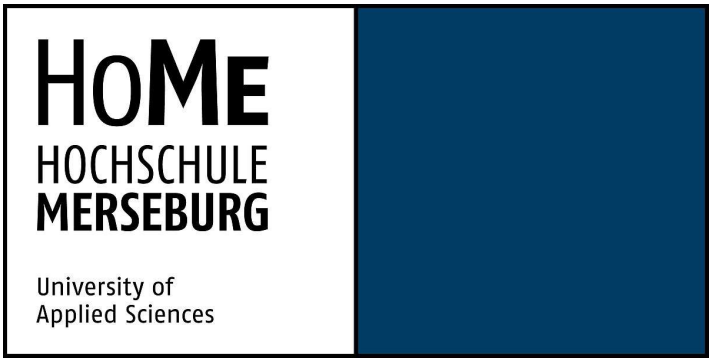
Hochschule Merseburg (FH)

University of Applied Sciences



Fachbereich Ingenieur- und Naturwissenschaften

Studiengang Mechatronik

Bachelorarbeit

zur Erlangung des Grades Bachelor of Engineering (B. Eng)

Entwurf und Umsetzung eines Praktikumsversuches für die Mobile Robotik

Vorgelegt bei

Prof. Dr.-Ing. Stephan Schmidt

Zweitprüfer: Dipl.-Ing. Andreas Goldner

eingereicht von:

Kai-Uwe Schmidt

Abgabetermin: 10.03.2023

Inhalt

Abbildungsverzeichnis

1 Einleitung

In der heutigen Zeit sind mobile Roboter bereits allgegenwärtig. Sie übernehmen Aufgaben in Bereichen wie der industriellen Fertigung, indem sie Bauteile oder Werkstücke für die Weiterverarbeitung an die entsprechend Orte transportieren oder Maschinen bestücken, der Logistik, wo sie Güter in Lagerhäusern befördern oder sortieren. Im Gesundheitswesen finden sie in der Distribution von Medikamenten oder bei Desinfektionsaufgaben Einsatz.[1] Aber auch in privaten Haushalten ist die Technologie in Form von kleinen Saugrobotern bereits angekommen. All diese mobilen Roboter dienen dem Zweck dem Menschen die Arbeit zu erleichtern, die Effizienz zu steigern oder gefährliche Arbeiten für den Menschen zu übernehmen. Ein zentraler Aspekt mobiler Roboter ist dabei die Fähigkeit zur Navigation und Lokalisierung in bekannten oder unbekannten Umgebungen. Hierbei muss der Roboter eigenständig den besten Weg zwischen Start- und Zielpunkt wählen und dabei dynamisch auf Hindernisse reagieren.

1.1 Zielstellung

Ziel dieser Arbeit ist es einen Praktikumsversuch für die mobile Robotik zu entwickeln, in dem die Grundlagen der Lokalisierung, Umgebungswahrnehmung, Prädikation, Planung und Regelung experimentell vermittelt werden sollen. Den Studierenden soll eine Einführung mit dem Umgang von kommerziell verfügbaren Robotersystemen und deren Inbetriebnahmen gegeben werden. Des Weiteren wird die dazu nötige Software-Toolkette erklärt. Am Ende des Praktikumsversuchs sollen die Studierenden in der Lage sein, ein kommerziell verfügbares mobiles Robotersystem in Betrieb zu nehmen und einfache Kartierungs- und Navigationsaufgaben zu bewältigen.

1.2 Motivation

An der Hochschule Merseburg soll im Rahmen der Professur Mechatronische Systeme ein Labor zur mobilen Robotik aufgebaut werden. Diese Arbeit soll mit der Entwicklung eines Praktikumsversuchs unterstützend zur Etablierung des Labors wirken.

1.3 Aufbau der Arbeit

Die Arbeit ist in drei Hauptteile gegliedert. Im ersten Teil werden die theoretischen Grundlagen besprochen. Es wird gezeigt, was ein mobiler Roboter ist, welche Sensorik dieser verwendet, welche Software zur Programmierung eingesetzt wird und wie die Roboternavigation umgesetzt werden kann. Im zweiten Teil wird eine Reihe von kommerziell verfügbaren Systemen vorgestellt, die im Praktikumsversuch verwendet werden können. Im

[1] Vgl. (IFR, 2021)

dritten Teil werden alle nötigen Installationsschritte beschrieben und der Praktikumsversuch wird im Detail erläutert.

2 Grundlagen

Dieses Kapitel vermittelt fundamentale Information zu mobilen Robotern, sowie deren Sensoren. Zudem bietet es eine Einführung in die verwendete Roboter-Software und erläutert die theoretischen Grundlagen der Roboternavigation.

2.1 Mobile Roboter

Bevor der Begriff „mobiler Roboter“ erklärt wird, sollte zunächst erläutert werden, was mit einem Roboter im Allgemeinen gemeint ist. Ein üblicher Startpunkt ist hierfür die Definition für Roboter aus der VDI-Richtlinie 2860:

> „Industrieroboter sind universell einsetzbare Bewegungsautomaten mit mehreren Achsen, deren Bewegungen hinsichtlich Bewegungsfolge und Wegen bzw. Winkeln frei (d. h. ohne mechanischen bzw. menschlichen Eingriff) programmierbar und gegebenenfalls sensorgeführt sind. Sie sind mit Greifern, Werkzeugen oder anderen Fertigungsmitteln ausrüstbar und können Handhabungs- und/oder Fertigungsaufgaben ausführen.“

Diese Definition lässt sich nur begrenzt auf die im Praktikumsversuch verwendeten mobilen Robotern anwenden. Die VDI-Richtlinie zielt auf Handhabungsroboter, wie z.B. Schweißroboter aus der Automobilindustrie, ab. Der Industrieroboter verläuft auf vordefinierten Bahnen in einer komplett bekannten Umgebung und dies ist für die Anwendung in der Industrie ausreichend.

Mobile Roboter unterscheiden sich dahingehend, dass ihre Aktion von ihrer aktuellen Umgebung abhängt, welche im Detail erst bei Ausführung der Aktion bekannt ist. Um diese Aufgabe bewältigen zu können, müssen mobile Roboter die Umgebung mit Sensoren erfassen und die gewonnenen Sensordaten auswerten. In einer zuvor nicht bekannten Umgebung kann der mobile Roboter keine fest programmierten Bahnen abfahren oder Aktionen ausführen. Er muss jederzeit umgebungsabhängig operieren.

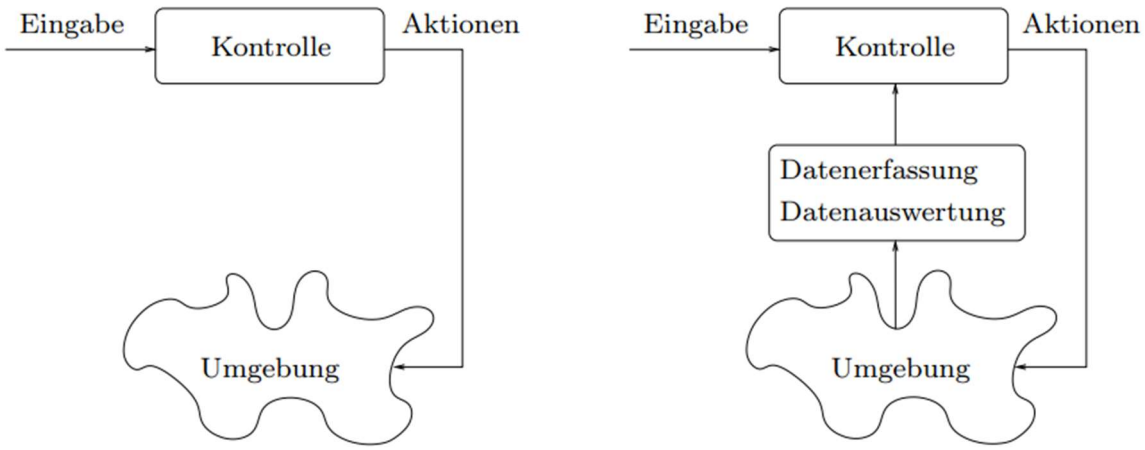


Abbildung 1 Links: Offene Steuerung. Rechts: Geschlossene Regelung unter Rückkopplung von Umgebungsdaten. (Quelle: Hertzberg, Lingemann, & Nüchter, 2012, S. 3)

Der Hauptunterschied zwischen Handhabungsrobotern und mobilen Robotern liegt in der offenen und geschlossenen Steuerung (siehe Abbildung 1). Das Kontrollprogramm eines Handhabungsroboters läuft in offener Steuerung. Das bedeutet, dass der Roboter ohne Sensordaten aus der Umgebung auf vordefinierten Bahnen agiert. Der mobile Roboter handelt, wie schon erwähnt, umgebungsabhängig. Das Kontrollprogramm läuft also in einer geschlossenen Steuerung.[2] Eine Beschreibung eines Roboters, wie er in diesem Praktikumsversuch Verwendung findet, kann wie folgt aussehen:

> „eine frei programmierbare Maschine, die auf Basis von Umgebungssensordaten in geschlossener Regelung in Umgebungen agiert, die zur Zeit der Programmierung nicht genau bekannt und/oder dynamisch und/oder nicht vollständig erfassbar sind.“[3]

Der Zusatz „mobil“ beschreibt dabei die Möglichkeit, dass der Roboter sich selbständig oder ferngesteuert in seiner Umgebung bewegen kann.[4] [5]

Oft werden mobile Roboter, die sich in Ihrer Umgebung selbständig bewegen können als autonome mobile Roboter (AMR) bezeichnet. Die Definition eines AMRs ist wie folgt: Ein AMR ist eine mobile Plattform, die sich innerhalb einer vorgegebenen Umgebung autonom bewegen kann. Dies erfordert die Fähigkeit zur Hindernisvermeidung und Kollisionsvermeidung durch Erkennung von Hindernissen mit Sensoren und Anpassung der Bahnplanung durch Berechnung eines hindernisfreien Pfades durch den freien Raum, anstelle eines vordefinierten Pfads.[6] Der in diesem Praktikumsversuch verwendete mobile Roboter ist demnach auch ein autonomer mobiler Roboter, da die beschriebenen Anforderungen an einen

[2] Vgl. (Hertzberg, Lingemann, & Nüchter, 2012, S. 1-4)
[3] (Hertzberg, Lingemann, & Nüchter, 2012, S. 3)
[4] Vgl. (Hertzberg, Lingemann, & Nüchter, 2012, S. 4)
[5] Vgl. (Iso.org, 2021)
[6] Vgl. (IFR, 2021, S. 5)

AMR im Praktikumsversuch umgesetzt werden. Im weiteren Verlauf der Arbeit sind mit mobilen Robotern die autonomen mobilen Roboter gemeint.

Beispiel für mobilen Roboter

Ein Beispiel für einen mobilen Roboter ist der Staubsaugerroboter. Dieser ist mit einer Kehr- und Saugvorrichtung ausgerüstet und fährt „intelligent“, mit Hilfe einer vom Roboter erstellten Karte der Umgebung, die gewünschten Flächen ab. Die nötige Energie bezieht der Saugroboter aus einem verbauten Akku. Um sich in seiner Umgebung zurechtzufinden, sind im Saugroboter mehrere Sensoren verbaut, darunter: Lidarsensor, inertiale Messeinheit (engl. Inertial Measurement Unit, IMU), Kameras, Stoßsensoren, etc. Je nach Modell unterscheiden sich die Kombinationen der verwendeten Sensoren. In Abbildung 2 ist ein solcher Saugroboter schematisch dargestellt.

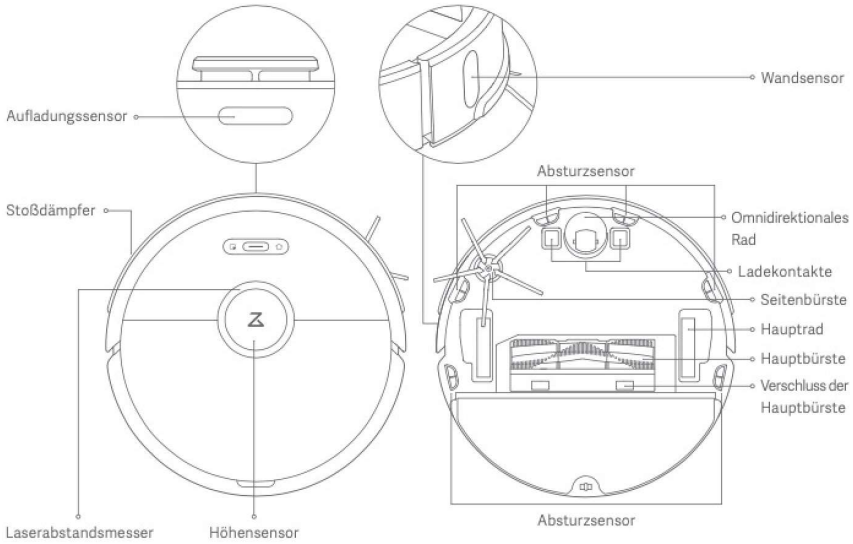


Abbildung 2 Saugroboter (Quelle: https://support.roborock.com/hc/de/article_attachments/360053489972/S6_CE_DE_manual.pdf S.82)

2.2 Sensorik

Mobile Roboter verwenden eine Vielzahl von Sensoren, um sich in Ihrer Umgebung zurechtzufinden. Einige wurden schon in Punkt 2.1 genannt. Im Zuge der folgenden Unterkapitel sollen für den Praktikumsversuch relevante Sensoren näher beschrieben werden, sodass ihre Wirkungsweise deutlich wird.

2.2.1 LiDAR Sensor

LiDAR steht für „Light Detection and Ranging“ und wird z.B. in Industrie, Automobilen und Robotern verwendet, um Entfernungsmessungen zu realisieren. Die Sensoren senden dazu Laserstrahlen aus, welche von Objekten reflektiert und von einem Empfänger wieder aufgenommen werden. Es gibt unterschiedliche Messprinzipien mit der die Entfernung eines Objektes errechnet werden kann. Hier sollen die zwei wichtigsten kurz erläutert werden.

Laufzeitmessung (engl. Time of Flight; TOF)

Um eine Entfernung zu errechnen, wird die Zeit zwischen Aussenden und Empfangen des Laserstrahls gemessen. Da die Geschwindigkeit des Lasers konstant gleich der Lichtgeschwindigkeit ist, ergibt sich die Berechnungsformel:

$$d = \frac{c_{Luft} * t}{2}$$

wobei c_{Luft} die Lichtgeschwindigkeit in der Luft, *t* die Zeit zwischen Aussenden und Empfangen des Laserstrahls und *d* die vom Laserstrahl zurückgelegte Distanz ist. Zu beachten ist hierbei, dass eine sehr präzise Zeitmessung notwendig ist, um eine brauchbare Abstandsmessung zu erzeugen.[7] Um z.B. ein Entfernungsunterschied von 1m aufzulösen, muss die Messung im Nanobereich liegen.

$$\Delta t = \frac{\Delta d}{c_{Luft}} = \frac{1m}{299705518m/s} = 3{,}34 * 10^{-9}s$$

Triangulation

Bei der Triangulation trifft der ausgehende Laserstrahl, abhängig von der Entfernung, unter einem bestimmten Winkel wieder auf ein Empfangselement. Da der Abstand zwischen Sensor und Empfänger, sowie die Auslenkung des Signals bekannt ist, kann mit folgender Formel die Entfernung vom detektierten Objekt ermittelt werden:

$$D = f\frac{L}{x}$$

wobei *D* die Distanz zum Objekt, *f* die Brennweite des Empfängers, *L* die Entfernung zwischen dem Sender und dem Empfänger und *x* die Auslenkung ist.[8] Dieser Zusammenhang ist in Abbildung 3 dargestellt.

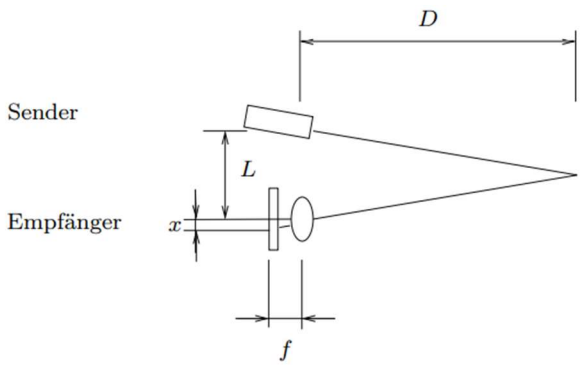


Abbildung 3 Triangulationsmethode zur Entfernungsmessung (Quelle: Hertzberg, Lingemann, & Nüchter, 2012, S. 39)

[7] Vgl. (Siciliano & Oussama, 2016, S. 786-788)
[8] Vgl. (Hertzberg, Lingemann, & Nüchter, 2012, S. 38,39)

2D-Lidar Sensor

Bei der Verwendung von mobilen Robotern wird oft ein 2D-360° Lidar Sensor eingesetzt. Dieser funktioniert gleich den genannten Prinzipien, mit dem Vorteil, dass Hindernisse in einer planen Ebene, unabhängig von der relativen Position zum Roboter erkannt werden können. Die Funktionsweise geht aus Abbildung 4 hervor. Der Laserstrahl wird nicht starr in eine Richtung gesendet und wieder empfangen, sondern durch einen rotierenden Spiegel abgelenkt. Dadurch wird es möglich Objekte innerhalb einer Fläche um den Roboter herum zu detektieren. Zu beachten ist jedoch, dass die Genauigkeit mit der Entfernung der zu detektierenden Objekte abnimmt, wie rechts im Bild zu sehen ist.[9]

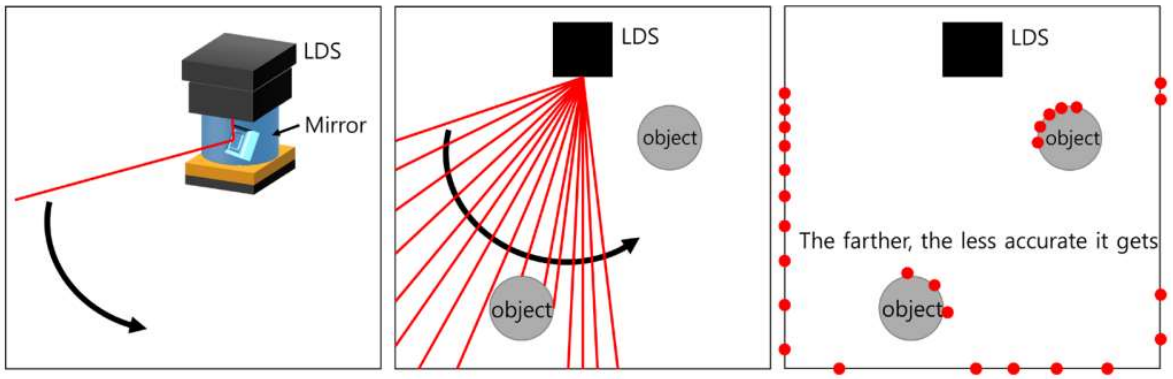


Abbildung 4 Funktionsweise 2D-Lidar (Quelle Pyo, Cho, Jung, & Lim, 2017, S. 218)

2.2.2 Inertiale Messeinheit (engl. Inertial Measurement Unit, IMU)

Eine inertiale Messeinheit ist ein Gerät, welches in der Regel aus Gyroskopen zur Messung der Winkelgeschwindigkeit und Beschleunigungssensoren zur Bestimmung einer anliegenden Kraft oder Beschleunigung besteht. Jeder Sensor kann jeweils nur in einer Achsenrichtung (X,Y,Z) messen, daher müssen für jede Achse ein Beschleunigungsmesser und ein Gyroskop verbaut werden. Trifft dies bei einem IMU zu, so wird dieser mit 6 DOF (Degrees Of Freedom; dt. Freiheitsgrade) betitelt. Es werden auch IMUs mit 9 DOF hergestellt. Diese verfügen zusätzlich noch über einem Magnetometer, der als Kompass oder Magnetfelddetektor fungiert.[10] In Robotern werden oft IMUs mit sogenannte MEMS-Sensoren (engl. Micro Electronic Mechanical System) verwendet, da sie sehr klein und kostengünstig sind. Im Folgendem wird kurz auf die Funktionsweise von MEMS-Beschleunigungsmessern und MEMS-Gyroskopen eingegangen.

[9] Vgl. (Pyo, Cho, Jung, & Lim, 2017, S. 218,219)
[10] Vgl. (Vectornav, 2023)

MEMS-Beschleunigungsmesser

Die Strukturen des Sensors werden ähnlich wie bei der Herstellung von Halbleiterchips auf einer Scheibe aus Silizium aufgebaut. Durch komplexe Oberflächenmikrobearbeitungen wird eine kammähnliche Struktur erstellt. Das in Abbildung 5 rot dargestellte Element (1) ist eine frei beweglich gelagerte Masse, welche mittels federnder Elemente (2) in der Mittelstellung gehalten wird. Die lila dargestellten Elemente (3,4) sind fest mit dem Gehäuse verbunden. Die ineinandergreifenden Zähne der Kämme stellen Kondensatoren dar, welche aufgrund ihrer Größe und ihres Abstandes zu den Anschlusspunkten A und C, bzw. B und C eine messbare Kapazität aufweisen. Die wirksame Fläche der Kondensatoren sind grün dargestellt.

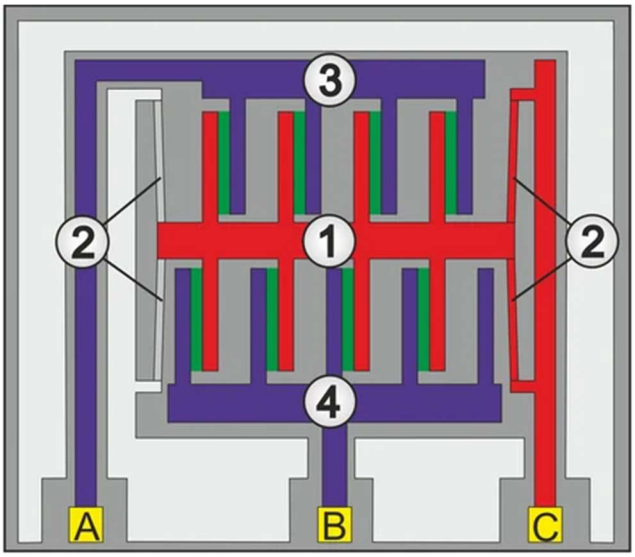


Abbildung 5 MEMS-Beschleunigungsmesser (Quelle: Conrad.de, 2023)

Wirkt nun eine Beschleunigung auf den Sensor, wird die bewegliche Kammstruktur nach links oder rechts ausgelenkt. Dies hat eine messbare Änderung der Kapazität zur Folge. Die Elektronik des Sensors kann dadurch die Stärke und Richtung der Beschleunigung feststellen.[11]

MEMS-Gyroskop

Bei den MEMS-Gyroskopen wird das physikalische Prinzip der Corioliskraft genutzt. Mehrere Lamellen werden über elektrostatische Anregung zu ihrer mechanischen Resonanz in Vibration versetzt. Wird der Sensor um die Querachse gedreht, wird das Konstrukt aufgrund der entstehenden Corioliskraft ein wenig ausgelenkt. Ähnlich wie bei dem Beschleunigungssensor kann dadurch eine Veränderung der Kapazität gemessen werden und dementsprechend eine Drehbewegung zu detektieren. Zwei solcher um 90° gedrehten Konstrukte

[11] Vgl. (Conrad.de, 2023)

erfassen die Drehbewegungen in X- und Y-Richtung. Für die Erfassung in der Z-Achse wird eine andere MEMS-Struktur verwendet, bei der die Lamellen auf einer anderen Raumrichtung schwingen.[12]

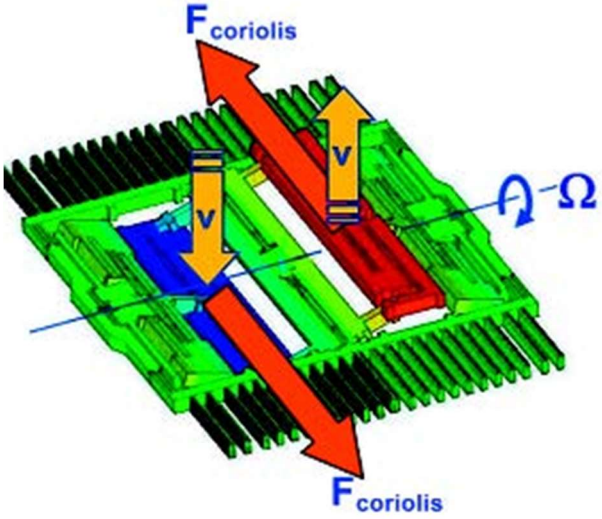


Abbildung 6 MEMS-Gyroskop (Quelle: Macnica, 2023)

2.2.3 Inkrementalgeber

Ist ein Motor oder Rad mit einem Inkrementalgeber ausgestattet, kann bestimmt werden, wie weit sich der Motor in einer bestimmten Zeit gedreht hat. Dies lässt auf die Geschwindigkeit, die gefahrene Strecke sowie die aktuelle Position des Roboters schließen.[13] Der Inkrementalgeber ist eine an der Drehachse befestigte kodierte Scheibe, welche sich mit der Achse mitdrehen. Die Scheibe ist dabei so aufgebaut, dass Signale von einer zur anderen Seite übertragen werden können. Meist werden diese Signale optisch mittels einer Photodiode oder magnetisch durch eine magnetisierte Scheibe übertragen. Es gibt zwei Varianten der Inkrementalgeber. Ein-Kanal-Inkrementalgeber (Tachometer) werden bei Systemen eingesetzt, bei denen die Rotationsrichtung irrelevant ist. Muss die Rotationsrichtung jedoch beachtet werden, wird ein Quadratur-Inkrementalgeber verwendet. Dieser besitzt zwei Kanäle, welche phasenverschoben sind. Die beiden Ausgangssignale bestimmen dann die Drehrichtung durch das Detektieren der steigenden bzw. der fallenden Flanke und deren Phasenverschiebung.[14]

[12] Vgl. (Macnica, 2023)
[13] Vgl. (spacehal.github, 2023)
[14] Vgl. (Hertzberg, Lingemann, & Nüchter, 2012, S. 30,31)

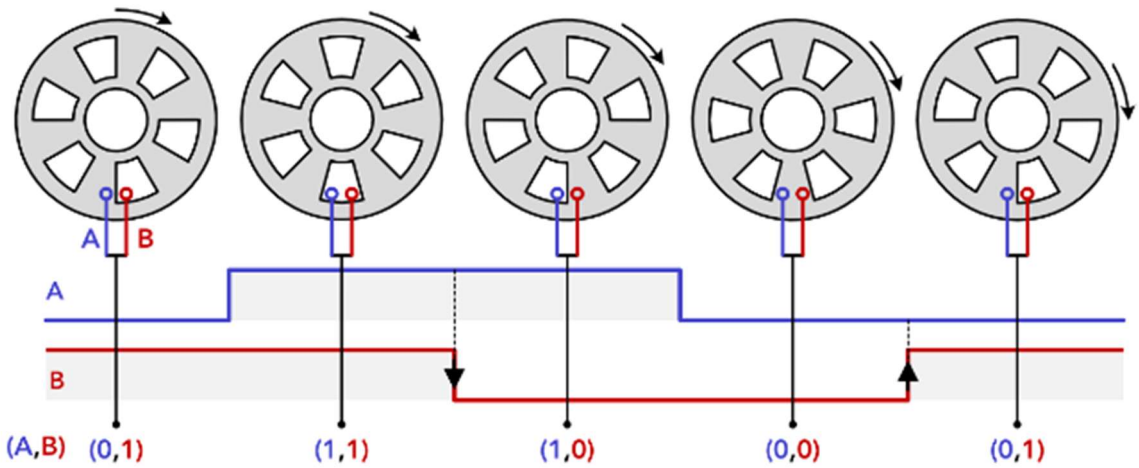


Abbildung 7 Quadratur-Inkrementalgeber mit Drehung im Uhrzeigersinn (Quelle: spacehal.github, 2023)

In Abbildung 7 ist ein Quadratur-Inkrementalgeber abgebildet, welcher sich im Uhrzeigersinn dreht. Aus der Signalreihenfolge lässt sich die Drehrichtung bestimmen. Bei einer Drehung im Uhrzeigersinn und der daraus resultierenden Zustandsänderung des Sensors B (1 → 0 oder 0 → 1), gilt für den Sensor A: B = !A. Dreht sich die Scheiben entgegen des Uhrzeigersinns, gilt für den Sensor B bei Zustandsänderungen des Sensors B: B = A.

