Hochschule Mannheim

**Schriftliche Arbeit**

# Kartierung eines und Navigation in einem Labyrinth

*David Siekacz*

05. August 2022

Betreut durch Prof. Dr. Thomas Ihme

**Zusammenfassung:**
Die Robotik nimmt ein immer größer werdendes Forschungsgebiet innerhalb der Wissenschaft ein. Roboter werden für unser alltägliches Leben immer wichtiger. Doch um sich in unser Umfeld, z.B. als Staubsaugroboter, integrieren zu können, müssen diese auch in diesem navigieren können. In diesem Projekt haben wir uns mit der Kartierung und darauf aufbauenden Navigation innerhalb eines unbekannten Labyrinthes beschäftigt. Wie wir Menschen zum Finden eines Ortes eine Karte brauchen, so brauchen auch Roboter diese, um zu navigieren. Daher geht es in der folgenden Arbeit sowohl um die Kartierung eines Labyrinthes, als auch die sichere Navigation in diesem, sodass jeder Punkt optimal erreicht werden kann.

# Inhaltsverzeichnis

# 1 ROS

## 1.1 Überblick

Das sogenannte Robot Operating System, kurz ROS genannt, ist ein Open-Source Betriebssystem für Roboter, welches viele Tools und Software-Bibliotheken bietet. Die Entwicklung von ROS begann 2007 im Stanford Artificial Intelligence Laboratory. Ab 2009 wurde ROS im Institut Willow Garage fortgeführt, bis ROS ab 2012 von der OSRF (Open Source Robotic Foundation) unterstützt wird. Seit der Veröffentlichung von ROS gibt es regelmäßig Updates und neuere Versionen von ROS, welche hauptsächlich mit Betriebssystem Linux, aber auch Windows oder MacOS kompatibel sind.

Die grundsätzliche Idee von ROS ist, alle Vorteile von vergleichbaren Produkten zu vereinen, während die Nachteile der jeweiligen Produkte behoben werden. Die Hauptbestandteile von ROS sind: Hardwareabstraktion, Gerätetreiber, Nachrichtenaustausch zwischen Programmen und Programmteilen und die Paketverwaltung. Alle diese Bestandteile sorgen dafür, dass der Roboter einfacher auf Daten zugreifen kann. Es handelt sich dabei um den aktuellen internationalen Standard, welcher auf den meisten Systemen einwandfrei läuft und dabei wenig Leistung beansprucht. Ein Gerätetreiber ist ein Programm, welches die Interaktion mit angeschlossenen, eingebauten oder virtuellen Geräten steuert. Beim Nachrichtenaustausch werden Nachrichten zum Empfänger versendet, bei welchen es sich um Signale oder Datenpakete handeln kann. Die Paketverwaltung hilft bei der Verwaltung von Software, die in Form von Programmpaketen vorliegt. Ein weiteres Ziel von ROS ist, oft wiederverwendbare Funktionen zugänglich zu machen.

## 1.2 Aufbau

Programme werden in ROS als Pakete implementiert. Ein Paket besteht aus einer Manifestdatei, welche Metadaten (z.B. den Autor) über das Paket beinhaltet, sowie ausführbaren Skripten (Programmen). Pakete werden meist unabhängig voneinander entwickelt, können doch miteinander kommunizieren (siehe Datenaustausch). Pakete können (ähnlich zu Apps auf einem Handy) einzeln installiert bzw. deinstalliert werden, wobei einige Pakete andere benötigen, um zu funktionieren. Die Programme werden meist in den Programmiersprachen C++ oder Python geschrieben, welche beide die größte Integration in das ROS vorweisen.

Der ROS-Core ist der Mittelpunkt des Betriebssystems. Dieses Programm steuert alle Vorgänge und sorgt dafür, dass Daten an korrekte Pakete weitergegeben werden, sowie vieles mehr.

