

Hochschule Karlsruhe
Fakultät für Maschinenbau und Mechatronik

**Autonome Navigation mehrerer
TurtleBots im Schwarm**

WS17_WI_TurtleBot

Forschungs- und Entwicklungsprojekt I
(EMFM 150)

Michael Meindl (memi1022)

Betreuer: Prof. Dr.-Ing. Joachim Wietzke
Koordinator: Prof. Dr.-Ing. Eckhard Martens

Karlsruhe, Wintersemester 2017/2018

Inhaltsverzeichnis

| | |
|---------------------------------------------------------|-----------|
| Abbildungsverzeichnis | v |
| Listings | vii |
| 1 Einleitung | 1 |
| 2 Aufbau des Gesamtsystems und der Infrastruktur | 5 |
| 3 Funktionsprinzip des globalen Planers | 11 |
| 3.1 Forward-Search: Breitensuche | 13 |
| 3.2 Forward-Search: Tiefensuche | 15 |
| 3.3 Optimale Suche und Planung | 16 |
| 3.3.1 A* | 17 |
| 4 Stochastische Modelle in der Robotik | 21 |
| 4.1 Geschwindigkeitsbasiertes Bewegungsmodell | 23 |
| 4.2 Messmodell | 27 |
| 4.3 Kontinuierliches Bayes Filter | 29 |
| 5 Kartenerstellung | 35 |
| 5.1 Occupancy-Grid-Mapping | 35 |
| 6 Lokaler Planer: Dynamic Window Approach | 39 |
| 6.1 Anwendungsbeispiel | 41 |
| 7 Erprobung der Navigation | 47 |
| 7.1 Anwendungsszenario 1: Triviale Navigation | 48 |
| 7.2 Anwendungsszenario 2: Hindernisdetektion | 50 |
| 7.3 Anwendungsszenario 3: Lokalisierung | 52 |
| 8 ROS Implementierung | 55 |
| 8.1 Systemstruktur | 56 |
| 8.2 Inbetriebnahme der Turtlebots | 57 |
| 8.2.1 Fernsteuerung eines Turtlebots | 58 |

| | | |
|----------|-----------------------------------------------|-----------|
| 8.2.2 | Aufzeichnung einer Karte | 59 |
| 8.2.3 | Navigation eines einzelnen Roboters | 61 |
| 8.2.4 | Navigation mehrerer Roboter | 63 |
| 9 | Ausblick und Fazit | 69 |
| | Literaturverzeichnis | 71 |

Abbildungsverzeichnis

| | | |
|-----|---------------------------------------------------------------------------|----|
| 2.1 | Überblick des Gesamtsystems | 5 |
| 2.2 | Übersicht des Navigationstack, Inhalt teilweise aus [21] | 8 |
| 3.1 | Suchraum als Baum dargestellt | 14 |
| 3.2 | Ergebnis der Breitensuche | 15 |
| 3.3 | Ergebnisse der Tiefensuche bei unterschiedlichen Reihenfolge der Aktionen | 16 |
| 3.4 | Ergebnis mit Dijkstra's Algorithmus | 18 |
| 3.5 | Ergebnis mit dem A* Algorithmus | 19 |
| 4.1 | Darstellung des Momentanpols am Zeitpunkt t [1, S. 126] | 23 |
| 5.1 | Schematische Darstellung der Karten und des inversen Messmodells . . | 38 |
| 6.1 | Darstellung der Trajektorie und Hindernissen | 42 |
| 7.1 | Übersicht von RViz | 48 |
| 7.2 | Globaler Plan | 49 |
| 7.3 | Globale Kostenkarte des Labors | 49 |
| 7.4 | Roboter und Sensordaten mit Hindernis | 50 |
| 7.5 | Integration der Sensordaten in die Pfadplanung | 51 |
| 7.6 | Ausgangszustand der Lokalisierung nach Positionsschätzung | 52 |
| 7.7 | Positionsschätzung nach kurzer Fahrdistanz | 53 |
| 7.8 | Positionsschätzung bei Erreichen der Zielposition | 53 |
| 8.1 | Transformationsbaum von Roboter 4 | 66 |
| 8.2 | Transformationsbaum nach gesetztem Transformationspräfix | 68 |

Listings

| | | |
|------|-----------------------------------------------------|----|
| 3.1 | Ablauf der Forward-Search in Pseudocode | 13 |
| 3.2 | Dijkstra's Algorithmus in Pseudocode | 17 |
| 4.1 | Bayes-Filter | 30 |
| 8.1 | EML_Hardware_Init_Slave.launch | 58 |
| 8.2 | EML_Mapping_Slave.launch | 59 |
| 8.3 | EML_Mapping_Master.launch | 59 |
| 8.4 | Anleitung zur Kartenaufzeichnung | 60 |
| 8.5 | EML_Navigation_Slave.launch | 61 |
| 8.6 | EML_Navigation_Master.launch | 61 |
| 8.7 | eml_move_base.launch | 62 |
| 8.8 | Anleitung Navigation eines Roboters | 63 |
| 8.9 | EML_NS_Hardware_Init_Slave.launch | 63 |
| 8.10 | EML_NS_Teleop_Master.launch | 64 |
| 8.11 | EML_NS_Navigation_Slave.launch | 64 |
| 8.12 | EML_NS_TF_Hardware_Init_Slave.launch | 67 |
| 8.13 | Ausschnitt aus EML_minimal.launch | 67 |
| 8.14 | Ausschnitt aus EML_mobile_base.launch.xml | 68 |

Kapitel 1

Einleitung



Seit einiger Zeit wird an der Hochschule Karlsruhe die Projektidee eines künstlichen Chors verfolgt, dessen Ziel darin besteht, eine Audiospur simultan in mehreren Tonhöhen wiederzugeben, um einen mehrstimmigen Chor zu erzeugen. Das zugrundeliegende technische Prinzip lässt sich recht leicht erläutern: die originale Tonspur wird mittels FFT in den Frequenzbereich transformiert, wo das Signal in die gewünschte Tonhöhe bzw. Frequenzbereich verschoben wird. Anschließend werden die Signale zurück in den Zeitbereich transformiert und abgespielt. Allerdings gestaltet sich die Umsetzung weitaus anspruchsvoller als das theoretische Konzept. Beispielsweise führen Aussetzer im WLA-Netzwerk, Puffergrößen der Audiotreiber und diskrete Stützstellen des Frequenzspektrums zu unerwarteten Problemen, die sich in Form von knackenden Lautsprechern Gehör verschaffen. Nichtsdestotrotz hat der Entwicklungsprozess den Punkt erreicht, an dem der Chor noch kaum unerwünschte Töne von sich gibt, weshalb nun die erste große Erweiterung angestrebt wird. Aktuell setzt sich das System aus einem Master-PC und vier BeagleBones zusammen, die jeweils mit einem Lautsprecher ausgestattet in einem Raum verteilt werden. Auf dem Master wird eine Tonspur eingespielt, die er in die gewünschten Tonhöhen pitcht und per WLAN an die BeagleBones sendet, welche jeweils eine der Stimmen über den Lautsprecher wiedergeben.



Im nächsten Schritt soll der Chor mobilisiert werden. Das heißt jedes BeagleBone plus Lautsprecher wird auf einem Roboter montiert, die bei einer Vorführung einen beliebigen Raum betreten können und diesen auf der Suche nach einem passenden Podium erkunden. Wurde Letzteres gefunden, so sollen sich die Roboter dort im Halbkreis formieren und anschließend vorsingen.

Die neue Aufgabe, die sich mit der autonomen Navigation der Roboter beschäftigt, kann vollkommen entkoppelt von dem Chor betrachtet und bearbeitet werden. Dieser zweite Entwicklungsstrang befindet sich allerdings noch im Anfangsstadium, da sich bisher nur ein einzelnes Entwicklungsprojekt [8] dem Thema gewidmet hat. In dieser Arbeit wurde ein erster Grundstein gelegt, der sich einerseits aus der Auswahl und Inbetriebnahme der Roboter, andererseits aus einem ersten Proof-of-Concept in Sachen

Navigation zusammensetzt.

Als mobile Plattform für die Lautsprecher fiel die Wahl auf den Turtlebot 2 [24], der von Willow Garage entwickelt wurde. Der Roboter setzt sich aus der Kobuki-Basis, einem Rechner und einer ASUS-Xtion-Pro-Live zusammen, wobei Letztere sowohl Kamera als auch Tiefensensor zur Verfügung stellt. Der Turtlebot 2 bringt als Vorteil mit sich, dass der Roboter weite Anwendung im Bereich von Hobby- und Forschungsanwendungen findet, weshalb eine breite Community-Untersützung zur Verfügung steht. Ein in dieser Arbeit wichtiges Beispiel stellt die Simulation des Roboters mithilfe von GAZEBO [25] dar: Hier besteht bereits eine vollständige Integration des Turtlebot 2, die quelloffen zugänglich ist. Im Rahmen der Vorgängerarbeit wurde der Roboter zusätzlich mit dem Laserscanner Tim551 der Marke SICK ausgestattet [3], der als primärer Sensor für die Navigation verwendet wird.

Die Programmierung erfolgt mithilfe des sogenannten Robot-Operating-System (ROS), wobei es sich ironischerweise um kein Betriebssystem sondern eine Middleware handelt, die eine Vielzahl von Tools und Paketen für die autonome Navigation bereitstellt. Dadurch müssen die Algorithmen für die Navigation und Kartographierung nicht eigenständig implementiert werden, sondern können in Form von ROS-Paketen eingesetzt werden, sodass sich die Inbetriebnahme auf die Parametrisierung der ROS-Funktionen beschränkt. Außerdem beinhaltet ROS ein Netzwerk, über das die Roboter kommunizieren können. Mithilfe der Kommunikationskanäle können auch externe Werkzeuge auf relevanten Daten zugreifen, womit komfortable und effiziente Debugging-Wege geschaffen werden. Umgekehrt können zu Beginn die Roboter auch aus dem Netz entfernt und durch ein Simulationstool wie GAZEBO ersetzt werden. So können die Algorithmen zunächst anhand einer Simulation erprobt und konfiguriert werden und im Anschluss unverändert auf die Roboter übertragen werden.

Die im Vorgängerprojekt [8] erarbeitete Navigationslösung basiert auf dem ROS-Navigation-Stack und setzt sich aus zwei Teilen zusammen. Im ersten Schritt wird mit dem ROS-Paket **hector_slam** [18] eine Karte der Umgebung aufgezeichnet. Anschließend navigieren die Roboter anhand der Karte zu dem vorgegebenen Ziel, wobei das ROS-Paket **move_base** [21] zum Einsatz kommt. Bei dessen Konfiguration wurden zwei verschiedene Ansätze verfolgt: Bei dem Ersten erfolgt die Lokalisierung des Roboters ausschließlich anhand der Odometriedaten. Der zweite Ansatz greift auf die AMC-Lokalisierung zurück. Der Vorteil der zweiten Variante liegt darin, dass die Ausgangsposition des Roboters nicht bekannt sein muss. Bei der Navigation mittels Odometrie übertragen sich sämtliche Fehler bei der Angabe der Anfangsposition unmittelbar auf die Navigation. Im Gegensatz dazu verspricht die AMCL-Variante - zumindest theoretisch - ein höheres Maß an Robustheit. Allerdings konnte diese Hoffnung in den Experimenten nicht bestätigt werden, weshalb letzten Endes die Odometrie-Navigation verwendet wurde [8, S. 43]. Zusätzlich sei angemerkt, dass diese Lösung nicht im Stande ist, Hindernisse, die nicht auf der Karte verzeichnet sind, während der Navigation

zu erkennen, geschweige denn, denen auszuweichen.

An dieser Stelle knüpft die vorliegende Arbeit an: Die Navigation soll an den Punkt gebracht werden, wo die Roboter im Stande sind, sich selbst auf der Karte zu lokalisieren, unerwartete Hindernisse zu detektieren und diesen entsprechend auszuweichen. Hierfür soll auf bestehende Möglichkeiten des ROS-Navigation-Stack zurückgegriffen werden, um die Menge von Lösungsmöglichkeiten einzuschränken. Auch die Anforderung, dass die Navigation auf eine mit **hector_slam** aufgezeichnete Karte zurückgreifen kann, wird übernommen. Im ersten Schritt wird lediglich ein einzelner Roboter betrachtet. Sobald dieser die Anforderungen im Bereich der autonomen Navigation erfüllt, werden die Ergebnisse auf eine Gruppe von vier Robotern übertragen.

Die funktionalen Anforderungen dieser Arbeit bestehen darin, die autonome Navigation mehrerer Roboter zu ermöglichen. Darüber hinaus besteht der Anspruch darauf, die zugrundeliegenden Algorithmen nachzuvollziehen und zu verstehen. Die ROS-Implementierung sollen nicht nur derartig parametrisiert werden, damit die funktionalen Anforderungen erfüllt sind. Sondern es soll der Zusammenhang zwischen den Parametern und dem Funktionsprinzip der Navigation erarbeitet werden, sodass Letzteres verstanden werden kann. Hierfür werden zu Beginn der Arbeit theoretische Grundlagen der Planung und stochastischen Modellierung diskutiert, deren elaborierte Ausführungen die Bausteine der autonomen Navigation bilden. Ein Hauptproblem der Navigationsaufgabe besteht darin, dass die Teilprobleme nicht entkoppelt betrachtet werden können, woraus eine unerwünschte Komplexität resultiert. Um die einzelnen Komponenten isoliert betrachten zu können, wird im Rahmen dieser Arbeit auf Simulationswerkzeuge wie MATLAB und GAZEBO zurückgegriffen, in denen Daten, die unter realen Umständen mittels zusätzlicher Algorithmen geschätzt werden müssen, unmittelbar abgegriffen werden können. Anhand dieser vereinfachten Beispiele erfolgt die Diskussion der Teilprobleme wie Planung und Lokalisierung. Im Anschluss wird auf die ROS-Implementierung der Algorithmen eingegangen, die zuletzt in Betrieb genommen und in realen Szenarien getestet wird.

Zu Beginn der Arbeit wird ROS' Navigation-Stack in seine einzelnen Komponenten unterteilt und deren Hauptfunktionen beschreiben. Im Anschluss wird das Funktionsprinzip der einzelnen Komponenten im Detail betrachtet. Diese Untersuchung beginnt bei dem globalen Planer, der als einzige Komponente mit deterministischen Modellen arbeitet und deshalb eine Sonderstellung unter den Komponenten einnimmt. Die restlichen Komponente basieren auf stochastischen Modellen, weshalb im Anschluss zuerst die Bedeutung von stochastischen Modellen in der Robotik diskutiert und am Beispiel eines Bewegungs- sowie Messmodells erläutert wird.

Der darauffolgende Abschnitt widmet sich der Kartenerstellung, wobei es sich um eine Aufgabe handelt, die nicht Teil des Navigation-Stacks ist. Allerdings bietet der hier verwendete Kartographieralgorithmus einen Einblick in das Funktionsprinzip stochastischer Modelle und rundet das Verständnis der Navigation insofern ab, dass

nach wie vor Karten hierfür erforderlich sind. Als letztes theoretisches Konzept wird der lokale Planer diskutiert, der auf den Dynamic Window Approach zurückgreift, um die Steuerbefehle des Roboters zu berechnen.

Zuletzt folgt die Inbetriebnahme der Roboter und die Umsetzung der Navigationsalgorithmen unter ROS.

Kapitel 2

Aufbau des Gesamtsystems und der Infrastruktur

In der Endvorstellung des Projektes navigieren vier Roboter simultan durch denselben Raum, woran recht leicht ersichtlich wird, dass ein Weg zum Datenaustausch zwischen den Robotern geschaffen werden muss. Auch eine zentrale Steuereinheit soll in der Lage sein, alle Roboter anzusprechen und zu dirigieren. Selbst ein einzelner Roboter stellt bereits einige Anforderungen an die Kommunikationsstruktur: Da eine erfolgreiche Navigation auf dem Zusammenspiel mehrerer komplexer Algorithmen basiert, müssen externe Analysetools relevante Daten abgreifen, die zwischen den Komponenten ausgetauscht werden. Diese Form des Debuggings ist unerlässlich, um das Verhalten des Roboters nachvollziehen zu können. In diesem Projekt erfüllt ROS diese Anforderungen, weshalb in den folgenden Abschnitten das Kommunikationskonzept unter ROS erläutert wird. Auch die Simulations- und Analysewerkzeuge, die in der Arbeit zum Einsatz kommen, werden vorgestellt und deren ROS-Schnittstellen erläutert.

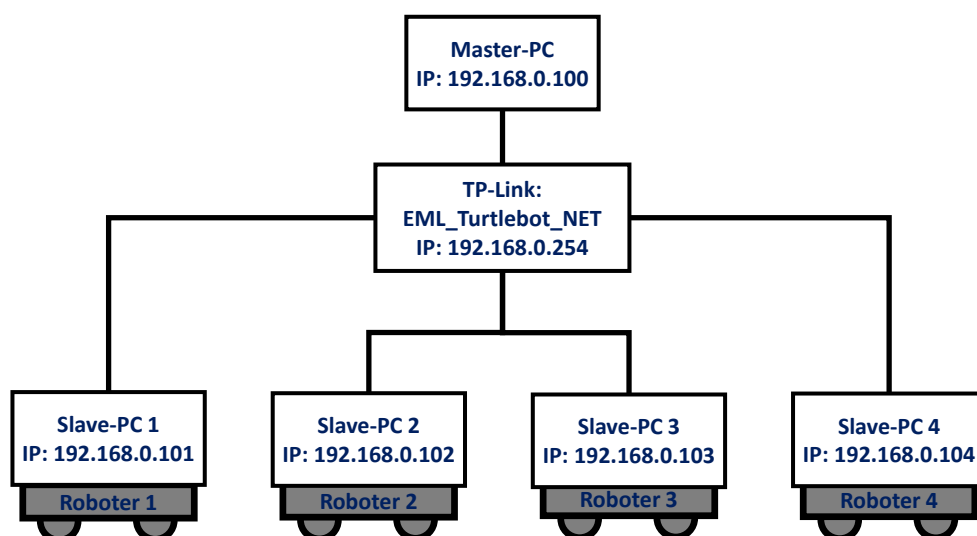


Abbildung 2.1: Überblick des Gesamtsystems

Jeder TurtleBot ist mit einem Mini-PC ausgestattet, auf denen Ubuntu 14.04 und ROS-Indigo installiert sind. Als zentrale Steuereinheit des Roboterverbundes fungiert ein weiterer Mini-PC, der über WLAN mit den Roboter-PCs verbunden ist. Das private Netzwerk wird mittels eines TP-Link Routers verwaltet, dem auch weitere Entwicklungsrechner beitreten können. Auf dem Master-PC wird **ros_core** ausgeführt, über den die Kommunikation abläuft. Als Kommunikationsmittel werden in ROS Nachrichten verwendet, die jeweils unter einer so genannten **topic** veröffentlicht werden. Beispielsweise versendet der Laserscanner zyklisch eine Nachricht des Typs **sensor_msg/laser_scan** unter der topic **/scan**. Alle Nodes, die an den Sensordaten interessiert sind, abonnieren die Topic. Indem Analysewerkzeuge wie MATLAB relevante Topics mithören, können die zugehörigen Daten aufgezeichnet und visualisiert werden. Umgekehrt ist es auch möglich den TurtleBot vollständig durch eine Simulation zu ersetzen. GAZEBO veröffentlicht alle Topics, die für gewöhnlich von dem TurtleBot versendet werden, wodurch dieser ersetzt wird. Die restlichen Bestandteile des Netzwerkes bleiben erhalten, weshalb der Wechsel zwischen Realität und Simulation mühelos abläuft.

Die Idee GAZEBO in Verbindung mit ROS für die Simulation zu verwenden entstammt der Arbeit [4]. Dort wurde unter einer ähnlichen Konfiguration ein P3-DX Roboter verwendet. Der primäre Vorteil dieser Simulationsstruktur besteht darin, dass die Algorithmen und Konfigurationen unverändert von der Simulation auf den Roboter übertragen werden können. GAZEBO bietet auch für die TurtleBots eine vollständige Unterstützung, das heißt es besteht eine frei zugängliche Integration des TurtleBot in GAZEBO. Allerdings existiert keine Implementation des hier verwendeten Laserscanner, Tim551, weshalb die Sensorik nicht exakt abgebildet werden kann. Prinzipiell können die Sensoren eingepflegt werden, was jedoch im Rahmen dieser Arbeit aus zeitlichen Gründen nicht erfolgt ist. Daraus resultiert der Nachteil, dass die Simulation nur beschränkt Schlüsse auf die Realität zulässt. Nichtsdestotrotz ergeben sich zwei wichtige Vorteile: Einerseits können die verschiedenen Konfigurationen des ROS-Navigation-Stack anhand der Simulation erprobt und auf ihre Richtigkeit überprüft werden, bevor sie auf reale Anwendungsfälle übertragen werden. Andererseits können im Rahmen der Simulation die Komponenten des Navigation-Stack vollkommen entkoppelt werden. Beispielsweise hängen die Ergebnisse der Navigationsalgorithmen stark von der Qualität der Lokalisierung ab, weshalb die Planungs- und Lokalisierungsproblematik in einem realen Umfeld nicht getrennt untersucht werden können. In der Simulation können Positionsdaten jedoch unmittelbar abgegriffen und der Navigation zur Verfügung gestellt werden. Insofern stellt GAZEBO ein mächtiges Werkzeug für die schrittweise Inbetriebnahme des Navigation-Stacks dar.

MATLAB kann im Rahmen der Robotics-Toolbox ebenfalls als ROS-Node betrieben werden, wodurch eine Schnittstelle zwischen MATLAB und ROS geschaffen wird. Hier bringt MATLAB den Vorteil mit sich, dass es für die Implementierung von Al-

gorithmen oftmals besser geeignet ist als C++ oder auch Python. Insbesondere die Möglichkeiten im Bereich der Visualisierung von Karten und Plots erweisen sich als mächtige Werkzeuge bei der Implementierung und Erprobung von Algorithmen. Außerdem können simple Anwendungsszenarien, wie sie zum Beispiel im Rahmen der diskreten Planung auftreten, vollständig mit MATLAB simuliert werden.

Als letztes Werkzeug sei an dieser Stelle RViz genannt, das als ROS-Paket vorliegt und für die Visualisierung von Robotikanwendungen konzipiert wurde. Mithilfe von RViz können sowohl Karten als auch Roboter und Pfade graphisch dargestellt werden. Da RViz für die Anwendung mit ROS entwickelt wurde, stehen vollständige Konfiguration für den Einsatz mit der autonomen Navigation zur Verfügung, die Out-of-the-Box genutzt werden können.

Aufbau des ROS-Navigation-Stack

Der ROS-Navigation-Stack ist das Ergebnis mehrerer zusammengeführter ROS-Pakete für die autonome Navigation von mobilen Robotern. Das Kernpaket, in dem die Navigation implementiert ist, heißt **move_base**. Die Kernaufgabe des Navigation-Stack besteht darin, eine Weg zwischen Start- und Zielposition zu planen und abzufahren, wofür auf eine Karte der Umgebung zurückgegriffen wird.

