



Hochschule für
Technik und Wirtschaft
Dresden
University of Applied Sciences

Thema:

Evaluierung und prototypische Realisierung von Algorithmen zur
Kartografie von Innenräumen durch mobile Roboterplattformen

Masterarbeit

zur Erlangung des akademischen Grades

Master of Science, M. Sc.

an der Fakultät Informatik/Mathematik
der Hochschule für Technik und Wirtschaft Dresden
im Masterstudiengang **Angewandte Informatik**
mit der Studienrichtung
Informations- und Kommunikationstechnologien

Author: Eric Hähner <s76727@htw-dresden.de>
geboren am: 02.06.1991
in: Dresden

Datum: 10. März 2020

1. Gutachter: Prof. Dr.-Ing. habil. Hans-Joachim Böhme
2. Gutachter: M.Sc. Robert Erzgräber
Betreuer: Dipl.-Inf. (FH) Frank Bahrmann

Danksagung

Zuerst möchte ich mich bei allen bedanken, die mich auf meinem bisherigen Weg bis zur Masterarbeit unterstützt haben. Das sind vor allem meine Eltern, mein Bruder und Freunde. Weiterhin danke ich dem Forschungsbereich für künstliche Intelligenz und kognitive Robotik der HTW Dresden für die Möglichkeit der Umsetzung dieser Arbeit. Ein besonderer Dank geht an Frank Bahrmann für die Betreuung meiner Arbeit sowie die vielen hilfreichen Anregungen und Unterstützungen.

Inhaltsverzeichnis

1 Einführung	1
1.1 Problemstellung und Motivation	1
1.2 Zielstellung	3
1.3 Aufbau dieser Arbeit	3
2 SLAM (Simultaneous Localization And Mapping)	5
2.1 Sensoren	5
2.1.1 Wahrnehmung der Umgebung	5
2.1.2 Eigenwahrnehmung	7
Inertiale Messeinheit	8
Odometrie	9
2.1.3 Behandlung von Messfehlern	10
2.2 Kartierung und Lokalisierung	10
2.2.1 Matching Verfahren	12
2.2.2 Kalman- und Partikelfilter	15
3 Google Cartographer	19
3.1 Hardwareanforderungen	19
3.2 Funktionsweise	20
3.2.1 Eingaben und Vorverarbeitung	21
3.2.2 LocalSLAM	22
3.2.3 GlobalSLAM	26
3.2.4 Fast Correlative Scan Matcher	28
3.3 Jackal Cartographer Demo	29
3.4 Ergebnisse mit Jackal	30
3.4.1 Karte 1: Langer Gang	30
3.4.2 Karte 2: Willowgarage	31
4 Anbindung des Google Cartographers in Java	35
4.1 Bundle Adjustment: Ceres-Solver und BoofCV	35
4.2 Java Native Interface	36
4.2.1 ROS-Schnittstellenanalyse	37
4.2.2 Vorbereitung und Planung	38

4.2.3 Erstellen einer C++ Schnittstelle	39
4.2.4 Erstellen der Java Schnittstelle	40
4.2.5 Erstellen der Java Native Interface (JNI)	41
5 Ergebnisse und Optimierung	43
6 Fazit und Ausblick	49
Literatur	51
Anhang	53
A Anhang	53

Acronyms

Notation	Description
DARPA	Defense Advanced Research Project Agency
ICP	Iterative Closest Point
IMU	Inertial Measurement Unit
JNI	Java Native Interface
LIDAR	Light Detection And Ranging
LM	Levenberg-Marquardt
LSA	Least Squares Adjustment
MEMS	Micro-Electro-Mechanical Systems
ROS	Robot Operating System
SLAM	Simultaneous Localization And Mapping
TSDF	Truncated Signed Distance Field

Glossar

Grid Map

Eine *Grid Map* beschreibt eine Einteilung der Umgebung in ein Raster/Gitter. In jeder Gitterzelle wird die Belegtheitswahrscheinlichkeit des zugehörigen gescannten Bereichs gespeichert. Die Verwendung wird in Kapitel 2.2 genauer beschrieben.

Pose

Eine *Pose* gibt eine Position und Orientierung eines Objektes bzw. Roboters an. Die Position wird in x,y -Koordinaten im zweidimensionalen Raum angegeben. Im dreidimensionalen Raum wird dem noch eine z -Koordinate hinzugefügt. Die Orientierung, auch Rotation genannt, kann in Euler Winkeln oder Quaternionen angegeben werden. Euler Winkel werden durch einen Wert (2D) bzw. drei Werten (3D: Pitch, Roll, Yaw) angegeben. Quaternionen werden durch vier Werte w,x,y,z definiert.

ROS

Robot Operating System (ROS) ist eine Menge von Software Tools, die zur Entwicklung von Anwendungen für Roboter verwendet werden. Es stellt unter anderem Softwarebibliotheken, Gerätetreiber, Visualisierungsmöglichkeiten und Kommunikationsmöglichkeiten zur Verfügung.

SLAM

Ein *Simultaneous Localization And Mapping (SLAM)*-Algorithmus dient der gleichzeitigen Kartierung der Umgebung und Selbstlokalisierung innerhalb dieser.

Trajektorie

Eine *Trajektorie* beschreibt den Bewegungspfad eines Objektes bzw. Roboters.

TSDF

Truncated Signed Distance Fields (TSDF) dient, ähnlich einer Grid Map, zur Beschreibung der Umgebung. Der Unterschied ist, dass jede Gridzelle die Entfernung zur jeweils am nächsten gelegenen Oberfläche speichert.

Abbildungsverzeichnis

2.1	Ein zweidimensionaler Grundriss eines Ganges der HTW Dresden.	5
2.2	Hochauflösendes 3D Mapping eines Light Detection And Ranging (LIDAR) Scanners einer städtischen Umgebung [SCHWARZ, 2010]	6
2.3	Einfluss unterschiedlicher Oberflächen auf Licht [YANG, 2011]	7
2.4	Apollo Inertial Measurement Unit (IMU) mit Gyroskop und Beschleunigungssensor [INTERBARTOLO, 2009]	8
2.5	Micro-Electro-Mechanical Systems (MEMS) Gyroskop, das den Corioliseffekt nutzt [WOODMAN, 2007].	9
2.6	In der Grid Map sind hohe Belegtheitswahrscheinlichkeiten dunkler und geringere Wahrscheinlichkeiten heller gekennzeichnet. Grün ist die Position des Roboters und rot sind die Abstandsmessungen.	10
2.7	Kreisschluss: Kartenfehler können behoben werden, wenn Bereiche mehrfach besucht werden. Rechts ist der gemessene Pfad. Links ist der real gefahrene/-korrigierte Pfad.	11
2.8	ICP Verfahren [ZHANG, 2018]: (a) Initiale Punktwolken (b) Ergebnis des ICP Algorithmus (c) Ergebnis des ICP-MCC Algorithmus (d) Ergebnis des ICP-BiMCC Algorithmus	13
2.9	Beispiel für Kalmanfilter: Der rote Kreis (links) ist ein Roboter, der den schwarzen Kreis (rechts) beobachten und dessen Position vorhersagen soll, wenn die nicht sichtbar ist. Das schwarze Rechteck unten verdeckt die aktuelle Position des Kreises. Der grüne Kreis gibt den möglichen Bereich des schwarzen Kreises an, nachdem dieser hinter dem Hindernis verschwunden ist.	16
2.10	Beispiel für Partikelfilter: Der große Kreis in der Mitte ist die aktuelle Position des Roboters. Die kleinen Kreise sind die Partikel. Rote Partikel entsprechen nicht der aktuellen Beobachtung und grüne repräsentieren eine mögliche Position.	17
3.1	Ein Rucksack, der mit mehreren Sensoren für den Cartographer ausgestattet ist.	19
3.2	Module des Google Cartographers: In der Umsetzung dieser Arbeit wurde der grau markierte Bereich durch eine Java Schnittstelle ausgetauscht, wie in Kapitel 4.2 näher beschrieben wird.	20
3.3	Probability Grid mit Hits (grau mit Kreuz) und Misses (grau) [HESS, 2016] .	25

3.4	Jackal Roboterplattform von Clearpath Robotics	29
3.5	Jackal Cartographer - langer Gang mit Standard-Konfiguration	30
3.6	Jackal Cartographer - langer Gang mit deaktiviertem GlobalSLAM	30
3.7	Jackal Cartographer - Willowgarage mit zwei Routen	31
3.8	Ergebnis Jackal Cartographer - Willowgarage Route 1 mit ermöglichten Kreisschlüssen: Die rote Linie im unteren linken Bereich markiert den Übergang einer Wand in den Scans an Start und Ziel. Es sind keine Abweichungen erkennbar.	32
3.9	Ergebnis Jackal Cartographer - Willowgarage Route 2 mit vermiedenen Kreisschlüssen: Die beiden roten Linien verlängern dieselbe Wand, die am Anfang und Ende der Route gescannt wurde. Es ist eine Abweichung erkennbar.	33
4.1	Dreidimensionale Szene, die durch Levenberg-Marquardt (LM) mit BoofCV erstellt wurde.	36
4.2	Architektur der entstandenen Schnittstelle zwischen Cartographer und Java mit JNI (vergrößert in Abbildung A.2 auf Seite 55)	38
5.1	Umgebungskarten mit nicht optimierter Konfiguration	43
5.2	HTW Dresden Bereich Fakultät Informatik: Odometrie und Ergebnisse vom Google Cartographer und GMapping, Länge der gefahrenen Distanz: 77m . .	45
5.3	HTW Dresden 3. Etage des S- und Z-Gebäudes: Dargestellt sind Odometrie und Ergebnisse vom Google Cartographer und GMapping. Die gefahrene Distanz beträgt 705,8m. Um die Unterschiede besser erkennen zu können, wurden die Abbildungen gestaucht.	46
A.1	Darstellung möglicher Cartographer Schnittstellen zur Analyse einer Verwendung in Java.	54
A.2	Architektur der entstandenen Schnittstelle zwischen Cartographer und Java mit JNI (vergrößert)	55
A.3	HTW Dresden 3. Etage des S- und Z-Gebäudes: Dargestellt sind Odometrie und Ergebnisse vom Google Cartographer und GMapping (nicht gestaucht). Die gefahrene Distanz beträgt 705,8m.	56

Tabellenverzeichnis

4.1 Verwendete Dateitypen zur Übergabe von Daten mit JNI	41
--	----

Listings

4.1 C++ Schnittstelle - Übergabe des Laserscans	39
4.2 Java Schnittstelle - Übergabe des Laserscans	40
4.3 JNI - Konvertierung des Laserscans	41
5.1 optimierte Cartographer Konfiguration	44

Kapitel 1

Einführung

1.1 Problemstellung und Motivation

In der heutigen Zeit werden Maschinen oft eingesetzt, um Menschen in vielen unterschiedlichen Bereichen zu unterstützen und ihnen Arbeit abzunehmen.

Die Unterstützung kann man beispielsweise im Automobilbereich sehen. Die eingesetzten Assistenzsysteme unterstützen den Fahrer z.B. beim Halten von Geschwindigkeit, Spur oder Sicherheitsabstand. Tesla, Audi, Mercedes-Benz und andere Hersteller bieten bereits unterschiedliche Stufen für autonomes Fahren an.

Maschinen werden auch in Gebieten eingesetzt, die zu weitläufig oder gefährlich sind bzw. an denen es zu teuer ist, Menschen einzusetzen [MONTEMERLO, 2007]. Solche Gebiete sind u.a. die Tiefsee, Höhlen und andere Planeten. An Orten, an denen kein GPS oder ähnliche Ortungsfunktionen verfügbar sind, müssen andere Möglichkeiten genutzt werden, um einen Roboter einsetzen zu können. Eine Variante hierfür ist es, ausschließlich onboard Systeme, ohne externe Unterstützung, zu nutzen. Die Orientierung und zielgerichtete Navigation durch unterschiedliche und vielfältige Umgebungen ist eine grundlegende Funktion vollautonomer Maschinen [JUANG, 2015; WANG, 2016].

Auch in nicht so speziellen Einsatzgebieten kann es sinnvoll sein, einen autonomen Roboter einzusetzen, da durch die Unabhängigkeit des Systems die Fehleranfälligkeit sinken kann. Ein Beispiel für einen solchen Einsatz ist ein geführter Rundgang mit Besuchern durch eine Ausstellung. Dabei könnte ein Roboter nacheinander unterschiedliche Ausstellungsstücke ansteuern und einen Text abspielen, der zu dem jeweiligen Gegenstand passt. Die Umgebung zur Navigation ist in diesem Fall wahrscheinlich wenigen Veränderungen ausgesetzt, da sich die Ausstellungsstücke an festen Positionen befinden und nicht ständig in Bewegung sind. Das vereinfacht die Umsetzung der autonomen Navigation, da der einmal aufgenommene Raum selten angepasst werden muss. Komplexer wird das Problem z.B. bei einem Roboter, der in einem Büro oder in einer Fabrik Gegenstände von einem Ort zum anderen bringen soll. Eine solche Umgebung ist mehr Veränderungen ausgesetzt als die des Ausstellungsszenarios.

Der Fachbereich für künstliche Intelligenz und kognitive Robotik der HTW Dresden nutzt in mehreren Forschungsprojekten Roboter, die selbstständig durch Umgebungen navigieren. Im

Projekt Care4All-Initial soll die Autonomie von Demenzpatienten verlängert werden, indem technische Assistenzsysteme eingesetzt werden und somit die Pflegebedürftigkeit so lange wie möglich hinausgezögert wird [BÖHME, 2018]. Dazu wird ein Roboter in die Therapie und sozialen Aktivitäten der Pflegeeinrichtung eingebunden. Dort benötigte Funktionen sind unter anderem die Erkennung von Hindernissen, Bewegungen, gefallenen Personen und die Navigation durch bekannte, dynamische Umgebungen.

Vor allem für die Navigation ist es vorteilhaft, wenn der Roboter die Umgebung selbst aufzeichnet, während er sich durch diese bewegt. Dazu werden Sensoren benötigt, aus deren Messungen eine Karte generiert werden kann. Innerhalb dieser kann sich der Roboter selbst lokalisieren und effizient eine optimale Route zu einem Ziel planen, um im Anschluss dahin zu navigieren. Bei Veränderungen der Umgebung kann diese Karte durch neue Messwerte angepasst werden. Dadurch wird die Navigation durch eine dynamische Umgebung möglich.

Die Karte kann z.B. in zweidimensionaler Form als Grundriss bzw. in Vogelperspektive visualisiert werden. Eine andere Variante ist die dreidimensionale Form, in der Gegenstände mit mehr Details dargestellt werden können. Bei verschiedenen Gegenständen kann man außerdem identifizieren und feststellen, ob diese verschiebbar, wie ein Karton, oder fest, wie eine Wand, sind.

Eine Familie an Algorithmen, die solche Karten erstellen, werden Simultaneous Localization And Mapping (SLAM) Algorithmen genannt. Unter der OpenSLAM Webseite¹ werden einige dieser Algorithmen erklärt und die Anforderungen dafür beschrieben. Darunter befindet sich GMapping, GridSLAM und RobotVision. Entwickler von SLAM Algorithmen können oft sehr exakte Ergebnisse präsentieren. Bei einer Nutzung in anderen Umgebungen oder Systemen können die Resultate sehr stark abweichen. Vermutlich liegt das an nicht optimal abgestimmten Konfigurationsparametern des Algorithmus. Im Jahr 2016 wurde der Google Cartographer in der Publikation [HESS, 2016] vorgestellt. Dieser Algorithmus soll sehr gute Ergebnisse in 2D und 3D liefern und weniger empfindlich auf nicht optimal eingestellte Konfigurationsparameter reagieren.

¹ OpenSLAM Webseite: <https://openslam-org.github.io/>

1.2 Zielstellung

Das Ziel dieser Arbeit ist der Wissensaufbau zu SLAM Algorithmen, sowie die Analyse einzelner Algorithmen des Google Cartographers. In einem Simulator soll vorerst die Qualität der Ergebnisse des Cartographers analysiert werden. Bisher wurden an der HTW Dresden zwei SLAM Algorithmen verwendet. Das sind FastSlam 2.0 und GMapping. FastSlam ist in Java implementiert. Für GMapping werden die aufgenommenen Daten gespeichert und im Anschluss über eine Implementierung in Robot Operating System (ROS) ausgewertet. Der Google Cartographer soll diese Algorithmen ersetzen um die Resultate zu verbessern. Die Herausforderung ist hierbei die Übertragung der Cartographer-Algorithmen von C++ und die Anbindung an die vorhandenen Sensoren und Ausgabegeräte.

1.3 Aufbau dieser Arbeit

In dieser Arbeit geht es vor allem um den SLAM Algorithmus „Google Cartographer“. Zu Beginn wird das grundlegende Problem beschrieben, was mit SLAM gelöst werden soll und welche Hardware benötigt wird. Es wird aufgezeigt, was das Ergebnis bzw. Ziel eines SLAM Algorithmus' ist. Da SLAM aus mehreren Algorithmen besteht, werden im Anschluss wichtige Konzepte erklärt. Diese Konzepte werden mit einer groben Ablaufbeschreibung des Google Cartographers verdeutlicht. Der Cartographer wurde im Rahmen dieser Arbeit an eine bereits vorhandene Roboterplattform angebunden. Probleme und Schwierigkeiten, die dabei aufgetreten sind, werden erklärt. Abschließend wird die Arbeit ausgewertet und zusätzlich auf Verbesserungsmöglichkeiten der Anbindung des Cartographers eingegangen.

Kapitel 2

SLAM (Simultaneous Localization And Mapping)

Simultaneous Localization And Mapping (SLAM) besteht aus zwei Teilen, die gleichzeitig ausgeführt werden. Einerseits muss der mobile Roboter seine eigene Position aufgrund des Umfeldes bestimmen. Andererseits müssen die aufgezeichneten Messungen gespeichert und kombiniert werden. Durch die gleichzeitige Ausführung der beiden Aufgaben ist es möglich eine Umgebungskarte zu erstellen. Diese ist z.B. ein Grundriss eines Raumes oder Gebäudes wie in Abbildung 2.1 zu sehen ist.



Abbildung 2.1: Ein zweidimensionaler Grundriss eines Ganges der HTW Dresden.

2.1 Sensoren

Nützliche Informationen bzw. Daten der Umgebung und des Roboters, die für einen SLAM-Algorithmus von Interesse sein können, werden aufgenommen und anschließend verarbeitet. Um die Informationen verarbeiten zu können, werden sie zunächst von Sensoren in digitale Signale umgewandelt.

