Daten über POSIX Pipes übergeben, und dem Filter Layer. Erhält diese Schicht einen Sensorwert, so wird ein Signal für den Filter Layer generiert und der dazu passende Slot aufgerufen. Letztlich stellt diese Schicht eine Kompatibilität zwischen dem asynchronen Programmablauf über selects auf POSIX Pipes und dem asynchronen Ablauf über Qt Signale und Slots her.

Filter Layer
Im Filter Layer gibt es für jeden Beacon, der zur Ortung verwendet werden soll, im HAL einen Filter, der die Sensormesswerte von unerwünschtem Rauschen oder Jitter befreit. Die Filter werden von einem Filter Control Modul gehandhabt - also instanziiert und ausgelesen. In einem Filter können auch Messwerte des gleichen Beacons aber von verschiedenen Treibern verarbeitet werden.

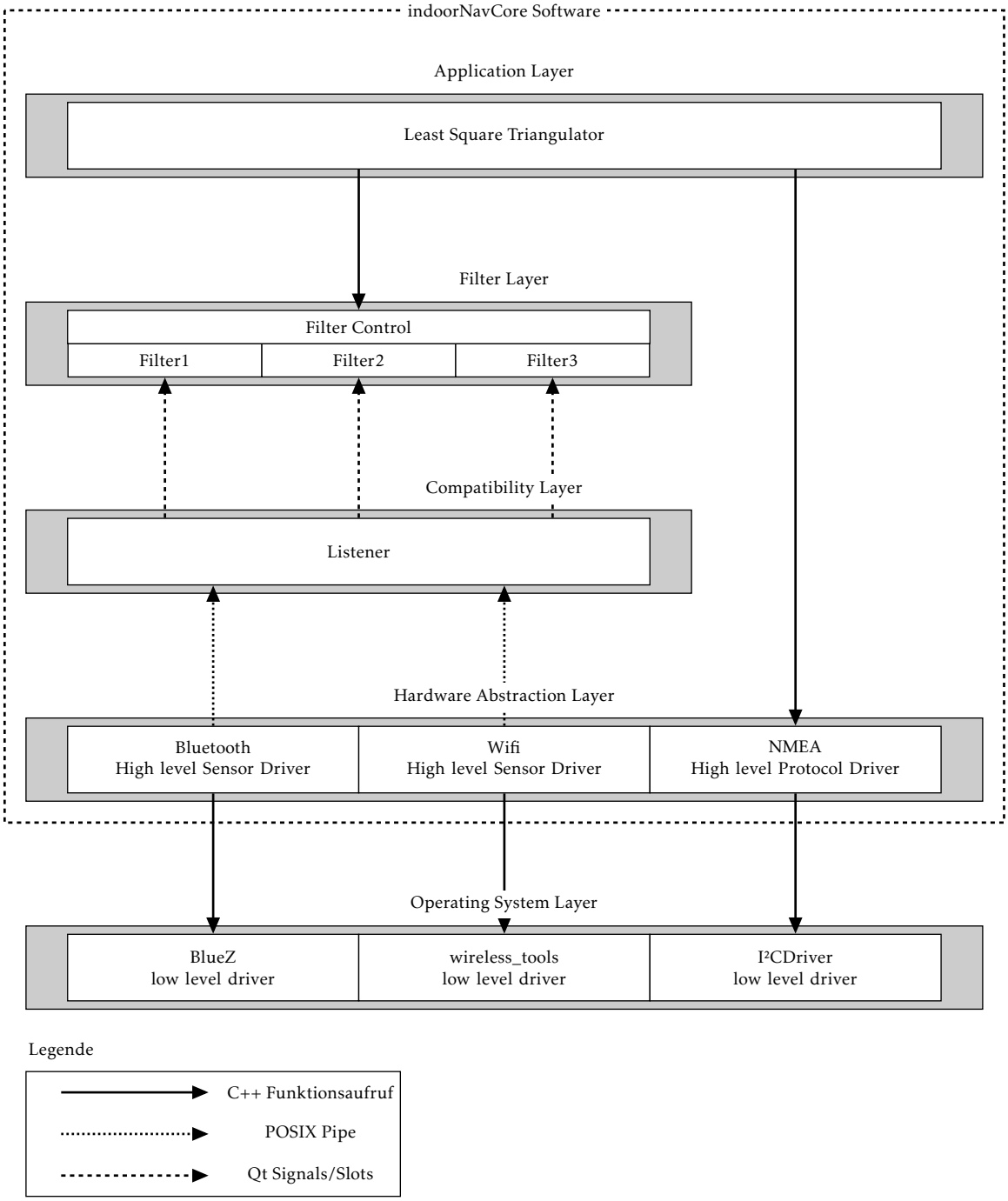

Abbildung 2: Nicht-strikte Schichtenarchitektur der indoor Navigationssoftware

Application Layer

Der Application Layer enthält die eigentliche Applikation - sprich das Ortungsmodul. Im Applikationsthread werden alle Module, die im Application Layer (APL) und im Filter Layer (FL) enthalten sind, ausgeführt. Abbildung 3 zeigt die einzelnen Threads, die parallel ausgeführt werden.