2.3 Robot Operating System (ROS)

Um einen Roboter steuern zu können, bzw. damit der Roboter autonom agieren kann, wird eine Software benötigt, welche das angestrebte Roboterverhalten auf dem Roboter umsetzen kann. Da ROS eine Vielzahl von kompatiblen Geräten und vorgefertigten Anwendungsprogrammen mit sich bringt, ist ROS ein beliebtes Tool, um die Roboterprogrammierung zu vereinfachen. ROS ist, obwohl es der Name vermuten lässt, kein richtiges Betriebssystem. Es handelt sich um ein SDK (Software Development Kit), welches die zur Roboterprogrammierung nötigen Bausteine zur Verfügung stellt. Es ist Open Source und wird auf eine passende Linux Distribution installiert und kann von dort aus betrieben werden. Genauere Installationshinweise werden im späteren Verlauf noch aufgeführt. Im Folgenden werden die Kernelemente von ROS und die Funktionsweise näher beschrieben.

2.3.1 ROS Kernelemente

Abbildung 8 bildet das ROS-Ecosystem ab und zeigt, woraus die Kernelemente bestehen. Diese sollen im Folgenden etwas genauer beleuchtet werden, bevor die Funktionsweise von ROS behandelt wird.

Abbildung 8 Das Ros-Ecosystem (Quelle: Ros, 2023)

Plumbing (Verrohrung)

ROS stellt ein Message-Passing System zur Verfügung, oft als „Middleware“ oder „Plumbing“ bezeichnet.[15] Eine Middleware ist Software, die sich zwischen Betriebssystem und darauf ausgeführten Anwendungen befindet. Es fungiert als Übersetzungsebene und ermöglicht so die Kommunikation und Datenverwaltung verteilter Anwendungen. Da Middleware Anwendungen miteinander verknüpft und so leicht Daten über eine Verbindung oder auch „pipe“ weitergegeben werden können, wird auch der Begriff Plumbing verwendet.[16] Auf ROS bezogen stellt die Middleware die Kommunikation zwischen den einzelnen Nodes (Roboteranwendungen) bereit.

Tools (Werkzeuge)

ROS bietet eine große Anzahl von Entwicklerwerkzeugen, welche das Erstellen von Roboteranwendungen erleichtern.

Capabilities (Fähigkeiten)

Das ROS-Ecosystem bietet eine Fülle von Robotersoftware, wie Gerätetreiber, Kartierungssysteme, Navigationsprogramme, um nur einige Beispiele zu nennen. Von Treibern über Algorithmen bis hin zu Benutzeroberflächen stellt ROS die passenden Bausteine für diverse Anwendungen bereit. Der Grund dafür liegt darin, dass die Einstiegshürde für die Entwicklung von Roboteranwendungen verringert werden soll. Mit ROS ist es möglich Robotersysteme zu entwickeln, ohne alles über die zugrunde liegende Hard- und Software verstehen zu müssen.

Community (Gemeinschaft)

Die ROS-Gemeinschaft besteht aus einem losen Zusammenschluss von Ingenieuren, Studenten und Hobbyentwicklern aus der ganzen Welt. Jeder kann seinen Teil dazu beitragen

[15] Vgl. (Ros, 2023)
[16] Vgl. (Azure.Microsoft, 2023)

und seine anwendungsspezifischen Lösungen öffentlich zugänglich machen. Dies sorgt für eine stetig wachsende Funktionalität und Verbesserungen bestehender Anwendungen.[17]

2.3.2 Funktionsweise von ROS

ROS ist ein modulares auf IP-basierendes Kommunikationsframework für Roboter und deren Sensoren und Aktoren. Die einzelnen Roboterkomponenten, wie z.B. Lidar-Sensor, Motoren, etc. werden als sogenannte Nodes implementiert und laufen als eigenständiger Prozess. Die Kommunikation zwischen den Nodes erfolgt direkt über eine Peer-to-Peer (P2P) - Verbindung. Eine P2P-Verbindung zeichnet sich durch eben dieses Datentransferstruktur aus. Die Daten laufen von einer Anwendung direkt zu einer anderen, ohne dabei einen zentralen Server zu passieren.[18] Allerdings müssen sich diese Nodes im Netzwerk auch finden können. Diese Aufgabe wird vom ROS Master übernommen, womit dieser eine zentrale Serverkomponente darstellt. Der Master agiert in einem ROS-Netzwerk jedoch nur als Register, um das gegenseitige Detektieren der Nodes zu ermöglichen. Sobald dies geschehen ist, kommunizieren die Nodes direkt miteinander, so dargestellt in Abbildung 9.[19]

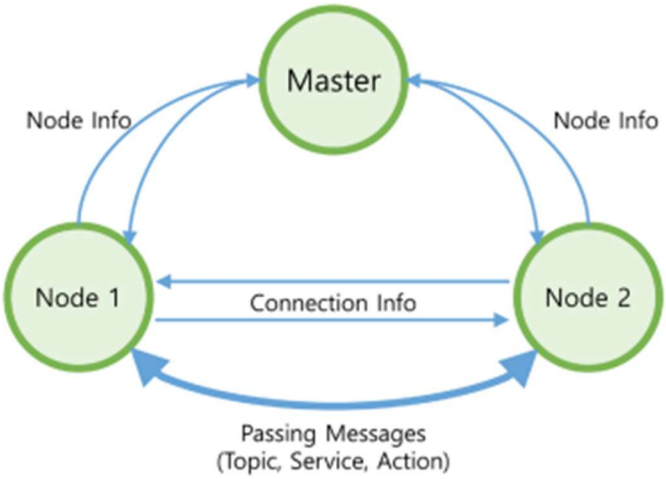


Abbildung 9 Nachrichten Kommunikation in ROS (Quelle: Pyo, Cho, Jung, & Lim, 2017, S.53)

Wie aus Abbildung 9 hervorgeht, gibt es drei unterschiedliche Varianten der Kommunikation von Nodes untereinander: Mit einem Topic, welches einen unidirektionalen[20] Nachrichtenempfang oder eine unidirektionale Nachrichtenübermittlung ermöglicht, mit einem Service, welcher eine bidirektionale Anfrage/Antwort-Nachricht bereitstellt, und mit einer Action, welche eine bidirektionale Ziel-/Ergebnis-/Feedback-Nachricht bereitstellt. Node-Parameter können ebenfalls als Kommunikationstyp gewertet werden, da die Parameter von

[17] Vgl. (Ros, 2023)
[18] Vgl. (xovi, 2023)
[19] Vgl. (Pohl, 2014, S. 2)
[20] Die Kommunikation erfolgt nur in eine Richtung.

außerhalb des Nodes modifiziert werden können. Abbildung 10 zeigt die Kommunikationstypen von Nodes.[21]

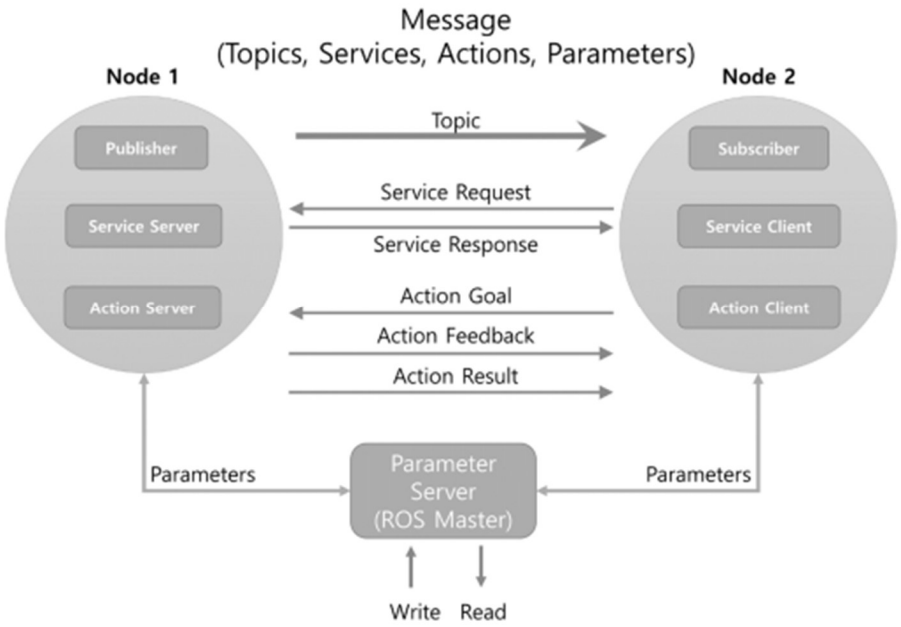


Abbildung 10 Nachrichten Kommunikation zwischen Nodes (Quelle: Pyo, Cho, Jung, & Lim, 2017, S.50)

Die einzelnen Kommunikationstypen, ausgenommen Parameter, werden im Folgenden kurz erläutert.

Topic

Bei diesem Kommunikationstypen gibt es zwei Parteien, einmal den Publisher-Node, welcher Daten über ein bestimmtes Topic sendet und zum anderen den Subscriber-Node, welche Daten über ein bestimmtes Topic empfängt. Abbildung 11 verdeutlicht das Kommunikationsschema. Die Verbindung wird über das Register im Master durchgeführt. Im Register steht, welche Nodes welchem Topic zugeordnet sind. Daraufhin wird die Verbindung zwischen den Nodes erstellt, sodass die Kommunikation direkt zwischen den Nodes stattfinden kann.

[21] Vgl. (Pyo, Cho, Jung, & Lim, 2017, S. 49)

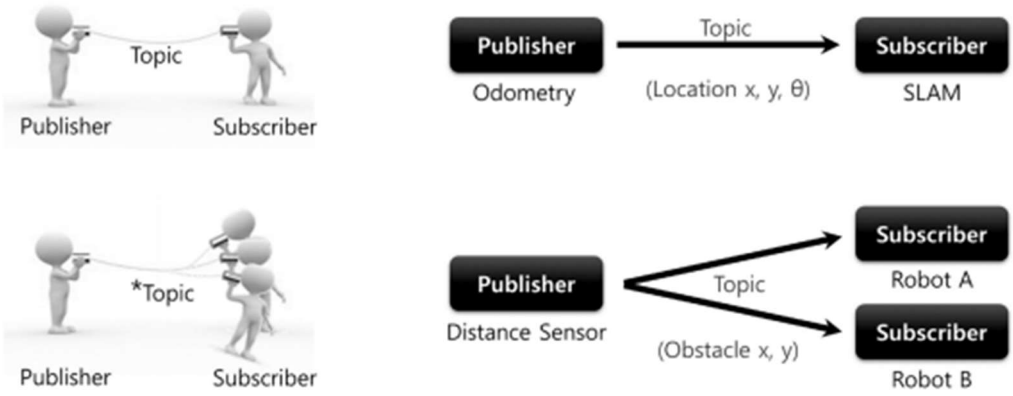


Abbildung 11 Topic Nachrichten Kommunikation (Quelle: Pyo, Cho, Jung, & Lim, 2017, S.51)

Da Topics unidirektional sind und eine ständige Verbindung haben, um kontinuierlich Daten zu senden, bietet sich die Verwendung dieses Nachrichtentyps bei Sensordaten an, welche periodisch gesendet/empfangen werden müssen. In Abbildung 11 werden als Beispiel die Odometriedaten, zur Ermittlung der aktuellen Position des Roboters von einem Publisher-Node über das Topic Location an den Subscriber-Node gesendet. Es ist ebenfalls möglich, dass mehrere Publisher Nachrichten über ein Topic senden, bzw. mehrere Subscriber Nachrichten über ein Topic empfangen können. Das zweite Beispiel zeigt eben diese Situation, indem Distanzsensordaten über ein Topic an zwei Subscriber-Nodes gesendet werden.

Service

Die Kommunikation über einen Service findet bidirektional und synchron zwischen dem Server-Client, welcher eine Service-Anfrage sendet und einem Service-Server, welcher auf diese Anfrage antwortet, statt. Abbildung 12 zeigt das Kommunikationsschema. Anders als beim Topic werden Nachrichten nicht kontinuierlich gesendet, sondern nur auf Anfrage. Nach erfolgreicher Anfrage und entsprechender Antwort wird die Verbindung der zwei kommunizierenden Nodes beendet. Dadurch kann die Auslastung des Netzwerkes verringert werden. Abbildung 12 zeigt eine beispielhafte Kommunikation zwischen dem Server und dem Client. Der Client fragt den Server nach der aktuellen Zeit (request), der Server prüft die aktuelle Zeit und sendet sie an den Client (response). Anschließend wird die Verbindung beendet.

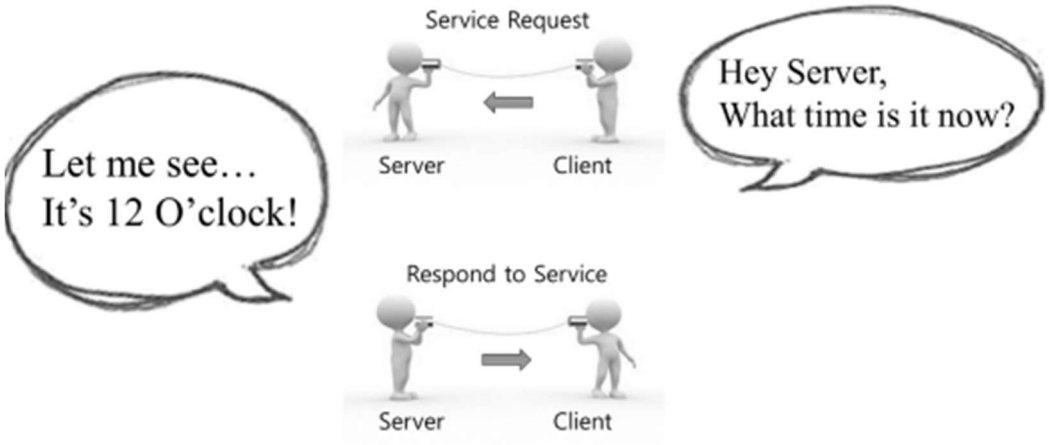


Abbildung 12 Service Nachrichten Kommunikation (Quelle: Pyo, Cho, Jung, & Lim, 2017, S.52)

Der Service wird oft eingesetzt, sobald ein Roboter oder ein Node eine bestimmte einmalige Aktion ausführen soll.

Action

Die Kommunikation über Action erfolgt ähnlich wie bei der Server-Kommunikation über einen Server und ein Client. Der Action-Client sendet ein Ziel (goal) und der Action-Server übermittelt bei Erfüllung ein Ergebnis (result). Der Unterschied ist, dass der Server dem Client periodisch, wie beim Topic, eine asynchrone bidirektionale Nachricht in Form von Feedback sendet. Also kann zu jeder Zeit nachverfolgt werden, in welchem Bearbeitungsstand sich der Server befindet. Diese Art der Kommunikation ist bei Aufgaben sinnvoll, die sehr viel Zeit in Anspruch nehmen. Abbildung 13 zeigt das Kommunikationsschema. Beispielhaft ist dort die Aufgabe des Action-Clients an den Action-Server den Haushalt zu machen. Während der Server seiner geforderten Aufgabe nachkommt, wird periodisch ein Feedback vom Server zum Client gesendet. Nach Erfüllung seiner Aufgabe sendet der Server ein Ergebnis. [22]

[22] Vgl. (Pyo, Cho, Jung, & Lim, 2017, S. 49-54)

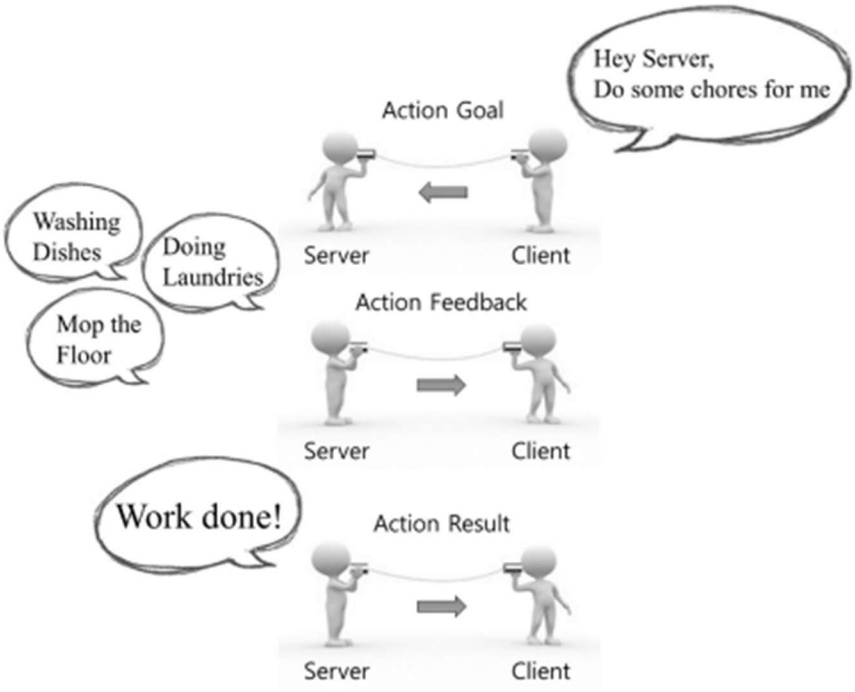


Abbildung 13 Action Nachrichten Kommunikation (Quelle: Pyo, Cho, Jung, & Lim, 2017, S.53)

Ziel dieses Kapitels war es eine Übersicht über die Kommunikationstypen innerhalb eines ROS-Netzwerkes zu geben. In den Praktikumsaufgaben, welche in Kapitel 4 folgen, werden diese Kommunikationstypen in verschiedenen Anwendungen verwendet.

2.4 Roboter-Navigation

Eines der Hauptmerkmale mobiler Roboter ist die Navigation. Diese beschreibt die Bewegung des Roboters von einem Punkt zu einem anderen innerhalb seiner Umgebung. Dazu muss der Roboter eine Karte seiner Umgebung besitzen, seine Lage innerhalb dieser Karte jederzeit kennen, immer den optimalen Weg von Punkt A zu Punkt B finden und Hindernissen ausweichen, die auf diesem Weg liegen. Zusammengefasst lässt sich sagen, dass die Grundvorrausetzungen des Roboters für die Navigation aus folgenden Punkten bestehen:

- Karte
- Pose des Roboters
- Sensordaten
- Pfadplanung und Fortbewegung[23]

Karte

Eine Karte ist eine genaue Repräsentation der Umgebung, in der sich der Roboter bewegen kann. Sie wird für die Lokalisierung des Roboters innerhalb dieser Karte verwendet. Dazu

[23] Vgl. (Pyo, Cho, Jung, & Lim, 2017, S. 313,314)

werden die Sensordaten des Roboters mit Kartenelementen, wie Punkte, Linien, etc. verglichen, um die Positionierung des Roboters zu bestimmen oder die Positioniergenauigkeit in Verbindung mit anderen Lokalisierungsalgorithmen zu optimieren. Eine weitere Anwendung für Karten ist die Pfadplanung, mittels welcher die optimale Route von einem Punkt zu einem anderen berechnet wird. Hierbei hilft die Karte mit Informationen zu Hindernissen, wie z.B. Wänden, um eine kollisionsfreie Bewegung des Roboters zu gewährleisten.[24] Eine Karte ist somit ein wichtiger Bestandteil eines mobilen Roboters. Oft ist die Karte der Umgebung beim Einsatz des Roboters jedoch nicht verfügbar. Der Roboter muss also über die Fähigkeit verfügen selbst eine Karte von seiner Umgebung zu erzeugen. Dies wird mit Hilfe von SLAM (Simultaneous Localization And Mapping; dt.: Simultane Positionsbestimmung und Kartierung) ermöglicht. SLAM erlaubt es, dass der Roboter eine Karte von einer ihm vorher unbekannten Umgebung erstellt und sich gleichzeitig in dieser Umgebung lokalisiert.[25]

Pose des Roboters

Der Roboter muss in der Lage sein, seine Pose zu bestimmen. Die Pose enthält die Position und die Orientierung des Roboters. Also wo sich der Roboter in einem Referenzsystem befindet und wie seine Ausrichtung relativ zum Referenzsystem ist. Eine oft gewählte Methode zur Bestimmung dieser Werte ist „Dead Reckoning“. Hierbei werden die Rotationen der Räder mit Inkrementalgebern gemessen, um eine Aussage über die Bewegung des Roboters treffen zu können. Die so berechnete Pose ist jedoch fehlerbehaftet und alleinstehend nicht exakt genug. Um dies auszugleichen werden die Sensordaten vom IMU hinzugezogen, um die berechnete Pose der realen Pose anzunähern. Die so erzeugte Pose ist hinreichend genau, um sie in Projekten der mobilen Robotik zu verwenden.

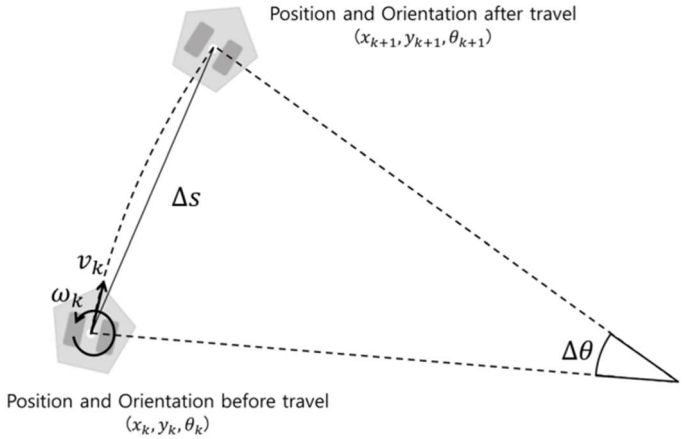


Abbildung 14 Dead Reckoning (Quelle: Pyo, Cho, Jung, & Lim, 2017, S. 315)

[24] Vgl. (Hertzberg, Lingemann, & Nüchter, 2012, S. 155,156)
[25] Vgl. (NavVis, 2023)

Abbildung 14 zeigt die relative Positions- und Orientierungsbestimmung auf einer Fläche mittels Dead Reckoning.

Sensordaten

Um Hindernisse zu erkennen, benötigt der Roboter Sensordaten von Distanz-Sensoren wie z.B. Lidar. Falls Objekte identifiziert werden sollen, können für diesen Zweck Vision-Sensoren wie z.B. Kameras oder Tiefenkameras verwendet werden. Die Erkennung von Hindernissen ist für die Navigation unabdingbar.

Pfadplanung und Fortbewegung

Der letzte wichtige Punkt für die Navigation von mobilen Robotern ist die Berechnung einer optimalen Route von einem Punkt zu einem anderen. Es gibt eine Reihe von Algorithmen, die genau diese Aufgabe übernehmen, darunter: Dynamic Window Approach (DWA), A* algorithm, potential field, particle Filter und RRT (Rapidly-exploring Random Tree).[26]

2.4.1 Lokalisierung innerhalb einer bekannten Karte

Die Navigation eines Roboters beinhaltet die Lokalisierung innerhalb eines Referenzsystems und die Pfadplanung von Punkt A zu Punkt B. In diesem Abschnitt soll es um die Lokalisierung des Roboters in einer bekannten Umgebung gehen. Die Karte der Umgebung ist somit vorhanden und der Roboter kann auf diese zugreifen. Um den Roboter in dieser Karte zu lokalisieren, können eine Reihe von Algorithmen angewendet werden. Häufig wird dazu die Monte-Carlo-Lokalisierung (MCL) verwendet. Dieser Algorithmus bildet die möglichen Posen des Roboters als Partikel innerhalb der Karte ab. Deshalb wird diese Methode auch als Partikelfilter bezeichnet. In der ersten Phase wird die Pose des Roboters, da sie noch nicht bekannt ist, randomisiert aber gleichverteilt über die ganze Karte angenommen. Abbildung 15 stellt diese Phase dar. Jeder rote Pfeil ist eine mögliche Pose des Roboters einschließlich einer Gewichtung. Die Gewichtung gibt eine Aussage über die Wahrscheinlichkeit, dass die Partikel die tatsächliche Pose des Roboters darstellt. Momentan besitzen alle Partikel die gleiche Gewichtung.