2.1.1 Wahrnehmung der Umgebung

Zur Messung von Entfernungen in bestimmten Richtungen kann Light Detection And Ranging (LIDAR) verwendet werden. Es funktioniert wie ein Radar, mit dem Unterschied, dass Licht- statt Radiowellen ausgesendet werden [SCHWARZ, 2010]. Die meisten LIDAR Systeme verwenden einen einzelnen Laserstrahl, der über einen Spiegel in die gewünschte Richtung gelenkt wird. Andere, hochauflösende Varianten, verwenden einen rotierenden Kopf, der mit 64 Lasern ausgestattet ist. Jeder Laser sendet 20.000-mal pro Sekunde einen Laserstrahl aus. Indem sich der Kopf bis zu 900-mal pro Minute dreht, kann ein Bereich von 360° abgedeckt werden. Bei dieser Geschwindigkeit entstehen circa 1.3 Millionen

Datenpunkte. Die Punktewolke ist so dicht, dass Bordsteinkanten oder Oberleitungen in einer Entfernung bis zu 100m erkannt werden können. Diese Technik wurde ursprünglich für die Defense Advanced Research Project Agency (DARPA) „Grand Challenge“, ein vom amerikanischen Verteidigungsministerium gesponsorter Wettbewerb, entwickelt um die technologische Innovation von unbemannten Bodenfahrzeugen anzukurbeln. Die Anforderung des Wettbewerbs war die Navigation eines autonomen Fahrzeugs durch eine Wüstenlandschaft. In einem späteren DARPA Wettbewerb sollte durch eine städtische Gegend navigiert werden. Dabei mussten auch Straßenmarkierungen und Verkehr beachtet werden.

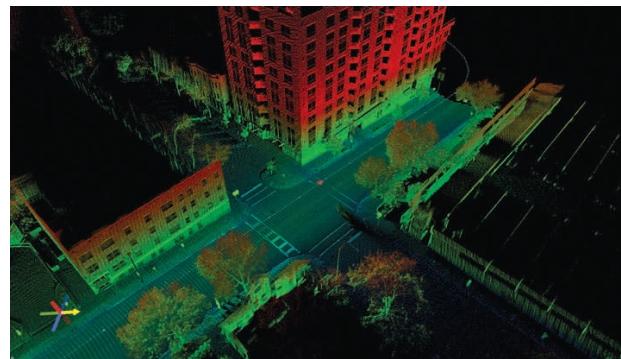


Abbildung 2.2: Hochauflösendes 3D Mapping eines LIDAR Scanners einer städtischen Umgebung [SCHWARZ, 2010]

Die Fahrzeuge mussten bei einer Geschwindigkeit von $65 \frac{\text{km}}{\text{h}}$ einen mindestens 15cm großen Gegenstand in einer Entfernung von 50m identifizieren und noch ausreichend Zeit haben, um diesen umfahren zu können. Auch die günstigeren konventionellen Systeme mit Spiegel können dichte Punktewolken erzeugen. Sie werden jedoch meist nur unter bestimmten Bedingungen z.B. bei gutem Wetter eingesetzt. Deshalb eignen sich solche Sensoren vor allem für Innenräume.

Zur Messung wird mit Hilfe des LIDAR System ein Lichtstrahl ausgesendet und über einen Spiegel in die gewünschte Richtung gelenkt. Befindet sich in dieser Richtung ein Gegenstand, wird das Licht reflektiert und zurück zum Sensor geworfen. Der Sensor misst die Zeit, die zwischen Aussenden und Empfang des Lichtstrahls liegt und kann damit die Entfernung des Gegenstandes errechnen. Wird kein Signal empfangen, ist an der gemessenen Position vielleicht kein Gegenstand. Diese Technik nennt sich „round-trip time of flight“ [YANG, 2011]. Aufgrund unterschiedlicher Oberflächen, Formen und Größen treffen nicht alle ausgesendeten Photonen wieder am Sender ein. Lichtstrahlen können von Objekten reflektiert, transmittiert oder absorbiert werden. Eine Reflexion kann gerichtet (Spiegel) oder diffus (matte Oberfläche) stattfinden. Bei einer gerichteten Reflexion ist der Einfallswinkel gleich dem Ausfallwinkel während diffus reflektiertes Licht in mehreren Winkeln ausfällt. Eine Glasscheibe lässt einen Großteil des Lichts hindurch, was als Transmission bezeichnet wird. Die Aufnahme des Lichts und Umwandlung in eine andere Energieform wird als Absorption bezeichnet.

Surface	Specular Reflection	Diffuse Reflection	Absorption	Transmission	Example
Smooth	○	△	△	×	Polished Metal
Rough	△	○	○	×	Dull Metal
Colored	○	○	△	×	White Surface
Dark	△	△	○	×	Black Surface
Transparent	△	△	△	○	Glass Window

In this table, ○, △, and × indicate that a large amount, a small amount, and none of light are affected, respectively.

Abbildung 2.3: Einfluss unterschiedlicher Oberflächen auf Licht [YANG, 2011]

In [YANG, 2011] bzw. Abbildung 2.3 wird die Auswirkung unterschiedlicher Oberflächen und Materialien auf Licht beschrieben. Beispielsweise absorbiert stumpfes Metall mehr Licht als eine glatte Metalloberfläche. Helle Oberflächen reflektieren mehr Licht als dunkle Oberflächen, jedoch absorbieren diese mehr Licht. Eine transparente Oberfläche, wie eine Glasscheibe, lässt das Licht des Lasers hindurch und es wird möglicherweise erst hinter der Glasscheibe reflektiert. Das würde dazu führen, dass das System nicht die Glasscheibe sondern den dahinter befindlichen Gegenstand misst. Ein Spiegel lenkt den Lichtstrahl um, was dazu führen kann, dass Objekte an falschen Positionen detektiert werden. Die Tabelle in Abbildung 2.3 verdeutlicht somit, dass ein LIDAR Sensor nicht eindeutig feststellen kann, ob sich an einer Position ein Gegenstand befindet. Zudem können solche Oberflächen einen großen Einfluss auf das Navigationsverhalten des Roboters haben und möglicherweise auch zu Schäden am Roboter führen. Um die aufgezeigten Situationen zu vermeiden, ist es möglich, diese Oberflächen zu erkennen und dadurch entstehende Fehler zu minimieren. Wie diese Erkennung funktioniert wird in [YANG, 2011] beschrieben.

2.1.2 Eigenwahrnehmung

Sind die Scans mit dem LIDAR System aufgezeichnet ist jedoch nicht bekannt, aus welcher Lage und Position diese ausgeführt wurden. Um dies festzustellen, gibt es Sensoren, die verschiedene Eigenschaften des Roboters zu einer bestimmten Zeit feststellen.

Inertiale Messeinheit

Eine Inertial Measurement Unit (IMU) dient zur Bestimmung der aktuellen Lage und Position des Roboters. Damit können gemessene Sensorwerte der Umwelt aufgrund dieser Lage ausgerichtet werden.

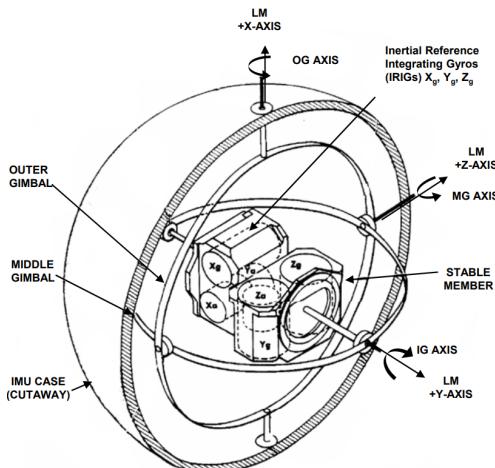


Abbildung 2.4: Apollo IMU mit Gyroskop und Beschleunigungssensor [INTERBARTOLO, 2009]

Ausgehend von einer Position, Orientierung und Geschwindigkeit werden relative Werte aufgezeichnet. Eine IMU besteht normalerweise aus drei Drehratensensoren (Gyroskop) und drei Beschleunigungssensoren, die Drehgeschwindigkeit und lineare Beschleunigung messen [WOODMAN, 2007, S. 5 ff.]. Der Aufbau einer solchen Messeinheit ist in Abbildung 2.4 zu sehen. Sie finden in vielen anderen Bereichen eine Anwendung, dazu gehören Luftfahrzeuge, Raketen, U-Boote und Schiffe. Die Messeinheit ermöglicht eine genauere Navigation. Diese Art der IMU wird auch heute noch für Bereiche verwendet, in denen hoch präzise Messungen benötigt werden [WOODMAN, 2007, S. 9]. Auch nach vielen Jahren der Entwicklung werden noch immer viele Komponenten zur Herstellung benötigt, was für einen hohen Preis sorgt.

Aus dem Grund werden oft Micro-Electro-Mechanical Systems (MEMS) Sensoren genutzt. MEMS Gyroskope bestehen aus weniger Komponenten und sind günstiger herzustellen. Zudem benötigen sie weniger Energie, sind kleiner, leichter und schneller einsatzbereit. Sie sind jedoch nicht so genau wie optische Geräte. Zur Bestimmung einer Drehgeschwindigkeit nutzen sie die Corioliskraft. Diese beschreibt, wie sich eine Masse (hier IMU) mit einer Geschwindigkeit auf einem rotierenden System (hier Planet Erde) verhält. Die Voraussetzung eines MEMS Gyroskops ist somit die Erdrotation. Die Gyroskope enthalten Elemente, die in Bewegungsrichtung ausgerichtet sind. Alle Vibrationen, die durch eine Bewegung erzeugt werden, können gemessen und ausgewertet werden.

Bei einer Drehung entsteht durch den Corioliseffekt eine Vibration senkrecht zur Bewegungsachse. Durch die Messung dieser Vibration kann die Drehgeschwindigkeit berechnet werden. Die benötigten Parameter zur Berechnung sind in Abbildung 2.5 veranschaulicht.

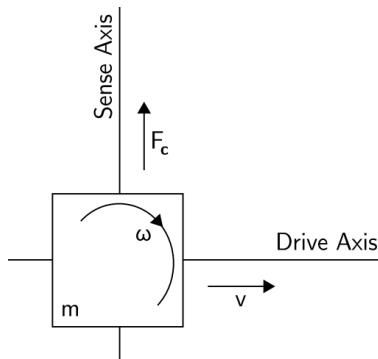


Abbildung 2.5: MEMS Gyroskop, das den Corioliseffekt nutzt [WOODMAN, 2007].

MEMS Beschleunigungssensoren messen entweder die Verschiebung einer Masse oder die Frequenz einer Vibration, um die Beschleunigung des Systems zu berechnen. Die Vorteile sind äquivalent zu den MEMS Gyroskopen.

Wie bei den meisten Sensoren, treten auch bei einer IMU Messfehler auf. Bei MEMS treten diese Fehler gewöhnlich häufiger auf, obwohl die Performance immer weiter optimiert wird [WOODMAN, 2007, S. 10–13]. Zudem sind diese Sensoren einer erhöhten Fehlerfortpflanzung ausgesetzt. Messfehler können z.B. durch eine falsche Kalibrierung, Sensorrauschen und Temperaturunterschiede auftreten.

Odometrie

Um die aktuelle Position des Roboters zu bestimmen, können Odometriedaten aufgezeichnet werden. Dazu werden die Drehzahlsensoren der Motoren zur Fortbewegung genutzt, um die Positionsänderung durch eine Bewegung zu bestimmen [BORENSTEIN, 1998]. Mit diesen Werten kann die Translation und Rotation des Roboters über die Zeit bestimmt werden. Diese Methode ist weit verbreitet, kostengünstig und erlaubt hohe Sample Raten. Auch bei diesem Sensor treten Messfehler auf, die sich außerdem kontinuierlich fortpflanzen. Dies kann zu großen Fehlern in der Positionsbestimmung führen, welche immer weiter wachsen, je mehr sich der Roboter bewegt.

Die Fehlerursachen können in systemeigen und systemfremd eingeteilt werden. Systemeigene Ursachen sind z.B. ungleich große Raddurchmesser und unterschiedliche Ausrichtungen der Räder. Systemfremde Ursachen können unebene Untergründe, unerwartete Gegenstände oder Rutschen der Räder sein.

2.1.3 Behandlung von Messfehlern

Wie schon häufig in diesem Kapitel beschrieben wurde, treten bei allen Sensoren Messfehler unterschiedlicher Art und Häufigkeit auf. Ein SLAM Algorithmus sollte möglichst viele Messfehler erkennen, um fehlerhafte Berechnungen und die Fortpflanzung von Fehlern zu verhindern. Dies benötigt Rechenleistung, da während des gesamten SLAM Prozesses immer wieder nach Fehlern bzw. Zusammenhängen, die wiederum Messfehler erkennbar machen sollen, gesucht wird. Um die Unsicherheit der Messwerte und die Zuverlässigkeit der Sensoren zu berücksichtigen, werden diese gewichtet. Die Ergebnisse werden in Wahrscheinlichkeiten angegeben. Das Minimieren von Fehlern ist ein zentraler Bestandteil von SLAM Algorithmen und wird in den folgenden Kapiteln genauer beschrieben.

2.2 Kartierung und Lokalisierung

Die aufgenommenen Sensordaten werden in eine Umgebungskarte eingetragen. Es gibt mehrere Möglichkeiten in welcher Form diese abgespeichert wird. Häufig wird eine Grid Map verwendet. Der Begriff wurde das erste Mal in [MORAVEC, 1985] genutzt und beschreibt eine Einteilung der Umgebung in ein Raster/Gitter. In jeder Gitterzelle wird die Belegtheitswahrscheinlichkeit des zugehörigen gescannten Bereichs gespeichert. Abbildung 2.6 zeigt ein sehr niedrig aufgelöstes Grid, da eine Gridzelle deutlich größer als der Roboter selbst ist. Alle Zellen in dem Grid sind initial mit einer Wahrscheinlichkeit von 50% belegt (grau). Wird ein Gegenstand gemessen (roter Messstrahl), erhöht sich die Belegtheitswahrscheinlichkeit in dieser Zelle. Da sich folglich zwischen diesem Objekt und dem Roboter keine weiteren Objekte befinden, sinkt die Wahrscheinlichkeit dieser Zellen.

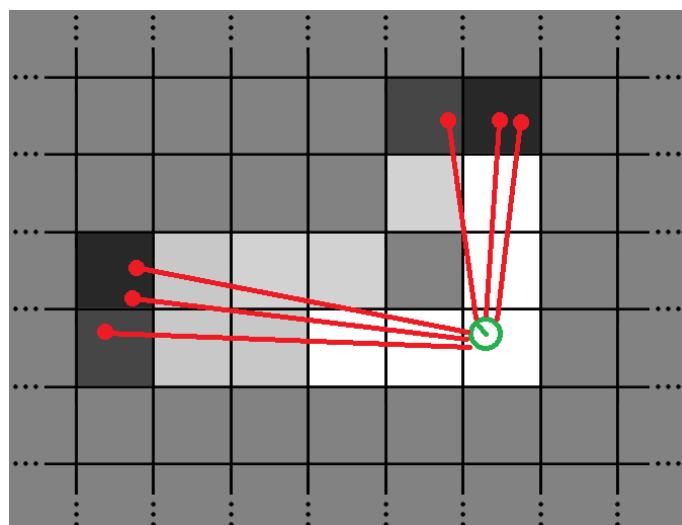


Abbildung 2.6: In der Grid Map sind hohe Belegtheitswahrscheinlichkeiten dunkler und geringere Wahrscheinlichkeiten heller gekennzeichnet. Grün ist die Position des Roboters und rot sind die Abstandsmessungen.

In Abbildung 2.6 sind Zellen mit geringen Werten heller und Zellen mit hohen Werten dunkler markiert. Diese Darstellung der Grid Map wird innerhalb dieser Arbeit konsistent gehalten. Nach mehreren Messungen ergeben sich, je nach Belegtheitswahrscheinlichkeit, unterschiedliche Grautöne. Während sich der Roboter bewegt, erweitert sich das Grid und die Unsicherheit über die Belegung der beobachteten Zellen sinkt mit jeder Messung. Dadurch baut sich eine Umgebungskarte mit „freien“ und „belegten“ Bereichen auf.

Ein Ergebnis ist in Abbildung 2.1 zu sehen. Abhängig von den Sensoren, den verwendeten Algorithmen und der verfügbaren Rechenleistung kann eine Umgebungskarte zwei- oder dreidimensional erzeugt werden. Ein dreidimensionales Grid besteht aus Quadern, in denen die Belegtheitswahrscheinlichkeiten gespeichert werden. In Abbildung 2.2 ist beispielhaft ein Grid in 3D zu sehen.

Wird ein Bereich der Karte mehrmals befahren, ist es möglich, dass vorher gemessene Objekte nicht mehr vorhanden oder neue Objekte hinzugekommen sind. Solche Veränderungen der Umgebung über die Zeit sollten auch in die Karte übernommen werden, um diese so aktuell wie möglich zu halten. Wird eine bereits besuchte Umgebung erneut erkannt, kann ein Kreis geschlossen werden.

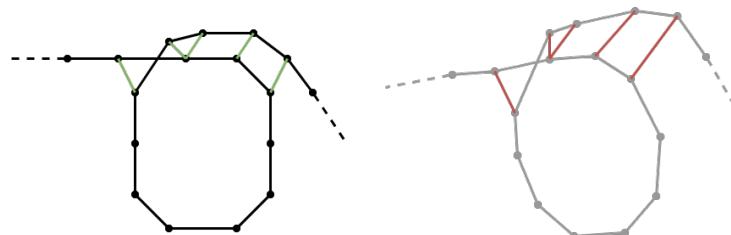


Abbildung 2.7: Kreisschluss: Kartenfehler können behoben werden, wenn Bereiche mehrfach besucht werden. Rechts ist der gemessene Pfad. Links ist der real gefahrene/korrigierte Pfad.^a

^a <https://google-cartographer.readthedocs.io/en/latest/evaluation.html>

Kreisschlüsse sind eine Möglichkeit, um grobe Fehler in der Erstellung der Karte zu korrigieren. Grobe Fehler können Verschiebungen oder Verzerrungen innerhalb der Karte sein, wie in Abbildung 2.7 rechts zu sehen ist. Diese Grafik zeigt links die real gefahrene Strecke und rechts die mit Sensoren gemessene Strecke (Trajektorie). Durch das Wiedererkennen der bereits besuchten Umgebung im Bereich des Schnitts der Trajektorie, können Abhängigkeiten zwischen der ersten und der zweiten Messung aufgestellt werden (rote Linien). Mit Hilfe dieser Abhängigkeiten und den beiden Messungen können die Positionen korrigiert und die gesamte Trajektorie und Umgebungskarte darauf angepasst werden (grüne Linien). In den Abbildungen von Kapitel 3.4.1 ist die Auswirkung von Kreisschlüssen auf die erstellte Karte ersichtlich. Abbildung 3.5 und Abbildung 3.6 zeigen die Ergebnisse von SLAM, die in derselben Umgebung aufgenommen wurden. In Abbildung 3.6 wurden keine Kreise

geschlossen, die zu einer Korrektur führen konnten. Die Umgebungskarte und der gefahrene Pfad weicht von der reell gefahrenen Strecke ab und der Fehler vergrößert sich über die Distanz. In Abbildung 3.5 wurden Kreise gefunden und die Karte korrigiert.