Das Planungskonzept des Navigation-Stack verfolgt einen zweischichtigen Ansatz, wobei zwischen einem globalen und einem lokalen Planer differenziert wird. Im ersten Schritt nutzt der globale Planer die vorliegende Karte, um einen ortsdiskreten Pfad zwischen Start- und Zielposition zu bestimmen. Der nächste Punkt des Pfads wird dem lokalen Planer übergeben, der wiederum Geschwindigkeiten berechnen soll, die den Roboter zu dem lokalen Ziel führen. Bei der lokalen Planung werden neben dem nächsten Punkt des globalen Pfades auch die Karte und aktuelle Sensorwerte beachtet. Dieses Vorgehen erinnert an das menschliche Verhalten. Vor einer Autofahrt planen wir anhand einer Karte die Route zum Ziel, was der Aufgabe des globalen Planers entspricht. Hier handelt es sich um eine ortsdiskrete Darstellung, da einzelne Punkte wie Auf- und Abfahrten zu Autobahnen genügen, um den Weg festzulegen. An diesem Zeitpunkt machen wir uns auch noch keine Gedanken, ob wir die Spur wechseln sollen, um den nächsten LKW zu überholen. Derartige Probleme fallen in den Aufgabenbereich des lokalen Planers. Sobald wir uns auf der Reise befinden, und das nächste lokale Ziel - wie zum Beispiel die kommende Autobahnabfahrt - anstreben, planen wir konkrete Überholmanöver. Dennoch können unerwartete Hindernisse, die sich erst während der Fahrt auftun, unsere globalen Pläne über den Haufen werfen. Beispielsweise ziehen wir im Angesicht eines Staus gerne eine Alternativroute in Erwägung. Diese Form der Planänderung wird im Navigation-Stack realisiert, indem einerseits die globale Planung zyklisch wiederholt wird, andererseits werden dabei auch aktuelle Sensorwerte beachtet. So werden Distanzmessungen in die Karte projiziert und als zusätzliche Hindernisse

interpretiert, die vom Pfad möglichst gemieden werden sollen.

Wie am Beispiel der Karte und Sensordaten ersichtlich wird, benötigt der Navigation-Stack weitere Informationen, die ihm von außen zur Verfügung gestellt werden müssen. Diese Aufgaben werden mithilfe von weiteren ROS-Paketen erfüllt. Als erstes sei das Paket **map_server** genannt, der die im Voraus aufgezeichnete Karte unter der Topic **\map** veröffentlicht. Daraufhin transformiert das Paket **costmap_2d** die Umgebungskarte in eine sogenannte Kostenkarte, die jeder Zelle einen mit ihrer Durchquerung verbundenen Aufwand zuordnet. Bei der Transformation können zusätzliche Informationen wie zum Beispiel Sensorwerte beachtet werden, wodurch auch Hindernisse, die nicht auf der Karte verzeichnet sind, die Planung beeinflussen. Innerhalb des Navigation-Stacks wird sowohl für den lokalen als auch den globalen Planer eine separate Kostenkarte generiert. Als letzte wichtige Zusatzinformation ist die Position des Roboters zu nennen, die über zwei relevante Wege bestimmt werden kann. Der einfachste Vorgehensweise nutzt die Odometriedaten, um die Position des Roboters zu berechnen. Einen elaborierteren Ansatz stellt die AMC-Lokalisierung dar, die stochastische Bewegungs- und Messmodelle nutzt, um anhand bekannter Informationen die Position des Roboters zu schätzen. Diese Lokalisierungsvariante wird in dem ROS-Paket **amcl** implementiert, das simultan zur Navigation ausgeführt wird.

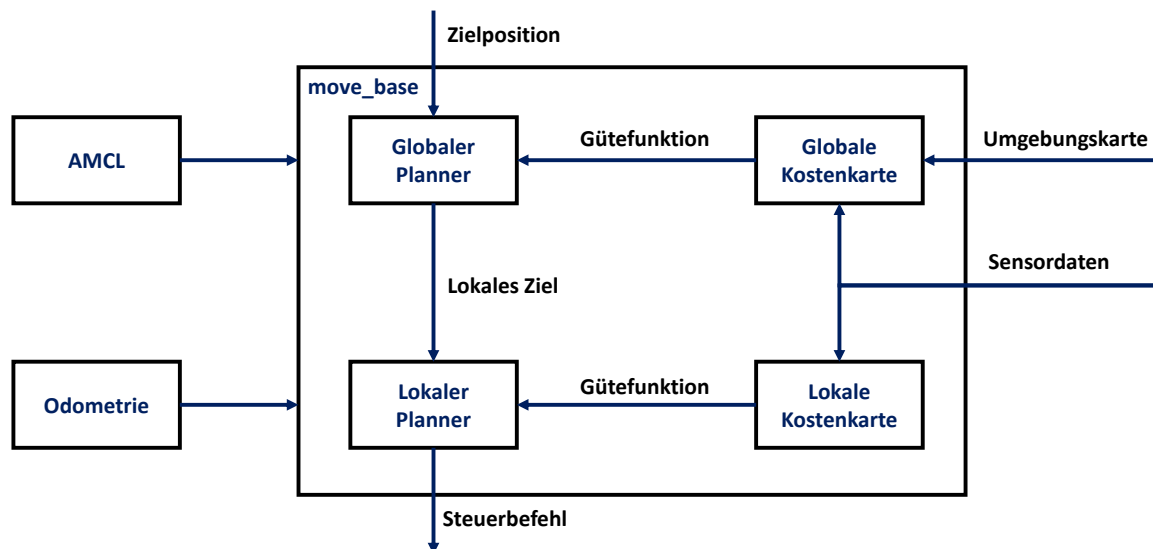


Abbildung 2.2: Übersicht des Navigationstack, Inhalt teilweise aus [21]

Der nächste Schritt besteht jetzt darin, die einzelnen Teile des Navigation-Stack getrennt zu betrachten und deren zugrundeliegenden Algorithmen zu verstehen. Als erstes wird der globale Planer untersucht, da er einerseits den Ausgangspunkt der Navigation darstellt, andererseits handelt es sich auch um den einzig deterministischen Algorithmus, wodurch er eine Sonderstellung einnimmt. Die restlichen Algorithmen basieren auf stochastischen Konzepten, weshalb anschließend die Rolle der stochastischen Weltanschauung in der Robotik diskutiert wird. Zur Illustration wird sowohl ein stochastisches Bewegungs- als auch Messmodell hergeleitet, die später bei der Loka-

lisierung und Bewegungsplanung wieder zum Einsatz kommen. Den Ausgangspunkt der meisten Algorithmen stellt das Bayes-Filer dar, das im Folgenden hergeleitet wird und am Beispiel einer simplen Kartographierung illustriert wird. Im nächsten Schritt werden fortgeschrittenere Anwendungen im Rahmen der Lokalisierung betrachtet. Als letzter Bestandteil bleibt der lokale Planer, dessen Dynamic Window Approach zuletzt erläutert wird.

Kapitel 3

Funktionsprinzip des globalen Planers

Wie bereits erwähnt, wird die Navigationsaufgabe mithilfe zweier sukzessiver Planungsalgorithmen gelöst. Im ersten Schritt berechnet ein globaler Planer einen Weg zwischen der Ausgangs- und Zielposition, wofür die vorgegebene Karte herangezogen wird. Als Planungsalgorithmus werden im Navigation-Stack entweder Dijkstra- oder der A*-Algorithmus verwendet, die an dieser Stelle näher untersucht werden. Prinzipiell besteht die Aufgabe darin, einen Weg und Trajektorie zu ermitteln, die den Ausgangszustand des Roboters mit dem gewünschten Endzustand verbinden. Bei den gesuchten Pfaden handelt es sich um zeit- und ortskontinuierliche Kurven, woraus ein kontinuierlicher Raum von Lösungen resultiert, der nach einer optimalen durchsucht werden muss. Die Kontinuität des Suchraums stellt auf der Planungsseite eine immense Herausforderung dar, da in diesem Fall Gütefunktionale optimiert werden müssen. Um dieser Komplexität Herr zu werden, wird der Suchraum diskretisiert und die Planungsaufgabe auf die Bestimmung einer Positionsfolge reduziert.

Allerdings wird die Diskretisierung des Raumes von mehr als einer reinen Vereinfachung des Suchproblems motiviert: Der globale Planer übernimmt lediglich den ersten Schritt bei der Berechnung der letztendlichen Trajektorie. Der lokale Planer ermittelt anhand der globalen Sollkurve eine lokale Trajektorie, die in Form von Stellgrößen realisiert wird. Dabei werden neben der globalen Ortskurve auch aktuelle Sensor- und Lokalisierungsinformationen miteinbezogen, welche unter Umständen dazu führen können, dass der Roboter von der global geplanten Kurve abweicht. Insofern ergibt es wenig Sinn, bei der globalen Planung Ressourcen in eine unnötig genaue Trajektorie zu investieren; besonders dann, wenn noch nicht alle nötigen Informationen vorliegen. Für den Anfang genügt ein ungenauer, fehlerbehafteter Pfad, der die Anfangs- und Zielposition verbindet, wofür eine diskrete Darstellung des Suchraums vollkommen genügt. Des Weiteren stimmt die Forderung nach einer diskreten Darstellung des Suchraums mit den üblichen Darstellungsformen von metrischen Karten überein, denn diese repräsentieren

die Umgebung in Form von diskreten Zellen.

Im Rahmen dieser Überlegung werden die vereinfachten Bedingungen zunächst in einer mathematischen Darstellung formuliert. Es wird angenommen, dass der Roboter in einer planen Umgebung manövriert, welche mittels einer zweidimensionalen Karte dargestellt werden kann. Hierfür wird eine metrische Karte verwendet, die den Raum in quadratische Zellen fixer Größe unterteilt. Die Positionen von Objekten innerhalb der Karten werden mithilfe des X- und Y-Indexes angegeben, wodurch die Positionsangaben von der Zellengröße entkoppelt werden. An dieser Stelle kann entweder ein deterministisches oder stochastisches Modell verwendet werden. Bei Erstem wird die Karte als Menge

$$M = \{m_{ij}\} \quad m_{ij} \in \{0, 1\} \quad (3.1)$$

dargestellt, wobei jede Zelle m_{ij} im belegten Fall den Wert 1 und ansonsten den Wert 0 annimmt. Bei den so genannten Occupancy-Grids handelt es sich eine stochastische Repräsentation: Für jede Zelle wird die Wahrscheinlichkeit angegeben, ob diese belegt ist.

Die Planungsaufgabe wird durch die Angabe einer Startposition \vec{x}_0 und einer Zielposition \vec{x}_G definiert, wobei angenommen wird, dass sowohl \vec{x}_0 als auch \vec{x}_G freie Zellen sind. Zuletzt muss die Menge der möglichen Aktionen U spezifiziert werden. Hier wird eine weitere Vereinfachung vorgenommen. Die eigentlichen Stellgrößen des Roboters sind dessen Translations- und Rotationsgeschwindigkeit, welche allerdings auf zeit- und ortskontinuierliche Trajektorien führen. Da lediglich ortsdiskrete Pfade berechnet werden sollen, ergibt es Sinn, diskrete Positionsänderungen als Aktionen zu definieren. Beispielsweise kann die Verrückung um eine Zelle entlang der X- und Y-Richtung zugelassen werden. Da die Positionen in Form von Zellenindizes beschrieben werden, kann die Verrückung durch die Addition eines Vektors ausgedrückt werden, woraus die Aktionsmenge

$$U = \left\{ \vec{u}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \vec{u}_2 = \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \vec{u}_3 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \vec{u}_4 = \begin{bmatrix} 0 \\ -1 \end{bmatrix} \right\} \quad (3.2)$$

resultiert. Von einem beliebigen Zustand \vec{x}_n ausgehend, kann mithilfe der Übergangsfunktion

$$\vec{x}_{n+1} = f(\vec{x}_n, \vec{u}) = \vec{x}_n + \vec{u} \quad (3.3)$$

der durch die Aktion \vec{u} erreichte Zustand \vec{x}_{n+1} ermittelt werden.

3.1 Forward-Search: Breitensuche



Wenn die Ausgangsposition \vec{x}_0 , die Zielposition \vec{x}_G , die Menge der zulässigen Zustände X , welche durch die Karte M definiert wird, und eine Aktionsmenge U zur Verfügung, können erste rudimentäre Planungsalgorithmen entwickelt werden. Die Aufgabe besteht darin eine Sequenz von Aktionen oder Zuständen zu bestimmen, welche die Positionen \vec{x}_0 und \vec{x}_G miteinander verbinden. Dabei spielt es keine Rolle, ob Aktionen oder wegzusammenhängende Zustände berechnet werden, da beide Darstellungsformen des Pfades ineinander überführt werden können. Als erster Ansatz für die Lösung des Planungsproblems kann die so genannte Forward-Search nach [2, S. 28] verfolgt werden:

Listing 3.1: Ablauf der Forward-Search in Pseudocode

```

1  Q.Insert( $\vec{x}_0$ ) and mark  $\vec{x}_0$  as visited
2  while Q not empty do
3     $\vec{x}_n = Q.GetFirst()$ 
4    if  $\vec{x}_n = \vec{x}_G$ 
5      return SUCCESS
6    forall  $\vec{u} \in U$ 
7       $\vec{x}_{n+1}$  not visited
8      Mark  $\vec{x}_{n+1}$  as visited
9      Q.Insert( $\vec{x}_{n+1}$ )
10 return FAILURE

```

Listing 3.1: Ablauf der Forward-Search in Pseudocode

Der Algorithmus basiert auf einer Datenstruktur Q , deren Funktionsprinzip die Charakteristik des Suchalgorithmus maßgeblich prägt. Bei der Initialisierung des Algorithmus wird die Anfangsposition \vec{x}_0 in die Datenstruktur eingefügt. Anschließend wird in einer Schleife die Struktur ausgelesen, wo zuerst geprüft wird, ob der aktuelle Zustand \vec{x}_n der gesuchten Zielposition \vec{x}_G gleichkommt; gegebenenfalls terminiert die Suche. Wurde das Ziel noch nicht erreicht, werden alle möglichen Aktionen \vec{u} auf den aktuellen Zustand \vec{x}_n angewandt, wodurch jeweils ein neuer Zustand \vec{x}_{n+1} erzeugt wird. Falls der neue Zustand \vec{x}_{n+1} noch nicht geprüft wurde, wird er ebenfalls in die Datenstruktur Q eingefügt. Dadurch wird der Suchraum der möglichen Zustände systematisch durchsucht, wobei eine mehrmalige Prüfung des gleichen Zustandes durch eine Markierung verhindert wird. Wenn die Datenstruktur Q keine Elemente enthält, wurden alle möglichen Zustände geprüft; folglich kann das gewünschte Ziel nicht von der gegebenen Anfangsposition erreicht werden.

Das Funktionsprinzip der Datenstruktur Q gibt die Reihenfolge vor, in der die neu erschlossenen Zustände überprüft werden, wodurch die Suchstrategie festgelegt wird. Arbeitet Q nach dem FIFO-Prinzip, wird der Suchraum zuerst in der Breite und anschließend in der Tiefe erforscht. Als Beispiel dient eine beliebige Startposition \vec{x}_0 und die bereits genannte Aktionsmenge

$$U = \left\{ \vec{u}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \vec{u}_2 = \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \vec{u}_3 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \vec{u}_4 = \begin{bmatrix} 0 \\ -1 \end{bmatrix} \right\}. \quad (3.2)$$

Die folgenden Zustände ergeben sich nach

$$\vec{x}_{n+1} = f(\vec{x}_n, \vec{u}) = \vec{x}_n + \vec{u}. \quad (3.3)$$

Folglich entspricht die Menge der am Iterationsschritt 1 möglichen Zustände dem Abbild

$$X_1 = f(\vec{x}_0, U) = \{\vec{x}_0 + \vec{u} \mid \vec{u} \in U\}. \quad (3.4)$$

Analog kann die Menge der möglichen Zustände an einem beliebigen Schritt $n + 1$ durch das Abbild

$$X_{n+1} = f(X_n, U) \quad (3.5)$$

bestimmt werden. Angenommen die Suche wird von einem beliebigen Startzustand \vec{x}_0 begonnen, so wird im ersten Schleifendurchlauf das Abbild X_1 bestimmt und in die Datenstruktur eingefügt. Arbeitet diese nun nach dem FIFO-Prinzip, so werden in den folgenden Durchläufen die Elemente der Menge X_1 bearbeitet, wodurch die Menge X_2 berechnet und in die Datenstruktur eingefügt werden. Da die Elemente von X_2 aber nach denen von X_1 in die Schlange eingefügt werden, wird sichergestellt, dass zuerst alle Elemente von X_1 geprüft werden, bevor die Bearbeitung von X_2 stattfindet. Aus diesem Grund wird diese Suchstrategie als Breadth-First bezeichnet, da zuerst alle Möglichkeiten eines Iterationsschrittes geprüft werden, bevor die nachfolgende Iteration in Erwägung gezogen wird. Stellt man den Suchraum als Baum dar, wird die Bedeutung der Begriffe Tiefe und Breite deutlich. Mit der Breadth-First-Search liegt

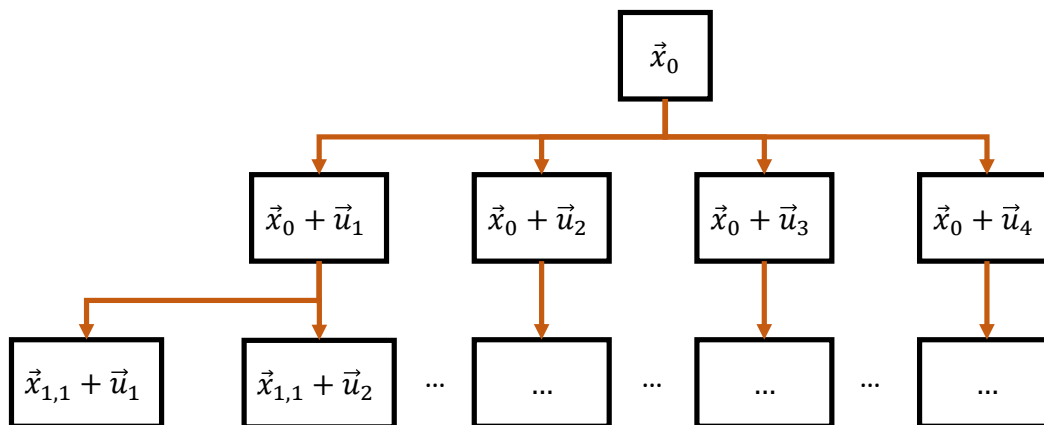


Abbildung 3.1: Suchraum als Baum dargestellt

ein systematischer Ansatz vor, der einen Pfad zum Ziel findet, falls ein solcher existiert. Außerdem ist recht leicht ersichtlich, dass der Algorithmus immer auf einen möglichst kurzen Lösungspfad führt. Um das Konzept zu illustrieren, wird folgendes Beispiel betrachtet: Ein Roboter befindet sich in einem Korridor an einer bekannten Position \vec{x}_0 .

wirkungen einer veränderten Aktionsreihenfolge anhand des Korridorbeispiels, wobei die Reihenfolge der Aktionen variiert wurde. Prinzipiell kann die Suche - unabhängig

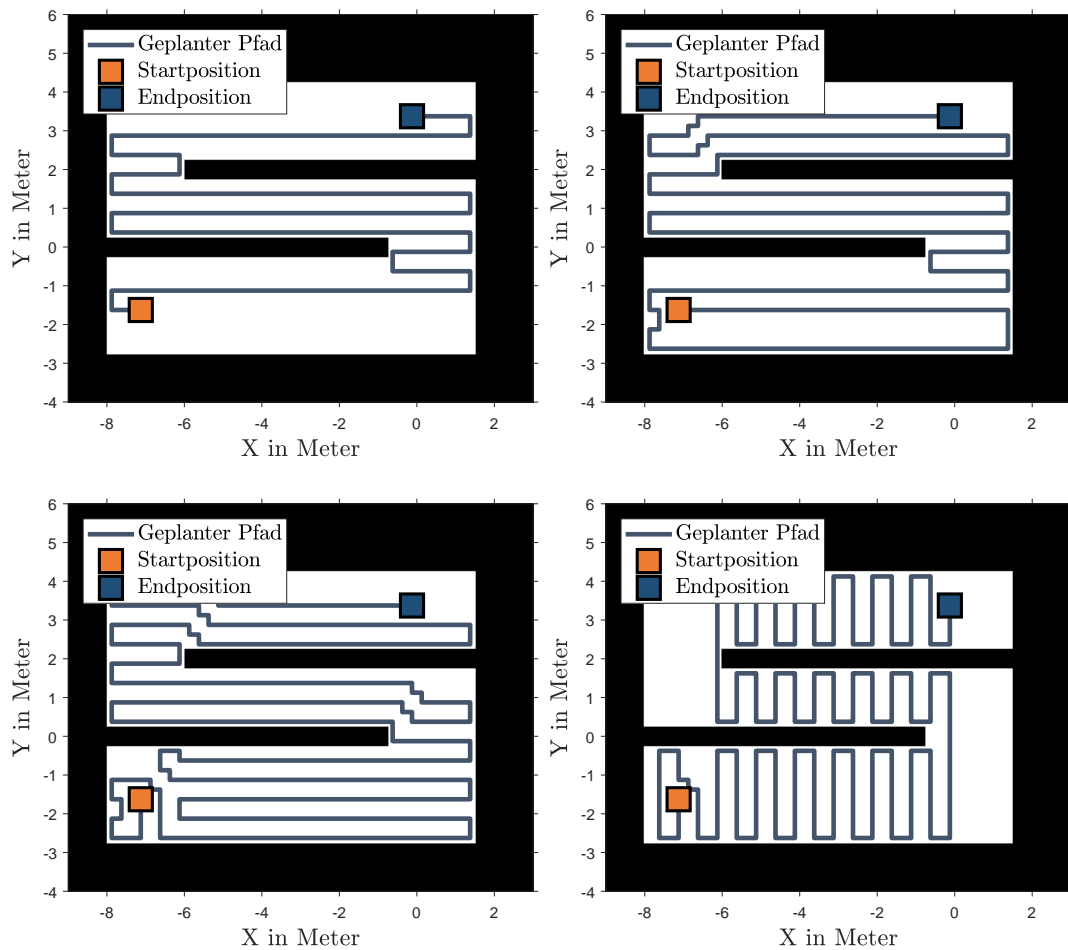



Abbildung 3.3: Ergebnisse der Tiefensuche bei unterschiedlichen Reihenfolge der Aktionen

von der gewählten Strategie - auch von der Zielposition ausgehend beginnen. In diesem Fall wird von einer so genannten Backwards-Search gesprochen. Werden Forward- und Backward-Search simultan verfolgt, resultiert die so genannten Bidirectional-Search. Diese Adaptionen können unter Umständen verkürzte Suchzeiten liefern, werden an dieser Stelle aber nicht weiterverfolgt, da dieser Aspekt bei dem hiesigen Anwendungsfall nicht zum Tragen kommt.

