Universität Augsburg

Fakultät für Angewandte Informatik

Bachelor Abschlussarbeit

Entwurf einer Hinderniserkennung und -vermeidung für die Mikromobilitätsplattform Scoomatic

---

vorgelegt von:	Henri Chilla
geboren am:	11. 02. 1997 in Öhringen
Studiengang:	Ingenieurinformatik
Anfertigung am Lehrstuhl:	Mechatronik
	Fakultät für Angewandte Informatik
Verantwortlicher Professor:	Prof. Dr. Ing. Lars Mikelsons
Wissenschaftlicher Betreuer:	M.Sc. Lennart Luttkus

Kurzfassung

Das Ziel der Arbeit ist die Realisierung einer Hinderniserkennung und -vermeidung für einen Elektrokleinstfahrzeug-Prototypen. Dabei wurde das Meta-Betriebssystem Robot Operating System (ROS) verwendet. Die notwendigen Treiber wurden programmiert und diverse Softwarepakete so konfiguriert, dass eine autonome Navigation möglich ist. Die Konfiguration wurde mit den ROS-spezifischen Tools vorgenommen. Außerdem wurden die ROS Analysetools für Analyse der Daten verwendet. Für die Treiber der Sensoren und Eingabegeräte, sowie sonstige Programme, welche selbst programmiert wurden, wurde Python 2.7 verwendet.
Das System wurde erfolgreich implementiert und kann autonom Navigieren. Dies funktioniert allerdings nicht immer fehlerfrei. Es bestehen weiterhin verschiedene Möglichkeiten Fehler zu diagnostizieren und zu analysieren. Außerdem besteht die Möglichkeit die Hardware-, als auch die Softwareplattform aufgrund der Modularität zu erweitern.
Es wurde gezeigt, dass die Kartenerstellung, neben verschiedenen Problemen, so gut funktioniert, dass damit eine Navigation möglich ist. Die Navigation hat jedoch Probleme damit jegliche Art von Hindernissen zu erkennen. Die Sensoren reichen nicht für eine allumfassende Analyse der Umgebung aus.
Deswegen ist ein sicheres Betreiben, das heißt ohne menschliche Kontrolle und Eingreifmöglichkeiten, noch nicht möglich. Schlussendlich wurden Ideen vorgestellt, um diese Probleme zu beheben.

Abstract

The goal of this work was the implementation of a system for obstacle recognition and avoidance. As hardware platform, a prototype of an electric motorized scooter was used. For software deployment, the meta-operating system ROS was used. The necessary driver were coded and the different kind of software packages were configured in such a way, that a autonomous navigation is now possible. But there are still some imperfect behaviors. For analysis and configuration, various tools of the ROS system were used. For the driver of the sensors, input devices and miscellaneous software, which were self coded, Python 2.7 was used.
The complete system is fully implemented and working. Furthermore, it's possible to use different kind of ROS tools to debug and analyze bugs. Because of the modularity of the software, it is still easy to extend the hard- as well as the softwarestack.
Overall it was shown, that the obstacle avoidance and navigation is working just as the mapping process. But there are still problems to tackle. The navigation stack doesn't recognize every kind of obstacle. The sensors used are insufficient. There is a need for more sensor data to sense and analyze the environment more detailed.
To conclude: the robot cannot operate safely without human monitoring. It still needs human intervention possibilities. To tackle these issues, ideas to fix these problems have been proposed.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation . . . 1
1.2 Ziel der Arbeit . . . 3

2 Grundlagen **5**
2.1 Einführung . . . 5
2.2 Robot Operating System . . . 6
2.3 Odometrie . . . 7
2.3.1 Art des Antriebs . . . 7
2.3.2 Odometrie bei Differential Antrieben . . . 8
2.3.3 Unechte Odometrie . . . 10
2.4 SLAM Beschreibung und Algorithmen . . . 10
2.4.1 SLAM Begriffsdefinition . . . 11
2.4.2 Formen von SLAM Algorithmen . . . 11
2.4.3 HectorSLAM Algorithmus . . . 12
2.4.4 Funktionsweise des 2D Lidar . . . 15
2.5 Funktionsweise der Lokalisierung und Navigation . . . 18
2.5.1 Lokalisierung mit Adaptive Monte Carlo Localization . . . 18
2.5.2 Trajektorienberechnung und Costmaps . . . 22

3 Technische Umsetzung **25**
3.1 Hardwarebeschreibung und Steuerung . . . 26
3.1.1 Verfügbare Sensorik . . . 27
3.1.2 Manuelle Steuerung . . . 28
3.2 Netzwerkinfrastruktur . . . 29
3.2.1 Verwendete Netzwerktechnik . . . 29
3.2.2 Verwendung des Netzwerks . . . 30
3.2.3 ROS Netzwerkprotokolle . . . 31
3.3 Softwarestruktur . . . 31
3.3.1 scoomatic_ros1 Paket . . . 32

3.3.2 scoomatic_drive Paket . . . 33
3.3.3 scoomatic_description Paket . . . 33
3.4 Softwarebeschreibung . . . 34
3.4.1 Daten- und Austauschformate . . . 34
3.4.2 Verwendete Tools . . . 38
3.4.3 Python Programmierung . . . 38
3.4.4 Verwendete Grafische Software . . . 38
3.5 Kartierung . . . 40
3.5.1 Datenfluss beim SLAM Prozess . . . 41
3.5.2 Vergleich von ROS SLAM Packages . . . 42
3.5.3 HectorSLAM . . . 43
3.6 Lokalisierung und Navigation . . . 44
3.6.1 Datenfluss bei der Navigation . . . 45
3.6.2 Navigationsproblemlöser: Recovery Behaviors . . . 49

4 Validierung des Systems **51**
4.1 Kartierung . . . 51
4.1.1 Längenmaßgenauigkeit . . . 51
4.1.2 Glastürproblematik . . . 53
4.2 Verhalten der Navigation . . . 54
4.3 Bestehende Herausforderungen . . . 55
4.3.1 IMU Probleme . . . 55
4.3.2 Mobiler Netzwerkzugriff . . . 55
4.3.3 Rechenkapazität des Raspberry Pi 3B . . . 56

5 Ausblick **57**

6 Fazit **59**

Abbildungsverzeichnis **61**

Akronyme **63**

Literatur **65**

1 Einleitung

In dieser Arbeit wurde eine Hinderniserkennung und die Vermeidung von statischen sowie dynamischen Hindernissen während einer autonomen Navigation eines Elektrokleinstfahrzeug-Prototyps realisiert. Nach einer kurzen Einführung in das Thema und den notwendigen technischen sowie mathematischen Grundlagen wird erläutert, wie das System realisiert wurde. Anschließend findet eine Überprüfung der Anforderungen und des Systems selbst statt. Schlussendlich endet die Arbeit mit einer Diskussion der Zukunftsfähigkeit des Systems und einem Fazit.
Dabei wird insbesondere das Robot Operating System (ROS) vorgestellt und wie dessen Funktionen und Software genutzt wurde. Das Meta-Betriebssystem ist die Plattformbasis für den gesamten Softwarestack.

1.1 Motivation

Mobilität ist ein alltägliches Bedürfnis von Menschen. Sie müssen einkaufen, zur Arbeit gehen oder sie bewegen sich in ihrer Freizeit in ihrer Umwelt. Es gibt verschiedenste Anlässe mobil sein zu müssen oder zu wollen. Dabei sind die Arten von Fortbewegungen in Deutschland sehr divers[3]. So können innerstädtische Strecken, neben den klassischen Verkehrsmitteln dem Zufußgehen, Fahrrad, Öffentlicher Nahverkehr und PKW, bspw. mit Skateboards, Rollstühlen oder E-Scootern absolviert werden. Letztere spielen insbesondere auf kürzeren Strecken, der sogenannten *letzten Meile* eine Rolle und können die Anschluss-Mobilität zu anderen Verkehrsmitteln darstellen. Das Vorankommen auf kurzen Strecken, insbesondere im städtischen Bereich und mit Elektrokleinstfahrzeugen, wird auch Mikromobilität genannt. Zu beachten ist die Begriffsdefinition des E-Scooters. Hier sind Elektrokleinstfahrzeuge gemeint, die gesetzlich maximal $20\frac{km}{h}$ fahren dürfen und in der Regel stehend zu fahren sind.
Es findet eine Verdichtung und Erhöhung des Automobil-Verkehrs[22] statt. Daraus resultieren Staus[11] und Umweltverschmutzungen[13]. Diese Faktoren machen Elektrokleinstfahrzeuge zu einer Alternative im städtischen Verkehr. Allerdings besteht in vielen Städten Deutschlands noch das Problem, dass keine Flächengerechtigkeit[21, S.

4f] herrscht. Das bedeutet, dass Kraftfahrzeugen überdurchschnittlich mehr Fläche im Verhältnis zu anderen Verkehrsmitteln zur Verfügung steht, wie Wege mit Kraftfahrzeugen absolviert werden. Eine gerechtere Flächenaufteilung würde Mikromobilität noch attraktiver machen.

Durch Smartphone-Apps wird die Nutzung von Sharing-Konzepten deutlich erleichtert. Diese Dienste bieten bspw. Fahrräder, Pedelecs oder E-Scooter zum Verleih an. Es existieren stationsbasierte als auch stationslose Konzepte, welche jeweils Vor- und Nachteile besitzen. Bei stationsbasierten Sharing-Konzepten können Fahrzeuge nur an ausgewählten Stationen entliehen und wieder zurückgebracht werden. Bei stationslosen Konzepten werden die Fahrzeuge im öffentlichen Raum abgestellt. Probleme beider Konzepte sind Vandalismus[10] an freistehenden Fahrzeugen. Bei stationslosen Konzepten ist zusätzlich noch das wiederkehrende Aufladen, wofür das Fahrzeug aktuell noch abgeholt werden muss, notwendig. Denn wenn der Akku der Elektrofahrzeuge entladen ist, muss meist das gesamte Fahrzeug zur Aufladestation gefahren werden, weil der Akku nicht entnehmbar ist. Für beide Probleme gibt es Lösungsmöglichkeiten. Eine Möglichkeit um die Probleme zu lösen wäre, dass die Fahrzeuge zu den Benutzenden autonom fahren. Somit wäre es den Fahrzeugen möglich autonom bei niedrigem Rest-Akkuladestand zur Ladestation zu fahren. Vandalismus wird vermieden, in dem die Fahrzeuge nicht im öffentlichen Bereich, sondern in geschützter Umgebung stehen.

Für das autonome Fahren sind jedoch umfangreiche technische Voraussetzungen notwendig. Die Umwelt muss erfasst, analysiert und die Daten evtl. gespeichert werden. Beim Planen von Routen im Raum müssen die Informationen zur Pfadplanung als auch der Start- und Zielort bekannt sein. Zudem muss der Verkehr beachtet werden. Für diese Disziplinen sind mehrere, verschiedene Sensoren notwendig um die Umwelt wahrzunehmen. Um Autonomie des Fahrzeugs sicherzustellen, ist es notwendig, dass die Daten auf dem Fahrzeug verarbeitet werden und somit genügend Rechenkapazität zur Verfügung steht.

Zur Erforschung der Mikromobilität wurde die Mikromobilitätsplattform Scoomatic am Lehrstuhl für Mechatronik der Univeristät Augsburg ins Leben gerufen. In diesem Projekt wird die bereits genannte Lösung eines autonomen Elektrokleinstfahrzeug realisiert. Die Idee der Mikromobilitätsplattform Scoomatic ist, dass die Benutzenden des Fahrzeugs dieses ordern können. Das Ziel ist dann, dass das Fahrzeug voll-autonom zu der Person fährt. Dabei fahren die Fahrzeuge aus einer geeigneten, sich in der Nähe befindenden zentraler oder dezentraler Ladestation. Somit wird es auch möglich sein, bei niedrigem Akkustand, dass das Fahrzeug sich selbstständig in einen Ladezustand versetzt.

In dieser Arbeit wurde die notwendige Software für die autonome Hinderniserkennung und -vermeidung entwickelt. Für die Autonomie des Roboters sind verschiedene Disziplinen zu absolvieren, die in den folgenden Kapiteln ausgeführt werden.

1.2 Ziel der Arbeit

Basis dieser Arbeit ist die Entwicklung eines Softwarestacks. Dieser soll die Hinderniserkennung und -vermeidung während einem Navigationsvorgang von einer gegebenen Start- und Zielpose ermöglichen. Die Hardware besteht aus einem umgebauten Hoverboard mit passiven Stützrädern, einer Plattform zum Stehen und der Sensoren sowie deren Halterungen. Die weiterführende Erklärung findet in Abschnitt 3.1 statt. Die Erstellung des Fahrzeugs war Teil einer vorherigen studentischen Arbeit von Martin Schoerner. In dieser Arbeit wurde der Scoomatic Prototyp baulich nicht verändert.
Im Zuge der Arbeit wurde die bestehende Software, basierend auf ROS2 und ROS1, auf eine einheitliche ROS1 Umgebung umgebaut. Das ROS ist ein Meta-Betriebssystem, das standardisierte Verfahren sowie Sofwarepakete verwendet und bereitstellt. Es ist also kein klassisches Betriebssystem, sondern benötigt ein solches als Basis. In diesem Fall wurde ROS auf einem Raspberry Pi 3B mit Ubuntu 18.04 ausgeführt, dabei wird ROS in Abschnitt 2.2 eingeführt.
Ziel war es, die vorhandenen Sensoren für eine Kartenerstellung und Navigation zu nutzen. Um schlussendlich eine Hindernisvermeidung für statische als auch für dynamische Hindernisse bei der Navigation im Raum zu ermöglichen. Hierzu wurde auf bestehende Pakete von ROS zurückgegriffen.
Dies umfasste eine Integration von HectorSLAM für die Kartenerstellung (Abschnitt 3.5) sowie das Navigation Paket für Lokalisierung und Navigation (Abschnitt 3.6).

2 Grundlagen

Dieses Kapitel dient dazu die in den nachfolgenden Kapitel verwendeten Begriffe und Konzepte besser zu verstehen. Im folgenden Abschnitt werden die Thematik eingeführt und die dafür notwendigen Begriffsdefinition definiert.

2.1 Einführung

Für autonome Aktionen müssen Informationen aus der Umwelt bekannt sein. Ohne Informationen ist sonst keine Aktion sinnvoll planbar. Die Umwelt muss also in einem geeignetem Modell für den auszuführenden Roboter vorliegen, damit mit diesem gearbeitet werden kann. Um die Umgebung wahrnehmen zu können, werden Sensoren benötigt. Für die Navigation bedarf es insbesondere einer Karte, um Pfade planen zu können. Diese Karte kann entweder 2-dimensional oder 3-dimensional vorliegen. Um die erforderlichen Maße innerhalb des Raumes zu messen, existieren verschiedene Technologien. Praktikabel sind für diesen Anwendungszweck jedoch nur solche, die rund um den Scoomatic herum, und nicht nur in einem Winkel oder Winkelbereich die Distanzen messen. So müssen die verschiedenen Datenquellen nicht fusioniert werden. Zudem ist wünschenswert, dass die Daten regelmäßig und in geeigneter Häufigkeit erfasst werden, damit eine durchgehende Bewegung und Verarbeitung der Daten ermöglicht wird. Es ist nicht erwünscht, dass zum Datenaufnehmen angehalten werden muss. Durch eine Bewegung des Scoomatic muss an jeder diskret berechneten Pose klar sein, in welche Richtung sich bewegt werden kann und in welche nicht.
Simultaneous Localization and Mapping (SLAM) ist ein Verfahren zur Erstellung von Karten. Es existieren online und offline Verfahren. Bei Ersterem steht die Karte während der Erstellung der Karte bereits zur Verfügung. Bei Letzterem passiert die Erstellung der Karte erst nach abschließen aller notwendigen Bewegungen. Es besteht jedoch in ROS die Möglichkeit auch online SLAM Verfahren ebenfalls erst nach Abschließen der Bewegungen in Karten umgewandelt werden. Hierbei werden die von den Sensoren versendeten Daten mit dem Tool *rosbag*[1] aufgenommen und in einer bag-Datei gespei-

[1] `https://wiki.ros.org/rosbag`

chert. Später, können diese dann wieder abgespielt werden. Dem SLAM-Algorithmus wird dann die realen Sensorwerte vorgetäuscht. Diese Funktion ist aber auch für andere Software in ROS möglich zu verwenden.
Sensoren, die klassisches SLAM ermöglichen sind Lidar-Sensoren.
Daneben existieren noch Visual-SLAM Techniken, die auf Basis von RGBD-Kameras funktionieren und eine 3D Karte erstellen können. RGBD-Kameras haben einen klassischen Fotosensor, der Farbinformationen aufnimmt und zusätzlich noch eine Tiefenkamera, die Tiefeninformationen aufnimmt. Solche Sensoren kommen hier allerdings nicht zum Einsatz.

2.2 Robot Operating System

Das Robot Operating System, im Folgenden ROS genannt, war in dieser Arbeit die Plattformbasis. Es wurde anfangs für einzelne Robotersysteme entwickelt und ist mittlerweile Netzwerkfähig. Dies ermöglicht es mehrere Systeme über Wireless Local Area Network (WLAN) oder Kabelgebundener Netzwerke mit einem gemeinsamen *ROS Master* zu betreiben. Der **Master**[2] ist zuständig für die Verwaltung bzw. Bereitstellung und Registrierung der Nodes, Services, Actions, Bags, Topics und Parameter-Server.
Nodes[3] sind eigenständige Prozesse also Programme, in der Regel mit eigenem Sourcecode. Sie kommunizieren über Topics. **Topics**[4] sind Datenströme, die *Messages* übertragen. **Messages** enthalten jegliche Arten von Daten. Sie werden über die Topics von den bereitstellenden Nodes zu anderen konsumierenden Nodes verschickt und nur an diese. Bereitstellende Nodes erzeugen Daten, konsumierenden Nodes konsumieren Daten. Abhängig davon, welche konsumierende Node die spezifische Topic abonniert hat. Der Master arbeitet nach dem *Publish-Subscriber Prinzip*. Das bedeutet, dass Nodes sich gewünschte Topics zur Laufzeit abonnieren und deabonnieren können. Die Daten werden dann von den bereitstellenden Nodes nur an diese konsumierenden Nodes übertragen.
Ein oder mehrere Nodes können mit Launchfiles gestartet und konfiguriert werden. **Launchfiles** werden in XML verfasst[5]. In diesen können dann ebenfalls Argumente für das Starten der Nodes bzw. Parameter festgelegt werden. Parameter stehen global im gesamten ROS Netzwerk über den Parameter Server zur Verfügung. Der **Parameter**

[2] `https://wiki.ros.org/Master`
[3] `https://wiki.ros.org/Nodes`
[4] `https://wiki.ros.org/Topics`
[5] `https://wiki.ros.org/roslaunch/XML`

Server[6] wird zusammen mit dem Master gestartet. Parameter können allerdings auch auf anderen Wegen festgelegt werden, die hier allerdings nicht verwendet wurden.

Für tiefer gehende Informationen wird auf die jeweiligen Seiten im ROS Wiki verwiesen: `https://wiki.ros.org/`.

2.3 Odometrie

Dieser Abschnitt beschreibt die mathematischen Grundlagen der Odometrie des Roboters. Daneben wird der Roboter noch basierend auf seiner Fortbewegungsart klassifiziert.

2.3.1 Art des Antriebs

Der Roboter besteht aus zwei Teilen. Das Chassis beinhaltet das umgebaute Hoverboard mit zwei aktiven, also angetriebenen und parallel ausgerichteten Rädern. Jedes Rad hat seinen eigenen Motor, der das jeweilige Rad beschleunigen kann. Dabei handelt es sich um ein Differentialantrieb. Daneben gibt es noch zwei weitere, passive Castor-Räder. Diese können sich um das Rad und an der Aufhängung drehen und sind nicht angetrieben.[9, S. 113] Die beiden passiven Castor-Rädern dienen nur der Stabilisierung des Gefährts.

Nun wird das mathematische Modell für einen Differentialantrieb vorgestellt. Damit ist es möglich die Position und den Gierwinkel zu berechnen.