Um neue Messungen korrekt in die Karte eintragen zu können und bereits besuchte Umgebungen wiederzuerkennen, ist es notwendig herauszufinden, welche Bewegungen vom Roboter zwischen den Messungen ausgeführt wurden. Eine Lokalisierung soll die aktuelle Position und Ausrichtung (Pose) des Roboters bestimmen. Die berechnete Pose dient auch einer zielgerichteten Navigation bei gegebener Umgebungskarte.

Es gibt unterschiedliche Möglichkeiten zur Lokalisierung, die in den folgenden Kapiteln genauer erklärt werden.

2.2.1 Matching Verfahren

Aufgrund einer Ausgangsposition kann z.B. mit Scan-to-Scan Matching (kurz Scan-Matching) oder Scan-to-Map Matching (kurz Map-Matching) die neue Position bestimmt werden.

Scan-Matching vergleicht dabei zwei Laserscans miteinander. Zum Vergleich zweier Scans werden mit mathematischen Funktionen die Abweichungen durch einen Wert dargestellt. Die Berechnung des Wertes führt zu einem nicht-linearen Optimierungsproblem. Ziel eines solchen Problems ist die Berechnung des Minimums einer mathematischen Funktion [BÖRLIN, 2013, S. 4]. Eine sogenannte Kostenfunktion beschreibt in diesem Fall die Güte einer Position in einer Karte aufgrund eines an dieser Stelle ausgeführten Laserscans. Im Beispiel eines SLAM Algorithmus entstehen dabei geringere Kosten, wenn gemessene Gegenstände in der Karte an der Position eingetragen werden, die bereits als belegt gekennzeichnet ist. Soll ein Gegenstand an einer freien Stelle in der Karte eingetragen werden, führt das zu höheren Kosten. Diese Funktion wird für alle gemessenen Punkte eines Laserscans berechnet und die Kosten addiert. Je geringer die Kosten sind, desto ähnlicher sind sich die beiden Scans.

Mit einem Iterative Closest Point (ICP) Verfahren wird die Ausgangspose so verschoben, dass möglichst geringe Kosten entstehen. ICP läuft nach [BESL, 1992] in vier Schritten ab und wird so lange wiederholt, bis eine Konvergenz eintritt.

1. Berechne die nächst gelegenen Punkte
2. Berechne die nächste Iteration
3. Wende die berechnete Iteration auf die Punktfolge an
4. Wiederhole, solange die Änderung des quadratischen Fehlers eine festgelegte Grenze nicht unterschreitet

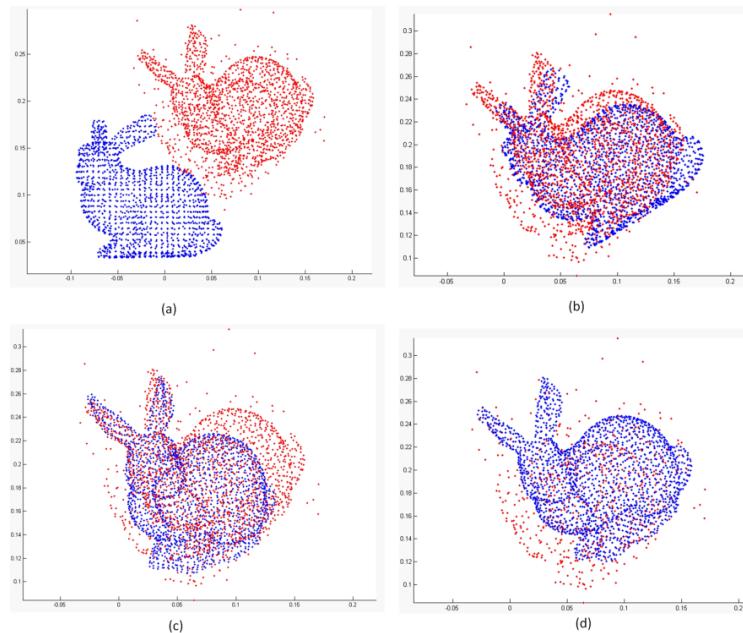


Abbildung 2.8: ICP Verfahren [ZHANG, 2018]: (a) Initiale Punktwellen (b) Ergebnis des ICP Algorithmus (c) Ergebnis des ICP-MCC Algorithmus (d) Ergebnis des ICP-BiMCC Algorithmus

Diese Position beschreibt dann mit hoher Wahrscheinlichkeit die Pose des Roboters zum Zeitpunkt des Laserscans und damit der neuen Position. Diese Position ist in der nächsten Messung die Ausgangsposition.

Die Ergebnisse unterschiedlicher ICP Algorithmen wird in Abbildung 2.8 dargestellt. In (a) sind zwei Punktwellen dargestellt. Die blaue Punktwolke soll so verschoben werden, dass möglichst viele blaue Punkte mit der roten Punktwolke übereinstimmen bzw. der Abstand zwischen allen blauen und roten Punkten minimal ist. In (b), (c) und (d) werden die Ergebnisse unterschiedlicher ICP Algorithmen gezeigt. In diesem Beispiel liefert der ICP-BiMCC Algorithmus in (d) den kleinsten Fehler.

Ein weiteres Matching Verfahren ist Map-Matching. Dieses versucht eine Punktwolke in eine Umgebungskarte zu legen, sodass der Fehler minimal ist. Nach [HESS, 2016] wird Scan-Matching häufig in Laser-basierten SLAM Verfahren genutzt und ist einer hohen Fehlerfortpflanzungsrate ausgesetzt.

Least Squares Adjustment (LSA) ist eine Methode um Parameter einer Beobachtung zu schätzen und damit die aktuelle Position der Beobachtung zu bestimmen. Es wird mit einem funktionalen und einem stochastischen Modell gearbeitet [BÖRLIN, 2013, S. 2]. Das funktionale Modell beschreibt, wie Parameter und Beobachtungen voneinander abhängen, wenn keine Fehler auftreten. Beispielsweise kann damit die Projektion eines Punktes im 3D Raum durch eine Kamera auf eine 2D Fläche (Bild) beschrieben werden. Das stochastische Modell gibt stochastische Eigenschaften der Beobachtung (Verteilung, Mittelwert,

Abweichung) an. Unbekannte Parameter sind die Position des Punktes im 3D Raum und evtl. auch die Kameraparameter. Ausgehend von initialen Werten sollen diese Parameter mit Beobachtungen und beiden Modellen geschätzt werden. Wenn das funktionale Modell ein nicht lineares Problem beschreibt und keine exakte Lösung existiert, kann eine optimale Lösung meist nur mit numerischen Verfahren gefunden werden. Aus diesem Grund sind solche Verfahren rechenaufwändig. SLAM Verfahren sollen die Ergebnisse in Echtzeit liefern und müssen diese Berechnungen möglichst effizient lösen.

Ein Verfahren zur Schätzung unbekannter Parameter ist der Bündelblockausgleich (englisch: Bundle Adjustment). Es wird in der Photogrammetrie bereits seit den 1960er Jahren verwendet [BÖRLIN, 2013, S. 15]. Gegeben sind mehrere aufgenommene Positionen von Features und deren Zusammenhänge [SAMEER AGARWAL, SNAVELY, 2010, S. 3]. Das Ziel ist eine dreidimensionale Rekonstruktion der Umgebung und die Ermittlung der Kameraparameter, sodass der Projektionsfehler minimiert wird ([ENGELS, 2006] und [SAMEER AGARWAL, SNAVELY, 2010, S. 3]). Ein Bündelblockausgleichsproblem kann als Nicht-lineares Optimierungsproblem betrachtet werden. Der Fehler zwischen den beobachteten Features und den rekonstruierten dreidimensionalen Positionen soll minimiert werden.

In der Publikation „Bundle Adjustment in the Large“ [SAMEER AGARWAL, SNAVELY, 2010] wird ein Algorithmus vorgestellt, der aus Tausenden von Bildern eine Umgebung rekonstruiert. Die Testdaten wurden aus zwei unterschiedlichen Quellen entnommen. Die erste Quelle war eine Kamera, die auf einem beweglichen Fahrzeug montiert wurde. Abhängigkeiten zwischen den Bildern wurde über die Aufnahmezeit und GPS Informationen erzeugt. Die Bilder der zweiten Quelle stammen von der Internetplattform „Flickr.com“. Abhängigkeiten zwischen den Bildern wurden bereits in der Publikation [S. AGARWAL, 2009] erstellt. Weiterhin wurde ein „Skeletal Set“ [SNAVELY, 2008] erzeugt. Dies ist eine Teilliste mit Bildern, in denen für das Bundle Adjustment nicht notwendige Bilder aussortiert wurden. Jedes Bild besitzt Informationen über den Zusammenhang zu anderen Bildern der Liste. Die Bilder, Features und deren Abhängigkeiten wurden an eine modifizierte Anwendung namens „Bundler“¹ übergeben. Das Programm kann dann Punkte unterschiedlicher Bilder miteinander verknüpfen und ein dreidimensionales Umgebungsmodell erstellen.

Einer der beliebtesten Algorithmen, um nicht-lineare Optimierungsprobleme und Bundle Adjustment zu lösen, ist von Levenberg-Marquardt (LM). Es existieren dafür Umsetzungen in vielen Programmiersprachen wie z.B. Java (BoofCV, DDogleg), C/C++/C#(ALGLIB, Ceres Solver, levmar), matlab (LMFnlsq, SMFsolve).

¹ <http://www.cs.cornell.edu/~snavely/bundler/>

Ein wesentlicher Unterschied zwischen Bundle Adjustment und Matching Verfahren besteht bereits in der Datengrundlage. Bundle Adjustment extrahiert Features aus Bildern und kombiniert diese miteinander. Matching Verfahren verwenden dagegen Laserscans und vergleichen diese.

2.2.2 Kalman- und Partikelfilter

Der Bayes-Filter ist ein Algorithmus, der mit Hilfe der vorherigen Position und neuen Sensormessungen eine neue Position (Posterior/Belief) schätzt [THRUN, 2006]. Der Algorithmus wird für jede Schätzung rekursiv aufgerufen. Da mit einer Positionsschätzung eine Lokalisierung in einer Umgebung möglich ist, eignet sich ein Bayes Filter als Alternative zu Matching Verfahren.

Der Kalman-Filter ist eine Möglichkeit zur Implementierung des Bayes-Filters. Die Technik existiert bereits seit 1958 und ist wahrscheinlich eine der am besten erforschte Technologie zur Filterung und Vorhersage von aufeinander folgenden Zuständen. Folgende zwei Schritte werden abwechselnd ausgeführt:

- Vorhersage aufgrund der zuletzt berechneten Position (engl. Prediction)
- Korrektur der Schätzung aufgrund der neuen Sensorwerte bzw. Beobachtungen (engl. Correction)

Voraussetzungen für die Verwendung des Kalman-Filters sind die Linearität der Parameter der Zustandsübergänge und Beobachtungen sowie eine Normalverteilung der Messfehler (Rauschen). Ein Einsatzgebiet ist die Navigation im Auto, um die Position auf Strecken ohne GPS oder Tunnel schätzen zu können. Z.B. können Kompass, Beschleunigungssensoren oder Drehzahlsensoren an den Rädern die notwendigen Signale für eine Korrektur liefern.

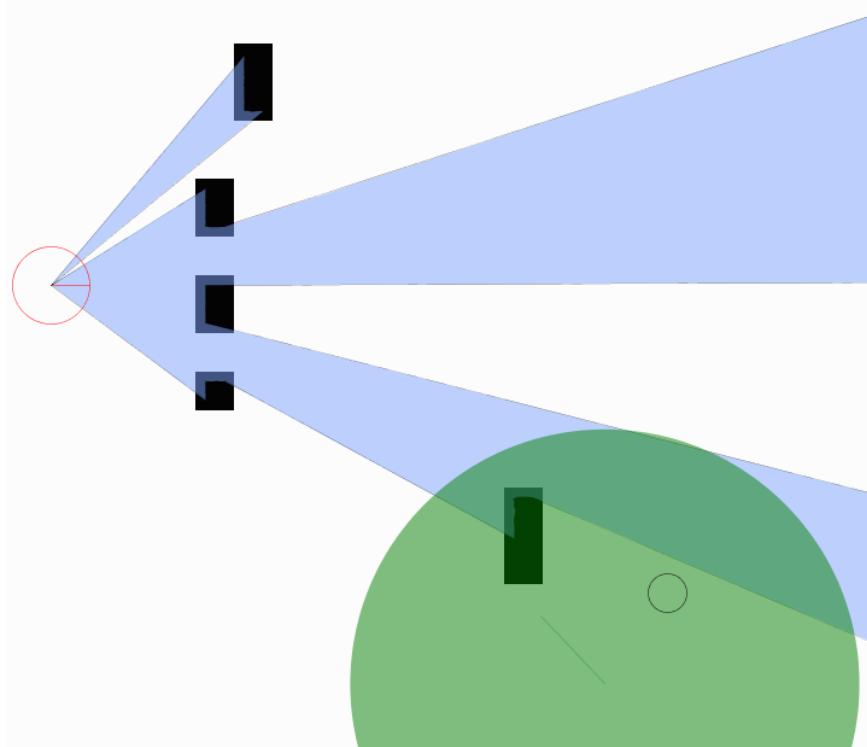


Abbildung 2.9: Beispiel für Kalmanfilter: Der rote Kreis (links) ist ein Roboter, der den schwarzen Kreis (rechts) beobachten und dessen Position vorhersagen soll, wenn die nicht sichtbar ist. Das schwarze Rechteck unten verdeckt die aktuelle Position des Kreises. Der grüne Kreis gibt den möglichen Bereich des schwarzen Kreises an, nachdem dieser hinter dem Hindernis verschwunden ist.

In Abbildung 2.9 soll der rote Kreis die Position des schwarzen Kreises beobachten. Verschwindet das Objekt hinter einem anderen Gegenstand (hier schwarzes Rechteck), wird der Bereich für eine mögliche Position geschätzt und mit einem grünen Kreis angezeigt. Sobald der schwarze Kreis wieder in den Sichtbereich tritt, verkleinert sich der grüne Kreis durch die Korrektur des Kalman-Filters.

Verhalten sich die Parameter nicht linear oder sind die Störeinflüsse nicht normalverteilt, eignet sich ein Partikel-Filter zur Positionsschätzung. Dieser Schätzer nutzt eine Menge gewichteter Samples/Partikel, die mögliche Posen repräsentieren. Aufgrund neuer Beobachtungen werden Partikel höher gewichtet, wenn deren Pose zur Beobachtung passt, oder sonst niedriger gewichtet. Sinkt die Wahrscheinlichkeit eines Partikels unter eine vorher festgelegte Grenze, werden diese aussortiert.

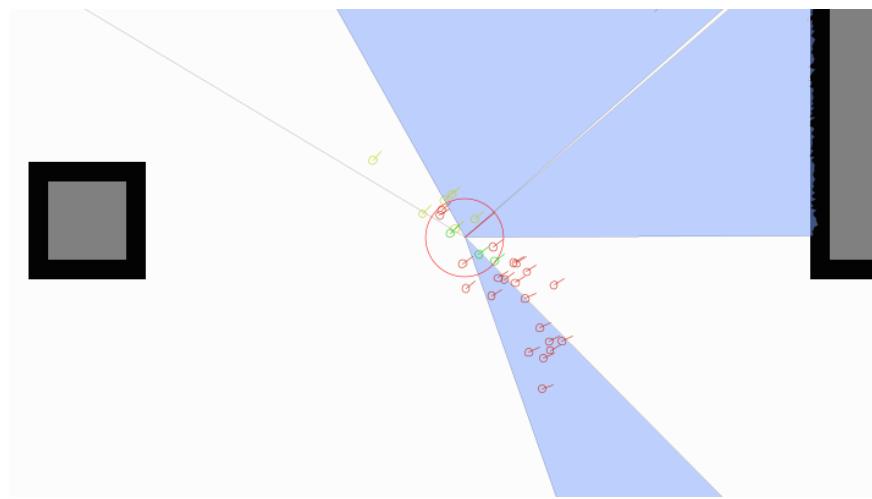


Abbildung 2.10: Beispiel für Partikelfilter: Der große Kreis in der Mitte ist die aktuelle Position des Roboters. Die kleinen Kreise sind die Partikel. Rote Partikel entsprechen nicht der aktuellen Beobachtung und grüne repräsentieren eine mögliche Position.

In Abbildung 2.10 sind rote Partikel unwahrscheinliche Posen und würden im nächsten Schritt aussortiert. Grüne Samples entsprechen der aktuellen Beobachtung. Deswegen können an diesen Stellen neue Partikel generiert werden, um die roten zu ersetzen.

Kapitel 3

Google Cartographer

Die erste Zielstellung dieser Arbeit ist die Analyse des Google Cartographers. Dies ist ein von Google erstellter Simultaneous Localization And Mapping (SLAM) Algorithmus. Im Vergleich zu vielen anderen SLAM Algorithmen unterscheidet sich der Cartographer in seiner Funktionsweise und seinen Features. Diese Unterschiede werden in diesem Kapitel genauer betrachtet.

3.1 Hardwareanforderungen

Der Cartographer ist für eine Vielzahl an möglichen Sensorkonfigurationen ausgelegt.

Einige davon sind in Demos auf der Dokumentationsseite² zu sehen. Das Deutsche Museum in München wurde in 2D und 3D durch einen mit Sensoren bestückten Rucksack kartiert³. Der Träger des Rucksacks lief durch die Gänge des Museums, während der Cartographer die Umgebung aufnahm und als Karte auf einem verbundenen Android Tablet darstellte. Auf dem Tablet konnten dann z.B. Markierungen für interessante Orte hinzugefügt werden. Mit einer ähnlichen Konfiguration wurde das 39 Stockwerke hohe Hotel San Francisco Marriott Marquis kartiert und in Google Maps publiziert. Eine weitere Demo wurde mit dem Revo Laser Distance Sensor eines Saugroboters von Neato Robotics aufgenommen. Die Kosten [KONOLIGE, 2008] für das Gerät betragen unter 30\$. Weitere Demos wurden mit einem humanoiden Roboter (PR2 R&D humanoid robot) und einem Mess- und Erkundungsroboter (Taurob) aufgenommen. Dies demonstriert, wie gut der Cartographer mit unterschiedlicher Hardware umgehen kann.