[26]Vgl. (Pyo, Cho, Jung, & Lim, 2017, S. 313-317)

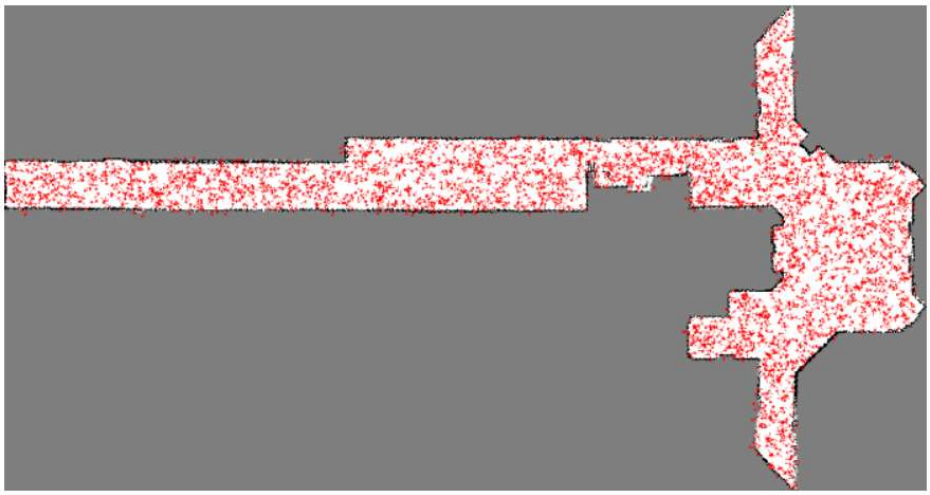


Abbildung 15 Monte-Carlo-Lokalisierung in einem Büro Flur; Roboter Pose über die Karte gleichverteilt (Quelle: Hertzberg, Lingemann, & Nüchter, 2012 S.208)

Führt der Roboter nun seine erste Messung durch, z.B. mit einem Lidarsensor, werden diese Daten mit den Partikeln verglichen. Partikel, die bei einer ersten Lidar-Messung ähnliche Daten erhalten würden, werden höher gewichtet und Partikel welche Daten erhalten würden, die nicht zu der tatsächlichen Messung passen, werden geringer gewichtet. Wird der Roboter zusätzlich bewegt, werden die Odometrie-Daten, also die Daten aus IMU und Inkrementalgeber, auf die Partikel angewendet. Bewegt sich der Roboter, bewegen sich auch die Partikel entsprechend. Nun werden die weniger gewichteten Partikel entfernt und durch neue Partikel, welche sich nahe der höher gewichteten Partikel befinden, ersetzt. Dieser Vorgang, ausgenommen der initialen randomisierten Anordnung von Partikeln, wiederholt sich ständig und ergibt nach kurzer Zeit eine relativ genaue Pose des Roboters innerhalb der Karte. Abbildung 16 zeigt die Ermittlung der realen Pose des Roboters innerhalb der Karte mit der Monte-Carlo-Lokalisierung.[27] [28]

[27] Vgl. (Hertzberg, Lingemann, & Nüchter, 2012, S. 207-210)
[28] Vgl. (Pyo, Cho, Jung, & Lim, 2017, S. 360-362)

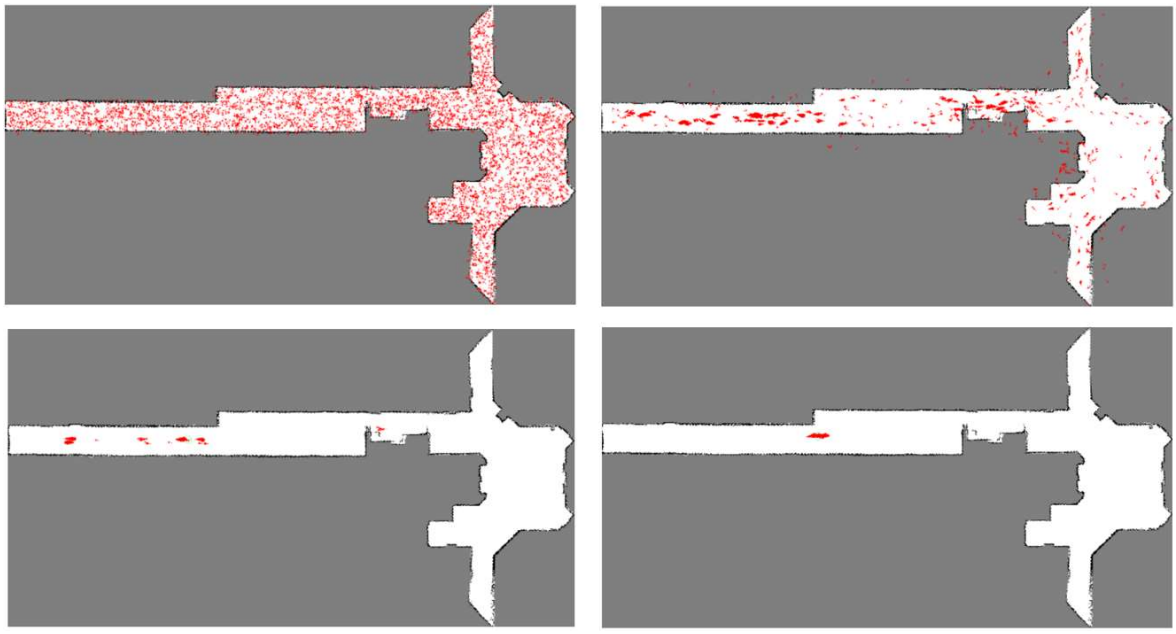


Abbildung 16 Roboter-Pose mit Monte-Carlo-Lokalisierung ermitteln (Quelle: Hertzberg, Lingemann, & Nüchter, 2012 S.208-210)

2.4.2 Lokalisierung und Kartierung innerhalb einer unbekannten Umgebung

Wenn keine Karte der Umgebung vorhanden ist, kann SLAM verwendet werden, um den Roboter innerhalb einer unbekannten Umgebung zu lokalisieren und zeitgleich die Umgebung zu kartieren. SLAM steht für Simultaneous Localization And Mapping (dt.: Simultane Positionsbestimmung und Kartierung). Bei dem SLAM-Vorgang handelt es sich jedoch um ein „Henne-Ei“ Problem, da für die Lokalisierung eine Karte benötigt wird. Um eine Karte erstellen zu können, wird wiederum eine Schätzung der Pose des Roboters innerhalb dieser Karte benötigt. Die SLAM-Problematik lässt sich mit einer Reihe von Verfahren lösen, darunter EKF (Extended Kalman Filter) SLAM, SLAM mit Partikelfiltern und Graph-basierende Verfahren. Bei SLAM mit Pose-Graph-Optimierung handelt es sich um ein Graph-basierendes Verfahren, welches im Folgenden kurz erläutert wird, um ein Lösungsansatz des Problems darzustellen. Ein Roboter, ausgestattet mit Lidar für die Umgebungswahrnehmung und IMU und Inkrementalgeber für die Odometriedaten, wird in einer unbekannten Umgebung ausgesetzt. Der Lidar-Sensor führt eine erste Messung durch und erhält Daten in Form einer Punktwolke von z.B. einer Wand und bildet diese Daten in einem Graphen ab. Fährt der Roboter weiter, wird die Bewegung des Roboters über die Odometriedaten gespeichert und auf die geschätzte Pose des Roboters innerhalb des Graphen angewandt. Da die Odometriedaten in den meisten Fällen Fehler aufweisen, entsteht eine Differenz von realer Pose zur geschätzten Pose. Der Roboter nimmt eine weitere Messung mit dem Lidar auf und bildet diese wieder im Graphen ab. Da jedoch eine Differenz zwischen realer und geschätzter Pose auftritt, wird die zweite Messung im Graphen um diese Differenz

verschoben angezeigt. Dadurch wird die Umgebung nicht korrekt abgebildet. Dies wird aber in den nächsten Schritten optimiert.

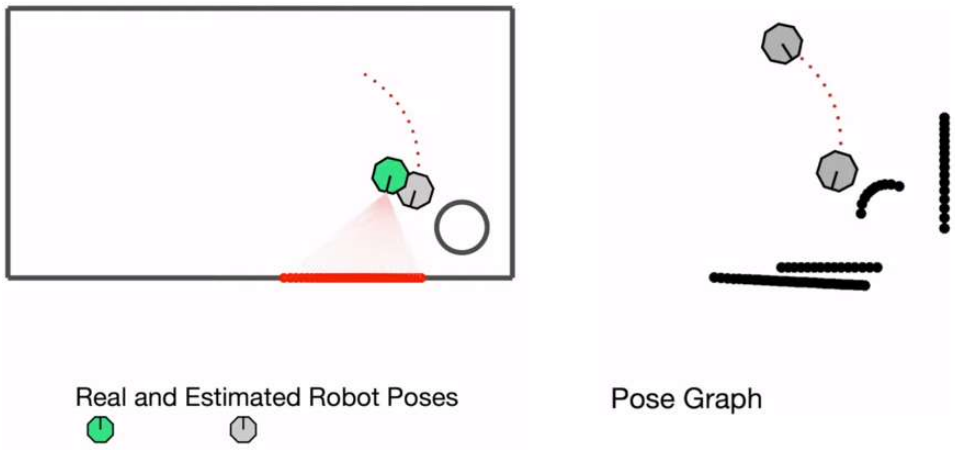


Abbildung 17 SLAM mit Pose Graph Optimization: Start der Kartierung (Quelle: Understanding SLAM Using Pose Graph Optimization | Autonomous Navigation, Part 3 von MATLAB https://www.youtube.com/watch?v=saVZtgPyyJQ)

In Abbildung 17 wird der Odometriefehler deutlich. Die im Pose Graph dargestellten Lidar-Messungen entsprechen nicht exakt der realen Umgebung. Da die Odometriedaten aber gespeichert werden, ist nun die ungefähre Entfernung und die Start- und Endpose relativ zueinander und relativ zur Lidar-Messung bekannt. Die zwei Posen im Graphen werden auch als Knoten bezeichnet. Die Knoten werden nun über eine Kante oder auch Constraint aneinandergebunden und die Länge der Kante entspricht der aus den Odometriedaten zurückgelegten Strecke. Diese Kante ist jedoch nicht fix und kann angepasst werden, sodass sich die Lage und Ausrichtung des angebundenen Knotens ebenfalls verändern kann. In welchem Maße sich diese Kante anpassen lässt, ist von der Sicherheit der Messungen abhängig. Sind die Odometriedaten sehr genau, lässt sich die Kante nur schwer verändern. Sind die Odometriedaten jedoch sehr ungenau, lässt sich die Kante leicht verändern. Der nächste Schritt ist, den Roboter weiter in der Umgebung zu bewegen. Bei jeder Messung werden Knoten und Kanten gebildet sodass nach einer Weile ein Pose Graph entsprechend Abbildung 18 entsteht.

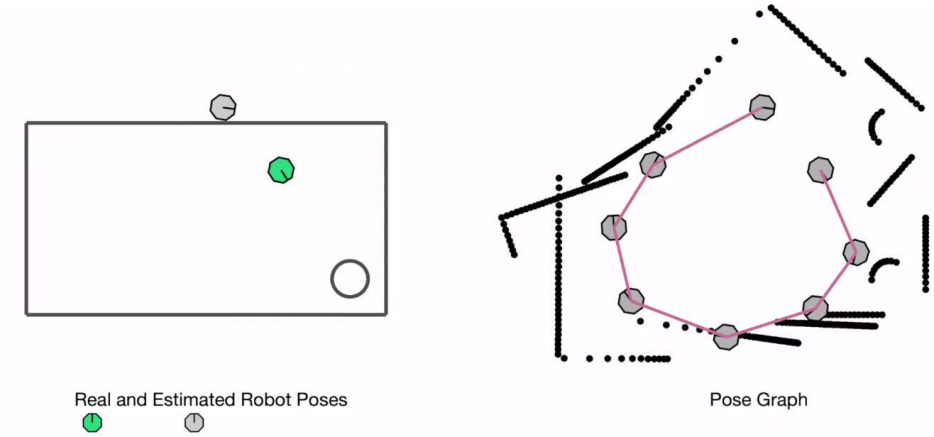


Abbildung 18 SLAM mit Pose Graph Optimization: Kartierungsprozess (Quelle: Understanding SLAM Using Pose Graph Optimization | Autonomous Navigation, Part 3 von MATLAB https://www.youtube.com/watch?v=saVZtgPyyJQ)

Wie in Abbildung 18 zu sehen ist, ist die Differenz zwischen realer und geschätzter Pose sehr groß und die Karte somit nicht verwendbar. Jedoch bilden die Lidar-Sensordaten bei der ersten und der aktuellen Pose die gleiche Punktwolke ab. Da die Lidar-Daten eine hohe Sicherheit aufweisen, kann gesagt werden, dass sich die letzte Pose auf der ersten Pose befinden muss. Die Knoten werden wieder mit einer Kante verbunden. Da diese Kante die Entfernung zwischen den Knoten darstellt, ist diese gleich null. Weil die Messung aufgrund der passenden Lidar-Sensordaten zwischen den ersten und letzten Knoten sehr genau ist, wird diese Kante sich nicht verändern. Die anderen Kanten können sich jedoch noch verändern, da die Messgenauigkeit auf Grundlage der Odometrie nicht sehr hoch waren. Wird nun die Entfernung des letzten Knotens und des ersten Knotens auf null gesetzt, verändern sich die vorherigen Knoten mit den dazugehörigen Messdaten entsprechend ihrer Messsicherheit und der Pose Graph wird optimiert (siehe Abbildung 19). Diese Optimierung findet jedes Mal bei einer „loop closure“ statt, also wenn der Roboter eine Stelle innerhalb der Karte erneut abfährt und sich die Lidarsensordaten mit einer vorherigen Lidar-Messung gleichen.

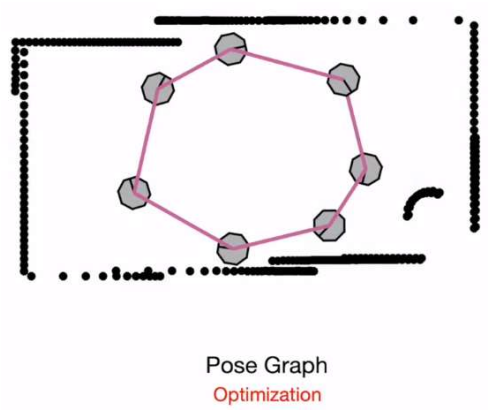


Abbildung 19 SLAM mit Pose Graph Optimization: Optimierungsprozess (Quelle: Understanding SLAM Using Pose Graph Optimization | Autonomous Navigation, Part 3 von MATLAB https://www.youtube.com/watch?v=saVZtgPyyJQ)

Nach jeder „loop closure“ wird die Karte weiter optimiert, bis eine hinreichend genaue Karte der Umgebung entsteht. Die Karte wird in einem Rasterfeld gespeichert, wobei blockierte Raster schwarz, freie Raster weiß und unbekannte Raster grau dargestellt werden (siehe Abbildung 20).[29] [30]

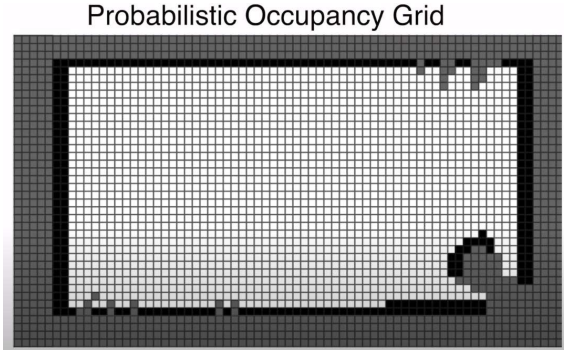


Abbildung 20 SLAM mit Pose Graph Optimization: Kartenspeicherung (Quelle: Understanding SLAM Using Pose Graph Optimization | Autonomous Navigation, Part 3 von MATLAB https://www.youtube.com/watch?v=saVZtgPyyJQ)

2.4.3 Pfadplanung

Bei der Pfadplanung geht es darum den optimalen Weg eines Roboters zwischen Start- und Zielpunkt zu ermitteln. Im Praktikumsversuch werden zwei Varianten in Kombination verwendet, um die Pfadplanung zu realisieren. Einmal eine globale Pfadplanung, welche den Pfad anhand der Karteninformationen ermittelt und eine lokale Pfadplanung, welche den Pfad anhand von Sensordaten des Roboters erstellt, um so auf plötzlich auftretende Hindernisse reagieren zu können.

Globale Pfadplanung

Für die globale Pfadplanung werden Algorithmen wie Dijkstra's Algorithmus und A* Algorithmus verwendet, um den optimalen Weg in einer bekannten statischen Karte zu finden. Dijkstra's Algorithmus verwendet Knoten als Orte und Kanten als Wege zu diesen Orten. Die Kanten haben dabei Kosten entsprechend des verbundenen Aufwandes diesen Weg zu gehen. Die Kosten können hierbei nur positive Werte annehmen. Die Idee hinter dem Algorithmus ist, dass die Kosten der günstigsten Wege vom Startknoten zu allen anderen Knoten berechnet werden. Die Funktionsweise ist dabei wie folgt: Der Algorithmus benötigt zur Ausführung eine Warteschlange in dem bekannte Knoten zwischengespeichert werden. Der Knoten mit den geringsten Kosten befindet sich ganz oben auf dieser Warteschlange. Zunächst wird der Algorithmus initialisiert. Der Weg vom Startknoten zum Startknoten beträgt 0. Die Wege zu den anderen Knoten sind noch nicht bekannt, weshalb diese Kosten mit unendlich angenommen werden. Der Startknoten wird auf die Warteschlange gesetzt.

[29] Vgl. (Burgard, Stachniss, Bennewitz, Tipaldi, & Spinello, 2013)
[30] Vgl. (Douglas, 2020)

Der vorderste Knoten auf der Warteschlange wird entnommen und betrachtet. Alle mit diesem Knoten verbundenen Knoten und deren Wege dorthin werden geprüft. Wenn die geprüften Knoten noch nicht auf der Liste sind, werden diese mit Informationen zum Vorgängerknoten und Wegkosten in die Warteschlange gesetzt. Sollten diese Knoten schon auf der Warteschlange sein, wird geprüft, ob der neue Weg günstiger ist und ggf. ersetzt. Es werden so lange Knoten aus der Warteschlange entnommen und deren nachfolgenden Knoten geprüft, bis sich keine Knoten mehr in der Warteschlange befinden oder ein Vordefiniertet Zielknoten erreicht wurde. [31]

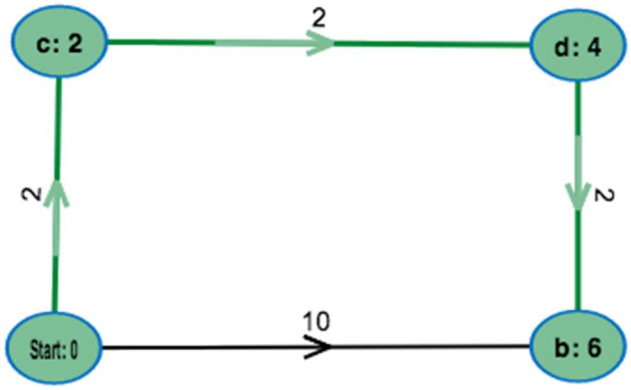


Abbildung 21 Dijkstra-Algorithmus (Quelle: https://algorithms.discrete.ma.tum.de/graph-algorithms/spp-dijkstra/index_de.html)

Abbildung 21 zeigt beispielhaft eine Pfadplanung. Der grüne Weg ist der günstigste Weg, um vom Starknoten zum Knoten b zu gelangen. Der Algorithmus geht in diesem Beispiel wie folgt vor:

- Die Kosten des Startknoten betragen 0 und die Kosten der anderen Knoten unendlich.
- Der Startknoten mit den Kosten 0 wird auf die Warteschlange gesetzt.
- Der Knoten mit den geringsten Gesamtkosten ist ganz oben in der Warteschlange, in diesem Fall der Startknoten.
- Alle angrenzenden Knoten werden überprüft. Der Knoten b ist über den Startknoten erreichbar. Die Kosten für diesen Weg betragen 10. Der Knoten b wird mit den Informationen zum Vorgängerknoten und zu den Kosten des Weges auf die Warteschlange gesetzt. Das gleiche geschieht mit Knoten c. Die Kosten wären 2 und der Weg führt über den Startknoten. Auch Knoten c wird auf die Warteschlange gesetzt.
- Die Warteschlange hat nun zwei Knoten. Knoten c und b. Da Knoten c geringere Gesamtkosten hat, wird dieser aus der Warteschlange genommen und seine Angrenzenden Knoten überprüft. Knoten d ist erreichbar und der Weg hat die Kosten

[31] Vgl. (Velden, algorithms.discrete.ma.tum[a], 2014)

2. Knoten d wird mit den Gesamtkosten 4 und Infos zu dem Vorgängerknoten gespeichert.
- Wieder hat die Warteschlange zwei Knoten. Knoten d und b. Da Knoten d die geringeren Gesamtkosten hat, wird dieser aus der Warteschlange genommen. Knoten b kann mit den Kosten 2 erreicht werden. Die Gesamtkosten belaufen sich auf den Wert 6. Nun ist der Knoten b schon auf der Warteschlange, aber mit den Gesamtkosten von 10. Der neue Weg ist damit günstiger und wird übernommen.
- Damit sind alle Kosten bekannt und der günstigste Weg kann durch die gespeicherten Infos zu den Vorgängerknoten rekonstruiert werden.

Bei diesem Verfahren werden alle möglichen Knoten in einem System gesucht. Dies kann zu einem unnötig hohen Aufwand führen, wenn der Zielknoten sich z.B. westlich befindet, der Algorithmus aber erst alle anderen Himmelsrichtungen nach dem kürzesten Weg zum Ziel absucht. Eine Verbesserung des Verfahrens ist der A* Algorithmus. Dieser funktioniert ähnlich wie der Dijkstra's Algorithmus mit dem Unterschied, dass nicht einfach der nächste erreichbare Knoten abgearbeitet wird, sondern der Knoten, der wahrscheinlich schneller zum Ziel führt.[32] Beide Algorithmen führen jedoch zu einer guten Pfadplanung und finden ihre Anwendung in der Navigation mobiler Roboter.

Lokale Pfadplanung

Die lokale Pfadplanung ermittelt ebenfalls den optimalen Weg zwischen Start- und Zielpunkt, mit dem Unterschied, dass dieser Planer nur eine kurze Strecke vorausplant und den Pfad abhängig von Sensordaten anpasst. Damit wird es möglich, dass der Roboter auf plötzlich auftretende Hindernisse reagieren kann, um eine Kollision zu vermeiden. Ein oft verwendeter Algorithmus zur Kollisionsvermeidung ist der Dynamic Window Approach (DWA). Dieser Algorithmus macht sich die bekannte Kinematik des Roboters zunutze und kann dadurch eine Vorhersage treffen, wo der Roboter, bei einer bestimmten Ausrichtung und Geschwindigkeit nach einer kurzen Zeit sein wird. Dadurch können ausgehend von der aktuellen Roboterpose und Geschwindigkeit eine Vielzahl von möglichen kurzen Pfaden erstellt werden, alle mit einer unterschiedlichen translatorischen und rotatorischen Geschwindigkeit. Da nun viele mögliche Pfade verfügbar sind, müssen diejenigen aussortiert werden, die nicht zielführend sind. Es werden unzulässige Pfade entfernt. Unzulässig sind Pfade, die zu einer Kollision mit einem Hindernis führen würden. Des Weiteren werden Pfade gestrichen, die so weit vorausplanen, dass der Roboter diesen Pfad in einem gegebenen kurzen Zeitintervall, aufgrund der maximalen Beschleunigung des Roboters nicht erreichen kann. Die gerade genannten Schritte erzeugen das namengebende dynamische

[32] Vgl. (Velden, algorithms.discrete.ma.tum[b], 2014)

Fenster, eine Ansammlung von möglichen Pfaden, die der Roboter in einem kurzen Zeitintervall unter Berücksichtigung der Umgebung und der Kinematik des Roboters abfahren kann. Dieser Prozess wird in Abbildung 22 dargestellt.

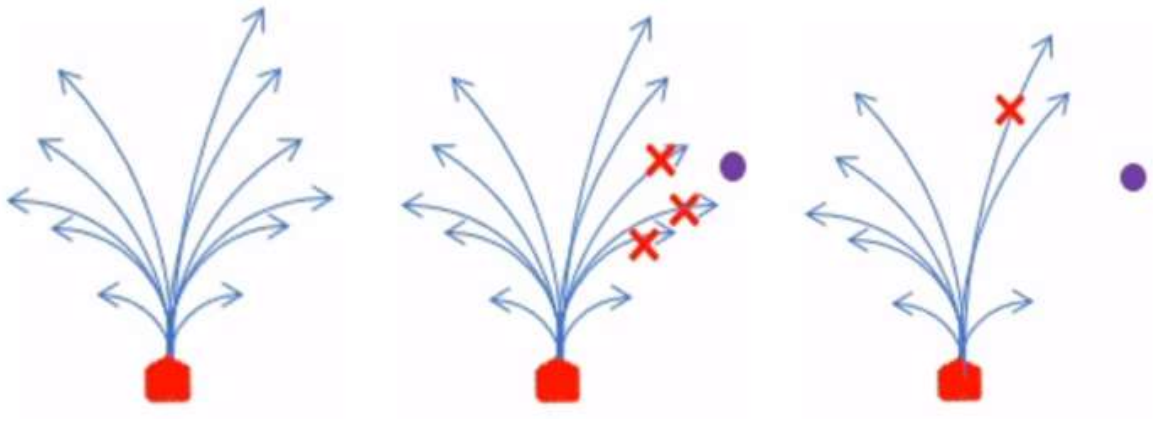


Abbildung 22 DWA mögliche Pfade eingrenzen (Quelle: https://www.youtube.com/watch?v=tNtUg-MBCh2g&t=101s)

Damit ist der erste Schritt des Algorithmus abgeschlossen und es folgt die Auswahl eines Pfades aus den gegebenen möglichen Pfaden durch eine Optimierungsfunktion. Der optimale Pfad wird anhand von drei Kriterien ausgewählt. Das erste Kriterium ist die Zielrichtung. Es wird bestimmt, welcher Pfad den Roboter am ehesten zum Ziel bringt. In Verbindung mit einem globalen Planer, wäre das Ziel, den globalen Plan zu folgen. Das zweite Kriterium beschreibt die Distanz zu einem Hindernis. Ist ein Pfad zu nah an einem Objekt wird ein anderer Pfad gewählt. Das letzte Kriterium ist die Geschwindigkeit des Roboters. Es wird der Pfad ausgewählt, der die höchste Geschwindigkeit erlaubt. Diese Kriterien werden in einer Optimierungsfunktion verarbeitet und bilden so den optimalen Pfad. Damit wurde der zweite Schritt des Algorithmus verrichtet und der optimale Pfad anhand der Kinematik des Roboters und der Umgebungsinformationen ausgewählt. Diese zwei Schritte werden ca. alle 0,25 Sekunden durchgeführt, wodurch eine hohe Dynamik in der Pfadplanung gegeben ist.[33]

Costmap

Mit Hilfe der Costmap wird eine funktionale Pfadplanung möglich. Die Costmap bestimmt, welche Bereiche in der Umgebung von dem Roboter befahren werden können und welche nicht. Dazu werden Informationen aus der statischen Karte und aus Sensordaten verwertet. Die Bereiche werden dabei in besetzte Bereiche, Bereiche bei der eine Kollision mit einem Hindernis möglich wären und freie Bereiche unterteilt. Die Costmap wird je nach

[33] Vgl. (Fox, Burgard, & Thrun, 1997)

verwendeter Navigationsart in zwei Teile aufgeteilt, die globale und die lokale Costmap. Die globale Costmap legt anhand der statischen Karte die befahrbaren Bereiche fest und bildet somit die Grundlage für die globale Pfadplanung mit den vorher genannten Algorithmen. Die lokale Costmap verwendet die Sensordaten des Roboters, um eine dynamische Aussage über die befahrbaren Bereiche in der Nähe des Roboters treffen zu können. Die lokale Costmap bildet die Grundlage für die lokale Pfadplanung mit dem DWA. Die Bereiche werden dabei mit Kostenwerten von 0 bis 255 versehen. Die Kostenwerte bilden die Bereiche wie folgt ab:

- 0: Freier Bereich. Der Roboter kann sich frei in dem Bereich bewegen
- 001~127: Bereiche mit einer geringen Kollisionswahrscheinlichkeit
- 128~252: Bereiche mit einer hohen Kollisionswahrscheinlichkeit
- 253~254: Kollisionsbereiche
- 255: Besetzter Bereich. Der Roboter kann diesen Bereich nicht befahren

Die Kostenwerte sind abhängig von den eingestellten Parametern, wie z.B. die Abmaße des Roboters. Des Weiteren können die Hindernisse künstlich aufgebläht werden, damit der Roboter einen größeren Abstand zu Hindernissen hält (Inflation).