3.3 Optimale Suche und Planung

In den bisherigen Suchalgorithmen bestand die Aufgabe lediglich darin,  eine Pfad zwischen Ausgangs- und Zielposition zu finden. Dies hatte unter anderem zur Folge, dass starke Unterschiede zwischen den Lösungen der verschiedenen Ansätze resultierten. Besonders bei der Depth-First-Search kamen zum Teil äußerst umständliche Pfade zu Stande. Aus diesem Grund soll im nächsten Schritt eine Bewertung der Lösungsmöglichkeiten eingeführt werden, die genutzt wird, um einen optimalen Pfad im Sinne

dieses Gütekriteriums zu ermitteln. Bei der globalen Pfadplanung sollen prinzipiell zwei Aspekte beachtet werden: Einerseits soll der resultierende Pfad möglichst kurz sein, andererseits soll ein gewisser Abstand zu belegten Zellen der Karte gehalten werden bzw. die belegten Zellen sollen von dem Pfad nicht durchkreuzt werden. Um diese Vorgaben in einem Gütekriterium auszudrücken, wird die Karte in eine sogenannte Kostenkarte transformiert. Diese ordnet jeder Zelle einen Aufwand zu, wodurch es möglich wird verschiedenen Pfade zu bewerten und miteinander zu vergleichen. Die Kosten eines Pfades ergeben sich aus der Summe der enthaltenen Zellen. Ein Beispiel für eine mögliche Bewertung der Zellen ordnet jeder freien Zelle den Aufwand 1 und jeder belegten Zelle den Wert ∞ zu. Durch letzteren wird sichergestellt, dass belegte Zellen vermieden werden. Die Gewichtung von freien Zellen führt dazu, dass kürzere Pfade mit geringeren Kosten verbunden sind. Eine Möglichkeit, um einen im Kontext der Kostenkarte optimalen Pfad zu berechnen, stellt Dijkstras Algorithmus dar. Dieser gleicht im Grundprinzip den bisher betrachteten Vorgehensweise, mit dem Unterschied, dass die Datenstruktur Q so sortiert wird, dass das Element mit den aktuell geringstem Kostenaufwand am Anfang der Liste steht. Zusätzlich muss in dem Fall, dass ein Zustand geprüft wird, der bereits besucht worden ist, verglichen werden, welcher der beiden Pfade den geringeren Aufwand benötigt.

Listing 3.2: Dijkstra's Algorithmus in Pseudocode

```

1  Q.Insert( $\vec{x}_0$ ) and mark  $\vec{x}_0$  as visited
2  while  $Q$  not empty do
3     $\vec{x}_n = Q.GetFirst()$ 
4    if  $\vec{x}_n = \vec{x}_G$ 
5      return SUCCESS
6    forall  $\vec{u} \in U$ 
7      if  $\vec{x}_{n+1}$  not visited
8        Mark  $\vec{x}_{n+1}$  as visited
9         $Q.Insert(\vec{x}_{n+1})$ 
10   else
11     resolve duplicate
12 return FAILURE

```

Die folgende Abbildung zeigt die Ergebnisse der Suche bei dem bekannten Anwendungsbeispiel.

3.3.1 A*

Indem bei Dijkstras Algorithmus die Liste Q nach aufsteigenden Kosten sortiert wird, kann gesichert werden, dass die Suche zu einem optimalen Ergebnis führt. Allerdings bringt dieses Vorgehen einen zum Teil unnötigen Suchaufwand mit sich, was recht leicht illustriert werden kann: Um möglichst kurze Pfade zu erhalten, wurde jeder weitere Schritt mit einem fixen Aufwand bewertet. Dies hat zur Folge, dass die kürzesten Pläne an den Anfang der Liste rücken, inklusive der Pfade, die zu kurz sind, um das Ziel überhaupt zu erreichen und insbesondere solche die in die falsche Richtung lau-

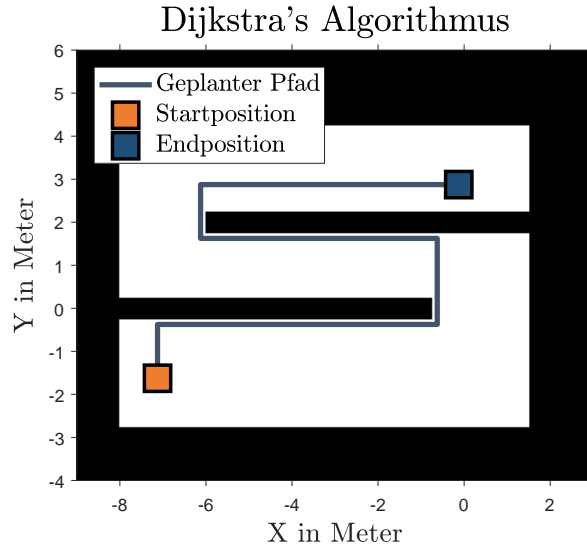


Abbildung 3.4: Ergebnis mit Dijkstra's Algorithmus

fen, aber verhältnismäßig kurz sind. Insofern erzwingt die sortierte Datenstruktur eine Breitensuche, da kurze Pfade fälschlicherweise - zumindest zum Teil fälschlicherweise - mit weniger Aufwand verbunden sind.

Ein Ansatz, um dieser Problematik vorzubeugen, stellt der A^* Algorithmus dar, der nach dem identischen Grundprinzip wie Dijkstra's Algorithmus arbeitet. Lediglich das Gewichtungskonzept der Pläne unterscheiden die beiden Vorgehensweisen. Für jede Position \vec{x}_n existiert ein optimaler Pfad, welcher \vec{x}_n mit der Zielposition \vec{x}_G verbindet. Die Kosten des optimalen Pfades werden von der Funktion $G^*(\vec{x}_n)$ beschrieben. Die Idee des A^* Algorithmus besteht darin, neben den Kosten des Pfades von \vec{x}_0 zu \vec{x}_n auch die verbleibenden Kosten nach \vec{x}_G zu beachten. Dadurch werden kürzere Pfade nicht mehr aus Prinzip bevorzugt, sondern lediglich dann, wenn ihre erwarteten Kosten bis zum Ziel ebenfalls gering sind. Da die optimalen Kosten $G^*(\vec{x}_n)$ nicht bekannt sind müssen sie approximiert werden, wofür eine heuristische Schätzung $G(\vec{x}_n)$ eingeführt wird. An die Heuristik ist die Bedingung geknüpft, dass die geschätzten Kosten $G(\vec{x}_n)$ stets kleiner oder gleich der optimalen Kosten $G^*(\vec{x}_n)$ sind:

$$G(\vec{x}_n) \leq G^*(\vec{x}_n) \quad \forall \vec{x}_n \in X. \quad (3.7)$$

Als Beispiel wird wieder die Navigation durch den Korridor betrachtet. Hier wurde jede Verrückung mit dem Aufwand 1 gewichtet, woraus folgt, dass die Kosten zwischen \vec{x}_n und \vec{x}_G mindestens gleich der absoluten Differenzen zwischen den X- und Y-Koordinaten der beiden Positionen sein muss. Im Falle, dass ein Hindernis den direkten Weg zwischen \vec{x}_n und \vec{x}_G versperrt nehmen die optimalen Kosten $G^*(\vec{x}_n)$ weiter zu, weshalb

$$G(\vec{x}_n) = |x_n - x_G| + |y_n - y_G| \leq G^*(\vec{x}_n) \quad \forall \vec{x}_n \in X \quad (3.8)$$

stets gilt. Somit stellt $G(\vec{x}_n)$ eine legitime Approximation der optimalen Kosten $G^*(\vec{x}_G)$ dar. Wird diese Forderung an die Kostenschätzung eingehalten, kann bewiesen werden, dass der A^* Algorithmus stets eine optimale Lösung findet, insofern diese existiert [2, S. 32][5, 6]. Bei der - recht konservativen - Schätzung $G(\vec{x}_n) = 0$ geht das A^* Verfahren in Dijkstras Algorithmus über.

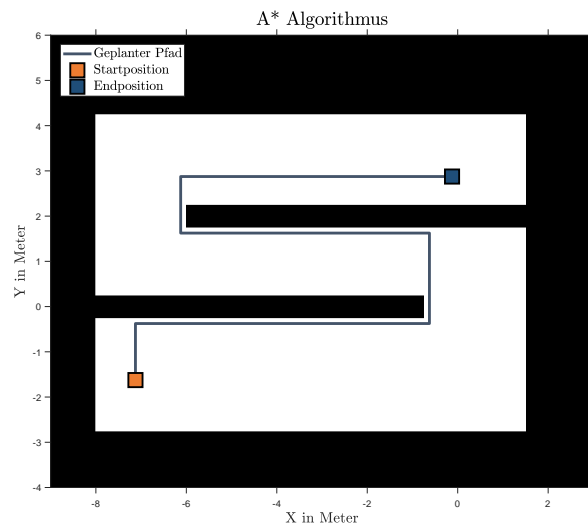


Abbildung 3.5: Ergebnis mit dem A^* Algorithmus



Kapitel 4

Stochastische Modelle in der Robotik

Wie alle Probleme der Robotik, kann auch die autonome Navigation im entferntesten Sinne als Interaktion eines Roboters mit seiner Umwelt aufgefasst werden. Anhand von gesammelten Informationen muss das System eine Entscheidung über seine zukünftigen Aktionen treffen, wobei ein entferntes Ziel ohne ungewollte Kollisionen angesteuert werden soll. Für diese Aufgaben spielen drei Größen eine fundamentale Rolle. Zunächst muss die Position des Roboters beachtet werden, welche in dem Positionsvektor $\vec{x}(t) \equiv \vec{x}_t$ erfasst wird. Mithilfe von Sensoren sammelt der Roboter Informationen über seine Umgebung, die in dem Messvektor \vec{z}_t zusammengefasst werden. Anhand der Mess- und Positionsvektoren wird über die nächste Aktion des Roboters entschieden, die in dem Steuervektor \vec{u}_t ausgedrückt wird. Die exakte Form der Positions-, Mess- und Steuervektoren hängt von dem gegebenen Anwendungsfall und gewählten Modellformen ab, die im Folgenden näher erläutert werden.

In anderen Fachgebieten, die sich mit der Planung von Steuersignalen bzw. -sequenzen beschäftigen - wie z.B. der Regelungstechnik -, haben sich modellbasierte Methoden bewährt. Zunächst wird auf Basis von physikalischen Gegebenheiten der Zusammenhang zwischen Steuer-, Zustands- und Messvektor hergeleitet, der anschließend genutzt wird, um eine Regelstrategie zu formulieren. Der resultierende Algorithmus berechnet die Stellgröße \vec{u}_t , wofür die aktuellen Mess- und Zustandsvektoren herangezogen werden. Insofern liegt es nahe modellbasierte Ansätze auch bei Problemen der Robotik zu verfolgen. Allerdings kommt dort die ungemeine Komplexität der Problemstellung zu tragen, die sich recht leicht am Beispiel der Navigation illustrieren lässt: Als erstes muss ein Modell für den Einfluss des Stellvektors \vec{u}_t auf den Positionsvektor \vec{x}_t erstellt werden. Bei mobilen Roboterplattformen handelt es sich um mechanische Systeme mit mehreren Freiheitsgraden, womit die analytische Modellbildung zwar möglich, jedoch mit einem beachtlichen Aufwand verbunden ist. Spätestens bei der Modellierung der Sensoren werden die Grenzen des Möglichen erreicht: Soll beispielsweise die Position


des Roboters mithilfe von Stereokameras erfasst werden kann praktisch kaum ein exaktes, deterministisches Modell für diesen Vorgang erfasst werden, da er von zu vielen unbekannten Einflussfaktoren betroffen ist. Zuletzt kann die Pfadplanung per Definition nicht anhand eines deterministischen Modells erfolgen, da der Roboter dynamischen Hindernissen ausweichen soll, deren Form und Bewegung in der Aufgabenstellung nicht näher spezifiziert werden.

Aus diesen Gründen haben sich in der Robotik stochastische Modellformen etabliert, wobei recht simple Ausgangsmodelle verwendet werden, die um Zufallsvariablen ergänzt werden, um die Ungenauigkeiten und Ungewissheiten des Modells zu repräsentieren. Das Ziel besteht nicht mehr darin, konkrete Aussagen über den Verlauf von Zustandsgrößen zu treffen, wie dies z.B. bei einer Zustandsraumdarstellung der Form

$$\vec{\mathbf{x}}(n+1) = \underline{\mathbf{A}} \cdot \vec{\mathbf{x}}(n) + \underline{\mathbf{B}} \cdot \vec{\mathbf{u}}(n) \quad (4.1)$$

erfolgt. Vielmehr soll mithilfe des Modells eine bedingte Wahrscheinlichkeit

$$p(\vec{\mathbf{x}}(n+1) \mid \vec{\mathbf{x}}(n), \vec{\mathbf{u}}(n)) \quad (4.2)$$

berechnet werden. Im Anschluss können die Methoden der Wahrscheinlichkeitstheorie genutzt werden, um Filter- und Planungsalgorithmen zu entwerfen.  Um einen ersten Eindruck für diese Modellformen zu erhalten, werden im Anschluss rudimentäre Ansätze für ein Bewegungs- und Sensormodell vorgestellt.

4.1 Geschwindigkeitsbasiertes Bewegungsmodell

¹ Als erstes Beispiel wird ein stochastisches Modell für die Roboterbewegung entworfen, wobei anhand des vergangenen Positionsvektor \vec{x}_{t-1} und des aktuellen Steuervektors \vec{u}_t die Wahrscheinlichkeitsverteilung des aktuellen Positionsvektors \vec{x}_t

$$p(\vec{x}_t | \vec{u}_t, \vec{x}_{t-1}) \quad (4.3)$$

bestimmt werden soll. In diesem Fall wird lediglich eine planare Bewegung betrachtet, das heißt der Roboter bewegt sich in der xy-Ebene. Der Positionsvektor setzt sich somit aus den drei Größen

$$\vec{x} = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix} \quad (4.4)$$

zusammen, welche die x-/y-Position und Ausrichtung des Roboters wiedergeben. θ gibt dabei den Winkel zwischen der x-Koordinatenachse und der Blickrichtung des Roboters an. Der Steuervektor \vec{u} gibt die aktuelle Translations- und Rotationsgeschwindigkeit

$$\vec{u} = \begin{pmatrix} v \\ \omega \end{pmatrix} \quad (4.5)$$

des Roboters an, wobei angenommen wird, dass die beiden Geschwindigkeiten zwischen zwei Abtastpunkten t und $t + 1$ konstant sind. v beschreibt die Translationsgeschwindigkeit in Blickrichtung, während ω die Änderung des Blickwinkels θ wiedergibt. Unter der Annahme dass die Geschwindigkeiten \vec{u} in dem Intervall $]t - 1, t]$ zwischen zwei Abtastpunkten konstant bleibt, kann die Bewegung als Rotation um einen konstanten Momentanpol $\vec{c} = (x_c \ y_c)^T$ betrachtet werden.

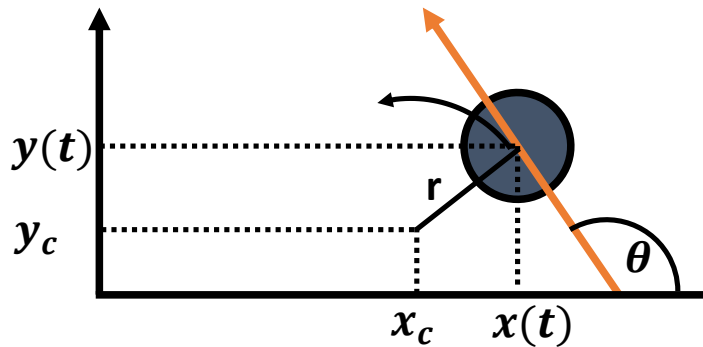


Abbildung 4.1: Darstellung des Momentanpols am Zeitpunkt t [1, S. 126]

Für den Radius gilt

$$r = \left| \frac{v}{\omega} \right|, \quad (4.6)$$

¹Das Bewegungsmodell und dessen Herleitung stammen aus [1, S. 121 ff]

wobei zu beachten ist, dass der Radius für eine Winkelgeschwindigkeit $\omega = 0$ gegen unendlich konvergiert, was wiederum einer reinen Translation entspricht. Aus dem Positionsvektors des Roboters \vec{x}_t an dem Zeitpunkt t kann der Momentanpol für die folgende Abtastperiode berechnet werden:

$$\begin{pmatrix} x_c \\ y_c \end{pmatrix} = \begin{pmatrix} x(t) - \frac{v}{\omega} \cdot \sin[\theta] \\ y(t) + \frac{v}{\omega} \cdot \cos[\theta] \end{pmatrix} \quad \leftrightarrow \quad \begin{pmatrix} x(t) \\ y(t) \end{pmatrix} = \begin{pmatrix} x_c + \frac{v}{\omega} \cdot \sin[\theta] \\ y_c - \frac{v}{\omega} \cdot \cos[\theta] \end{pmatrix}. \quad (4.7)$$

Im nächsten Schritt wird die Bewegung über eine Abtastperiode Δt betrachtet, wodurch der Roboter um die Winkeldifferenz $\Delta t \cdot \omega$ auf dem Kreisbogen wandert. Nach 4.7 folgt für den Positionsvektor am Zeitpunkt $t + \Delta t$

$$\begin{aligned} \begin{pmatrix} x(t + \Delta t) \\ y(t + \Delta t) \\ \theta(t + \Delta t) \end{pmatrix} &= \begin{pmatrix} x_c + \frac{v}{\omega} \cdot \sin[\theta(t) + \omega \cdot \Delta t] \\ y_c - \frac{v}{\omega} \cdot \cos[\theta(t) + \omega \cdot \Delta t] \\ \theta(t) + \omega \cdot \Delta t \end{pmatrix} \\ &= \begin{pmatrix} x(t) \\ y(t) \\ \theta(t) \end{pmatrix} + \begin{pmatrix} -\frac{v}{\omega} \cdot \sin[\theta(t)] + \frac{v}{\omega} \cdot \sin[\theta(t) + \omega \cdot \Delta t] \\ \frac{v}{\omega} \cdot \cos[\theta(t)] - \frac{v}{\omega} \cdot \cos[\theta(t) + \omega \cdot \Delta t] \\ \omega \cdot \Delta t \end{pmatrix}. \end{aligned} \quad (4.8)$$

Bisher wurden lediglich deterministische Bewegungen betrachtet. Um nun mögliche Fehler des Steuervektors \vec{u} zu beachten, werden die störbehafteten Geschwindigkeiten

$$\vec{u} = \begin{pmatrix} \hat{v} \\ \hat{\omega} \end{pmatrix} = \begin{pmatrix} v + v_{\text{err}} \\ \omega + \omega_{\text{err}} \end{pmatrix} \quad (4.9)$$

eingeführt. Die Zufallsvariablen v_{err} und ω_{err} dienen der Fehlermodellierung und ihre Wahrscheinlichkeitsverteilungen ε_v und ε_ω werden dem Anwendungsfall nach angepasst. Einsetzen des stochastischen Steuervektors \vec{u} liefert

$$\begin{pmatrix} x(t + \Delta t) \\ y(t + \Delta t) \\ \theta(t + \Delta t) \end{pmatrix} = \begin{pmatrix} x(t) \\ y(t) \\ \theta(t) \end{pmatrix} + \begin{pmatrix} -\frac{\hat{v}}{\hat{\omega}} \cdot \sin[\theta(t)] + \frac{\hat{v}}{\hat{\omega}} \cdot \sin[\theta(t) + \hat{\omega} \cdot \Delta t] \\ \frac{\hat{v}}{\hat{\omega}} \cdot \cos[\theta(t)] - \frac{\hat{v}}{\hat{\omega}} \cdot \cos[\theta(t) + \hat{\omega} \cdot \Delta t] \\ \hat{\omega} \cdot \Delta t \end{pmatrix}. \quad (4.10)$$

In diesem Modell wurden lediglich zwei Zufallsvariablen eingeführt, um die Störung von drei Positionsvariablen zu modellieren. Aus diesem Grund entsteht eine ungewollte stochastische Abhängigkeit zwischen den Elementen des Positionsvektors $\vec{x}(t + \Delta t)$. Dieses Problem wird behoben, indem eine dritte Zufallsvariable

$$\gamma_{\text{err}} \equiv \hat{\gamma} \quad (4.11)$$

eingeführt wird, die zu einer zusätzlichen Störung der Orientierung $\theta(t + \Delta t)$ in Form

von

$$\begin{pmatrix} x(t + \Delta t) \\ y(t + \Delta t) \\ \theta(t + \Delta t) \end{pmatrix} = \begin{pmatrix} x(t) \\ y(t) \\ \theta(t) \end{pmatrix} + \begin{pmatrix} -\frac{\hat{v}}{\hat{\omega}} \cdot \sin[\theta(t)] + \frac{\hat{v}}{\hat{\omega}} \cdot \sin[\theta(t) + \hat{\omega} \cdot \Delta t] \\ \frac{\hat{v}}{\hat{\omega}} \cdot \cos[\theta(t)] - \frac{\hat{v}}{\hat{\omega}} \cdot \cos[\theta(t) + \hat{\omega} \cdot \Delta t] \\ \hat{\omega} \cdot \Delta t + \hat{\gamma} \cdot \Delta t \end{pmatrix} \quad (4.12)$$

führt. Die Aufgabe besteht jetzt darin, ein Bewegungsmodell in Form der bedingten Wahrscheinlichkeit

$$p(\vec{\mathbf{x}}_t | \vec{\mathbf{u}}_t, \vec{\mathbf{x}}_{t-1}) \quad (4.13)$$

zu formulieren. Das heißt es soll eine Funktion aufgestellt werden, die berechnet wie wahrscheinlich der Zustand $\vec{\mathbf{x}}_t$ auf den Zustand $\vec{\mathbf{x}}_{t-1}$ und die Aktion $\vec{\mathbf{u}}_t$ folgt. Dafür wird nach Gleichung (4.12) die Werte der stochastisch unabhängigen Zufallsvariablen v_{err} , ω_{err} und γ_{err} berechnet. Die gesuchte Wahrscheinlichkeit ergibt sich dann aus deren gemeinsamer Verteilung

$$p(\vec{\mathbf{x}}_t, \vec{\mathbf{u}}_t, \vec{\mathbf{x}}_{t-1}) = p(\vec{\mathbf{x}}_t | \vec{\mathbf{u}}_t, \vec{\mathbf{x}}_{t-1}) = \varepsilon_v(v_{\text{err}}) \cdot \varepsilon_\omega(\omega_{\text{err}}) \cdot \varepsilon_\gamma(\gamma_{\text{err}}). \quad (4.14)$$