[6] `https://wiki.ros.org/Parameter%20Server`

2.3.2 Odometrie bei Differential Antrieben

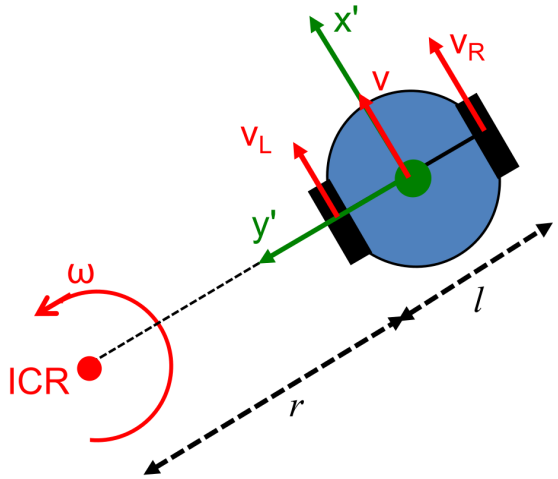

Abbildung 2.1: Modell eines Differentialantriebs
Bildquelle: [2, S. 14]

Mit dem folgenden Modell wird die Pose des lokalen Koordinatensystem des Roboters berechnet.
Es bestehen folgende Zusammenhänge, wie in Abbildung 2.1 zu erkennen ist:

$$v_l = \omega r$$

$$v = \omega(r + l/2)$$

$$v_r = \omega(r + l)$$

Daraus folgt:

$$v = \frac{v_r + v_l}{2}$$

und

$$\omega = \frac{v_r - v_l}{l} \tag{2.1}$$

l ist die Breite des Fahrzeuges und entspricht $62,2cm$.
Dies ist die Vorwärtskinematik des Roboters mit Differentialantrieb.
v_r ist die Geschwindigkeit des Roboters, welche durch das rechte angetriebene Rad verursacht wird. Analog gilt dies für die Geschwindigkeit v_l und das linke Rad. Diese Geschwindigkeiten zeigen immer in die Richtung der x'-Achse des lokalen kartesischen Koordinatensystems des Roboters. Dieses Koordinatensystem zeigt immer in die Richtung nach vorne, also senkrecht zum Kurvenradius r. Die Geschwindigkeit v ist die

Gesamtgeschwindigkeit des Roboters. ω entspricht der Kreisgeschwindigkeit des Roboters und dreht sich um den Instantaneous center of rotation (ICR). Letztere wird auch Momentanpol genannt und gibt den Drehpunkt eines starren Körpers, also dem Roboter, in einer Ebene an.[2, S. 11]
Jetzt kann die Pose des Roboters in Abhängigkeit der beiden einzelnen Rädergeschwindigkeiten berechnet werden. Die Pose wird inkrementell, ausgehend von einem Startpunkt berechnet.[2, S. 14]
Um die Pose, also die Position und den Gierwinkel berechnen zu können, werden die beiden Geschwindigkeiten v und ω benötigt. Dazu wird die Pose angenähert durch das inkrementelle Addieren der Differenz-Posen innerhalb der der Zeitspanne dt. In der Zeitspanne dt wird das restliche Programm in einer Endlosschleife ausgeführt.[7]:
Aus den Winkelbeziehungen des Roboters in Abbildung 2.1 folgen dann für das lokale Koordinatensystem die Position und der Gierwinkel.
Die relativen Differenzpositionen zwischen einzelnen Zeitschritten werden folgendermaßen berechnet ([14]):

$$\Delta x = (v_x * \cos(th) - v_y * \sin(th)) * dt$$

und

$$\Delta y = (v_x * \sin(th) - v_y * \cos(th) * dt$$

Der Differenz-Gierwinkel wird berechnet mit:

$$\Delta\theta = \omega * dt$$

Somit kann die absolute Pose für den aktuellen Zeitschritt berechnet werden:

$$x_t = x_{t-1} + \Delta x$$

$$y_t = y_{t-1} + \Delta y$$

$$\theta_t = \theta_{t-1} + \Delta\theta$$

Dabei steht t für einen beliebigen, diskreten Zeitschritt. Initialisiert werden die Werte jeweils mit Null:
$x_0 = 0$, $y_0 = 0$ und $\theta_0 = 0$.
Das Koordinatensystem des Roboters ist in ROS so festgelegt, dass die x-Achse immer nach Vorne zeigt, also in Fahrtrichtung des Roboters. Deswegen ist hier im speziellen

[7] `https://wiki.ros.org/navigation/Tutorials/RobotSetup/Odom#Writing_the_Code`

Fall $v_x = v$, da sich der Roboter aufgrund des Differentialantriebs und der Anordnung der Räder niemals in y'-Richtung bewegen kann. Deswegen ist $v_y = 0$. Abgesehen von Erhöhungen im Untergrund, auf dem der Roboter fährt, wird die Geschwindigkeit v_z in der z-Achse grundsätzlich mit 0 festgelegt. Zudem ist bei einer Lokalisierung auf einer 2-dimensionalen Karte die Lokalisierung nur 2-dimensional möglich.

2.3.3 Unechte Odometrie

Zu beachten ist, dass die beiden Geschwindigkeiten der beiden Motoren nicht als echte Geschwindigkeitsmessung vorliegen, sondern sie sind nur eine Schätzung auf Basis der Leistung der Motoren (vgl. Abschnitt 3.4.1). Es gibt also keinen expliziten Sensor, der die Raddrehung misst. Einzig und alleine aufgrund der Soll-Leistung des Motors gibt es eine Information des Hoverboard-Mainboards. Deshalb werden die Pseudo-Geschwindigkeiten (einheitslos im Bereich [0,1000]) zunächst mit einem Faktor multipliziert, damit diese in der Einheit $\frac{m}{s}$ vorliegen. Zudem werden nur Geschwindigkeiten über 43 genutzt. Unterhalb dessen wurde keine Bewegung des Roboters ohne zusätzliche Masse festgestellt und führte so zu ungewolltem Verhalten. Ohne diesen Filter wurde in ROS eine Bewegung detektiert und weiterverarbeitet, die so nicht stattfand.
Die Geschwindigkeiten v_r des rechten Rads und v_l des linken Rads sind angenähert mit:

$$v_r = 0,006 * v_{r-pseudo}$$

$$v_l = 0,006 * v_{l-pseudo}$$

daraus kann dann die Drehgeschwindigkeit des Roboters mit

$$\omega = 0,0053 * \omega_{pseudo}$$

berechnet werden (Gleichung 2.1).
Die Werte $v_{r-pseudo}$, $v_{l-pseudo}$ und ω_{pseudo} sind dabei die Pseudo-Geschwindigkeiten und stehen als eigene ROS Topics (*/speed_r und /speed_l*)bereit. Die Faktoren wurden experimentell festgestellt indem Soll- und Ist-Strecke in RViz mit der realen Strecke verglichen wurde.

2.4 SLAM Beschreibung und Algorithmen

Die Pseudo-Odometrie steht nun zur Verfügung, folgend ist das Kartografieren der Umgebung notwendig. Dies wird mit SLAM realisiert. Dabei ist SLAM entscheidend für

die Hinderniserkennung und -vermeidung, denn es bestimmt später bei der Navigation, ob und wie gut der Globale Planer den Pfad zum Ziel planen kann. Zudem wird die Karte als Basis für die Globale Costmap verwendet.
Im Folgenden wird der SLAM Prozess näher erläutert. Dieser wird dazu verwendet um eine Karte der Umgebung zu erstellen. Es werden außerdem verschiedene SLAM Algorithmen vorgestellt, unter anderem HectorSLAM, der für diese Arbeit verwendet wurde.

2.4.1 SLAM Begriffsdefinition

SLAM steht für Simultaneous Localization and Mapping. Der Ausdruck bedeutet, dass die beiden Disziplinen gleichzeitig ausgeführt werden. Allerdings stellt dies ein Kausales Problem dar. Denn ohne die Existenz einer Karte ist keine Lokalisierung möglich. Ohne die Ortsbestimmung ist die Erstellung einer Karte des Raumes allerdings auch nicht möglich, denn so können die Sensordaten nicht korrekt abgeglichen werden (Scanmatching). Somit bedarf es spezieller Algorithmen, die dieses Problem lösen. Im Folgenden Abschnitt werden die unterschiedlichen Formen und Arten von SLAM vorgestellt.

2.4.2 Formen von SLAM Algorithmen

Es gibt verschiedene SLAM Algorithmen, die mit verschiedenen mathematischen Ansätzen das SLAM-Problem lösen. Dabei lassen sich SLAM Algorithmen aus Wahrscheinlichkeitstheoretischer Sicht in zwei Kategorien aufteilen. Zum Einen existiert Vollständiges-SLAM (Full-SLAM), das alle zurückgelegten Posen (die Zustände $x_{1:t}$) sowie die Karte m auf Basis der vorherigen Steuersignalen u und Beobachtungen z berechnet. Steuersignale sind die Motorkommandos, Beobachtungen entsprechen den Sensordaten. Mathematisch wird das Problem folgendermaßen beschrieben:

$$p(x_{1:t}, m|z_{1:t}, u_{1:t}) \tag{2.2}$$

Zum Anderen existiert sogenanntes Inkrementelles- oder Online-SLAM, welches die nächste Pose immer nur basierend auf der aktuellen Pose und den Steuersignalen bzw. Sensordaten berechnet. Die mathematische Definition sieht sehr ähnlich zu Full-SLAM aus:

$$p(x_t, m|z_{1:t}, u_{1:t}) \tag{2.3}$$

Enthält aber den wichtigen Unterschied, dass nur der aktuelle Zustand x_t und nicht $x_{1:t}$, also alle zurückgelegten Posen, berechnet werden.[20, S. 309f].
Full-SLAM hat den Vorteil, dass es Fehler besser korrigieren kann oder diese erst gar nicht auftreten. Denn Fehler der Pose bei Online-SLAM führen meistens zu Folgefehlern und damit zu nutzlosen Karten. Allerdings ist Full-SLAM deutlich Rechenaufwändiger und ist ein sogenannter Offline-SLAM. Das Ergebnis, also die Karte wird nicht kontinuierlich inkrementell erstellt und steht während der Berechnung nicht zur Verfügung. Ein bekannter Full-SLAM Algorithmus ist GraphSLAM[20, S. 337].
Der Term $p(x|y)$ ist dabei eine bedingte Wahrscheinlichkeit[20, S. 16]. Bei allen drei Verfahren: SLAM, der Lokalisierung und Navigation werden hier immer nur Schätzungen berechnet. Diese beinhalten immer eine gewisse Unsicherheit und erreichen keine absolute Genauigkeit. Das bedeutet unter anderem, dass die Karte, die mit SLAM erstellt wird in jedem Fall nur eine Schätzung mit Ungenauigkeiten ist. Die Schreibweise der bedingten Wahrscheinlichkeit ist so zu verstehen, dass der Algorithmus anhand von den Beobachtungen (also den Sensordaten) sowie Steuersignalen, die bekannt sind, eine oder eben alle Posen und die Karte schätzt, die unbekannt sind. Es handelt sich hier also um eine A-posteriori-Wahrscheinlichkeit. Eine A-posteriori-Wahrscheinlichkeit beschreibt die Wahrscheinlichkeit eines Systems bei unbekanntem Systemzustand, der abhängig von einer anderen Zufallsgröße ist[20, S.16f].

2.4.3 HectorSLAM Algorithmus

Im Abschnitt 3.5.2 wird erläutert, warum HectorSLAM als SLAM Algorithmus verwendet wird. Außerdem wird dargestellt, welche abstrakten Eigenschaften es hat und wird dabei mit anderen SLAM Packages verglichen.
In diesem Abschnitt wird die Funktionsweise von HectorSLAM erläutert. Das ROS Package stammt, wie die Idee, vom Team Hector von der Universität Darmstadt.
HectorSLAM ist ein Online-SLAM Verfahren. Es besitzt einen anderen mathematischen Ansatz wie die klassischen FastSLAM, EKF- und Landmarkenbasierten SLAM Ansätze. Odometrische Daten, die für diese klassischen SLAM Algorithmen unverzichtbar sind, sind hier nicht zwingend notwendig sondern optional. HectorSLAM verfolgt einen Scanmatching-basierten Ansatz, der die bisherige Karte mit den jeweils aktuellen Lidardaten vergleicht und versucht den Fehler zu minimieren, damit die Übereinstimmung optimal ist. Somit wird die Karte inkrementell aufgebaut und verfolgt einen probabilistischen Ansatz. HectorSLAM kann die Pose in 6 Freiheitsgraden bestimmen, wenn eine Inertial Measurement Unit (IMU) verwendet wird.

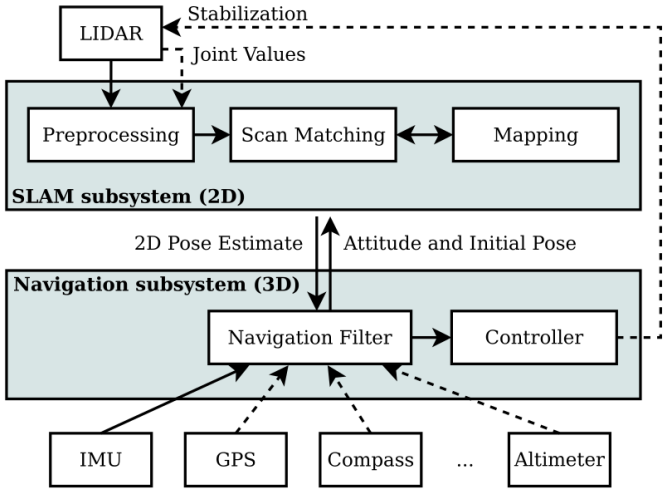

Abbildung 2.2: Übersicht über Navigation und Mapping Prozess
[12]

Dieser Sensor bildet zusammen mit weiteren Sensoren das *Navigation subsystem*, wie in Abbildung 2.2 zu sehen ist.[12, S. 156]
Das Navigation Subsystem wird hier jedoch nicht tiefer gehend erklärt, stattdessen liegt der Fokus auf dem *SLAM subsystem*.

Scanmatching

Zentraler Bestandteil des HectorSLAM Algorithmus ist das Scanmatching Verfahren. Die Idee des Scanmatching ist, dass zwei Scans - in diesem Fall Lidarscans - optimal überlagert werden. Dabei werden auf die Scans euklidische Transformationen ausgeführt, die abstands- und winkelerhaltend sind. Scanmatching setzt voraus, dass nacheinander folgende Scans des Light detection and ranging (Lidar) überwiegend übereinstimmen. Nur so können die Strukturen des Raumes zwischen nachfolgenden Scans gut überlagert werden.[9, S. 177] So ist ersichtlich, dass eine hohe Scanfrequenz des Lidar von Vorteil ist und eine gewisse Mindestgeschwindigkeit der Drehvorrichtung erreicht werden muss. Genauso wie Scans können auch Scan und Karte überlagert werden[9, S. 177].
HectorSLAM verwendet eine diskrete *Occupancy Grid Map* um die Darstellung jeglicher Formen von 2D-Karten zu ermöglichen. Diese wird in ROS zur Verfügung gestellt bzw. kann konsumiert werden. Weiterführende Informationen werden im nächsten Abschnitt erläutert.
Die Lidardaten werden zunächst in eine Punkt Wolke (Point Cloud) umgewandelt. Es wird dann ein Filter angewendet, der Punkte außerhalb der Scanebene entfernt.

Es handelt sich bei HectorSLAM um ein Gauß-Newton Verfahren, also ein Minimierungsproblem. Es wird versucht den Fehler von $\triangle\boldsymbol{\xi}$ des Laserscans zur Anfangsschätzung $\boldsymbol{\xi}$ in Bezug auf die bestehende Karte, also zur optimalen Orientierung und Translation, zu minimieren.
Für die Taylorentwicklung von $M(\boldsymbol{S}_i(\boldsymbol{\xi} + \triangle\boldsymbol{\xi}))$ beim Gauß-Newton Verfahren wird jeweils von der Karte, $\boldsymbol{S}_i(\boldsymbol{\xi} + \triangle\boldsymbol{\xi})$ (s.u.) und $\boldsymbol{\xi}$ die erste Ableitung bzw. der Gradient benötigt. Das kann aber nicht in jedem Fall direkt berechnet werden.
Der Gradient der Karte kann nicht einfach berechnet werden, da die Karte nur in einem diskreten Format vorliegt. Dementsprechend muss eine kontinuierliche Annäherung der Karte berechnet werden, was hier mit einer bilinearen Interpolation erfolgt.
Beim Scanmatching wird einem Scan eine 2-dimensionale Transformation $\boldsymbol{\xi}$ zugewiesen: eine 2-dimensionale translatorische und 1-dimensionale rotatorischen, ein Gierwinkel (Yaw):

$$\boldsymbol{\xi} = (p_x, p_y, \psi)^T$$

Die minimale Transformation des Scans um auf die Karte zu passen, für die das Optimum gesucht wird - also die Grundidee des Algorithmus - ist:

$$\boldsymbol{\xi}^* = \underset{\boldsymbol{\xi}}{\operatorname{argmin}} \sum_{i=1}^{n} [1 - M(\boldsymbol{S}_i(\boldsymbol{\xi}))]^2$$

Der Algorithmus minimiert mithilfe der Methode der kleinsten Fehlerquadrate die Fehler in der Ausrichtung der Laserscans im Vergleich mit der bestehenden Karte.
$M(\boldsymbol{S}_i(\boldsymbol{\xi}))$ berechnet hier die Wahrscheinlichkeit eines Hindernisses einer Zelle i in der Karte an den Koordinaten von $\boldsymbol{S}_i(\boldsymbol{\xi})$.
Wobei $\boldsymbol{S}_i(\boldsymbol{\xi})$ die Weltkoordinaten eines Scan-Endpunkt i sind. Ein Scan-Endpunkt ist der Punkt *einer* Distanzmessung, als Teil eines gesamten Scans, des Lidar im Koordinatensystem. Dementsprechend wird die Optimierung für jeden Scan-Endpunkt durchgeführt, aber nicht für jeden einzeln, sondern in seiner Gesamtheit. [12, S. 156f]

Kartenrepräsentation: Occupancy Grid Maps

Die verarbeiteten Daten, die zu einer Karte führen, müssen für die weitere Verarbeitung in einer passenden Art und Weise gespeichert werden. Allgemein ist die Frage, wie eine solche Karte gespeichert werden kann.
Zunächst muss festgestellt werden, was gespeichert werden muss. In diesem Fall sind verschiedene Wahrscheinlichkeitswerte zu speichern, also die Wahrscheinlichkeit, dass

an einer bestimmten Position ein Hindernis vorkommt.
Eine Occupancy Grid Map, zu deutsch Belegungs-Raster-Karte, ist also eine diskretisierte Wahrscheinlichkeitsverteilung in einem 2-dimensionalen Raum - einem kartesischem Koordinatensystem. Eine mögliche mathematische Repräsentation ist eine $n \times n$ Matrix. Die Zellen werden jedoch fortlaufend bezeichnet. Jeder einzelnen Zelle $m_i \in [0, 1]$ kann eine Belegungswahrscheinlichkeit zugewiesen werden. Somit ergibt sich dann die Karte $m = \{m_i\}$ aus 2.2 bzw 2.3, die aus mehreren Zellen m_i besteht.
Für $m_i = 1$ würde dies bedeuten, dass die Zelle i mit einer Wahrscheinlichkeit von 100% okkupiert ist.[20, S. 285]

Multi-Resolution Kartendarstellung

Bei gradientenbasierten Verfahren besteht die Gefahr, dass das Verfahren nur ein lokales Minimum findet.[12, S. 158] Dies ist zu vermeiden. Um dieses Problem zu umgehen werden in HectorSLAM mehrere Occupancy Grid Maps mit unterschiedlichen Auflösungen erstellt. Diese werden allerdings nicht ausgehend von *einer* Karte generiert, sondern jede Karte wird einzeln mit jedem Pose-Update aktualisiert. Somit wird rechenaufwendiges Downsampling der Karten vermieden. Das Ausrichten des Scans erfolgt dabei zu erst mit der gröbsten Karte. Das Ergebnis der gröberen Karte wird dann für die nächst feinere Karte als Anfangswert des Algorithmus verwendet.[12, S. 158]

2.4.4 Funktionsweise des 2D Lidar

Der Lidar bildet die Basis um alle entscheidende Verfahren der Hinderniserkennung und -vermeidung zu realisieren. Deswegen wird die Funktionsweise allgemein und im speziellen des hier verwendeten 2D Lidar erläutert.

Allgemeines Prinzip von Lidar

Das allgemeine Prinzip von Lidar ist: Aussenden und Erkennen von Licht um daraus die Entfernung zu einem reflektierenden Objekt zu messen. Das bedeutet, dass das Objekt, zu dem die Entfernung gemessen wird, dieses Licht nicht absorbieren darf sondern zu einem gewissen Grad reflektieren muss. Sonst kann der reflektierte Laser nicht detektiert werden. Notwendig für diese Technik ist immer ein aktiver Laseremitter, der einen Laser aussendet und ein geeigneter Sensor, der diesen wieder über den Photoeffekt erkennen kann. Eine technische Vorrichtung, die auf Letzterem basiert sind insbesondere CCD- und CMOS-Sensoren.

Lidar Sensoren gibt es in 2D und 3D Varianten. 3D Sensoren nehmen, im Vergleich zu 2D Lidar Sensoren, mehrere und nicht nur eine horizontale Ebene auf. Dementsprechend können mit SLAM auch entweder 2D Karten oder 3D Karten erstellt werden. Dafür sind jedoch unterschiedliche SLAM Algorithmen notwendig. In dieser Arbeit wurde ein 2D Lidar verwendet, der im Vergleich zu 3D Lidar Kosten spart. Der 2D-Lidar führte aber zu Problemen, die in 4.1.2 diskutiert werden.