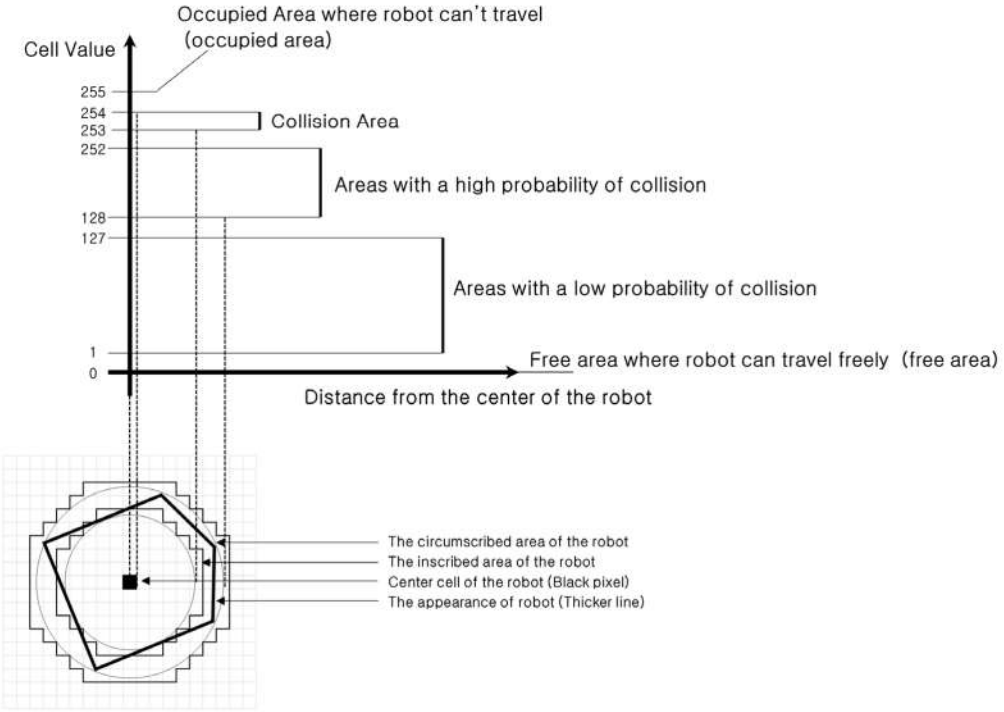


Abbildung 23 Costmap Zellenkosten (Quelle: Pyo, Cho, Jung, & Lim, 2017, S. 356)

Abbildung 23 verdeutlicht, wie sich die Kostenwerte der Bereiche in Abhängigkeit von der Entfernung des Roboters zu einem Hindernis zusammensetzen. Je näher der Mittelpunkt des Roboters einem Hindernis kommt, desto höher wird der Kostenwert für diesen Bereich. Dabei sind die Maße des Roboters bestimmend. Diese Informationen werden bei der

Ausführung von Navigationsanwendungen innerhalb der Karte dargestellt. Abbildung 24 zeigt einen Roboter (rote dünne Linie) innerhalb einer Karte. Die roten Punkte stellen die realen Hindernisse dar und der blaue Bereich die Vergrößerung der Hindernisse (Inflation).

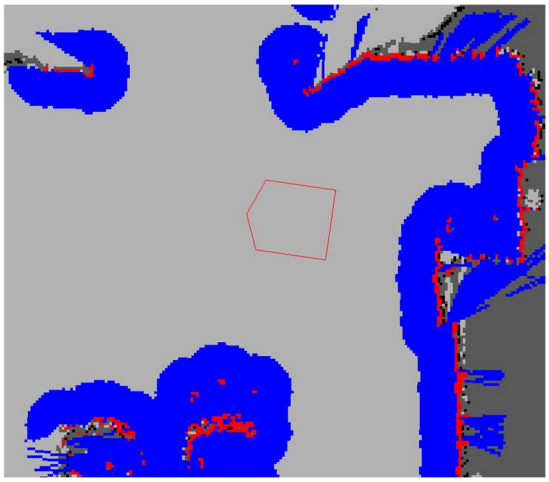


Abbildung 24 Costmap Darstellung innerhalb einer Karte (Quelle: Wiki.ros[a], 2018)

Um eine Kollision zu vermeiden, darf der Roboterumfang (rote dünne Linie) niemals eine rote Zelle berühren und der Robotermittelpunkt darf den blauen Bereich nicht überqueren.[34][35]

Damit sind die grundlegenden Methoden und Algorithmen, die für die Roboternavigation verwendet werden, bekannt.

3 Auswahlprozess eines geeigneten Systems

Durch das Grundlagenkapitel ist bis hierher bekannt, was ein mobiler Roboter ist, über welche wichtige Sensorik er für den Praktikumsversuch verfügen muss und welches Software-Entwicklungsprogramm im Versuch verwendet wird. Mit diesem Wissen kann eine Auswahl an Systemen für den Praktikumsversuch getroffen werden, was den Schwerpunkt dieses Kapitels darstellt.

[34] Vgl. (Wiki.ros[a], 2018)
[35] Vgl. (Pyo, Cho, Jung, & Lim, 2017, S. 355-357)

Anforderungen an ein passendes System sind:

- ROS-Kompatibel
- Lizenzfreie-Toolkette
- Sensorik:
    - IMU
    - Lidar-Sensor
    - Motor-Encoder (Inkrementalgeber)
- Preis bis ca. 2000€
- Umfangreiche Dokumentation

Bei den genannten Anforderungen handelt es sich um Mindestanforderungen. So kommen auch Systeme, die zusätzlich zum Lidar-Sensor über eine Kamera zur Bildauswertung verfügen für den Versuch ebenso infrage. Im Folgenden werden einige geeignete Systeme kurz vorgestellt.

System 1: Waveshare JetBot ROS AI Robot

Der JetBot ROS von Waveshare (siehe Abbildung 25) ist ein mit ROS kompatibler mobiler Roboter. Er verfügt über einen 360° 2D-Lidar-Sensor, womit eine Karte der Umgebung erstellt werden kann. Zusätzlich besitzt der Roboter eine IMU mit Beschleunigungsmesser und Gyroskop, sowie zwei Moter-Encoder zur Ermittlung der Antriebsradstellung und eine Kamera. Die Antriebsart ist ein 2WD, also ein Zweirad-Differentialantrieb. Die Kommunikation nach außen erfolgt über WLAN oder Bluetooth. Der verbaute Einplatinenrechner ist ein Jetson Nano. Dieser verfügt über ausreichend Rechenleistung, um alle gängigen Roboteranwendungen zu realisieren.[36] Der Jetson Nano ist zudem durch seine performante GPU für die Verwendung von Machine Learning geeignet.[37] Dies ist in diesem Praktikumsversuch nicht relevant, ist aber eventuell für spätere Praktikumsversuche im Bereich mobiler Robotik interessant. Die erforderliche Software ist auf der mitgelieferten SD-Karte vorinstalliert und als Download auf der Herstellerseite kostenlos verfügbar. Somit ist eine Lizenzfreie-Toolkette gegeben und es entstehen nach Erwerb keine Folgekosten.[38] Eine umfangreiche Dokumentation sowie Anleitungen für diverse Anwendungen sind auf der Herstellerseite verfügbar und erleichtern somit den Einstieg.

[36] Vgl. (Waveshare[a], 2023)
[37] Vgl. (ALL3DP, 2023)
[38] Vgl. (Waveshare[b], 2023)

Abbildung 25 Waveshare JetBot ROS (Quelle: Waveshare[a], 2023)

Der Preis beläuft sich auf 454,99$ (418,27€; Stand: 31.01.2023). [39] Der JetBot ROS bietet durch seine verbaute Sensorik und dem Jetson Nano ein gutes Gesamtpaket für den Start in die Mobile Robotik mit ROS.

System 2: Robotis TurtleBot3 Burger

Der TurtleBot3 Burger von Robotis (siehe Abbildung 26) ist vom Aufbau her ähnlich zu dem JetBot von Waveshare. Der ROS-kompatible Roboter von Robotis verfügt über einen 360° 2D-Lidar-Sensor zur Umgebungserkennung, IMU mit Beschleunigungsmesser und Gyroskop und Motor-Encoder zur Erkennung der Radbewegung. Als Netzwerkschnittstelle ist ein WLAN-Chip und ein Bluetooth-Chip verbaut. Der Antrieb ist auch hier ein Zweirad-Differentialantrieb. Der verbaute Einplatinenrechner ist ein Raspberry PI 4 mit wahlweise 2GB oder 4GB Arbeitsspeicher. Auch dieser Rechner liefert genug Rechenleistung, um alle gängigen Roboteranwendungen realisieren zu können. Die verwendete Software kann kostenlos auf der Herstellerseite heruntergeladen werden. Es entstehen keine Folgekosten für die Verwendung des Roboters, da auch hier eine Lizenzfreie Toolkette gegeben ist. Eine ausführliche Dokumentation, mit Installationshinweisen und diverse Anwendungen ist auf der Herstellerseite verfügbar.[40] Der Preis beläuft sich zwischen 804,44€ und 840,90€, je nach Größe des Arbeitsspeichers (Stand: 02.02.2023).[41] Auch dieser mobile Roboter eignet sich für den Einstieg in die Programmierung von mobilen Robotern mit ROS. Falls weitere Sensoren, wie z.B. eine Kamera gewünscht sind, können diese nachgerüstet werden.

[39] Vgl. (Waveshare[a], 2023)
[40] Vgl. (Robotis e-Manual[a], 2023)
[41] Vgl. (Generation Robots, 2023)

Abbildung 26 Robotis TurtleBot3 Burger (Quelle: Generation Robots, 2023)

System 3: Roboworks Rosbot Nano

Der Roboworks Rosbot Nano (siehe Abbildung 27) verfügt über einen 360° 2D-Lidar-Sensor zur Umgebungserkennung, IMU mit Beschleunigungsmesser und Gyroskop, Moter-Encoder und eine Tiefen-Kamera auch RGB-D Kamera genannt (Red Green Blue-Depth). Eine RGB-D Kamera kann die Tiefeninformationen eines Bildes bestimmten. Dadurch lassen sich Abstände zwischen Roboter und aufgenommen Objekten ermitteln.[42] Somit kann eine Karte der Umgebung auch ohne Lidar-Sensor erstellet werden.[43] Der Antrieb ist ein Ackermann-Antrieb, bei denen die Hinterräder durch Motoren angetrieben werden und die Steuerung durch einen Lenkwinkel der Vorderräder umgesetzt wird.[44] Der verbaute Einplatinenrechner ist ein Jetson Nano. Die benötigte Software ist bei Erhalt des Roboters schon auf dem Speichermedium installiert und einsatzbereit. Die Netzwerkschnittstellen sind WLAN und Bluetooth. Der Rosbot kommt, im Gegensatz zum JetBot und dem TurtleBot3, fertig montiert. Auch der Rosbot Nano ist mit ROS kompatibel und lässt sich somit einfach Programmieren. Der Preis für den Rosbot Nano beläuft sich auf 1991,66€ (Stand: 02.02.2023). [45] Der Rosbot Nano ist ebenfalls ein guter Roboter für den Einstieg in die ROS-Programmierung. Da er zusätzlich noch über eine Tiefenkamera verfügt kann z.B. die Kartenerstellung und Lokalisierung des Roboters in einem weiteren Praktikumsversuch über diese Kamera realisiert werden.

[42] Vgl. (Scanner-Imagefact, 2023)
[43] Vgl. (Mathworks, 2023)
[44] Vgl. (Bittel, 2023)
[45] Vgl. (Robotshop, 2023)

Abbildung 27 Roboworks Rosbot Nano (Quelle: Robotshop, 2023)

Selektiertes System

Auch wenn alle zuvor genannten Systeme für den Praktikumsversuch geeignet sind, wird nur der TurtleBot3 Burger von Robotis verwendet, da dieser zum Zeitpunkt der Erstellung des Versuches einen guten Kompromiss zwischen Funktionalität, Preis und Lieferzeit dargestellt hat.

4 Praktikumsversuch

Dieses Kapitel beschäftigt sich mit der Aufgabenstellung des Praktikumsversuchs und dessen Umsetzung. Der Versuch ist in vier Module unterteilt, welche in Gänze ein erstes Bild von der Arbeit mit mobilen Robotern und ROS vermitteln sollen. Jedes Modul hat seine eigene Aufgabenstellung. Als erstes folgt jedoch eine Installationsanleitung für das benötigte Betriebssystem und die Installation von ROS. Für den Praktikumsversuch wird der TurtleBot3 von Robotis verwendet. Dieser verfügt über eine ausreichende Funktionalität, um die Module des Versuchs umzusetzen.

4.1 Installationsanleitung Ubuntu/ROS/TurtleBot3

Um den Praktikumsversuch starten zu können, bedarf es einiger Vorarbeit. Das korrekte Betriebssystem und die passende ROS-Version müssen auf dem Computer, sowie dem Roboter installiert werden. Des Weiteren sind Netzwerkkonfigurationen vorzunehmen, um die IP-Kommunikation zwischen den Geräten zu gewährleisten. Dies wird im Folgenden erklärt. Die Anleitungen sind aus dem Online-Handbuch des TurtleBot3 entnommen und können dort nachgelesen werden, falls Komplikationen bei der Installation entstehen sollten.[46] Ein bestehendes internetfähiges WLAN-Netzwerk wird vorausgesetzt.

[46] Vgl. (Robotis e-Manual[a], 2023)

4.1.1 Ubuntu Installation

Die richtige Ubuntu Version hängt von der verwendeten ROS-Version ab. Der TurtleBot3 ist mit einer Reihe von ROS-Versionen kompatibel. Für diesen Praktikumsversuch wird ROS Noetic verwendet. Um diese Version ausführen zu können, wird Ubuntu 20.04 benötigt. Auf der TurtleBot3 Herstellerseite[47] ist ein Link für den Download der passenden Ubuntu Version vermerkt. Zu finden ist dieser unter Punkt 3.1.1. „Download and Install Ubuntu on PC“. Wichtig hierbei ist, dass im oberen Bereich der Webseite die passende ROS-Version ausgewählt wird, in diesem Fall „Noetic“. Nach Öffnen des Links erscheint die Ubuntu Website mit dem entsprechendem Downloadlink. Durch Klicken auf „64-bit PC (AMD64) desktop image“ wird der Download gestartet. Dieser hat eine Größe von 3,6GB und ist im ISO-Dateiformat. Um dieses ISO-Image auf einem Computer installieren zu können, muss das ISO-Image erst auf ein Speichermedium kopiert werden. Als Speichermedium kommt entweder eine CD oder ein USB-Stick infrage. Im Folgenden wird nur auf die Installation mit USB-Stick eingegangen. Der USB-Stick sollte über mind. 8GB Speicherplatz verfügen. Als erstes muss der USB-Stick bootfähig gemacht werden, damit der Computer Ubuntu vom USB-Stick aus installieren kann. Dies ist mit verschiedenen kostenlosen Tools, wie z.B. Rufus[48] oder Etcher[49] möglich. Diese Tools können kostenlos auf den jeweiligen Webseiten heruntergeladen werden. Folgend wird die Herangehensweise mit Rufus erläutert.

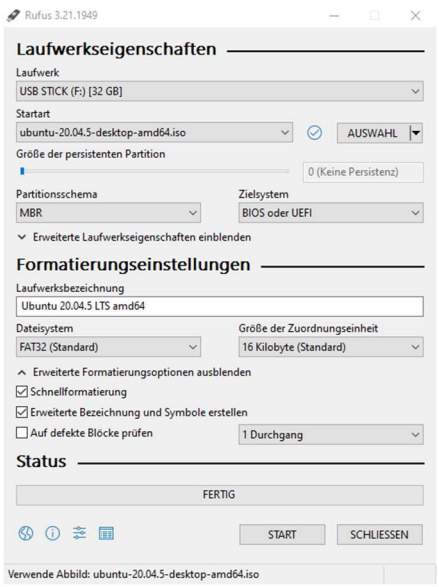


Abbildung 28 Bootfähigen USB-Stick mit Rufus erstellen

Nach dem Öffnen von Rufus erscheint eine Oberfläche wie in Abbildung 28 dargestellt. Nun muss der zu formatierende USB-Stick unter dem Punkt „Laufwerk“ ausgewählt und die

[47] https://emanual.robotis.com/docs/en/platform/turtlebot3/quick-start/#pc-setup
[48] https://rufus.ie/de/
[49] https://www.balena.io/etcher

passende ISO-Datei über „Auswahl“ hinzugefügt werden. Alle anderen Einstellungen können unverändert bleiben. Mit Klicken auf „Start“ wird der USB-Stick formatiert und ist im Anschluss einsatzbereit. Um Ubuntu auf dem Zielrechner zu installieren, muss der USB-Stick an den Rechner angeschlossen werden, bevor dieser gestartet wird. Während des Starvorganges wird durch Drücken der Taste F12, F10, F2, F1 oder ESC (je nach Hersteller unterschiedlich) das Bootmenü geöffnet. Hier kann der USB-Stick als Bootmedium ausgewählt werden und der Installationsprozess wird nach Schließen des Bootmenüs gestartet. Die geführte Ubuntu Installation ist in wenigen Minuten abgeschlossen. Falls Komplikation während der Installation auftreten sollten, kann eine komplette Ubuntu Installationsanleitung mit Etcher als USB-Formatierungstool auf der Ubuntu Website nachgeschlagen werden.[50] Anschließend folgt die Installation von ROS Noetic, welche im nächsten Kapitel beschrieben wird.

4.1.2 ROS-Installation und Netzwerkkonfiguration

Wie schon erwähnt, wird ROS Noetic in Verbindung mit dem TurtleBot3 Burger verwendet. Um diese ROS-Distribution herunterzuladen und zu installieren, werden die Downloadbefehle in das Ubuntu Terminal eingegeben. Das Terminal wird mit STRG+Alt+T geöffnet. Folgende Befehle müssen ausgeführt werden:

```
$ sudo apt update
$ sudo apt upgrade
$ wget https://raw.githubusercontent.com/ROBOTIS-GIT/ro-
botis_tools/master/install_ros_noetic.sh
$ chmod 755 ./install_ros_noetic.sh
$ bash ./install_ros_noetic.sh
```

Hinweis: Im Folgenden werden die Ubuntu-Terminalbefehle immer in dieser Form dargestellt. Das $-Zeichen wird nicht mit übernommen, es sagt nur aus, dass es sich um einen eigenständigen Befehl handelt. Jeder Befehl muss einzeln eingegeben und ausgeführt werden. Bei Befehlen mit sudo wird nach dem Benutzerpasswort gefragt, bevor dieser ausgeführt wird. Sudo führt Befehle mit Administrator-Rechten aus.

Mit den ersten beiden Befehlen wird überprüft, ob Ubuntu und alle installierten Programme auf dem aktuellen Stand sind. Die nachfolgenden Befehle starten die Installation. Im Anschluss müssen ROS spezifische Packages installiert werden. Packages sind Softwareprogramme, in diesem Fall für ROS. Ein Package kann z.B. aus ROS Nodes, Konfigurationen, Datensätzen, Drittanbietersoftware, etc bestehen[51]. Die folgenden Befehle installieren die benötigten ROS-Packages:

[50] https://ubuntu.com/tutorials/install-ubuntu-desktop#1-overview
[51] Vgl. (Wiki.ros[b], 2019)

```
$ sudo apt-get install ros-noetic-joy ros-noetic-teleop-twist-joy \
ros-noetic-teleop-twist-keyboard ros-noetic-laser-proc \
ros-noetic-rgbd-launch ros-noetic-rosserial-arduino \
ros-noetic-rosserial-python ros-noetic-rosserial-client \
ros-noetic-rosserial-msgs ros-noetic-amcl ros-noetic-map-server \
ros-noetic-move-base ros-noetic-urdf ros-noetic-xacro \
ros-noetic-compressed-image-transport ros-noetic-rqt* ros-noetic-
rviz \
ros-noetic-gmapping ros-noetic-navigation ros-noetic-interactive-
markers
```

Zum Schluss ist es noch erforderlich einige TurtleBot3 spezifische Packages herunterzuladen, mit den Befehlen:

```
$ sudo apt install ros-noetic-dynamixel-sdk
$ sudo apt install ros-noetic-turtlebot3-msgs
$ sudo apt install ros-noetic-turtlebot3
```

Die ROS-Installation auf dem Computer ist damit abgeschlossen. Da der ROS Master und die ROS Nodes über ein IP-Netzwerk kommunizieren, müssen die passenden Netzwerkkonfigurationen am PC vorgenommen werden. Als erstes gilt es die IP-Adresse des Computers herauszufinden. Hierzu wird der Computer über WLAN oder Ethernet-Kabel an ein bestehendes Netzwerk angeschlossen. Das Netzwerk sollte hierbei das im Praktikumsversuch verwendete Netzwerk sein, da sich die IP-Adresse des Computers je nach Netzwerk unterscheiden kann und damit die Einstellungen hinfällig wären. Mit dem Terminalbefehl**:**

```
$ ifconfig
```

werden alle netzwerkrelevanten Informationen angezeigt, darunter auch die IP-Adresse des Computers. Die IP-Adresse steht neben dem Tag „inet addr“ und sieht in etwa wie folgt aus: 192.168.0.100 (siehe Abbildung 29).

```
wlp2s0    Link encap:Ethernet  HWaddr ac:2b:6e:6d:08:ee
          inet addr:192.168.0.100  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::77a:7d5c:9ca8:bd9c/64 Scope:Link
```

Abbildung 29 IP-Adresse Computer

Im Anschluss wird die IP-Adresse des Computers an ROS übergeben und der Computer wird als ROS Master deklariert. Dies geschieht in der Konfigurations-Datei bashrc. Mit dem Befehl:

```
$ nano ~/.bashrc
```

wird diese Datei geöffnet und kann bearbeitet werden. Die IP-Adresse muss nun, wie in Abbildung 30 dargestellt, neben dem Punkt „export ROS_MASTER_URI“ und „export ROS_HOSTNAME“ eingefügt werden. Die „:11311“ beschreibt den verwendeten Port und kann übernommen werden.

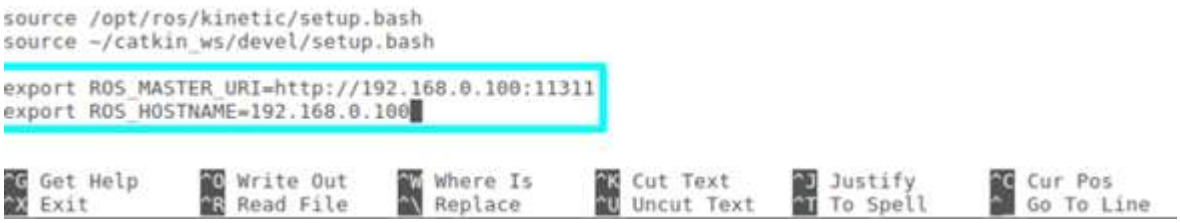


Abbildung 30 Netzwerkkonfiguration für ROS Master

Mit STRG+S wird die Datei gespeichert und mit STRG+X geschlossen. Damit die Änderungen wirksam werden, muss im Anschluss der Befehl:

```
$ source ~/.bashrc
```

eingegeben werden. Damit ist ROS installiert und die nötigen Netzwerkeinstellungen für den Computer wurden vorgenommen. Im nächsten Schritt werden alle nötigen Vorbereitungen für den TurtleBot3 besprochen.

4.1.3 TurtleBot3 Burger Installation und Netzwerkkonfiguration

Auch bei dem TurtleBot3 bedarf es einiger Vorarbeit. Es muss das passende Betriebssystem installiert und es müssen Netzwerkeinstellungen vorgenommen werden, damit der TurtleBot3 mit dem ROS Master kommunizieren kann. Auf der Herstellerseite ist ein Link[52] mit dem passenden Betriebssystem aufgeführt. Dieser ist unter dem Punkt 3.2.2. auf der Herstellerseite zu finden. Es ist darauf zu achten, dass ROS Noetic ausgewählt ist. Je nach verwendeten Raspberry Pi Modell gibt es unterschiedliche Versionen. Der verwendete TurtleBot3 besitzt ein Raspberry Pi 4, also ist dementsprechend der passende Link auszuwählen. Nach dem Download ist das Betriebssystem auf einer SD-Karte zu installieren. Dazu muss die SD-Karte an einen Ubuntu Computer angeschlossen werden. Um das Betriebssystem auf der SD-Karte zu installieren, können verschiedene Tools verwendet werden. In diesem Fall wird der Raspberry Pi Imager verwendet.

[52] https://emanual.robotis.com/docs/en/platform/turtlebot3/sbc_setup/#sbc-setup

Abbildung 31 Raspberry Pi Imager (Quelle: https://www.monsterli.ch/blog/hardware/raspberry-pi/raspberry-pi-os-lite-installieren-mit-dem-raspberry-pi-imager/)

Der Raspberry Pi Imager ist über das Download Center „Ubuntu Software“ verfügbar und erlaubt die Installation von Betriebssystemen auf beliebigen Speichermedien. Nach dem Starten des Imagers öffnet sich das in Abbildung 31 dargestellte Fenster. Durch Klicken auf „CHOOSE OS“ öffnet sich ein Dropdown-Menü aus dem „Use custom“ auszuwählen ist. Im Anschluss kann das zuvor heruntergeladene Betriebssystem selektiert werden. Unter „CHOOSE SD CARD“ ist das Zielmedium zu bestimmen, in diesem Fall die SD-Karte. Durch Klicken auf „WRITE“ wird das Betriebssystem auf die SD-Karte geschrieben. Das System verbraucht ca. 5GB Speicherplatz auf der SD-Karte. Restlicher Speicher auf dem Medium ist nicht zugewiesen und kann somit nicht verwendet werden. Um dies zu beheben und den Speicherplatz freizugeben, ist es nötig die vom Imager auf der SD-Karte erstellte Partition anzupassen. Hierfür kann der Partitions-Editor „GParted“ verwendet werden. Auch dieser ist in „Ubuntu Software“ verfügbar.

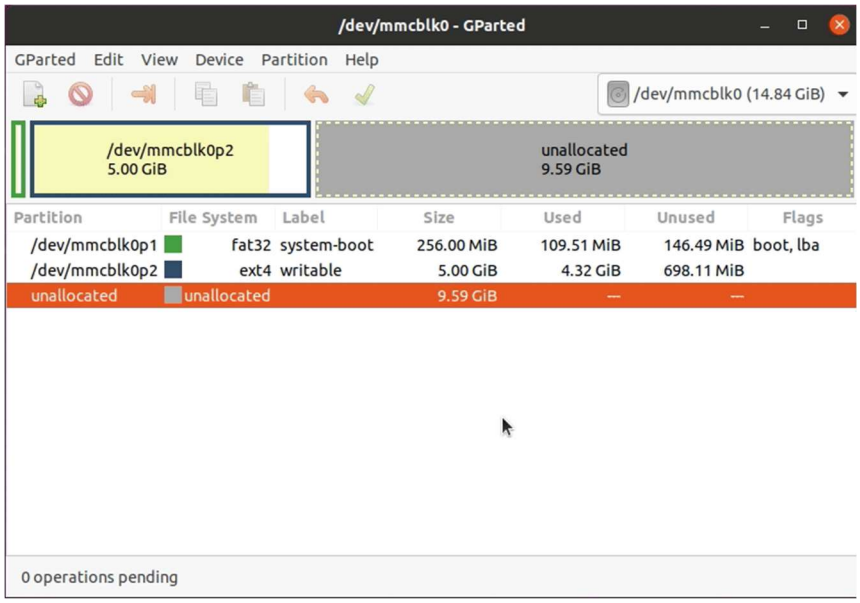


Abbildung 32 Partition anpassen mit GParted (Quelle: https://emanual.robotis.com/docs/en/platform/turtlebot3/sbc_setup/#sbc-setup)

Nach Öffnen von GParted (siehe Abbildung 32) sind folgende Schritte notwendig, um die Partition anzupassen:

1. SD-Karte aus dem Dropdownmenü auswählen
2. Rechtsklick auf die gelbe Partition
3. Resize/Move auswählen
4. Mit dem Mauscursor die Kante der gelben Partition komplett nach rechts ziehen
5. Auf den grünen „Apply All Operations“ Haken klicken und bestätigen

Nun ist die Partition der SD-Karte angepasst und der komplette Speicher ist verfügbar. Zuletzt sind noch die Netzwerkkonfigurationen vorzunehmen. Als erstes werden die WLAN-Zugangsdaten des Zielnetzwerkes eingegeben. Dazu muss das netplan directory auf der SD-Karte geöffnet und die 50-cloud-init.yaml-Datei bearbeitet werden. Folgende Terminalbefehle öffnen die Datei:

```
$ cd /media/$USER/writable/etc/netplan
$ sudo nano 50-cloud-init.yaml
```

In der Datei ist WIFI_SSID durch den Namen des WLAN-Zugangspunktes zu ersetzen und WIFI_PASSWORD durch das entsprechende WLAN-Passwort (siehe Abbildung 33). Gespeichert wird die Datei mit STRG+S und geschlossen mit STRG+X.