Da die direkte Umformung von Gleichung (4.7) zur expliziten Darstellung der gesuchten Fehlergrößen zu einem unhandlichen Ergebnis führt, wird eine indirekte Berechnung über den Momentanpol gewählt. Im ersten Schritt werden die Koordinaten des Momentanpols $\vec{\mathbf{c}}$ berechnet, wofür sich nach [1, S. 130]²

$$\begin{pmatrix} x_c \\ y_c \end{pmatrix} = \begin{pmatrix} \frac{x+\tilde{x}}{2} + \mu(y - \tilde{y}) \\ \frac{y+\tilde{y}}{2} + \mu(\tilde{x} - x) \end{pmatrix} \quad \mu = \frac{1}{2} \frac{(x - \tilde{x})\cos[\theta] + (y - \tilde{y})\sin[\theta]}{(y - \tilde{y})\cos[\theta] - (x - \tilde{x})\sin[\theta]} \quad (4.15)$$

ergibt. Woraus sich sowohl der Rotationsradius

$$r_c = \sqrt{(x - x_c)^2 + (y - y_c)^2} \quad (4.16)$$

als auch der auf der Kreisbahn zurückgelegte Winkel

$$\Delta\theta = \text{atan}\left[\frac{\tilde{y} - y_c}{\tilde{x} - x_c}\right] - \text{atan}\left[\frac{y - y_c}{x - x_c}\right] \quad (4.17)$$

berechnen lassen. Mithilfe der der Winkeldifferenz lässt sich wiederum auf die zurückgelegte Strecke

$$\Delta s = r_c \cdot \Delta\theta \quad (4.18)$$

schließend, welche zusammen auf den gestörten Stellvektor

$$\vec{\mathbf{u}} = \begin{pmatrix} \hat{v} \\ \hat{\omega} \end{pmatrix} = \frac{1}{\Delta t} \cdot \begin{pmatrix} \Delta s \\ \Delta\theta \end{pmatrix} \quad (4.19)$$

²Schreibweise für $\vec{\mathbf{x}}_t = (x \ y \ \theta)^T$, $\vec{\mathbf{x}}_{t+\Delta t} = (\tilde{x} \ \tilde{y} \ \tilde{\theta})^T$

führen. Zuletzt fehlt der Orientierungsfehler $\hat{\gamma}$, der sich mithilfe von Gleichung (4.12) erschließen lässt.

$$\tilde{\theta} - \theta = \hat{\omega} \cdot \Delta t + \hat{\gamma} \cdot \Delta t \Leftrightarrow \hat{\gamma} = \frac{\tilde{\theta} - \theta}{\Delta t} - \hat{\omega}. \quad (4.20)$$

4.2 Messmodell

³ Als weiteres Beispiel wird ein stochastisches Messmodell vorgestellt, das später bei der Lokalisierung wiederverwendet wird. Das Ziel des Messmodells ist es den Zusammenhang zwischen einem Zustandsvektor \vec{x}_t und einem Messvektor \vec{z}_t in Form der bedingten Wahrscheinlichkeit

$$p(\vec{z}_t | \vec{x}_t) \quad (4.21)$$

herzustellen. Die Verteilung sagt aus, mit welcher Wahrscheinlichkeit ein Messvektor \vec{z}_t von einer vorgegebenen Position \vec{x}_t aus erzielt wird. Das Messmodell dient als Paradebeispiel dafür, wie heuristische Argumente in stochastische Modelle einfließen können. Unabhängig von dem exakten Sensor wird angenommen, dass es sich um ein auf Messstrahlen basiertes Messprinzip handelt. Das heißt der Messvektor \vec{z}_t setzt sich aus mehreren Messwerten z_t^k zusammen, die jeweils die auf einem Strahl gemessene Distanz wiedergeben.

Als erste Störgröße wird ein gewöhnliches Messrauschen eingeführt, das als normal verteilt angenommen wird. Der Mittelwert der Verteilung entspricht der tatsächliche Distanz auf dem Messstrahl, die mit z_t^{k*} denotiert wird. Als zweiter Parameter muss die Varianz σ_{hit} festgelegt werden, die angibt wie stark der Messwert von dem Rauschen beeinflusst wird. Somit ergibt sich als Modell für das Sensorrauschen

$$p_{\text{hit}}(z_t^k | \vec{x}_t) = \begin{cases} \eta \cdot \mathcal{N}(z_t^k, z_t^{k*}, \sigma_{\text{hit}}^2) & \forall z_t^k \in [0, z_{\text{max}}] \\ 0 & \forall z_t^k \notin [0, z_{\text{max}}] \end{cases} \quad (4.22)$$

Die Messwerte werden außerdem von unerwarteten Objekten beeinflusst. Angenommen es liegt eine Karte der Umgebung vor und es bewegt sich ein Mensch innerhalb des Raumes. Wenn der Mensch nun die Messstrahlen unterbricht, tritt ein geringerer Messwert als anhand der Karte erwartet ein. Im Allgemeinen gilt, dass im Falle von Störungen durch unerwartete Objekte die Messwerte lediglich kleiner, aber niemals größer werden. Des Weiteren nimmt die Wahrscheinlichkeit, dass ein Messwert von einem Objekt beeinflusst wird mit zunehmendem Messdistanz ab. Für das Modell wird eine exponentiell abfallende Verteilung

$$p_{\text{short}}(z_t^k | \vec{x}_t) = \begin{cases} \eta \cdot \lambda_{\text{short}} \cdot e^{-\lambda_{\text{short}} \cdot z_t^k} & \forall z_t^k \in [0, z_t^{k*}] \\ 0 & \forall z_t^k \notin [0, z_t^{k*}] \end{cases} \quad (4.23)$$

gewählt, wobei λ_{short} als wählbarer Parameter die Häufigkeit der Störungen widerspiegelt. Die Funktion p_{short} besagt, wie mit welcher Wahrscheinlichkeit es sich bei dem Messwert z_t^k um einen, auf Grund eines unerwarteten Hindernisses, zu kurze Messung handelt.

³Diser Abschnitt orientiert sich inhaltlich an [1, S. 153 ff]

Als dritte Ursache für Messfehler wird das sporadische Versagen des Sensors herangezogen. Derartige Phänomene können bei Laserscannern auftreten, wenn beispielsweise schwarze Oberflächen die Strahlen vollständig absorbieren. Ähnliche Probleme werden bei Sonarsensoren durch Interferenz oder schallabsorbierende Medien hervorgerufen. Unabhängig von dem Messprinzip des Sensors werden in all den beschriebenen Fällen - fälschlicherweise - maximale Distanzen gemessen, wodurch die Verteilung

$$p_{\max}(z_t^k | \vec{\mathbf{x}}_t) = \begin{cases} 1 & \forall z = z_{\max} \\ 0 & \forall z \neq z_{\max} \end{cases} \quad (4.24)$$

motiviert wird. Die Aussage der Zufallsvariable besteht darin, dass im Falle eines maximalen Messwertes von einem Messfehler ausgegangen werden kann

Als letzter Fall werden rein zufällig falsche Phänomene betrachtet. Als Argument dient die Beobachtung, dass es unerklärliche Messwerte gibt, die - zu Gunsten der Simplität - durch eine Gleichverteilung der Form

$$p_{\text{rand}}(z_t^k | \vec{\mathbf{x}}_t) = \begin{cases} \frac{1}{z_{\max}} & \forall z_t^k \in [0, z_{\max}] \\ 0 & \forall z_t^k \notin [0, z_{\max}] \end{cases} \quad (4.25)$$

repräsentiert werden.

Die gesuchte Wahrscheinlichkeit $p(z_t^k | \vec{\mathbf{x}}_t)$ wird aus Summe der vier Verteilung berechnet, wobei diese mit den Faktoren z_{hit} , z_{short} , z_{\max} und z_{rand} gewichtet werden. Die Gewichtung erlauben es dem Anwender, die Heuristiken nach ihrer Glaubhaftigkeit zu bewerten.

$$p(z_t^k | \vec{\mathbf{x}}_t) = \begin{bmatrix} z_{\text{hit}} & z_{\text{short}} & z_{\max} z_{\text{rand}} \end{bmatrix} \cdot \begin{bmatrix} p_{\text{hit}}(z_t^k | \vec{\mathbf{x}}_t) \\ p_{\text{short}}(z_t^k | \vec{\mathbf{x}}_t) \\ p_{\max}(z_t^k | \vec{\mathbf{x}}_t) \\ p_{\text{rand}}(z_t^k | \vec{\mathbf{x}}_t) \end{bmatrix}. \quad (4.26)$$

4.3 Kontinuierliches Bayes Filter

In der Robotik werden stochastische Filter genutzt, um anhand von gegebenen Information wie Mess- und Stellgrößen eine Schätzung über unbekannte Zustände des Systems zu treffen. Die Grundlage für derartige Methoden stellt das Bayes-Filter für kontinuierlich verteilte Zustände dar. Als Beispiel für die Herleitung und Illustration des Filters wird die Lokalisierungsaufgabe herangezogen, bei welcher die aktuelle Position \vec{x}_t des Roboters ermittelt werden soll. Die Schätzung stützt sich dabei auf die bisher gesammelten Messwerte $\vec{z}_{1:t}$ und Stellgrößen $\vec{u}_{1:t}$. Insofern kann die Lokalisierungsaufgabe als die Bestimmung der bedingten Wahrscheinlichkeit

$$p(\vec{x}_t \mid \vec{z}_{1:t}, \vec{u}_{1:t}) \quad (4.27)$$

reformuliert werden, die im Folgenden auch mittels der Definition

$$\text{bel}(\vec{x}_t) \equiv p(\vec{x}_t \mid \vec{z}_{1:t}, \vec{u}_{1:t}) \quad (4.28)$$

abgekürzt wird. Im Falle, dass der aktuelle Messwert \vec{z}_t bei der Verteilung nicht beachtet wird, gilt

$$\overline{\text{bel}}(\vec{x}_t) \equiv p(\vec{x}_t \mid \vec{z}_{1:t-1}, \vec{u}_{1:t}) . \quad (4.29)$$

Für die Bestimmung der Filtergleichungen wird auf Bayes Theorem zurückgegriffen:

$$\begin{aligned} \text{bel}(\vec{x}_t) &= p(\vec{x}_t \mid \vec{z}_t, \vec{z}_{1:t-1}, \vec{u}_{1:t}) = \frac{p(\vec{z}_t \mid \vec{x}_t, \vec{z}_{1:t-1}, \vec{u}_{1:t}) \cdot p(\vec{x}_t \mid \vec{z}_{1:t-1}, \vec{u}_{1:t})}{p(\vec{z}_t \mid \vec{z}_{1:t-1}, \vec{u}_{1:t})} \\ &= \mu \cdot p(\vec{z}_t \mid \vec{x}_t, \vec{z}_{1:t-1}, \vec{u}_{1:t}) \cdot p(\vec{x}_t \mid \vec{z}_{1:t-1}, \vec{u}_{1:t}) . \end{aligned} \quad (4.30)$$

Der Einfluss der bedingte Wahrscheinlichkeit $p(\vec{z}_t \mid \vec{x}_t, \vec{z}_{1:t-1}, \vec{u}_{1:t})$ wird dabei durch den Faktor μ ersetzt, der für die Normalisierung reserviert wird, um sicherzustellen, dass es sich bei $\overline{\text{bel}}(\vec{x}_t)$ um eine legitime Wahrscheinlichkeitsverteilung handelt. Eine weitere Vereinfachung basiert auf der Annahme, dass \vec{x}_t einen so genannten vollständigen Zustand darstellt, das heißt \vec{x}_t beinhaltet bereits alle vergangen Information. Aus diesem Grund verändern zusätzliche Informationen aus vergangen Mess- und Stellgrößen eine mit \vec{x}_t bedingte Wahrscheinlichkeit nicht, woraus

$$p(\vec{z}_t \mid \vec{x}_t, \vec{z}_{1:t-1}, \vec{u}_{1:t}) = p(\vec{z}_t \mid \vec{x}_t) \quad (4.31)$$

resultiert. Wird dieses Ergebnis in die ursprüngliche Gleichung eingesetzt, ergibt sich

$$\begin{aligned} \text{bel}(\vec{x}_t) &= \mu \cdot p(\vec{z}_t \mid \vec{x}_t) \cdot p(\vec{x}_t \mid \vec{z}_{1:t-1}, \vec{u}_{1:t}) \\ &= \mu \cdot p(\vec{z}_t \mid \vec{x}_t) \cdot \overline{\text{bel}}(\vec{x}_t) . \end{aligned} \quad (4.32)$$

Mithilfe von weiteren Annahmen kann auch die Wahrscheinlichkeit $\overline{\text{bel}}(\vec{x}_t)$ ermittelt werden, wofür diese zunächst erweitert wird.

$$\begin{aligned}\overline{\text{bel}}(\vec{x}_t) &= p(\vec{x}_t \mid \vec{z}_{1:t}, \vec{u}_{1:t}) \\ &= \int p(\vec{x}_t \mid \vec{x}_{t-1}, \vec{z}_{1:t-1}, \vec{u}_{1:t}) \cdot p(\vec{x}_{t-1} \mid \vec{z}_{1:t-1}, \vec{u}_{1:t}) d\vec{x}_{t-1}.\end{aligned}\quad (4.33)$$

Falls \vec{x}_{t-1} ebenfalls ein vollständiger Zustand ist kann eine Vereinfachung der Form

$$p(\vec{x}_t \mid \vec{x}_{t-1}, \vec{z}_{1:t-1}, \vec{u}_{1:t}) = p(\vec{x}_t \mid \vec{x}_{t-1}, \vec{u}_t) \quad (4.34)$$

durchgeführt werden. Wenn zusätzlich angenommen werden kann, dass \vec{u}_t zufällig gewählt wird und somit kein Zusammenhang zwischen \vec{x}_{t-1} und \vec{u}_t besteht, folgt

$$p(\vec{x}_{t-1} \mid \vec{z}_{1:t-1}, \vec{u}_{1:t}) = p(\vec{x}_{t-1} \mid \vec{z}_{1:t-1}, \vec{u}_{1:t-1}) = \text{bel}(\vec{x}_{t-1}). \quad (4.35)$$

Die Substitution der beiden Ergebnisse (4.34) und (4.35) in Gleichung (4.33) liefert

$$\overline{\text{bel}}(\vec{x}_t) = \int p(\vec{x}_t \mid \vec{x}_{t-1}, \vec{u}_t) \cdot \text{bel}(\vec{x}_{t-1}) d\vec{x}_{t-1} \quad (4.36)$$

und bildet zusammen mit Gleichung (4.32) die folgende Berechnungsvorschrift des Bayes-Filter.

```

1 Algorithm: BayesFilter(bel( $\vec{x}_{t-1}$ ),  $\vec{u}_t$ ,  $\vec{z}_t$ ) :
2   for all  $\vec{x}_t$  do:
3      $\overline{\text{bel}}(\vec{x}_t) = \int p(\vec{x}_t \mid \vec{x}_{t-1}, \vec{u}_t) \cdot \text{bel}(\vec{x}_{t-1}) d\vec{x}_{t-1}$ 
4      $\text{bel}(\vec{x}_t) = \mu \cdot p(\vec{z}_t \mid \vec{x}_t) \cdot \overline{\text{bel}}(\vec{x}_t)$ 
5   endfor
6   return  $\text{bel}(\vec{x}_t)$ 

```

Listing 4.1: Bayes-Filter

Applikation, Interpretation und kritische Anmerkungen

Die bisherige Untersuchung hat sich auf die Herleitung der Berechnungsvorschrift und deren mathematischen Gültigkeit beschränkt, weshalb an dieser Stelle die Ergebnisse auf die konkrete Problemstellung der Lokalisierung übertragen werden. Dafür werden zunächst die Argumente des Algorithmus betrachtet: Bei den Vektoren \vec{u}_t und \vec{z}_t handelt es sich um die aktuellen Stell- und Messwerte. Der Erstgenannte enthält für gewöhnlich die aktuelle Geschwindigkeit des Roboters. In \vec{z}_t werden die aktuellen Sensorinformationen zusammengefasst, wobei es sich in diesem Fall um Distanzmessungen handelt, die mit einem Laserscanner gewonnen wurden. Das dritte Argument enthält die Wahrscheinlichkeit $bel(\vec{x}_{t-1})$ am vorherigen Abtastzeitpunkt, die für den Fall \vec{x}_0 initialisiert werden muss. Bei der Wahl der Startverteilung wird auf a priori Kenntnisse und Heuristiken zurückgegriffen. Beispielsweise kann der Anwender anhand einer Karte Positionen, die von einem Hindernis belegt sind, vorab ausschließen. Liegen keinerlei a priori Kenntnisse über die Ausgangsposition des Roboters vor, wird die Wahrscheinlichkeit $bel(\vec{x}_0)$ als Gleichverteilung initialisiert.

Als letzter Bestandteil des Filters müssen die bedingten Wahrscheinlichkeiten $p(\vec{x}_t | \vec{x}_{t-1}, \vec{u}_t)$ und $p(\vec{z}_t | \vec{x}_t)$ diskutiert werden. Die Verteilung $p(\vec{x}_t | \vec{x}_{t-1}, \vec{u}_t)$ gibt an, wie wahrscheinlich eine Position \vec{x}_t ausgehend von der Position \vec{x}_{t-1} mit dem Stellvektor \vec{u}_t erreicht werden kann. Es liegt also ein stochastisches Bewegungsmodell vor. Im Gegensatz zu seinem deterministischen Pendant wird anhand der Position \vec{x}_{t-1} und Geschwindigkeit \vec{x}_t keine exakte Position \vec{x}_t berechnet, sondern eine probabilistische Aussage darüber getroffen, mit welcher Wahrscheinlichkeit die gegebene Position \vec{x}_t erreicht werden wird.

Eine ähnlich Rolle übernimmt die Wahrscheinlichkeit $p(\vec{z}_t | \vec{x}_t)$, die ein Messmodell darstellt. Eine deterministische Analogie besteht in der Ausgangsgleichung eines Zustandsraummodells

$$\vec{y} = C(\vec{x}), \quad (4.37)$$

bei der anhand des Zustandsvektors \vec{x} ein Ausgangsvektor \vec{y} berechnet wird, der sich in der Regelungstechnik für gewöhnlich aus den gemessenen Größen zusammensetzt. Ähnlich wie bei dem Bewegungsmodell soll bei einem stochastischem Messmodell keine exakte Berechnung des Messvektors \vec{z}_t durchgeführt werden - insbesondere weil dieser bereits vorliegt. Die Aufgabe besteht vielmehr darin, zu schätzen wie wahrscheinlich ein gemessener Vektor \vec{z}_t aus einem gegebenen Zustand \vec{x}_t hervorgeht. Beispielsweise lässt sich mit recht hoher Wahrscheinlichkeit schließen, dass der Roboter unmittelbar vor einem Hindernis steht, wenn alle Abstandsmessungen sehr kleine Werte liefern.

Aus der Sicht des Applikators stellen die Verteilungen $p(\vec{x}_t | \vec{x}_{t-1}, \vec{u}_t)$ und $p(\vec{z}_t | \vec{x}_t)$ nicht nur Modelle sondern die relevanten Stellschrauben dar, um die Ergebnisse und Qualität des Filteralgorithmus zu beeinflussen. Folglich besteht die Hauptaufgabe darin, das stochastische Bewegungs- und Messmodell an die gegebenen Rahmenbedingun-

gen anzupassen. Hier stellt sich die Frage, welche Form die Qualität eines stochastischen Modells annimmt. Aus der deterministischen Modellbildung ging bereits die Erkenntnis hervor, dass sich qualitativ hochwertige Modelle nicht durch einen unendlichen Detaillierungsgrad und der damit einhergehenden Genauigkeit auszeichnen. Das Gegenteil ist der Fall: Die hohe Kunst der Modellbildung besteht darin einen Kompromiss zwischen Detail und Aussagekraft zu finden. Detailgetreue Modellierungstiefe bringt als unvermeidlichen Nachteil Komplexität mit sich; eine Komplexität, welche die Handhabung des Modells ungemein erschwert und die Nutzung des Modells behindert. Diese Aspekte motivieren die Philosophie einer Modellierungsform, die sich auf die markanten Charakteristika eines Prozesses beschränkt. Die für die Aufgabenstellung relevanten Eigenschaften geben die erforderliche Detailierungstiefe des Modells vor.

In diesem Sinne stellt das stochastische Modell aus mehreren Gründen einen mächtigen Ansatz dar. Während bei deterministischen Modellen die Vereinfachungen und Vernachlässigungen lediglich in Form von Annahmen in der begleitenden Dokumentation artikuliert werden können, bietet ein probabilistisches Modell die Möglichkeit Ungenauigkeiten als Wahrscheinlichkeit auszudrücken. Ebenso können heuristische Argumente, die sich nur schwer in einem deterministischen Kontext formulieren lassen, mithilfe einer probabilistischen Abschätzung in das mathematische Modell aufgenommen werden. Dieses Argument kommt besonders dann zum Tragen, wenn eine Problemstellung betrachtet wird, für die keinerlei physikalischen Gesetze bestehen. Beispielsweise steht im Fall des Bewegungsmodells das elaborierte Feld der technischen Mechanik bereit, um exakte Bewegungsgleichungen der Roboterdynamik zu formulieren. Lediglich deren unhandliche Komplexität motiviert die Verwendung simpler stochastischer Modelle. Eine andere Situation ergibt sich im Falle des Messmodells: Angenommen es werden Stereokameras verwendet, um die Distanz der umgebenden Hindernisse zu ermitteln. Es ist praktisch unmöglich einen exakten Zusammenhang zwischen der Umgebung, dem Messprinzip der Stereokameras und dem resultierenden Messvektor herzustellen. Somit bleibt dem Anwender keine andere Wahl als das Modell anhand von Heuristiken zu konstruieren, die wiederum auf stark vereinfachten Modellen und Annahmen basieren.

Aus diesen Betrachtungen folgt bereits ein wichtiger Gesichtspunkt für die Beurteilung der Filtermethoden: Ein schlechtes Ergebnis kann unter Umständen nicht von dem Algorithmus sondern von den verwendeten Modellen verschuldet sein. Insofern muss Vorsicht bei der Auswertung und Beurteilung von Experimenten walten, um ein vorzeitiges Verwerfen potenter Ansätze zu vermeiden. Aus diesem Blickwinkel sollen nun auch die bei der Herleitung des Bayes-Filter getroffenen Annahmen betrachtet werden. Es kann recht leicht argumentiert werden, weshalb die Annahmen nicht zutreffen können.

Beispielsweise folgte aus der Vollständigkeit des Zustandes \vec{x}_t die Gleichung

$$p(\vec{z}_t \mid \vec{x}_t, \vec{z}_{1:t-1}, \vec{u}_{1:t}) = p(\vec{z}_t \mid \vec{x}_t) , \quad (4.38)$$

welche wiederum bedeutet, dass die Messung \vec{z}_{t-1} nicht von vergangenen Messungen abhängt. Diese Umstand trifft nicht mehr zu sobald das Messglied einer PT1-Dynamik unterliegt.