Funktionsweise Laser Triangulation

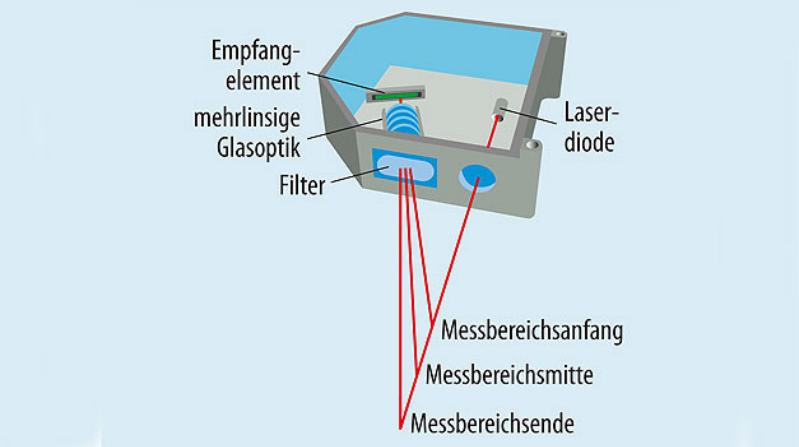

Abbildung 2.3: Veranschaulichung des allgemeinen Laser Triangulation Prinzips
Bildquelle: [8]

Der Lidar in dieser Arbeit verwendet das Prinzip der Lasertriangulation. Das Prinzip ist in Abbildung 2.3 zu sehen. Dabei sendet eine Laserdiode zunächst einen Laserimpuls aus. Der Sensor sitzt seitlich neben der Laserdiode und ist leicht gedreht, um die Reflektionen detektieren zu können. Das einstrahlende Licht wird durch entsprechende Glasoptik gebrochen, damit der gesamte, vorgesehene Messbereich mit dem Sensor erfasst werden kann.[8] Eine 1-dimensionale Sensorzeile detektiert dann durch den Photoelektrischen-Effekt den Punkt auf dem Sensor und kann so mit den Winkelbeziehungen die Distanz zum Objekt messen. Weil der RPLidar nur eine horizontale Ebene aufnimmt, wird nur ein 1-zeiliger Sensor benötigt. Die Genauigkeit ist hier abhängig von der Distanz des Objektes und verschlechtert sich, um so weiter entfernt das Objekt ist.[1, Figure 8]. Der Messbereich wird durch die Größe des Sensors sowie den Linsen maßgeblich begrenzt. Er könnte durch eine schlechtere Auflösung bei Austauschen den Linsen, die das Licht stärker brechen, einfach vergrößert werden.
Der Vorteil von Laser Triangulation ist der vergleichsweise günstige Preis der Technologie. Allerdings sind damit Glas und Flüssigkeiten schwer zu erkennen.[1, S. 449] Diese

Problematik wird weitergehend in 4.1.2 erläutert.

Beschreibung des RPLidar A1

Abbildung 2.4: Schrägansicht des RPLidar A1
Bildquelle: `https://www.slamtec.com/en/Lidar/A1`

Der in dieser Arbeit verwendeter Lidar ist ein RPLidar A1 von SLAMTEC. Er kann als drei Sub-Teile betrachtet werden: Eine Plattform, mit der der Scanner fixiert werden kann, der Antriebsmotor und die Scaneinheit selbst. Er ist in Abbildung 2.4 abgebildet. Die Scaneinheit hat eine Öffnung für den auszusendenden, sowie eine für den zu empfangenden Laserimpuls. Der Motor ist seitlich der Scaneinheit platziert und treibt über ein Gummiband die runde Scaneinheit an.
Der RPLidar A1 hat eine maximale Reichweite von 12m und benötigt einen Mindestabstand von 15cm zu Objekten, um diese messen zu können. Dabei hat er eine Genauigkeit von $< 1\%$ Genauigkeit der Distanz und eine Genauigkeit von $< 1°$ bei der Winkelbestimmung. Er verwendet einen Infrarot Laserimpuls mit einer typischen Wellenlänge von $785nm$[19, S. 9] Er hat eine Universal Asynchronous Receiver Transmitter (UART) Schnittstelle, die mit einem UART-USB-Adapter am Raspberry Pi verwendet wurde. UART ist eine sehr einfache Datenkommunikation, welche genauso wie USB Daten seriell überträgt. Er kann mit USB mit Strom versorgt werden und überträgt auch über diese Schnittstelle die Daten.
Um rundherum um den Scanner messen zu können, und nicht nur in eine bestimmte Richtung, ist die vorhandene Drehvorrichtung notwendig. Diese wird mit einem Motor gedreht. Wenn der Lichtimpuls an einem Objekt reflektiert wird, wird dieser durch eine Linse wird dieser im optimalen Fall von dem Lichtsensor erkannt. Der Winkel des Motors wird dabei zusammen mit der Distanz ausgegeben[19, S. 7]. Während der Motor gedreht wird, werden ständig Messungen vorgenommen, damit der gesamte Raum gescannt

wird. Somit kann jeder Messung ein Winkel und die Distanz zugewiesen werden, was in der Gesamtheit in einem 2-dimensionalen, kartesischem Koordinatensystem eine fast Karten-ähnliche Struktur ergibt.
Nachdem die Grundlagen des Kartenerstellungsprozess abgeschlossen wurde, folgt die Einführung in die Lokalisierung und Navigation.

2.5 Funktionsweise der Lokalisierung und Navigation

Der Navigationsprozess kann erst gestartet werden, wenn der SLAM-Prozess abgeschlossen ist, um dann eine Karte bereitstellen zu können. Die Lokalisierung kann grundsätzlich unabhängig von der Navigation verwendet werden, wird in dieser Arbeit aber insbesondere im Zusammenspiel mit der Navigation benötigt. Mit dem System ist die Navigation während der Kartenerstellung nicht möglich. Im Folgenden Abschnitt werden die mathematischen und algorithmischen Begebenheiten der Lokalisierung sowie Navigation und deren Zusammenspiel erläutert.

2.5.1 Lokalisierung mit Adaptive Monte Carlo Localization

In dieser Arbeit wurde für die Lokalisierung ein ROS Package mit der Bezeichnung AMCL[8] verwendet. Dahinter steckt eine Adaptive Monte Carlo Localization (AMCL). Dies ist eine spezielle Form einer Monte Carlo Localization (MCL), vgl. [20, S. 252f]. Eine Monte Carlo Lokalisierung ist ein Partikelfilter, der zur Zustandsschätzung verwendet wird. Ein Partikel stellt - in diesem Fall auf der Karte - einen möglichen Systemzustand, eine Pose dar. Durch iteratives Abtasten und Bewegen des Sensors bzw. des Roboters kann fortwährend eine bessere Ortsbestimmung erreicht werden. Die Wahrscheinlichkeit der Pose wird ebenfalls berechnet und als Kovarianzmatrix zur Verfügung gestellt. Dies kann als Gradmesser für die Genauigkeit der Posen-Schätzung dienen. Im optimalen Fall konvergieren die Partikel im Laufe des Lokalisierungsprozess zur tatsächlichen Pose.
Die Adaptive Monte Carlo Localization (AMCL) besitzt im Vergleich zur MCL ein Kullback-Leibler-Divergenz-Sampling (KLD-Sampling), das die Performance des Algorithmus verbessert. KLD-Sampling verbessert die Effizienz der Lokalisierung durch Benutzen einer variablen Partikel-Anzahl. MCL verwendet sonst eine statische Anzahl von Partikeln. Bei der globalen Lokalisierung wird eine hohe Anzahl von Partikeln benötigt, da diese über die gesamte Karte verteilt werden. Im Vergleich dazu bedarf es

[8] AMCL im ROS Wiki: `https://wiki.ros.org/amcl`

beim Verfolgen der Pose während der Bewegung jedoch eine geringere Anzahl, wenn bereits eine gute Lokalisierung gefunden ist. Denn in diesem Fall konzentrieren sich viele Partikel auf der Karte um den Roboter herum. Viele Partikel sind dann redundant und nicht notwendig.[20, p.263]

Beschreibung AMCL Algorithmus

Es folgt der Pseudocode des AMCL Algorithmus (Algorithmus 1), welcher aus [20, p.264] entnommen ist .

Algorithmus 1 Adaptive Monte Carlo Localization

1: **procedure** AMCL($\chi_{t-1}, u_t, z_t, \epsilon, \delta$)
2: $\chi_t = \emptyset$
3: $M = 0, M_\chi = 0, k = 0$
4: **for all** b in H **do**
5: $b =$ empty
6: **end for**
7: **do**
8: draw i with probability $\propto \omega_{t-1}^{[i]}$
9: $x_t^{[M]} =$ **motion_model**$(u_t, x_{t-1}^{[i]})$
10: $\omega_t^{[M]} =$ **measurement_model**$(z_t, x_t^{[M]}, m)$
11: $\chi_t = \chi_t + \langle x_t^{[M]}, \omega_t^{[M]} \rangle$
12: **if** $x_t^{[M]}$ falls into empty bin b **then**
13: $k = k + 1$
14: $b =$ non-empty
15: **if** $k > 1$ **then**
16: $M_\chi := \frac{k-1}{2\epsilon}\{1 - \frac{2}{9(k-1)} + \sqrt{\frac{2}{9(k-1)}} z_{1-\delta}\}^3$
17: **end if**
18: **end if**
19: $M = M + 1$
20: **while** $M < M_\chi$ *or* $M < M_{\chi min}$
21: **return** χ_t
22: **end procedure**

Die Parameter des Algorithmus 1 sind folgende Daten:

- χ_{t-1}: Die A-priori-Verteilung der Samples
- u_t: Die Steuerkommandos zum Zeitpunkt t
- z_t: Die Sensordaten zum Zeitpunkt t
- ϵ: Der Soll-Fehler zwischen zwei Wahrscheinlichkeitsverteilungen

- δ: Das gewünschte, definierbare Quantil

Zunächst werden die benötigten Variablen initialisiert. Dazu gehören die tatsächliche Anzahl von Partikeln M und die optimale Anzahl M_χ, die durch KLD berechnet wird. H bezeichnet ein Histogram, wobei b (bins) die Klassen in diesem definieren.
Wie bei MCL auch, werden die bestehenden Samples mit den Sensordaten in Zeile 8 gewichtet (Sampling). Die Funktion *motion_model* aktualisiert die A-priori Samples mit den aktuellen Steuerkommandos, siehe nächster Abschnitt Bewegungsmodell. Die Funktion *measurement_model* aktualisiert dementsprechend die Gewichte der gerade erzeugten Partikel.
In Zeile 11 werden die Samples zusammen mit ihren Gewichtungen in Tupeln zwischengespeichert. Ab Zeile 12 unterscheiden sich MCL und AMCL. Die Partikel werden nun in die Klassen im Histogramm eingeordnet. k gibt dabei die Anzahl der nicht leeren Klassen an. Die entscheidende Berechnung der optimalen Samples-Größe wird in Zeile 16 ausgeführt. Damit gilt, dass bei konzentrierteren Samples um den Roboter herum mehr Samples in weniger Klasse fallen und damit mehr Klassen leer sind. In diesem Fall wird die Anzahl der Partikel weniger, im umgekehrten Fall sind in vielen Klassen mindestens ein Partikel enthalten und somit wird von einer Lokalisierung mit großer Unsicherheit ausgegangen. Die Samples-Größe ist dementsprechend größer.
Solange die optimale Größe M_χ oder die Mindestanzahl $M_{\chi min}$ noch nicht erreicht ist, werden mehr Samples erzeugt (Zeile 20).
Die Parameter δ und ϵ beeinflussen die Anzahl der Samples in Zeile 16. δ stellt das obere $1 - \delta$ Quantil der Standardnormalverteilung dar. KLD ist ein Maß für die Unterschiedlichkeit zwischen zwei Wahrscheinlichkeitsverteilungen. ϵ gibt dabei den Fehler zwischen der tatsächlichen Samples-Verteilung und die durch das Histogramm an.
Somit gilt auch hier der Ablauf, wie bei MCL:

1. Initiales verteilen der Samples
2. Sensordaten einlesen und Samples gewichten
3. Resampling: Partikel neu verteilen und dabei Gewichtung beachten
4. Bewegung: Partikel mithilfe des Bewegungsmodell bewegen

Abgesehen von Punkt 1. werden die Aktionen in einer Schleife wiederholt. Mit AMCL ist im Vergleich zu MCL nur die Anzahl der Partikel variabel.
Am Ende des Algorithmus ist die neue Schätzung der Pose berechnet und wird zurückgegeben.

Bewegungsmodell

Die Funktion *motion_model* (Bewegungsmodell) sowie *measurement_model* (Sensormodell) sind beide jeweils abhängig vom Roboter bzw. vom verwendeten Sensor. Das Bewegungsmodell kann in AMCL festgelegt werden und basiert hier auf dem Algorithmus *sample_motion_model_odometry* von [20, S. 136]. Dieser löst das Problem $P(x_t|u_t, x_{t-1})$ und damit einen Teil des SLAM Problems, vgl. Gleichung 2.3. Jedoch soll hier für die Navigation auch nur eine Lokalisierung stattfinden. Der Algorithmus nimmt als Parameter den posteriori Zustand x_{t-1} und generiert zusammen mit den Odometriedaten u_t eine neue Schätzung x_t. [20, S. 136]
Dieser verwendet vier anpassbare Parameter $\alpha_1 - \alpha_4$, die die Unsicherheit der Odometrie festlegen für AMCL[9]:

- α_1: gibt die Unsicherheit der Rotation basierend auf der rotatorischen Bewegung des Roboters an
- α_2: gibt die Unsicherheit der Rotation basierend auf der translatorischen Bewegung des Roboters an
- α_3: gibt die Unsicherheit der Translatorischen Schätzung basierend auf der translatorischen Bewegung des Roboters an
- α_4: gibt die Unsicherheit der Translatorischen Schätzung basierend auf der rotatorischen Bewegung des Roboters an

Abbildung 2.5 zeigt ein Sampling für einen Roboter, der keine Sensordaten aufnimmt. Die einzelnen Partikel (Samples) zeigen den geschätzten Zustand (Belief) zu verschiedenen Zeiten, wobei die durchgezogene Linie eine Aktion, also eine Bewegung, darstellt. Die Partikel werden nur durch das Bewegungsmodell, welches die Daten der Odometrie verwendet, verändert. Hier ist gut ersichtlich, dass die Unsicherheit mit jeder Iteration vergrößert wird. Es zeigt die Notwendigkeit eines Sensors, hier eines Lidarsensors, auf. Zu beachten ist, dass die Form der Veränderungen der Samples immer maßgeblich vom Bewegungsmodell abhängig ist.

[9]Beschreibung der α-Werte von AMCL: `https://wiki.ros.org/amcl#Parameters`

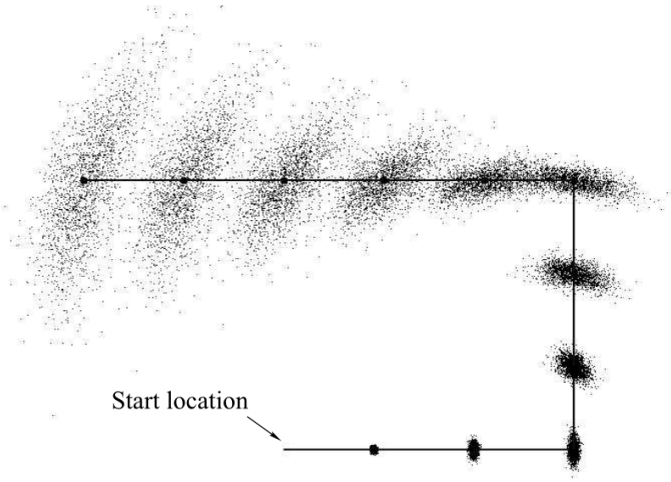

Abbildung 2.5: Belief des Roboters zu verschiedenen Zeitpunkten ohne Sensordaten
Bildquelle: [20, S. 138]

2.5.2 Trajektorienberechnung und Costmaps

Es existiert für ROS einen de facto Standard für Navigation[10] in einer 2D Karte. Dafür müssen die zuvor beschriebenen Voraussetzungen gelten. Das schließt insbesondere das Funktionieren einer Lokalisierung (hier AMCL), das Existieren einer Grid-Map und die Verfügbarkeit der entsprechenden Sensoren ein.
Notwendig für das Berechnen eines Pfades in der Karte ist ein passender Algorithmus.

Globale Trajektorienberechnung: Dijkstra Algorithmus

Der Dijkstra Algorithmus berechnet den kürzesten Pfad in einem aus nicht-negativen, gewichteten Kanten und Knoten bestehenden Graphen. Die Eingabe entspricht also einem Graphen und dem Startknoten. Der Algorithmus berechnet dann ausgehend vom Startknoten für jeden Knoten im Graphen den kürzesten Weg.[4]
Nun ist eine Karte jedoch kein Graph und muss erst in die entsprechende Form umgewandelt werden, damit dieser Algorithmus angewendet werden kann.
Für das Berechnen des Navigation-Pfades ist hier der *global_planner*[11] zuständig.
Basis für die Berechnungen ist hier eine *costmap_2d*. Diese besteht aus Zellen, die Kosten enthalten, je nachdem wie nahe sie einem Hindernis sind. Es kann hier also jede Zelle als Knoten angesehen werden. Zudem ist die Größe jeder Zelle bekannt, die sie in der Realität hat.

[10] Navigation im ROS Wiki: `https://wiki.ros.org/navigation`
[11] global_planner im ROS Wiki: `https://wiki.ros.org/global_planner?distro=melodic`

Bevor der Dijkstra Algorithmus ausgeführt wird, wird ein Potentialfeld generiert. Ausgehend von der Ziel-Position werden für jede Zelle die Entfernung dorthin berechnet. Dies muss auch nur einmal ausgeführt werden, wenn die Ziel-Pose festgelegt wird. Denn auch wenn der Pfad sich ändert, die Entfernungen der Zellen zum Ziel bleiben gleich. So hat die Zelle, in dem das Ziel ist, den Wert 0, weil null Schritte zum Ziel absolviert werden müssen. Durch die Costmap werden Hindernissen erhöhte Kosten zugeschrieben.[9, S. 281ff] Die Kosten werden auf die Kosten der einzelnen Zellen addiert und sind durch den Faktor *cost_factor* im *globale_planner* festgelegt. Kosten können als Aufwand betrachtet werden, höhere Kosten machen den Pfadverlauf weniger wahrscheinlich.

Lokale Trajektorienberechnung: Dynamic window approach

Innerhalb einer lokalen Costmap wird, zusätzlich zur globalen, eine lokale Trajektorie berechnet um korrekt auf dynamische Hindernisse reagieren zu können. In dieser Arbeit wurde dafür der *Dynamic Window Approach* verwendet. Dieser wird folgend beschrieben, basierend auf dem Paper von Dieter Fox, Wolfram Burgard und Sebastian Thrun[6].
Der Vorteil gegenüber bestehenden Lösungen ist die höhere Effizienz, durch das Verkleinern des Suchraumes: dem dynamischen Fenster. Dies geschieht durch die Verwendung des Dynamikmodells des Fahrzeuges, welche hier in einer YAML Ain't Markup Language (YAML)-Datei festgelegt wurden. Die Trajektorie besteht dabei ausschließlich aus zirkulären Trajektorien. Der Suchraum ist ein zwei dimensionaler der Raum, dessen Größe durch die lokale Costmap festgelegt wird. Abhängig von den Parametern der Beschleunigungen sowie der Geschwindigkeiten, die in der *base_local_planner_params.yaml* Datei festgelegt werden, wird er Suchraum dann eingeschränkt. Die Trajektorie wird dabei so geplant, dass der Roboter in jedem Fall innerhalb des Dynamic Windows zu einem Halt kommen kann.
Der Algorithmus funktioniert wie folgt:[12]

1. Der Suchraum wird ausgehend von der aktuellen Geschwindigkeit festgelegt.
2. Für verschiedene Geschwindigkeiten (v, ω) wird die Vorwärtskinematik berechnet.
3. Jede berechnete Trajektorie wird bewertet, dabei werden unmögliche Trajektorien verworfen.
4. Die am besten bewertete Trajektorie wird ausgewählt und an den Motorcontroller weitergeleitet.

[12]DWA Local Planner im ROS Wiki: `https://wiki.ros.org/dwa_local_planner`

5. Der Suchraum und die berechneten Trajektorien werden verworfen und es wird von Vorne begonnen

Das Geschwindigkeitstupel (v, ω) entspricht den linearen und rotatorischen Sample-Geschwindigkeiten des Roboters.
Die beste Trajektorie wird anhand von verschiedenen Kriterien ausgewählt. Die Kriterien sind [6, S. 9]:

- Wie viel Fortschritt erzielt die Trajektorie hin zum globalen Ziel? (näher ist besser)
- Wie groß ist das Distanz des am nächsten gelegenen Hindernis? (mehr Abstand ist besser)
- Wie hoch ist die Geschwindigkeit des Roboters? (schneller ist besser)

Unmögliche Trajektorien zeichnen sich durch eine voraussichtliche Kollision aus.