Abbildung 33 WLAN-Zugangsdaten einfügen für TurtleBot3 (Quelle: https://emanual.robotis.com/docs/en/platform/turtlebot3/sbc_setup/#sbc-setup)

Im Anschluss kann die SD-Karte aus dem Computer entfernt und in den Raspberry Pi eingesetzt werden. Die jetzt folgenden Anleitungen erfolgen direkt auf dem Raspberry Pi. Es ist dazu notwendig einen Monitor via HDMI, Tastatur via USB und eine Stromversorgung an den Raspberry Pi anzuschließen. Nach Anschließen der Stromversorgung, startet der Raspberry Pi automatisch mit dem Bootvorgang. Nachdem dieser abgeschlossen ist, wird nach den Anmeldedaten gefragt. Hier ist die Login ID: **ubuntu** und das Passwort: **turtlebot** einzutragen. Auch auf dem Raspberry müssen Netzwerkkonfigurationen vorgenommen

werden, damit ROS die IP-Adresse des Masters und die IP-Adresse des Raspberrys kennt. Dazu muss als erstes die IP-Adresse des Gerätes ermittelt werden. Hier kann ebenfalls der Befehl

```
$ ifconfig
```

eingegeben werden, um diese herauszufinden. Im Anschluss wird mit dem Befehl

```
$ nano ~/.bashrc
```

die Konfigurationsdatei geöffnet, um diese bearbeiten zu können. Bei den Zeilen:

- export ROS_MASTER_URI=http://{IP_ADDRESS_OF_REMOTE_PC}:11311
- export ROS_HOSTNAME={IP_ADDRESS_OF_RASPBERRY_PI_3}

ist der Teil {IP_ADDRESS_OF_REMOTE_PC} durch die IP-Adresse des ROS-Masters zu ersetzen und {IP_ADDRESS_OF_RASPBERRY_PI_3} durch die IP-Adresse des Raspberry Pis. Die Datei wird mit STRG+S gespeichert und mit STRG+X geschlossen. Im Anschluss müssen die Änderungen mit dem Befehl:

```
$ source ~/.bashrc
```

wirksam gemacht werden. Die Netzwerkkonfigurationen sind damit abgeschlossen und der TurtleBot3 kann nun mit dem ROS-Master Computer kommunizieren.

Da der TurtleBot3 seit 2022 mit einem neuen LDS-Sensor ausgestattet wurde und der Treiber noch nicht implementiert ist, muss dieser aktualisiert werden. Dies erfolgt mit den folgenden Terminalbefehlen:

```
$ sudo apt update
$ sudo apt install libudev-dev
$ cd ~/catkin_ws/src
$ git clone -b develop https://github.com/ROBOTIS-
GIT/ld08_driver.git
$ cd ~/catkin_ws/src/turtlebot3 && git pull
$ rm -r turtlebot3_description/ turtlebot3_teleop/
turtlebot3_navigation/ turtlebot3_slam/ turtlebot3_example/
$ cd ~/catkin_ws && catkin_make
```

Anschließend muss die Konfigurationsdatei bashrc wie folgt angepasst werden:

```
$ echo 'export LDS_MODEL=LDS-02' >> ~/.bashrc
$ source ~/.bashrc
```

Damit ist die Installation des Raspberry Pis abgeschlossen. Anschließend muss die Firmware des Open CR Board installiert werden. Das Open CR Board ist die Schnittstelle zwischen dem Raspberry Pi und den Antriebsmotoren. Zudem beinhaltet das Board die IMU-Sensoren. Die Installation der Firmware wird mit genannten Schritten vorgenommen:

1. Installation der erforderlichen Packages auf dem Raspberry, um die Firmware auf dem OpenCR Board hochladen zu können:

```
$ sudo dpkg --add-architecture armhf
$ sudo apt-get update
$ sudo apt-get install libc6:armhf
```

2. Verwendeter Port und verwendetes TurtleBot3 Modell definieren:

```
$ export OPENCR_PORT=/dev/ttyACM0
$ export OPENCR_MODEL=burger_noetic
$ rm -rf ./opencr_update.tar.bz2
```

3. Firmware und Loader herunterladen und die Dateien entpacken:

```
$ wget https://github.com/ROBOTIS-GIT/OpenCR-Binaries/raw/mas-
ter/turtlebot3/ROS1/latest/opencr_update.tar.bz2
$ tar -xvf opencr_update.tar.bz2
```

4. Firmware auf dem OpenCR Board hochladen:

```
$ cd ./opencr_update
$ ./update.sh $OPENCR_PORT $OPENCR_MODEL.opencr
```

Eine erfolgreiche Installation sollte in etwas wie folgt aussehen:

```
opencr_update/burger_turtlebot3_core.ino.bin
opencr_update/waffle_turtlebot3_core.ino.bin
opencr_update/opencr_ld_shell_arm
opencr_update/update.sh
opencr_update/
opencr_update/opencr_ld_shell_x86
opencr_update/waffle.opencr
opencr_update/burger.opencr
opencr_update/released_1.0.17.txt
armv7l
arm
OpenCR Update Start..
opencr_ld_shell ver 1.0.0
opencr_ld_main
[  ] file name          : burger.opencr
[  ] file size          : 172 KB
[  ] fw_name            : burger
[  ] fw_ver             : 1.0.17
[OK] Open port          : /dev/ttyACM0
[  ]
[  ] Board Name         : OpenCR R1.0
[  ] Board Ver          : 0x17020800
[  ] Board Rev          : 0x00000000
[OK] flash_erase        : 0.92s
[OK] flash_write        : 1.84s
[OK] CRC Check          : 11A1E12 11A1E12 , 0.005000 sec
[OK] Download
[OK] jump_to_fw
turtlebot@turtlebot:~$
```

Abbildung 34 Erfolgreiches OpenCR Firmwareupdate (Quelle: https://emanual.robotis.com/docs/en/platform/turtlebot3/opencr_setup/#opencr-setup)

Zu Überprüfungszwecken können die Tasten SW1 oder SW2 auf dem OpenCR Board gedrückt werden. Dadurch werden die Radmotoren angesteuert und der Roboter fährt entweder ein kurzes Stück vorwärts oder dreht sich 180° um sich selbst.

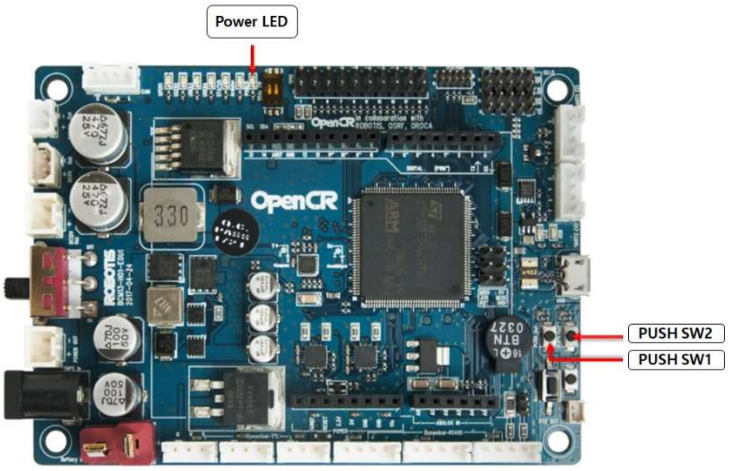


Abbildung 35 OpenCR Board Test (Quelle: https://emanual.robotis.com/docs/en/platform/turtlebot3/opencr_setup/#opencr-setup)

Abbildung 35 zeigt das OpenCR Board mit den zu drückenden Tasten SW1 und SW2. Folgende Tests können durchgeführt werden:

- Taste SW1 für einige Sekunden drücken → Roboter bewegt sich 30cm vorwärts
- Taste SW2 für einige Sekunden drücken → Roboter dreht sich 180° um sich selbst

Hat der Roboter die oben genannten Aktionen ausgeführt, ist der TurtleBot3 korrekt montiert. Damit sind nun alle Vorbereitungen für sowohl den ROS-Master als auch dem TurtleBot3 durchgeführt. Alle Einstellungen und Installationen müssen jeweils nur einmal durchgeführt werden. Ändert sich jedoch das Netzwerk, in dem die Geräte kommunizieren, müssen die Netzwerkeinstellungen angepasst werden.

4.1.4 Zusatz: Feste IP-Adresse vergeben

Bei der Verwendung eines handelsüblichen Routers werden die IP-Adressen der Teilnehmer mittels DHCP automatisch vergeben.[53] Es kann vorkommen, dass sich die automatisch zugewiesenen IP-Adressen ändern, wenn sich die Geräte nach einiger Zeit neu verbinden müssen. Darum ist es hilfreich den Geräten eine feste IP-Adresse zuzuweisen, damit die Netzwerkkonfiguration nicht erneut durchgeführt werden muss. Der Vorgang wird hier an einem TP-Link Router bespielhaft dargestellt. Die Durchführung sollte sich bei den meisten Routern ähneln. Als erstes muss der Computer und der TurtleBot3 über eine aktive Router-Verbindung verfügen. Im Anschluss werden die Router-Einstellungen geöffnet, indem die IP-Adresse des Routers in einen Web-Browser eingegeben wird. Um die IP-Adresse des Routers herauszufinden, kann der Befehl:

```
$ arp -a
```

[53] Vgl. (Jaeger, 2022)

eingegeben werden. Die IP-Adresse des Routers steht neben „_gateway“ und entspricht oft 192.168.0.1. Nachdem diese Adresse im Browser eingegeben wurde, öffnet sich ein Anmeldefenster für den Router. Der Login und das Passwort befinden sich auf der Rückseite des Routers. Wenn diese nicht stimmen, muss der Router auf Werkseinstellungen zurückgesetzt werden. In der Benutzeroberfläche befindet sich unter „DHCP“ ein Punkt namens „DHCP Client List“. Dort sind alle verbundenen Geräte aufgeführt, darunter auch der ROS-Master-Computer und der TurtleBot3 (ubuntu).

Abbildung 36 DHCP-Client Liste

Neben dem Namen sind die MAC- und die zugewiesenen IP-Adressen abgebildet. Unter dem Menüpunkt „Adress Reservation“ kann den Geräten eine feste IP-Adresse zugewiesen werden. Dazu muss zu diesem Menüpunkt navigiert werden und der Button „Add New“ geklickt werden. Es öffnet sich eine Eingabemaske, in der die MAC-Adresse und die gewünschte IP-Adresse des jeweiligen Gerätes eingegeben werden können. Die MAC-Adresse ist der DHCP-Client Liste zu entnehmen. Es ist zweckdienlich die via DHCP zugewiesenen IP-Adressen zu übernehmen, da diese für die Netzwerkkonfiguration verwendet wurden. Nun besitzen die Geräte immer dieselbe IP-Adresse und die Netzwerkkonfiguration müssen, sofern der Router nicht gewechselt oder zurückgesetzt wird, nicht erneut durchgeführt werden.

4.2 Praktikumsmodul 1: Publisher/Subscriber-Modell

Das Kommunikationskonzept innerhalb von ROS wurde in Kapitel 2.3.2 erklärt, darunter auch was ein Subscriber-Node und ein Publisher-Node ist. Um eine Einführung in die Arbeit mit ROS zu geben, soll im Folgenden ein Publisher/Subscriber-Modell in Python implementiert werden. Die Funktion ist dabei wie folgt: Der Publisher-Node soll periodisch eine Textnachricht an den Subscriber-Node senden. Grundlage dieses Moduls ist das ROS-Noetic Tutorial auf der Webseite wiki.ros.[54] Diese und weiterführende Erklärungen können dort eingesehen werden.

[54] Vgl. (Wiki.ros[c], 2022)

4.2.1 Catkin Workspace erstellen

Als erstes muss eine ROS-Arbeitsumgebung erstellt werden, indem die erstellten Programme (Packages) laufen können. Diese Arbeitsumgebung nennt sich Catkin Workspace. Der Catkin Workspace ist ein Ordner (directory) in dem ROS-Packages erstellt oder bearbeitet werden können. Das Catkin-Tool vereinfacht dabei den Installationsprozess für eben diese ROS-Packages.[55][56] Catkin ist bei der ROS-Installation schon enthalten und muss nicht extra hinzugefügt werden. Um den Catkin Workspace zu erstellen, sind die anschließenden Schritte durchzuführen:

1. ROS-Version sourcen.
   Um die ROS-Version zu sourcen, wird folgender Terminalbefehl eingegeben:

```
$ source /opt/ros/noetic/setup.bash
```

   Der "source"-Befehl führt die Kommandos aus der angegeben Datei im verwendeten Terminal aus. In diesem Fall werden die Kommandos aus der "setup.bash"-Datei ausgeführt, welche die ROS-Version festlegt und im aktuellen Terminal hinterlegt.

2. Erstellen und kompilieren des Catkin Workspaces:

```
$ mkdir -p ~/catkin_ws_uebung1/src
$ cd ~/catkin_ws_uebung1/
$ catkin_make
```

   Hierdurch wurden im Dateipfad /home/benutzername/catkin_ws_uebung1 drei Ordner hinzugefügt: build, devel und src.

3. Catkin Workspace sourcen.
   Damit ROS die Packages im erstellten Workspace finden kann, muss auch der Catkin Workspace gesourced werden:

```
$ cd ~/catkin_ws_uebung1/
$ source devel/setup.bash
```

4. Überprüfung, ob der Workspace korrekt eingerichtet ist:

```
$ echo $ROS_PACKAGE_PATH
```

   Die Rückmeldung sollte wie folgt aussehen:
   `/home/benutzername/catkin_ws_uebung1/src:/opt/ros/noetic/share`

Damit ist der Workspace korrekt eingerichtet und es kann mit der Erstellung des Packages fortgeführt werden.

Hinweis: Bei der ROS-Installation, wie in Kapitel 4.1 beschrieben, ist schon ein Workspace eingerichtet worden. Daher ist im /home/Benutzer-Ordner schon ein Workspace mit dem

[55] Vgl. (subscription.packtpub, 2023)
[56] Vgl. (Wiki.ros[d], 2020)

*Namen „catkin_ws“ vorhanden. Dieser wurde in der bashrc-Konfigurationsdatei gesourced, was bedeutet, dass nach jedem öffnen eines Terminals automatisch die ROS-Umgebung und der Catkin Workspace „catkin_ws“ gesourced werden. Solange das **Praktikumsmodul 1** durchgeführt wird, wird jedoch im Catkin Workspace „catkin_ws_uebung1“ gearbeitet und dementsprechend muss der Punkt 3 nach **jedem Öffnen** eines neuen Terminals ausgeführt werden.*

4.2.2 ROS-Package erstellen

Als nächstes wird ein ROS-Package mit Hilfe des Catkin Tools erstellt. Innerhalb dieses Packages wird sich später der geschrieben Code für den Publisher- und Subscriber-Node befinden.

1. In den Catkin Workspace src Ordner navigieren

```
$ cd ~/catkin_ws_uebung1/src
```

2. Erstellung eines neuen Packages mit den benötigten Abhängigkeiten (dependencies). Dependencies fügen dem Package die notwendigen Funktionalitäten durch externe Libraries und Tools hinzu.[57] Für die Erstellung eines einfachen Publisher/Subscriber-Modells werden folgende Dependencies benötigt:
    - roscpp:
      C++ Implementierung für ROS[58]
    - rospy:
      Python Implementierung für ROS[59]
    - std_msgs:
      Message-Types für Datenübertragung[60]

   Der Terminalbefehl zu Erstellung des Packages mit den entsprechenden Dependencies sieht wie folgt aus:

```
$ catkin_create_pkg uebung1 std_msgs rospy roscpp
```

[57] Vgl. (Wiki.ros[e], 2019)
[58] Vgl. (Wiki.ros[f], 2015)
[59] Vgl. (Wiki.ros[g], 2017)
[60] Vgl. (Wiki.ros[h], 2017)

3. Kompilieren des Catkin Workspaces

   Der Catkin Worskpace muss nach dem Einfügen eines Packages erneut kompiliert werden. Hierfür wird wieder in den Workspace navigiert und der catkin_make Befehl ausgeführt.

```
$ cd ~/catkin_ws_uebung1
$ catkin_make
```

4. Catkin Workspace sourcen

```
$ . ~/catkin_ws_uebung1/devel/setup.bash
```

4.2.3 Publisher-Node erstellen

Damit sind nun alle nötigen Vorbereitungen getroffen und es kann mit der Erstellung der Nodes für das Publisher/Subscriber-Model gestartet werden. Zuerst wird der Publisher-Node erstellt. Hierfür muss in das zuvor erstellte uebung1 Package navigiert werden. (Falls ein neues Terminal geöffnet wurde, muss der Catkin Workspace „catkin_ws_uebung1“ erneut gesourced werden.)

```
$ roscd uebung1
```

Als nächstes wird ein scripts-Ordner erstellt, in dem die ausführbaren Skripte enthalten sind und in diesen navigiert.

```
$ mkdir scripts
$ cd scripts
```

Nun wird das vorgefertigte Skript heruntergeladen. In diesem Fall wird das Python Skript verwendet. Da aber die Dependencies roscpp und rospy im Package enthalten sind, kann auch ein C++ Skript verwendet werden. Nach dem Herunterladen muss das Python Skript ausführbar gemacht werden.

```
$ wget https://raw.github.com/ros/ros_tutorials/kinetic-
devel/rospy_tutorials/001_talker_listener/talker.py
$ chmod +x talker.py
```

Falls sich Python noch nicht auf dem Ubuntu Computer befindet, ist die Installation mit folgenden Terminalbefehlen durchzuführen:

```
$ sudo apt update
$ sudo apt install python3
```

Der folgende Code muss in der CMakeLists.txt Datei, welche sich im Ordner /home/benutzername/catkin_ws_uebung1/src/uebung1 befindet, hinzugefügt werden. In welche Zeile der Code eingesetzt wird, ist dabei unwichtig. Der Übersicht halber bietet es sich an diese Codezeilen an den Schluss der Textdatei zu setzen. Der Code sorgt dafür, dass das Python-Skript korrekt installiert und der passende Python Interpreter verwendet wird.

```
catkin_install_python(PROGRAMS scripts/talker.py
      DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

Der Inhalt des gerade heruntergeladenen Codes sieht wie folgt aus:

```
 1 #!/usr/bin/env python
 2 # license removed for brevity
 3 import rospy
 4 from std_msgs.msg import String
 5
 6 def talker():
 7     pub = rospy.Publisher('chatter', String, queue_size=10)
 8     rospy.init_node('talker', anonymous=True)
 9     rate = rospy.Rate(10) # 10hz
10     while not rospy.is_shutdown():
11         hello_str = "hello world %s" % rospy.get_time()
12         rospy.loginfo(hello_str)
13         pub.publish(hello_str)
14         rate.sleep()
15
16 if __name__ == '__main__':
17     try:
18         talker()
19     except rospy.ROSInterruptException:
20         pass
```

Der Code wir nun etwas genauer erläutert.

```
1 #!/usr/bin/env python
```

Dieser Teil steht immer am Anfang eines Python ROS Nodes. Dadurch wird sichergestellt, dass das Skript als Python-Skript ausgeführt wird.

```
3 import rospy
4 from std_msgs.msg import String
```

„Rospy“ muss importiert werden, da ein Node in Python erstellt wird. Zudem wird der String Message Type aus den „std.msgs.msg“ benötigt, um die gewählte Nachricht zu senden.

```
6 def talker():
  7     pub = rospy.Publisher('chatter', String, queue_size=10)
  8     rospy.init_node('talker', anonymous=True)
```

Zeile 6 erstellt die Funktion „talker“. Anschließend folgt der Code der Funktion. Die Codezeilen 7 und 8 definiert die Schnittstelle zwischen dem „talker“-Publisher-Node und ROS. Zeile 7 sagt aus, dass der Node über das Topic „chatter“ im Message-Type String sendet. Die „queue_size“ limitiert die Anzahl an wartenden Nachrichten, falls kein Subscriber diese Nachrichten schnell genug empfängt. Zeile 8 gibt dem Node einen Namen, was die Kommunikation zum ROS-Master ermöglicht. In diesem Fall hat der Node den Namen „talker“. „Anonymous=True“ versichert, dass der Node einen einzigartigen Namen hat, indem zufällige Nummern am Ende des Namens gehangen werden.

```
  9     rate = rospy.Rate(10) # 10hz
```

Diese Zeile setzt die Veröffentlichungsrate der Nachricht fest. In diesem Fall zehn Mal pro Sekunde.

```
 10     while not rospy.is_shutdown():
 11         hello_str = "hello world %s" % rospy.get_time()
 12         rospy.loginfo(hello_str)
 13         pub.publish(hello_str)
 14         rate.sleep()
```

Bei Zeile 10 bis 14 handelt es sich um eine While-Not-Schleife. Diese Schleife wird ausgeführt solange „rospy.is_shutdown()“ inaktiv ist, also der Vorgang nicht mit STRG+C o.Ä. abgebrochen wurde. Ist der Vorgang nicht abgebrochen worden, wird „hello_str“ definiert. „hello_str“ beinhaltet den Textinhalt der Nachricht und der aktuellen Zeit in Sekunden. Das dargestellte Zeitformat ist der Unix-Timestamp. Dieser stellt die Sekunden ab dem 1.1.1970, 0:00Uhr UTC. dar.[61] Mit „rospy.loginfo(hello_str)“ wird die Nachricht plus Zeit im Terminal angezeigt und die Daten ins Logfile des Nodes und in rosout gespeichert. Das Logfile und rosout dienen zur Dokumentation und Fehleranalyse. Der Befehl zum Senden der Nachricht über das Topic „chatter“ erfolgt mit „pub.publisher(hello_str)“. „Rate.sleep()“ pausiert die Schleife passend zur festgelegten Rate.

```
16 if __name__ == '__main__':
```

Diese Zeile überprüft, ob das ausgeführte Python-Programm das Hauptprogramm ist oder ob es von einem anderen Python Programm aufgerufen wurde.[62] Wenn das Python-

[61] Vgl. (unixtimestamp, 2023)
[62] Vgl. (RealPython, 2022)

Programm als Skript gestartet wird, wie in diesem Fall, ist es das Hauptprogramm und die nachfolgende if-Funktion wird ausgeführt.

```
17      try:
18          talker()
19      except rospy.ROSInterruptException:
20          pass
```

Nachdem die if-Bedingung erfüllt ist, wird die vorher definierte „talker()“-Funktion ausgeführt. Wird diese nicht mehr benötigt, kann das Skript mit STRG+C beendet werden. Ist die Funktion aber gerade durch den „rate.sleep()“-Befehl aus Zeile 14 pausiert, wird die Funktion nicht beendet, da kein Tastaturinput entgegen genommen werden kann. Zeile 19 und 20 lösen dieses Problem, da „rospy.ROSInterruptException“ durch „rate.sleep()“ ausgelöst werden kann, wenn das Skript durch STRG+C o.Ä. während des Pause-Befehls beendet wird. Ist dies der Fall, wird mit „pass“ die Funktion beendet. Damit ist der Publisher-Node programmiert und der Code nachvollziehbar. Nun kann der Subscriber-Node erstellt werden.

4.2.4 Subscriber-Node erstellen

Als erstes muss der Python-Code heruntergeladen, im scripts-Ordner gespeichert und ausführbar gemacht werden. Dafür wird wieder das Terminal mit STRG+ALT+T geöffnet und folgendes eingegeben:

```
$ roscd uebung1/scripts/
$ wget https://raw.github.com/ros/ros_tutorials/kinetic-
devel/rospy_tutorials/001_talker_listener/listener.py
$ chmod +x listener.py
```

Auch hier muss ein Zusatz in die CMakeLists.txt Datei, welche sich im Ordner /home/benutzername/catkin_ws_uebung1/src/uebung1 befindet:

```
catkin_install_python(PROGRAMS scripts/talker.py scripts/listener.py
      DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

Der Python-Code für den Subscriber-Node sieht wie folgt aus:

```
 1 #!/usr/bin/env python
 2 import rospy
 3 from std_msgs.msg import String
 4
 5 def callback(data):
 6     rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)
 7
 8 def listener():
 9
10     # In ROS, nodes are uniquely named. If two nodes with the same
11     # name are launched, the previous one is kicked off. The
12     # anonymous=True flag means that rospy will choose a unique
13     # name for our 'listener' node so that multiple listeners can
14     # run simultaneously.
15     rospy.init_node('listener', anonymous=True)
16
17     rospy.Subscriber("chatter", String, callback)
18
19     # spin() simply keeps python from exiting until this node is stop
ped
20     rospy.spin()
21
22 if __name__ == '__main__':
23     listener()
```

Der Code für den Subscriber ähnelt dem des Publishers in vielen Bereichen. Einige Unterschiede gibt es dennoch.

```
15     rospy.init_node('listener', anonymous=True)
16
17     rospy.Subscriber("chatter", String, callback)
```

In Zeile 15 wird dem Node den Namen „listener“ gegeben und auch hier ist „anonymous=True“, damit ROS eine zufällige Nummer zur eindeutigen Identifikation anhängt. Zeile 17 verbindet den Node mit der „chatter“ Topic, welches im String-Format ist. Wenn neue Nachrichten empfangen werden, wird der Callback mit der Nachricht als erstem Argument aufgerufen.

```
 5 def callback(data):
 6     rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)
```

Die Funktion „callback“ stellt die empfangende Nachricht dar und zeigt zusätzlich, dass dieser Node die Nachricht empfangen hat.

```
20      rospy.spin()
```

„Rospy.spin()“ verhindert das Beenden des Python-Skripts, solange der Node nicht beendet wurde. Damit ist nun auch der Subscriber-Node erstellt worden und der verwendete Code ist nachvollziehbar. Im Anschluss müssen die Nodes noch kompiliert werden. Dazu muss in den Workspace navigiert und der „catkin_make“-Befehl ausgeführt werden.

```
$ cd ~/catkin_ws_uebung1
$ catkin_make
```

4.2.5 Ausführen der Publisher/Subscriber Nodes

Um die gerade erstellten Nodes zu verwenden, muss zunächst der Roscore gestartet werden. Dies ist bei jedem ROS-Programm notwendig, da der Roscore den ROS-Master aktiviert und nur so die Nodes untereinander kommunizieren können.[63] Roscore wird in einem separaten Terminal gestartet. Dieses Terminal kann in der Zeit nicht für andere Zwecke verwendet werden.

```
$ roscore
```

Im Anschluss muss ein weiteres Terminal geöffnet werden, in dem der Publisher-Node ausgeführt wird. Der Workspace in dem die Skripte für die Nodes liegen, ist catkin_ws_uebung1, also muss auch dieser wieder gesourced werden.

```
$ cd ~/catkin_ws_uebung1
$ source ./devel/setup.bash
```

Nun ist der Publisher-Node ausführbar.

```
$ rosrun uebung1 talker.py
```

In dem Terminal sollte jetzt die veröffentlichte Nachricht und der Zeitpunkt dargestellt werden. Dies sieht wie folgt aus: [INFO] [1676554150.888998]: hello world 1676554150. 8886447. Diese Nachricht, mit angepassten Zeitdaten, wird ca. zehn Mal pro Sekunde gesendet. „hello world“ ist der Inhalt der Nachricht und 1676554150.8886447 das Datum, wann diese Nachricht gesendet wurde. 1676554150.8886447 entspricht dabei dem 16.02.2023 13:29:10 GMT+0000.