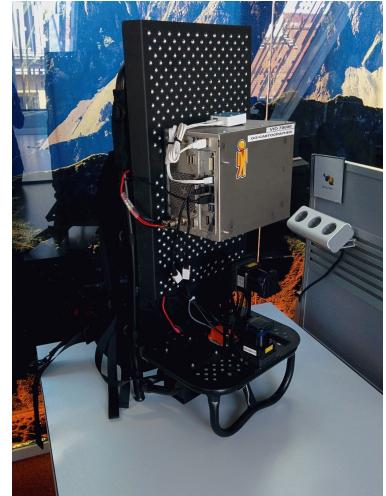


Fig. 3.1: Ein Rucksack, der mit mehreren Sensoren für den Cartographer ausgestattet ist.¹

1 Bildquelle: <https://maps.googleblog.com/2014/09/making-of-maps-cornerstones.html>

2 ROS Demo Bags: <https://google-cartographer-ros.readthedocs.io/en/latest/demos.html>

3 Google Blog: <https://maps.googleblog.com/2014/09/making-of-maps-cornerstones.html>

3.2 Funktionsweise

Auf der Dokumentationsseite⁴ wird ein Überblick über die Funktionsweise des Cartographer Algorithmus gegeben.

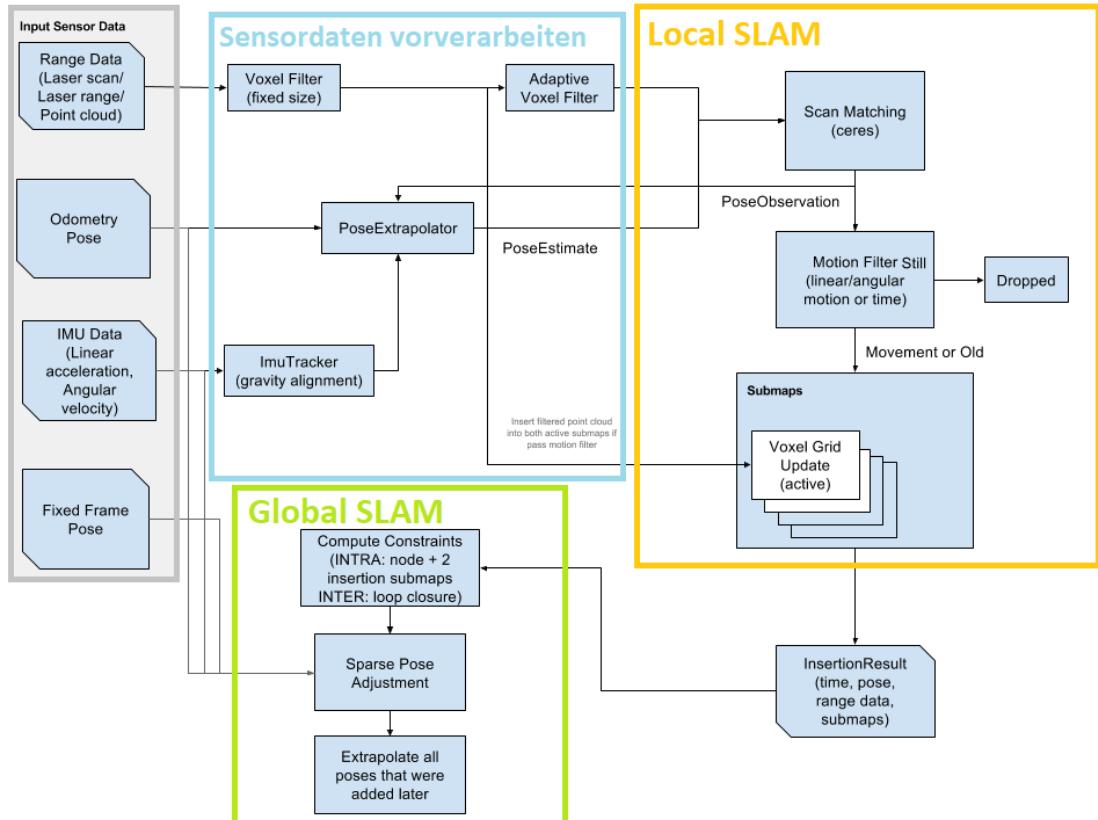


Abbildung 3.2: Module des Google Cartographers: In der Umsetzung dieser Arbeit wurde der grau markierte Bereich durch eine Java Schnittstelle ausgetauscht, wie in Kapitel 4.2 näher beschrieben wird.

Durch die Analyse des Cartographers konnte diese Dokumentation erweitert werden um die Unterschiede zu anderen SLAM Algorithmen zu zeigen.

Die Kommunikation zwischen den Komponenten funktioniert über TCP/IP mittels Protobuf Dateien. Protobuf ist ein von Google entwickeltes Dateiformat zur Serialisierung. Der Cartographer basiert auf dem Robot Operating System (ROS)⁵. ROS ist eine Menge von Software Tools, die zur Entwicklung von Anwendungen für Roboter verwendet werden. Es stellt unter anderem Softwarebibliotheken, Gerätetreiber, Visualisierungsmöglichkeiten und Kommunikationsmöglichkeiten zur Verfügung. Die zur Kommunikation verwendeten ROS Messages werden vom Cartographer zur Eingabe der Daten vom Roboter genutzt.

⁴ Cartographer ROS Dokumentation: https://google-cartographer-ros.readthedocs.io/en/latest/algo_walkthrough.html

⁵ <http://wiki.ros.org/>

Um den Cartographer auf unterschiedliche Umgebungen und Systeme anzupassen, können Konfigurationsparameter gesetzt werden.

3.2.1 Eingaben und Vorverarbeitung

Abstandssensoren (z.B. Light Detection And Ranging (LIDAR)) dienen zum Messen von Entfernungen in der Umgebung. In der Übersicht (Abbildung 3.2) ist dieser Bereich grau gekennzeichnet. Die Sensoren sind nicht Bestandteil des Cartographer Algorithmus, sondern dienen nur der Datenübergabe. Die Konfiguration von `TRAJECTORY_BUILDER_nD.min_range` und `TRAJECTORY_BUILDER_nD.max_range` ist notwendig um fehlerhafte Messungen und Sensorrauschen zu filtern. Eine Vorverarbeitung der Daten findet im blau markierten Bereich der Abbildung 3.2 statt. Gefiltert werden z.B. fehlerhafte Messungen eines Roboterteils (fehlerhafte Positionierung des Sensors) oder falsche Informationen durch Schmutz auf dem Sensor. Die fehlerhaften Daten (die über `max_range` liegen) werden durch `TRAJECTORY_BUILDER_2D.missing_data_ray_length` ersetzt. Grundsätzlich sind alle Distanzangaben in Meter definiert. Während sich der Roboter bewegt werden Messdaten über eine gewisse Zeit gesammelt und dann als ROS Message weiter gegeben.

Die Messungen können durch Timestamps unabhängig betrachtet werden, um mögliche Verzerrungen durch Bewegungen zu erkennen. Je öfter Daten an den Cartographer übergeben werden, desto besser kann dieser die Messungen entzerren und zu einem Scan zusammenfügen. Die Anzahl der zu sammelnden Messungen kann durch `TRAJECTORY_BUILDER_nD.num_accumulated_range_data` konfiguriert werden.

Normalerweise werden nahe Gegenstände öfter gemessen, als für SLAM notwendig. Weit entfernte Gegenstände werden dagegen seltener gemessen. Um den Rechenaufwand für den Cartographer zu reduzieren, sollen nicht benötigte Punkte gefiltert werden. Dazu kommen zwei Voxel Filter zum Einsatz, die mit sogenannten Cubes (Würfel) die Messpunkte filtern, indem diese über den gemessenen Raum verteilt werden und damit ein räumliches Gitter bilden. Im ersten Voxel Filter sind die Kantenlängen der Cubes fest. Jeder Messpunkt wird einem Würfel zugeordnet. Im Anschluss werden alle Cubes aussortiert, denen kein Messpunkt zugeordnet wurde und der Mittelpunkt der restlichen Cubes sind die gefilterten Messpunkte. Die Größe der Cubes bestimmt die Dichte der Punktfolke. Sind sie zu groß, entsteht viel Datenverlust. Sind sie zu klein, erhöht sich der Rechenaufwand. Mit der Konfiguration `TRAJECTORY_BUILDER_nD.voxel_filter_size` kann die Größe festgelegt werden.

Nach dem Voxel Filter mit fester Größe wird ein zweiter Voxel Filter eingesetzt, der die Größe der Cubes dynamisch bestimmt. Zur Bestimmung der Größe wird eine Anzahl an Punkten festgelegt, die ein Cube enthalten soll.

Für ein 3D-Umgebungsmodell mit SLAM werden zwei dynamische Voxel Filter mit unterschiedlicher Auflösung benutzt. Die dynamischen Voxel Filter können in der maximalen Kantenlänge (`TRAJECTORY_BUILDER_nD.*adaptive_voxel_filter.max_length`) und der Mindestanzahl an Punkten (`TRAJECTORY_BUILDER_nD.*adaptive_voxel_filter.min_num_points`) konfiguriert werden.

Zusätzlich zu den Entfernungsmessungen können Inertial Measurement Unit (IMU)-Daten verwendet werden (`TRAJECTORY_BUILDER_2D.use_imu_data`). Für eine 2D-Karte ist dies optional und für 3D-SLAM notwendig. Diese Daten verbessern die Ergebnisse des Cartographers, da durch Gravitation und Rotationswerte die Lage und Position des Roboters besser bestimmt werden kann. In 3D wird die Gravitation als Initialschätzung benutzt, um die Lage des Roboters zu bestimmen. Zudem reduziert es den Aufwand beim Scan-Matching. Mit `TRAJECTORY_BUILDER_nD imu_gravity_time_constant` kann konfiguriert werden, über welche Zeit der Roboter die Gravitationsdaten sammelt, um Rauschen zu filtern.

Nach der Vorverarbeitung der Daten werden diese an LocalSLAM weitergegeben.

3.2.2 LocalSLAM

LocalSLAM ist in Abbildung 3.2 auf Seite 20 orange markiert und dient dem einfügen von Scans in eine Submap. Der Cartographer verwendet keinen Partikelfilter (erklärt in Kapitel 2.2.2) um die Hardwareanforderungen möglichst gering zu halten.

Der initiale Schätzwert (Prior für Ceres) kommt von einem „Pose Extrapolator“. Dieser nutzt neben LIDAR auch Odometriedaten und IMU Daten, um eine Position zu bestimmen, an welcher der nächste Scan in der Submap eingefügt werden kann.

Hier gibt es zwei Möglichkeiten:

1. Ceres Scan Matcher - Scan-Matching (empfohlen):

Mit einer non-linear-least-square Methode wird die Position in der Submap bestimmt, die am besten zu den Subpixeln des Scans passt. Ceres stellt eine Transformationsmatrix auf, die zwei Scans passend übereinanderlegt und daraus die Bewegung ermittelt. Zum Aufstellen dieser Matrix wird ein Gradientenverfahren mit einer festen Anzahl Iterationen genutzt. Die Anzahl kann konfiguriert werden, um das Verfahren schneller konvergieren zu lassen.

$$\underset{\xi}{\operatorname{argmin}} \sum_{k=1}^K \left(1 - M_{\text{smooth}}(T_{\xi} h_k) \right)^2 \quad (3.1)$$

$$T_{\xi} p = \underbrace{\begin{pmatrix} \cos \xi_{\theta} & -\sin \xi_{\theta} \\ \sin \xi_{\theta} & \cos \xi_{\theta} \end{pmatrix}}_{R_{\xi}} p + \underbrace{\begin{pmatrix} \xi_x \\ \xi_y \end{pmatrix}}_{t_{\xi}} \quad (3.2)$$

$h_k \dots$ Submap-Frame

$\xi \dots$ Pose

$M \dots$ Probability Grid

$T_{\xi} p \dots$ Transformationsmatrix

$R_{\xi} \dots$ Rotationsmatrix

$t_{\xi} \dots$ Translationsmatrix

Die Formel 3.1 transformiert den Scan-Frame T passend zum Submap-Frame h_k und findet als Ergebnis die Pose, an welcher der Scan in die Submap eingefügt werden kann. Zur Berechnung wird die Transformationsmatrix in Formel 3.2 verwendet. Messfehler, die deutlich größer als die Grid Map-Auflösung der Submaps sind, können nicht erkannt werden.

Der Ceres Scan Matcher kann über die Konfiguration auf das verwendete System angepasst werden. Er sollte genutzt werden, wenn die Sensoren regelmäßig Messdaten liefern und nur wenig fehlerbehaftet sind. Über die Konfiguration kann jedem Sensor ein Gewicht zugeordnet werden, welches die Zuverlässigkeit des Sensors repräsentiert. Aufgrund dieses Gewichts wird der jeweilige Sensor mehr bzw. weniger in die Berechnung einfließen.

2. Real-Time-Correlative-Scan-Matcher - (Sub)map-Matching:

Falls keine zusätzlichen Sensoren (außer LIDAR) verfügbar oder die Werte nicht zuverlässig (unregelmäßig, stark fehlerbehaftet) sind, muss der Real-Time-Correlative-Scan-Matcher verwendet werden. In diesem Fall werden die Messdaten, ähnlich dem Kreisschluss in Kapitel 2.2, gegen die aktuelle Submap gematcht. Es wird nach ähnlichen Scans innerhalb eines definierten Suchfensters gesucht. Das Suchfenster ist definiert durch Entfernung und Winkel. Für dieses Matching können unterschiedliche

Gewichte für Rotation und Translation konfiguriert werden, wenn sich z.B. der Roboter nicht viel dreht, da ein langer Gang gescannt werden soll. Das Matching wird dann als Prior für Ceres genutzt. Die Nachteile des Real-Time-Correlative-Scan-Matcher sind die benötigte hohe Rechenleistung und das Überschreiben aller Sensorsignale außer der Tiefenmessung. Der Vorteil ist, dass diese Methode sehr stabil in Umgebungen mit vielen Features ist und keine zusätzlichen Sensoren benötigt werden. In der Publikation [OLSON, 2009] wird die Funktionsweise dieses Scan-Matchers genauer erklärt und mit anderen Matching Verfahren (z.B. Iterative Closest Point (ICP)), in Hinsicht auf Performance und Qualität der Ergebnisse, verglichen. Die Performanceanalyse zeigt eine meist höhere, aber echtzeitfähige, Berechnung der Ergebnisse. Der Algorithmus unterstützt die Berechnung auf einer GPU, um die Geschwindigkeit zu erhöhen und die CPU für andere Anwendungen nutzen zu können. In der Qualität der Ergebnisse ist der Real-Time-Correlative-Scan-Matcher meistens besser.

Wenn beim Scan-Matching eine Bewegung erkannt wurde, wird diese durch einen „Motion Filter“ überprüft. Ist die Bewegung signifikant genug wird der Scan in die Submap eingetragen, ansonsten nicht betrachtet. Die Kriterien für die Signifikanz einer Bewegung sind eine Zeitspanne (`max_time_seconds`), ein Winkel (`max_angle_radians`) und eine Distanz (`max_distance_meters`). Diese drei Parameter können über `TRAJECTORY_BUILDER_nD.motion_filter.*` konfiguriert werden. Wird keiner der drei Parameter überschritten, ist die Bewegung nicht signifikant.

Die signifikanten Scans werden in zwei identische Submaps eingetragen, die aus einem Grid bestehen. Wenn eine konfigurierte Anzahl an Scans in eine Submap eingefügt wurde, ist diese Submap abgeschlossen (`TRAJECTORY_BUILDER_nD.submaps.num_range_data`) und es können keine weiteren Scans eingetragen werden. LocalSLAM kann ab diesem Zeitpunkt die Submap nicht mehr verändern.

Nach der Fertigstellung der beiden Submaps wird

- die erste Karte in die Gesamtkarte integriert und
- die zweite Karte für das Scan-Matching der nächsten Submap verwendet.

Der Ursprung des ersten eingetragenen Scans bestimmt den Ursprung (Origin) der Submap. Eine Submap startet mit einem Grid der Größe 100×100 . Liegt die Messung außerhalb der aktuellen Größe der Submap, wird das Grid in x - und y -Dimension verdoppelt.

Über längere Zeitspannen entstehen jedoch Fehler, die sich fortpflanzen. Solche Fehler sind z.B. Verschiebungen oder Krümmungen (Drifts). Die Fehlerfortpflanzung ist bei scan-to-scan Matching höher als bei scan-to-map Matching [HESS, 2016]. Deshalb verwendet der Cartographer hauptsächlich scan-to-map (bzw. Submap) Matching.

Um die Fehler so gering wie möglich zu halten, soll GlobalSLAM diese Drifts erkennen und beheben. Die Voraussetzung dafür ist, dass die Submaps jeweils korrekt sind, da GlobalSLAM nur die Krümmungen zwischen Submaps erkennen kann. Ihre Größe bzw. die Anzahl der einzutragenden Scans ist so zu konfigurieren, dass die Drifts in einer Submap geringer sind als deren Auflösung. Ist diese zu groß gewählt (Krümmung innerhalb der Submap erkennbar), kann GlobalSLAM den Drift nicht erkennen. Mit einer zu kleinen Größe funktioniert unter Umständen der Kreisschluss nicht korrekt und damit könnte eine Korrektur auch fehlschlagen.

Es gibt zwei mögliche Datenstrukturen, um die Entfernungsdaten in einer Submap zu speichern (`TRAJECTORY_BUILDER_2D.submaps.grid_options_2d.grid_type`). Die erste Möglichkeit ist das Probability Grid, welches für 2D und 3D SLAM verwendet werden kann. Das Probability Grid ist ein Raster, welches die Submap in Zellen einteilt. Jede Zelle enthält einen Wert, der beschreibt, mit welcher Wahrscheinlichkeit diese Zelle belegt ist. Belegte Zellen sind z.B. Wände oder Gegenstände. Diese Werte werden bei einem Hit oder Miss aktualisiert. Jede Distanzmessung endet an einem Hindernis. Diese gemessene Distanz geht als Hit für die jeweils gemessene Zelle in das Grid ein. Alle Zellen zwischen Sensor und Hindernis werden mit einem Miss belegt.

Hits und Misses können, je nach Vertrauenswürdigkeit des Sensors, unterschiedliche Gewichtungen bei der Berechnung der Belegtheitswahrscheinlichkeiten besitzen. Für den 2D Modus wird pro Submap ein Grid gespeichert. Für den 3D Modus werden pro Submap zwei hybride Grids gespeichert: ein hochauflösendes Grid für nahe Messungen und ein niedrig aufgelöstes Grid für weite Entfernung.