Costmaps

In ROS wird das Konzept der Costmaps verwendet. Genauer handelt es sich hier um *costmap_2d*[13] Package, also 2-dimensionalen Karten. Es wird hier zwischen lokalen und globalen Costmaps unterschieden. Diese unterscheiden sich allerdings nur in ihrer Bedeutung und Verwendung. Eine Lokale Costmap wird benutzt um dynamische Hindernisse um den den Roboter herum zu erfassen, globale um die gesamte globale Karte zu definieren. Der technische Aufbau ist jedoch der Gleiche. Das Package berechnet anhand von LaserScan Daten oder einer Occupancy Grid Map (vgl. Abschnitt 2.4.3) eine Occupancy Grid Map, welches die Kosten für eine Navigation enthält. Hauptbestandteil des Package ist das Aufblähen (*engl. Inflation*) der Hindernisse im Raum, damit eine sichere und zuverlässige Navigation realisiert werden kann. Das Aufblähen wird benötigt, damit der Roboter beim Navigieren nicht an einem Hindernis hängen bleibt bzw. dagegen fährt und eventuell Objekten oder Lebewesen schadet. In der entsprechenden Konfigurationsdatei *costmap_common_params.yaml* kann der sogenannte *inflation_radius* in Metern angegeben werden. Dieser Wert legt im inflation_radius ausgehend von der Zelle die Kosten für die Navigation absteigend fest. Die einheitslose Skala ist definiert im Bereich $[0, 255]$. Das bedeutet, dass im freien Raum keine Kosten (also Zellwert 0) gelten und in einem Hindernis, wo eine Kollision offensichtlich ist, 100% Kosten bestehen (also 255). Zwischenwerte bedeuten eine eventuelle Kollision, kann also nicht ausgeschlossen werden.

[13] costmap_2d im ROS Wiki: `https://wiki.ros.org/costmap_2d`

3 Technische Umsetzung

Zunächst wird sich in diesem Kapitel mit der Hardware des Scoomatic Prototypen beschäftigt, wie dieser aufgebaut ist und welche Interaktionsmöglichkeiten er bietet. Danach wird die softwaretechnische Struktur beschrieben und die Inhalte der einzelnen Softwarepakete. Dabei wird auf die ROS spezifischen Tools und Software eingegangen, die verwendet und die Modelle, die damit erzeugt wurden. Zuletzt folgt eine Beschreibung der Prozesse des Kartierens und der Navigation innerhalb von Räumen. Dabei wird darauf eingegangen wie diese Prozesse die benötigten Daten verwenden und weiterverarbeiten. Daraus folgt dann schlussendlich die gewünschte Hinderniserkennung und -vermeidung.

3.1 Hardwarebeschreibung und Steuerung

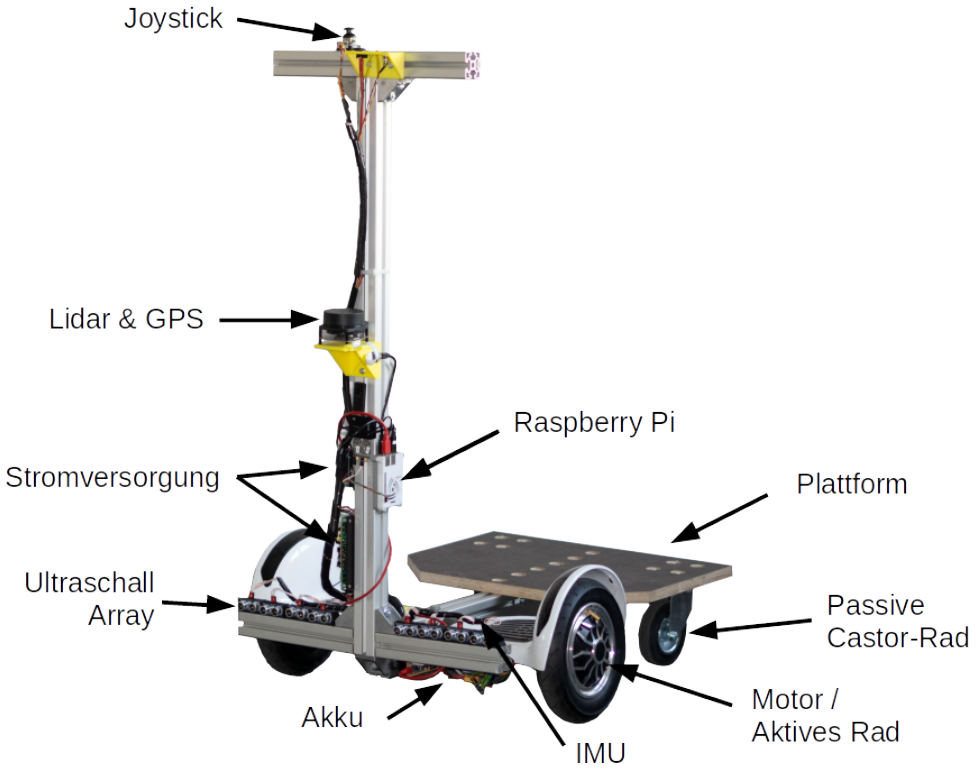

Abbildung 3.1: Der Hardware Prototyp des Scoomatic Roboters

Für diese Arbeit wurde aufbauend auf einer bestehende Hardware, die maßgeblich von Martin Schoerner aufgebaut wurde, die Software entwickelt. Der verwendete Scoomatic Prototyp ist in Abbildung 3.1 zu sehen. Er ist mit den in 3.1.1 beschriebenen Sensoren ausgestattet.
Daneben existieren noch zwei Motoren, welche Teil eines umgebauten Hoverboards sind. Das Hoverboard wurde so angepasst, dass es an die verwendeten Rexroth Aluminium Profile angebracht werden konnte. Die Aluminium Profile stellen die Basis-Struktur des Fahrzeuges her. Hinter dem Hoverboard ist eine starr verbundene Plattform angebracht, die mit den zwei Castor-Rädern ausgestattet ist.
An dem Haupt-Aluminium-Profil ist insbesondere der Raspberry Pi und der Lidar angebracht. Letzterer ist höhen-variabel, und bisher auf etwa 1m Höhe, befestigt. Diese Anordnung wird später noch in Abschnitt 4.1.2 diskutiert.
Es existieren 8 Ultraschallsensoren, welche in etwa 20cm oberhalb des Bodens befestigt sind. Sie wurden hier nicht verwendet, könnten jedoch in Zukunft ein zusätzliche Datenquelle darstellen.

Wohlgemerkt ändert sich die Höhe des Fahrzeuges auch mit dem Luftdruck der beiden angetriebenen Räder, die am Hoverboard angebracht sind. Dadurch können Fehler bzw. unvorhergesehene Probleme hervorgerufen werden. Zum Einen wird der Schlupf der Räder dadurch geändert, also stimmt die gemessene Geschwindigkeit nicht mehr mit der tatsächlichen überein. Zum Anderen kann es bei unterschiedlichen Drücken der zwei angetriebenen Rädern zu einer ungewollten bogenförmiger Trajektorie kommen. Dieses Verhalten verschlechtert unter anderem die Lokalisierung, weil der erwartete Zustand für AMCL nicht eintritt. Somit wird die Unsicherheit der Pose größer und damit kann sich der Fehler fortpflanzen. Es kann schließlich dazu führen, dass der Roboter nicht mehr zum Ziel findet. Diese Problematik gilt es zu vermeiden.
Außerdem ist hier zu beachten, dass der Motor keine echte Odometrie zur Verfügung stellt. Mehr dazu in Abschnitt 2.3.3.

3.1.1 Verfügbare Sensorik

Aktuell stehen folgende Sensoren zur Verfügung:

- 2D LIDAR: RPLidar A1 (bis zu 10 U/s, bis zu 8000 Samples)
- 8 Ultraschall Sensoren: HC-SR04 (2cm bis 3m & bis zu 50 Samples/s)
- IMU: MPU9250 (9DOF)
- GPS: Ublox SAM-M8Q

Der 2D Lidar stellt hier den bedeutsamsten Sensor dar. Denn für die Lokalisierung und Navigation ist dieser zwingend notwendig. So können die Distanzen vom Roboter aus rund herum gemessen werden. Dies erfolgt allerdings nur horizontal auf der Höhe des Lidar. Es wurde ein RPLidar A1 verwendet, der eine maximale Reichweite von 12m erreicht. Es existieren 3 Modi zur Auswahl, die sich in Reichweite und Sampling-Rate unterscheiden[18, S. 12]. Hier wurde der Boost Modus gewählt. Er hat die maximal möglichen Samples pro Sekunde (SpS), also 8000 Messwerte bei 5,5 Umdrehungen pro Sekunde des Lasers. Das entspricht $8000SpS/5,5Hz \approx 1455SpU$ (*SpU entsprechend Samples pro Umdrehung*). Mit dieser Sample Rate ist also im Vergleich zu anderen Modi eine genauere Messung der Raumgeometrie möglich, weil kleinere Hindernisse besser erkannt werden können und Strukturen besser aufgelöst werden.
In dieser Arbeit wurden von den 4 möglichen Sensoren nur der Lidar verwendet. Die IMU konnte nicht verwendet werden, da Störungen auftraten. Diese Problematik der IMU wird in Abschnitt 4.3.1 erklärt.

Das GPS Modul wurde vorerst nicht verwendet, weil zunächst nur eine Funktion im Innenbereich bereitgestellt werden sollte. Im Innenbereich ist das GPS jedoch nicht richtig funktionsfähig, da die Signale der GPS Satelliten schlecht empfangen werden können. Zudem besteht im Innenbereich auch keine Notwendigkeit GPS Informationen zu verwenden, weil eine Globale Positionsbestimmung keinen Mehrwert besitzt.

3.1.2 Manuelle Steuerung

Neben der autonomen Steuerung existieren noch die folgenden Eingabequellen. Diese dienen der manuellen Steuerung des Roboters: Ein **Gamepad** (EasySMX 2.4Ghz Controller), das sich kabellos mit dem Raspberry Pi über einen USB-Empfänger verbindet. Sowie ein **Joystick Modul** (KY-023), das über eine serielle Schnittstelle am Raspberry Pi angeschlossen ist. Das Joystick ist fest verbaut.

Abbildung 3.2: Das verwendete Gamepad zur manuellen Steuerung

Das verwendete Gamepad, zu sehen in Abbildung 3.2, wurde mithilfe der Library *inputs*[1] verwendet. Diese bietet die Möglichkeit in Python die Eingaben der Joysticks und Buttons auf dem Gamepad durch einfache Art und Weise in Python weiterzuverarbeiten. Dabei ist es irrelevant in welcher Form das Gamepad angeschlossen ist, ob kabellos oder kabelgebunden. Somit wird das Austauschen eines Gamepad deutlich vereinfacht, weil keine oder nur wenige spezifischen Änderungen am Code vorgenommen werden müssen.
Mit dem programmierten Treiber werden die Steuersignale dann in ein ROS Message kompatibles Format, einer Twist Message, umgewandelt. Eine Twist Message enthält, Linear- und Dreh-Geschwindigkeiten im 3-dimensionalen Raum.
Genauso sendet der Joystick Twist Messages. Zusätzlich dazu aber auch noch eine Bool Message, die die Information enthält ob der Joystick gedrückt wurde. Der Treiber wurde mithilfe von *pySerial*[2] realisiert. Die Library hilft die serielle Schnittstelle auszulesen.

[1]Website der inputs Bibliothek: `https://pypi.org/project/inputs/`
[2]Dokumentation von pySerial: `https://pyserial.readthedocs.io/en/latest/`

3.2 Netzwerkinfrastruktur

Der Roboter benötigt eine Netzwerkverbindung. Die Steuerung ist so über eine grafische Oberfläche möglich und die Verarbeitung findet teilweise auf einem externen Rechner statt. Da der Raspberry Pi keine integrierte grafische Anzeige besitzt, kann der Zugriff entweder über den HDMI-Anschluss und einem Display erfolgen oder per Netzwerkzugriff. Der HDMI-Anschluss wird jedoch nur für das Debugging verwendet und in dem Fall, wenn keine Netzwerkverbindung erfolgreich aufgebaut werden kann.

3.2.1 Verwendete Netzwerktechnik

Auf dem Raspberry Pi ist ein WiFi Modul Broadcom BCM43143 verbaut. Das Modul unterstützt die WLAN Standards IEEE802.11b/g/n bei 2,4 GHz. Das Modul wird verwendet um sich entweder mit einem WLAN Access Point eines TP-Link Routers oder WLAN Hotspot des externen Rechners unter Ubuntu zu verbinden.
Als externer Router wurde ein TP-Link TL-WR841N[3], der die WLAN Standards IEEE802.11b/g/n unterstützt, verwendet.
Es wurden mehrere, verschiedene Netzwerke auf dem Raspberry Pi eingerichtet und können mit verschiedenen Prioritäten verwendet werden. Das bedeutet, dass wenn ein Netzwerk ausfällt oder deaktiviert wird, wird ein niedriger priorisiertes Netzwerk verwendet.
Ein Internetzugang ist optional und kann dafür verwendet werden den Code bzw. die ROS Packages zu aktualisieren. Der Code dieser Arbeit ist in einem Git-Repository verwaltet und kann dementsprechend vom Server auf den Raspberry Pi oder jeglichen anderen Rechner geladen werden. Git ist ein Versionsverwaltungsprogramm, auch Version Control System *(engl., Abk. VCS)*. Änderungen können jedoch auch vom Raspberry Pi in die andere Richtung, also zum Server und PC weitergegeben und gespeichert werden.
Zugang zu der Kommandozeile der Rechner über das Netzwerk erfolgt über Secure Shell (SSH). SSH bietet die Möglichkeit von einem Rechner einen externen, über das Netzwerk verfügbaren Rechner per Kommandozeile anzusteuern. Damit kann der Raspberry Pi, auf dem Scoomatic von einem PC gesteuert werden. Die Ansteuerung von ROS erfolgt maßgeblich von der Kommandozeile aus. Denn auch die grafischen Programme werden über die Kommandozeile gestartet.

[3]TP-Link Website zum Gerät und dessen Spezifikationen: `https://www.tp-link.com/de/home-networking/wifi-router/tl-wr841n#specifications`

3.2.2 Verwendung des Netzwerks

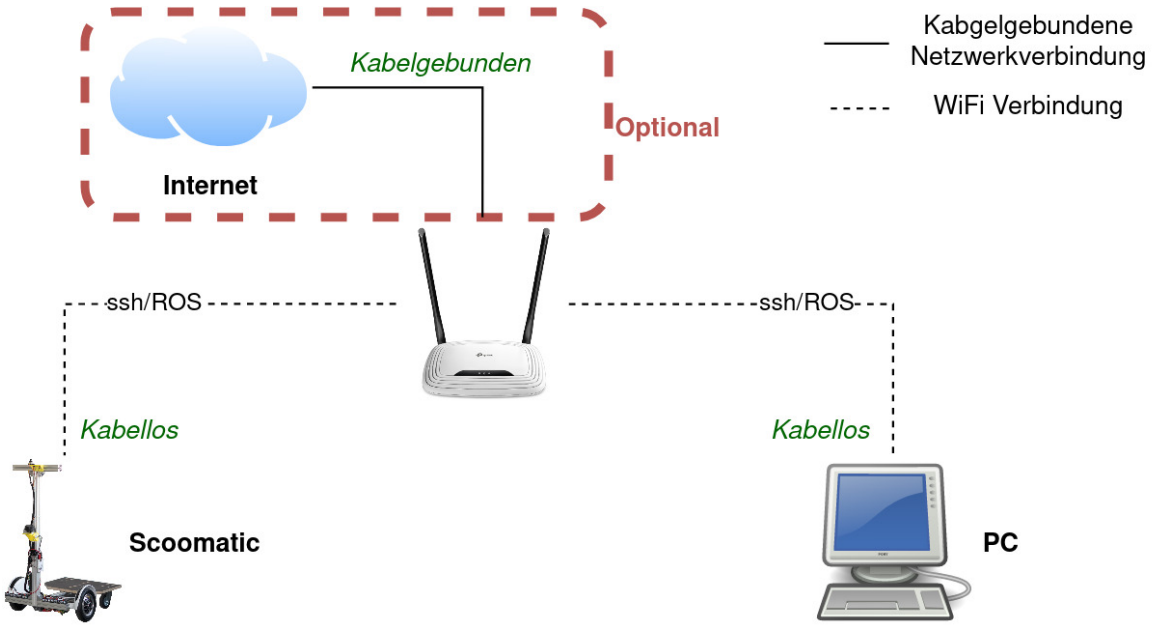

Abbildung 3.3: Netzwerkübersicht über das Gesamtsystem
Bildquelle Router:
`https://www.tp-link.com/de/home-networking/wifi-router/tl-wr841n/`

Eine Übersicht der Netzwerkinfrastruktur mit den beteiligten Geräten und verwendeten Verbindungen ist zu sehen in Abbildung 3.3.
In dieser Abbildung ist der Fall abgebildet, dass der Scoomatic und der externe Rechner sich mit dem TP-Link Router verbinden, das jedoch einer von verschiedenen Spezialfällen darstellt. Allerdings stellte diese Konstellation die Regel bei der Entwicklung dar. Der Scoomatic stellt automatisch eine Verbindung mit dem WLAN Netzwerk des TP-Link Routers her. Dasselbe geschieht mit dem externen Ubuntu Rechner. Dieser war in der Regel ebenfalls, über Kabel, mit dem Internet verbunden, was jedoch nicht notwendig für das Betreiben des Scoomatics ist. Der TP-Link Router hat eine Anbindung an das Internet über eine Kabelverbindung. Somit war es möglich bei Änderungen des Codes dieser Arbeit über Git den Code auf dem Scoomatic sofort zu aktualisieren. Zudem konnten bei Updates der ROS Pakete diese ebenfalls direkt aktualisiert werden.
Die Verwaltung und Kommandoeingabe des Scoomatics erfolgte über SSH. Es wurden SSH Konfigurationen erstellt, damit die Verbindung vom Ubuntu Rechner einfach per:

```
ssh scoomatic
```

erfolgen kann. Nun können Git-Befehle, aber insbesondere ROS Kommandos, ausgeführt werden. SSH emuliert also die Kommandozeile des externen Scommatics auf den Ubuntu

Rechner. Nachdem ROS gestartet wurde, können aufgrund der Netzwerkverbindung, praktisch alle ROS Befehle auch auf der lokalen Kommandozeile auf dem Ubuntu Rechner ausgeführt werden, ohne SSH. Dies ist aber nur möglich, weil die Programme zuvor installiert wurden. Vor allem für grafische Programme ist das Vorteilhaft, weil die SSH Verbindung dafür nicht geeignet ist.
ROS verwendet eigene Netzwerkprotokollen.

3.2.3 ROS Netzwerkprotokolle

XML-RPC wird dafür genutzt die Verbindungen zwischen Nodes zu initialisieren, aber dient nicht dem Austausch von Sensor, Navigations, usw. Daten. XML-RPC ist ein Zustandsloses Protokoll, das bedeutet, es werden keine Daten zur Identifizierung der Sitzung oder der Benutzenden bei der Anfrage mitgeschickt. Nodes registrieren mit XML-RPC ihre Publisher und Subscriber, genauso wie der Parameter Server damit angesprochen werden kann.[4]
XML-RPC setzt sich aus *XML* und *RPC* zusammen. Die Extensible Markup Language (XML) ist eine Auszeichnungssprache zur Darstellung von strukturierten Daten[5]. RPC steht für *Remote Procedure Call* und ist eine Interprozesskommunikation. Sie dient dem Aufruf von Funktionen eines Programms, in einem fremden Adressraum[6]. XML-RPC ist über eine Serveradresse und einen Port ansteuerbar. Aus diesen beiden Technologien ist dann XML-RPC entstanden[7]
Wenn eine Verbindung aufgebaut wurde muss noch das Transport Protokoll vereinbart werden, was ebenfalls mit XML-RPC passiert. Dabei stehen *TCPROS*[8] oder *UDPROS*[9] zur Auswahl, die wie der Name schon beinhaltet, auf Transmission Control Protocol (TCP) bzw. User Datagram Protocol (UDP) basieren.

3.3 Softwarestruktur

Die Software ist maßgeblich durch die ROS Package Struktur festgelegt. Diese Struktur soll aus verschiedenen Gesichtspunkten in diesem Abschnitte beleuchtet werden.
Die Software ist in 3 ROS Packages aufgeteilt. So ist Modularität, Erweiterbarkeit und Anpassbarkeit besser gewährleistet.

[4]Beschreibung der Netzwerkprotokollen in ROS: `https://wiki.ros.org/ROS/TechnicalOverview`
[5]W3C Dokumentation von XML: `https://www.w3.org/TR/2008/REC-xml-20081126/#sec-intro`
[6]IETF RFC 5531 welcher RPC beschreibt: `https://tools.ietf.org/html/rfc5531`
[7]Offzielle Website von XML-RPC: `http://xmlrpc.com/spec.md`
[8]TCPROS Beschreibung im ROS Wiki: `https://wiki.ros.org/ROS/TCPROS`
[9]UDPROS Beschreibung im ROS Wiki: `https://wiki.ros.org/ROS/UDPROS`

Intern werden die Packages wie folgt genannt: *scoomatic_ros1*, *scoomatic_drive* und *scoomatic_description*. Diese werden nun vorgestellt und ausgeführt.