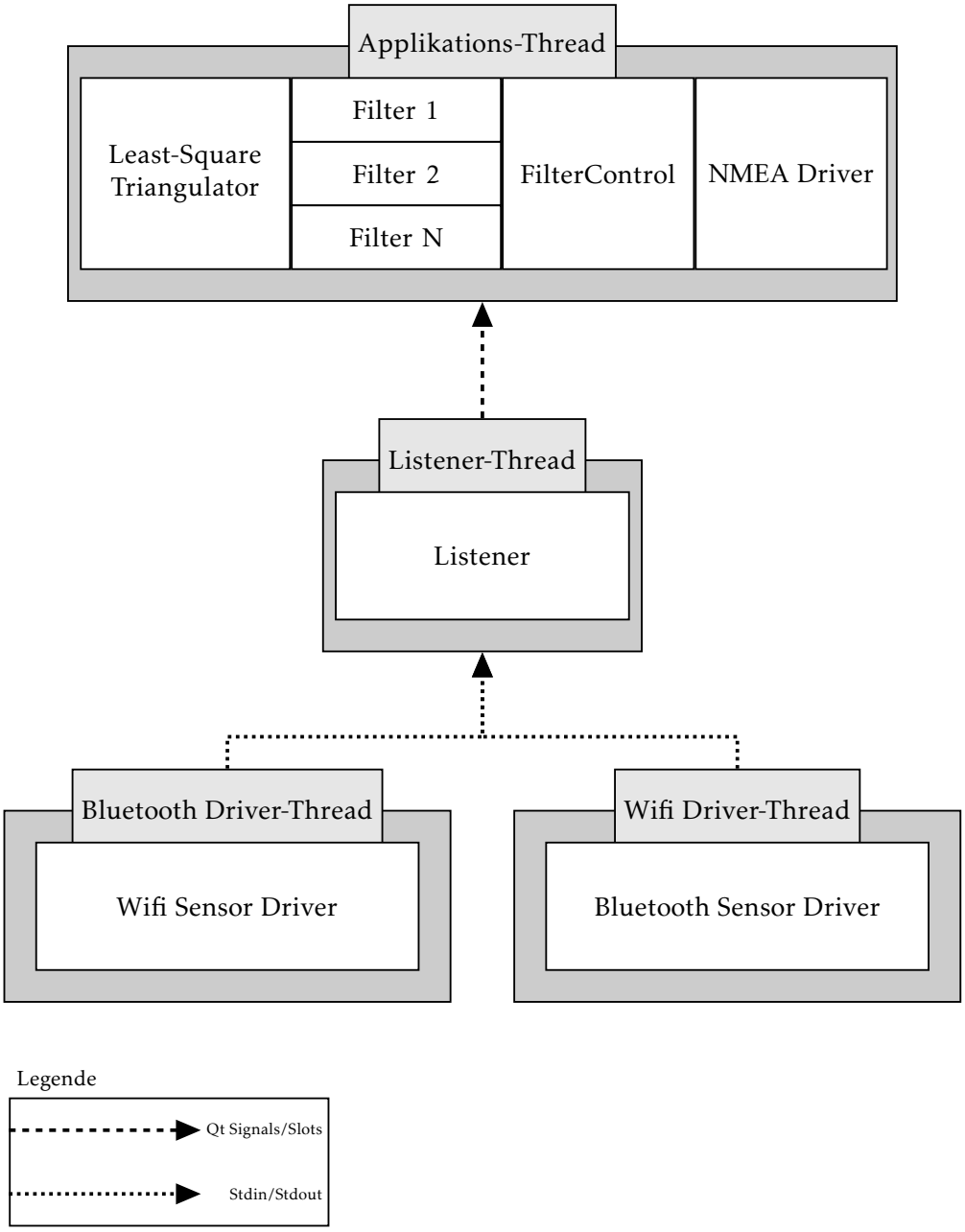

Abbildung 3: Ausführungsthreads der indoor Navigationssoftware

4.3. Schnittstellenbeschreibung

Im folgenden sollen die Strukturen der Daten, die zwischen Softwaremodulen übergeben werden, erläutert werden

Schnittstelle zwischen HAL und dem Compatibility Layer
Wie bereits erwähnt wird diese Schnittstelle durch die POSIX Pipes *Stdin* bzw. *Stdout* realisiert. Die Sensordaten werden dabei in folgendem Ascii-Format übertragen:

Typ MAC value
Dabei ist der Typ ein String, der angibt ob der nachfolgende Wert beispielsweise ein Receive Signal Strength Indicator (RSSI)- oder ein dBm-Wert ist. Diese beiden Möglichkeiten sind im Moment die einzig implementierten. Ein RSSI-Wert wird mit dem String "rssiünd ein dBm-Wert mit dem String "dbm"gekennzeichnet. Anschließend wird die Media Access Control (MAC)-Addresse des Beacons, zu dem der Messwert gehört, in sechs Gruppen Hexadezimaler Nummern getrennt durch einen Doppelpunkt übertragen. Dieses Ascii-Format für MAC-Addressen wird in der Unix-Welt häufig verwendet. Als letztes wird der Messwert als String übertragen. Dezimalstellen werden dabei mit einem Punkt, nicht mit einem Komma von den ganzen Stellen seperiert. Beispiele für Übertragene Messwerte sind:
dbm 00:00:00:00:00:00 -54
rssi 00:00:00:00:00:01 -10
dbm 00:00:00:00:00:02 -20

Schnittstelle zwischen Compatibility Layer und Filter Layer
Der Compatibility Layer verwendet die Interfaceklasse *cBeaconSignalParameter*, um Messwerte an die Filter weiterzugeben. Abbildung 4 zeigt ein UML-Diagramm der Klasse. Für jeden Parameter Typ (im Moment RSSI oder dBm) wird eine eigene Kindklasse von *cBeaconSignalParameter* abgeleitet, in der die Methode *getRange() : double* implementiert wird. Diese Methode berechnet aus dem Messwert den Abstand zum Beacon.
Der Datentyp *typ* ist ein in der Klasse definierter Enum-Typ, mit den Ausprägungen *dbm* oder *rssi*.

Schnittstelle zwischen Filter Layer und Application Layer
Für diese Schnittstelle wird die Klasse *cFilteredSignalParameter* (siehe Abbildung 5 verwendet. Die Klasse enthält einen gefilterten Abstand zu einem Beacon und eine Güteinformation in Form der Standardabweichung aller Werte, die für die Filterung verwendet wurden.

Schnittstelle zwischen Protokoll Treiber und Ortungsmodul
Die Struktur tPositionSolution enthält die Parameter einer Positionlösung und wird vom Ortungsmodul an den Protokolltreiber übergeben. Die Elemente der *tPositionQuality* Struktur sind sogenannte Dilution Of Precision (DOP)-Faktoren. Sie geben an, wie sich das Rauschen der gemessenen Signalstärken auf die Varianz der Position auswirkt [**?**, S. 103]. Je kleiner diese DOP-Werte, desto besser ist die Güte

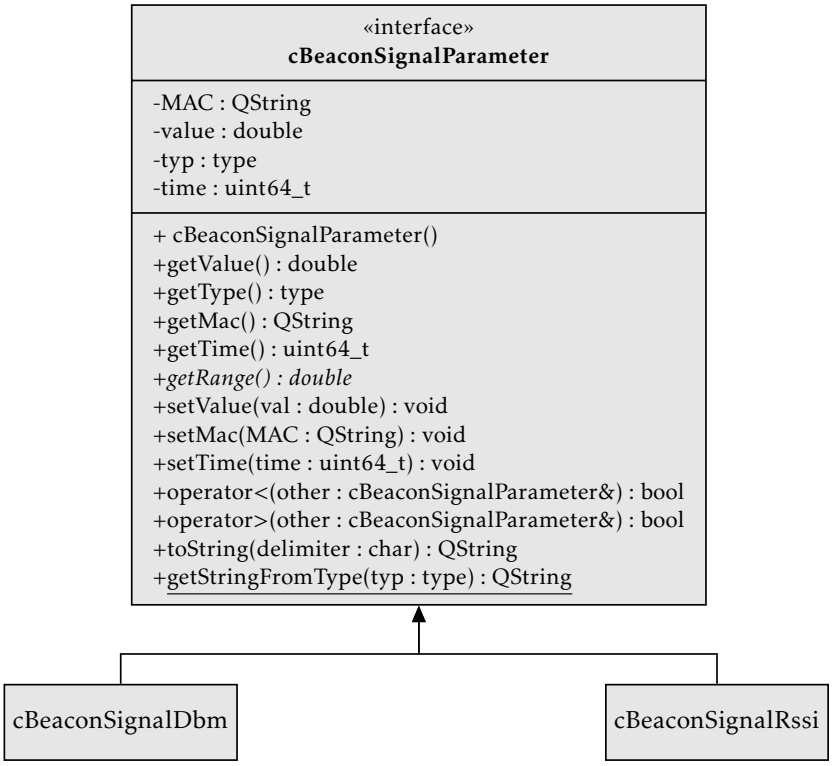

Abbildung 4: UML-Diagramm der *cBeaconSignalParameter* Klasse und deren Kindklassen