Eine weitere mögliche Datenstruktur zur Speicherung der Entfernungsdaten ist Truncated Signed Distance Fields (TSDF), die jedoch nur für 2D SLAM verfügbar ist [CANELHAS, 2017]. Ein TSDF funktioniert ähnlich der Probability Grid Methode. Bei dieser wird auch ein zweidimensionales Grid erstellt. Jede Gridzelle speichert die Entfernung zur jeweils nächstgelegenen Oberfläche. Hier kann durch `options.update_free_space` konfiguriert werden, ob auch andere Zellen entlang des Messstrahls bis zum Gegenstand aktualisiert werden sollen oder nur die innerhalb der Truncation Distance um den Hit liegenden Zellen.

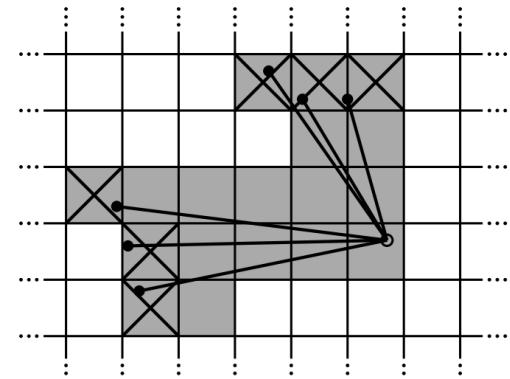


Fig. 3.3: Probability Grid mit Hits (grau mit Kreuz) und Misses (grau) [HESS, 2016]

3.2.3 GlobalSLAM

LocalSLAM erzeugt eine Folge von Submaps, die jeweils in sich nur wenige Fehler besitzen sollten. Jedoch können Fehler submapübergreifend auftreten. GlobalSLAM ist in Abbildung 3.2 auf Seite 20 grün markiert und dient der Fehlerkorrektur im Hintergrund. Dabei werden die Submaps so zueinander ausgerichtet, dass eine global kohärente Map entsteht. Diese Optimierung verändert die aktuell in Konstruktion befindliche Trajektorie indem Kreisschlüsse gesucht und die Submaps daran angepasst werden. Eine solche Optimierung findet statt, sobald eine konfigurierte Anzahl an Knoten der Trajektorie hinzugefügt wurde. Mit der Konfiguration `POSE_GRAPH.optimize_every_n_nodes` kann diese Anzahl festgelegt werden. Ist der Wert auf null gesetzt, wird kein GlobalSLAM durchgeführt. Dies ermöglicht ein Testen und Debuggen von LocalSLAM z.B. um Konfigurationsparameter zu optimieren.

GlobalSLAM versucht Abhängigkeiten zwischen Knoten und Submaps festzustellen und den daraus entstandenen Abhängigkeitsgraphen zu optimieren. Es entsteht ein Pose-Graph. Zum besseren Verständnis kann man die Abhängigkeiten als Seile interpretieren, die mit Knoten zusammen gehalten werden. Im Pose-Graph werden dann alle Verbindungen fester zusammen gezogen, sodass sich eine eindeutige Trajektorie bildet. Mit Rviz, einer Anwendung zur Steuerung und Anzeige der Ergebnisse, kann `POSE_GRAPH.constraint_builder.log_matches` genutzt werden, um ein Histogramm über die gefundenen Abhängigkeiten zu erstellen.

Es gibt zwei Typen von Abhängigkeiten. Nicht-globale Abhängigkeiten sind nah aufeinander folgende Knoten auf einer Trajektorie innerhalb einer Submap. Globale Abhängigkeiten werden zwischen neuen Submaps und bereits erstellten „nahen“ Submaps innerhalb eines Suchfensters erzeugt (Kreisschluss). Die Distanzen können konfiguriert werden. Weiterhin können auch Abhängigkeiten zwischen verschiedenen Trajektorien genutzt werden z.B. durch mehrere Roboter oder mehrere Durchläufe.

$$\begin{aligned}\Xi^m &= \{\xi_i^m\}_{i=1,\dots,n_m} \\ \Xi^s &= \{\xi_j^s\}_{j=1,\dots,n_s}\end{aligned}\tag{3.3}$$

Die Gleichung 3.4 beschreibt ein nichtlineares Optimierungsproblem. Wie in 3.3 dargestellt, enthalten die Mengen Ξ^m und Ξ^s die Posen aller Submaps bzw. Scans. Entsprechend ist ξ_i^m und ξ_j^s die Pose einer Submap i und eines Scans j . ξ_{ij} ist die Stelle an der ein bestimmter Scan i in einer Submap j gematcht wurde. Die Kovarianzmatrix Σ_{ij} kann z.B. von Ceres geschätzt werden. Über relative Posen ξ_{ij} und die damit verknüpften Kovarianzmatrizen Σ_{ij} können Abhängigkeiten erstellt werden. E ist eine Residuumfunktion zur Bewertung des Fehlers der Abhängigkeiten. ρ ist eine Verlustfunktion um die Auswirkung von Ausreißern

zu minimieren. Die Berechnung wird in der Publikation [HESS, 2016] genauer beschrieben. Mehrmals pro Minute wird mit Ceres eine Lösung berechnet um alle Posen zu optimieren.

$$\operatorname{argmin}_{\Xi^m, \Xi^s} \frac{1}{2} \sum_{ij} \rho(E^2(\xi_i^m, \xi_j^s; \Sigma_{ij}, \xi_{ij})) \quad (3.4)$$

$$E^2(\xi_i^m, \xi_j^s; \Sigma_{ij}, \xi_{ij}) = e(\xi_i^m, \xi_j^s; \xi_{ij})^T \Sigma_{ij}^{-1} e(\xi_i^m, \xi_j^s; \xi_{ij}) \quad (3.5)$$

$$e(\xi_i^m, \xi_j^s; \xi_{ij}) = \xi_{ij} - \begin{pmatrix} R_{\xi_i^m}^{-1}(t_{\xi_i^m} - t_{\xi_j^s}) \\ \xi_{i;\theta}^m - \xi_{j;\theta}^s \end{pmatrix} \quad (3.6)$$

m . . . Submap bzw. i als Index

s . . . Scan bzw. j als Index

Ξ . . . Menge von Posen

ξp . . . einzelne Pose

Σ . . . Kovarianzmatrix

E . . . Residuum

ρ . . . Verlustfunktion (z.B. Huber-Loss)

Um die Anzahl der Abhängigkeiten und damit den Rechenaufwand in Grenzen zu halten, wird nur eine Teilmenge aller nahen Knoten zur Bestimmung von Abhängigkeiten genutzt. Die Menge kann über `POSE_GRAPH.constraint_builder.sampling_ratio` konfiguriert werden. Bei der Nutzung zu weniger Knoten können nur wenige Abhängigkeiten gefunden werden und der Kreisschluss funktioniert nicht optimal. Wenn zu viele Knoten betrachtet werden, beeinträchtigt dies die Rechenzeit und es ist evtl. nicht möglich in Echtzeit Kreisschlüsse zu finden.

3.2.4 Fast Correlative Scan Matcher

Wurde eine Abhängigkeit gefunden, überprüft der FastCorrelativeScanMatcher dieses Matching und vergibt einen Score. Der Matcher basiert auf einem „Branch and Bound“ Mechanismus und kann in unterschiedlich aufgelösten Maps falsch erkannte Abhängigkeiten erkennen, damit diese im Anschluss eliminiert werden. „Branch and Bound“ dient zur Lösung ganzzahliger Optimierungsprobleme. Für eine Lösung wird das Problem in einfacher zu lösende Teilprobleme aufgeteilt (branch). Mit unteren und oberen Schranken (Heuristik) werden die Teilprobleme aussortiert (bound), bis eines der Teilprobleme als optimale Lösung akzeptiert wird.

Mittels Gleichung 3.7 kann eine pixelgenaue Pose ξ^* berechnet werden.

$$\xi^* = \underset{\xi \in \mathcal{W}}{\operatorname{argmax}} \underbrace{\sum_{k=1}^K M_{\text{nearest}}(T_\xi h_k)}_{\text{Score}} \quad (3.7)$$

\mathcal{W} . . . Suchfenster

M_{nearest} . . . Grid M erweitert auf \mathbb{R}^2

Eine genaue Beschreibung der Berechnung und des Algorithmus existiert in der Publikation von [HESS, 2016, S. 3 ff.]. Wenn der Score einer Abhängigkeit ein konfigurierbares Minimum überschreitet (`POSE_GRAPH.constraint_builder.min_score`), korrigiert der Ceres Scan Matcher im Anschluss die Pose. Der Berechnungsaufwand den Ceres erzeugt, kann über verschiedene Konfigurationsparameter gesteuert werden. Die Korrektur geschieht durch die Berechnung mehrerer Residuen, welche wiederum durch gewichtete Kostenfunktionen berechnet werden. Durch die Kostenfunktionen werden mehrere Datenquellen ausgewertet: globale Abhängigkeiten (Kreisschlüsse), nicht globale Abhängigkeiten (Scan-Matching), Beschleunigungs- und Rotationswerte der IMU, Schätzung der Pose, Odometriedaten oder GPS Daten. Die Gewichte der Eingabedaten können konfiguriert werden. Über den Parameter `POSE_GRAPH.max_num_final_iterations` können zusätzliche Informationen über die genutzten Residuen ausgegeben werden. Die Berechnung des IMU Residuums erlaubt, je nach Verlässlichkeit der Werte, eine flexible IMU Pose. Ceres versucht die extrinsische Kalibrierung zwischen der IMU Pose und dem aktuell getrackten Frame zu optimieren. Wenn die IMU Pose nicht zuverlässig ist, können die Ergebnisse der globalen Optimierung von Ceres durch die Konfiguration von `POSE_GRAPH.optimization_problem.log_solver_summary` geloggt werden. Dadurch ist es möglich, die extrinsische Kalibrierung zu verbessern. Wenn man sich

stattdessen sicher ist, dass die extrinsische Konfiguration korrekt ist, kann die Pose konstant gesetzt werden.

Der Einfluss von Ausreißern kann durch eine Huber-Loss Funktion gesteuert werden. Zur Skalierung der Funktion dient die Konfiguration `POSE_GRAPH.optimization_problem.huber_scale`. Je höher die Skalierung ist, desto größer ist der Einfluss von Ausreißern. Nachdem alle globalen Optimierungen abgeschlossen sind und die Trajektorie fertig gestellt ist, wird erneut eine globale Optimierung mit meist deutlich mehr Iterationen, als bei den vorherigen globalen Optimierungen, gestartet. Diese muss dann nicht mehr in Echtzeit abgeschlossen werden, da alle Scans bereits abgeschlossen sind. Daher ist es ratsam möglichst viele Iterationen zu konfigurieren (`POSE_GRAPH.max_num_final_iterations`). Diese abschließende Optimierung dient der Glättung des Ergebnisses.

3.3 Jackal Cartographer Demo

Jackal ist eine mobile Roboterplattform, die mit GPS und IMU ausgestattet ist und bereits ab Werk mit ROS ausgeliefert wird. Als Zusatzmodul kann unter anderem ein LIDAR Scanner verbaut werden, wie in Abbildung 3.4 abgebildet ist.



Abbildung 3.4: Jackal Roboterplattform von Clearpath Robotics⁶

Auf der Jackal Webseite⁷ gibt es ein Shellskript welches den Cartographer und dessen Abhängigkeiten auf ein bereits vorhandenes System mit ROS installiert. Es wird auch eine Erweiterung installiert, die Jackal im Gazebo Simulator verfügbar macht. Der Simulator wird mit dem Cartographer verbunden. Die Steuerung und Anzeige der Ergebnisse erfolgt mit Rviz, einem Paket, das mit ROS installiert werden kann. Mit Hilfe dieser Demo war

⁶ Bildquelle: <https://clearpathrobotics.com/jackal-small-unmanned-ground-vehicle/>

⁷ Jackal Webseite: <https://www.clearpathrobotics.com/assets/guides/jackal/cartographer.html>

es möglich, einige Algorithmen und Abläufe des Cartographers genauer zu verstehen und Testdaten für den Praxisteil dieser Arbeit aufzunehmen (Kapitel 4.2).

3.4 Ergebnisse mit Jackal

Für einen Eindruck der Ergebnisse des Cartographers wurden mit dem Jackal im Simulator zwei unterschiedliche Karten mit verschiedenen Parametern kartografiert. Die im Folgenden gezeigten Karten wurden in möglichst schwierig zu kartografierenden Umgebungen und Situationen aufgenommen, um die Stabilität des SLAM Algorithmus zu untersuchen.

3.4.1 Karte 1: Langer Gang

Eine mögliche Schwierigkeit für SLAM Algorithmen sind lange Gänge, in denen es nur wenige Orientierungspunkte gibt. Für diesen Test wurde im Simulator ein 300m langer und 3,5m breiter Raum mit kleinen Verengungen alle 5m erstellt, die als Orientierungspunkte dienen sollen. Der Startpunkt war in der Mitte des Ganges. Die Route führte zuerst zum einen Ende und danach zum anderen Ende des Ganges. Probleme, die im Ergebnis auftreten können, sind z.B. Krümmungen und Verkürzungen/Verlängerungen der Karte.



Abbildung 3.5: Jackal Cartographer - langer Gang mit Standard-Konfiguration

In Abbildung 3.5 ist das Ergebnis des SLAM Algorithmus mit der Standard-Konfiguration der Jackal Cartographer Demo zu sehen. Es ist eine minimale Krümmung zu erkennen. Weiterhin sind die Wände an einigen Stellen sehr breit, doppelt oder gar nicht aufgezeichnet. Diese Fehler können durch zu hohe Unsicherheiten entstanden sein. Die Länge des Gangs entspricht ungefähr der originalen Länge. Nur zwei Verengungen wurden nicht erkannt.

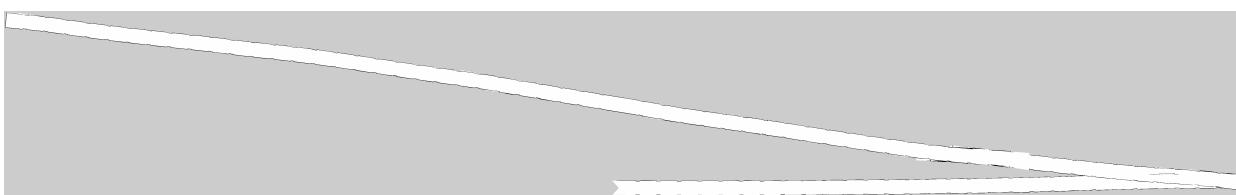
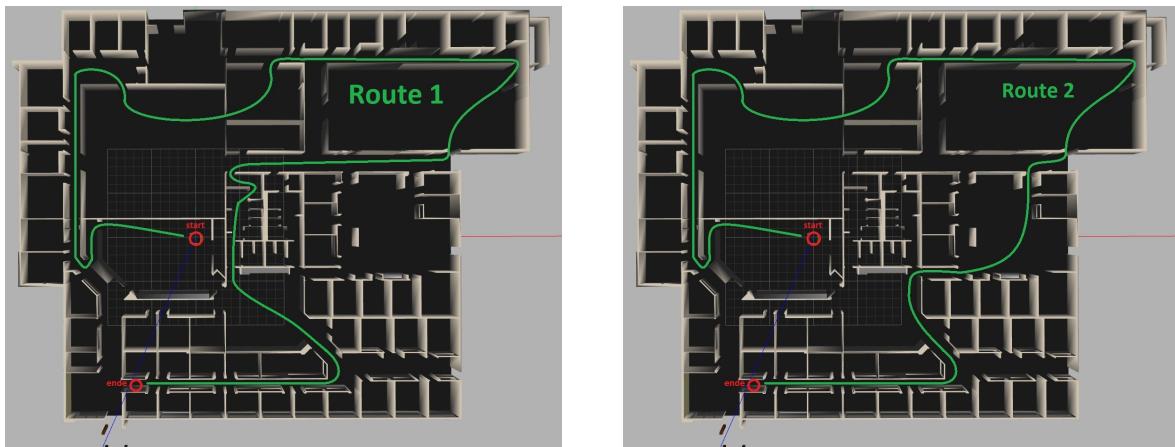


Abbildung 3.6: Jackal Cartographer - langer Gang mit deaktiviertem GlobalSLAM

In einem weiteren Test wurde GlobalSLAM abgeschaltet, indem `POSE_GRAPH.optimize_every_n_nodes` auf null gesetzt wurde. Dies dient der Demonstration, welches Ergebnis LocalSLAM liefert und wie GlobalSLAM dieses verbessert. Im Ergebnis in Abbildung 3.6 ist zu sehen, dass LocalSLAM die Scans zusammenfügt, aber bei jeder Rotation des Roboters Fehler entstehen, die nicht korrigiert werden. Der erste Abschnitt von der Mitte des Raumes nach rechts ist noch relativ gerade. Die leichten Krümmungen sind durch kleine Drehungen des Roboters entstanden. Am rechten Ende der Karte führte die 180° Rotation zu einem

größeren Fehler, der zu Artefakten und einer Krümmung der darauf folgenden Kartenteile führte. Diese Fehler sind entstanden, weil keine Kreisschlüsse durch GlobalSLAM gefunden wurden, die zu Korrekturen führen würden.

3.4.2 Karte 2: Willowgarage



(a) Jackal Cartographer - Willowgarage Route 1 mit ermöglichten Kreisschlüssen

(b) Jackal Cartographer - Willowgarage Route 2 mit vermiedenen Kreisschlüssen

Abbildung 3.7: Jackal Cartographer - Willowgarage mit zwei Routen

Ein weiterer Test wurde in einer realistischeren Umgebung ausgeführt. Die Karte „Willowgarage“ ist eine Demokarte, die bereits mit dem Gazebo Simulator installiert wird. Im Simulator wird ein büroähnlicher Aufbau erzeugt, durch den in zwei leicht voneinander abweichenden Routen navigiert wurde. Die Karte und beide Routen sind in Abbildung 3.7 dargestellt. Die erste Route in Abbildung 3.7(a) soll Kreisschlüsse ermöglichen während das Route 2 in Abbildung 3.7(b) verhindern soll. Auch bei diesem Test wurde die Standard-Konfiguration des Jackal Simulators verwendet.