In einem neuen Terminal wird der Subscriber-Node gestartet. Auch hier muss erst wieder der entsprechende Workspace gesourced werden.

```
$ cd ~/catkin_ws_uebung1
$ source ./devel/setup.bash
```

[63] Vgl. (Wiki.ros[i], 2019)

Anschließend kann der Subscriber-Node ausgeführt werden.

```
$ rosrun uebung1 listener.py
```

Der Subscriber-Node sollte nun die gesendeten Nachrichten vom Publisher-Node empfangen. Die Ausgabe auf dem Terminal sollte demnach wie folgt aussehen: [INFO] [1676555051.440851]: /listener_5993_1676555048083I heard hello world 1676555051.4349885.

/listener_5993_1676555048083I ist der Name des Subscriber-Nodes und auch hier wird die Nachricht „hello world“ vom Publisher-Node angezeigt. Damit ist das Publisher/Subscriber-Modell erstellt und funktionsfähig. Mit dieser Übung wird auch gezeigt, ob ROS und seine Komponenten korrekt installiert wurden.

4.2.6 Visualisierung mit rqt_graph

In ROS gibt es zahlreiche Tools zur Visualisierung von Daten, Zusammenhängen, usw. Eines davon ist Rqt_graph. Mit Rqt_graph können alle aktiven Nodes und ihre Kommunikation untereinander dargestellt werden. Dies ist eine große Hilfe, um ROS-Programme besser nachvollziehen zu können und es unterstützt ebenfalls bei der Fehlersuche. Die Publisher/Subscriber Nodes sind weiterhin im Hintergrund aktiv. Um rqt_graph zu starten, wird folgendes in ein neues Terminal eingegeben:

```
$ rqt_graph
```

Nach dem Öffnen von Rqt_graph erscheint ein Fenster entsprechend Abbildung 37. Die Einstellungen können aus der Abbildung übernommen werden. Aus dem Graphen wird ersichtlich, dass der Node /talker_63… dem Node /listener_63… über das Topic /chatter Daten sendet. Nodes werden im Rqt_graph in Ellipsen dargestellt und Topics in Rechtecken. Zur Aktualisierung des Graphen kann der blaue Pfeil links neben „Nodes/Topics(all)“ angeklickt werden. Wenn der Mauszeiger über einem Topic liegt, werden alle „angeschlossenen“ Nodes farbig hervorgehoben.

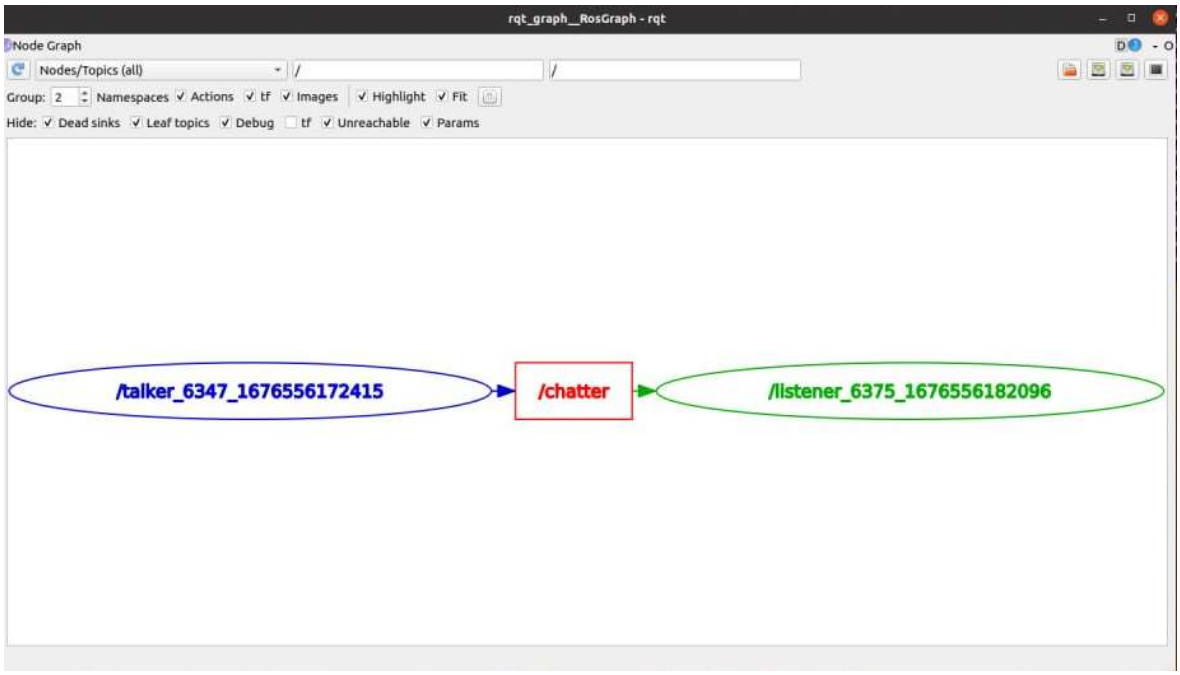


Abbildung 37 rqt_graph "chatter topic"

Mit diesem Tool sind auch ROS-Programme mit vielen Nodes und Topics nachvollziehbar analysierbar.

4.2.7 Fazit

Damit ist das erste Praktikumsmodul abgeschlossen. Ziel dieses Moduls war es den Umgang mit gängigen ROS- und Linux-Befehlen zu lernen, zu verstehen wie ein einfacher ROS-Node aufgebaut ist, das Publisher/Subscriber-Modell nachvollziehen zu können, zu lernen wie laufende ROS-Nodes in rqt-graph dargestellt werden können und zu zeigen, wie ein ROS-Workspace erstellt und darin gearbeitet wird. Zudem kann dieses Modul genutzt werden, um zu überprüfen, ob ROS korrekt installiert wurde.

Der Zeitaufwand für dieses Modul beträgt ca. 45-60min.

4.3 Praktikumsmodul 2: TurtleBot3 fernsteuern

Im Zuge dieses Praktikumsmoduls wird praktisch mit dem Roboter gearbeitet. Dieser soll von einem Computer ferngesteuert werden. Als Eingabe wird die Tastatur des Computers verwendet. Es ist nicht wie in Praktikumsmodul 1 nötig, eine neue Arbeitsumgebung zu erstellen. Dies wurde bereits bei der ROS-Installation erledigt. Ebenso wurde die ROS-Version sowie der passende Workspace gesourced und in der bashrc-Datei gespeichert. Damit wird die ROS-Version und der Workspace automatisch beim Öffnen eines neuen Terminals gesourced und es ist nicht mehr nach jedem Öffnen eines Terminals notwendig. *(Die bashrc-Datei kann mit **$ nano ~/.bashrc** geöffnet und bearbeitet werden, falls dies nötig sein sollte.)* Bei der ROS-Installation wurden zudem schon viele Packages mitinstalliert, darunter auch das „turtlebot3_teleop“-Package. Mit Hilfe dieses Packages wird die

geforderte Funktionalität geliefert, um den Roboter von einem Computer über die Tastatur fernsteuern zu lassen. Die zugrunde liegenden Befehle sind aus dem TurtleBot3 e-manual entnommen.[64][65]

4.3.1 TurtleBot3 hochfahren

Zunächst muss der Roscore auf dem Computer gestartet werden.

```
$ roscore
```

Im Anschluss wird der TurtleBot3 initialisiert. Dafür muss dieser mit Spannung versorgt werden. Über den Kippschalter an der Seite des Roboters kann dieser angeschaltet werden. Nach einer kurzen Zeit ist der Roboter einsatzbereit. Wenn sich die Netzwerkeinstellungen nicht geändert haben, kann jetzt eine Verbindung zwischen Computer und TurtleBot3 aufgebaut werden. Dafür wird eine SSH-Verbindung genutzt. SSH bedeutet Secure Shell und stellt eine sichere Netzwerkverbindung von einem Ubuntu Rechner zu einem anderen her, um diesen aus der Ferne bedienen zu können. Dieses Netzwerktool ist in der Regel standardmäßig in Ubuntu enthalten. Falls das Tool noch nicht installiert ist, kann es mit folgenden Befehlen hinzugefügt werden:[66]

```
$ apt-get install openssh-server openssh-client
```

Um die Verbindung herstellen zu können, muss der ROS-Master Computer und der Raspberry über eine aktive Netzwerkverbindung im selben Netzwerk verfügen. Auf dem ROS-Master Computer wird nun folgender Terminalbefehl eingegeben:

```
$ ssh ubuntu@{IP_ADDRESSE_VON_RASPBERRY_PI}
```

Ist die Verbindung aufgebaut, wird nach den Logindaten des Raspberrys gefragt. Diese sind:

- Login ID: **ubuntu**
- Passwort: **turtlebot.**

Die Verbindung ist nun aktiv und es können Terminalbefehle vom ROS-Master Computer auf dem Raspberry Pi ausgeführt werden. Nur Befehle die im Terminal, welches die Verbindung aufgebaut hat, eingegeben werden, werden auch auf dem Raspberry ausgeführt. Wird ein neues Terminal auf dem ROS-Master Computer geöffnet, werden die dort eingegebenen Befehle nur auf dem Computer ausgeführt. Die SSH-Verbindung wird abgebrochen, sobald das entsprechende Terminal geschlossen wird.

[64] Vgl. (Robotis e-Manual[b], 2023)
[65] Vgl. (Robotis e-Manual[c], 2023)
[66] Vgl. (heise, 2023)

Bevor der TurtleBot3 ROS-Operationen ausführen kann, muss dieser einige Basis-Packages laden. Das Hochfahren oder der sogenannte „bringup“ wird mit folgenden Befehlen innerhalb des SSH-Terminals gestartet:

```
$ export TURTLEBOT3_MODEL=burger
$ roslaunch turtlebot3_bringup turtlebot3_robot.launch
```

Der „export“ Terminalbefehl gibt an, um welche TurtleBot3-Variante es sich handelt. Dies ist wichtig, da bei der ROS-Installation Packages für alle TurtleBot3-Varianten enthalten sind und so die passenden verwendet werden. Wird ein TurtleBot3 Package in einem neuen Terminal geladen, muss zuvor dieser Befehl ausgeführt werden. Falls es gewünscht ist diesen Schritt zu automatisieren, kann die Modellvariante in der bashrc-Datei gespeichert werden. Dazu ist folgendes sowohl auf dem TurtleBot3 via SSH-Terminal als auch auf dem Computer auszuführen:

```
$ echo 'export TURTLEBOT3_MODEL=burger' >> ~/.bashrc
$ source ~/.bashrc
```

Nun wird das Modell bei jedem Terminalstart automatisch festgelegt. Der Befehl zur Modelauswahl wird trotzdem in den folgenden Terminalbefehlen aufgeführt.

Der „roslaunch“ Befehl aktiviert alle notwendigen Nodes und Topics auf dem TurtleBot3, damit dieser korrekt verwendet werden kann. Die Nodes und Topics werden später noch genauer betrachtet. Damit ist der TurtleBot3 hochgefahren und einsatzbereit.

4.3.2 TurtleBot3 steuern

Mit der Bereitschaft des Systems kann das entsprechende Package zur Steuerung des Roboters geladen werden. Das Package hat den Namen „turtlebot3_teleop“. Die folgenden Befehle sind in einem separaten Terminal auf dem Computer auszuführen:

```
$ export TURTLEBOT3_MODEL=burger
$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

Nach Ausführung dieser Befehle kann der Roboter mit der Tastatur ferngesteuert werden. Im Terminal erscheint eine Bedienungsanleitung zur Steuerung des Roboters.

Abbildung 38 teleop-Package

Wie in Abbildung 38 zu sehen ist, wird die Geschwindigkeit mit „w“ erhöht und mit „x“ verringert. Mit „a“ und „d“ wird jeweils die Winkelgeschwindigkeit angepasst, damit der Roboter sich nach links oder rechts dreht. Um die Bewegung des Roboters zu stoppen und alle Geschwindigkeiten auf 0 zurückzusetzen, kann die Taste „s“ betätigt werden. Die aktuellen Geschwindigkeitswerte werden unter „currently:“ angezeigt. Damit die Tastatureingaben vom Package entgegengenommen werden können, muss das Terminal aktiv ausgewählt sein (einfacher Klick auf das Terminal). Um die Anwendung zu beenden, wird die Tastenkombination STRG+C gedrückt. Nun ist es die Aufgabe sich mit der Bewegung des Roboters vertraut zu machen.

4.3.3 Visualisierung mit rqt_graph

Für ROS-Anwendungen werden oft vorgefertigte Packages verwendet, um Funktionalitäten wie die Steuerung des Roboters umzusetzen. ROS bietet den Vorteil, dass diverse Anwendungen umgesetzt werden können, ohne ein tiefergehendes Verständnis von dessen Programmierung und Implementierung zu haben. Dennoch ist es hilfreich einen Einblick in die grundlegende Funktion zu erlangen und zu sehen welche Nodes wie miteinander kommunizieren. Im vorigen Praktikumsmodul wurde der rqt_graph vorgestellt und damit ein hilfreiches Tool, um genau diese Beziehungen zu visualisieren. Der rqt_graph wird mit dem Befehl:

```
$ rqt_graph
```

in einem separaten Terminal geöffnet. Wichtig ist hierbei, dass das „teleop“-Package aktiv ist, da dessen Nodes und Topics sonst nicht dargestellt werden können.

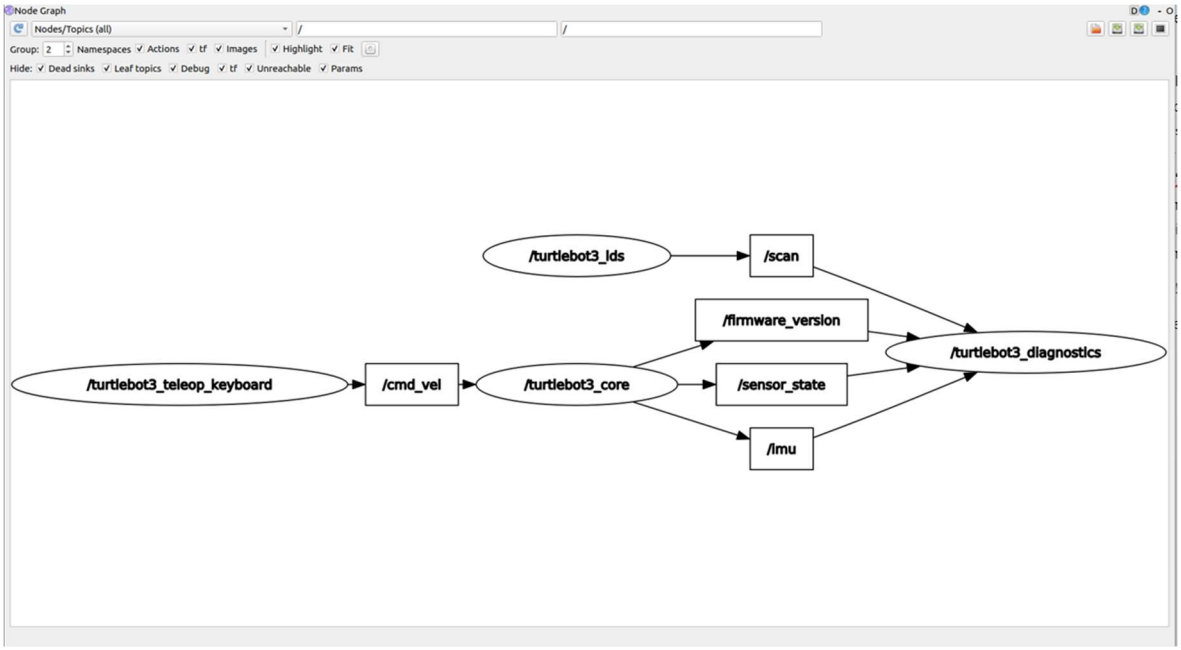


Abbildung 39 rqt_graph teleop & bringup

In Abbildung 39 sind die aktiven Nodes und Topics abgebildet. Der Node /turtlebot3_core repräsentiert den TurtleBot3 Roboter. Dieser Node verarbeitet alle Daten, die der Roboter benötigt, bzw. die externe ROS-Anwendungen vom Roboter benötigen, wie z.B. Position der Räder, Aktoren, IMU-Daten, usw. Dieser Node ist sowohl Publisher als auch Subscriber. Der Node kann also Daten über ein Topic empfangen und Daten über ein Topic senden. Die dargestellten Topics sind je nach Anwendung unterschiedlich, da nicht immer alle Daten benötigt werden. Die Sensordaten vom Lidar-Sensor werden von dem /turtlebot3_lds Node separat verarbeitet und über das Topic /scan gesendet. Der /turtlebot3_diagnostics Node empfängt alle roboterspezifischen Daten, um die Selbstdiagnose des Roboters durchzuführen. Die gerade genannten Nodes werden beim „bringup“ des TurtleBot3 gestartet und sind immer aktiv, wenn dieser gestartet wird. Der /turtlebot3_teleop_keyboard Node ist mit der Aktivierung des „teleop“-Packages hinzugekommen. Dieser Node empfängt die geforderten translatorischen und rotatorischen Geschwindigkeitswerte des ROS-Master-Computers, welche über die Tastatur eingegeben wurden und sendet diese über das /cmd_vel Topic. Der /turtlebot3_core Node empfängt die Daten aus diesem Topic und setzt die geforderten Geschwindigkeitswerte um, indem die Antriebsmotoren entsprechend angesteuert werden.[67] Damit ist ersichtlich geworden, welche Nodes und Topics benötigt werden, damit der Roboter über den Computer mit der Tastatur gesteuert werden kann.

[67] Vgl. (Pyo, Cho, Jung, & Lim, 2017, S. 290-294)

4.3.4 Fazit

Damit ist das zweite Praktikumsmodul beendet. In diesem Modul war es das Ziel, die Herangehensweise für die Verwendung eines vorgefertigten ROS-Packages zu zeigen und erste Berührungspunkte mit einem realen mobilen Roboter zu haben. Des Weiteren konnte durch den rqt_graph nachvollzogen werden, welche ROS Nodes für die Steuerung des Roboters benötigt werden und wie diese miteinander kommunizieren. Zusätzlich wurde vermittelt, welche Schritte notwendig sind, um den Computer und den TurtleBot3 für die Verwendung von ROS vorzubereiten.

Der Zeitaufwand bei diesem Modul beträgt ca. 30 min.

4.4 Praktikumsmodul 3: Kartierung der Umgebung (SLAM)

In diesem Praktikumsmodul soll der TurtleBot3 eine Karte seiner Umgebung erstellen. Die Umgebung ist dabei eine kleine Styropor-Arena, die eine beliebige Form haben kann. Die Karte wird mittels Sensordaten vom Roboter und dem SLAM-Verfahren erstellt. Wie schon in Kapitel 2.4 erwähnt, ist es mit SLAM möglich eine Karte der neuen Umgebung zu erstellen, während sich der Roboter in dieser befindet. Es wird also eine Karte erstellt und zeitgleich kann der Roboter sich innerhalb dieser Karte lokalisieren. Die verwendeten Konsolenbefehle sind dem Robotis e-Manual entnommen.[68]

4.4.1 Arena erstellen

Die Arena dient dazu eine klar definierte Karte erzeugen zu können. Sie kann aus beliebigem Material bestehen, in diesem Fall wird die Arena aus Styroporteilen zusammengesetzt. Die Form der Arena ist dabei variabel. Es sollten nur keine Engpässe ≤ 30cm erstellt werden, da der Roboter diese Stelle sonst nicht durchfährt. Grund dafür ist der verwendete Algorithmus „Costmap“, welcher dem Roboter sagt, ob eine Fläche passierbar ist oder nicht. Zudem ist es wichtig, dass die Styroporteile mindestens so hoch sind wie der TurtleBot3, da der Lidar-Sensor die Styroporteile sonst nicht wahrnehmen kann.

Abbildung 40 zeigt eine Beispielhafte Arena, welche sich gut zur Kartierung eignet.

[68] Vgl. (Robotis e-Manual[d], 2023)

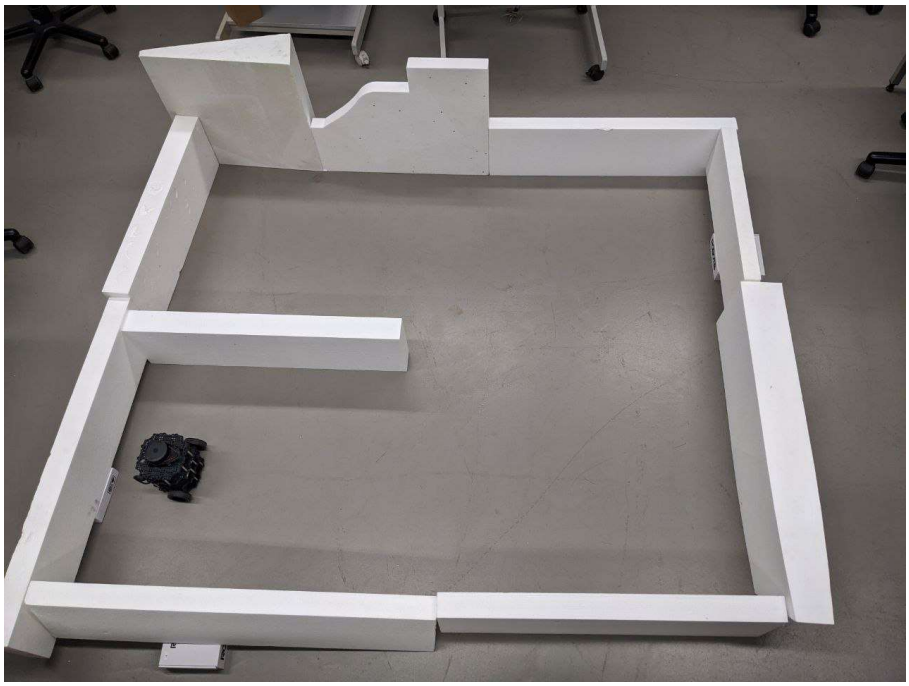


Abbildung 40 Beispielhafte Arena

4.4.2 Kartierung der Arena

Bevor mit der Kartierung begonnen wird, muss der Roscore und der „bringup“ gestartet werden, falls dies noch nicht erfolgt ist. Siehe dazu Kapitel 4.3.1. Nun kann der Roboter innerhalb der Arena platziert werden. Der Startpunkt ist dabei variabel. Um die Kartierung zu starten wird der Befehl:

```
$ export TURTLEBOT3_MODEL=burger
$ roslaunch turtlebot3_slam turtlebot3_slam.launch
```

in ein separates Terminal auf dem Computer eingegeben. Damit wird das SLAM-Package mit allen dazugehörigen Programmen gestartet. Darunter Rviz, ein Visualisierungsprogramm, welches u.a. Sensordaten des Roboters darstellen kann. Rviz wird in diesem Fall genutzt um die Pose des Roboters, die Laserscans des Lidar-Sensors und die mittels SLAM erzeugte Karte darzustellen. Die Sensordaten des Lidars werden in kleinen grünen Punkten dargestellt und zeigen die Wahrnehmung des Roboters. Nachdem das SLAM-Package gestartet wurde, muss der Roboter die Arena erkunden, um eine vollständige Karte erzeugen zu können. Dazu wird das „teleop“-Package mit

```
$ export TURTLEBOT3_MODEL=burger
$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

in einem separaten Terminal geladen. Nun kann der Roboter mit der Tastatur des Computers innerhalb der Arena navigiert werden. Abbildung 41 zeigt den Start der Kartierung. SLAM hat schon ein Teil der Arena kartiert, aber um die Karte zu vervollständigen, muss

die Arena komplett abgefahren werden. Dabei sollte sich der TurtleBot3 nicht zu schnell bewegen oder drehen. Eine langsame Fortbewegung sorgt für die besten Ergebnisse.

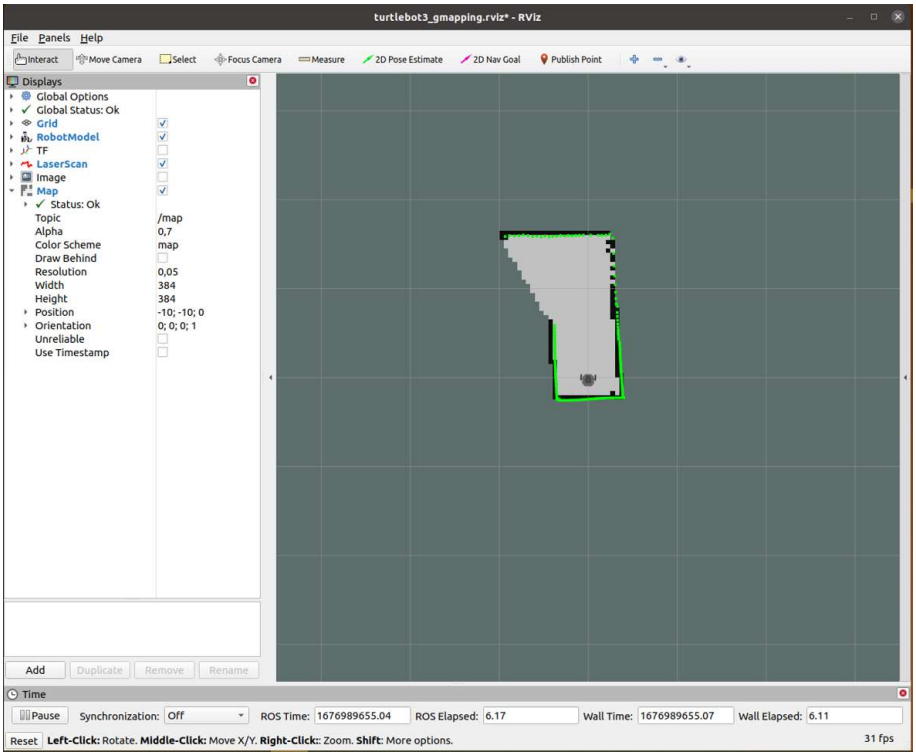


Abbildung 41 Kartierung der Arena

Nachdem die Arena komplett abgefahren wurde (siehe Abbildung 42) muss die Karte gespeichert werden. Dies erfolgt in einem separaten Terminal mit folgendem Befehl:

```
$ rosrun map_server map_saver -f ~/map
```

Die Karte wird in den Formaten „.pgm“ und „.yaml“ im Ordner /home/Benutzername/ gespeichert. Nun ist die Karte erstellt und für die weitere Verwendung bereit.

Hinweis: Der TurtleBot3 setzt seine interne Uhr nach jedem Start auf einen fixen Wert zurück, da er nicht über die notwendige Hardware verfügt, um die aktuelle Zeit nach dem Ausschalten beizubehalten. Bei einer aktiven Internetverbindung kann der TurtleBot3 die aktuelle Uhrzeit jedoch online abfragen. Die Uhrzeit ist wichtig, da die Sensordaten des Roboters mit einem Zeitstempel der Systemzeit an den Computer gesendet werden. Wenn diese eine zu große Zeitdifferenz zwischen Senden und Empfangen aufweisen, tritt unerwünschtem Verhalten auf und die Kartierung kann nicht vorgenommen werden. Falls kein Internetzugang bestehen sollte, kann die Zeit auch manuell eingegeben werden. Dazu kann folgende Anleitung verwendet werden.[69] Wenn die Zeit manuell eingegeben werden sollte, muss dies nach jedem Neustart des Roboters geschehen.