3.3.1 scoomatic_ros1 Paket

Historisch bedingt sind im ROS Package *scoomatic_ros1* alle Packages, die die Sensordaten in ein ROS kompatibles Format umwandeln. Für die IMU und den RPLidar existieren bereits fertige Lösungen, die nur benutzt werden müssen. Für die anderen Sensoren wurden eigene Programme geschrieben.
Enthalten sind:

- **rplidar** (externe Library für RPLidar von ROBOPEAK[10])
- **mpu_9250** (externe IMU Library): Veröffentlicht die Gyroskop-, Beschleunigungssensor- und Kompass- als IMU Messages[11]
- **OdomPublisher**: Verarbeitet die Sensordaten des von MotorDiag zu einer Odometry Message
- **MotorDriver**: Ansteuerung des Motors mit Twist Messages
- **MotorDiag**: Stellt verschiedene Diagnosedaten zur Verfügung, unter anderem Batterie Spannung, Motor Spannung und Temperatur des Hoverboards
- **GamepadDriver**: Erzeugt Twist Messages für Motor
- **JoyDriver**: Kann wie GamepadDriver Twist Messages erzeugen
- **SonarDriver**: Noch nicht funktionierende PointCloud Messages, für zukünftige Integration in RViz bzw. SLAM Algorithmus

Twist Messages enthalten Linear- und Dreh-Geschwindigkeiten im 3-dimensionalen Raum. Der MotorDriver konsumiert genau dieses Format und verwendet diese Befehle als Steuersignale für die beiden Motoren.

[10] RPLidar Treiber im ROS Wiki: `https://wiki.ros.org/rplidar`
[11] IMU Message Dokumentation:`https://docs.ros.org/melodic/api/sensor_msgs/html/msg/Imu.html`

3.3.2 scoomatic_drive Paket

Das *scoomatic_drive* Package enthält keinen Programmcode sondern nur Konfigurationsdateien und Launchfiles. Mit diesen Dateien werden HectorSLAM, AMCL und die Navigation korrekt eingestellt und bieten die Möglichkeit die Parameter zu verbessern bzw. zu ändern.

3.3.3 scoomatic_description Paket

Das Package scoomatic_description enthält die 3-dimensionale visuelle bzw. kollisions Beschreibung des Scoomatics durch ein Unified Robot Description Format (URDF) Modell. Dieses wird in Abschnitt 3.4.1 beschrieben. Das Package besteht aus der URDF-Datei, die den Roboter beschreibt und einem Launchfile.
Das Launchfile startet einen *joint_state_publisher* sowie *robot_state_publisher*. Dabei liest der *robot_state_publisher* die XML-Datei ein und veröffentlichen die Joints, also die Verbindungen zwischen den Einzelnen Roboter-Teilen. Der *joint_state_publisher* veröffentlicht dabei nur die beweglichen Joints (Gelenke). Derzeit bestehen derartige jedoch noch nicht. Der *joint_state_publisher* besteht also für zukünftige Erweiterungen des Roboters, die noch erfolgen werden.
Der *robot_state_publisher* veröffentlicht die fixen Gelenke, also alle starren, nicht beweglichen. Das ist derzeit bei allen Gelenken der Fall.
Das 3D Modell kann dann bspw. in RViz empfangen und angezeigt werden.
Die Joints werden zu Transform Library (TF) veröffentlicht, das in Abschnitt 3.4.1 eingeführt wird. Somit ist zu allen Zeitpunkten immer bekannt wo sich welche Roboterteile, und insbesondere der Roboter selbst, im Raum befinden.

3.4 Softwarebeschreibung

In diesem Abschnitt werden die verwendeten Tools, Software und Software-Bibliotheken vorgestellt.

3.4.1 Daten- und Austauschformate

Innerhalb des ROS Systems bestehen theoretisch unbegrenzte Vielfalt von Austauschformaten von Daten. Es können eigene Messagetypen entworfen werden, die wiederrum aus mehreren, anderen Messagetypen bestehen. Hier wurde allerdings nur Standard-Message-Typen verwendet, die im ROS Package *common_msgs* definiert sind.

TF (transform library)

In ROS existiert ein System zum speichern aller Koordinatensystem und dessen Transformationen. Das System heißt TF und ist fester Bestandteil von ROS.

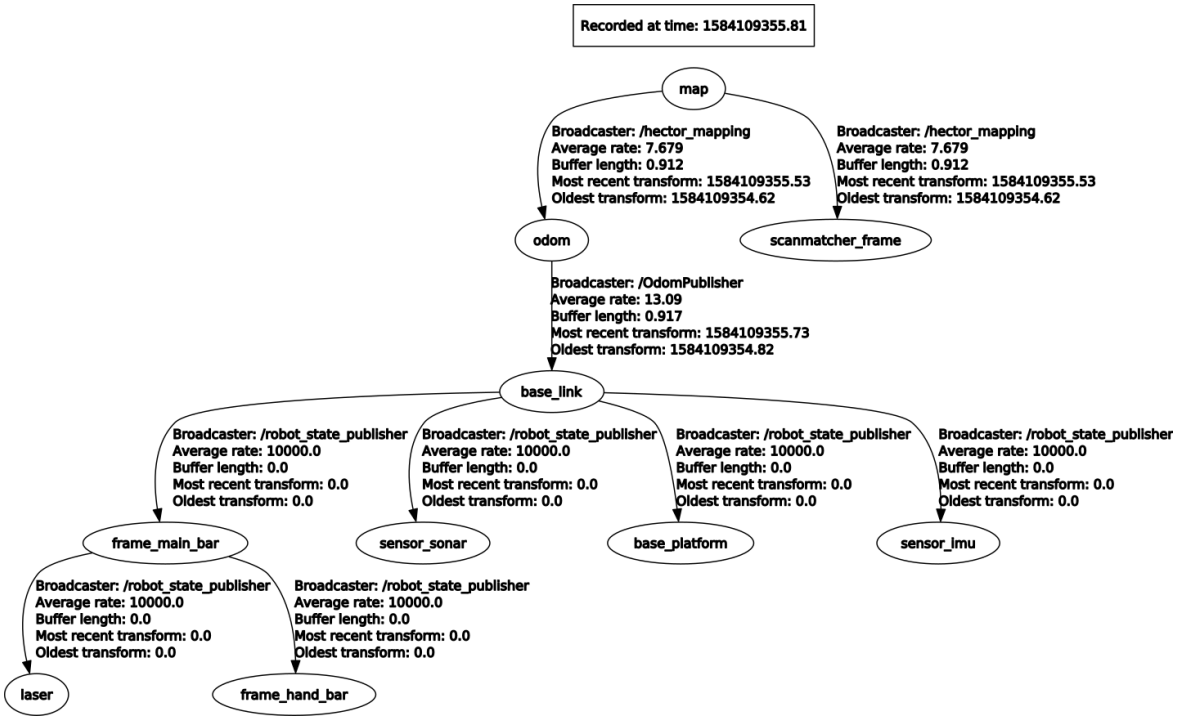

Abbildung 3.4: TF Frame-Baumstruktur während des SLAM

TF ist das Bindeglied zwischen allen Teilen des Roboters und seiner Umwelt.Es ist dafür gedacht die Transformationen zwischen der Weltumgebung, dem Roboter oder Robotern und dessen Teile über die Zeit zu verwalten. Dabei hat jedes Element ein eigenes Koordinatensystem. Die Welt hat dabei ein fixes Koordinatensystem, alle

anderen ein bewegliches. In Abbildung 3.4 sind die sogenannten Frames des gesamten Systems abgebildet. Die Abbildung wurde mit Rqt erstellt (siehe auch Abschnitt 3.4.4). Die Daten der TF-Frames werden in einer Baum-Struktur gehalten, was bedeutet, dass jeder TF-Frame nur ein Elter, also Elternteil, hat.[5]
Die Frames *map*, *odom* sowie *base_link* sind eine speziell festgelegte Frames und haben besondere Bedeutung. Sie wurden in REP 105 festgelegt.
ROS Enhancement Proposal (REP) enthält technische Spezifikationen und Standards von ROS[15, Abs. 1, What is a REP?]. REP 105 beschreibt die Frames map, odom und base_link[16, Abs. 3, Specifications / Abs. 1 Coordinate Frames]. Danach ist der map Frame ein "world fixed frame". Er dient als Welt-Referenz. Der odom Frame ist im Vergleich zu map zwar auch ein world fixed frame, dieser aber enthält keine diskreten Sprünge im Raum, die Referenzierung ist immer kontinuierlich.
Der base_link TF-Frame ist der Basis-Frame des Roboters und ist in der Rotationsachse des Scoomatics, also mittig zwischen den beiden Motoren festgelegt. Dies gilt aber nur hier und ist nicht Standard in ROS. Ausgehend von diesem Frame werden alle anderen, roboterbezogenen Frames festgelegt. Der *frame_main_bar* Frame steht für das Haupt-Aluminium-Profil des Scoomatics, an dem die Sensoren befestigt sind. Am *sensor_sonar* Frame wurden die Ultraschallsensoren befestigt. Der *base_platform* Frame ist die erweiterte Plattform zum darauf-stehen. Der *sensor_imu* Frame ist wie in der Realität in der Nähe der Rotationsachse befestigt. Daneben gibt es noch den *laser* Frame, der den Lidar Sensor darstellt, der ebenso wie der *frame_hand_bar* am frame_main_bar Frame befestigt ist.
Eine Besonderheit bei HectorSLAM im Vergleich zur Navigation ist der *scanmatcher_frame*. Er wurde nicht verwendet, wird aber von HectorSLAM bereitgestellt. Dieser entfällt bei der Navigation. Bei der Navigation wird zudem der map Frame von AMCL bereitgestellt, weil es die Beziehung map→odom bereitstellt.
Des Weiteren ist in der Abbildung 3.4 die *Average rate*, also durchschnittliche Veröffentlichungsrate zu sehen. Diese gibt an, mit welcher Häufigkeit in Hz die Daten veröffentlicht werden. Diese Rate kann bei Bedarf eingestellt werden. Bei HectorSLAM und AMCL ist diese vor allem von der Daten-Häufigkeit der Lidardaten abhängig. Die Zeiten, welche angezeigt werden, sind Zeitstempel und im Unixzeit Format, also den Sekunden seit der sogenannten Epoche[12].
Der Bufferlänge gibt die Zeit in Sekunden an, wie lange TF vergangene Daten vorhält, die noch abgefragt werden können. Ansonsten wird noch der Zeitstempel der ältesten und jüngsten Transformation angegeben.

[12]Time Klasse Dokumentation von rospy

URDF Modell

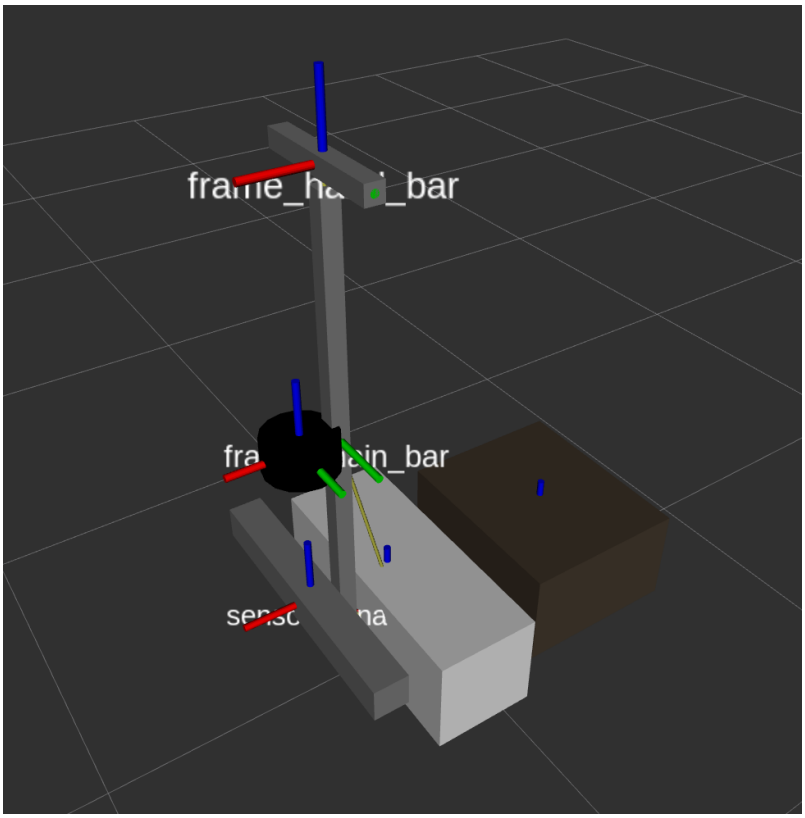

Abbildung 3.5: URDF Scoomatic Modell mit TF Koordinatensystemen

In Abbildung 3.5 ist das URDF Modell des Scoomatics mit den Koordinatensystemen der einzelnen Roboterteilen zusehen. Die rote Achse ist die x-Achse, die grüne Achse ist die Y-Achse und die blaue Achse entspricht der Z-Achse[13]. Die Mitte der einzelnen Boxen entsprechen den gerade besprochenen TF Koordinatensysteme. Aus dem URDF-Modell folgen also direkt die TF-Frames, wie in Abbildung 3.4 zu sehen.
Das URDF Modell ist nicht notwendigerweise Maßstabsgetreu. Es dient lediglich einer ansprechenden Darstellung des Modells in RViz und einer eventuellen zukünftigen Kollisionsverarbeitung. Außerdem kann es das Debugging erleichtern. Da die Navigation bzw. SLAM nur 2-dimensional erfolgt ist eine 3D Beschreibung des Roboters nicht zwingend für eine Kollisionsüberprüfung notwendig.

[13]Beschreibung der Achsen in RViz siehe: `https://wiki.ros.org/rviz/DisplayTypes/TF`

OccupancyGrids als PGM und YAML Dateien

Portable Graymap (PGM) ist eine Rastergrafik, die Grauwerte speichern kann. Sie entspricht damit einem OccupancyGrid, das in den Zellen nur zwischen Frei und Hindernis unterscheidet und liegt als Datei vor. Wenn mit der *map_server* Node eine Karte gespeichert wird, werden eine PGM Datei, welche die Bildaten enthält sowie eine YAML Datei, welche Metadaten enthält, gespeichert. Die PGM Datei kann also, wie ein OccupancyGrid, als eine Belegungsmatrix gesehen werden, welche die Wahrscheinlichkeit für ein Hindernis darstellt. Die YAML Datei legt den Mittelpunkt der Karte fest, wo der Roboter bei der Initialisierung gesetzt wird. Außerdem wird der Schwellwert für das Eintragen als Hindernis bzw. als freien Raum festgelegt. Als Beispiel soll das Eintragen von Hindernissen dienen. Der Standardwert für das Eintragen von Hindernissen wird im Parameter *occupied_thresh* festgelegt und ist 0.65. Das bedeutet, dass bei einer Wahrscheinlichkeit von 0.65 des im OccupancyGrid die Zelle als belegt gekennzeichnet wird.
Zudem werden weitere Metadaten wie bspw. die Auflösung der Karte festgelegt. Diese sind PGM ist Teil der Netpbm Bibliothek und wurde dort definiert.[14]

Odometrie

Für AMCL und die Navigation werden Odometriedaten benötigt. In HectorSLAM wurden diese ebenfalls verwendet, dies ist allerdings optional.
Das Hoverboard stellt über die serielle Schnittstelle Debug-Informationen zur Verfügung, die hier als Odometrie Alternative verwendet wurden. Es stand kein Wheel-Encoder oder ähnliches Instrument zur Verfügung, das zum genauen Messen der Umdrehungen, Geschwindigkeiten oder Beschleunigungen verwendet werden konnte.
In den Debug-Informationen gibt es lediglich ein Datenteil, der die Soll-Leistung der Motoren zurückgibt, also keine echten Odometriedaten. Es werden ausschließlich Werte über einem gewissen Schwellwert verwendet. Dieser Wert wurde experimentell herausgefunden.
Dieser Umstand ist notwendig, weil der Roboter sich bei geringen Werten nicht bewegt, trotz dessen, dass am Motor eine Spannung anliegt. Bei höheren Geschwindigkeiten hat sich herausgestellt, dass die Werte für die Lokalisierung und Navigation nutzbar sind.
Die Geschwindigkeiten werden wie in Abschnitt 2.3 berechnet.
Die berechnete Pseudo-Odometrie steht dann in der Topic */odom* zur Verfügung.

[14]Definition von PGM: `http://netpbm.sourceforge.net/doc/pgm.html`

3.4.2 Verwendete Tools

Die folgenden Abschnitte erläutern die verwendeten Tools und die verwendete Programmiersprache.

3.4.3 Python Programmierung

Der für diese Arbeit entwickelte Code wurde mit Python 2.7 geschrieben. Auch wenn Python 2.7 in Zukunft nicht mehr unterstützt bzw. weiterentwickelt wird[15]. Grund dafür ist, dass ROS Melodic bzw. die Abhängigkeiten von rospy nur Python 2 unterstützen. Die Python Abhängigkeiten, einschließlich rospy, wurden über die offiziellen Ubuntu Paketquellen installiert.

Neben Python 2 wird auch C++ offiziell von ROS zur Programmierung unterstützt und bietet dafür Bibliotheken zur Nutzung an. Für die Programmierung wurde die offizielle Bibliothek *rospy*[16] für Python verwendet. Damit ist es möglich mit den von ROS zur Verfügung gestellten Funktionen, Topics, Services, Parameter, usw. zu interagieren.

ROS bietet dabei alle Standard-Message-Datentypen durch das ROS Package *common_msgs* an[17].

Daneben wurde noch pySerial für die Ansteuerung von seriell kommunizierender Hardware verwendet. Für das Gamepad wurde die Library, wie in Abschnitt 3.1.2 beschrieben, verwendet.

3.4.4 Verwendete Grafische Software

Um die Interaktionen mit dem Scoomatic zu vereinfachen und übersichtlicher zu gestalten, wurden grafische Tools verwendet. ROS stellt dabei einige verschiedene zur Verfügung. Die zwei verwendeten Tools werden im folgenden Abschnitt vorgestellt.

[15] Python 2.7.18, the last release of Python 2
[16] rospy im ROS Wiki: `https://wiki.ros.org/rospy`
[17] common_msgs im ROS Wiki: `https://wiki.ros.org/common_msgs?distro=melodic`

RViz

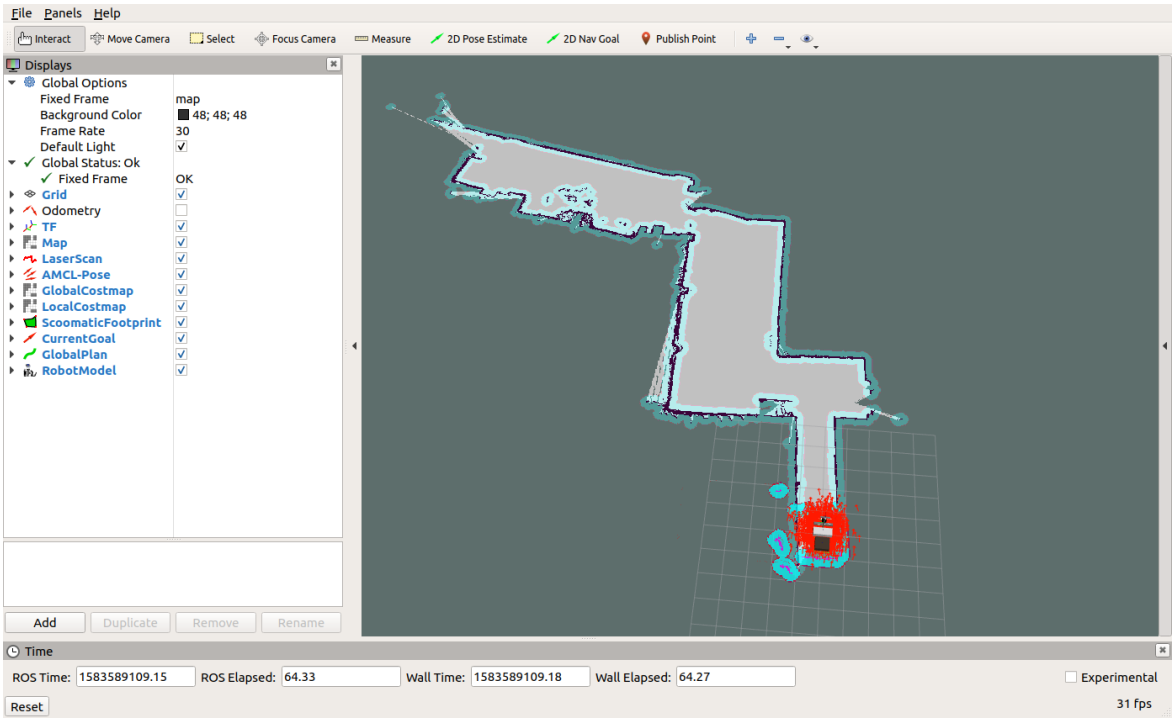

Abbildung 3.6: Beispiel von RViz mit Anzeige von Navigationsdaten

Maßgeblich für das Entwickeln und Debuggen des Systems wurde die Software RViz verwendet. RViz ist Teil des Softwarepakets von ROS. Die Software bietet die Möglichkeit die Daten für die jeweilige Disziplin, also SLAM, Lokalisierung und Navigation, und deren Erzeugnisse grafisch darstellen zu lassen.

Um die Benutzung zu vereinfachen wurden 3 verschiedene Ansichten für die Software erstellt, um die jeweils passende Ansicht für die entsprechende Disziplin bereitzustellen. Die Dateien mit Endungen *.rviz* können bei Bedarf in die Software geladen werden.