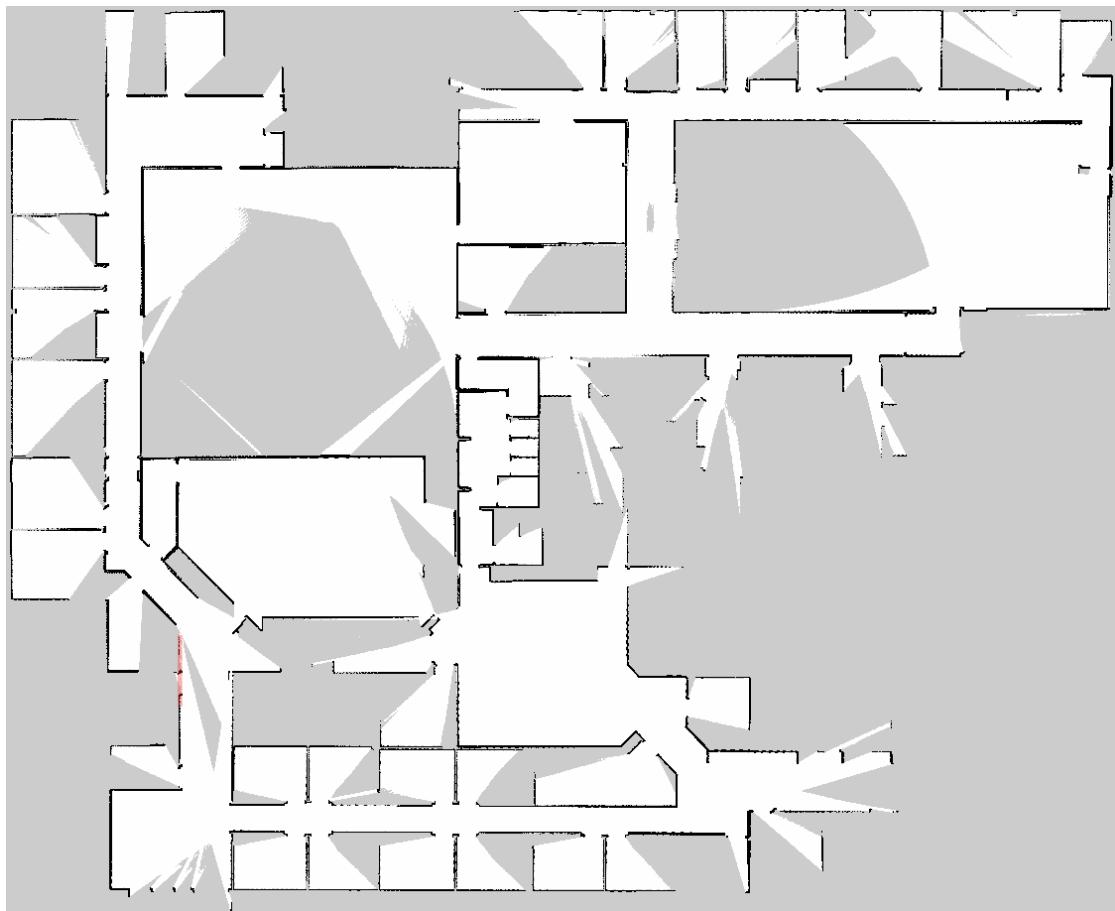


Abbildung 3.8: Ergebnis Jackal Cartographer - Willowgarage Route 1 mit ermöglichten Kreisschlüssen: Die rote Linie im unteren linken Bereich markiert den Übergang einer Wand in den Scans an Start und Ziel. Es sind keine Abweichungen erkennbar.

Abbildung 3.8 zeigt die vom Cartographer erzeugte Umgebungskarte der ersten Route aus Abbildung 3.7(a). Es sind keine Verzerrungen der durchfahrenen Räume zu erkennen. Die rote Linie ist an einer Position, an welcher die ersten und letzten Scans des Tests aufeinander treffen. Die Verbindung dieser Messungen funktioniert sehr genau, wenn im Laufe der Route Kreise geschlossen werden können.



Abbildung 3.9: Ergebnis Jackal Cartographer - Willowgarage Route 2 mit vermiedenen Kreisschlüssen: Die beiden roten Linien verlängern dieselbe Wand, die am Anfang und Ende der Route gescannt wurde. Es ist eine Abweichung erkennbar.

Mehr Fehler entstehen, wenn keine oder nur sehr wenige Kreise gefunden werden. Das zeigt das Ergebnis nach dem Test auf Route 2 (Abbildung 3.7(b)) in der Abbildung 3.9. Bereits im großen Raum oben rechts ist eine kleine Verschiebung zu erkennen, die nicht korrigiert werden konnte. Die beiden roten Linien verlängern dieselbe Wand, die am Anfang und Ende der Route aufgenommen wurde. Die Abweichung, der mit der roten Linie verlängerten Wand, beträgt ungefähr einen halben Meter.

Kapitel 4

Anbindung des Google Cartographers in Java

Um den Google Cartographer in Java umzusetzen, ist es notwendig die Abhängigkeiten des Cartographers durch Java Bibliotheken zu ersetzen.

4.1 Bundle Adjustment: Ceres-Solver und BoofCV

Eine der am meisten genutzten und damit wichtigsten Abhängigkeiten ist der Ceres Solver. Ceres ist eine C++ Bibliothek, die nicht-lineare Optimierungsprobleme und auch Bundle Adjustment lösen kann [SAMEER AGARWAL, MIERLE, 2010]. Sie ist unter Linux, Android, Mac OS X, iOS und Windows verfügbar. Die Windows Version besitzt jedoch weniger Features und wurde weniger getestet.

Der Google Cartographer nutzt den Ceres Scan Matcher, welcher eine Erweiterung des Ceres Solvers ist. Dies wird für das Matchen der Scans im LocalSLAM benötigt. Weiterhin wird der Ceres Solver im GlobalSLAM für das Finden von Kreisschlüssen genutzt.

BoofCV ist eine native Java Umsetzung von OpenCV. Darin wird der Levenberg-Marquardt (LM) Algorithmus implementiert, der als Alternative zum Ceres Solver¹ dienen könnte. Der Vorteil ist, dass durch Java sehr viele Betriebssysteme unterstützt werden. Die BoofCV Bibliothek befindet sich zwar noch in einer frühen Entwicklungsphase, soll aber trotzdem, für moderate Probleme, bereits eine ähnliche Performance wie Ceres besitzen. Zum Vergleich wurden die Daten aus [SAMEER AGARWAL, SNAVELY, 2010] verwendet. Es wird der Bundle Adjustment LM Algorithmus verwendet, der auch in Ceres genutzt werden kann. Im Beispiel für einen Bündelblockausgleich auf der Webseite von BoofCV kann eine dreidimensionale Szene aus bereits extrahierten Features erstellt werden. Das Ergebnis ist in Abbildung 4.1 zu sehen.

¹ BoofCV Homepage: <https://boofcv.org/>

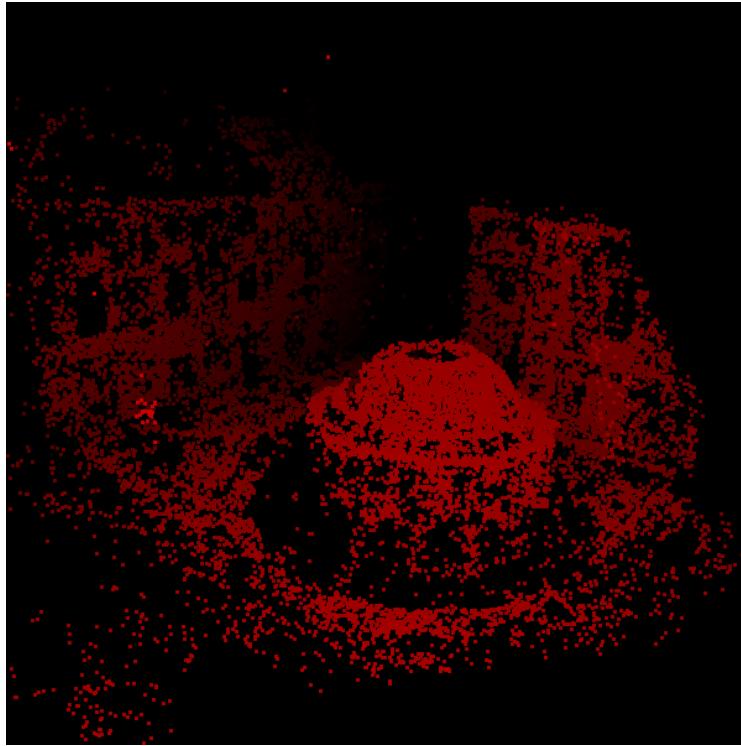


Abbildung 4.1: Dreidimensionale Szene, die durch LM mit BoofCV erstellt wurde.

Ein Unterschied zwischen Ceres Scan Matcher und BoofCV ist die Extraktion der Features. Im Beispiel auf der BoofCV Webseite werden die Datensets von [SAMEER AGARWAL, SNAVELY, 2010] genutzt, in denen bereits Features aus Bildern extrahiert und in einer Datei abgelegt wurden. Aus dem vom Roboter aufgenommenen Laserscan muss dieser Schritt jedoch vorher stattfinden, der im Ceres Scan Matcher bereits implementiert ist. Somit muss für eine Verwendung von BoofCV die Datengrundlage angepasst oder eine Feature Extraction implementiert werden. Zudem ist nicht sicher, ob BoofCV für die im Cartographer anfallenden Probleme eine ähnliche Performance liefert und, falls dem nicht so ist, Simultaneous Localization And Mapping (SLAM) in Echtzeit möglich ist. Durch den hohen Implementierungsaufwand und die Ungewissheit über die Leistungsfähigkeit der Bibliothek war es zeitlich nicht möglich diese Variante in Java umzusetzen.

4.2 Java Native Interface

Ein weiterer Ansatz zur Umsetzung war die Verwendung des C++ Cartographer Codes. Java Native Interface (JNI) bietet die Möglichkeit eine C oder C++ Bibliothek in Java einzubinden. Ein Nachteil ist der Verlust der Plattformunabhängigkeit von Java, da alle C/C++ Bibliotheken für alle Plattformen verfügbar sein müssen. Hinzu kommt ein Problem, wenn ein System veraltete oder inkompatible Versionen von Abhängigkeiten der C/C++ Bibliothek installiert hat. Der Vorteil ist, dass der bereits erstellte und getestete Cartographer Code verwendet werden kann und ein Performanceverlust nur durch JNI entstehen könnte.

4.2.1 ROS-Schnittstellenanalyse

Mit Hilfe des Jackal Simulators wurden die genutzten Schnittstellen zwischen Robot Operating System (ROS) und dem Cartographer analysiert. In Abbildung A.1 sind die Schnittstellen grafisch dargestellt. Die `main` Methoden der beiden genutzten ROS Knoten `cartographer_node` und `occupancy_grid_node` sind blau markiert. Der Knothen `cartographer_node` ist für die Übergabe der Sensordaten zuständig. Es werden Handler für alle Sensoren erstellt, die über Callback Methoden aufgerufen werden, sobald eine ROS-Message mit neuen Sensordaten verfügbar ist. Diese werden entpackt und in Cartographer Objekte `TimedPointCloudData` und `OdometryData` eingefügt. Alle Zeiten werden in Nanosekunden und alle Positionsdaten in Metern angegeben.

`TimedPointCloudData` enthält die Aufnahmezeit, einen Ursprung des Scans und die gemessenen Laserdaten. Die Zeit dient der Synchronisierung aller Sensordaten. Der Ursprung beschreibt durch x -, y - und z -Koordinaten die relative Position des Laserscanners zum Roboter und ist bei einem fest verbauten Scanner immer gleich. Die gemessenen Laserdaten sind relativ zum Sensor in x, y, z und Zeit angegeben. Die Zeit eines Laserdatums ist relativ zum vorherigen Datum, falls diese zeitversetzt aufgenommen wurden. Jackal führt alle Scans gleichzeitig aus, wodurch dieser Wert immer null beträgt.

`OdometryData` enthält die Aufnahmezeit, eine Translation und eine Rotation. Odometriedaten geben die Veränderung der Position zum Startpunkt an. Die Translation wird in x -, y - und z -Koordinaten beschrieben. Die Rotation gibt die Drehung in Quaternionen w, x, y und z an.

Über die angelegten Handler werden die Objekte mit einer Sensor ID an den Cartographer SLAM Algorithmus übergeben. Dieser verarbeitet die Daten und erstellt die Submaps. Dieser Schritt ist in Abbildung A.1 grün markiert.

Der Knothen `occupancy_grid_node` erstellt einen Handler, dessen Callback Methode das Ergebnis des Cartographers übernimmt. In diesem sind Submap IDs und deren Texturen enthalten. Beides muss kombiniert werden zu sogenannten „Submap Slices“. Diese sind Teilkarten und enthalten mehrere aufeinander folgende Scans. Um eine komplette Umgebungskarte zu erhalten, müssen die Slices an den Cartographer übergeben werden. Die Teilkarten werden in einem Cairo-Image kombiniert und zurückgegeben. Das Bild besitzt eine Breite, Höhe und Pixeldaten. In den Pixeldaten ist gespeichert, ob die jeweilige Position der Karte besucht wurde und wie hoch die Belegtheitswahrscheinlichkeit ist. Ein Timer sorgt dafür, dass diese Karte regelmäßig an registrierte Handler in anderen Knoten publiziert wird.

4.2.2 Vorbereitung und Planung

Die in Kapitel 4.2.1 beschriebenen Schnittstellen sollen durch andere, ROS-unabhängige Schnittstellen, ersetzt werden. Um die geplante Schnittstelle testen zu können, wurden zuerst Sensordaten des Light Detection And Ranging (LIDAR) Scanners und Odometriedaten mit Jackal (Kapitel 3.3) aufgenommen und in Dateien geschrieben. Diese können zur Verifikation genutzt werden. Da Jackal nur mit ROS funktioniert, muss zunächst der Cartographer ohne ROS Anbindung installiert werden. Eine Anleitung dafür gibt es auf der Cartographer Dokumentationsseite.²

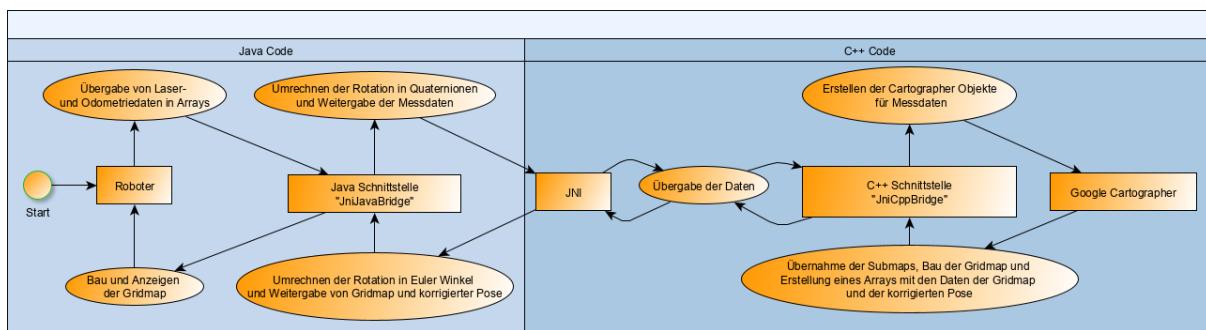


Abbildung 4.2: Architektur der entstandenen Schnittstelle zwischen Cartographer und Java mit JNI (vergrößert in Abbildung A.2 auf Seite 55)

Die Umsetzung der im Rahmen dieser Arbeit erstellten Schnittstelle (Abbildung 4.2) erfolgt in drei Schritten:

1. Erstellen einer C++ Schnittstelle zum Cartographer
2. Erstellen einer Java Schnittstelle zum Roboter
3. Erstellen des JNI zur Verbindung der C++ und Java Schnittstelle.

Die C++ und Java Schnittstellen können mit den vom Jackal aufgezeichneten Daten getestet werden. Alle Cartographer Konfigurationen, die bei der Aufzeichnung verwendet wurden, können auch zum Testen genutzt werden. Die Belegtheitswahrscheinlichkeiten der entstandenen Umgebungskarte können mit deren Position testweise in eine Datei geschrieben werden. Die Karte kann dann z.B. geplottet werden, um sie zu visualisieren. Das Ergebnis sollte immer ähnlich dem Ergebnis des Jackal Simulators sein.

² Cartographer Dokumentation: <https://google-cartographer.readthedocs.io/en/latest/>

4.2.3 Erstellen einer C++ Schnittstelle

Im ersten Schritt soll eine C++ Schnittstelle für den Cartographer erstellt werden (Abbildung 4.2 rechte Seite). Die „JniCppBridge“ genannte Klasse dient der Steuerung des Cartographers und soll zunächst nur grundlegende Funktionen bereitstellen, um eine Umgebungskarte erstellen zu können.

```

1 void JniCppBridge::HandleLaserScan(const std::string &sensor_id, const long
2   ↪ timeValue, std::vector<float> scanData) {
3 // Zeit in Cartographer Object umwandeln
4 cartographer::common::Time time = cartographer::common::FromUniversal(timeValue);
5   ↪
6 cartographer::sensor::TimedPointCloudData timedPointCloud =
7   ↪ cartographer::sensor::TimedPointCloudData();
8 timedPointCloud.time = time;
9 // Ursprung: x, y, z
10 timedPointCloud.origin = Eigen::Vector3f(scanData.at(0), scanData.at(1),
11   ↪ scanData.at(2));
12 // iterieren über Laserdaten: x,y,z
13 for (int scanIndex = 3; scanIndex + 3 < scanData.size(); scanIndex += 4) {
14   timedPointCloud.ranges.push_back(Eigen::Vector4f(
15     scanData.at(scanIndex), scanData.at(scanIndex + 1),
16     scanData.at(scanIndex + 2), scanData.at(scanIndex + 3)));
17 }
18 trajectory_builder_->AddSensorData(sensor_id, timedPointCloud);
19 }
```

Listing 4.1: C++ Schnittstelle - Übergabe des Laserscans

In Listing 4.1 ist beispielhaft der Code zur Übergabe der Laserdaten an den Cartographer zu sehen. Diese Methode wird im Kapitel 4.2.5 von JNI aufgerufen. Aufgabe der C++ Schnittstelle ist sowohl die Erstellung der Cartographer Objekte aus den in Java erzeugten Daten als auch die Übernahme der vom Cartographer erzeugten Daten. Die in Listing 4.1 dargestellte Methode übernimmt den Laserscan. Falls nur ein Laserscanner im Cartographer registriert wurde, ist die Sensor ID immer gleich. Der Parameter `timeValue` enthält die Aufnahmezeit der Daten. In `scanData` ist der Ursprung des Laserscans relativ zum Roboter in den ersten drei Elementen des Arrays gespeichert. Danach folgen alle Messdaten mit relativen Koordinaten zum Ursprung (x,y,z) und der vergangenen Zeit zum vorherigen Messpunkt. Am Schluss erfolgt die Übergabe der erzeugten Cartographer Objekte.