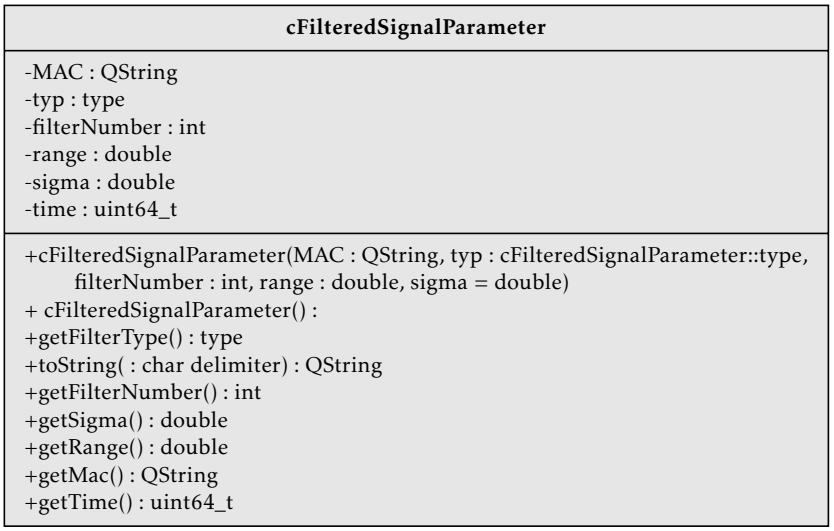

Abbildung 5: UML-Diagramm der *cFilteredSignalParameter* Klasse

der Positionslösung.

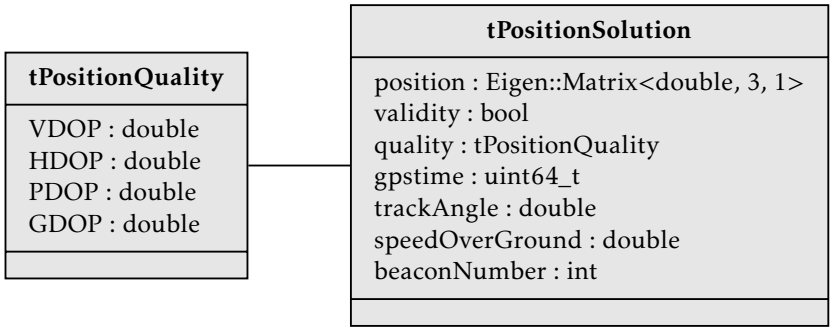

Abbildung 6: UML-Diagramm der *tPositionSolution* und *tPositionQuality* Strukturen

5. Implementierung

Die Implementierung wird gemäß einem objektorientierten Ansatz in *C++* mit dem Cross-Platform Framework *Qt 5.5.1* durchgeführt. Zunächst wird die gesamte Software auf dem PC entwickelt und anschließend eine mit Hilfe einer Cross-Compiler ToolChain eine Ausführbare Datei für den Beaglebone Black kompiliert. Für die Implementierung aller Funktionalitäten in der Software werden neben dem *Qt*-Framework auch noch andere Bibliotheken verwendet. Abbildung 7 zeigt eine Tabelle aller verwendeten Bibliotheken. In den folgenden Abschnitten wird

Bibliothek	Gebrauchte Funktionalitäten	Homepage
Qt 5.5.1	Threads, Inter-Thread-Kommunikation, Framework-Klassen	http://www.qt.io
Eigen 2.2.8	Lineare Algebra, Nicht-lineare Optimierungsverfahren	http://eigen.tuxfamily.org
GeographicLib 1.4.6	Berechnung von Geoidhöhe und magnetischer Deklination (für NMEA)	http://geographiclib.sourceforge.net
wirless-tools v.30	iwlib für das Auslesen der Signalstärke aus dem Wifi-Treiber	http://www.labs.hpe.com/personal/Jean_Tourrilhes/Linux/Tools.html

Abbildung 7: Verwendete Bibliotheken und deren Homepage

nur auf die relevanten Implementierungen eingegangen.

5.1. Implementierung des Least-Square Verfahrens (Gauß-Newton)

Das Least-Square Verfahren ist ein Verfahren zur Lösung nicht-linearer Gleichungssysteme durch eine lineare Annäherung. Es wird in der Numerik oft mit dem Gauß-Newton-Verfahren verwendet, um iterativ von einer Startposition gegen den kleinsten, quadratischen Fehler zu konvergieren. Mit quadratischem Fehler ist hier

die quadratische Differenz zwischen gemessenen Abstandswerten und Abständen der bekannten Beaconpositionen zur geschätzten Position gemeint. Tiefgründigere Informationen zu diesem Verfahren können in [**?**] gefunden werden.
Im Falle der Triangulation durch Messung von n Abständen zu bekannten Punkten mit den Positionen $\vec{r_{\mathrm{B,i}}}$ lautet die zu linearisierende Gleichung [**?**, S. 100]

$$\tilde{\vec{p}} = \begin{pmatrix} |\vec{r_{\mathrm{est}}} - \vec{r_{\mathrm{B,1}}}| \\ |\vec{r_{\mathrm{est}}} - \vec{r_{\mathrm{B,2}}}| \\ \vdots \\ |\vec{r_{\mathrm{est}}} - \vec{r_{\mathrm{B,n}}}| \end{pmatrix} + \begin{pmatrix} \vec{e_1} \\ \vec{e_2} \\ \vdots \\ \vec{e_{\mathrm{n}}} \end{pmatrix} \tag{5.1}$$

Abbildung 8 zeigt ein Ablaufschema des Verfahrens. Das Gleichungssystem wird mit Hilfe der Jakobi-Matrix H linearisiert [**?**, S. 100]

$$\tilde{\vec{p}} \approx \vec{h}(\vec{r}_{\mathrm{est}}) + H \cdot \Delta\vec{r} \qquad \text{mit} \qquad h_{\mathrm{i}} = |\vec{r}_{\mathrm{est}} - \vec{r}_{\mathrm{B,i}}| \tag{5.2}$$

In einem Iterationsschritt wird die aktuell geschätzte Position um den Vektor $\Delta\vec{r}$ verschoben

$$\vec{r}_{\mathrm{est,n+1}} = \vec{r}_{\mathrm{est,n}} + \Delta\vec{r} \qquad \text{mit} \qquad \Delta\vec{r} = (H^T \cdot R_{\mathrm{inv}} \cdot H)^{-1} \cdot H^T \cdot \Delta\vec{d} \tag{5.3}$$

Wobei $\Delta\vec{p}$ die Differenz zwischen den gemessenen Abständen und den aktuell berechneten und R_{inv} die Kovarianzen der Messwerte in seiner Diagonalen enthält. Ein Ablaufschema des Gauß-Newton Verfahrens ist in Abbildung 8 abgebildet. Die Startposition wird als geometrisches Mittel der Beacon-Positionen initialisiert. Errechnet der Algorithmus eine valide Position, so wird die Startposition auf diese Position gesetzt - sozusagen nachgeführt.