Die zweite Annahme bestand darin, dass die Stellgröße \vec{u}_t zufällig gewählt wird, und somit nicht von der Position \vec{x}_t abhängt. Diese Annahme ist vollkommen haltlos, da die Lokalisierung im Rahmen dieser Arbeit nur verfolgt wird, um das Ergebnis \vec{x}_t zur Berechnung einer Stellgröße \vec{u}_t zu verwenden. Zur Verteidigung des Verfahrens wird die Ungenauigkeit der Modelle aufgeführt, wobei argumentiert wird, dass die Modellfehler den Widerstoß gegen die bestehenden Annahmen überwiegen. Allerdings ist kritisch zu vermerken, dass es sich bei diesem Argument auch lediglich um eine Annahme handelt. An dieser Stelle offenbart sich eine Schwachstelle der stochastischen Ansätze: Es ist kaum möglich eine analytische Beurteilung der Verfahren durchzuführen. Der Vorteil, dass heuristische Argumente in Form von Wahrscheinlichkeiten ausgedrückt werden, verhindert gleichermaßen ein deterministisches, also ein absolutes Urteil. Als entscheidendes Kriterium bleibt lediglich das Experiment, wobei es wieder schwer fällt zwischen den Einflüssen des Algorithmus und der probabilistischen Modelle zu differenzieren.

Kapitel 5

Kartenerstellung

In diesem Kapitel wird die Erstellung von metrischen Karten betrachtet. Für die Navigation wurde zwar die Annahme getroffen, dass eine Karte der Umgebung im Voraus aufgezeichnet wurde und zur Verfügung steht, allerdings wird das Thema an dieser Stelle aus Gründen der Vollständigkeit kurz angeschnitten. Als einfaches Beispiel dient die Methode des Occupancy-Grid-Mappings, das nicht nur ein Verständnis für die Kartenerstellung liefert, sondern auch einen Einblick in Funktionsweise der stochastischen Modellierung bietet.

5.1 Occupancy-Grid-Mapping

In dem hiesigen Anwendungsfall sollen metrische Karten verwendet werden, um das Navigationsproblem lösen zu können. Ein passender Kartentyp sind die so genannten Occupancy-Grids, welche die Umgebung als 2- oder 3-dimensionales, diskretes Raster darstellen. Jeder Zelle der Karte wird eine Wahrscheinlichkeit zugeordnet, die wiedergibt, ob die Zelle belegt ist. Durch den stochastischen Charakter des Umgebungsmodells können Ungewissheiten bei der Kartographierung und anschließenden Navigation beachtet werden.

Mathematisch wird die Umgebung als diskrete Menge von binären Zufallsvariablen

$$M = \{m_i\} \tag{5.1}$$

beschrieben, die entweder den Zustand frei oder belegt annehmen können. Die Aufgabe des Kartographierungsalgorithmus besteht darin, eine Karte zu erstellen, wobei es sich um die a posteriori Wahrscheinlichkeit

$$p(M = \text{belegt} \mid \vec{z}_{1:t}, \vec{x}_{1:t})^1 \tag{5.2}$$

¹Im Folgenden wird die Wahrscheinlichkeit für $M = \text{belegt}$ bzw. $m_i = \text{belegt}$ als $p(M)$ bzw. $p(m_i)$ geschrieben. Die Wahrscheinlichkeiten für freie Zellen werden mit $p(\neg m_i)$ denotiert.

handelt. Für jede Zelle soll die Wahrscheinlichkeit, dass diese unter Beachtung aller bisheriger Sensorwerte $\vec{z}_{1:t}$ und aller Zustandswerte $\vec{x}_{1:t}$ belegt ist, ermittelt werden. Um dieses Problem zu lösen, wird als erster Ansatz ein Bayes-Filter für binäre Zufallsvariablen angeführt. Es werden die beiden folgenden Annahmen getroffen: Die Umgebung ist stationär, das heißt keine Objekte werden während der Kartenaufzeichnung verschoben. Somit kann der Algorithmus nicht genutzt werden, um dynamischen Hindernissen auszuweichen. Zweitens wird angenommen, dass die Wahrscheinlichkeiten der einzelnen Zellen unabhängig sind.

$$p(M \mid \vec{z}_{1:t}, \vec{x}_{1:t}) = \prod_i p(m_i \mid \vec{z}_{1:t}, \vec{x}_{1:t}) . \quad (5.3)$$

Dadurch können die a posteriori Wahrscheinlichkeiten der Zellen separat berechnet werden. Allerdings kann diese Annahmen unter realen Umständen kaum verteidigt werden, da Hindernisse für gewöhnlich größer als eine einzelne Kartenzelle sind.

Für die Berechnung der Karte wird das logarithmische Verhältnis

$$l(x) = \log \left[\frac{p(m_i)}{1 - p(m_i)} \right] = \log \left[\frac{p(m_i)}{p(\neg m_i)} \right] , \quad (5.4)$$

das bei binären Zufallsvariablen rechentechnische Vorteile mit sich bringt [1, S. 94 f]. Bei der Kartenaufzeichnung soll das logarithmische Verhältnis $l(m_i)_t$ am Zeitpunkt t anhand des vergangen Wertes $l(m_i)_{t-1}$, des Messvektors \vec{z}_t und des Zustandsvektors \vec{x}_t für alle i Zellen der Karte bestimmt werden. Die gesuchte Wahrscheinlichkeit ergibt sich aus

$$p(x \mid =) = \frac{1}{1 + e^{l(x)_t}} . \quad (5.5)$$

Für die Herleitung des Bayes-Filter wird zunächst der Fall betrachtet, dass die Zelle m_i belegt ist. Nach Bayes-Theorem folgt für die a posteriori Wahrscheinlichkeit

$$\begin{aligned} p(m_i \mid \vec{z}_{1:t}) &= p(m_i \mid \vec{z}_t, \vec{z}_{1:t-1}) \\ &= \frac{p(\vec{z}_t \mid m_i, \vec{z}_{1:t-1}) \cdot p(m_i \mid \vec{z}_{1:t-1})}{p(\vec{z}_t \mid \vec{z}_{1:t-1})} \\ &= \frac{p(\vec{z}_t \mid m_i) \cdot p(m_i \mid \vec{z}_{1:t-1})}{p(\vec{z}_t \mid \vec{z}_{1:t-1})} . \end{aligned} \quad (5.6)$$

Wie Wahrscheinlichkeit $p(\vec{z}_t \mid m_i)$ wird als Messmodell bezeichnet, da sie angibt, wie wahrscheinlich ein Messwert \vec{z}_t aus einer gegebenen Umgebung m_i folgt. Aus Bayes-Theorem folgt wiederum

$$p(\vec{z}_t \mid m_i) = \frac{p(m_i \mid \vec{z}_t) \cdot p(\vec{z}_t)}{p(m_i)} \quad (5.7)$$

und die anschließende Substitution in 5.6 liefert

$$p(m_i | \vec{z}_{1:t}) = \frac{p(m_i | \vec{z}_t) \cdot p(\vec{z}_t) \cdot p(m_i | \vec{z}_{1:t-1})}{p(m_i) \cdot p(\vec{z}_t | \vec{z}_{1:t-1})}. \quad (5.8)$$

Analog ergibt sich für das komplementäre Ereignis $\neg m_i$

$$p(\neg m_i | \vec{z}_{1:t}) = \frac{p(\neg m_i | \vec{z}_t) \cdot p(\vec{z}_t) \cdot p(\neg m_i | \vec{z}_{1:t-1})}{p(\neg m_i) \cdot p(\vec{z}_t | \vec{z}_{1:t-1})}. \quad (5.9)$$

Im nächsten Schritt folgt der Quotient der beiden Verteilungen

$$\begin{aligned} \frac{p(m_i | \vec{z}_{1:t})}{p(\neg m_i | \vec{z}_{1:t})} &= \frac{p(m_i | \vec{z}_t) \cdot p(\vec{z}_t) \cdot p(m_i | \vec{z}_{1:t-1}) \cdot p(\neg m_i) \cdot p(\vec{z}_t | \vec{z}_{1:t-1})}{p(\neg m_i | \vec{z}_t) \cdot p(\vec{z}_t) \cdot p(\neg m_i | \vec{z}_{1:t-1}) \cdot p(m_i) \cdot p(\vec{z}_t | \vec{z}_{1:t-1})} \\ &= \frac{p(m_i | \vec{z}_{1:t-1})}{p(\neg m_i | \vec{z}_{1:t-1})} \cdot \frac{p(m_i | \vec{z}_t)}{p(\neg m_i | \vec{z}_t)} \cdot \frac{p(\neg m_i)}{p(m_i)}, \end{aligned} \quad (5.10)$$

dessen Logarithmus das gesuchte Verhältnis liefert:

$$\begin{aligned} l(m_i)_t &\equiv \log \left[\frac{p(m_i | \vec{z}_{1:t})}{p(\neg m_i | \vec{z}_{1:t})} \right] \\ &= \underbrace{\log \left[\frac{p(m_i | \vec{z}_{1:t-1})}{p(\neg m_i | \vec{z}_{1:t-1})} \right]}_{=l(m_i)_{t-1}} - \underbrace{\log \left[\frac{p(\neg m_i)}{p(m_i)} \right]}_{=l(m_i)_{t=0} \equiv l_0} + \log \left[\frac{p(m_i | \vec{z}_t)}{p(\neg m_i | \vec{z}_t)} \right]. \end{aligned} \quad (5.11)$$

Somit setzt sich die gesuchte a posteriori Wahrscheinlichkeit aus drei Summanden zusammen. Bei dem Ersten handelt es sich um die Wahrscheinlichkeit $l(m_i)_{t-1}$ am vorherigen Abtastpunkt. Der Zweite gibt das Verhältnis der a priori Wahrscheinlichkeiten zum Zeitpunkt $t = 0$ wider und der letzte Teil ergibt sich aus dem logarithmischen Verhältnis von $p(m_i | \vec{z}_t)$, wobei es sich um ein inverses Messmodell handelt. Die Wahrscheinlichkeit $p(m_i | \vec{z}_t)$ wird von dem Entwickler vorgegeben und basiert auf der Charakteristik der Sensorik. Als Beispiel dient an dieser Stelle ein rudimentäres Modell für einen Laserscanner. Der Sensor gibt einen Vektor von Distanzmessungen zurück, die jeweils einen Messwinkel φ relativ zu dem Roboter zuzuordnen sind. Somit liegt ein Messbereich in Form eines Kegels mit dem Öffnungswinkel $\Delta\varphi$ vor, der durch die gemessenen Distanzen begrenzt wird. Im inversen Messmodell werden nun drei Fälle unterschieden. Liegt eine Zelle außerhalb des Messkegels, kann keine Aussage über den Zustand der Zelle getroffen werden, weshalb der Wert l_0 zurückgegeben wird der in 5.11 eingesetzt zu

$$l(m_i)_t = l(m_i)_{t-1} + l_0 - l_0 = l(m_i)_{t-1} \quad (5.12)$$

führt; die Wahrscheinlichkeit der Zelle bleibt also unverändert.

In dem Fall, dass die Zelle innerhalb des Messkegels liegt, wird geprüft auf welchem Messstrahl und in welcher Distanz die Zelle sich befindet. Ist der Abstand zwischen der Zelle und der gemessenen Distanz kleiner als ein Toleranzmaß α , wird angenommen,

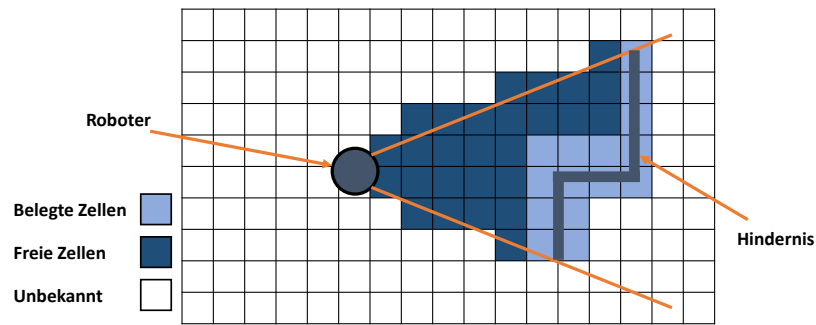


Abbildung 5.1: Schematische Darstellung der Karten und des inversen Messmodells

dass die Zelle belegt ist und ein entsprechender Wert l_{occupied} zurückgegeben. Liegt die Zelle mehr als α Einheiten vor der gemessenen Distanz, gilt sie als frei und das Modell liefert den Wert l_{free} .

Kapitel 6

Lokaler Planer: Dynamic Window Approach

Nachdem die Berechnung eines Pfades zwischen Ausgangs- und Zielposition mittels A* gelöst wurde, muss als nächstes eine Folge von Steuerkommandos ermittelt werden, um den geplanten Pfad abzufahren. Den Ausgangspunkt stellt die Dynamik des Roboters dar: Die Position des Roboters wird in dem Zustandsvektor

$$\vec{\mathbf{x}}(t) = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix} \quad (6.1)$$

erfasst, wobei die ersten beiden Größen die Position und der Winkel θ die Blickrichtung des Roboters beschreiben. Nach wie vor gilt die Annahme, dass der Roboter in einer planen Umgebung manövriert. Die Positionsänderung wird mithilfe einer Translationsgeschwindigkeit v und einer Rotationsgeschwindigkeit ω erfasst, die wiederum in dem Vektor

$$\vec{\mathbf{u}} = \begin{pmatrix} v \\ \omega \end{pmatrix} \quad (6.2)$$

zusammengefasst werden. Bei den Turtlebots werden die Motoren drehzahl geregelt, weshalb der Geschwindigkeitsvektor $\vec{\mathbf{u}}$ die Stellgröße zur Ansteuerung des Roboters darstellt. wird die Annahme getroffen, dass die an einem Zeitpunkt $n \cdot T_a$ eingestellt Geschwindigkeit für die folgende Abtastperiode konstant ist, kann die Positionsänderung in X- und Y-Richtung mittels der folgenden Gleichungen berechnet werden:

$$x_{n+1} = x_n + \Delta x(n) \quad (6.3)$$

$$y_{n+1} = y_n + \Delta y(n) \quad (6.4)$$

$$\Delta x(n) = \begin{cases} \frac{v_n}{\omega_n} \cdot (\sin[\theta_n] - \sin[\theta_n + \omega_n \cdot T_a]) & \forall \omega_n \neq 0 \\ v_n \cdot \cos[\theta_n] \cdot T_a & \forall \omega_n = 0 \end{cases} \quad (6.5)$$

$$\Delta y(n) = \begin{cases} -\frac{v_n}{\omega_n} \cdot (\cos[\theta_n] - \cos[\theta_n + \omega_n \cdot T_a]) & \forall \omega_n \neq 0 \\ v_n \cdot \sin[\theta_n] \cdot T_a & \forall \omega_n = 0 \end{cases} \quad (6.6)$$

Aus den zeitweise konstanten Geschwindigkeiten resultiert eine Kreisbahn mit dem Radius v_n/ω_n im Falle, dass $\omega_n \neq 0$ gilt. Im Extremfall mit $\omega_n = 0$ bewegt der Roboter sich rein translativ entlang einer Geraden. Somit wird der Raum der möglichen Trajektorien auf Kreisbahnen reduziert, wodurch die Komplexität des folgenden Suchproblems drastisch reduziert wird.

Unter den obigen Annahmen soll nun ein Geschwindigkeitsvektor \vec{u} gewählt werden, um den Roboter in Richtung eines gegebenen Zielpunktes zu fahren, der in der nächsten Umgebung liegt. Im ersten Schritt wird der Versuch unternommen, den Suchraum der möglichen Geschwindigkeiten durch Nebenbedingungen weiter einzugrenzen. Sowohl für die Translations- als auch die Rotationsgeschwindigkeit bestehen Maxima, die nicht überschritten werden dürfen, woraus die Menge

$$V_s = \{(v, \omega) \mid v < v_{\max} \wedge \omega < \omega_{\max}\} \quad (6.7)$$

resultiert. Neben den Geschwindigkeiten sind auch die Beschleunigungen beschränkt, weshalb die Menge der erreichbaren Geschwindigkeiten durch die aktuellen Geschwindigkeiten v_a und ω_a , und die maximalen Beschleunigungen eingeschränkt werden.

$$V_a = \{(v, \omega) \mid v \in [v_a - \dot{v}_{\max} \cdot T_a, v_a + \dot{v}_{\max} \cdot T_a] \wedge \omega \in [\omega_a - \dot{\omega}_{\max} \cdot T_a, \omega_a + \dot{\omega}_{\max} \cdot T_a]\} \quad (6.8)$$

Als dritte Begrenzung des Suchraums werden Hindernisse in der Umgebung herangezogen. Jedem Geschwindigkeitspaar (v, ω) kann eine Kreisbahn zugeordnet werden, wobei die Funktion $\text{dist}(v, \omega)$ die Distanz des nächsten Hindernisses am Ende der Abtastperiode auf der zugehörigen Kreisbahn wiedergibt. Wenn nun eine maximale Verzögerung von \dot{v}_{\max} und $\dot{\omega}_{\max}$ erreicht werden kann, folgt für die maximal erlaubte Geschwindigkeit, um vor dem Hindernis bremsen zu können:

$$V_d = \left\{ (v, \omega) \mid v \leq \sqrt{2 \cdot \text{dist}(v, \omega) \cdot \dot{v}_{\max}} \wedge \omega \leq \sqrt{2 \cdot \text{dist}(v, \omega) \cdot \dot{\omega}_{\max}} \right\} \quad (6.9)$$

Der Raum der zulässigen Geschwindigkeiten ergibt sich aus dem Schnitt der drei Mengen

$$V = V_s \cap V_a \cap V_d \quad (6.10)$$

Nun gilt es eine Geschwindigkeit aus dem Suchraum V auszuwählen, wofür wiederum ein Gütekriterium definiert wird, dessen Optimum erzielt werden muss. Bei der Auswahl der Zielfunktion werden drei Aspekte berücksichtigt: Der Roboter soll sich möglichst weit in Richtung des Zielpunktes bewegen, die Blickrichtung des Roboters soll am Ende der folgenden Abtastperiode auf den Zielpunkt gerichtet sein, und die Distanz zwischen dem Roboter und Hindernissen soll möglichst groß gehalten werden. Diese drei Teilziele

werden jeweils durch eine separate Funktion beschrieben, deren Summe bei der Suche maximiert werden soll. Die Richtungsabweichung wird durch die Funktion

$$\text{heading}(v, \omega) = \pi - \varphi \quad \varphi = \angle \quad (6.11)$$

erfasst, wobei φ den Winkel zwischen Blickrichtung und Zielpunkt beschreibt. Die Gewichtung der Geschwindigkeit erfolgt mittels einer simplen Gerade der Form

$$\text{velocity}(v, \omega) = a \cdot v. \quad (6.12)$$

Um den Abstand von Hindernissen zu bewerten, wird wieder die bekannte Funktion $\text{dist}(v, \omega)$ verwendet, wobei zu beachten ist, dass im Falle einer Kurve ohne Hindernisse eine recht große Konstante zurückgeliefert wird. Zusammengeführt ergeben die einzelnen Teile die Zielfunktion

$$G(v, \omega) = \alpha \cdot \text{heading}(v, \omega) + \beta \cdot \text{velocity}(v, \omega) + \gamma \cdot \text{dist}(v, \omega), \quad (6.13)$$

wobei die Faktoren α , β und γ zur Normierung der Teilzielfunktionen verwendet werden. Dadurch wird sichergestellt, dass keine der drei Funktionen das Ergebnis der Optimierung überproportional beeinflusst.

6.1 Anwendungsbeispiel

Um die Funktionsprinzipien des lokalen Planers besser nachvollziehen zu können, wird an dieser Stelle schrittweise eine Implementierung erarbeitet, die anschließend in der Simulation verifiziert wird. Im ersten Schritt werden die Bedingungen für die zulässigen Geschwindigkeiten konstruiert, wofür Angaben für die maximalen Geschwindigkeiten benötigt werden. Diese werden im Datenblatt des TurtleBot [7] mit

$$v_{\max} = 0.65 \quad \omega_{\max} = \pi \quad (6.14)$$

spezifiziert. Über die maximalen Verzögerung der Aktoren sagt das Datenblatt nichts aus, weshalb an dieser Stelle die Annahmen von

$$\dot{v}_{\max} = 0.65 \quad \dot{\omega}_{\max} = \pi \quad (6.15)$$

getroffen wird. Für die spätere Anwendung können die maximalen Beschleunigungen experimentell ermittelt werden und auch in der Simulation adaptiert werden. Die aktuellen Geschwindigkeiten v_a und ω_a können den Odometriedaten entnommen werden, welche über die ROS-Topic `\odom` veröffentlicht werden.

Als letzte Einschränkung ist der Abstand zwischen nächstem Hindernis und dem

Turtlebot zu beachten, der mithilfe der Funktion $\text{dist}(v, \omega)$ berechnet wird. Die Implementierung der Distanzfunktion macht sich wieder den Umstand zu nutzen, dass die Menge der möglichen Trajektorien begrenzt ist: Der Roboter bewegt sich auf einer Kreisbahn mit dem Radius $r = \frac{v}{\omega}$, deren Mittelpunkt M simultan als Ursprung des Koordinatensystems verwendet wird.