Rqt

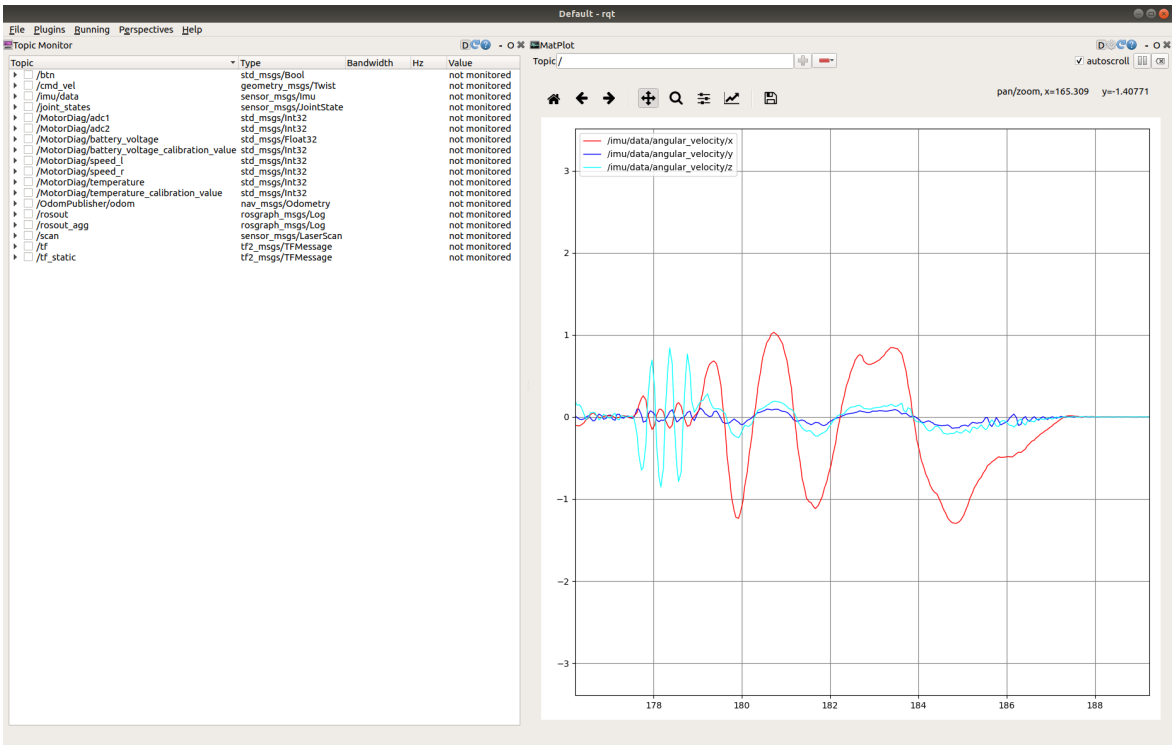

Abbildung 3.7: Beispiel von Rqt mit Anzeige von IMU Daten

Mit Rqt gibt es eine weitere Möglichkeit Daten zu visualisieren bzw. anzuzeigen. In Rqt ist es möglich jegliche Topics zu abonnieren und die Werte numerische oder grafisch anzeigen zu lassen. Dazu können noch Visualisierungen eingeschaltet werden. Eine Beispiel Ansicht mit IMU-Daten ist in Abbildung 3.7 zu sehen. Es erleichtert das Debugging des Systems. Dazu werden alle Topics, die über den ROS Master zur Verfügung stehen angezeigt. Ausgewählte Topics können dann visuell dargestellt werden. Außerdem ist es möglich verschiedene Schaubilder zur Struktur des Systems zu erstellen. Ein Beispiel dafür ist die Abbildung 3.4 oder Abbildung 3.10.

3.5 Kartierung

Die LaserScan Daten werden von der *hector_mapping* ROS Node eingelesen. Diese Daten enthalten nur Distanzen um den Roboter herum. Dabei gehen allerdings etwa 19° der nach Hinten gerichteten LaserScan Daten verloren. Das Rexroth Aluminium Profil, an dem der Lidar befestigt ist, sowie diverse Kabel versperren den Weg der Laserstrahlen.

Jedoch werden in diesem Bereich vom RPLidar keine Daten an den Raspberry Pi gesendet. Denn der angegebene Mindestabstand zu Objekten zum Messen muss mindestens 15cm sein. Das ist hier nicht der Fall. Dementsprechend werden diese Messungen verworfen.

3.5.1 Datenfluss beim SLAM Prozess

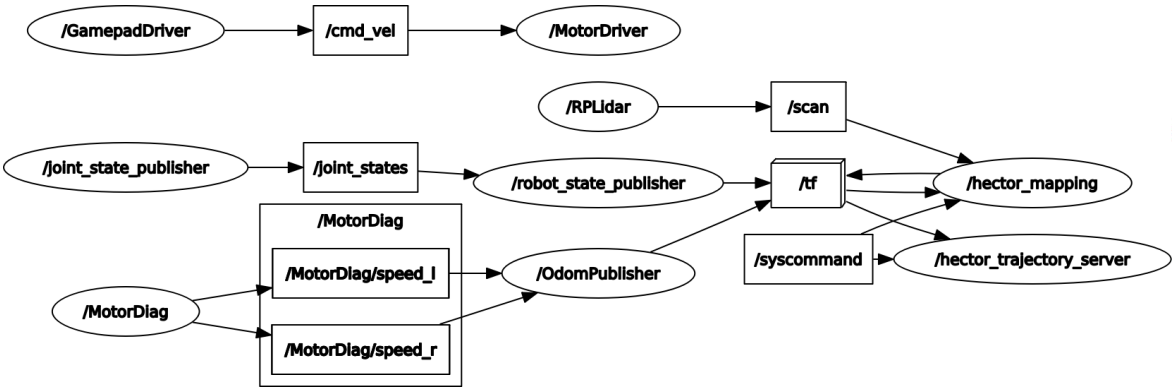

Abbildung 3.8: Datenfluss der ROS Nodes und Topics bei der Navigation
Ovale sind Nodes, Rechtecke entsprechen Topics. Pfeile zeigen auf Konsumierende Nodes. Abbildung wurde mit Rqt (Abschnitt 3.4.4) erstellt.

Die Erstellung der Karte unterscheidet sich deutlich von der Navigation und wird folgend erläutert. In Abbildung 3.8 ist für eine Verdeutlichung des Datenflusses eine Übersicht zu sehen.
Während dem SLAM Prozess wird das Fahrzeug mit dem Gamepad manuell gesteuert. Die Anweisungen werden dabei von der Node *GamepadDriver* über die Topic *cmd_vel* direkt an die Motorsteuerung geschickt.
Die Ausgangsdaten sind ebenfalls die Lidar- und Odometriedaten. Wie bei der Navigation stehen diese Daten von der Node */RPLidar* über die Topic */scan* bzw. von der Node */OdomPublisher* über TF verfügbar. Die Daten werden anschließend, wie in Abschnitt 2.4.3 beschrieben, verarbeitet. HectorSLAM veröffentlicht dann über die ROS Node *hector_mapping* eine Karte über die Topic */map*. Zusätzlich zur Karte wird, wenn der Roboter über das Gamepad oder anderweitig bewegt wurde, die Trajektorie als *Path* Message über die Node und Topic *hector_trajectory_server* veröffentlicht. Also der Pfad, den der Roboter bisher zurückgelegt hat. Dieser kann, genauso wie die Karte, in RViz betrachtet werden.
Als abschließende Aktion kann nach dem Abschließen des SLAM Prozesses manuell die erstellte Karte als Datei gespeichert werden, damit diese für die Navigation verwendet

werden kann. Dies erfolgt mit der *map_server* Node. Sie bietet den Service *map_saver*, mit dessen Hilfe es möglich ist die Karte als *OccupancyGrid* Message in einer PGM bzw. YAML Datei zu speichern, wie in Abschnitt 3.4.1 erläutert wurde. Nun ist die Karte erstellt und gespeichert. Der Prozess kann beendet werden.

3.5.2 Vergleich von ROS SLAM Packages

Tabelle 3.1: Vergleich der ROS SLAM Packages

ROS Package	hector_slam	gmapping	tiny_slam	cartographer
LaserScan	✓	✓	✓	✓
Odometrie	optional	✓	✓	✓
ROS Melodic	✓	✓	✗	✓
RGBD-Kamera	✗	✗	✗	partiell
Algorithmus	Abschnitt 2.4.3	RBPF	RBPF	Graph-SLAM
Loop Closure	✗	✓	?	✓

Rao-Blackwellized Partikel Filter (RBPF) wurde in diesem Paper eingeführt: [7]. Dies soll hier aber nicht weitergeführt werden.

Für ROS existieren verschiedene SLAM Algorithmen als Packages. Diese können unter Ubuntu über die Paketquellen heruntergeladen und installiert werden.
Es wurden vier Packages näher evaluiert. Sie wurden wie in Tabelle 3.1 zu sehen, bewertet. Dabei wurden die folgenden Kriterien verwendet:

- **LaserScan**: Können die vorhanden LaserScan Daten verwendet werden?
- **Odometrie**: Sind Odometrie Daten notwendig und wenn ja, in welcher Form?
- **ROS Melodic**: Ist das Package verfügbar für die verwendete ROS Version?
- **RGBD-Kamera**: Gibt es die Möglichkeit RGBD-Kamera Informationen zusätzlich zu den LaserScan Daten zu verwenden?
- **Algorithmus**: Welcher Algorithmus wird verwendet und kann eine Abschätzung über die Rechenkapazität für den verwendeten Rechner gemacht werden?
- **Loop Closure**: Unterstützt der Algorithmus einen aktiven Loop Closure?

Loop Closure bedeutet, dass die Karte während eines SLAM-Prozesses rückwirkend angepasst werden kann. Dies kann erfolgen, wenn vom Algorithmus erkannt wird, dass im Bereich der Position des Roboters bereits Daten aufgenommen und der Teil der Karte erzeugt wurde. So würde eventuell bei einem Kreisschluss der Roboterfahrt eine Korrektur der Karte stattfinden, wenn dies notwendig ist.
Es ist zu beachten, dass jedes Package außer *hector_slam* zwingend Odometrie benötigt.
Es wurden von anderen Teams bereits Vergleich zwischen SLAM Packages durchgeführt. Im Gegensatz zu [17] waren die Ergebnisse von HectorSLAM bei [23] deutlich schlechter wie bei cartographer und gmapping. Dies begründeten das Team Rauf Yagfarov et al. damit, dass HectorSLAM keine Odometrie Daten verwendet. Es gibt also scheinbar keine konsistente Kartenqualität von HectorSLAM auf verschiedenen Systemen. In Kapitel 4 wird aber gezeigt, dass beim Scoomatic selbst mit Standardeinstellungen gute Ergebnis erzielt wurden.
tiny_slam konnte nicht verwendet werden, weil es nicht für ROS Melodic verfügbar ist. Dies ist ein Ausschluss-Kriterium, weil die Lauffähigkeit auf dem Ist-Zustand des Systems, also ROS Melodic, notwendig ist. Abgesehen davon, konnte nicht abschließend geklärt werden ob der Algorithmus Loop Closure unterstützt.
Der Nachteil von cartographer ist die hohe Rechenleistung, die durch den Graph-SLAM Ansatz benötigt wird. Die Anforderung an das System ist jedoch, so wenig Rechenkapazität zu benötigen, wie möglich, damit diese Berechnungen mobil erfolgen können.
Die Packages hector_slam und gmapping sind sich in ihren hier dargestellten Eigenschaften sehr ähnlich. Jedoch ist der entscheidende Unterschied die Odometrie. Dies ist der ausschlaggebende Punkt, warum sich HectorSLAM entschieden wurde. Denn zum Anfang der Arbeit war nicht bekannt wie gut die Pseudo-Odometrie des Scoomatic funktioniert. Außerdem haben Vergleiche zwischen den beiden Packages gezeigt, dass mit beiden ähnlich gute Ergebnisse erzielt werden können (s.u.).

3.5.3 HectorSLAM

Es wurde HectorSLAM[18] vom Team Hector der TU Darmstadt[19] verwendet. HectorSLAM ist ein Subsystem eines Systems des Team Hector für Urban Search an Recovery (USAR), also dem städtischem Suchen und Finden von Verletzten. Den Autoren des Paper von HectorSLAM zufolge zeichnet sich das Paket durch das Benutzen von wenig Rechenkapazität aus. Dadurch sei es für leichte, energiesparende und kostengünstige

[18]HectorSLAM im ROS Wiki: `https://wiki.ros.org/hector_slam`
[19]Website des Team Hector `https://www.teamhector.de/resources`

Rechner geeignet[12, S. 1]. Dies zeigte auch ein Vergleich zu anderen 2D SLAM Algorithmen[17, S. 4-6]. Ein solcher Rechner wird in dieser Arbeit mit dem Raspberry Pi verwendet. Außerdem ist es mit der verwendeten Ubuntu bzw. ROS Version kompatibel. Darüber hinaus verfügt es zwar über keinen aktiven Loop Closure, erzielt aber ähnlich gute Ergebnisse, wie gmapping, das solch einen Loop Closure durchführt. Positiv ist zudem, dass HectorSLAM keine Odometriedaten zwingend benötigt. Weil anfangs nicht sicher gesagt werden konnte, wie zuverlässig diese Daten des Hoverboards sind, wurde diese Eigenschaft positiv bewertet.

3.6 Lokalisierung und Navigation

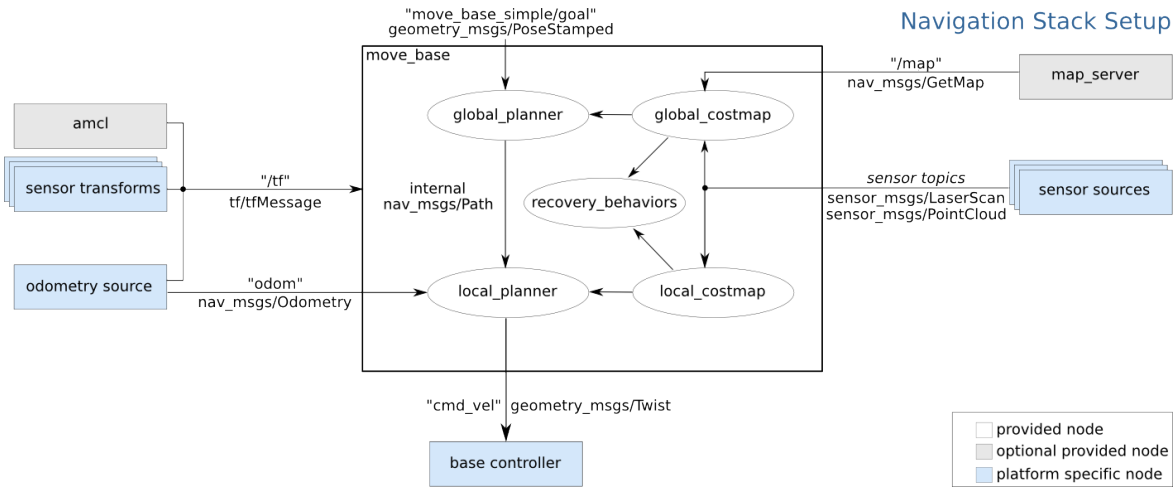

Abbildung 3.9: Übersicht über den ROS Navigation Stack

In ROS ist für die lokale Trajektorienberechnung der *base_local_planner*[20] verantwortlich, der Teil des Navigation Stack ist. Mit dem Parameter *dwa*[21] kann zwischen den Algorithmen Dynamic Window Approach (DWA) und dem *Trajectory Rollout Algorithmus* gewechselt werden. In ROS wird DWA vorgezogen, da er bei gleich guter Qualität weniger Rechenkapazität verwendet.
Zentraler Bestandteil der Navigation ist das ROS Package *move_base*. In Abbildung 3.9 ist dieser mittig als Whitebox mit den enthaltenden Nodes zu sehen. Es laufen dort alle entscheidende Daten, die zur Navigation notwendig sind zusammen. Dort findet die Erstellung der Anweisungen für die Motoren, über die Node *MotorDriver* statt (in Abb. 3.9 als *base_controller*).

[20]base_local_planner im ROS Wiki: `https://wiki.ros.org/base_local_planner?distro=melodic`
[21]Definition des *dwa* Parameters im ROS Wiki

Abbildung 3.9 ist der Abbildung 3.10 sehr ähnlich, enthält allerdings alle Nodes und Topics des gesamten Systems. Erstere ist Allgemein gehalten und bezieht sich hier nur auf den Standard Navigation Stack von ROS.

3.6.1 Datenfluss bei der Navigation

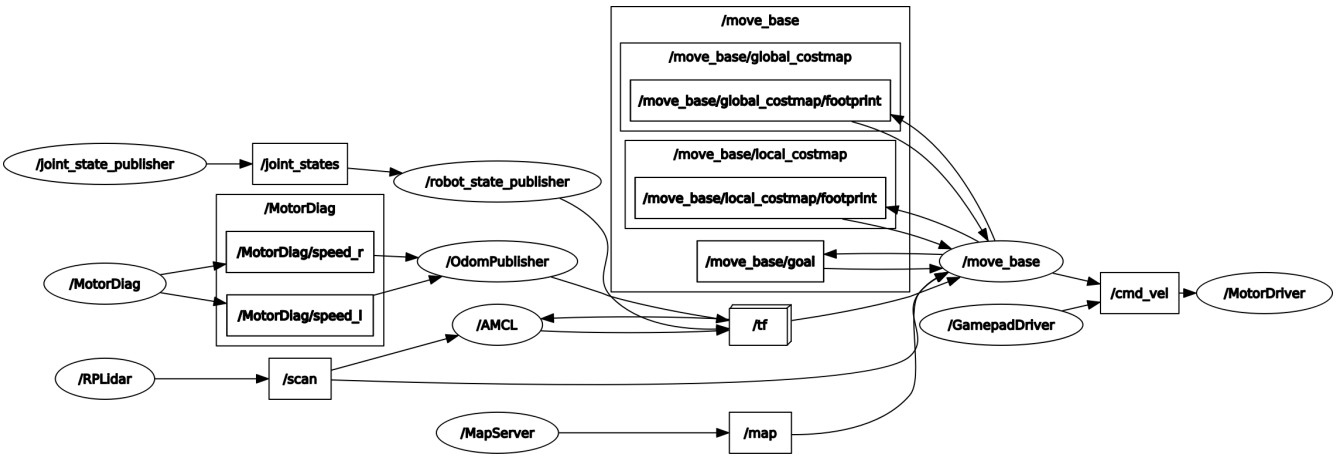

Abbildung 3.10: Datenfluss der ROS Nodes und Topics bei der Navigation
Ovale sind Nodes, Rechtecke entsprechen Topics. Pfeile zeigen auf Konsumierende Nodes. Abbildung wurde mit Rqt erstellt (Abschnitt 3.4.4)

Es wird der Datenfluss und der Ablauf des Gesamt-System *im Regelfall* erläutert. Ausgehend vom Beobachten der Sensoren, über das Verarbeiten der Daten, hin zu der Ausführung von Aktionen und der Manipulation der Realität im Raum. Also das Zusammenspiel des gesamten Systems bei der Navigation. Die Hinderniserkennung und -vermeidung ist dabei die zentrale Disziplin für eine erfolgreiches Abschließen der Navigationsaufgabe.
Der Ausgangspunkt und Basis der Datenverarbeitung ist der RPLidar A1. Er ist maßgeblich für die Erfassung der Umwelt zuständig. Er dreht sich kontinuierlich mit dem Laser im Kreis und nimmt Distanzen auf, wie in Abschnitt 2.4.4 beschrieben. Das bedeutet es liegen nun im *LaserScan* Datentyp einzelne Distanzmessungen vor, die jeweils einem Winkel zugewiesen sind. Diese Daten treffen in der Regel mit $5,5Hz$ ein. Somit wird der Raum, in dem der Lidar ist, etwa 5,5 mal in der Sekunde vollständig vermessen. Diese Messpunkte können in einem kartesischen, 2-dimensionalen Koordinatensystem aufgetragen werden.
Die unechte Odometrie der Motoren liefert einen *Odometry*[22] Messagetyp, die im Package *scoomatic_ros1* zu einer Position in diesem Koordinatensystem umgerechnet

[22]Odometry Doku:`https://docs.ros.org/melodic/api/nav_msgs/html/msg/Odometry.html`

wird. Dabei müssen Linear- und Dreh-Bewegungen gesondert behandelt werden. Somit existiert schon mal eine grobe Schätzung der Pose des Roboters. Nun gilt es diese beiden Informationsquellen sinnvoll zu fusionieren. Der Odometry Messagetyp enthält dabei die beiden Geschwindigkeiten sowie die aktuelle Pose jeweils mit einer Kovarianzmatrix. Dies erfolgt mit AMCL. Es vergleicht den aktuellen Systemzustand mit einer vorhandenen Karte und erzeugt eine Wahrscheinlichkeitsverteilung über die Pose des Roboters in dieser Karte. Unterstützend wirkt dabei die Odometrie, welche eine Aktualisierung der Pose triggert. Dazu ermöglicht es AMCL eine voraussichtliche neue Pose zu berechnen, basierend auf der aktuellen Bewegungsform des Roboters. In RViz kann nun eine Initiale Pose festgelegt werden. Diese beschleunigt den Prozess um eine möglichst genaue Lokalisierung zu erreichen. Ist diese Initiale Pose nicht bekannt oder kann nicht festgelegt werden, kann auch eine globale Lokalisierung erfolgen, die den Roboter innerhalb der gesamten Karte lokalisiert. Dies ist unter Umständen allerdings deutlich langsamer und erfordert mehr initiale Bewegung. Durch das Fahren im Raum bekommt AMCL mehr Daten und kann somit die Pose verbessern. Die Ansicht des Zustandes vor einer globalen Lokalisierung des Systems ist in Abbildung 3.11 zu sehen. Die Lokalisierung muss wohlgemerkt jedoch vor dem Starten der Navigation erfolgen.