Die Übergabe der Odometriedaten erfolgt ähnlich. Ein Unterschied ist, dass ein Winkel für die aktuelle Position übergeben wird. Der Cartographer arbeitet mit Quaternionen. Um auch Euler-Winkel nutzen zu können, wurde zusätzlich eine Methode implementiert, die eine Umrechnung übernimmt.

Das Erstellen der Grid Map ist umfangreicher als die Übergabe der Messdaten. Zuerst müssen alle erzeugten Submap IDs und deren Texturen abgefragt werden. Im Anschluss kann der Cartographer diese zu einer gesamten Karte zusammenbauen. Um den Umfang der übergebenen Daten im Laufe der Zeit zu reduzieren, wäre es sinnvoll, nur den veränderten Bereich der Karte zu übergeben. Dazu können nur die neuen und durch GlobalSLAM angepassten Submaps abgerufen und zu einer Karte gebaut werden. Die Größe und Position kann dann zusammen mit der neuen Teilkarte an Java übergeben und dort die gesamte Karte angepasst werden. Aufgrund der verfügbaren Zeit für diese Arbeit war es nicht möglich dieses Feature zu implementieren.

4.2.4 Erstellen der Java Schnittstelle

Mit den vorhandenen C++ Schnittstellen kann eine Java Schnittstelle geschrieben werden, die mit JNI kommuniziert. Die Klasse wurde „JniJavaBridge“ genannt und ist in Abbildung 4.2 links dargestellt. Zur Verbindung von JNI und Java werden alle benötigten Methoden in einer Java Klasse mit dem Schlüsselwort `native` definiert.

Die benötigten Funktionen sind:

- Erstellen notwendiger Cartographer Objekte: Konfiguration, Map Builder, Sensoren registrieren, Trajektorie erzeugen
- Übergeben der LIDAR- und Odometrie-Sensordaten von C++ Schnittstelle zum Cartographer
- Übernehmen der entstandenen Umgebungskarte
- Finalisieren der Karte und der Trajektorie

Listing 4.2 zeigt beispielhaft die Definition der Methode zur Übergabe des Laserscans. Diese ähnelt der Definition einer abstrakten Methode.

```
1 public native void sendLaserScanData(String sensorId, long time, float[] data);
```

Listing 4.2: Java Schnittstelle - Übergabe des Laserscans

Sind die nativen Methoden definiert, kann mit `javadoc` in Java Versionen kleiner oder gleich 1.8 bzw. `javac -h` eine C++ Header Datei generiert werden. Diese enthält die Methodenköpfe, die JNI erwartet.

4.2.5 Erstellen der JNI

Mit Hilfe der im vorigen Kapitel erstellten Header Datei können jetzt die JNI Methoden in C++ implementiert werden. In Abbildung 4.2 befindet sich diese Schnittstelle in der Mitte zwischen der „JniJavaBridge“ und der „JniCppBridge“. Um das JNI möglichst einfach zu halten, erfolgt die Kommunikation zwischen Java und C++ über die primitiven Objekte, die JNI unterstützt. In Tabelle 4.1 sind die verwendeten Objekte und die Umwandlung der Typen von C++ in Java/JNI Objekte zusammengefasst.

Tabelle 4.1: Verwendete Dateitypen zur Übergabe von Daten mit JNI

Übergabetyp	C++ Typ → JNI-Typ
Konfigurationsdateipfad	<code>std::string</code> → <code>jstring</code>
Sensor ID/Typ	<code>std::string</code> → <code>jstring</code>
Zeit	<code>long</code> → <code>jlong</code>
Messwerte	<code>std::vector<float></code> → <code>jfloatArray</code>
Grid Map	<code>std::vector<int></code> → <code>jintArray</code>
korrigierte Pose	<code>std::vector<float></code> → <code>jfloatArray</code>

Beispielhaft zeigt Listing 4.3 wie die Umwandlung der Objekte funktioniert. Strings können vom `JNIEnv` geholt werden. Für primitive Typen funktioniert meist ein Type-Casting. Arrays sind etwas komplizierter zu übergeben. Zunächst muss die Größe des Arrays aus dem `JNIEnv` abgefragt werden. Danach kann über die Elemente des Arrays iteriert und diese in einen neu erstellten Vektor eingefügt werden. Am Schluss wird die gewünschte Methode der C++ Schnittstelle aufgerufen, um die Daten dort an den Cartographer zu übergeben.

```

1 JNIEXPORT void JNICALL
2     ↪ Java_cartographer_jni_CartographerJniJavaBridge_sendLaserScanData
3 (JNIEnv *env, jobject, jstring j_sensor_id, jlong j_time, jfloatArray j_data) {
4     // konvertiere die Sensor ID
5     std::string sensor_id = env->GetStringUTFChars(j_sensor_id, NULL);
6     // Konvertiere die Zeit
7     long time = (jlong) j_time;
8
9     // Konvertiere das float Array in einen float Vektor
10    jfloat *data_array = (*env).GetFloatArrayElements(j_data, NULL);
11    if (NULL == data_array)
12        return;
13    jsize length = env->GetArrayLength(j_data);
14
15    std::vector<float> data_vector;
16    for (int i = 0; i < length; i++) {

```

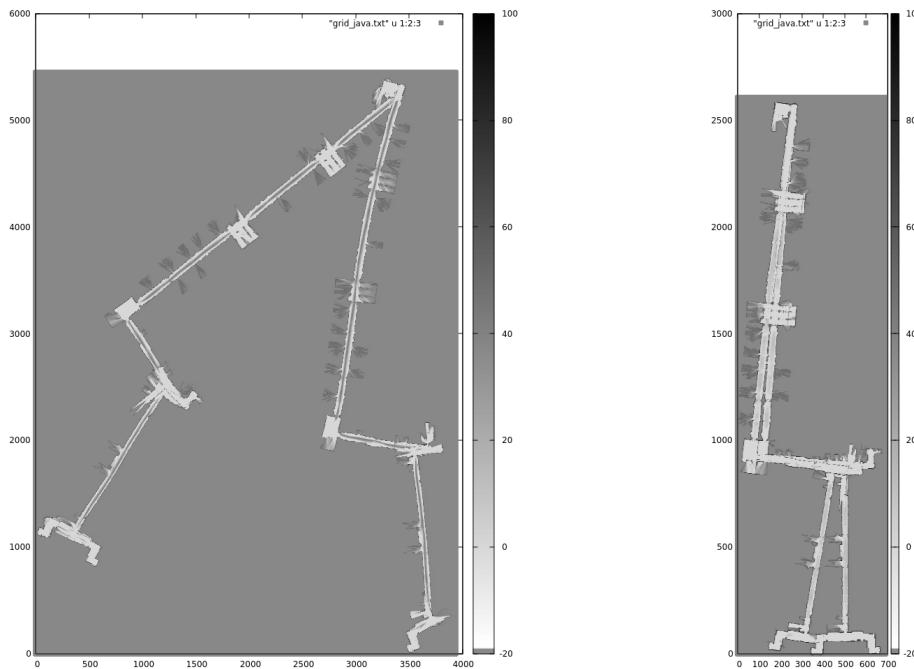
```
16     data_vector.push_back(data_array[i]);  
17 }  
18 // Aufrufen der C++ Schnittstelle  
19 jniCppBridge.HandleLaserScan(sensor_id, time, data_vector);  
20 }
```

Listing 4.3: JNI - Konvertierung des Laserscans

Kapitel 5

Ergebnisse und Optimierung

Zum Test des Cartographer Algorithmus wurden im S- und Z-Gebäude der HTW Dresden Sensordaten aufgenommen. Im ersten Testlauf wurde die Konfiguration des Jackal Cartographers genutzt, da diese im Gazebo Simulator gute Ergebnisse lieferte.



(a) Umgebungskarte mit Jackal Cartographer Konfiguration

(b) Umgebungskarte mit angepasster Jackal Cartographer Konfiguration

Abbildung 5.1: Umgebungskarten mit nicht optimierter Konfiguration

Abbildung 5.1(a) zeigt das Ergebnis dieser Konfiguration. Es sind starke Krümmungen der Gänge und verschobene Räume zu erkennen. Scheinbar konnten auch keine Kreisschlüsse gefunden werden, um die Verschiebung auf dem Rückweg zu korrigieren.

Aufgrund des schlechten Ergebnisses, wurden in einem weiteren Lauf verschiedene Parameter der Jackal Konfiguration entfernt, um die Standardwerte des Cartographers zu nutzen. Abbildung 5.1(b) zeigt das Ergebnis. Fehler aus dem ersten Versuch konnten deutlich reduziert werden. Dennoch war scheinbar kein Kreisschluss möglich, da die Umgebung auf dem Rückweg nicht erkannt wurde und damit ein neuer Gang entstand. Nach weiteren Testläufen im Java-Simulator konnte die Ursache gefunden werden.

Folgende Parameter sorgten für die Fehler:

```

1 TRAJECTORY_BUILDER_2D.submaps.num_range_data = 1
2 POSE_GRAPH.global_sampling_ratio = 0.00001
3 POSE_GRAPH.constraint_builder.sampling_ratio = 0.0001

```

Die Nutzung der Standard Cartographer Werte für `sampling_ratio` und die Optimierung von `num_range_data` auf 30 führte zu einem deutlich besseren Ergebnis.

Die Aufnahmen des S- und Z-Gebäudes wurden nach der Optimierung der Konfigurationsparameter für einen Vergleich der Ergebnisse von Cartographer und GMapping verwendet. In Listing 5.1 sind die angepassten Parameter dargestellt. Mit -- wurden die Standardwerte auskommentiert und durch die davor stehenden Werte überschrieben.

```

1 MAP_BUILDER.use_trajectory_builder_2d = true -- false
2 TRAJECTORY_BUILDER_2D.use_imu_data = false -- true
3 TRAJECTORY_BUILDER_2D.min_range = 0.05 -- 0.
4 TRAJECTORY_BUILDER_2D.max_range = 10 -- 30.
5 TRAJECTORY_BUILDER_2D.use_online_correlative_scan_matching = true -- false
6 TRAJECTORY_BUILDER_2D.submaps.num_range_data = 30 -- 90
7 TRAJECTORY_BUILDER_2D.submaps.grid_options_2d.resolution = 0.1 --0.05

```

Listing 5.1: optimierte Cartographer Konfiguration

Die im Folgenden dargestellten Karten des Cartographers wurden mit „August dem Smarten“, einer Scitos G5 Plattform¹, aufgenommen.

¹ MetraLabs GmbH: <https://www.metralabs.com/mobiler-roboter-scitos-g5/>

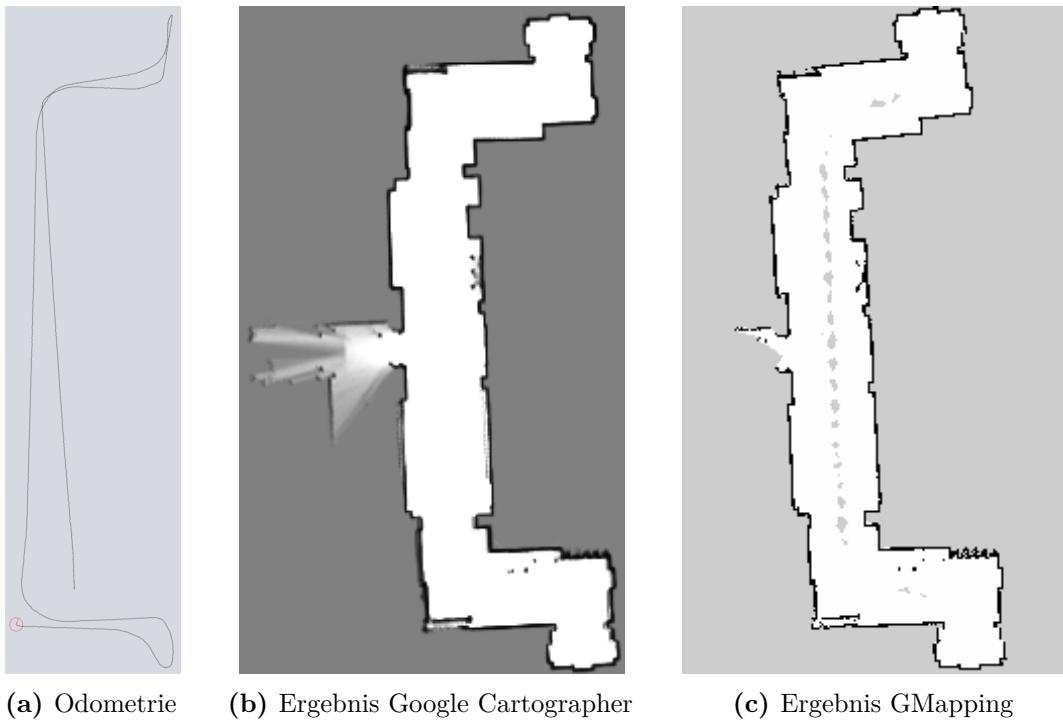


Abbildung 5.2: HTW Dresden Bereich Fakultät Informatik: Odometrie und Ergebnisse vom Google Cartographer und GMapping, Länge der gefahrenen Distanz: 77m

Die drei Teilabbildungen unter Abbildung 5.2 zeigen Aufnahmen des Bereiches der Fakultät Informatik an der HTW Dresden. Abbildung 5.2(a) bildet den nicht optimierten Odometriepfad ab. Der rote Kreis im unteren linken Bereich war die Endpose dieser Aufnahme. Das Ergebnis des Google Cartographers mit der erstellten Schnittstelle ist in Abbildung 5.2(b) dargestellt. Im Vergleich dazu ist in Abbildung 5.2(c) das Ergebnis mit dem bisher an der HTW Dresden verwendeten GMapping Algorithmus zu sehen. Die Wände im Cartographer sind breiter als im GMapping und etwas verschwommen. In der Mitte des Gangs sind in Abbildung 5.2(c) graue Bereiche zu erkennen, die im Nachgang korrigiert werden müssen. Außerdem ist in der oberen Hälfte eine Verzerrung nach links sichtbar, da sich das obere und untere Ende des Raumes ungefähr auf derselben Höhe befinden sollte. Diese beiden Auffälligkeiten sind im Ergebnis des Cartographers nicht erkennbar. Solche Verzerrungen können in größeren Umgebungen zu Fehlern führen, die nicht korrigiert werden. Aus diesem Grund wurde der Testlauf auf die gesamte dritte Etage des S- und Z-Gebäudes der HTW Dresden erweitert.

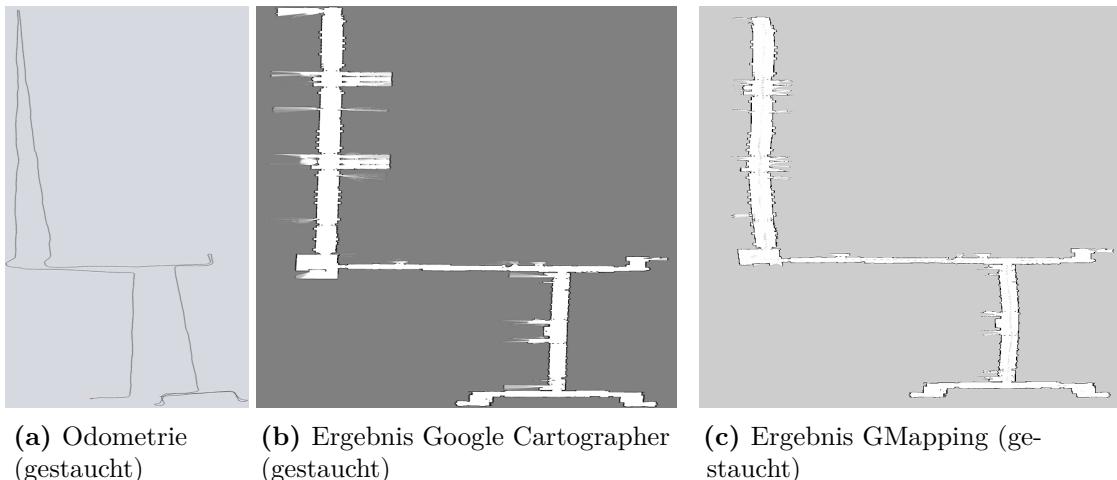


Abbildung 5.3: HTW Dresden 3. Etage des S- und Z-Gebäudes: Dargestellt sind Odometrie und Ergebnisse vom Google Cartographer und GMmapping. Die gefahrene Distanz beträgt 705,8m. Um die Unterschiede besser erkennen zu können, wurden die Abbildungen gestaucht.

Die Ergebnisse sind in Abbildung 5.3 zu sehen. Um mögliche Fehler besser erkennen zu können, wurden die Abbildungen vertikal gestaucht. Die Bilder mit korrektem Seitenverhältnis befinden sich im Anhang auf Seite 56, Abbildung A.3. Unten rechts ist noch einmal die Umgebung aus Abbildung 5.2 zu sehen. Dort befand sich auch der Startpunkt. Zu Beginn wurde wieder der Bereich der Fakultät Informatik aufgenommen. Im Anschluss wurde der Gang durch das Z- und S-Gebäude gescannt und danach wieder zum Startpunkt zurückgekehrt. Ein Unterschied ist im oberen Teil zu erkennen. Dort wurden drei Räume an den Treppen vom Cartographer komplett erfasst. GMmapping konnte nur Fragmente der Scans nutzen und die Karte an den drei Stellen nicht vollständig abbilden. Weiterhin sind einige Verzerrungen der Gänge oben links und unten rechts im GMmapping Ergebnis zu sehen. Eine Verzerrung wie im ersten Testlauf scheint nicht stattgefunden zu haben, da die Umgebungskarte im Großen und Ganzen nur wenige Abweichungen aufzeigt. In der nicht gestauchten Abbildung A.3(c) sind, wie auch im ersten Testlauf, graue Bereiche in der Mitte der Gänge sowie doppelte Wände und Fragmente im linken Gang zu erkennen.

Im Ergebnis des Cartographers sind weniger Fehler zu sehen. Eine Verzerrung im unteren rechten Gang führt zu einer leichten Verschiebung des oberen Abschnitts. Im Vergleich zu GMmapping sind die einzelnen Gänge aber deutlich gerader. Eine weitere Auffälligkeit ist die unterschiedliche Höhe der Karten bei gleicher Auflösung. Das bedeutet entweder, dass der Gang vom Cartographer verlängert oder vom GMmapping verkürzt wurde. Eine genaue Messung war leider nicht möglich. Die Messung über Google Maps ergab eine Gesamtlänge des S- und Z-Gebäudes von ca. 255m. Der Mittelgang zwischen beiden Gebäuden ist 57m lang. Die Grid Map des Cartographers ist 2560 Zellen hoch. Der Gang in der Mitte der Abbildung A.3(b) ist 575 Zellen breit. Bei einer Auflösung des Grids von 10cm sind die Abweichungen sehr gering. Im Ergebnis des GMmapping Algorithmus ist die mit 2440 Zellen

hohe Karte über 10m zu kurz. Der 575 Zellen lange mittlere Gang ist genau gleich lang wie im Cartographer Ergebnis.