5.2. Implementierung des Levenberg-Marquardt Verfahrens

Das Levenberg-Marquardt Verfahren ist ebenfalls ein Verfahren zur Lösung von nicht-linearen Gleichungssystemen auf Basis einer linearen Annäherung. Es ist eine Mischung von Gradientenabstieg und dem oben aufgeführten Gauß-Newton Verfahren. Wegen des größeren Konvergenzraums im Hinblick auf Startpositionen findet dieses Verfahren in der Praxis weit öfter Anwendung als das Gauß-Newton Verfahren.
Zusätzlich zur Minimierung der quadratischen Fehlerfunktion wird hier die Nebenbedingung $|\Delta\vec{r}| < \rho_{\mathrm{trust}}$ [**?**, S. 186] definiert. Der Parameter ρ_{trust} ist die sogenannte *Trust Region*. Anschaulich stellt er eine Kugel um die aktuell geschätzte Position da, in der der Linearisierung des Gradienten vertraut wird - man also

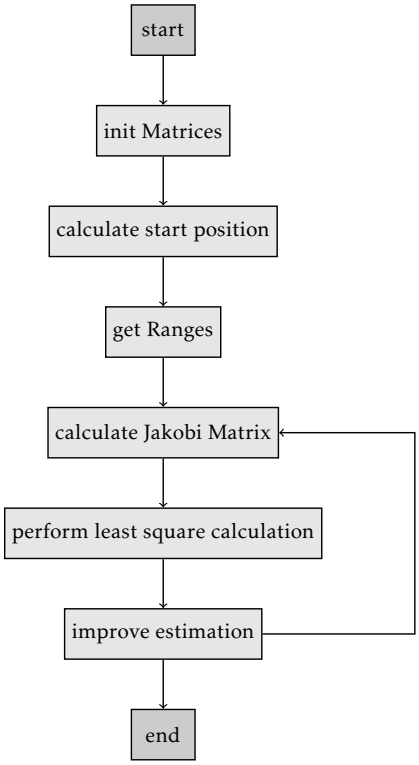

Abbildung 8: Veranschaulichung des Least-Square Algorithmus zur Schätzung der Position

mit einem vernachlässigbaren Fehler rechnet. Anders als beim Verfahren aus **??** wird hier nicht das Newton-Verfahren sondern das Hebden-Verfahren zur Näherung der Nullstellen verwendet. Wird bei einem Iterationsschritt ein $\Delta\vec{r}$ berechnet, das innerhalb der *Trust Region* liegt, so wird ρ_{trust} vergrößert und die neue Positionsschätzung übernommen. Wird die Bedingung $|\Delta\vec{r}| < \rho_{\text{trust}}$ verletzt, so wird die Positionsschätzung nicht übernommen und die *Trust Region* verkleinert. Die Bibliothek Eigen, die bereits als Template-Bibliothek für Matrixoperationen verwendet wird, stellt eine Implementierung dieses Algorithmus zur Verfügung. Einen tieferen Einblick in das Verfahren bietet [**?**].

5.3. Aufsetzen einer Cross-Compiler Toolchain

Um die Software für den Beaglebone Black zu kompilieren, muss eine Cross-Compiler Toolchain erstellt werden. Ein solches Werkzeugbündel erlaubt es auf einem Host-System mit einer festgelegten Rechnerarchitektur ausführbare Binärdateien für ein Client-System mit einer anderen Rechnerarchitektur zu kompilieren. Diese Toolchain enthält einen Cross-Compiler, Cross-Assembler und Cross-Linker, sowie eine Implementierung der Standard C-Bibliothek kompiliert für die Zielarchitek-

tur. Alle diese Werkzeuge müssen auf dem Host-System von einer bereits bestehenden Toolchain für die Host-Architektur kompiliert werden. Die Cross-Compiler Toolchain sind von der Host-Architektur ausführbare Binärdateien, die aber Binärdateien für die Zielarchitektur kompilieren können. Abbildung 9 zeigt das Vorgehen, beim erstellen solcher Werkzeuge. Eine Cross-Compiler Toolchain muss Bi-

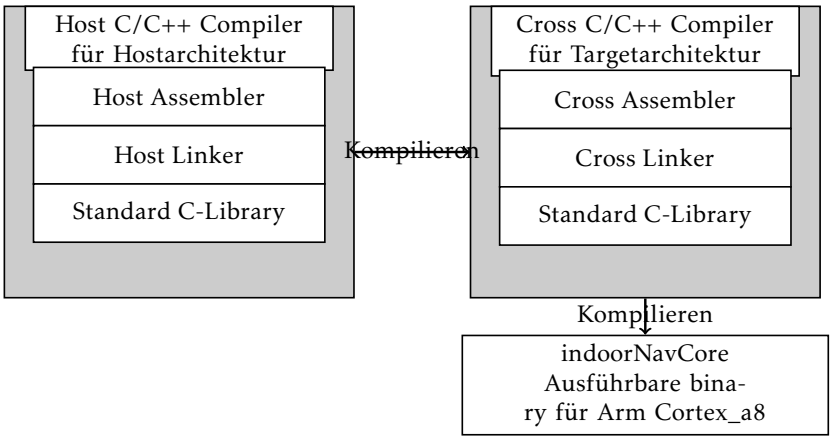

Abbildung 9: Erstellen einer Cross-Compiler Toolchain

närdateien für die auf dem Beaglebone verbaute ARM Cortex_A8 Architektur erstellen. Manche Linux Distributionen stellen bereits Pakete mit fertigen Toolchains für ARM Architekturen zur Verfügung. Übersichtlicher ist es jedoch, wenn man ein neues Wurzelverzeichnis (Sysroot) für die Cross-Compiler Toolchain auf dem Host-System einrichtet und dort alle benötigten Komponenten selber kompiliert. Ein Automatisieren des Vorgangs kann beispielsweise das Programm *crosstool-ng*(ornehmen. Im Folgenden wird beschrieben wie mir *crosstool-ng* eine Cross-Compiler Toolchain auf einer beliebigen Linux Distribution aufgesetzt wird.