[69] https://wiki.ubuntuusers.de/Systemzeit/

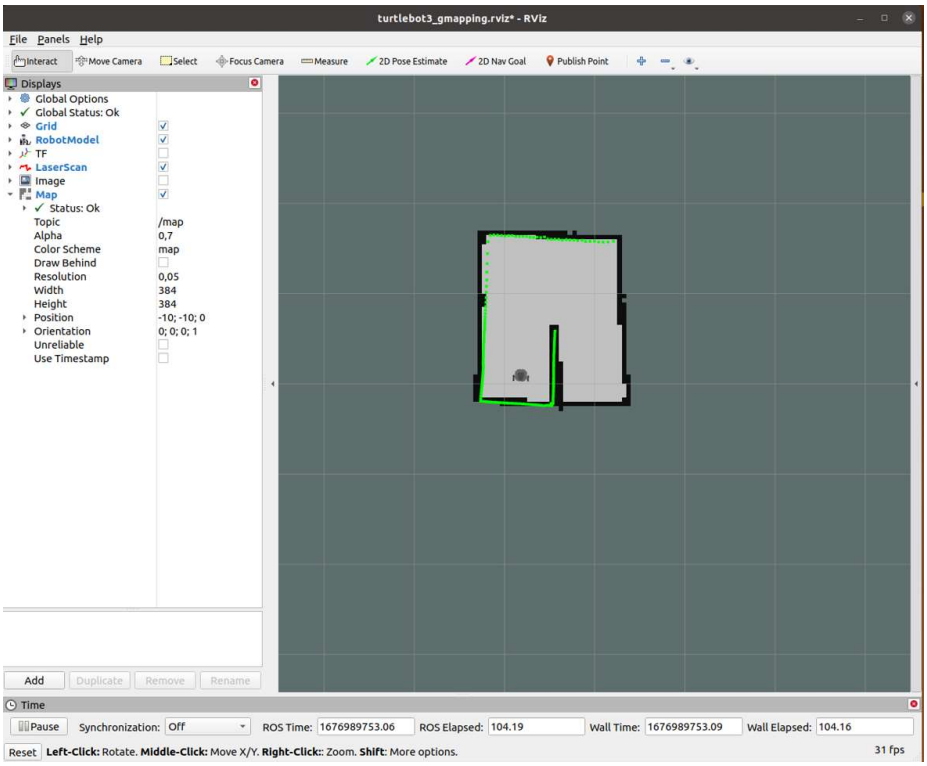


Abbildung 42 Arena komplett kartiert

4.4.3 Fazit

Damit ist das dritte Praktikumsmodul abgeschlossen. Die erstellte Karte kann nun für das nächste Modul verwendet werden. Ziel dieses Praktikumsmodul war es zu zeigen, wie eine Karte einer unbekannten Umgebung mit SLAM und ROS erstellt und gespeichert wird. Zudem gab es erste Berührungspunkte mit dem RViz Visualisierungstool, welches in vielen ROS Anwendungen Verwendung findet.

Der Zeitaufwand bei diesem Modul beträgt ca. 45-60 min.

4.5 Praktikumsmodul 4: Navigation des Roboters

In diesem Modul soll der Roboter autonom innerhalb der mit SLAM erstellten Karte zu einem vorgegebenen Zielpunkt fahren. Dabei soll die Roboternavigation in mehreren Szenarien getestet werden. Das erste Szenario ist die einfache Navigation von einem Start- zu einem Zielpunkt. Beim zweiten Szenario soll ein neuer Gegenstand in die Arena gestellt werden und der Zielpunkt ist so zu wählen, dass der Roboter um dieses neue Hindernis navigieren muss. Beim dritten Szenario soll der Roboter auf plötzlich auftauchende Hindernisse reagieren und zur Änderung seines Pfades gezwungen werden. Das erste Szenario prüft den globalen Pfadplaner und die zwei nachfolgenden Szenarios den lokalen Pfadplaner (Siehe Kapitel 2.4.3). Zudem sollen Navigationsparameter eingestellt werden, die den Roboter fehlerfrei innerhalb der Karte fahren lassen. Als erstes gilt es jedoch die initiale

Lokalisierung mit der Monte-Carlo-Lokalisierung durchzuführen. Die entsprechenden Konsolenbefehle wurden zum Teil aus dem Robotis e-Manual entnommen.[70]

4.5.1 Initiale Lokalisierung

In Kapitel 2.4.1 wurde die Lokalisierung innerhalb einer bekannten Karte thematisiert. Zur Durchführung wird die Monte-Carlo-Lokalisierung verwendet. Zunächst muss der Roscore gestartet und die SSH-Verbindung mit dem TurtleBot3 aufgebaut werden, falls dies noch nicht erfolgt ist. Der Roscore und die SSH-Verbindung werden dabei in separaten Terminals ausgeführt.

```
$ roscore
```

```
$ ssh ubuntu@{IP_ADDRESS_OF_RASPBERRY_PI}
$ export TURTLEBOT3_MODEL=burger
$ roslaunch turtlebot3_bringup turtlebot3_robot.launch
```

Im nächsten Schritt wird das TurtleBot3 Navigations-Package geladen. Dazu wird ein neues Terminal geöffnet.

```
$ export TURTLEBOT3_MODEL=burger
$ roslaunch turtlebot3_navigation turtlebot3_navigation.launch
map_file:=$HOME/map.yaml
```

Mit diesem Befehl wird die Karte „map“, welche im SLAM-Modul erstellt und gespeichert wurde und die dazugehörigen Navigationsprogramme, geladen. Wurde die Karte an einem anderen Ort oder unter einem anderen Namen gespeichert als im SLAM Kapitel angegeben, muss der Dateipfad nach „map_file:=“ eingegeben werden. Nun öffnet sich das RViz-Tool mit der gespeicherten Karte entsprechend Abbildung 43. Hier ist eine Reihe von Informationen abgebildet, die nach und nach erklärt werden. Als erstes fällt jedoch auf, dass die Lidarsensordaten (grüne Punktlinie) nicht mit der erstellten Karte übereinstimmt. Dies ist damit zu begründen, dass der Roboter seine Lage innerhalb dieser Karte noch nicht genau kennt. Um dies zu beheben, muss dem Roboter bei dieser Navigationsmethode eine initiale Pose vorgegeben werden. Die Pose muss dabei nicht exakt sein, sollte aber ungefähr die reale Pose abbilden. Um den Roboter die Initialpose vorzugeben, muss die Schaltfläche „2D Pose Estimate“ gedrückt werden. Im Anschluss wird mit dem Mauszeiger auf die ungefähre Position innerhalb der Karte geklickt. Die Maustaste bleibt dabei gedrückt. Es erscheint ein großer grüner Pfeil, der die Orientierung des Roboters darstellt (siehe Abbildung 44). Wenn diese ungefähr abgeschätzt wurde, kann die Maustaste losgelassen werden und

[70] Vgl. (Robotis e-Manual[e], 2023)

die initiale Pose ist vorgegeben. Falls das Ergebnis nicht hinreichend genau ist, kann der Vorgang beliebig oft wiederholt werden.

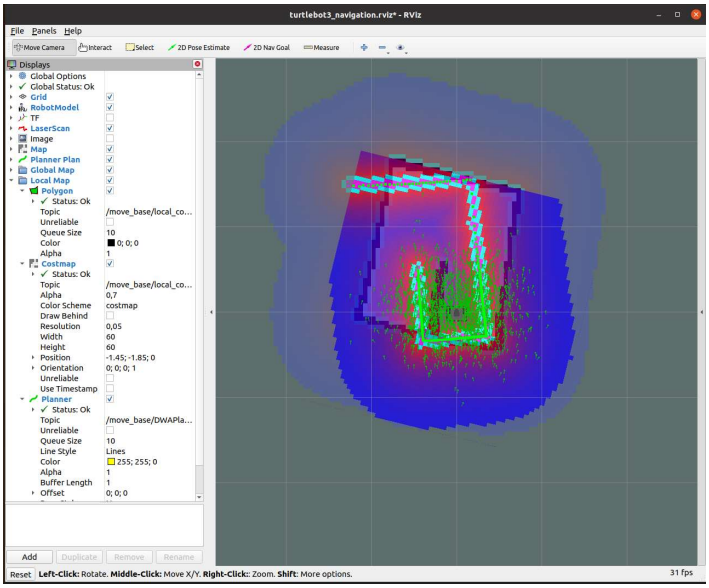


Abbildung 43 1. Schritt Navigation: Öffnen des Navigations-Packages mit der erstellten Karte

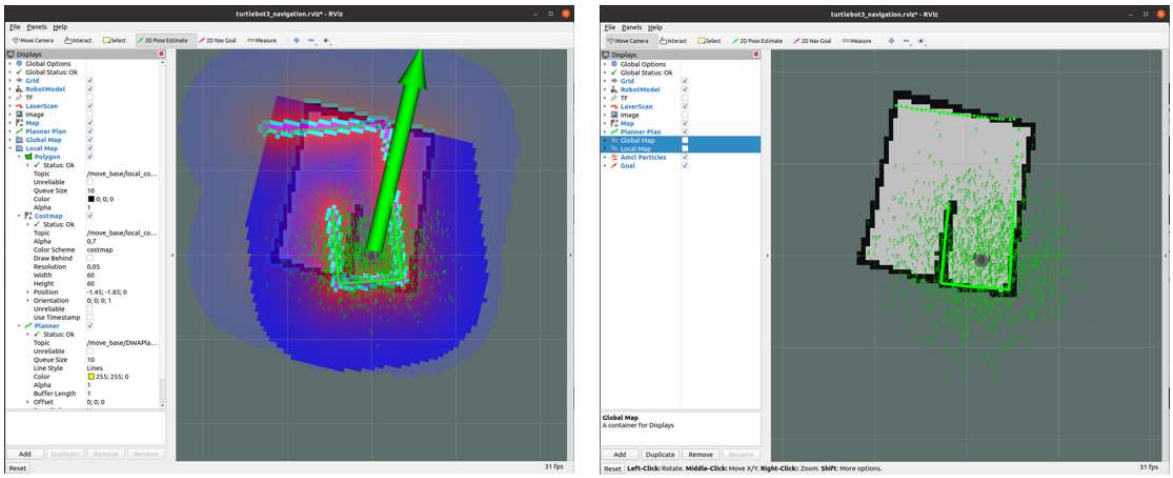


Abbildung 44 2.Schritt Navigation: Initiale Pose vorgeben

In der Abbildung 44 wurden im rechten Bild die Haken bei Global Map und bei Local Map entfernt. Dadurch wird die Costmap nicht dargestellt, was der Übersicht dienlich ist. In den weiteren Schritten wird die Darstellung der Costmaps wieder benötigt. Wie zu sehen ist, entsprechen die Lidardaten ungefähr den Kartendaten. Die initiale Pose ist damit hinreichend genau bestimmt. Die Lokalisierung ist damit aber noch nicht abgeschlossen. Im Bild sind zahlreiche grüne Pfeile abgebildet. Da hier die Monte-Carlo-Lokalisierung verwendet wird, stellt jeder dieser Pfeile eine mögliche Roboterpose dar. Der Roboter ist sich also seiner Lage innerhalb der Karte noch nicht sicher. Um die möglichen Posen einzugrenzen, benötigt der Roboter Lidardaten, über die er in diesem Augenblick schon verfügt und

Odometriedaten aus IMU und Moter-Encoder. Um die Odometriedaten zu erzeugen, muss der Roboter bewegt werden. Dazu wird das „teleop“-Package gestartet.

```
$ export TURTLEBOT3_MODEL=burger
$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

Nachdem der Roboter etwas bewegt wurde, wobei eine Kreisbewegung sich als am effektivsten bewiesen hat, sammeln sich die grünen Pfeile um das Robotermodell, wie in Abbildung 45 dargestellt. Die Lokalisierung ist damit abgeschlossen.

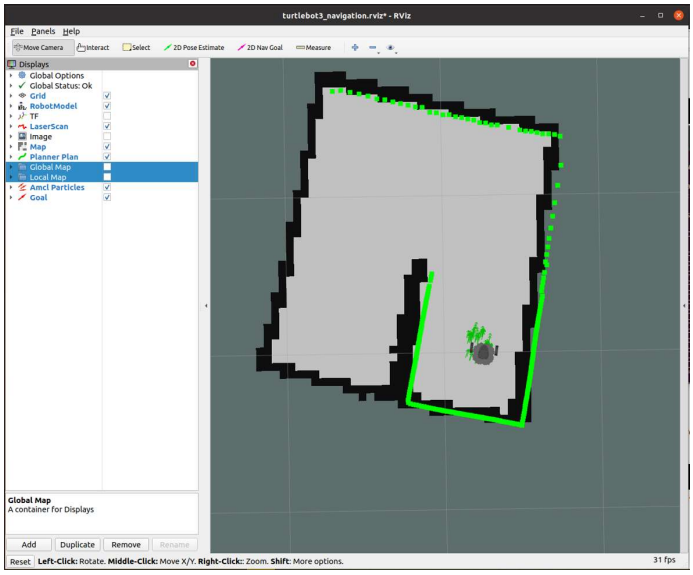


Abbildung 45 3.Schritt: MCL durchgeführt

Im Anschluss können die Haken bei der Global und Local Map wieder gesetzt werden. Hier fällt nun auf, dass die erstellte Costmap nicht mehr mit der Karte übereinstimmt. Um das zu beheben, können die Costmaps mit einem Terminalbefehl zurückgesetzt werden.

```
$ rosservice call /move_base/clear_costmaps "{}"
```

Dieser Befehl kann immer dann eingesetzt werden, wenn die Costmap bei der Navigation Fehler aufweist. Das Resultat des Zurücksetzens der Costmaps ist in Abbildung 46 dargestellt.

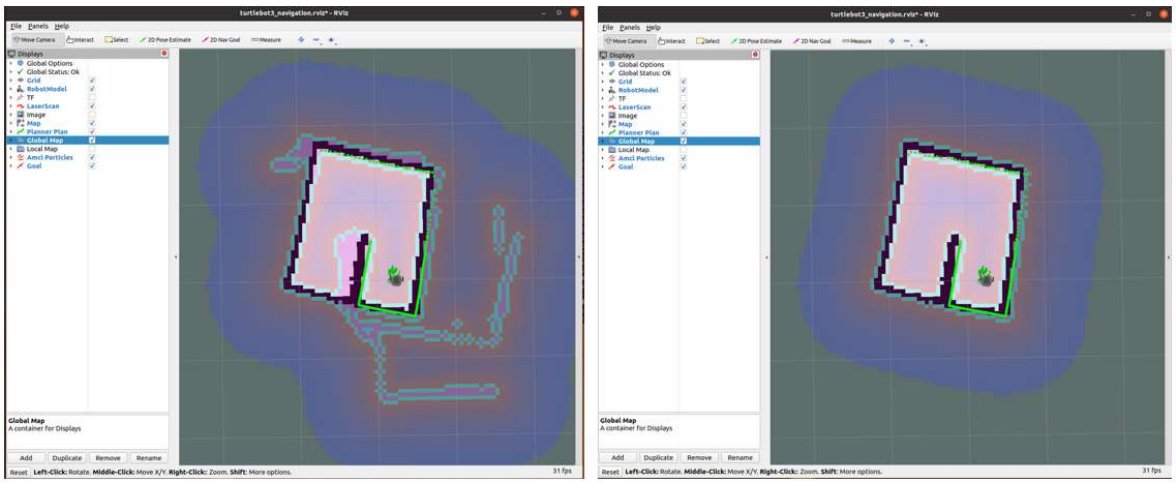


Abbildung 46 Zurücksetzen der Costmaps

4.5.2 Optimierung

Die Pfadplanung kann mit den voreingestellten Parametern der Costmaps und des lokalen Planers zu unerwünschtem Verhalten des Roboters bei der Navigation führen. Daher sollten die Parameter angepasst werden. Dazu kann das Tool rqt_reconfigure verwendet werden, welches die Parameteranpassung während der Ausführung des Navigationsprogrammes erlaubt. Um das Tool zu startet muss folgender Terminalbefehl ausgeführt werden:

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

Im Anschluss öffnet sich ein Fenster, indem alle gerade verwendeten Packages aufgeführt werden (siehe Abbildung 47). Hier können die gewünschten Packages ausgewählt und die Parameter live geändert werden. Die eingestellten Parameter werden nach Beenden des Tools auf die Standardparameter zurückgesetzt.

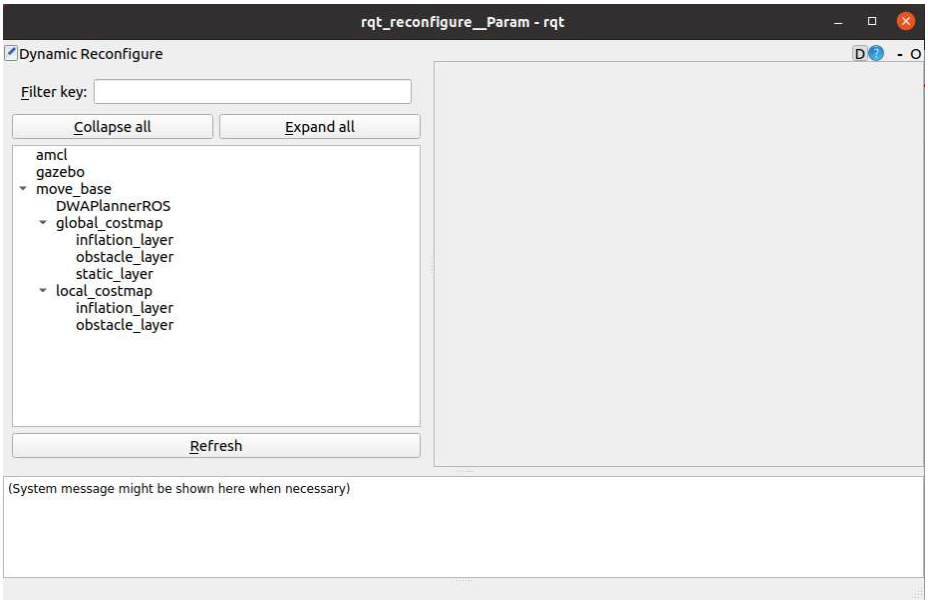


Abbildung 47 Dynamische Parameteranpassung mit rqt_reconfigure

Sind die passenden Parameter gefunden und sollen permanent beibehalten werden, können die entsprechenden Parameterdateien im Terminal bearbeitet und gespeichert werden. Zu finden sind diese unter /opt/ros/noetic/share/turtlebot3_navigation/param. Um diese zu bearbeiten, ist folgender Befehl in das Terminal einzugeben:

```
$ sudo nano /opt/ros/noetic/share/turtlebot3_navigation/param/{Datei-
name.yaml}
```

Nachdem das Benutzerpasswort eingetragen wurde, öffnet sich die Textdatei im Terminal. Mit STRG+S werden die Änderungen gespeichert und mit STRG+X wird die Datei wieder geschlossen.

Die für diesen Praktikumsversuch relevanten Parametereinstellungen können in den folgenden Dateien vorgenommen werden:

- costmap_common_params_burger.yaml
    - Parameter: „inflation_radius“, „cost_scaling_factor“
- local_costmap_params.yaml
    - Parameter: „resolution“
- dwa_local_planner_params_burger.yaml
    - Parameter: „sim_time“, „path_distance_bias“

Lokaler Planer

Mit einem Klick auf „DWAPlannerROS“ öffnen sich alle verfügbaren Parametereinstellungen für den lokalen Planer. Die relevantesten Einstellung für diesen Versuch sind „sim_time“ und „path_distance_bias“ (siehe Abbildung 48). Der „sim_time“-Parameter gibt die Zeit in Sekunden an, die der lokale Planer vorausplanen soll. Der Standardwert liegt bei 2 Sekunden. Im Versuchsaufbau hat sich eine Einstellung von 1.2 Sekunden bewährt. Wird die Zeit zu hoch angesetzt, kreist der Roboter um sein Zielpunkt umher, ohne es zu erreichen. Das könnte zwar mit einer Zielpunkt-Toleranzeinstellung behoben werden, würde aber zu einer ungenaueren Navigation führen. Der „path_distance_bias“ gibt an, in welchem Maße der lokale Planer dem globalen Planer folgt. Im Versuch hat sich eine Gewichtung von 40 bewährt. Ist dieser Wert zu niedrig, wird der lokale Planer eher vom globalen Plan abweichen, wenn das Ziel schneller erreicht werden kann. Dies kann aber zu unerwünschtem Verhalten führen, z.B. könnte sich der Roboter an einer Kante festfahren, da er diese nicht umfahren möchte.

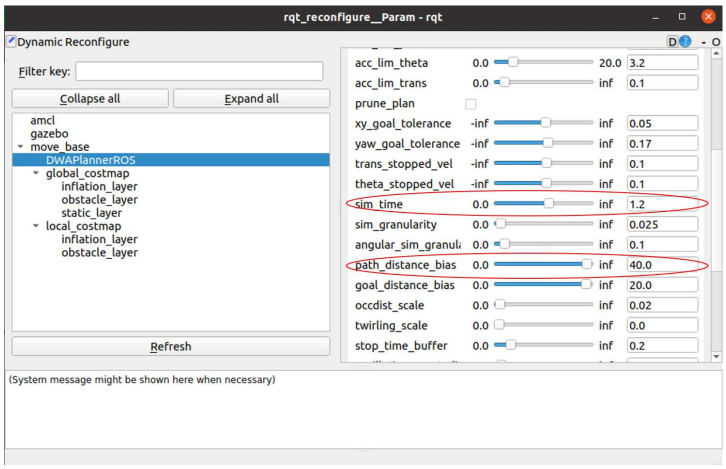


Abbildung 48 DWAPlannerROS Parameter

Globale und lokale Costmap

Wie in Kapitel 2.4.3 aufgeführt, bildet die Costmap die Grundlage der Pfadplanung, indem sie Bereiche innerhalb der Karte mit Kostenwerten belegt. Sind die Kostenwerte zu hoch wird die Pfadplanung einen alternativen, kostengünstigeren Weg wählen. Zudem gibt die Costmap vor, welche Bereiche befahren werden können oder welche Bereiche von Hindernissen versperrt sind. Die nicht befahrbaren Bereiche ergeben sich aus den Abmaßen des Roboters und werden in der Visualisierung türkis dargestellt. Dieser Bereich wird von dem Roboter niemals überschritten, da es sonst zu einer Kollision kommen würde. Zudem werden die Hindernisse künstlich vergrößert (Inflation). Dies dient dazu, Kostenwerte über den Kollisionsbereich hinaus auftragen zu können. Die Kostenwerte nehmen mit der Entfernung zum Objekt innerhalb des Inflationsradius ab, dargestellt durch einen rot-blauen Farbverlauf. Die Abmaße und Werte für den Inflationsradius und der Kostenverteilung in diesem Radius können in den Parametereinstellungen angepasst werden. Initial sind die Abmaße korrekt eingetragen und bedürfen keiner Anpassung. Die Parametereinstellungen für die Inflation ist unter dem Punkt „inflation layer“ zu finden (siehe Abbildung 49). Der „inflation_radius“ sagt aus, wie weit ein Hindernis künstlich vergrößert werden soll. Die Angabe ist dabei in Meter. Der Wert sollte nicht den Radius des Roboters unterschreiten. Die Kostenverteilung innerhalb des Radius kann mit dem „cost_scaling_factor“ angepasst werden. Dabei handelt es sich um eine Funktion mit reziproker Proportionalität. Wird dieser Wert vergrößert, verringern sich die Kosten innerhalb der Inflation. Wird der Wert verringert, vergrößern sich die Kosten. Dargestellt in den Farben blau, für geringere Kosten und rot, für hohe Kosten. Der Roboter strebt immer den Weg mit den geringsten Kosten an. Im Versuch hat sich der Wert 0.25 für den „inflation_radius“ und der Wertebereich zwischen 5 - 10 für den „cost_scaling_factor“ bewährt.

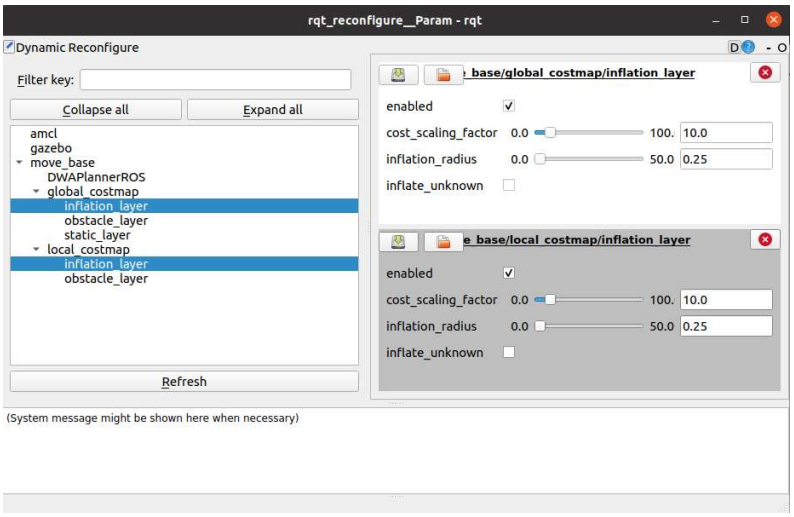


Abbildung 49 Costmap Inflation Parameter

Zudem ist es noch sinnvoll die Auflösung der Costmap anzupassen. Die Auflösung der Costmap ist unter den Punkten „global_costmap“ und „local_costmap“ als „resolution“ zu finden (siehe Abbildung 50). Der Wert ist dabei in Meter pro Zelle angegeben. Wird dieser Wert verringert, erhöht sich die Auflösung und kann so zu einer genaueren Pfadplanung führen. Die benötigte Rechenleistung erhöht sich jedoch, weshalb dieser Wert nicht zu klein sein sollte. Im Versuch hat sich ein Wert zwischen 0.025 und 0.1 bewährt.

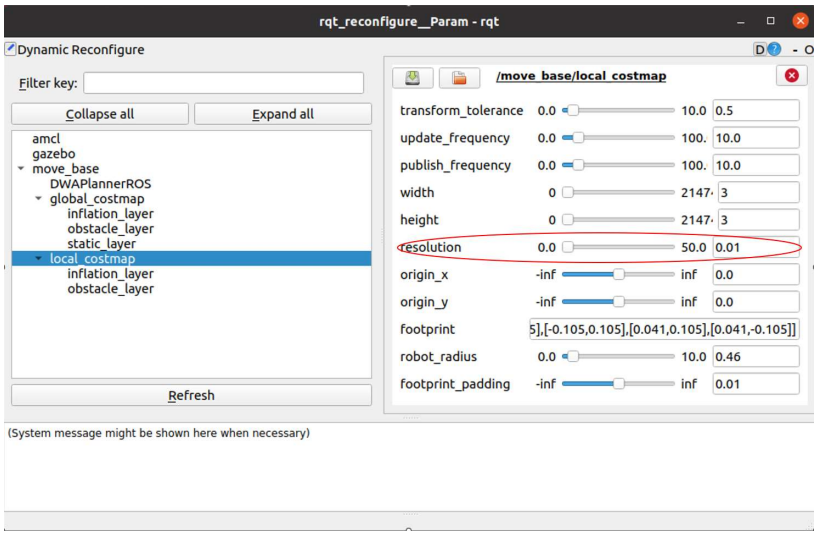


Abbildung 50 Costmap Auflösung einstellen

Die Änderungen der Parameter werden direkt in der Visualisierung sichtbar. In Abbildung 51 ist der direkte Vergleich zwischen den Standardparametern und den optimierten Parametern zu sehen. Damit ist die Optimierung abgeschlossen und es sollte kein unerwünschtes Verhalten seitens des Roboters auftreten.

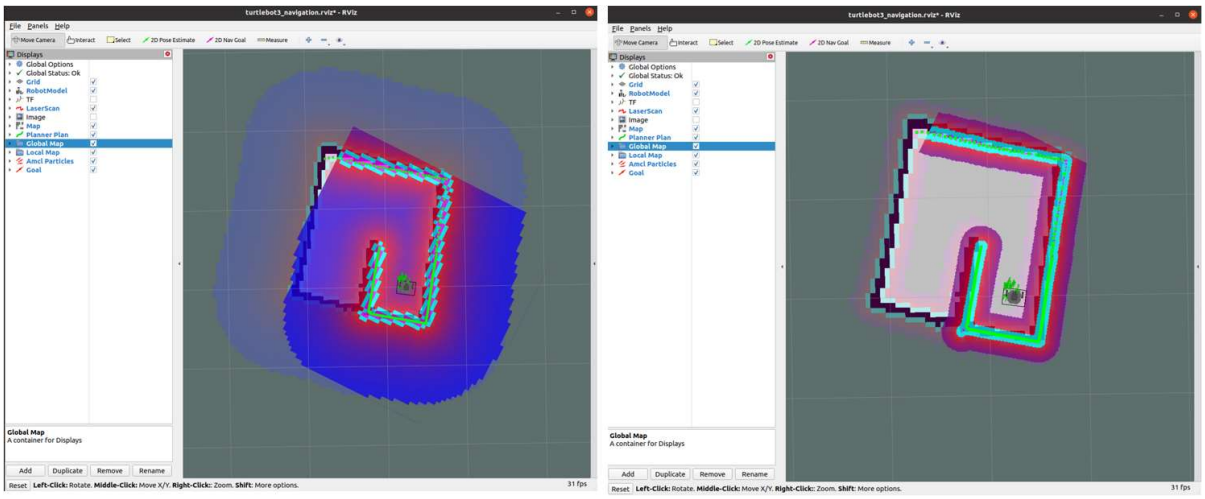


Abbildung 51 Costmap Optimierung; links: vor Optimierung; rechts: nach Optimierung

4.5.3 Navigation

Damit ist die Lokalisierung des Roboters innerhalb der Karte und die Optimierung der Navigationsparameter abgeschlossen und die Navigation kann vorgenommen werden. Als erstes sollte das teleop-Package geschlossen werden. Wird dies nicht beendet, werden dem Roboter Geschwindigkeitswerte aus zwei Quellen vorgegeben, was zu Komplikationen führen kann. Um dem Roboter ein Ziel vorzugeben, muss die „2D Nav Goal“-Schaltfläche angeklickt werden. Die Bedienung ist wie bei der Einstellung der initialen Pose. Auch hier wird der gewünschte Zielpunkt innerhalb der Karte angeklickt und die Zielorientierung ausgewählt. Nach Loslassen der Maustaste wird der Pfad zum Ziel berechnet und der Roboter setzt sich in Bewegung. Dieser Vorgang ist in Abbildung 52 dargestellt.

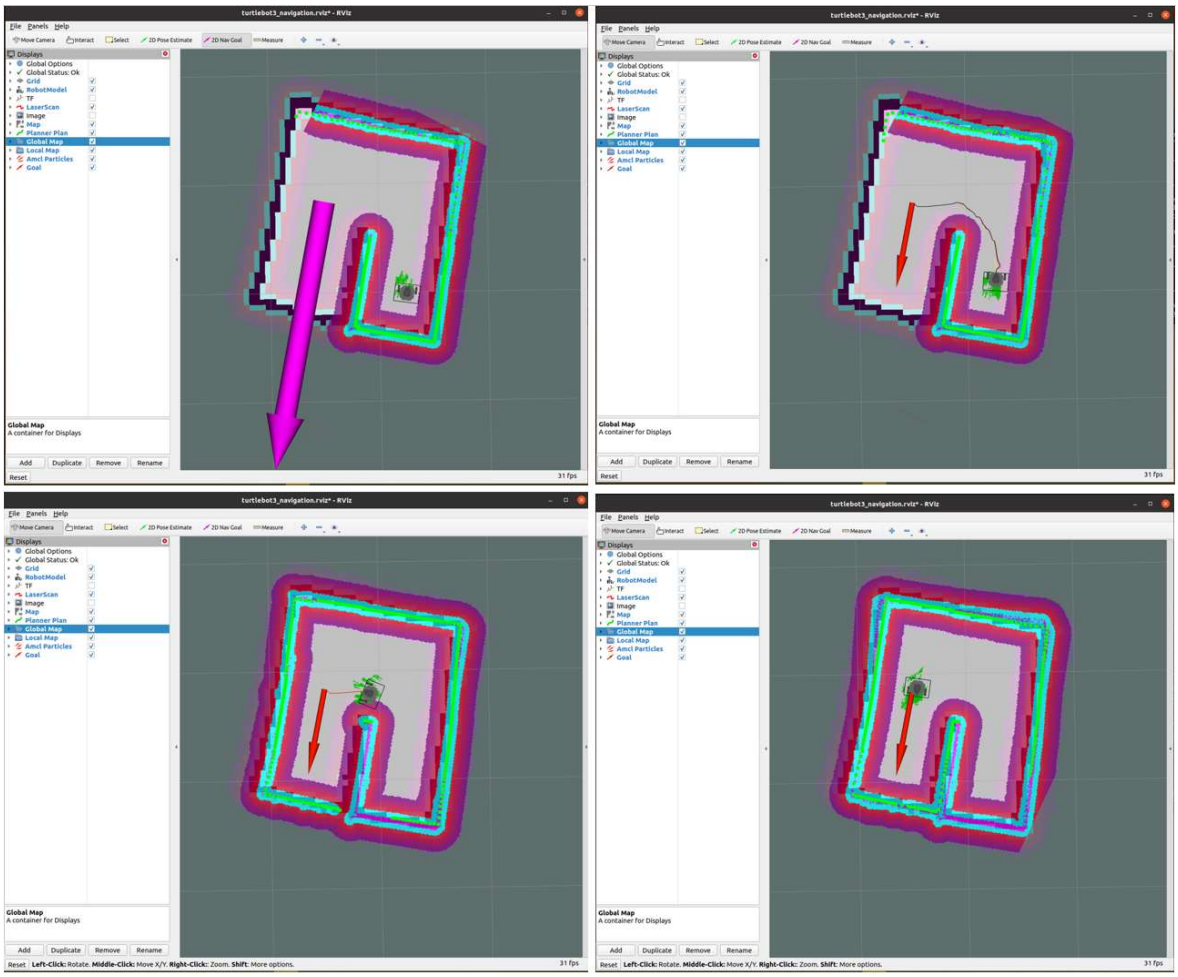


Abbildung 52 Schritt 4: Zielpose eingeben

Bei näherer Betrachtung sind die verschiedenen Pfadplaner erkennbar. Der globale Planer erstellt direkt einen Weg zum Zielpunkt und der lokale Planer plant den Weg nur eine kurze Zeit voraus.

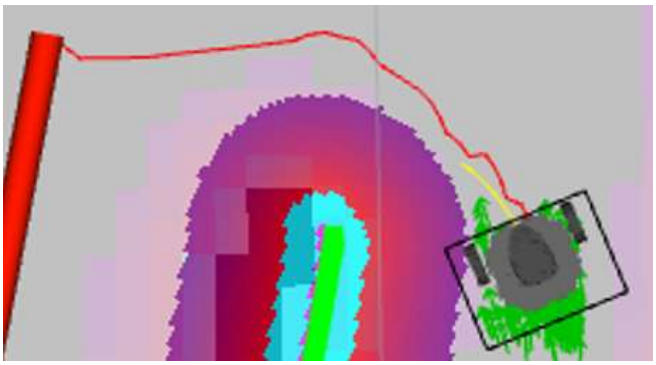


Abbildung 53 Globale und lokale Pfadplanung

Die beiden Planer werden in Abbildung 53 dargestellt. Die rote Linie folgt dem globalen Plan exakt und die gelbe Linie ist der Pfad des lokalen Planers.

Navigation mit nicht kartierten Objekten

Im nächsten Schritt soll ein nicht kartiertes Objekt in die Arena gestellt werden, um zu sehen, wie die Pfadplanung auf dieses Objekt reagiert. Das Navigations-Package bleibt dabei aktiv. Die Arena aus Abbildung 40 wird mit einem Styroporquader erweitert. Dies ist in Abbildung 54 dargestellt.

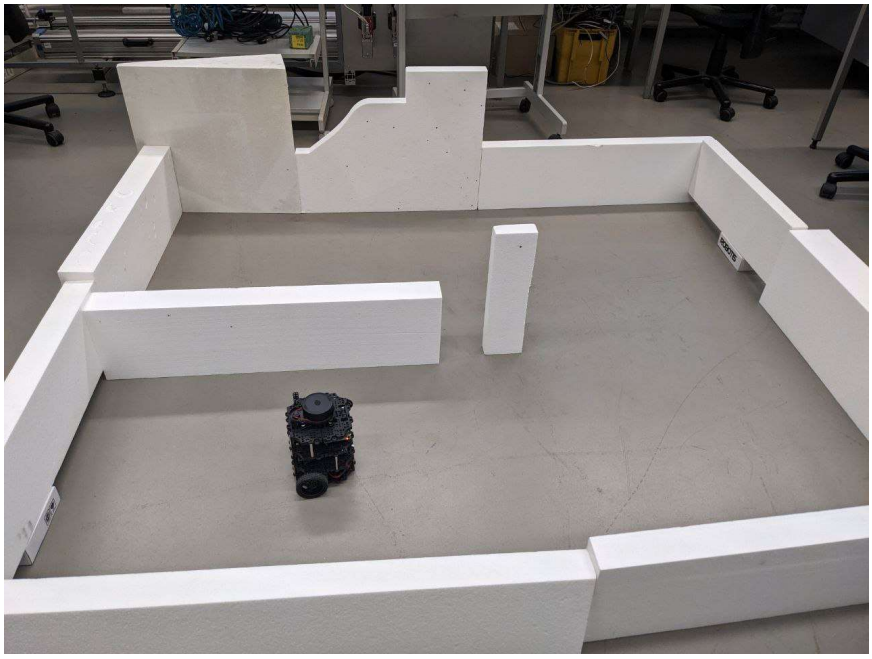


Abbildung 54 Arena mit nicht kartiertem Objekt

Wenn der Roboter sich nah genug am Objekt befindet, wird dieses direkt in der Visualisierung dargestellt. Nachdem das neue Objekt erkannt wurde, wird eine entsprechende lokale Costmap um dieses Objekt gebildet. Wird nun eine Zielpose in die untere linke Arenahälfte gesetzt, muss der Roboter auf das neue Objekt reagieren und die Pfadplanung dementsprechend anpassen. Dieser Vorgang ist in Abbildung 55 dargestellt.

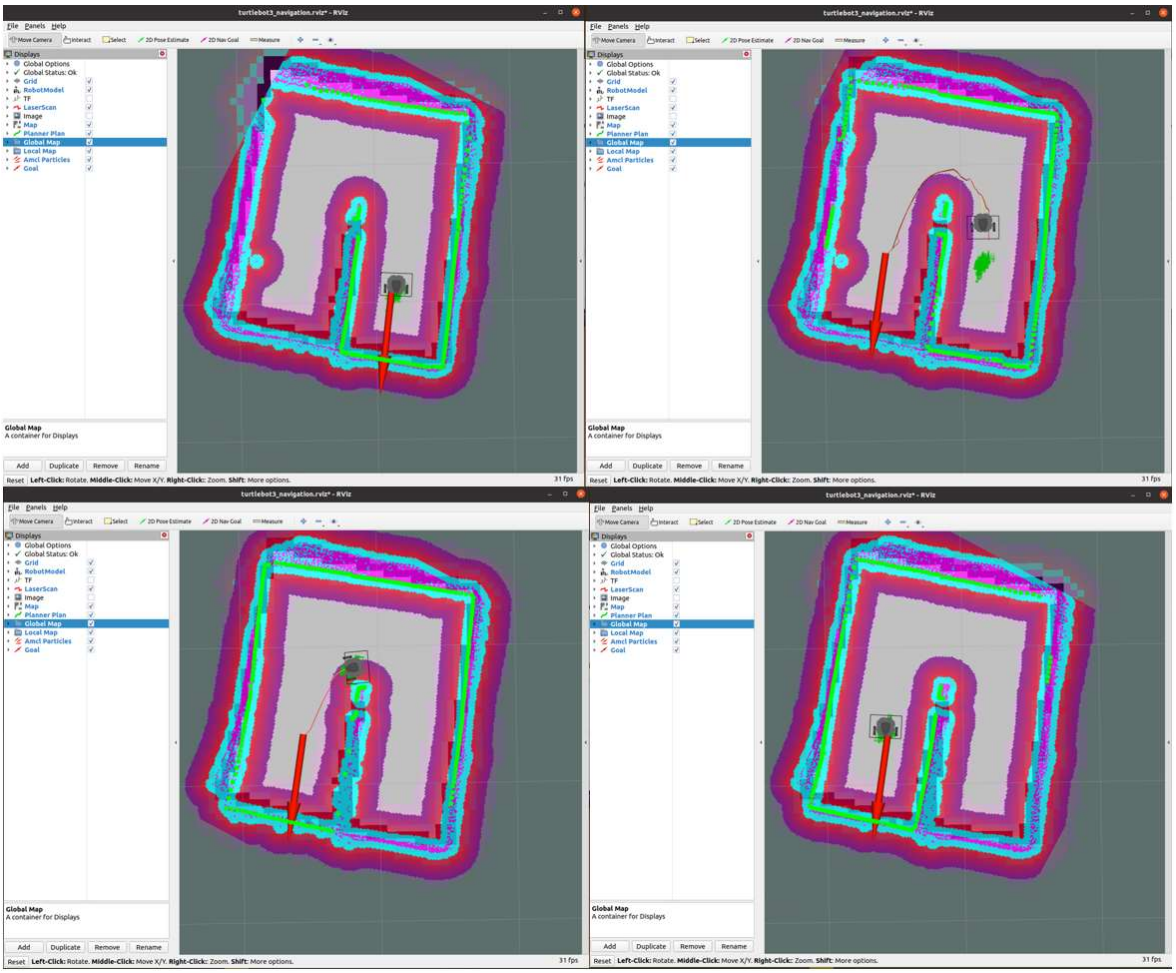


Abbildung 55 Navigation mit nicht kartiertem Objekt

Im letzten Schritt soll der Roboter auf dynamische Hindernisse reagieren. Um dies zu testen, kann der Styroporquader in die Hand genommen und vor den Roboter gestellt werden, um diesen so zu einer Pfadänderung zu zwingen. Alternativ kann sich auch eine Person vor den Roboter stellen. Hier soll die hohe Dynamik der verwendeten Pfadplanung dargestellt werden.

4.5.4 Fazit

Damit ist das letzte Praktikumsmodul beendet. Ziel dieses Modules war es die Navigation mit dem Roboter kennen zu lernen, die Pfadplanung des Roboters nachvollziehen zu können und zu sehen, wie der Roboter aufgrund des DWA-Planers auf dynamische Umgebungsänderungen reagiert. Des Weiteren wurden gezeigt, wie Parametereinstellungen vorgenommen werden können, um die Pfadplanung während der Navigation zu optimieren.

Der Zeitaufwand bei diesem Modul beträgt ca. 60-75 min.

5 Fazit und Ausblick

Ziel dieser Arbeit war es einen Praktikumsversuch zu entwickeln, der die Grundlagen der mobilen Robotik vermittelt. Darunter wurden Themengebiete behandelt, wie Navigation, Lokalisierung, Kartierung einer Umgebung und die Verwendung der Robotersoftware, sowie die Inbetriebnahme eines Systems. Die einzelnen Praktikumsmodule wurden ausführlich beschrieben und die technischen Grundlagen in den vorherigen Grundlagenkapiteln besprochen. Mit Hilfe dieser Arbeit ist es möglich die grundlegenden Aufgaben eines mobilen Roboters, speziell mit dem TurtleBot3, zu bewältigen und die einzelnen Schritte nachvollziehen zu können. Der Praktikumsversuch wurde in vier Module unterteilt, die je einen wichtigen Schritt in der Gesamtkonzeption einnehmen. Das erste Praktikumsmodul sollte den Umgang mit der verwendeten Robotersoftware ROS vermitteln und die zugrundeliegende Kommunikationsart zwischen den Teilnehmern innerhalb einer ROS-Umgebung darstellen. Dazu wurde eine ROS-Arbeitsumgebung von Grund auf neu erstellt und ein einfaches Publisher/Subscriber-Model gebildet. Das Augenmerk lag dabei nicht auf der Erstellung des Codes, sondern auf der Implementierung, Erstellung und Ausführung eines ROS-Packages und der Darstellung der Kommunikation innerhalb einer ROS-Umgebung. Zudem wurden eine Reihe von wichtigen ROS-Befehlen verwendet und diese an einem kurzen Beispiel dargestellt. Im zweiten Praktikumsmodul wurde ROS praktisch angewendet, indem der TurtleBot3 Roboter über eine Tastatur ferngesteuert wurde. Hierzu wurde beschrieben, wie der TurteBot3 hochgefahren wird und eine Verbindung zu dem Hauptrechner aufbauen kann. Zudem wurde gezeigt, welche ROS-Packages notwendig sind und wie diese zu implementieren waren. Im dritten Praktikumsmodul wurde eine selbst erstellte Umgebung von dem TurtleBot3 mit Hilfe von SLAM kartographiert. Dies stellt einen wesentlichen Teil in der mobilen Robotik dar, da mobile Roboter oft in einer Umgebung agieren müssen, die von Anfang an nicht bekannt ist. Dazu wurde das entsprechende SLAM-Package geladen und ausgeführt. Die Darstellung der Sensordaten und der Karte erfolgte mit dem RViz Tool. Mit diesem Tool konnte nachvollzogen werden, was der Roboter wahrnimmt und wie diese Daten verarbeitet werden, um zum Schluss eine komplette Karte der Umgebung zu ergeben. Im Anschluss wurden die nötigen Befehle gezeigt, um die erstellte Karte zu speichern. Im letzten Modul wurde die Navigation innerhalb der erstellten Karte mit dem TurtleBot3 besprochen. Hier wurden die Lokalisierungs- und Navigationsmethoden aus den Grundlagenkapitel ausgiebig getestet und dargestellt. Zudem wurde eine Anleitung gegeben, wie die Navigationsparameter angepasst werden können, um eine hinreichend genaue Navigation zu erhalten.

Zusammenfassend bilden die besprochenen Module einen guten ersten Einstig in die Arbeit mit einem kommerziell verfügbaren Roboter, unter Verwendung von Open-Source-Programmen.

Die in diesem Praktikumsversuch beschriebenen Anwendungen bilden in erster Linie die Grundlagen der mobilen Robotik ab. Für weitere Praktikumsversuche sind noch eine Vielzahl von Möglichkeiten offen, um die Arbeit mit mobilen Robotern zu vermitteln. Ein Vorschlag wäre die Erweiterung des Roboters mit einer Kamera. Mit Hilfe dieser können Bilddaten verarbeitet werden, womit sich neue Anwendungsbereiche eröffnen. Es könnte z.B. ein Versuch umgesetzt werden, in dem der Roboter einer Person folgen muss oder einem Gegenstand mit einer bestimmten Form und Farbe. Mit einer Kamera könnte die Funktionalität des Roboters erhöht werden. Dies ist nur ein Beispiel, wie die Praktikumsversuche erweitert werden könnten. Auch diese Anwendungen sind innerhalb von ROS verfügbar und können als fertige Packages verwendet werden.

6 Literaturverzeichnis

ALL3DP. (31. 1 2023). Von https://all3dp.com/2/raspberry-pi-vs-jetson-nano-differences/ abgerufen

Azure.Microsoft. (25. 1 2023). Von https://azure.microsoft.com/de-de/resources/cloud-computing-dictionary/what-is-middleware/ abgerufen

Bittel, P. D. (2. 2 2023). *home.htwg-konstanz*. Von http://www-home.htwg-konstanz.de/~bittel/ain_robo/Vorlesung/03_Roboterkinematik.pdf abgerufen

Burgard, W., Stachniss, C., Bennewitz, M., Tipaldi, D., & Spinello, L. (2013). *ais.informatik.uni-freiburg*. Von http://ais.informatik.uni-freiburg.de/teaching/ss13/robotics/slides/16-graph-slam.pdf abgerufen

Conrad.de. (20. 01 2023). Von https://www.conrad.de/de/ratgeber/technik-einfach-erklaert/mems.html#einsatz abgerufen

Douglas, B. (8. 7 2020). *Youtube*. Von https://www.youtube.com/watch?v=saVZtgPyyJQ abgerufen

Fox, D., Burgard, W., & Thrun, S. (1997). *ri.cmu.edu*. Von https://www.ri.cmu.edu/pub_files/pub1/fox_dieter_1997_1/fox_dieter_1997_1.pdf abgerufen

Generation Robots. (2. 2 2023). Von https://www.generationrobots.com/de/402707-turtlebot3-burger.html abgerufen

heise. (12. 2 2023). Von https://www.heise.de/tipps-tricks/Ubuntu-SSH-Wie-und-Warum-4203356.html abgerufen

Hertzberg, J., Lingemann, K., & Nüchter, A. (2012). *Mobile Roboter.* Springer Vieweg.

IFR. (6 2021). Von https://ifr.org/downloads/hidden/Information_Paper_Mobile_Robots_v01.pdf?utm_source=CleverReach&utm_medium=email&utm_campaign=Paper+Download&utm_content=Mailing_12323895 abgerufen

Iso.org. (2021). Von https://www.iso.org/obp/ui/#iso:std:iso:8373:ed-3:v1:en:term:4.16 abgerufen

Jaeger, M. (23. 2 2022). *atera.com*. Von https://www.atera.com/de/blog/was-ist-dhcp/#:~:text=DHCP%20steht%20f%C3%BCr%20Dynamic%20Host,in%20Netzwerk%20Services%20einw%C3%A4hlen%20k%C3%B6nnen. abgerufen

Macnica. (20. 1 2023). Von https://www.macnica.eu/de/knowledge-base/technology-transfer/2012-10-09-ein-gyroskop-fur-jeden-roboter abgerufen

Mathworks. (2. 2 2023). Von https://www.mathworks.com/help/vision/ug/visual-slam-with-an-rgbd-camera.html abgerufen

NavVis. (6. 2 2023). Von https://www.navvis.com/de/technology/slam abgerufen

Pohl, P. D. (2014). *Robot Operating System (ROS): Safe & Insecure.* SoftCheck GmbH Köln.

Pyo, Y., Cho, H., Jung, R., & Lim, T. (2017). *ROS Robot Programming.* ROBOTIS Co., Ltd.

RealPython. (21. 9 2022). Von https://realpython.com/if-name-main-python/#:~:text=name%2Dmain%20idiom.-,In%20Short%3A%20It%20Allows%20You%20to%20Execute%20Code%20When%20the,is%20executed%20as%20a%20script. abgerufen

Robotis e-Manual[a]. (2. 2 2023). Von https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/ abgerufen

Robotis e-Manual[b]. (3. 5 2023). Von https://emanual.robotis.com/docs/en/platform/turtlebot3/bringup/#bringup abgerufen

Robotis e-Manual[c]. (5. 3 2023). Von https://emanual.robotis.com/docs/en/platform/turtlebot3/basic_operation/#basic-operation abgerufen

Robotis e-Manual[d]. (5. 3 2023). Von https://emanual.robotis.com/docs/en/platform/turtlebot3/slam/ abgerufen

Robotis e-Manual[e]. (5. 3 2023). Von https://emanual.robotis.com/docs/en/platform/turtlebot3/navigation/#navigation abgerufen

Robotshop. (2. 2 2023). Von https://eu.robotshop.com/de/products/rosbot-nano abgerufen

Ros. (25. 1 2023). Von https://www.ros.org/blog/ecosystem/ abgerufen

Scanner-Imagefact. (2. 2 2023). Von https://www.scanner.imagefact.de/de/depthcam.html#:~:text=Der%20synchronisierte%20Ausgabestream%20von%20Tiefe,nicht%20sichtbares%20codiertes%20Punktmuster%20aus. abgerufen

Siciliano, B., & Oussama, K. (2016). *Springer Handbook of Robotics.* Springer Verlag.

spacehal.github. (23. 1 2023). Von https://spacehal.github.io/docs/robotik/wheelEncoder abgerufen

subscription.packtpub. (13. 2 2023). Von https://subscription.packtpub.com/book/hardware-and-creative/9781782175193/1/ch01lvl1sec11/creating-a-catkin-workspace abgerufen

unixtimestamp. (15. 2 2023). Von https://www.unixtimestamp.com/index.php abgerufen

Vectornav. (21. 1 2023). Von https://www.vectornav.com/resources/inertial-navigation-articles/what-is-an-inertial-measurement-unit-imu abgerufen

Velden, L. (2014). *algorithms.discrete.ma.tum[a]*. Von https://algorithms.discrete.ma.tum.de/graph-algorithms/spp-dijkstra/index_de.html abgerufen

Velden, L. (2014). *algorithms.discrete.ma.tum[b]*. Von https://algorithms.discrete.ma.tum.de/graph-algorithms/spp-a-star/index_de.html abgerufen

Waveshare[a]. (31. 1 2023). Von https://www.waveshare.com/product/jetbot-ros-ai-kit.htm?sku=22791 abgerufen

Waveshare[b]. (31. 1 2023). Von https://www.waveshare.com/wiki/How_to_Install_Jetson_Nano_Image abgerufen

Wiki.ros[a]. (10. 1 2018). Von http://wiki.ros.org/costmap_2d#Component_API abgerufen

Wiki.ros[b]. (14. 4 2019). Von http://wiki.ros.org/Packages abgerufen

Wiki.ros[c]. (18. 10 2022). Von http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29 abgerufen

Wiki.ros[d]. (26. 3 2020). Von https://wiki.ros.org/catkin/conceptual_overview abgerufen

Wiki.ros[e]. (27. 1 2019). Von http://wiki.ros.org/ROS/Tutorials/rosdep abgerufen

Wiki.ros[f]. (2. 11 2015). Von http://wiki.ros.org/roscpp abgerufen

Wiki.ros[g]. (8. 11 2017). Von http://wiki.ros.org/rospy abgerufen

Wiki.ros[h]. (4. 3 2017). Von http://wiki.ros.org/std_msgs abgerufen

Wiki.ros[i]. (9. 5 2019). Von http://wiki.ros.org/roscore abgerufen

xovi. (23. 1 2023). Von https://www.xovi.de/was-bedeutet-peer-to-peer/ abgerufen

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit eigenständig und ohne fremde Hilfe angefertigt habe. Textpassagen, die wörtlich oder dem Sinn nach auf Publikationen oder Vorträgen anderer Autoren beruhen, sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Halle (Saale), 10.03.2023

_____

Name (Unterschrift)