Allgemein kann man sagen, dass GMapping ein gutes Ergebnis liefern konnte. Der Cartographer konnte jedoch mehr Informationen aus den Messungen nutzen und damit ein genauereres Ergebnis erzielen.

Kapitel 6

Fazit und Ausblick

Diese Arbeit zeigt, dass Simultaneous Localization And Mapping (SLAM) Algorithmen ein wichtiger Bestandteil autonomer Roboter sind. Der Google Cartographer liefert mit einer auf die Hardware angepassten Konfiguration genaue Umgebungskarten, mit denen eine zielgerichtete Navigation möglich ist. Trotz der nicht gelungenen nativen Java-Implementierung ermöglichte es Java Native Interface (JNI) den Cartographer dennoch mit dieser Programmiersprache zu nutzen. Die innerhalb dieser Arbeit entstandene Schnittstelle bietet grundlegende Methoden, um eine Umgebungskarte zu erstellen und die korrigierte Pose des Roboters vom Cartographer zu erhalten. Damit kann der Cartographer so genutzt werden, wie er in C++ implementiert wurde. Da keine Bibliotheken ausgetauscht wurden, sind auch keine Performanceverluste zu befürchten. Für eine Aktualisierung auf eine neue Version muss vermutlich die Schnittstelle angepasst werden, da es seit Release 1.0 viele Änderungen gab. Leider ist durch die Verwendung des C++ Codes die Plattformunabhängigkeit verloren gegangen, die Java ermöglicht. Das wird vor allem bei einem zukünftigen Update einer Abhängigkeit des Cartographers deutlich. In diesem Fall muss die Schnittstelle mit CMake erneut gebaut oder der JNI Code angepasst werden. Die innerhalb der Arbeit entstandenen Ergebnisse werden in Echtzeit erzeugt und zeigen die Genauigkeit der entstandenen Karten. Mit der Anpassung weiterer Konfigurationsparameter ist es wahrscheinlich noch möglich die in Kapitel 5 gezeigten Abweichungen weiter zu minimieren. Zudem könnte es in Zukunft auch sinnvoll sein, dreidimensionale Karten mit dem Cartographer zu bauen. Dadurch wäre es z.B. möglich, Karten über mehrere Etagen aufzuzeichnen oder Hindernisse erkennen und unterscheiden zu können. Der aktuelle Laserscanner müsste dafür ausgetauscht oder umgebaut werden, damit dreidimensionale Laserscans erstellt werden können. Eine zusätzliche Einbindung des Laserscanners an der Rückseite des Roboters sowie die Verwendung von GPS Daten oder Orientierungspunkten können die Genauigkeit der Ergebnisse weiter steigern. Die zwei- und dreidimensionalen Karten können wahrscheinlich auch in anderen Forschungsprojekten des Fachbereichs eingesetzt werden.

Literatur

- AGARWAL, S., N. SNAVELY, I. SIMON, S. M. SEITZ und R. SZELISKI (Sep. 2009): „Building Rome in a day“. *2009 IEEE 12th International Conference on Computer Vision*: S. 72–79 (siehe S. 14).
- AGARWAL, SAMEER, KEIR MIERLE (2010): *Ceres Solver*. <http://ceres-solver.org> (siehe S. 35).
- AGARWAL, SAMEER, NOAH SNAVELY, STEVEN M. SEITZ, RICHARD SZELISKI und GOOGLE INC (2010): *Bundle Adjustment in the Large* (siehe S. 14, 35, 36).
- BESL, P. J. und N. D. MCKAY (Feb. 1992): „A method for registration of 3-D shapes“. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Bd. 14(2): S. 239–256 (siehe S. 12).
- BÖHME, HANS-JOACHIM, FRANK BAHRMANN, CHRISTIAN BISCHOFF, ROBERT ERZGRÄBER, ANDREAS HERMANN, STEFFI FRIMMEL, ELMAR GRÄSSEL, CATHARINA WASIC und FRANK WEBER (2018): „CARE4ALL-Initial - A new Human-Technology Interaction Concept for Care of People with Dementia“. (Siehe S. 2).
- BORENSTEIN, JOHANN (1998): „Experimental Results from Internal Odometry Error Correction with the OmniMate Mobile Robot“. *IEEE TRANSACTIONS ON ROBOTICS AND AUTOMATION*, Bd. 14: S. 963–969 (siehe S. 9).
- BÖRLIN, NICLAS und PIERRE GRUSSENMEYER (1. Dez. 2013): *Bundle Adjustment With and Without Damping* (siehe S. 12–14).
- CANELHAS, DANIEL RICÃO (2017): „Truncated Signed Distance Fields Applied To Robotics“. Diss. Örebro University, School of Science und Technology: S. 161 (siehe S. 25).
- ENGELS, CHRIS, HENRIK STEWÉNIUS und DAVID NISTÉR (2006): „Bundle adjustment rules“. In *Photogrammetric Computer Vision* (siehe S. 14).
- HESS, WOLFGANG, DAMON KOHLER, HOLGER RAPP und DANIEL ANDOR (2016): „Real-Time Loop Closure in 2D LIDAR SLAM“. *2016 IEEE International Conference on Robotics and Automation (ICRA)*: S. 1271–1278 (siehe S. 2, 13, 24, 25, 27, 28).
- INTERBARTOLO, MICHAEL (Jan. 2009): *Apollo Guidance, Navigation, and Control (GNC) Hardware Overview*. Techn. Ber. NASA Johnson Space Center (siehe S. 8).
- JUANG, JIH GAU und JIA AN WANG (Juni 2015): „Indoor Map Building by Laser Sensor and Positioning Algorithms“. *Modern Design Technologies and Experiment for Advanced Manufacture and Industry*. Bd. 764. Applied Mechanics and Materials. Trans Tech Publications: S. 752–756 (siehe S. 1).

- KONOLIGE, KURT, JOSEPH AUGENBRAUN, NICK DONALDSON, CHARLES FIEBIG und PANKAJ SHAH (2008): „A Low-Cost Laser Distance Sensor“. (Siehe S. 19).
- MONTEMERLO, MICHAEL und SEBASTIAN THRUN (2007): *FastSLAM: A Scalable Method for the Simultaneous Localization and Mapping Problem in Robotics*. Springer Publishing Company, Incorporated (siehe S. 1).
- MORAVEC, HANS P. und ALBERTO ELFES (1985): „High resolution maps from wide angle sonar“. *Proceedings. 1985 IEEE International Conference on Robotics and Automation*, Bd. 2: S. 116–121 (siehe S. 10).
- OLSON, EDWIN (Juni 2009): „Real-Time Correlative Scan Matching“. *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. Kobe, Japan: IEEE: S. 4387–4393 (siehe S. 24).
- SCHWARZ, BRENT (Juli 2010): „Mapping the world in 3D“. *Nature Photonics*, Bd. 4 (siehe S. 5, 6).
- SNAVELY, NOAH, STEVEN M. SEITZ und RICHARD SZELISKI (2008): „Skeletal graphs for efficient structure from motion“. *Proc. Computer Vision and Pattern Recognition* (siehe S. 14).
- THRUN, SEBASTIAN (2006): *Probabilistic robotics*. eng. [Nachdruck]. Intelligent robotics and autonomous agents (siehe S. 15).
- WANG, PENG, ZONGHAI CHEN, QIBIN ZHANG und JIAN SUN (2016): „A loop closure improvement method of Gmapping for low cost and resolution laser scanner“. *IFAC-PapersOnLine*, Bd. 49(12). 8th IFAC Conference on Manufacturing Modelling, Management and Control MIM 2016: S. 168–173 (siehe S. 1).
- WOODMAN, OLIVER J. (Aug. 2007): *An introduction to inertial navigation*. Techn. Ber. UCAM-CL-TR-696. University of Cambridge, Computer Laboratory (siehe S. 8, 9).
- YANG, SHAO-WEN und CHIEH-CHIH WANG (1. Apr. 2011): *On Solving Mirror Reflection in LIDAR Sensing* (siehe S. 6, 7).
- ZHANG, XUETAO, LIBO JIAN und MEIFENG XU (Mai 2018): „Robust 3D point cloud registration based on bidirectional Maximum Correntropy Criterion“. *PLOS ONE*, Bd. 13: e0197542 (siehe S. 13).

A Anhang

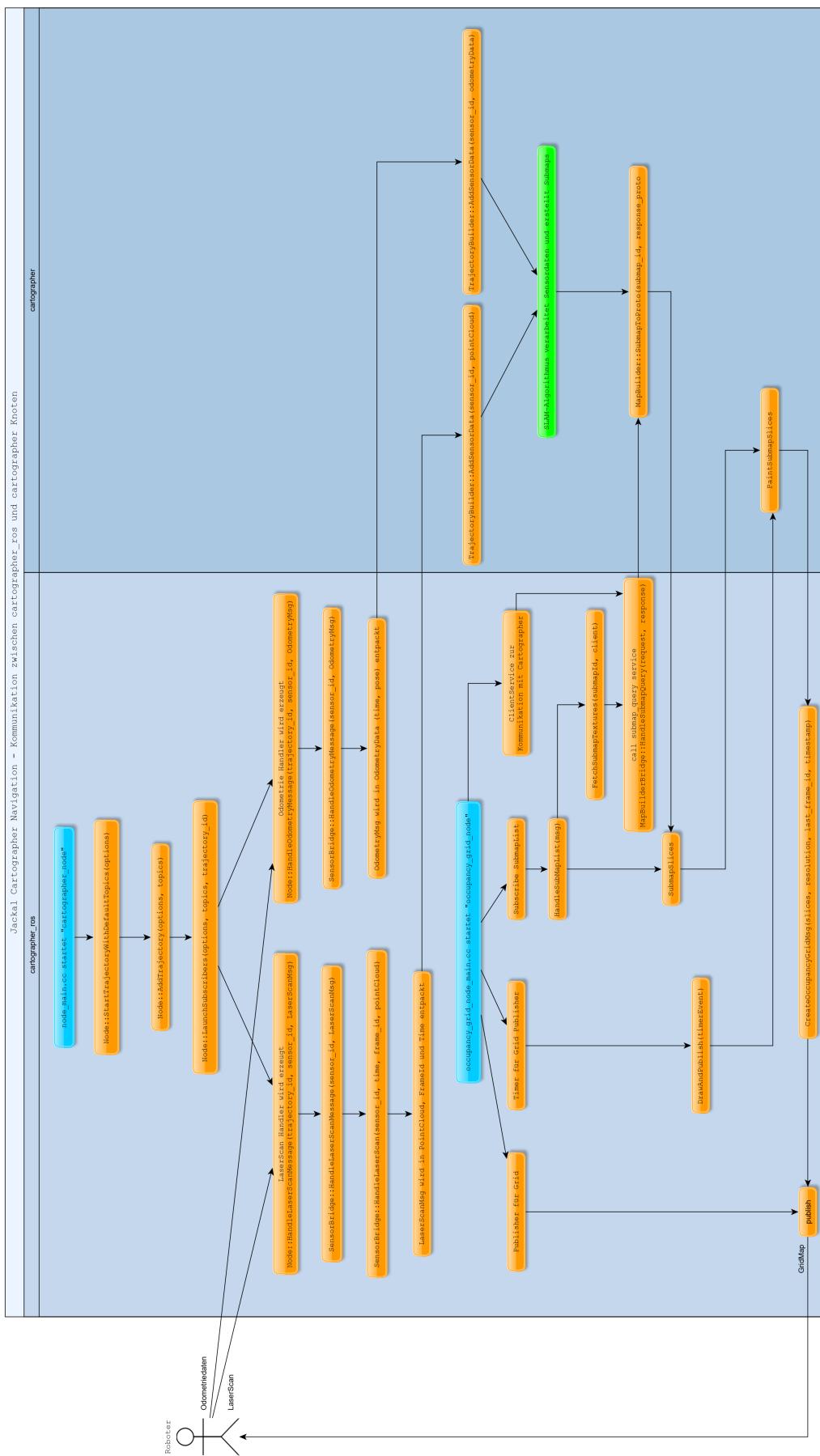


Abbildung A.1: Darstellung möglicher Cartographer Schnittstellen zur Analyse einer Verwendung in Java.

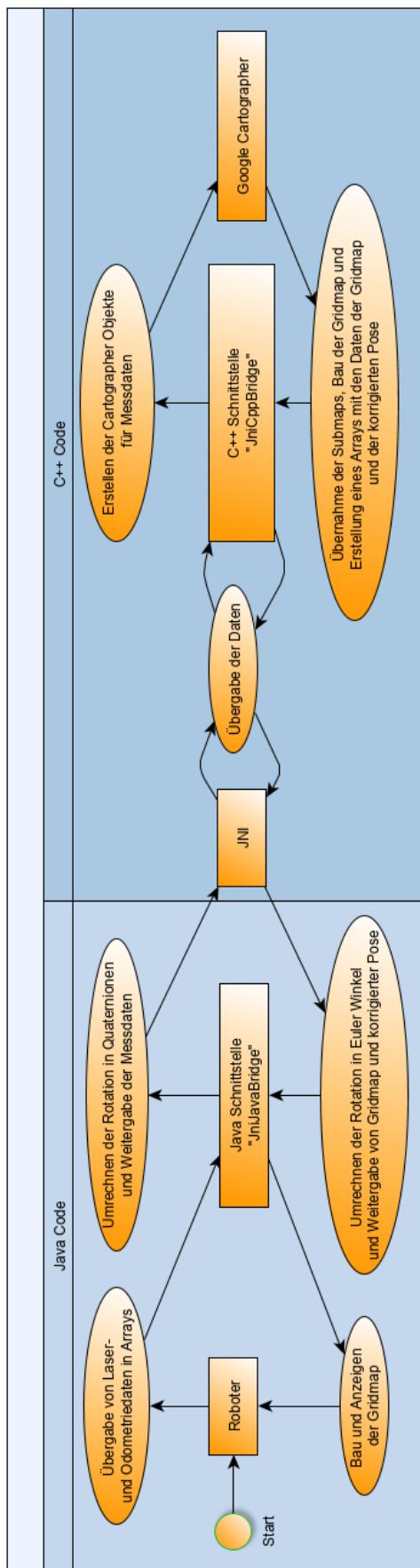


Abbildung A.2: Architektur der entstandenen Schnittstelle zwischen Cartographer und Java mit Java Native Interface (JNI) (vergrößert)

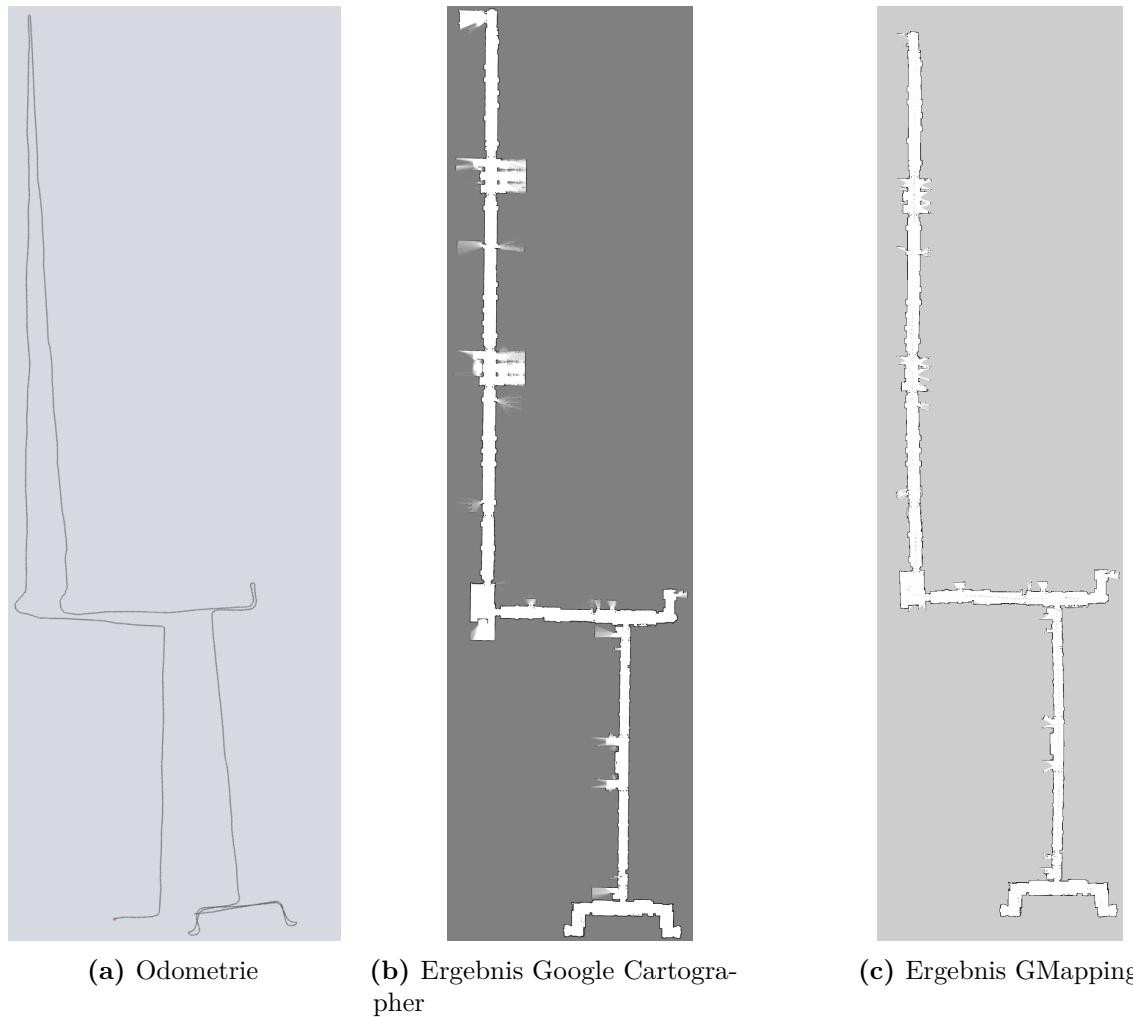


Abbildung A.3: HTW Dresden 3. Etage des S- und Z-Gebäudes: Dargestellt sind Odometrie und Ergebnisse vom Google Cartographer und GMapping (nicht gestaucht). Die gefahrene Distanz beträgt 705,8m.

Selbstständigkeitserklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Ort, Datum

Unterschrift