Kompilieren der Toolchain mit crosstool-ng

Im Programm kann zunächst zwischen verschiedenen Beispielkonfigurationen gewählt werden. Die Konfiguration, die der für den Beaglebone gewünschten am nächsten kommt, nennt sich *arm-cortex_a8-linux-gnueabi*. Diese kann mit

```
1 ct-ng arm-cortex_a8-linux-gnueabi
```

ausgewählt werden. Da der Beaglebone jedoch nicht die Version 4.2 des Linux Kernels verwendet, muss dies noch an der Toolchain konfiguriert werden. Der Befehl

```
ct-ng menuconfig
```

öffnet ein Menü, in dem unter *Operating System* ein spezielles Verzeichnis, für die zu verwendende Linux Kernel Version angegeben werden kann. Die passende Version 3.8.13 des Kernels kann von zogen werden. Nach dem entpacken muss der Pfad dieses Verzeichnisses im Menü eingegeben werden. Außerdem empfiehlt es sich unter *Path and Misc Options –> Debug crosstool-ng* die Punkte *Save intermediate Steps* und *Interactive failed commands* auszuwählen. Dies dient dazu, dass bei einem Fehlschlag des Kompilierens nicht wieder ganz von vorne angefangen werden muss. Der Kompiliervorgang wird mit

```
ct-ng build
```

gestartet und kann bis zu einigen Stunde dauern. Falls Fehler beim erstellen einiger Komponenten auftreten, lohnt es sich in den Tips von Anhang A nach einer Lösung zu suchen. *Crosstool-ng* erstellt die konfigurierte Toolchain im Verzeichnis */x-tools*. Das Wurzelverzeichnis für die Cross-Compiler Umgebung befindet sich in */x-tools/arm-cortex_a8-linux-gnueabi/arm-cortex_a8-linux-gnueabi/sysroot*. In dieses Verzeichnis müssen nun mit der Cross-Compile Toolchain alle Bibliotheken, die verwendet werden sollen, kompiliert und installiert werden. Im Fall der indoorNavCore Software sind dies alle Bibliotheken, die in Abbildung 7 aufgelistet sind. Beim Kompilieren von Bibliotheken muss darauf geachtet werden, dass die richtigen Werkzeuge (die Cross-Compile Toolchain) und Umgebungsvariablen im Makefile spezifiziert sind. Manche Bibliotheken bieten hierfür zusätzlich zum Sourcecode ein *configure*-Script, dem die benötigten Parameter übergeben werden können und das automatisch diese Parameter in das Makefile überträgt. Konfigurationen und Makefiles, mit denen die Bibliotheken aus Abbildung 7 kompiliert wurden, sind in Anhang A zu finden. Eine ausführlichere Einführung in das Cross-Compile Thema bietet [**?**].

5.4. Modulliste

Im folgenden soll eine Tabelle die implementierten Softwaremodule, deren Dateien und das zugehörige Projekt, sowie eine Kurzbeschreibung zum Modul darstellen (Abbildung 10).

6. Evaluation

Im Folgenden werden die Anforderungen und Zielsetzung reflektiert.

Softwaremodul	Projekt	Dateien	Kurzbeschreibung	Abhängigkeiten
initialConfig	indoorNavCore	initialconfig.h initialconfig.cpp	Definiert Beaconparameter, Filter	-
global	indoorNavCore	global.h global.pp	Definiert globale Datentypen	Qt, Eigen
ApplicationThread	indoorNavCore	capplicationthread.h capplicationthread.cpp	Applikationsthread, startet alle Module die in ihm laufen	Qt
Log	indoorNavCore	clog.h clog.cpp	Klasse zum schreiben in ein Logfile	Qt
ProtocolDriver	indoorNavCore	cprotocoldriver.h cprotocoldriver.cpp	Interface-Klasse für Protokolltreiber	-
NmeaProtocolDriver	indoorNavCore	cdmeaProtocoldriver.h cnmeaprotocolDriver.cpp	Nmea-Protokolltreiber	GeographicLib
Filter	indoorNavCore	cfilter.h cfilter.cpp	Interface-Klasse für Filter	Qt
MeanFilter	indoorNavCore	cmeanfilter.h cmeanfilter.cpp	Gleitendes Mittelwertfilter	Qt
MedianFilter	indoorNavCore	cmedianfilter.h cmedianfilter.cpp	Gleitendes Medianfilter	Qt
FilterControl	indoorNavCore	cfiltercontrol.h cfiltercontrol.cpp	Handhabt Filter und Schnittstelle zu APL	Qt
ListenerThread	indoorNavCore	clistenerthread.h clistenerthread.cpp	Instanziiert Listener-Modul	Qt
Listener	indoorNavCore	clistener.h clistener.cpp	Selektiert auf STDIN und liest Messwerte der Treiber	Qt
Triangulator	indoorNavCore	clriangulator.h ctriangulator.cpp	Interface-Klasse für Triangulator-Klassen	Qt, Eigen
LevenbergMarquardTriangulator	indoorNavCore	clevenbergmarquardttriangulator.h clevenbergmarquardttriangulator.cpp	Implementiert Triangulation über Levenberg Marquardt Verfahren	Qt, Eigen
LeastSquareTriangulator	indoorNavCore	cleastsquaretriangulator.h cleastsquaretriangulator.cpp	Implementiert Triangulation über Gauß-Newton-Verfahren	Qt, Eigen
MathTools	indoorNavCore	mathtools.h, constants.h mathtools.cpp	Koordinatensystemtransformationen Mathematische, geographische Konstanten	-
TestDriver	testDriver	testdriver.h testdriver.cpp, main.cpp	Generiert Signalstärken in dBm mit variablem Rauschen für best. Beaconparameter	Qt, Eigen
wifiDriver	wifiDriver	main.cpp	Liest Signalstärken aller verfügbaren WIFIs und gibt sie formatiert aus	wireless_tools, Qt

Abbildung 10: Übersicht aller implementierten Softwaremodule, Dateien inklusive Kurzbeschreibung

6.1. Ortung

Problem des zweiten Minimums

Ein Problem der Ortung ist, dass beide implementierten Verfahren versuchen in das lokale Minimum zu konvergieren. Werden wie anfänglich gedacht drei Beacons verwendet, so gibt es im Allgemeinen zwei Minima der quadratischen Fehlerfunktion, da die Radien zu den Beacons nie ideal gemessen werden können und sich so auch nie in genau einem Punkt schneiden. **??** und Abbildung 12 veranschaulichen dies im zweidimensionalen. Gezeichnet ist jeweils die quadratische Fehlerfunktion über X und Y. Für die Koordinaten der Beacons wurde angenommen:
$\vec{B}_1 = (1\ \ 0)^T \quad ; \quad \vec{B}_2 = (-1\ \ 0)^T \quad ; \quad \vec{P}_{\text{ant}} = (0\ \ 0)^T$
Als gemessener Abstand zu allen Beacons wurde 1.2 m gewählt. Der rote Punkt in den Grafiken ist die Startposition, von der das least-square Verfahren »losläuft«, während der grüne Punkt die Endposition ist, gegen die der Algorithmus konvergiert. Wie man sieht, hängt das Resultat des Algorithmus von der Startposition ab. Das Problem lässt sich nur mit der Verwendung zusätzlicher Beacons beheben. **??** zeigt die Fehlerfunktion über X-Y-Koordinaten bei drei verwendeten beacons. Man stellt fest, dass es nun nur noch ein Minimum gibt, gegen das auch in beiden Startpositionen aus **??** und Abbildung 13 konvergiert wird.