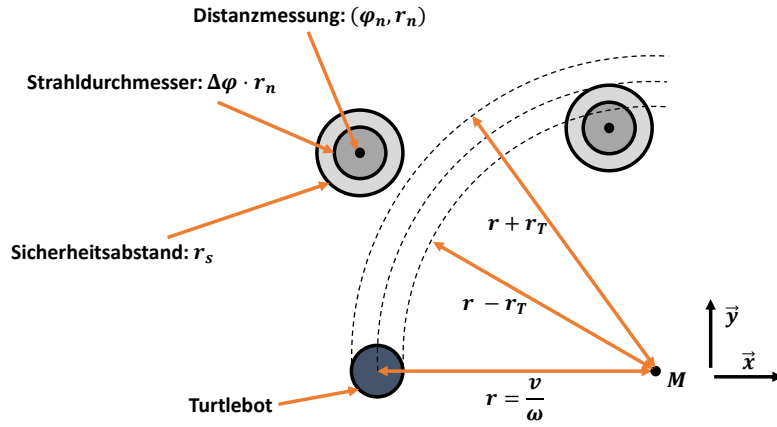


Abbildung 6.1: Darstellung der Trajektorie und Hindernissen

Somit folgt für die Position des TurtleBot

$$\vec{p}_R = \begin{cases} -\begin{pmatrix} r \\ 0 \end{pmatrix} & \forall \omega < 0 \\ \begin{pmatrix} r \\ 0 \end{pmatrix} & \forall \omega > 0 \end{cases}, \quad (6.16)$$

wobei der Fall $\omega = 0$ separat betrachtet werden muss. Da es sich bei dem Roboter um keinen Punkt sondern eine Kreisscheibe mit dem Radius r_R handelt, wird ein Schlauch mit Innenradius $r_I = r - r_R$ und Außenradius $r_A = r + r_R$ überfahren. Diese Fläche kann in Polarkoordinaten als die Punktmenge

$$K_R = \{(\phi, s) \mid s \in [r_I, r_A] \wedge \phi \in [0, 2 \cdot \pi]\} \quad (6.17)$$

dargestellt werden. Der Laserscanner liefert einen Messvektor \vec{z} , dessen Element jeweils ein Paar (φ_n, r_n) sind, das sowohl den Winkel zwischen Messstrahl und Blickrichtung als auch die darauf gemessene Entfernung r_n beschreibt. Folglich stellt das Paar die Position eines Messpunktes relativ zu dem TurtleBot dar, wobei die Darstellung in Form von Polarkoordinaten vorliegt. Liegt der Messpunkt zwischen den Kreisbahnen $r + r_T$ und $r - r_T$ blockiert er die Bahn des Roboters. Des Weiteren muss beachtet werden, dass eine finite Anzahl von Messstrahlen vorliegt, weshalb jeder Messpunkt einen Messkegel mit Öffnungswinkel $2 \cdot \Delta\varphi$ repräsentiert. Insofern ergibt es Sinn zu prüfen, ob der Kreis mit Mittelpunkt (φ_n, r_n) und Radius $\Delta\varphi \cdot r_n$ die Bahn des Roboters schneidet. Als zusätzliche Absicherung kann der Kreis um den Messpunkt um einen

Sicherheitsabstand r_s erweitert werden. Eine mathematische Formulierung resultiert, indem die Position des Messpunktes $\vec{\mathbf{p}}_n$ zunächst in dem Koordinatensystem dargestellt wird:

$$\vec{\mathbf{p}}_n = \vec{\mathbf{p}}_R + \begin{pmatrix} \sin [\varphi_n] \cdot r_n \\ \cos [\varphi_n] \cdot r_n \end{pmatrix}. \quad (6.18)$$

Die im Kreis enthaltene Menge ergibt sich aus allen Punkten, deren Abstand zu dem Messpunkt kleiner als der Radius ist:

$$K_M = \{\vec{\mathbf{p}} \mid \|\vec{\mathbf{p}} - \vec{\mathbf{p}}_n\| \leq \Delta\varphi \cdot r_n + r_s\}. \quad (6.19)$$

Somit kann formal geprüft werden, ob das in einer Messung (φ_n, r_n) identifizierte Hindernis auf einer durch das Geschwindigkeitspaar (v, ω) definierten Kreisbahn liegt. Ergibt der Schnitt der Mengen

$$K_R \cap K_M \quad (6.20)$$

die leere Menge \emptyset , so wird die Trajektorie nicht von dem Hindernis blockiert. Allerdings stellt die Berechnung der Schnittmenge kein elegantes Verfahren dar, um die Position des Hindernisses zu prüfen. An dieser Stelle bietet es sich an, den Abstand d eines Messpunktes $\vec{\mathbf{p}}_n$ zu dem Ursprung M zu berechnen;

$$d = \|\vec{\mathbf{p}}_n\|. \quad (6.21)$$

Alle Punkte des Messkreises K_M liegen zwischen dem maximalen Abstand

$$d_{\max} = d + \Delta\varphi \cdot r_n + r_s \quad (6.22)$$

und dem minimalen Abstand

$$d_{\min} = d - \Delta\varphi \cdot r_n - r_s, \quad (6.23)$$

wodurch sich die Prüfung auf eine simple Fallunterscheidung reduziert. Liegt einer der Abstände d_{\max} und d_{\min} zwischen den Radien r_A und r_I , so blockiert das Hindernis die Kreisbahn. Das selbe Resultat ergibt sich in dem Fall, dass sowohl $d_{\max} > r_A$ als auch $d_{\min} < r_I$ gilt. Liegt ein Hindernis auf der Bahnkurve, so entspricht dessen Winkelposition γ der Polarkoordinate des Messpunktes

$$\gamma = \angle \vec{\mathbf{p}}_n, \quad (6.24)$$

woraus für die auf der Kreisbahn zurückzulegende Distanz

$$\text{dist}(v, \omega) = \gamma \cdot r \quad (6.25)$$

folgt. In dem Falle einer nicht blockierten Trajektorie, gibt die Funktion die maximale Distanz zurück, die in einem Intervall passiert werden kann.

$$\text{dist}(v, \omega) = \begin{cases} v_{\max} \cdot T_a & \forall n : K_R \cap K_M = \emptyset \\ \gamma \cdot r & \exists n : K_R \cap K_M \neq \emptyset \end{cases} . \quad (6.26)$$

Somit liegen nun alle Mittel bereit, um den begrenzten Suchraum

$$V = V_s \cap V_a \cap V_d \quad (6.27)$$

zu konstruieren. Im nächsten Schritt muss die Zielfunktion

$$G(v, \omega) = \alpha \cdot \text{heading}(v, \omega) + \beta \cdot \text{velocity}(v, \omega) + \gamma \cdot \text{heading}(v, \omega) \quad (6.28)$$

maximiert werden, wofür zunächst die Funktionen $\text{heading}(v, \omega)$ und $\text{velocity}(v, \omega)$ implementiert werden müssen. Letztere gibt lediglich den Geschwindigkeitswert zurück:

$$\text{velocity}(v, \omega) = v . \quad (6.29)$$

Die Funktion $\text{heading}(v, \omega)$ gestaltet sich als etwas schwieriger, da hier der Winkel zwischen dem Zielpunkt und der Blickrichtung des Roboters berechnet werden soll. Hier stellt sich die Frage, welcher Punkt als Ziel anvisiert werden soll. Immerhin wurde bei der globalen Planung eine Folge von Positionen berechnet, die den Pfad zum letztendlichen Ziel bilden. Ein Ansatz besteht darin, den geplanten Pfad schrittweise abzuarbeiten, d.h. jeweils der erste Punkte wird angefahren. Wurde das lokal aktuelle Ziel erreicht wird der nächste Punkt des Pfades als lokales Ziel vorgegeben. Bei dieser Variante ergibt sich das Problem, dass eventuell mehrere Punkte des globalen Pfades in einem Abtastintervall erreicht werden können. Wird nun lediglich der erste anvisiert, bewegt sich der Roboter unnötig langsam. Aus diesem Grund kann ein zweiter Ansatz verfolgt werden, bei dem der Punkt des Pfades als Ziel verwendet wird, der einerseits in dem Abtastintervall erreicht werden kann, andererseits aber möglichst weit von der aktuellen Position entfernt ist. In den folgenden Simulationen werden beide Vorgehensweisen implementiert und miteinander verglichen.

Steht ein Zielpunkt fest und ist die Position des Roboters bekannt - was in der ersten Simulationsreihe angenommen wird -, so kann der Winkel ϕ zwischen Blickrichtung und Zielposition mithilfe simpler Trigonometrie berechnet werden. Es folgt

$$\text{heading}(v, \omega) = \pi - \phi . \quad (6.30)$$

Zuletzt müssen die Gewichtungsfaktoren α , β und γ gewählt werden. Sollen alle drei Funktionen das Optimierungsergebnis gleichermaßen beeinflussen bieten sich die

Gewichtungen

$$\alpha = \frac{1}{\pi}; \quad \beta = \frac{1}{v_{\max}}; \quad \gamma = \frac{1}{v_{\max} \cdot T_a} \quad (6.31)$$

an. Alternativ können die Ziel unterschiedlich stark in die Bewertung einfließen indem die Gewichte ungleich verteilt werden. Für die Lösung des Suchproblems bietet es sich an, den Suchraum zu diskretisieren, das heißt es werden Geschwindigkeitsinkremente Δv und $\Delta \omega$ gewählt, woraus die Menge $G(V)$ berechnet wird. Das Optimum

$$\operatorname{argmax} G(V) \quad (6.32)$$

kann durch einen simplen brute-force Ansatz bestimmt werden. Bei der Wahl der Abtastintervalle Δv und $\Delta \omega$ ist ein Kompromiss zwischen Rechenaufwand und Qualität des Ergebnisses zu treffen. Dieser Einfluss wird in der anschließend Simulation weiter untersucht.

Kapitel 7

Erprobung der Navigation

In diesem Kapitel werden die Ergebnisse der zuvor diskutierten Algorithmen in realen Anwendungsszenarien untersucht. Als Beispiel dient der Laborraum F001 der Hochschule Karlsruhe, von dem im Voraus eine Karte mithilfe des ROS-Pakets **hector_slam** aufgezeichnet wurde. An dieser Stelle werden drei verschiedene Anwendungsfälle untersucht. Im Ersten wurde der Raum im Vergleich zur Karte nicht verändert. Außerdem wird angenommen, dass die Position des Roboters zu Beginn der Navigation bekannt ist. Somit handelt es sich um den trivialsten Anwendungsfall der Navigation, womit die Grundfunktionen nachgewiesen werden sollen. Im zweiten Szenario wird der Fall untersucht, dass der auf der Karte unmittelbare Weg zum Ziel durch ein Hindernis blockiert wird, wodurch gezwungenermaßen Sensordaten beachtet werden müssen, um einen Weg zum Ziel zu planen. Der letzte Test dient zur Beurteilung der Lokalisierung. In diesem Fall wird die Roboterposition als unbekannt angenommen und durch mittels einer ungenauen Schätzung initialisiert.

7.1 Anwendungsszenario 1: Triviale Navigation

Das erste Szenario dient auf der einen Seite dem Nachweis der Grundfunktionen der Navigation. Auf der anderen Seite wird das simple Beispiel genutzt, um die Bedienung und Auswertung des Experiment mittels **RViz** zu erläutern. Den Ausgangspunkt aller Experimente stellt die Karte des Labors dar, die unter Anderem in der folgenden Abbildung zu sehen ist. Die schwarzen Pixel stellen Hindernisse, die hellgrauen freie Flächen und die restlichen undefinierte Flächen dar. Die Karte wird von **RViz** angezeigt. Um die

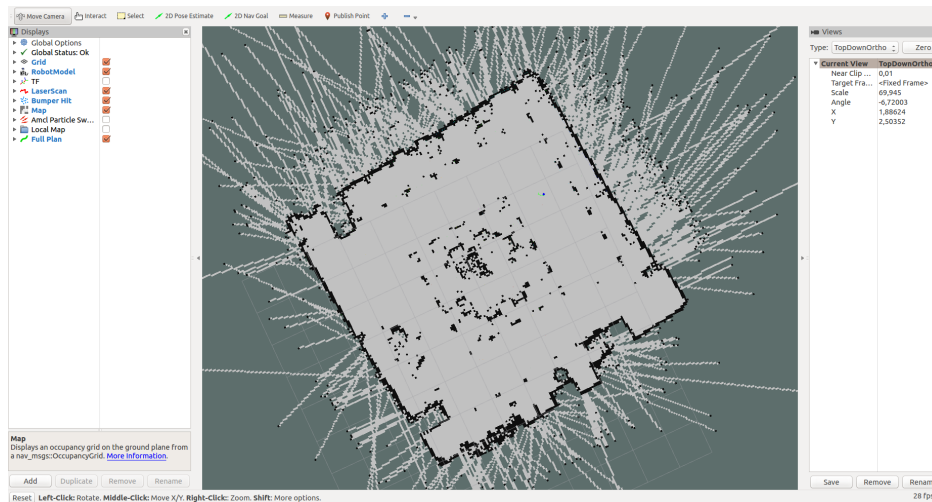


Abbildung 7.1: Übersicht von **RViz**

Visualisierung zu konfigurieren, können die Elemente auf der linken Seite nach Bedarf an- und abgewählt werden. In der Toolbar können die Elemente **2D Pose Estimate** und **2D Nav Goal** genutzt werden, um die eine Positionsschätzung anzugeben und das Ziel der Navigation anzugeben. Mithilfe Letzteren wird ein Ziel in der entgegengesetzten Ecke des Raums vorgegeben, woraufhin ein globaler Pfad geplant wird, der in der folgenden Abbildung zu sehen ist.

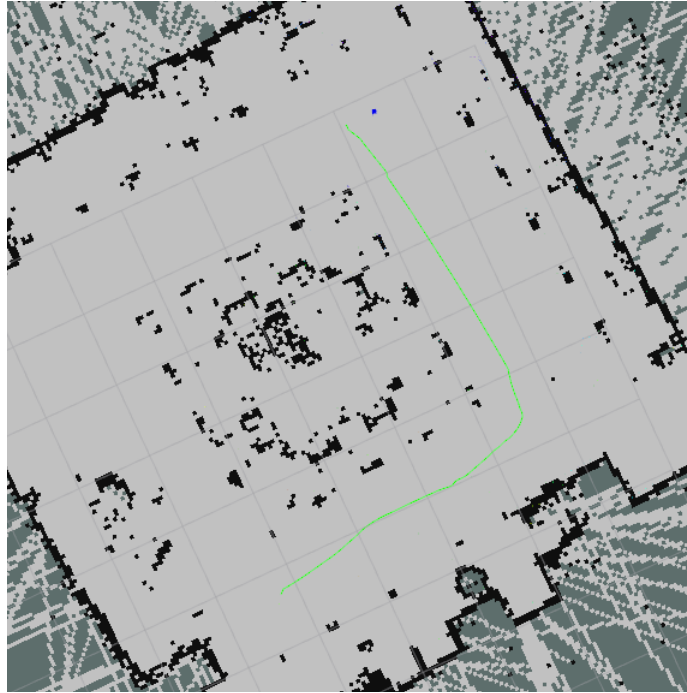


Abbildung 7.2: Globaler Plan

Wie der oben eingezeichnete Pfad entsteht, wird recht leicht ersichtlich, wenn die globale Kostenkarte betrachtet wird. Die Bewertung der Zellen beginnt jeweils in den belegten Hindernissen und wird exponentiell fallend von diesen weg propagiert. Der

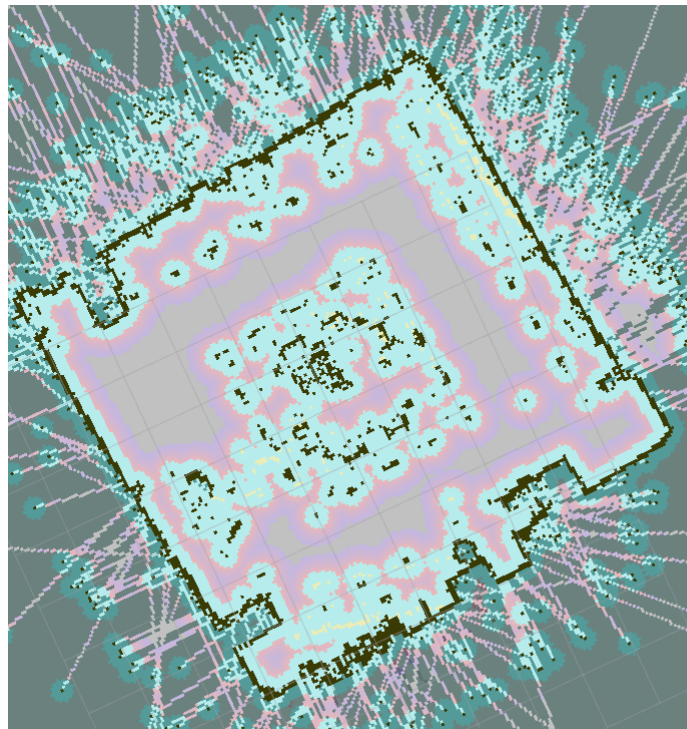


Abbildung 7.3: Globale Kostenkarte des Labors

gezeigt Pfade wurde nach der Planung problemlos von dem Roboter abgefahren, womit die Grundfunktion der Navigation nachgewiesen ist.

7.2 Anwendungsszenario 2: Hindernisdetektion

Im nächsten Schritt wird der Fall betrachtet, dass der Weg des Roboters durch ein unbekanntes Hindernis versperrt wird. Darunter ist ein Objekt zu verstehen, dass bei der Kartenaufzeichnung nicht vorhanden war, weshalb der Roboter auf seine Sensordaten zurückgreifen muss, um den Gegenstand zu detektieren und in die Wegplanung einzuarbeiten. In der folgenden Abbildung sind die Position des Roboters als auch die aktuellen Sensordaten zu erkennen. An letzteren lässt sich die Position des Hindernisses deutlich erkennen.

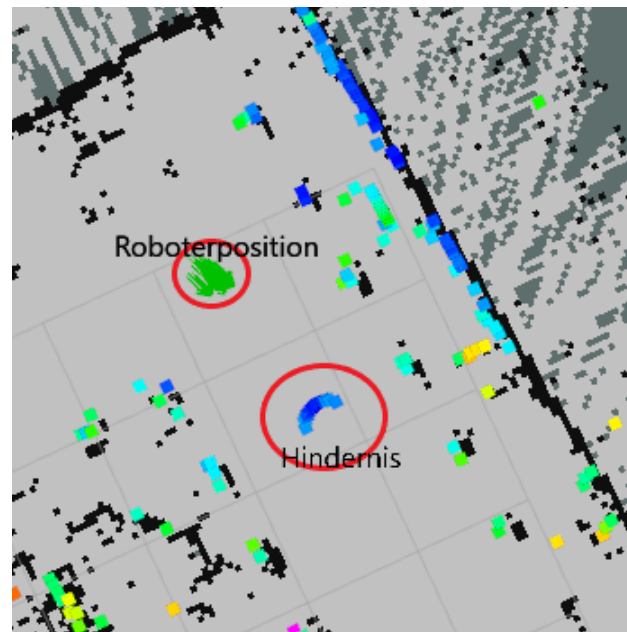


Abbildung 7.4: Roboter und Sensordaten mit Hindernis

Die beiden weiteren Abbildungen zeigen wie die Sensordaten in die globale Kostenkarte integriert werden und wie daraufhin der globale Plan um die Distanzmessungen herumführt.

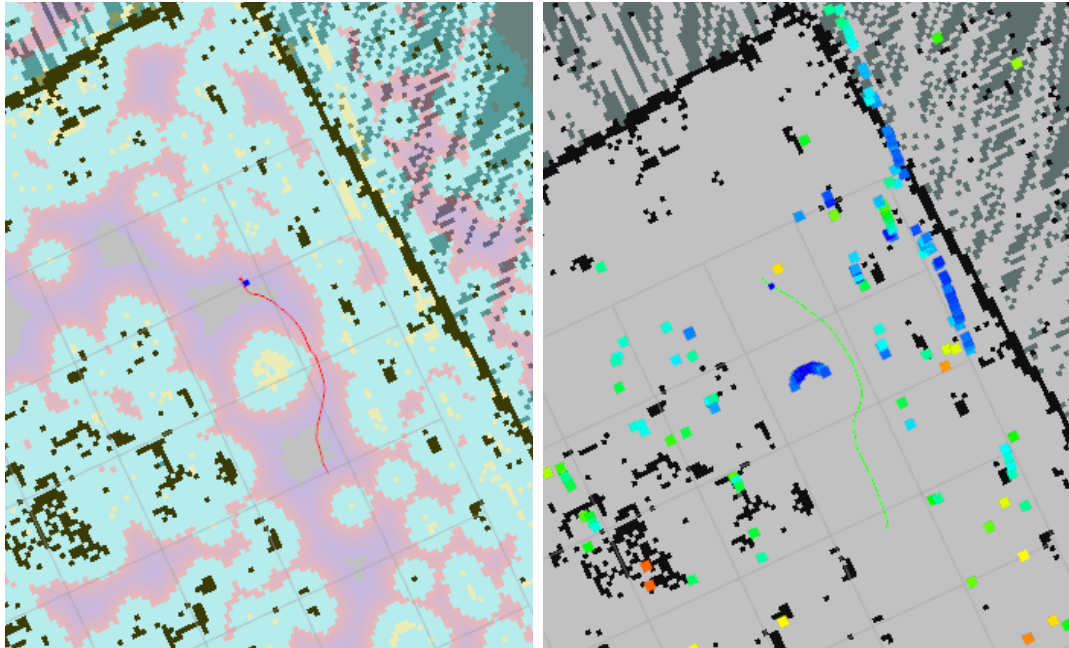


Abbildung 7.5: Integration der Sensordaten in die Pfadplanung

In dem Experiment hat der Roboter das Hindernis nicht nur erfolgreich erkannt und in die Planung aufgenommen, sondern konnte die Route auch abfahren. Allerdings wurde hierbei die Geschwindigkeit deutlich reduziert, was darauf zurückzuführen ist, dass der Plan nach wie vor durch suboptimale Zonen führt. Der Grund hierfür liegt darin, dass keine alternative Route zum Ziel führt, was aber auch zur Folge hat, dass der lokale Planer die Geschwindigkeit des Roboters reduziert, um auf potentielle Kollisionen reagieren zu können.

7.3 Anwendungsszenario 3: Lokalisierung

Im letzten Versuch soll die Performanz der AMC-Lokalisierung untersucht werden. Hierfür wird der Roboter an einer Position im Raum ausgesetzt, die der Navigation nur als fehlerbehaftete, ungenaue Schätzung übergeben wird. Das Partikelfilter der Lokalisierung wird durch eine Positionsschätzung zurückgesetzt, das heißt die Partikel werden neu gezogen, wobei die geschätzte Position als Mittelwert der mehrdimensionalen Normalverteilung verwendet wird. Die initiale Varianz wird als Parameter des Algorithmus festgelegt. Die folgende Abbildung zeigt diesen Ausgangszustand, wobei die grünen Pfeile die verschiedenen Partikel darstellen, die wiederum als mögliche Positionen des Roboters verstanden werden. Außerdem sind die aktuellen Sensorwerte im Verhältnis zum Mittelwert der Partikelwolke eingezeichnet. Aus dem Vergleich der Distanzmessungen und des Wandverlaufs in der Karte wird ersichtlich, wie die Positionsschätzung des Roboters angepasst werden muss, sodass die Sensordaten mit der Umgebung übereinstimmen.