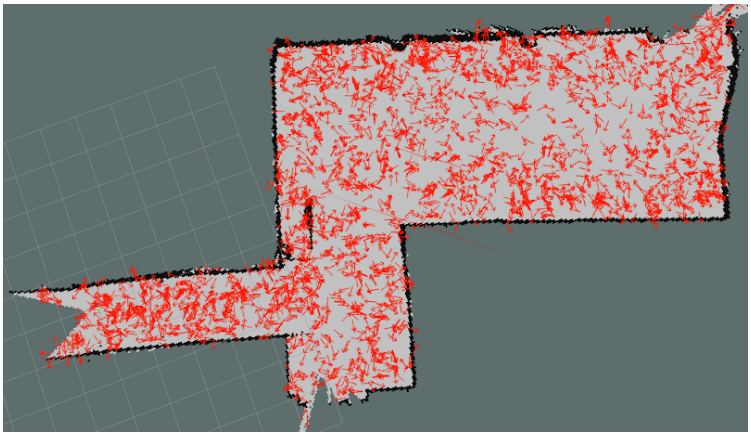

Abbildung 3.11: Darstellung des Zustandes vor der Globalen Lokalisierung in RViz
Zu sehen sind die einzelnen Partikel, als Pfeile dargestellt. Diese sind über den gesamten Raum verteilt, der frei von statischen Hindernissen ist.

Die Karte wird dabei vom *map_server*[23] über die Topic */map* veröffentlicht. Dafür benötigt dieser eine Datei auf dem Laufwerk des Raspberry Pi, die zuvor erstellt und mit dem selbigen abgespeichert wurde.
Ist eine ausreichend gute Lokalisierung erreicht, kann die Navigation gestartet werden.

[23]map_server im ROS Wiki:`https://wiki.ros.org/map_server`

Um die Navigation zu starten, benötigt das System zunächst einen Auftrag. Dieser wird in RViz gestartet, indem der Button *2D Nav Goal* betätigt wird. Dann kann eine Pose im Raum per grafischer Oberfläche und Maus festgelegt werden. Die Navigation erzeugt zunächst aus der bestehenden Karte die global_costmap. Somit wird sichergestellt, dass der Roboter nicht an einem Hindernis steckenbleibt, dieses berührt bzw. anfährt. Diese Hindernisvermeidung ist zentrales Element der Navigation. Dazu muss aber zunächst festgestellt werden, dass ein solches im aktuellen oder zukünftigen Fahrweg enthalten ist. Statische Hindernisse, wie Wände, werden durch die Karte vermieden. Dynamische Hindernisse sind zum Erstellungszeitpunkt der Karte noch nicht existent und können nur durch den Lidar erkannt werden. Dynamische Hindernisse werden also durch die lokale Costmap dargestellt und können so bei der lokalen Trajektorienberechnung per Dynamic Window Approach (DWA) (Abschnitt 2.5.2) berücksichtigt werden. Wird also eine Belegung in einer Zelle auf der Karte, die auf dem Fahrweg liegt, erkannt, wird dieser angepasst. Die lokale Costmap hat dabei immer eine feste Größe, wird dabei in das Zentrum des Roboters gelegt und wird bei der Fahrt stetig aktualisiert.

In Abbildung 3.9 ist zu sehen, dass der *global_planner* die Daten von der *global_costmap*, der *local_planner*, die Daten von der *local_costmap* bezieht. Der *local_planner* bezieht dabei den Navigationspfad (*nav_msgs/Path*) vom *global_planner*. Die global_costmap bezieht ihre Daten von der Node *map_server*, also der SLAM-Karte und vom Lidar. Die local_costmap bezieht sie vom Lidar (*sensor_msgs/LaserScan*).

Nun kann eine Ziel-Pose in RViz festgelegt werden. Der Startpunkt ist automatisch der aktuelle Standort des Robotors. Die Ziel-Pose wird über eine Maus festgelegt und enthält eine 2D-Position und eine Ausrichtung. Dies wird als *MoveBaseActionGoal*[24], das schlussendlich nur eine Identifikationsnummer, eine Pose und Zeitstempel enthält, an move_base geschickt.

Nun wird automatisch der Fahrweg berechnet. Dieser wird basierend auf der Costmap mit einem Dijkstra Algorithmus berechnet. Der berechnete Pfad wird als eigenständige Topic veröffentlicht und ist vom Typ *Path*[25]. Dieser besteht aus mehreren *PoseStamped* Daten. Also Posen mit Zeitstempel. Der Weg, der sich innerhalb der lokalen Costmap befindet, wird nochmal gesondert berechnet. Für die globale Navigation ist der *global_planner*, für die lokale Costmap bzw. Navigation der *local_planner*. Letzteres ist notwendig, damit die dynamischen Hindernisse ebenfalls korrekt behandelt werden, also in der Regel umfahren werden. Die Navigationsalgorithmen verwenden also die jeweilige Costmap, die aus einer Belegungsmatrix besteht und berechnen dann einen

[24]MoveBaseActionGoal Dokumentation
[25]Dokumentation von Path: `https://docs.ros.org/api/nav_msgs/html/msg/Path.html`

Weg anhand der einzelnen Zellen. Im Falle vom global_planner wird der kürzeste Weg berechnet, im Fall vom local_planner der Weg mit der geringsten Wahrscheinlichkeit an ein Hindernis zu stoßen und gleichzeitig am nächsten zum Ziel zu kommen.

Bei der Berechnung der Pfade wird die Grundfläche des Roboters über die Topic */footprint* verwendet, die ein Polygon[26] verwendet. Dieses Polygon besteht aus den Eckpunkten der Grundfläche des Roboters. Somit ist eine Kollisionserkennung mit Hindernissen möglich.

Wenn der Pfad bekannt ist, können einzelne Steuerbefehle an die Motoren gegeben werden. Diese bestehen aus Geschwindigkeitsanweisungen und werden schon im Zuge des lokalen Weges berechnet. Dieser besteht aus einzelnen Anweisungen und werden nun stückweise an die Motoren geleitet. Allerdings wird durch die Bewegung des Roboters die lokale Costmap erneuert. Somit wird auch bei jeder Aktualisierung der Costmap der Pfad neu berechnet und kann sich dadurch ständig ändern.

Die Motoren führen dann die Aktion aus. Dadurch wird die Pose des Roboters und dessen Teile aktualisiert, den neuen Sensordaten entsprechend. Dann ändert sich die Umgebung und der Lidar nimmt ebenso wie die Odometrie neue Daten auf und der Prozess um die Costmaps beginnt von neuem.

Dabei kann die Navigation zu jeder Zeit - also auch beim Fehlschlagen - neu gestartet werden, also die Ziel-Pose neu gesetzt werden.

Es existiert jedoch noch der nicht unwahrscheinliche Fall, dass der Roboter keinen Ausweg aus seiner aktuellen Pose findet. Es kann also der Fall sein, dass bspw. die Hindernisse zu nah sind, ohne dass der Roboter sicher gehen kann, diese nicht zu berühren. Für diese Szenarien existieren verschiedene Strategien. Hier wird sich den Recovery Behaviors bedient.

Im besten Fall erreicht der Roboter jedoch sein Ziel und beendet somit die aktuelle Navigation. Es kann nun - wenn gewünscht - eine neue Navigation ausgeführt werden.

[26] Dokumentation von Polygon: `https://docs.ros.org/api/geometry_msgs/html/msg/Polygon.html`

3.6.2 Navigationsproblemlöser: Recovery Behaviors

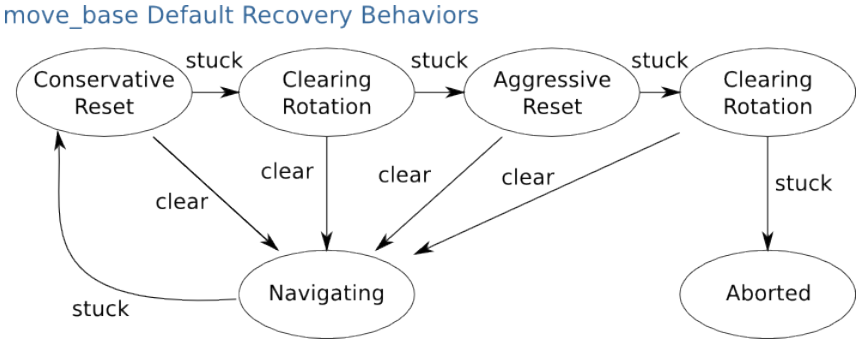

Abbildung 3.12: Zustandsdiagramm der Recovery Behaviors von ROS
Bildquelle: `https://wiki.ros.org/move_base#Expected_Robot_Behavior`

Recovery Behaviors ist Teil von der *move_base* ROS Node und definiert verschiedene Aktionsabläufe für den Roboter während der Navigation. Die Aktionen zielen darauf ab den Roboter, wenn dieser feststeckt und mit der Pfadplanung keine Möglichkeit erkennt sich selbstständig aus der Situation zu befreien, einen Ausweg zu finden. In Abbildung 3.12 sind die verschiedenen Zustände und Übergänge von move_base definiert. Nach jeder Aktion wird eine erneute Pfadplanung von move_base versucht. Wenn diese wieder fehlschlägt (*stuck*), bedeutet dies ein erneutes oder bestehendes Feststecken. Der Übergang *clear* bedeutet, dass move_base wieder einen Pfad gefunden hat. Die Navigation kann fortgesetzt werden (Navigating).
Es bestehen vier Zustände, in denen move_base versucht einen Ausweg zu finden. Zunächst löscht move_base einen Teil der lokalen Costmap mit *clear_costmap_recovery*[27] (Conservative Reset). Ausgehend vom Zentrum mit der Distanz, die im Parameter *conservative_reset_dist* festgelegt ist und hier auf $3m$ festgelegt ist. Wenn dies weiterhin fehlschlägt wird eine 360°Rotation des Roboters um die Z-Achse durchgeführt, mithilfe von *rotate_recovery*[28] (Clearing Rotation). Falls dies immer noch nicht erfolgreich ist, wird der Reset erneut umfassender (Aggressive Reset) versucht. Das bedeutet, dass der Reset der Costmap mit dem vierfachen Wert von */local_costmap/circumscribed_radius* passiert. Auch dieser Wert kann angepasst werden und sollte größer als der *conservative_reset_dist* Parameter sein. Danach würde erneut eine rotate_recovery Aktion durchgeführt werden[29]. Falls keine der Strategien erfolgreich sind, gibt move_base auf und bricht die Navigation ab (Aborted).

[27] clear_costmap_recovery im ROS Wiki: `https://wiki.ros.org/clear_costmap_recovery`
[28] Besrotate_recovery im ROS Wiki: `https://wiki.ros.org/rotate_recovery`
[29] Definition der Recovery Behaviors

4 Validierung des Systems

Nachdem das System fertig aufgesetzt und lauffähig ist, soll die Qualität und der Grad der Funktionsfähigkeit der einzelnen Verfahren beurteilt werden. Dazu werden die erstellten Karten untersucht, die Lokalisierung begutachtet und das korrekte Verhalten der Navigation nachgeprüft.
Das System wurde in einer Büroflur-Umgebung getestet. Dabei ging es darum die verschiedenen Verfahren unabhängig von einander zu überprüfen. Das bedeutet auch, dass vor dem Testen der Navigation und Hindernisvermeidung zunächst eine Qualität der SLAM Karte erreicht wurde, dass die Navigation davon nicht oder so gering wie möglich beeinflusst wird.
Dazu wurden Überprüfungen zur Genauigkeit und Problematiken in erstellten Karten und Problematiken in der Navigation angestellt.

4.1 Kartierung

Damit ein Maß für Genauigkeit des SLAM Algorithmus entwickelt werden kann, wurde entschieden die Längenmaße einer erstellten Karte zu analysieren.

4.1.1 Längenmaßgenauigkeit

Für Problematiken in Bezug auf Längenmaße wurden in Abbildung 4.1 die tatsächlichen, realen Maße in fetter Schrift eingezeichnet. Darunter stehen die Maße, welche in der Karte gemessen wurden, in normaler Schriftstärke. Die realen Maße wurden mithilfe eines Kaleas Laserdistanzmesser LDM 500-60 ermittelt. Dieser besitzt eine Messgenauigkeit von $\pm 1,5mm$[1]. Eine solche Genauigkeit kann in der Karte bzw. mit dem Lidar jedoch nicht erreicht werden.
Grundsätzlich sind die ermittelten Werte in der Karte mit Vorsicht zu genießen. Denn an einigen Stellen ist das Rauschen so stark, dass keine exakte Stelle zum Messen gefunden werden konnte. Es wurde beim Ablesen also teilweise Augenmaß verwendet

[1] Produktseite des Kaleas LDM 500-60

und versucht das digitale Maßband mittig zu setzen, falls die Karte nicht eindeutig war. Die Umrechnung von Pixel in Meter erfolgt folgendermaßen: Anzahl [Pixel] * Auflösung $[\frac{m}{Pixel}]$ = Distanz $[m]$
Beispiel:

$$238 \text{ Pixel} * 0,05 \frac{m}{\text{Pixel}} = 11,9m$$

Die Auflösung kann immer der YAML-Datei zur Zugehörigen PGM-Kartendatei entnommen werden.

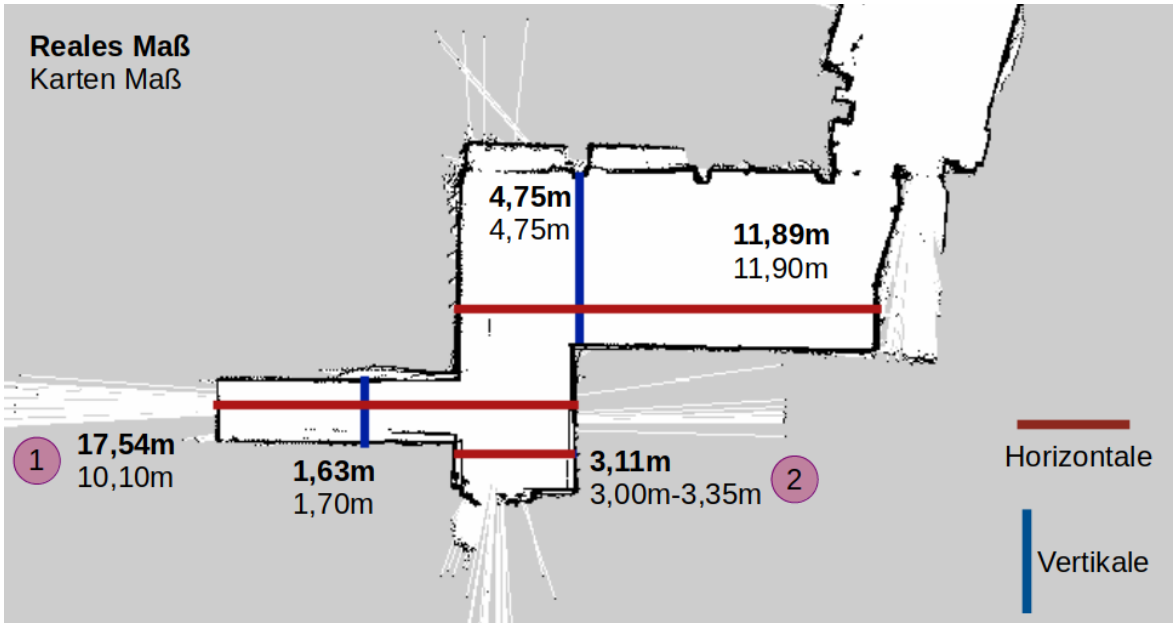

Abbildung 4.1: Vergleich der Längenmaße zwischen realen und kartografierten Maßen Horizontale Maße sind rot, vertikale in blau gekennzeichnet. Weiße Fläche bedeutet frei von Hindernissen, Schwarz ist ein Hindernis und Grau ist unbekanntes Gebiet.

In Abbildung 4.1 wurden zwei Stellen in lila Kreisen Nummeriert.
An **Kartenstelle 1** ist eine Differenz zwischen realem Maß und Kartenmaß von $7,44m$ zu sehen. Dieses Phänomen lässt sich folgendermaßen erklären. In langen Gängen, Fluren oder Tunneln ist es möglich, dass wenig Unterscheidungsmerkmale vorhanden sind. Das bedeutet, dass der Scanmatcher von HectorSLAM keine Bewegung detektiert, weil dieser aus seiner Sicht immer die gleichen Lidardaten erhält. Es kann also nicht unterschieden werden ob die leichten Unterschiede in den Lidardaten lediglich als Rauschen behandelt werden sollen oder ob dies neue Daten aufgrund von Bewegung sind. Selbst wenn dies aufgrund von Bewegung sein sollte, könnte der Scanmatcher dies aufgrund der nicht Unterscheidbarkeit der Daten nicht ermitteln, wie viel bewegt

wurde. Ein weiteres Indiz ist, dass links sowie rechts des Flures Ausreißer zu sehen sind.
An **Kartenstelle 2** wurde keine eindeutige Stelle gefunden, an der gemessen werden konnte. Es sind auf beiden Seiten der Messung fast parallele schwarze Linien zu sehen. Dies ist auf eine unsauberes bzw. nicht korrekt ausgeführtes Scanmatching von Lidardaten des Scanmatchers zurückzuführen. Jedoch scheint sich der Fehler offensichtlich nicht fortgepflanzt zu haben.
Ansonsten stimmen die Kartenmaße mit den realen im Fehlerbereich von einstelligen Zentimetern überein. Für die Navigation ist diese Ungenauigkeit nicht weiter relevant, denn bei dieser Arbeit lag der Fokus nicht auf Genauigkeit der Navigation. Die Lokalisierung hatte größere Abweichungen bzw. Unsicherheiten zu verzeichnen.

4.1.2 Glastürproblematik

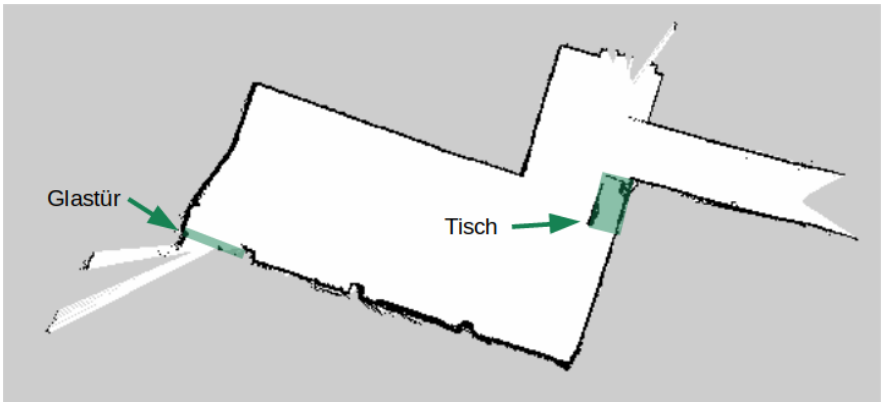

Abbildung 4.2: Ausschnitt einer Karte der Testumgebung eines Büroflurs
Halbtransparent Grün sind die Problemstellen markiert

Bei der Kartenerstellung fiel auf, dass Barrieren aus Glas, also bspw. Glastüren, nicht vom Lidar erkannt wurden. Somit konnte auch der SLAM Algorithmus diese nicht als Hindernis detektieren und in der Karte als solches festlegen. Die halbtransparente, grün markierte Fläche in Abbildung 4.2 soll eine dort vorhandene, teilweise aus Glas bestehende Tür symbolisieren. Allerdings ist innerhalb dieser Fläche nur ein schwarzer Punkt zu erkennen, der eine nicht aus Glas bestehende Säule aus Metall, dem Türrahmen, entspricht.
In der Navigation wäre dies fatal, weil die Glastür nicht als Hindernis erkannt werden würde und der Roboter voraussichtlich dagegen fahren würde.
Eine ähnliche Problematik wurde an Tischen entdeckt. Aufgrund der Tatsache, dass der Lidar nur 2-dimensional, also genau auf der Höhe auf der er verbaut ist, Sensordaten

aufnimmt, war der Tisch offensichtlich ein Grenzfall. Denn die Fläche in der Karte ist weiß gefärbt, das frei von Hindernissen bedeutet. Das ist auch in sofern korrekt, dass unterhalb der Tischplatte kein Hindernis außer den Tischbeinen anzutreffen ist. Jedoch sollte die gesamte Fläche grau markiert sein und damit eine Nicht-Passierbarkeit definieren. Auch dies kann negative Folgen haben. Denn auch hier könnte der Roboter unter Umständen gegen den Tisch fahren, weil dieser nicht korrekt in der lokalen Costmap als Hindernis erkannt wird.
Es wäre zu Prüfen ob mit Visual-SLAM oder 3D SLAM Algorithmen diese Probleme behoben werden könnten.