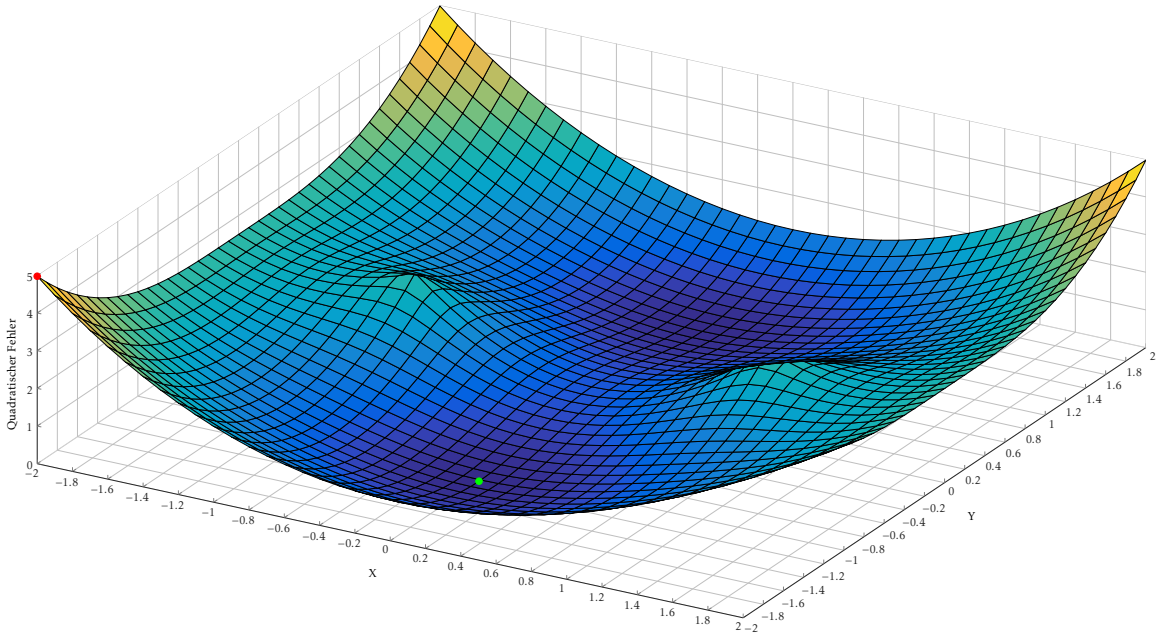

Abbildung 11: Quadratischer Fehler über X-Y-Koordinaten im zweidimensionalen mit 2 Beacons mit Startposition $\vec{r_{\text{start}}} = (2 \;\; 2 \;\; 2)$

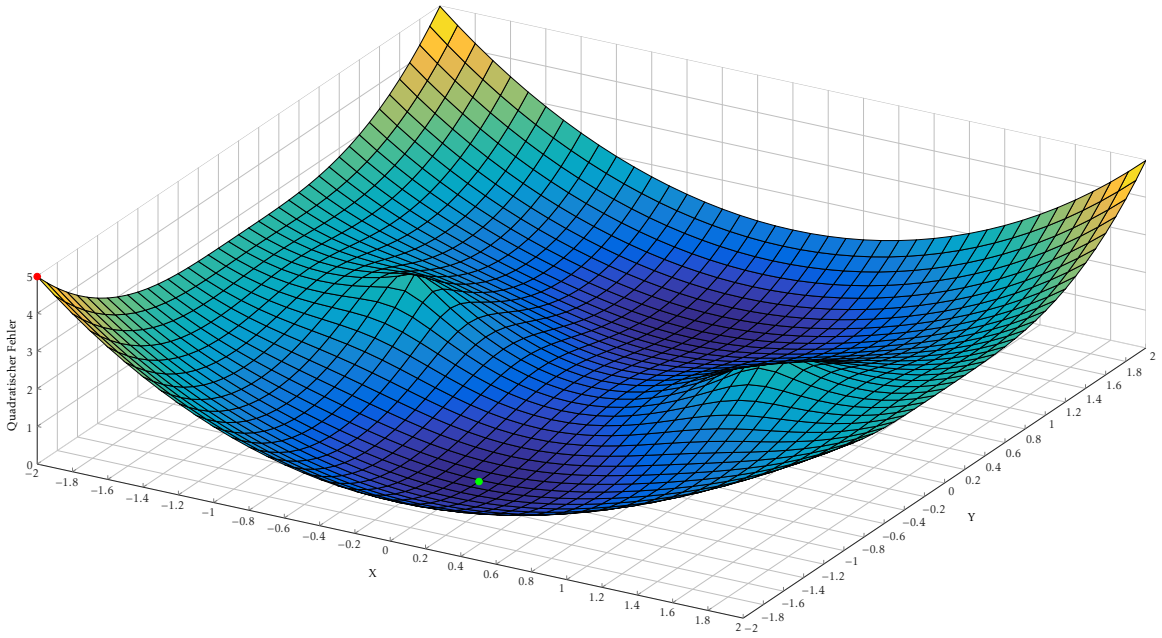

Abbildung 12: Quadratischer Fehler über X-Y-Koordinaten im zweidimensionalen mit 2 Beacons mit Startposition $\vec{r_{\text{start}}} = (-2 \;\; -2 \;\; -2)$

Problem der Divergenz

Ein allgemeines Problem des Gauß-Newton Verfahrens ist, dass bei ungünstig gewählter Startposition der Vektor $\Delta\vec{r}$, um den die Position propagiert wird, betragsmäßig sehr groß werden kann. Die Linearisierung der nicht-linearen Fehlerglei-

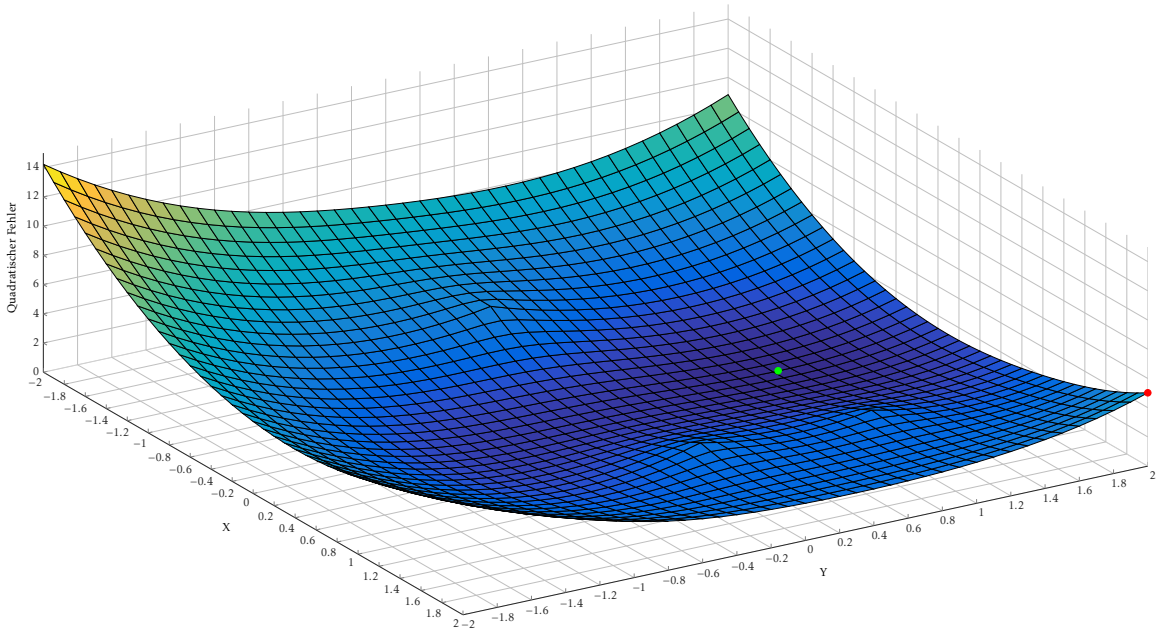

Abbildung 13: Quadratischer Fehler über X-Y-Koordinaten im zweidimensionalen mit 3 Beacons