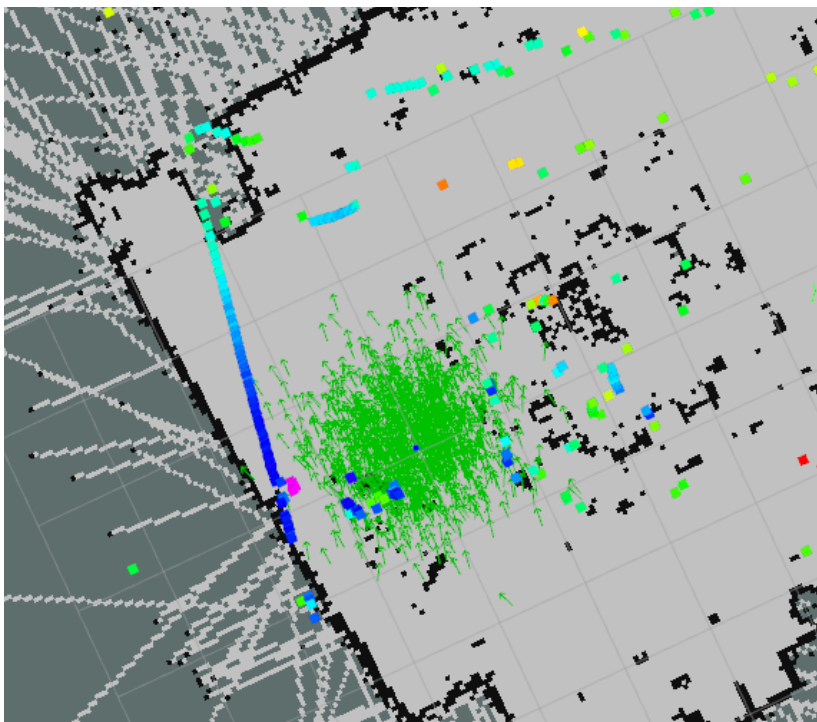


Abbildung 7.6: Ausgangszustand der Lokalisierung nach Positionsschätzung

Im nächsten Schritt der Navigation ein Zielpunkt in kurzer Distanz unmittelbar vor dem Roboter vorgegeben, woraufhin dieser die Bewegung startet. Während der Bewegung wird deutlich, dass sich der Mittelwert der Positionsschätzung verändert, was sich durch die Ausrichtung der Distanzmessungen zu dem Verlauf der Wände manifestiert. Dies zeigt sich in der nächsten Abbildung, wobei das Zusammenrücken der Partikel verdeutlicht, dass die Sicherheit der Lokalisierung zunimmt.

Die nächste Abbildung zeigt den Zustand der Lokalisierung in dem Moment, als der Zielpunkt erreicht wird. Nun stimmen die Punkte der Distanzmessung nahezu

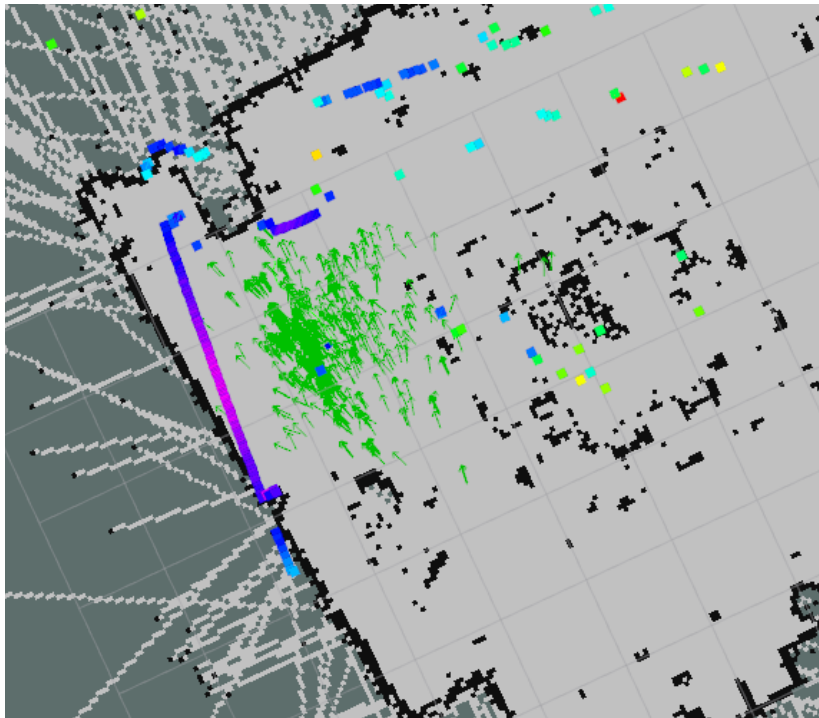


Abbildung 7.7: Positionsschätzung nach kurzer Fahrdistanz

vollkommen mit den in der Karte eingezeichneten Hindernissen überein. Außerdem haben sich die Partikel weiter zusammengezogen, was für ein hohes Maß an Sicherheit der Lokalisierung spricht.

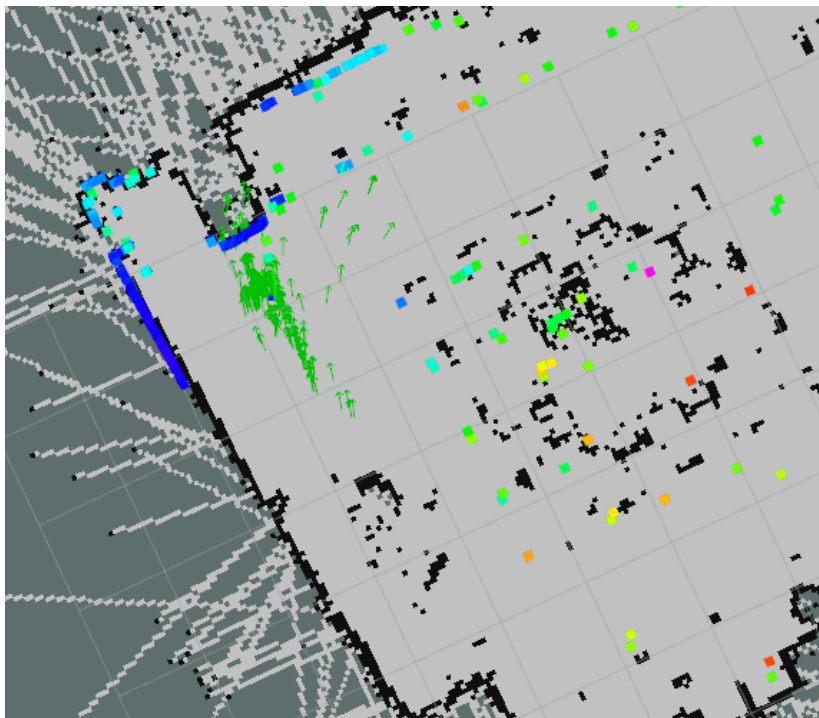


Abbildung 7.8: Positionsschätzung bei Erreichen der Zielposition

Im letzten Schritt wird ein weiteres Navigationsziel gesetzt, woraufhin der Roboter sich weiter durch den Raum bewegt. Hier zeigt sich eine weitere Verdichtung der Parti-

kelwolke, was sich darauf zurückführen lässt, dass die Distanzmessungen weiterhin mit der Karte übereinstimmen. Somit kann die Lokalisierung mit einem erhöhten Maß an Sicherheit als richtig angenommen werden.

An dieser Stelle sei eine negative Kopplung der Navigation und Lokalisierung erwähnt. Im Fall, dass die Lokalisierung in Form einer Positionsschätzung reinitialisiert wird und nur eine kurz entfernte Ziel angesteuert wird, wird während der Navigation die Position des Roboters durch den Lokalisierungsalgorithmus angepasst. Dies hat wiederum zur Folge, dass der Plan des Roboters überarbeitet werden muss, da die ursprüngliche Position verworfen wurde. Dies hat zur Folge, dass der Roboter wiederholte Rotationsmanöver ausführt, die von kurzen Translationsbewegungen gefolgt werden. In diesem Muster bewegt sich der Roboter mehrere Male auf der Stelle, bis die Lokalisierung ein ausreichend hohes Maß an Sicherheit erreicht hat, sodass die geschätzte Position sich nur noch in einer für die Navigation nicht relevanten Größenordnung ändert.


Kapitel 8

ROS Implementierung

Nachdem in den vorherigen Kapiteln die theoretischen Grundlagen der Navigationsalgorithmen erläutert wurden, widmet sich dieser Teil der Arbeit der Umsetzung der Konzepte in ROS. Zunächst wird der Aufbau des Systems, Anordnung der Rechner und die Netzwerkkonfiguration betrachtet. Im Anschluss werden die Roboter schrittweise in Betrieb genommen, wobei die relevanten Dateien erklärt werden und ein Schritt für Schritt Anleitung gezeigt wird.

Als erste Demonstration wird die Fernsteuerung der Roboter betrachtet, wofür eine Launch-Datei zur Initialisierung der Turtlebots angelegt wird. Außerdem wird auf dem Master eine Anwendung ausgeführt, um den Roboter per Tastatur zu steuern. Im nächsten Schritt wird die Kartenaufzeichnung implementiert. Hier wird auf dem Master-PC zusätzlich das Paket **hector_mapping** ausgeführt, welches die Karte erstellt. Außerdem wird **Rviz** verwendet, um die aktuelle Karte und Roboterposition zu visualisieren. Im dritten Anwendungsbeispiel wird die zuvor aufgezeichnete Karte für die autonome Navigation genutzt. Hierfür wird auf Seite des Master die vollständigen Navigationsalgorithmen konfiguriert und ausgeführt. Außerdem zeigt **Rviz** in diesem Beispiel nicht nur die Karte und den Roboter an, sondern dient auch als Eingabemöglichkeit für die Zielposition.

Im Anschluss wird der Endscenario diskutiert, in dem zwei Robotern simultan navigieren. Hier entsteht auf ROS-Seite das Problem, dass Konflikte in der Namensgebung von Nachrichten und Topics entstehen. Eine mögliche Lösung stellen die unter ROS verfügbaren Namensräume dar, wodurch die Daten der beiden Roboter entkoppelt werden können. Allerdings sind in der Umsetzung dieses Lösungsansatzes Probleme mit der Lokalisierung aufgetreten, die bisher noch nicht erklärt bzw. gelöst werden konnten. An dieser Stelle wird der Fehlerfall aufgezeigt und mögliche Lösungen analysiert.

Als eine Lösung für das Problem werden unterschiedliche Launch-Dateien für die beiden Robotern angelegt, in denen die Nachrichtennamen den Robotern angepasst werden. Hierbei handelt es sich um eine recht umständliche und unflexible Lösung, die allerdings funktioniert. 

8.1 Systemstruktur



In dieser Arbeit wird die Navigation von maximal zwei Robotern zur selben Zeit betrachtet. Jeder der Roboter ist mit einem PC ausgestattet, die über WLAN mit einem TP-Link-Router verbunden sind, der als Access-Point des Netzwerks fungiert. Der Roboterverbund ist nach einem Master-Slave-Prinzip konzipiert, wobei die Roboter bzw. deren PCs als Slave agieren. Ein weiterer PC ist mit dem Router verbunden und agiert als Master. Der Master-PC ist zusätzlich mit Tastatur, Maus und Bildschirm ausgestattet und stellt somit die Steuereinheit des Systems dar. Daher stammt auch die Bezeichnung Master-PC, da von diesem die Befehlskette bedient wird. Die Slave-PCs, welche sich auf den Robotern befinden, nehmen die Befehle entgegen, arbeiten also als Slaves.



Das Netzwerk läuft in dem IP-Adressbereich 192.168.0.X, wobei die Adressen des Master- und der Slave-PCs statisch zugewiesen werden. Der Master-PC ist unter der Adresse 192.168.0.100 erreichbar; Roboter R2 unter der Adresse 192.168.0.102; Roboter R4 unter der Adresse 192.168.0.104. Außerdem ist es möglich, dass sich weitere Rechner mit dem Netzwerk verbinden, wodurch auf sämtliche Daten des ROS-Netzes zugegriffen werden kann. Somit können auch Entwicklungswerkzeuge wie MATLAB verwendet werden, um die Daten abzugreifen und weiterzuverarbeiten.

Das Netzwerk trägt den Namen **EML_Turtlebot_NET** mit dem Passwort **turtlebot**. Die statische Konfiguration der IP-Adressen kann im Detail in [8, S. 23] nachgelesen werden. Auf jedem der Rechner existiert ein Nutzer mit dem Namen **turtlebot** und Passwort **turtlebot**, der genutzt werden kann, um eine ssh-Verbindung aufzubauen.

8.2 Inbetriebnahme der Turtlebots

In diesem Teil der Arbeit wird die praktische Umsetzung der Navigationsalgorithmen diskutiert, wofür ROS verwendet wird. Unter ROS werden so genannte Launch-Dateien genutzt, um die beteiligten Pakete zu parametrisieren und zu starten. Typischerweise werden in jedem ROS-Projekt mehrere Launch-Dateien angelegt, die für die jeweiligen Anwendungsfälle ausgelegt sind. In dieser Arbeit werden zwei verschiedene Projekte angelegt. Auf dem Master-PC wird das Projekt **EML_Navigation_Master** genutzt; auf den Slave-PCs das Projekt **EML_Navigation_Slave**. Die Projektordner sind jeweils auf den Desktops der PCs zu finden. In dem Master-Projekt werden die vier Launch-Dateien

- **EML_Mapping_Master.launch**
- **EML_Navigation_Master.launch**
- **EML_Navigation_Robot2_Master.launch**
- **EML_Navigation_Robot4_Master.launch**

genutzt, wobei die erste für die Kartenerstellung, die zweite für die Navigation eines einzelnen Roboters und die beiden letzten für die simultane Navigation der beiden Roboter verwendet werden.

In den Slave-Projekten sind die fünf Launch-Dateien

- **EML_Hardware_Init_Slave.launch**
- **EML_Mapping_Slave.launch**
- **EML_Navigation_Slave.launch**
- **EML_Navigation_Robot2_Slave.launch**
- **EML_Navigation_Robot4_Slave.launch**

zu finden. Die Erste führt die Initialisierung der vorhandenen Hardware durch. Die vier weiteren Dateien stellen das Pendant zu den oben beschriebenen Master-Dateien dar.

8.2.1 Fernsteuerung eines Turtlebots

Im aller ersten Schritt werden die beiden Roboter in Betrieb genommen, wofür die Fernsteuerung der beiden Geräte als Demonstration dienen soll. Auf den Slaves wird dafür die Launch-Datei **EML_Hardware_Init_Slave.launch** angelegt, die die vorhandene Hardware initialisiert, sodass der Roboter über die entsprechenden ROS-Nachrichten bewegt werden kann.

```

1 <launch>
2   <!-- Basis-Initialisierung -->
3   <include file="$(find turtlebot_bringup)/ ...
4     launch/minimal.launch"/>

6   <!-- Initialisierung des SICK-TIM551 -->
7   <param name="robot_description" command="$(find xacro)/ ...
8     xacro.py '$(find sick_tim)/urdf/example.urdf.xacro'"/>
9   <include file="$(find turtlebot_chor_navigation)/launch/ ...
10     include/sick_tim551_2050001_timefix.launch"/>
11 </launch>

```

Listing 8.1: EML_Hardware_Init_Slave.launch

In der Launch-Datei wird zunächst eine Basisinitialisierung durchgeführt, wofür die Datei **minimal.launch** aus dem offiziellen ROS-Paket **turtlebot_bringup**[16] inkludiert wird. Als Zweites wird mithilfe einer der Launch-Datei **sick_tim_551_2050001_timefix.launch** aus dem Vorgängerprojekt [8] der SICK-Laserscanner initialisiert.

Um den Turtlebot über den Master-PC zu steuern, wird die Launch-Datei auf dem Slave ausgeführt.

```

1 roslaunch EML_Navigation_Slave EML_Hardware_Init_Slave.launch

```

Die Eingabe der Steuerbefehle erfolgt auf dem Master-PC, wofür auf das Paket **turtlebot_teleop** [17] zurückgegriffen wird. Zum Start wird der Befehl

```

1 roslaunch turtlebot_teleop keyboard_teleop.launch

```

auf dem Master-PC ausgeführt. Daraufhin kann der Roboter mittels der Tastatur bewegt werden.

8.2.2 Aufzeichnung einer Karte

Im nächsten Schritt wird die Kartographierung der Umgebung betrachtet, wobei die Ergebnisse der Vorgängerarbeit [8, S. 47, ff] verwendet werden können. Auf Slave-Seite wird die Launch-Datei **EML_Mapping_Slave.launch** verwendet, in zunächst die bereits angesprochene Datei zur Hardware-Initialisierung inkludiert wird. Im Anschluss werden die Launch-Dateien für die Kartenaufzeichnung des Pakets **hector_mapping** [18] eingebunden.

```

1 <launch>
2   <!-- Hardware-Initialisierung -->
3   <include file="$(find EML_Navigation_Slave)/launch/ ...
4     EML_Hardware_Init_Slave.launch" />

6   <!-- Hector-Mapping -->
7   <node name="hector_mapping" pkg="hector_mapping" ...
8     type="hector_mapping" output="screen">
9     <param name="base_frame" value="base_link"/>
10    <param name="odom_frame" value="base_link"/>
11    <param name="map_resolution" value="0.05"/>
12    <param name="map_size" value="2048"/>
13    <param name="map_start_x" value="0.5"/>
14    <param name="map_start_y" value="0.5"/>
15    <param name="map_update_distance_thresh" value="0.4"/>
16    <param name="map_update_angle_thresh" value="0.02"/>
17    <param name="map_pub_period" value="0.05"/>
18    <param name="map_multi_res_levels" value="2"/>
19    <param name="update_factor_free" value="0.4"/>
20    <param name="update_factor_occupied" value="0.9"/>
21    <param name="laser_min_dist" value="0.4"/>
22    <param name="laser_max_dist" value="10"/>
23    <param name="laser_z_min_value" value="-1.0"/>
24    <param name="laser_z_max_value" value="1.0"/>
25    <param name="pub_map_odom_transform" value="true"/>
26    <param name="output_timing" value="false"/>
27    <param name="scan_subscriber_queue_size" value="1"/>
28    <param name="pub_map_scanmatch_transform" ...
29      value="true"/>
30    <param name="pub_map_scanmatch_transform" ...
31      value="true"/>
32    <param name="tf_map_scanmatch_transform_ ...
33      frame_name" value="scanmatcher_frame"/>
34  </node>
35  <include file="$(find hector_geotiff)/launch/ ...
36    geotiff_mapper.launch"> </include>
37  <node pkg="tf" type="static_transform_publisher" ...
38    name="map_nav_broadcaster" args="0.12 0 0 0 0 0 ...
39    base_link laser 100"/>
40 </launch>

```

Listing 8.2: EML_Mapping_Slave.launch

Auf dem Master wird die Launch-Datei **EML_Mapping_Master.launch** verwendet, die inhaltlich ebenfalls aus dem Vorgängerprojekt [8] übernommen ist. In der Datei wird lediglich **Rviz** gestartet, um die Ergebnisse der Kartenerstellung zu visualisieren.

```
1 <launch>
2   <!-- Starte Rviz, um Karte anzuzeigen. -->
3   <param name="robot_description" command="$(find xacro)/ ...
4       xacro.py '$(find sick_tim)/urdf/example.urdf.xacro' "/>
5   <node pkg="rviz" type="rviz" name="rviz" args="-d $(find ...
6       turtlebot_chor_navigation)/rviz_cfg/rviz_sick.rviz"/>
7 </launch>
```

Listing 8.3: EML_Mapping_Master.launch

Neben den beiden Launch-Dateien muss wieder eine Anwendung zur Steuerung des Turtlebots gestartet werden. Außerdem muss am Ende der Kartenaufzeichnung die Applikation **map_server** ausgeführt werden, um die Karte abzuspeichern. Es ergibt sich die folgende Anleitung für die Kartenaufzeichnung. Die einzelnen Schritten sind in [8, S. 47 ff] ausführlich beschrieben.

```
1 [M] roscore
2 [S] roslaunch EML_Navigation_Slave EML_Mapping_Slave.launch
3 [M] roslaunch EML_Navigation_Master EML_Mapping_Master.launch
4 [M] roslaunch turtlebot_teleop keyboard_teleop.launch
5 [M] rosrn map_server map_saver
```

Listing 8.4: Anleitung zur Kartenaufzeichnung

8.2.3 Navigation eines einzelnen Roboters

Nachdem eine Karte der Umgebung erstellt wurde, kann diese für die Navigation eines Roboters herangezogen werden. Auf dem Slave wurde dafür die Launch-Datei **EML_Navigation_Slave.launch** angelegt, die lediglich eine Hardware-Initialisierung durchführt.

```

1 <launch>
2   <include file="$(find EML_Navigation_Slave)/launch/ ...
3     EML_Hardware_Init_Slave.launch" />
4 </launch>

```

Listing 8.5: EML_Navigation_Slave.launch

Auf dem Master wird die Launch-Datei **EML_Navigation_Master.launch** verwendet, in der sämtliche Algorithmen der Navigation parametrisiert und gestartet werden.

```

1 <launch>
2   <!-- Map-Server -->
3   <arg name="map_file" default="$(find EML_Navigation_Master)...
4     /maps/test_map.yaml"/>
5   <node name="map_server" pkg="map_server" type="map_server" ...
6     args="$(arg map_file)"/>
7   <!-- AMC-Localization -->
8   <arg name="3d_sensor" default="$(env TURTLEBOT_3D_SENSOR)"/>
9   <arg name="initial_pose_x" default="0.0"/>
10  <arg name="initial_pose_y" default="0.0"/>
11  <arg name="initial_pose_a" default="0.0"/>
12  <arg name="custom_amcl_launch_file" default="$(find ...
13    turtlebot_navigation)/launch/includes/amcl/$...
14    (arg 3d_sensor)_amcl.launch.xml"/>
15  <include file="$(arg custom_amcl_launch_file)">
16    <arg name="initial_pose_x" ...
17      value="$(arg initial_pose_x)"/>
18    <arg name="initial_pose_y" ...
19      value="$(arg initial_pose_y)"/>
20    <arg name="initial_pose_a" ...
21      value="$(arg initial_pose_a)"/>
22    <arg name="odom_frame_id" value="odom"/>
23    <arg name="base_frame_id" value="base_footprint"/>
24    <arg name="global_frame_id" value="map"/>
25    <arg name="use_map_topic" value="true"/>
26  </include>
27  <!-- Static Transform from base_footprint to base_link -->
28  <node pkg="tf" type="static_transform_publisher" ...
29    name="base_link_footprint_tf_broadcaster"...
30    args="0 0 .5 0 0 0 1 base_footprint base_link 100"/>
31  <!-- Rviz -->
32  <node name="rviz" pkg="rviz" type="rviz" args="-d ...
33    $(find turtlebot_rviz_launchers)/rviz/navigation.rviz"/>
34  <!-- move_base -->
35  <include file="$(find EML_Navigation_Master) ...
36    /launch/include/eml_move_base.launch"/>
37 </launch>

```

Listing 8.6: EML_Navigation_Master.launch

In der Datei wird zunächst die AMC-Lokalisierung gestartet, wofür die Standardparameter des ROS-Pakets **amcl** [19] genutzt werden. Des Weiteren wird **Rviz** gestartet, womit die Karte, Position des Roboters sowie die Zielposition graphisch dargestellt werden. Als Konfiguration dient die Datei **navigation.rviz**, die in dem ROS-Paket **turtlebot_rviz_launchers** [20] enthalten ist. Zuletzt wird das Paket **move_base** [21] gestartet, in dem die letztendliche Navigation berechnet wird. Als Konfigurationsdatei dient **eml_move_base.launch**, welche die **move_base**-Instanz parametrisiert.

```

1 <launch>
2   <include file="$(find turtlebot_navigation)/launch/ ...
3     includes/velocity_smoother.launch.xml"/>
4   <include file="$(find turtlebot_navigation)/launch/ ...
5     includes/safety_controller.launch.xml"/>

7   <arg name="odom_frame_id"    default="odom"/>
8   <arg name="base_frame_id"    default="base_footprint"/>
9   <arg name="global_frame_id"  default="map"/>
10  <arg name="odom_topic"       default="odom" />
11  <arg name="laser_topic"       default="scan" />
12  <arg name="custom_param_file" default="$(find ...
13    turtlebot_navigation)/param/dummy.yaml"/>

15  <node pkg="move_base" type="move_base" respawn="false" ...
16    name="move_base" output="screen">
17    <rosparam file="$(find EML_Navigation_Master)/param/ ...
18      eml_costmap_common_params.yaml" command="load" ...
19      ns="global_costmap" />
20    <rosparam file="$(find EML_Navigation_Master)/ ...
21      param/eml_costmap_common_params.yaml" ...
22      command="load" ns="local_costmap" />
23    <rosparam file="$(find turtlebot_navigation)/param/ ...
24      local_costmap_params.yaml" command="load" />
25    <rosparam file="$(find turtlebot_navigation)/param/ ...
26      global_costmap_params.yaml" command="load" />
27    <rosparam file="$(find EML_Navigation_Master)/param/...
28      eml_dwa_local_planner_params.yaml" command="load" />
29    <rosparam file="$(find turtlebot_navigation)/param/...
30      move_base_params.yaml" command="load" />
31    <rosparam file="$(find turtlebot_navigation)/param/...
32      global_planner_params.yaml" command="load" />
33    <rosparam file="$(find turtlebot_navigation)/param/...
34      navfn_global_planner_params.yaml" command="load" />

36  <!-- reset frame_id parameters using user input data -->
37  <param name="global_costmap/global_frame" ...
38    value="$(arg global_frame_id)"/>
39  <param name="global_costmap/robot_base_frame" ...
40    value="$(arg base_frame_id)"/>
41  <param name="local_costmap/global_frame" ...
42    value="$(arg odom_frame_id)"/>
43  <param name="local_costmap/robot_base_frame" ...
44    value="$(arg base_frame_id)"/>
45  <param name="DWAPlanerROS/global_frame_id" ...
46    value="$(arg odom_frame_id)"/>

48  <remap from="cmd_vel" ...