4.2 Verhalten der Navigation

Beim Ausführen der Navigation wurden einige Beobachtungen zum Verhalten der Navigation gemacht. Zunächst ist eine ausreichende Lokalisierung notwendig. Selbst wenn diese erreicht wurde, kam es vor, dass sich diese während der Navigation verschlechterte und, wie in Abbildung 4.3 zu sehen, uneindeutig wurde. Das bedeutet, dass AMCL mehrere Positionen für wahrscheinlich hält, selbst außerhalb des kartierten Gebietes.

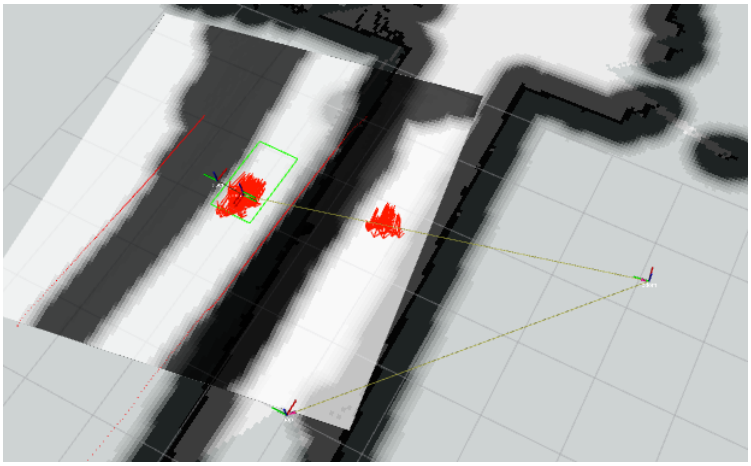

Abbildung 4.3: AMCL mit uneindeutiger Lokalisierung

Aber auch wenn die Lokalisierung im Verlauf der Navigation nicht der realen Position entsprach, wurde nie ein Hindernis angefahren. Allerdings wurde die Lokalisierung dann auch nicht verbessert, was dazu führte, dass das Navigationsziel nicht erreicht wurde. Drehungen, die insbesondere bei den Recovery Beahviors vorkommen, führten meist zu einer deutlichen Verschlechterung der Lokalisierung.
Im Fall, dass die Lokalisierung weiterhin gut blieb, während des Ansteuern der Zielpose, wurden auch Hindernisse korrekt umfahren, wie gewünscht. Durch schmale Verengungen

des Raumes, die in etwa doppelt so breit wie der Scoomatic selbst waren, war kein Durchfahren möglich. Durch den inflation_radius wurden die Durchfahrten so schmal, dass move_base keinen Pfad finden konnte und die Navigation abbrach. Der inflation_radius konnte jedoch nicht kleiner eingestellt werden, weil sonst unter Umständen Hindernisse durch Drehungen angefahren worden wären.

4.3 Bestehende Herausforderungen

Nach Abschluss der Arbeit bestehen weiterhin noch Herausforderungen, für die es noch Problemlösungen bedarf.

4.3.1 IMU Probleme

Während der Bearbeitung dieser Arbeit wurden Probleme an der Hardware festgestellt. Die IMU stellt nicht zuverlässig Daten zur Verfügung. Allerdings ist dies nicht immer der Fall. Das umgebaute Hoverboard kann getrennt vom Raspberry Pi und somit vom Softwarestack ein- und ausgeschalten werden. Wenn das Hoverboard, also letztlich die Motoren, eingeschalten werden, schickt die IMU sporadisch Rauschen. Dieses Rauschen ist jedoch so bestimmend, dass dieses nicht herausgerechnet werden könnte.

Aus diesem Grund konnte die IMU nicht für das System, insbesondere für die Odometrie, verwendet werden. Aufgrund dem Fehlen einer echten Odometrie, hätte das Vorhandensein einer IMU voraussichtlich deutliche Verbesserungen in der Lokalisierung und Navigation erbracht.

4.3.2 Mobiler Netzwerkzugriff

Das Ziel der Mikromobilitätsplattform Scoomatic ist ein voll-autonomer Betrieb von Fahrzeugen im Außenbereich. Für Innenbereiche ist WLAN als Netzwerkverbindung ausreichend. WLAN ist jedoch nicht für flächendeckendes Versorgen von Städten konzipiert. Wie der Name schon sagt: *Wireless Local Area Network*, ist das System nur auf lokale Flächen und keine Regionen geeignet. Dementsprechend müsste eine Mobilfunk Verbindung zum Internet hergestellt werden, damit ein Steuern, Überprüfen und Speichern von Ladezustand, Standort, usw. für die Betreibenden möglich wäre.

4.3.3 Rechenkapazität des Raspberry Pi 3B

HectorSLAM ist nicht sinnvoll auf dem Raspberry Pi 3B, also mobil auszuführen. Die Rechenleistung des Computers ist während der Ausführung zu 100% ausgelastet. Somit ist keine manuelle Steuerung mehr möglich, weil die Motorsteuerkommandos nicht am Motor ankommen. Deshalb wird hier HectorSLAM auf dem externen Rechner ausgeführt. Somit ist allerdings keine vollständige Autonomie möglich. Deshalb ist ein leistungsstärkeres System notwendig.

5 Ausblick

Aktuell bestehen an dem System folgende Problematiken:
Es werden nicht alle Hindernisse erkannt. Hindernisse, welche nicht auf der Höhe des Lidar sind, werden nicht erkannt. Dies kann fatal sein, und ist unbedingt zu vermeiden. Vor allem ist so ein sicheres Betreiben im Beisammensein mit Menschen, also in der Öffentlichkeit ohne menschliche Kontrolle, nicht sichergestellt.
Insbesondere Glas bzw. Gegenstände die ganz oder teilweise aus Glas bestehen, werden nicht zuverlässig erkannt. Diese Herausforderung könnte mit zusätzlichen Sensoren oder einem anderen Lidar gelöst werden, der Glas erkennen kann.
Des weiteren ist die Reichweite des RPLidar A1 mit 12m verhältnismäßig gering und kann im Außenbereich unter Umständen zu gering sein. So kann eine Lokalisierung bzw. die Kartierung nicht Fehlerfrei garantiert werden.
Für den Außenbereich wäre zudem eine globale Lokalisierung notwendig, die mit GPS bzw. GALILEO realisiert werden könnte. Dafür besteht aktuell noch keine Implementierung, wäre aber notwendig damit sich der Roboter weltweit orten kann.
Die unechte Odometrie ist unzureichend, weil sie keine zuverlässigen Daten liefert. Die Leistung des Motors kann nicht als Odometrie verwendet werden. Es müssen Alternativen ausprobiert werden und externe Sensoren verwendet werden.
Die IMU Daten sind nicht zuverlässig abrufbar. Durch das Beheben dieses Fehlers, könnte eine erhöhte Genauigkeit der Navigation und SLAM erfolgen. Zudem wäre es so in HectorSLAM möglich unebenen Untergrund zu modellieren.
Aktuell ist es noch nicht möglich einen Notstopp zu betätigen, der das gesamte System im Notfall abschaltet. Ein Notstopp dürfte jedoch nicht nur durch Abschalten des Stromes erfolgen. Denn so ist es unter Umständen nicht möglich nach der Ursache zu forschen und wichtige Daten würden verloren gehen.
Für die zuverlässige Steuerung des Scoomatics müsste eine Motor-Regelung integriert sein, die aktuell noch nicht existiert. So wäre es bspw. möglich einen PID-Regler zu implementieren, welcher bei einer vorgegebener Soll-Leistung oder Soll-Geschwindigkeit den Motor so regelt, dass dieser Wert auch erreicht wird.
ROS Melodic ist veraltet, unter anderem auch aufgrund von Python 2.7. Dies stellt

ein potentielles Sicherheitsrisiko dar, da der Roboter mit dem Internet verbunden ist. Außerdem ist die Zukunftsfähigkeit der Plattform so nicht sichergestellt. Das Problem ist jedoch, dass es von ROS2 - zum aktuellen Zeitpunkt - noch keine stabile Version existiert.
Die Rechenleistung des Raspberry Pi 3B ist zu gering, damit SLAM sinnvoll darauf ausgeführt werden kann. Dies wird in Zukunft gelöst durch Verwenden der NVidia Jetson Plattform. Diese stellt eine mehrfach erhöhte Rechenleistung zur Verfügung.
Außedem ist HectorSLAM ungeeignet für das Ziel der Mikromobilitätsplattform Scoomatic. Karten sind nachträglich nicht erweiterbar, können also nicht pausiert oder ähnliches werden. Vor allem aber wird ein 2D-Verfahren verwendet, also auch ein 2D-Lidar, der die entscheidende Schwachstelle darstellt. Durch einen 3D-Lidar könnten Hindernisse, die eine Höhenausdehnung besitzen sicher erkannt werden. Durch ein geeignetes 3D-SLAM Verfahren könnten auch die bereits genannten Kritikpunkte ausgemerzt werden. Gleiches gilt für einen kamerabasierten Ansatz, welches Visual-SLAM zur Kartenerstellung verwenden könnte.
Wünschenswert wäre außerdem eine autonome Erkundung und Kartierung der Umgebung durch den Roboter. Aktuell muss die Karte manuell, bei Beisammensein eines Menschen, erstellt werden. Dies ist zeitaufwendig und muss unter Umständen bei Veränderungen der Umgebung wiederholt werden. Bei einer autonomen Erkundung könnte der Algorithmus dies selbstständig erledigen. Für HectorSLAM existiert aktuell bereits eine Möglichkeit dafür[1].
Schlussendlich wäre eine Validierung der Hindernisvermeidung notwendig. Es muss garantiert werden können, dass auf Hindernisse jeglicher Art immer und überall zuverlässig und sicher reagiert wird. Die autonome Navigation ist zweitrangig, denn die Sicherheit des Systems hat oberste Priorität.

[1] hector_exploration_planner im ROS Wiki: `https://wiki.ros.org/hector_exploration_planner`

6 Fazit

Es bestehen verschiedene Probleme und Herausforderungen, für die jedoch Lösungsvorschläge empfohlen wurden. Diese könnten in bestehenden Prototypen eingearbeitet werden. So kann die Mikromobilitätsplattform Scoomatic und dessen Prototyp weiterentwickelt werden, um in Zukunft ein nachhaltiges Verkehrsmittelangebot zu erschaffen.
Das Ziel dieser Arbeit, die Hinderniserkennung und -vermeidung, wurde erfolgreich implementiert und realisiert. Sie funktioniert unter realen Bedingungen gelegentlich und nicht zuverlässig. Für das voll-autonome Betreiben in der realen Umgebung reicht das System nicht aus.
Für die Realisierung der Funktionen wurde die Software von ROS2 auf ROS1 migriert, weitere Nodes programmiert und die notwendigenROSPackages konfiguriert und dokumentiert. Somit steht die Kartenerstellung, Lokalisierung und Navgation zur Verfügung.

Abbildungsverzeichnis

2.1 Modell eines Differentialantriebs 8
2.2 Übersicht über Navigation und Mapping Prozess 13
2.3 Veranschaulichung des allgemeinen Laser Triangulation Prinzips 16
2.4 Schrägansicht des RPLidar A1 17
2.5 Belief des Roboters zu verschiedenen Zeitpunkten ohne Sensordaten . . 22

3.1 Der Hardware Prototyp des Scoomatic Roboters 26
3.2 Das verwendete Gamepad zur manuellen Steuerung 28
3.3 Netzwerkübersicht über das Gesamtsystem 30
3.4 TF Frame-Baumstruktur während des SLAM 34
3.5 URDF Scoomatic Modell mit TF Koordinatensystemen 36
3.6 Beispiel von RViz mit Anzeige von Navigationsdaten 39
3.7 Beispiel von Rqt mit Anzeige von IMU Daten 40
3.8 Datenfluss der ROS Nodes und Topics bei der Navigation 41
3.9 Übersicht über den ROS Navigation Stack 44
3.10 Datenfluss der ROS Nodes und Topics bei der Navigation 45
3.11 Darstellung des Zustandes vor der Globalen Lokalisierung in RViz . . . 46
3.12 Zustandsdiagramm der Recovery Behaviors von ROS 49

4.1 Vergleich der Längenmaße zwischen realen und kartografierten Maßen . 52
4.2 Ausschnitt einer Karte der Testumgebung eines Büroflurs 53
4.3 AMCL mit uneindeutiger Lokalisierung 54

Akronyme

AMCL Adaptive Monte Carlo Localization. 18–22, 27, 33, 35, 37, 46, 54

DWA Dynamic Window Approach. 44, 47

ICR Instantaneous center of rotation. 9

IMU Inertial Measurement Unit. 12, 27, 32, 40, 55, 57, 61

Lidar Light detection and ranging. 13–17, 26, 27, 35, 40, 45, 47, 48, 53, 57

MCL Monte Carlo Localization. 18, 20

PGM Portable Graymap. 37, 42, 52

RBPF Rao-Blackwellized Partikel Filter. 42

REP ROS Enhancement Proposal. 35

ROS Robot Operating System. I, II, 1, 3, 5–7, 9, 10, 12, 13, 18, 22, 24, 25, 28–32, 34, 35, 38–45, 49, 59

SLAM Simultaneous Localization and Mapping. 5, 6, 10–12, 16, 18, 21, 32, 36, 39, 41–44, 51, 53, 54, 57, 58

SSH Secure Shell. 29–31

TCP Transmission Control Protocol. 31

TF Transform Library. 33–36

UART Universal Asynchronous Receiver Transmitter. 17

UDP User Datagram Protocol. 31

URDF Unified Robot Description Format. 33, 36

USAR Urban Search an Recovery. 43

WLAN Wireless Local Area Network. 6, 29, 30

XML Extensible Markup Language. 31, 33

YAML YAML Ain't Markup Language. 23, 37, 42, 52

Literatur

[1] Garry Berkovic und Ehud Shafir. „Optical methods for distance and displacement measurements“. In: *Advances in Optics and Photonics* 4.4 (31. Dez. 2012), S. 441. ISSN: 1943-8206. DOI: 10.1364/AOP.4.000441 (siehe S. 16).

[2] Oliver Bittel. *Kinematik mobiler Roboter*. Mobile Roboter. URL: http://www-home.htwg-konstanz.de/~bittel/ain_robo/Vorlesung/03_Roboterkinematik.pdf (siehe S. 8, 9).

[3] BMVI. *Mobilität in Deutschland 2017 Ergebnisbericht*. Dez. 2018. URL: https://www.bmvi.de/SharedDocs/DE/Anlage/G/mid-ergebnisbericht.pdf?__blob=publicationFile (siehe S. 1).

[4] E. W. Dijkstra. „A note on two problems in connexion with graphs“. In: *Numerische Mathematik* 1.1 (1. Dez. 1959), S. 269–271. ISSN: 0945-3245. DOI: 10.1007/BF01386390 (siehe S. 22).

[5] Tully Foote. „tf: The transform library“. In: 2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA). Woburn, MA, USA: IEEE, Apr. 2013, S. 1–6. ISBN: 978-1-4673-6225-2 978-1-4673-6223-8 978-1-4673-6224-5. DOI: 10.1109/TePRA.2013.6556373 (siehe S. 35).

[6] D. Fox, W. Burgard und S. Thrun. „The dynamic window approach to collision avoidance“. In: *IEEE Robotics Automation Magazine* 4.1 (März 1997). Conference Name: IEEE Robotics Automation Magazine, S. 23–33. ISSN: 1558-223X. DOI: 10.1109/100.580977 (siehe S. 23, 24).

[7] G. Grisettiyz, C. Stachniss und W. Burgard. „Improving Grid-based SLAM with Rao-Blackwellized Particle Filters by Adaptive Proposals and Selective Resampling“. In: 2005 IEEE International Conference on Robotics and Automation. Barcelona, Spain: IEEE, 2005, S. 2432–2437. ISBN: 978-0-7803-8914-4. DOI: 10.1109/ROBOT.2005.1570477 (siehe S. 42).

[8] Matthias Heise. *Laser-Triangulation: Von ein bis drei Dimensionen*. URL: https://www.elektroniknet.de/elektronik/messen-testen/von-ein-bis-drei-dimensionen-141137.html (besucht am 19. 04. 2020) (siehe S. 16).

[9] Joachim Hertzberg, Kai Lingemann und Andreas Nüchter. *Mobile Roboter*. eXamen.press. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. ISBN: 978-3-642-01725-4 978-3-642-01726-1. DOI: 10.1007/978-3-642-01726-1 (siehe S. 7, 13, 23).

[10] Vivian Ho. *Stolen, burned, tossed in the lake: e-scooters face vandals' wrath*. The Guardian. 28. Dez. 2018. URL: https://www.theguardian.com/us-news/2018/dec/28/scooters-california-oakland-los-angeles-bird-lime (besucht am 27. 04. 2020) (siehe S. 2).

[11] INRIX. *INRIX Verkehrsstudie: Stau verursacht Kosten in Milliardenhöhe*. Inrix. Library Catalog: inrix.com. URL: https://inrix.com/press-releases/2019-traffic-scorecard-german/ (besucht am 27. 04. 2020) (siehe S. 1).

[12] Stefan Kohlbrecher u. a. „A flexible and scalable SLAM system with full 3D motion estimation". In: 2011 IEEE International Symposium on Safety, Security, and Rescue Robotics. ISSN: 2374-3247. Nov. 2011, S. 155–160. DOI: 10.1109/SSRR.2011.6106777 (siehe S. 13–15, 44).

[13] Pawel Lewicki. *Straßenverkehrslärm*. Umweltbundesamt. 8. Juli 2013. URL: https://www.umweltbundesamt.de/themen/verkehr-laerm/verkehrslaerm/strassenverkehrslaerm (besucht am 27. 04. 2020) (siehe S. 1).

[14] *Publishing Odometry Information over ROS*. URL: http://wiki.ros.org/navigation/Tutorials/RobotSetup/Odom#Publishing_Odometry_Information_Over_ROS (besucht am 05. 05. 2020) (siehe S. 9).

[15] *REP 1 – REP Purpose and Guidelines (ROS.org)*. 20. Feb. 2012. URL: https://www.ros.org/reps/rep-0001.html (besucht am 14. 04. 2020) (siehe S. 35).

[16] *REP 105 – Coordinate Frames for Mobile Platforms (ROS.org)*. 20. Okt. 2010. URL: https://www.ros.org/reps/rep-0105.html (besucht am 14. 04. 2020) (siehe S. 35).

[17] Joao Machado Santos, David Portugal und Rui P. Rocha. „An evaluation of 2D SLAM techniques available in Robot Operating System". In: *2013 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*. 2013 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR).

Linkoping, Sweden: IEEE, Okt. 2013, S. 1–6. ISBN: 978-1-4799-0880-6 978-1-4799-0879-0. DOI: 10.1109/SSRR.2013.6719348 (siehe S. 43, 44).

[18] Shanghai Slamtec.Co.,Ltd. *RPLIDAR A1 - Interface Protocol and Application Notes*. 28. März 2019. URL: http://bucket.download.slamtec.com/ccb3c2fc1e66bb00bd4370e208b670217c8b55fa/LR001_SLAMTEC_rplidar_protocol_v2.1_en.pdf (besucht am 27. 04. 2020) (siehe S. 27).

[19] Ltd Shanghai Slamtec.Co. *RPLIDAR A1 - Introduction and Datasheet*. 2. Dez. 2020. URL: https://download.slamtec.com/api/download/rplidar-a1m8-datasheet/2.4?lang=en (besucht am 27. 04. 2020) (siehe S. 17).

[20] Sebastian Thrun, Wolfram Burgard und Dieter Fox. *Probabilistic robotics*. Intelligent robotics and autonomous agents. OCLC: ocm58451645. Cambridge, Mass: MIT Press, 2005. 647 S. ISBN: 978-0-262-20162-9 (siehe S. 12, 15, 18, 19, 21, 22).

[21] VCD Verkehrsclub Deutschland e.V. *Rückeroberung der Straße*. Juni 2016. URL: https://www.vcd.org/fileadmin/user_upload/Redaktion/Publikationsdatenbank/Fussverkehr/2016_Position_Rueckeroberung_der_Stasse.pdf (besucht am 27. 04. 2020) (siehe S. 1).

[22] Sibylle Wilke. *Fahrleistungen, Verkehrsaufwand und „Modal Split“*. Umweltbundesamt. Library Catalog: www.umweltbundesamt.de Publisher: Umweltbundesamt. 1. Juli 2013. URL: https://www.umweltbundesamt.de/daten/verkehr/fahrleistungen-verkehrsaufwand-modal-split (besucht am 27. 04. 2020) (siehe S. 1).

[23] Rauf Yagfarov, Mikhail Ivanou und Ilya Afanasyev. „Map Comparison of Lidar-based 2D SLAM Algorithms Using Precise Ground Truth“. In: *2018 15th International Conference on Control, Automation, Robotics and Vision (ICARCV)*. 2018 15th International Conference on Control, Automation, Robotics and Vision (ICARCV). Nov. 2018, S. 1979–1983. DOI: 10.1109/ICARCV.2018.8581131 (siehe S. 43).

Erklärung

Die vorliegende Arbeit habe ich selbstständig ohne Benutzung anderer als der angegebenen Quellen angefertigt. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Quellen entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer oder anderer Prüfungen noch nicht vorgelegt worden.

Stuttgart, den 06. 05. 2020 Henri Chilla