chung ist jedoch nur in einem bestimmten Bereich ausreichend gut. So kann es passieren, dass bei ungünstiger Startposition der quadratische Fehler nach einer Iteration größer ist, als vor der Iteration. Das Bedeutet, dass sich der Gradient der Fehlerfunktion über die Schrittweite von $\Delta\vec{r}$ sehr weit von der linearen Annäherung verändert hat. Ein solches Verhalten kann zu einer divergierenden Oszillation führen. Ím Allgemeinen ist der Konvergenzraum bezüglich der Startpositionen bei dem Gauß-Newton Verfahren so klein, dass es in der hier thematisierten Anwendung nicht praktikabel ist. Deswegen wurde nachträglich noch das Levenberg-Marquardt-Verfahren, das sich mit einem wesentlich größeren Konvergenzraum auszeichnet, implementiert. Bei den durchgeführten Tests ist das Verfahren kein einziges mal divergiert - auch bei sehr verrauschten Signalstärken konvergiert es.

Problem des Positionsjitter

Das Problem der Divergenz bei verrauschten Signalen ist zwar mit dem Levenberg-Marquardt-Verfahren behoben. Allerdings wird die errechnete Positionslösung mit zunehmender Rauschleistung, also abnehmendem Signal-Rauschleistungs-Verhältnis, immer schlechter. Außerdem fängt die Position von Berechnungsschritt zu Berechnungsschritt zu springen. Die Rauschleistung kann über eine größere Fenstergröße der Filter im FilterLayer verkleinert werden. Das wiederum verringert allerdings die Dynamik des Systems. Falls sich das UAV bewegt, folgt die Position dann nur langsam der tatsächlichen Bewegung. Es hat sich gezeigt, dass die Positionslösung

wie folgt verbessert werden kann:

- günstiges Setzen der Beacon-Positionen (gut räumlich verteilt, nicht in der gleichen Richtung zur Antenne!)
- verwenden von zusätzlichen

Um die Funktionsfähigkeit der Ortung zu evaluieren, werden mit dem *testDriver*-Modul künstlich Werte erzeugt. Es wird eine lineare Trajektorie der Antenne von $(-10 \quad -6 \quad -3)^T$ nach $(10 \quad 6 \quad 3)$ simuliert. Der *testDriver* generiert die zugehörigen Signalstärken und überlagert diese mit einem Rauschen der Standardabweichung $\sigma = 0.3$. Diese Signalstärken werden in der *indoorNavCore*-Software eingelesen, mit einem Mittelwertfilter der Fenstergröße 5 gefiltert und mit Hilfe des Levenberg-Marquardt Triangulators eine Positionslösung generiert. Abbildung 14 zeigt die realen und berechneten X-Werte über der Zeit. Abbildung 15 zeigt die Differenzen

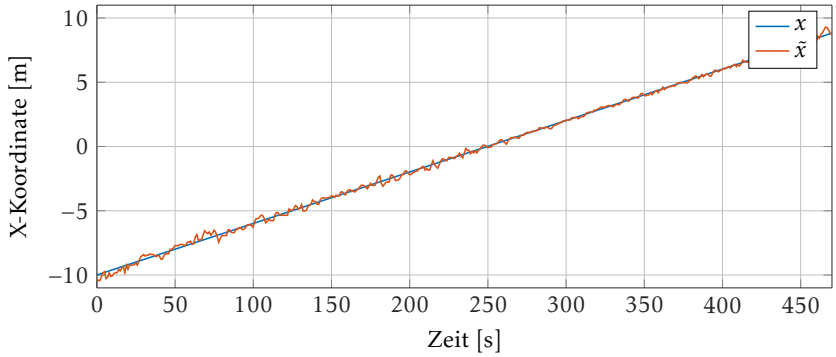

Abbildung 14: Reale X-Werte x und berechnete X-Werte $\tilde{x}$ bei einer linearen Trajektorie

zwischen den kalkulierten und realen Positionen in den jeweiligen Achsen. Diese

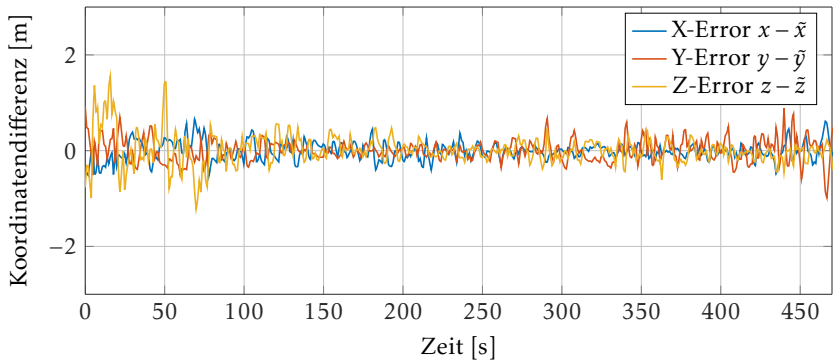

Abbildung 15: Fehler zwischen realen und berechneten Positionen in den jeweiligen Achsen mit simulierten Daten und $\sigma = 0.3$

Grafiken geben zwar nicht quantitative Aussagen über die Qualität der errechneten Positionslösungen, da die Standardabweichung frei gewählt wurde, allerdings lässt sich erkennen, dass der implementierte Levenberg-Marquardt Algorithmus funktionsfähig ist. Die großen Fehler am Anfang in **??** lassen sich dadurch erklären, dass die Mittelwertfilter zunächst ihr Fenster füllen müssen, um voll zum Tragen zu kommen. Die Simulation der Fehler wurde mit einer Standardabweichung $\sigma = 3$ wiederholt. Abbildung 16 zeigt, dass die Positionsfehler nun sehr groß werden. Ob-