```

```

49         to="navigation_velocity_smoother/raw_cmd_vel"/>
50     <remap from="odom" to="$(arg odom_topic)"/>
51     <remap from="scan" to="$(arg laser_topic)"/>
52 </node>
53 </launch>

```

Listing 8.7: eml_move_base.launch

Allerdings wurden hier lediglich Standardparameter verwendet, die aus dem Demonstrationsbeispiel des **move_base** [21] Paket stammen. In der Konfigurationsdatei werden wiederum Parameterdateien geladen, wobei die meisten aus dem Paket **turtlebot_navigation** [22] stammen. Für Versuchszwecke wurden in dieser Konfiguration Parameterdateien für die Kostenkarte und den lokalen Planer durch eigene Implementierungen ersetzt. Dadurch können Änderung an den Parametersätzen vorgenommen werden.

Um die Navigation auszuführen, muss die beiden Launch-Dateien auf Slave und Master ausgeführt werden. Allerdings ist anzumerken, dass nach dem Neustart der Systeme deren Uhrzeiten synchronisiert werden müssen, wofür der Befehl **ntpdate** herangezogen wird.

```

1 [S] sudo ntpdate 192.168.0.100
2 [M] soscore
3 [S] roslaunch EML_Navigation_Slave EML_Navigation_Slave.launch
4 [M] roslaunch EML_Navigation_Master EML_Navigation_Master.launch

```

Listing 8.8: Anleitung Navigation eines Roboters

8.2.4 Navigation mehrerer Roboter

Das Endziel dieser Arbeit besteht darin, dass zwei Roboter sich simultan durch einen Raum bewegen. Dabei kommunizieren beide Geräte über dasselbe ROS-Netzwerk, weshalb Namenskonflikte bei der Nachrichtenkommunikation auftreten. Beispielsweise wird die Topic **scan**, welche die Laserscandaten enthält, von beiden Robotern veröffentlicht, wenn die obige Launch-Datei auf beiden Slave-PCs ausgeführt wird. So kann bei der Auswertung der Sensordaten nicht mehr zwischen den beiden Robotern differenziert werden.

Eine mögliche Lösung stellen die **group**-Tags [23] unter ROS dar, mit denen jeder ROS-Node ein Namensraum zugeordnet werden kann. Als erstes Beispiel dient die Launch-Datei **EML_NS_Hardware_Init_Slave.launch** auf dem Slave, welche die Hardware unter der Verwendung eines Namensraum initialisiert.

```

1 <launch>
2     <arg name="namespace" default="Robot"/>
3     <group ns="$(arg namespace)">
4         <!-- Basis-Initialisierung -->
5         <include file="$(find turtlebot_bringup)/ ...
6             launch/minimal.launch"/>
7
8     <!-- Initialisierung des SICK-TIM551 -->

```

```

 9          <param name="robot_description" command= ...
10          "$(find xacro)/xacro.py '$(find ...
11          sick_tim)/urdf/example.urdf.xacro'"/>
12          <include file="$(find turtlebot_chor_ ...
13          navigation)/launch/include/ ...
14          sick_tim551_2050001_timefix.launch"/>
15      </group>
16 </launch>

```

Listing 8.9: EML_NS_Hardware_Init_Slave.launch

Die Launch-Datei kann mit dem zusätzlichen Argument **namespace** gestartet werden, wodurch der Namensraum der Knoten festgelegt wird. So führt der Befehl

```

1  roslaunch EML_Navigation_Slave ...
2      EML_NS_Hardware_Init_Slave.launch namespace:="R4"

```

dazu, dass jeder Topic der Präfix **/R4/** hinzugefügt wird, wodurch eine klare Zuordnung zwischen Nachricht und Roboter erfolgt. Auf dem zweiten Roboter muss lediglich der **namespace**-Parameter angepasst werden.

Ebenso wird auf dem Master die Launch-Datei **EML_NS_Teleop_Master.launch** angelegt, womit die Fernsteuerung der Roboter unter Namensräumen erfolgen kann.

```

1 <launch>
2     <arg name="namespace" default="Robot"/>
3     <group ns="$(arg namespace)">
4         <include file="$(find turtlebot_teleop)/ ...
5         launch/keyboard_teleop.launch"/>
6     </group>
7 </launch>

```

Listing 8.10: EML_NS_Teleop_Master.launch

Prinzipiell sollte es möglich sein, die Navigation nach demselben Ansatz mit einem Namensraum zu versehen und dadurch die beiden Roboter zu entkoppeln. Hierfür wird die Slave-Datei **EML_NS_Navigation_Slave.launch** angelegt.

```

1 <launch>
2     <arg name="namespace" default="Robot"/>
3     <group ns="$(arg namespace)">
4         <!-- Map-Server -->
5         <arg name="map_file" default="$(find EML_Navigation_ ...
6         Master)/maps/test_map.yaml"/>
7         <node name="map_server" pkg="map_server" ...
8         type="map_server" args="$(arg map_file)"/>
9
10        <!-- AMC-Localization -->
11        <arg name="3d_sensor" default="$ ...
12        (env TURTLEBOT_3D_SENSOR)"/>
13        <arg name="initial_pose_x" default="0.0"/>
14        <arg name="initial_pose_y" default="0.0"/>
15        <arg name="initial_pose_a" default="0.0"/>
16        <arg name="custom_amcl_launch_file" ...
17        default="$(find turtlebot_navigation)/launch/ ...
18        includes/amcl/$(arg 3d_sensor)_amcl.launch.xml"/>
19
20        <include file="$(arg custom_amcl_launch_file)">

```

```

21      <arg name="initial_pose_x" value= ...
22      "$(arg initial_pose_x)"/>
23      <arg name="initial_pose_y" value= ...
24      "$(arg initial_pose_y)"/>
25      <arg name="initial_pose_a" value= ...
26      "$(arg initial_pose_a)"/>
27      <arg name="odom_frame_id" value="odom"/>
28      <arg name="base_frame_id" value="base_footprint"/>
29      <arg name="global_frame_id" value="map"/>
30      <!-- arg name="scan_topic" value= ...
31      "$(arg namespace)/scan"/-->
32      <!-- param name="tf_prefix" value=...
33      "$(arg namespace)"/-->
34      <arg name="use_map_topic" value="true"/>
35      </include>

37      <!-- Static Transform from base_footprint to base_link -->
38      <node pkg="tf" type="static_transform_publisher" ...
39      name="base_link_footprint_tf_broadcaster" ...
40      args="0 0 .5 0 0 0 1 base_footprint base_link 100"/>

42      <!-- Rviz -->
43      <node name="rviz" pkg="rviz" type="rviz" args="-d ...
44      $(find turtlebot_rviz_launchers)/ ...
45      rviz/navigation.rviz"/>

47      <!-- move_base -->
48      <include file="$(find EML_Navigation_Master)/launch/...
49      include/eml_move_base.launch"/>
50  </group>
51 </launch>

```

Listing 8.11: EML_NS_Navigation_Slave.launch

An dieser Stelle entsteht das Problem, dass in der **Rviz**-Konfiguration festgelegt wird, unter welchen Namen die relevanten Nachrichten veröffentlicht werden. Daraus folgt, dass ein Weg gefunden werden muss, die **Rviz**-Konfiguration ebenfalls mit einem Namensraum zu parametrisieren. Diese Problematik kann gelöst werden, indem über die Benutzeroberfläche die fehlende Präfixe nachträglich eingefügt werden. Somit ist es möglich **Rviz** an beliebige Namensräume anzupassen. Aus praktikablen Gründen bietet es sich an, die Konfigurationen zu speichern und nach Programmstart manuell zu laden, um den Aufwand einer manuellen Rekonfiguration zu ersparen.

Als zweites Problem sind die **tf**-Transformationen zu nennen, bei denen ebenfalls Namenskonflikte entstehen. ROS verwendet einen so genannten Transformation-Tree bzw. **tf**-Tree, um die Position von Objekten darzustellen. Bei einer Transformation - kurz **tf** - handelt es sich um eine Abbildung zwischen zwei Bezugssystemen, die im Kontext von ROS als **frames** bezeichnet werden. Für gewöhnlich fungiert die Karte als Inertialsystem bzw. **fixed frame**. Eine Transformation gibt dann den Zusammenhang zwischen den Systemen **map** und **robot** wieder, wodurch die Position des Roboters festgelegt wird. Offensichtlich übernimmt diese Aufgabe die AMC-Lokalisierung. An dieser Stelle treten wieder dieselben Namenskonflikte wie bei den ROS-Nachrichten

auf. Jeder Roboter benötigt einen Transformationsbaum, der von dem des anderen Roboters entkoppelt ist. An dieser Stelle sei auf das ROS-Werkzeug **tf** hingewiesen, das

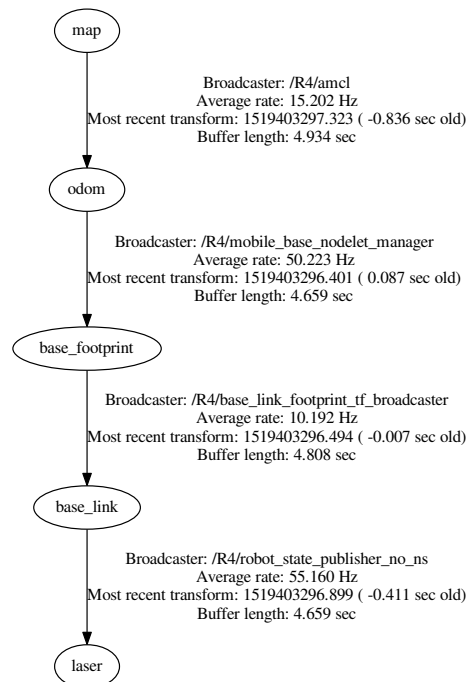


Abbildung 8.1: Transformationsbaum von Roboter 4

es ermöglicht mittels dem Befehl

```
1 rosrund tf view_frames
```

Grafiken wie die obige zu generieren, wodurch die Fehlerfindung erleichtert wird. In dem aktuellen Transformationsbaum wird zuerst von **map** nach **odom** transformiert, wofür der AMCL-Knoten verantwortlich ist. Anschließend veröffentlicht der Knoten **mobile_base_nodelet_manager** eine Transformation von **odom** zu **base_footprint**. Von **base_footprint** wird wiederum in **base_link** überführt. Veröffentlicht wird diese Transformation von **base_link_footprint_tf_broadcaster**. Zuletzt erfolgt die Transformation von **base_link** zu **laser**, wofür der Knoten **robot_state_publisher_no_ns** zuständig ist. In dem Bezugssystem **laser** werden die Daten des Laserscanners interpretiert. Bei **base_footprint** und **base_link** handelt es sich jeweils um körperfeste Bezugssysteme des Roboters, deren relative Ausrichtung zueinander konstant ist.

Die Knoten **base_link_footprint_tf_broadcaster** und **robot_state_publisher_no_ns** werden jeweils über das Slave-Skript gestartet; **mobile_base_nodelet_manager** und **amcl** von der Master-Datei aus.

Ein Ansatz, um die Namenskonflikte aufzulösen, besteht darin, den Parameter **tf_prefix** zu nutzen, womit ähnlich zum **group**-Tag Präfixe an die Bezugssysteme der Roboter angefügt werden können. Dieser Ansatz wird mittel der Launch-Datei **EML_NS_TF_Hardware_Init_Slave.launch** verfolgt.

```

1 <launch>
2   <arg name="namespace" default="Robot"/>
3   <group ns="$(arg namespace)">
4     <!-- Basis-Initialisierung -->
5     <include file="$(find EML_Navigation_Slave)/launch/ ...
6       include/EML_minimal.launch">
7       <arg name="namespace" value="$(arg namespace)"/>
8     </include>

10    <!-- Initialisierung des SICK-TIM551 -->
11    <param name="robot_description" command="$(find xacro)/ ...
12      xacro.py '$(find sick_tim)/urdf/example.urdf.xacro'"/>
13    <include file="$(find turtlebot_chor_navigation)/...
14      launch/include/sick_tim551_2050001_timefix.launch"/>
15  </group>
16 </launch>

```

Listing 8.12: EML_NS_TF_Hardware_Init_Slave.launch

Der primäre Unterschied zu der vorherigen Variante besteht darin, dass die Standarddatei **minimal.launch** aus dem **turtlebot_bringup**-Paket durch die Datei **EML_minimal.launch** ersetzt wurde. Hierbei handelt es sich um eine Kopie der Ausgangsdatei, die um den Namensraumparameter ergänzt wurde, der den jeweiligen ROS-Knoten beim Start übergeben wird. Im relevant Teil von **EML_minimal.launch** werden drei weitere Launch-Dateien inkludiert, die wiederum durch angepasste Kopien ersetzt werden.

```

1   <arg name="namespace"                default="Robot"/>

3   <param name="/use_sim_time" value="$(arg simulation)"/>

5   <include file="$(find EML_Navigation_Slave)/launch/ ...
6     include/EML_robot.launch.xml">
7     <arg name="stacks" value="$(arg stacks)" />
8     <arg name="3d_sensor" value="$(arg 3d_sensor)" />
9     <arg name="namespace" value="$(arg namespace)" />
10  </include>
11  <include file="$(find EML_Navigation_Slave)/launch/ ...
12    include/EML_mobile_base.launch.xml">
13    <arg name="serialport" value="$(arg serialport)" />
14    <arg name="namespace" value="$(arg namespace)" />
15  </include>
16  <include file="$(find EML_Navigation_Slave)/launch/ ...
17    include/EML_netbook.launch.xml">
18    <arg name="battery" value="$(arg battery)" />
19    <arg name="namespace" value="$(arg namespace)" />
20  </include>

```

Listing 8.13: Ausschnitt aus EML_minimal.launch

Die drei Dateien **EML_robot.launch.xml**, **EML_mobile_base.launch.xml** und **EML_netbook.launch.xml** sind wiederum angepasste Kopien der Dateien **robot.launch.xml**, **mobile_base.launch.xml** und **netbook.launch.xml**, die ebenfalls aus dem Paket **turtlebot_bringup** entnommen sind.

Die Hauptproblematik entsteht in der Datei **EML_mobile_base.launch.xml**, in der die Node **mobile_base_nodelet_manager** mit den folgenden Zeile gestartet wird.

```

1 <node pkg="nodelet" type="nodelet" name="mobile_base_ ...
2   nodelet_manager" args="manager">
3   <param name="tf_prefix" value="R4" />
4 </node>

```

Listing 8.14: Ausschnitt aus EML_mobile_base.launch.xml

In diesem Beispiel wird der Parameter **tf_prefix** auf den fixen Wert R4 gesetzt. Die korrekte Ausführung der Parametrisierung wurde mithilfe des ROS-Befehls

```

1 rosparam get mobile_base_nodelet_manager/tf_prefix

```

geprüft, mit dem Ergebnis, dass der Präfix erfolgreich gesetzt wurde. Allerdings zeigt der Transformationsbaum, dass die ROS-Node den Präfix bei der Transformation nicht verwendet. Das Setzen des Präfixparameter bei dem Knoten **robot_state_publisher**

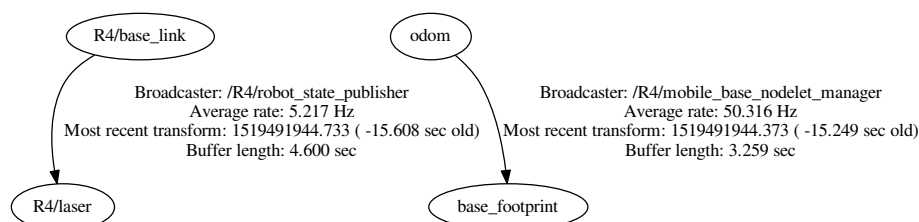


Abbildung 8.2: Transformationsbaum nach gesetztem Transformationspräfix

funktioniert, wie auch in der obigen Abbildung ersichtlich wird, wie erwartet. Insofern handelt es sich um kein prinzipielles Problem des **tf_prefix**-Parameter.

Das hier beschriebene Problem macht es zu dem aktuellen Zeitpunkt unmöglich zwei Roboter parallel in Betrieb zu nehmen.

Kapitel 9

Ausblick und Fazit

Im Rahmen der vorliegenden Arbeit wurde das Funktionsprinzip des ROS-Navigation-Stack erläutert und an einzelnen Beispielen illustriert. Im Anschluss wurde dieses Wissen auf den gegebenen Anwendungsfall übertragen. Die Navigationsaufgabe konnte im Fall eines einzelnen Roboters erfolgreich gelöst werden. Der Roboter ist in der Lage mithilfe einer Karte durch den Raum zu navigieren, unerwartete Hindernisse zu erkennen, ihnen auszuweichen und sich selbstständig auf der Karte zu lokalisieren. Insofern wurden sämtliche Anforderungen an die Navigation erfüllt.

Allerdings konnte das Ergebnis nicht auf die parallele Navigation mehrerer Roboter übertragen werden. Der Grund hierfür liegt in den Namenskonflikten der ROS-Transformationen zwischen den Robotern, die nicht vollständig aufgelöst werden konnten. In dieser Arbeit wurde versucht, mittels des ROS-Parameters **tf_prefix** den Transformationen der verschiedenen Roboter Namensräume zuzuweisen, wodurch eine klare Trennung erfolgt. Allerdings hat dieser Schritt nicht die erwarteten Ergebnisse geliefert. Die exakten Ursachen des Fehlverhalten konnten in dieser Arbeit nicht geklärt werden.

Hieraus resultiert auch das offensichtliche Ziel für eine Folgearbeit: Es muss ein Konzept entwickelt werden, um die Namenskonflikte unter ROS aufzulösen.

Literaturverzeichnis

- [1] Sebastian, Thrun; Wolfram, Burgard; Dieter Fox: „Probabilistic Robotics“, 1. Auflage, Massachusetts Institute of Technology 2006, MIT Press
- [2] LaValle, Steven M. „Planning Algorithms“, 1. Auflage, Cambridge 2006, Cambridge University Press
- [3] SICK AG „TiM-Serie “,
- [4] Takaya, Kenta; Asai, Toshinori; Kroumov, Valeri; Smarandache, Florentin „Simulation Environment for Mobile Robots Testing Using ROS and Gazebo“, 2016, 20th ICSTCC
- [5] Pearl, J. „Heuristics“, Addison-Wesley, 1984
- [6] Fikes, R. E.; Nilsson, N. J. „STRIPS: A new approach to the application of theorem proving.“ Artificial Intelligence Journal, 1971
- [7] Clearpath Robotics „TurtleBot Data Sheet “2015
- [8] Timon, Eßlinger; Jonas, Langmann; Michael, Reibert; Lukas, Schimpf: „Autonom fahrender Roboter TurtleBot 2“, Entwicklungsprojekt Hochschule Karlsruhe, 2017
- [9] global_planner: http://wiki.ros.org/global_planner , aufgerufen am 30.11.2017
- [10] navfn algorism[sic]: <https://answers.ros.org/question/11388/navfn-algorism/?answer=16891#answer-container-16891>, aufgerufen am 30.11.2017
- [11] costmap_2d: http://wiki.ros.org/costmap_2d, aufgerufen am 30.11.2017
- [12] staticmap: http://wiki.ros.org/costmap_2d/hydro/staticmap, aufgerufen am 30.11.2017
- [13] obstacles: http://wiki.ros.org/costmap_2d/hydro/obstacles, aufgerufen am 30.11.2017

- [14] inflation: http://wiki.ros.org/costmap_2d/hydro/inflation, aufgerufen am 30.11.2017
 - [15] dwa_local_planner: http://wiki.ros.org/dwa_local_planner, aufgerufen am 30.11.2017
 - [16] turtlebot_bringup: http://wiki.ros.org/turtlebot_bringup, aufgerufen am 23.02.2018
 - [17] turtlebot_teleop: http://wiki.ros.org/turtlebot_teleop, aufgerufen am 23.02.2018
 - [18] hector_mapping: http://wiki.ros.org/hector_mapping, aufgerufen am 23.02.2018
 - [19] amcl: <http://wiki.ros.org/amcl>, aufgerufen am 23.02.2018
 - [20] turtlebot_rviz_launchers: http://wiki.ros.org/turtlebot_rviz_launchers, aufgerufen am 23.02.2018
 - [21] move_base: http://wiki.ros.org/move_base, aufgerufen am 23.02.2018
 - [22] turtlebot_navigation: http://wiki.ros.org/turtlebot_navigation, aufgerufen am 23.02.2018
 - [23] group: <http://wiki.ros.org/roslaunch/XML/group>, aufgerufen am 24.02.2018
 - [24] TurtleBot2: <http://www.turtlebot.com/turtlebot2/>, aufgerufen am 24.02.2018
 - [25] GAZEBO: <http://gazebo-sim.org/>, aufgerufen am 23.02.2018
-