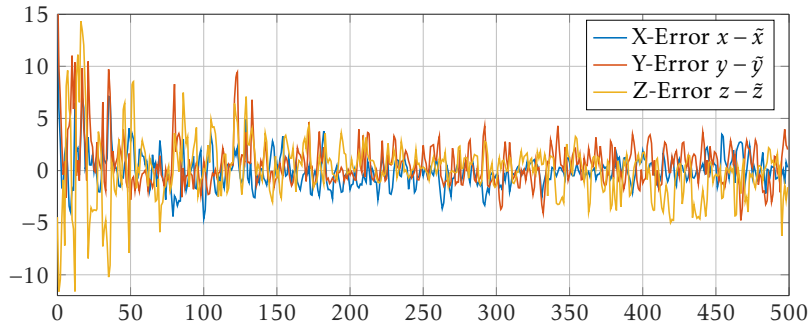

Abbildung 16: Fehler zwischen realen und berechneten Positionen in den jeweiligen Achsen mit simulierten Daten und $\sigma = 3$

wohl die Lösung nicht divergiert, sind die Positionslösungen aus Signalstärken mit einem solchen Rauschen nicht verwertbar. Eine Messung von realen Messwerten mit einer laptop-internen WLAN-Antenne zeigt, dass das zweite Szenario wesentlich näher and der Wirklichkeit ist, als das erste. Zusätzlich zum Rauschen kommt zu realen Messwerten noch Interferenz durch Mehrwegausbreitung und durch andere Signale in dem Frequenzbereich dazu.

6.2. Flugtest

7. Ausblick

Die im Rahmen dieses Projekts erstellte Software stellt eine Basis für zukünftige Arbeiten in Richtung Indoor-Navigation dar. Nicht alle Funktionen einer Navigationssoftware sind vollständig implementiert worden und die Testabdeckung ist aus Zeitgründen bestenfalls mangelhaft. Allerdings ist die Software modular genug gestaltet um jegliche auf Abstandmessung basierte Ortungsansätze zu unterstützen. Dazu müssen nur neue Interface-Klassen von *cBeaconSignalParameter* abgeleitet werden. So kann die Software in Zukunft weiterentwickelt werden. Eine TODO Liste aller noch nicht oder unvollständig implementierten Funktionen findet sich in Anhang B und im Code vor allen *//TODO* Kommentaren.

A. Tips

A.1. Aufbau des Repository

Das Repository mit Dokumentation und Software kann unter funden werden. Dort sind Dokumente unter */doc/* und Software unter */sw/* enthalten. Die Software ist in drei Projekte unterteilt:

- indoorNavCore –> Eigentliche Navigationssoftware
- wifiDriver –> Treiber zur Auslese der Signalstärken
- testDriver –> Treiber zum Simulieren von Signalstärken

Die indoorNavCore-Software ist wiederum unterteilt in die Ordner

- HAL –> alle Module des Hardware-Abstraction-Layers (ohne wifiDriver)
- CL –> alle Module des Compatibility-Layers
- APL –> alle Module des Application-Layers
- FPL –> alle Module des Filter-Layers
- MATH_TOOLS –> global verwendete

A.2. Probleme beim Erstellen der Cross-Compiler Toolchain mit ct-ng

GCC
Der Host GCC (6.0) kompiliert GCC der früheren Versionen nicht mehr –> der Build bricht ab mit einer Fehlermeldung ähnlich
error: 'const char libc_name_p(const char*, unsigned int)' redeclared inline with 'gnu_inline' attribute*
Der Fehler lässt sich mit dem Patch aus http://patchwork.openembedded.org/patch/122469/ beheben.

GDB
Der Cross-GDB kann nicht erstellt werden –> GDB wird zum Cross-Kompilieren nicht benötigt. Debuggen kann man nach wie vor im lokalen Build für die Host-Architektur.

A.3. Konfigurationen und Makefiles zum Cross-Kompilieren der Bibliotheken

Für einige der Libraries ist im Repository unter *sw/libraries/* bereits ein Makefile zu finden.

Qt 5.5.1
Qt wurde mit dem folgenden Befehl für den BeagleBone konfiguriert:

```
./configure -opensource -confirm-license -xplatform arm-cortex_a8-linux-gnueabi-g++ -prefix /home/uli/x-tools/arm-cortex_a8-linux-gnueabi/lib/qt5.0 -no-pulseaudio -qt-libpng -no-glib -no-cups -no-xcb -qt-xkbcommon -qt-zlib -qt-libjpeg
```

Diese Konfiguration ist allerdings nicht ideal. Man könnte einige mehr an Qt-Paketen aus dem Build entfernen. Für das Erstellen wurden die Umgebungsvariablen *ARCH=arm*, *PATH=/home/uli/x-tools/arm-cortex_a8-linux-gnueabi/bin* und *CROSS=/home/uli/x-tools/arm-cortex_a8-linux-gnueabi/arm-cortex_a8-linux-* gesetzt. Eine Make-Spezifikation findet sich unter *sw/libraries/Qt/mkspecs* im Repository.

GeographicLib
GeographicLib wurde mit folgendem Befehl für den BeagleBone konfiguriert:

```
./configure  --host=arm-cortex_a8-linux-gnueabi -8-linux-gnueabi --prefix=/home/uli/x-tools/arm-cortex_a8-linux-gnueabi/arm-cortex_a8-linux-gnueabi/sysroot
```

Dabei wurden die Umgebungsvariablen *ARCH=arm*, *PATH=/home/uli/x-tools/arm-cortex_a8-linux-gnueabi/bin* und *CROSS=/home/uli/x-tools/arm-cortex_a8-linux-gnueabi/arm-cortex_a8-linux-* gesetzt.

A.4. Dynamisches linken von Libraries auf dem BBB

Werden Bibliotheken nicht statisch beim Kompilieren auf dem Host-System in die Binärdatei gelinkt sondern dynamisch, so müssen die Bibliotheken auch als dynamisch linkbare Bibliothek auf dem BBB zur Verfügung stehen. Dies ist beispielsweise bei Qt der Fall (andernfalls müsste man statische Bibliotheken von Qt erstellen). Dazu kann man sie entweder in ein Verzeichnis kopieren, in dem der Linker bereits nach Bibliotheken sucht oder ein neues Verzeichnis erstellen und es in */etc/ldconf.so.d/meine-bibliothek.conf* hinzufügen. Nach dem Befehl

```
ldconfig
```

findet der Linker die Bibliothek.

B. TODO List

clevenbergmarquardttriangulator.cpp:
l. 63 –> Check if lmder1() converged

ctriangulator.cpp:
l. 109 –> Remove the QList() once it's no longer necessary
l. 175 –> Implement cTriangulator::getTrackAngle(Eigen::MatrixXd pos)

cfiltercontrol.cpp:
l. 35 –> Check if dublicate filter for one MAC?

cnmeaprotocoldriver.cpp:
l. 40 –> Implement calculation of *track made good* and *magnetic track made good*
overall: –> Test driver output formats and output to UART tty

overall:
–> document software properly
–> test filter configurations and their impact on the quality of position solution
–> look for another gps protocol that supports more than just 3 decimals for lat/lon
–> parse filter and beaconparameters on runtime via configuration file not on compile-time by global variable
–> look for memory leaks