Eine Grundkomponente von ROS sind sogenannte Nodes (Knotenpunkte), welche von Programmen initiiert werden können: Jede dieser Nodes hat eine gewisse Funktion (z.B. das ansteuern von Motoren). Diese Punkte sind des Weiteren austauschbar, sodass es für verschiedene Motorenmodelle beispielsweise andere Nodes gibt, welche jedoch alle die gleiche Funktion erfüllen, nämlich das ansteuern der Motoren.

## 1.3 Datenaustausch

Der Austausch von Daten kann sich als Graph vorgestellt werden. Jeder Knotenpunkt kann Daten aussenden oder empfangen. Der Datentyp, welcher ausgetauscht wird, kann beliebig bestimmt werden, jedoch beruht alles auf vier Grundtypen.

1. Ganzzahlen (Integer)
2. Kommazahlen (Float)
3. Zeichen (Char)
4. Ja/Nein (Boolean)

Basierend auf diesen Grundtypen können neue Datentypen gebaut werden. So kann man beispielsweise sagen, dass der Datentyp Auto

1. eine Kommazahl hat, welche den Füllstand beschreibt,
2. eine ganze Zahl hat, welche die Anzahl an Passagieren beschreibt,
3. ein aus mehreren Zeichen bestehendes Kennzeichen hat,
4. einen Ja/Nein-Wert hat, welcher bestimmt, ob der Motor an ist,

hat. Die Datenstruktur der ausgetauschten Daten wird Nachricht genannt. Nodes tauschen also Nachrichten untereinander aus. Jede Nachricht bekommt eine sogenannte Topic, was die Nachricht und deren Struktur eindeutig identifiziert. Ein Beispiel hierfür ist die *cmd_vel* Topic, welche Nachrichten des Types *geometry_msgs/Twist* weitergibt. Diese Daten geben Informationen über die aktuelle Begungsrichtung wieder. Zur vereinfachung zeige ich nur die Richtungen der Bewegung im 3D-Raum, nicht aber der Drehung, da diese für das Verständnis nicht notwendig ist. Die Struktur der Nachricht sieht daher (in gekürzter Fassung folgendermaßen aus:

linear:

- float64 x
- float64 y
- float64 z

*float64* stellt in der Programmierung eine Kommazahl dar. Daraus folgt, dass diese Nachricht drei Zahlen für jede Richtung hat, woraus sich die Bewegungsrichtung des Roboters ergibt. Im Falle meiner NAvigation ist die z-Koordinate irrelevant, da der Roboter sich nicht nach oben/unten bewegt, sondern nur auf der 2 dimensionalen Ebene.

### 1.3.1 Publisher-Subscriber

Nodes sind von Programmen initiierte Knotenpunkte, auf welche das Programm ,welches sie initiiert hat, Zugriff hat. Über diese Punkte können Daten bereitgestellt werden, indem das Programm diese publiziert (Publisher). Dies kann man sich wie ein Forum darstellen, in welchem eine Person mit einem Megafon die Daten verkündet. Andere Nodes (Listener/Subscriber) können diesen Punkt nun abonnieren, was bedeutet, dass die Daten an sie weitergeleitet werden. Diese Listener-Nodes sind vergleichbar mit Personen, die das Forum betreten und daher alles hören, was der Redner (Publisher-Node) sagt. bei dieser Variante des Datenaustausches bestimmt der Publisher, wann Daten verbreitet werden. Mehrere Listener-Nodes können eine Publisher-Node abonnieren.

### 1.3.2 Services

Services bilden das Gegenstück zum Publisher-Subscriber-Modell. Hierbei stellt eine Node Aufgaben bereit, welche sie auf Anforderung erfüllt. Node A bietet beispielsweise an, die Lichter in einem Raum an bzw. auszuschalten. Node B kann in diesem Fall auf den Service ”Ändere Lichtzustand”von Node A zugreifen, woraufhin Node A das Licht an bzw. ausschaltet. Ohne Aufruf führt Node A dies jedoch nicht durch. Dies zeigt, dass bei diesem Modell nicht, die Node, welche einen Service bereitstellt, den Zeitpunkt der Ausführung definiert, sondern ein anderer Punkt, welcher den Service aufruft.

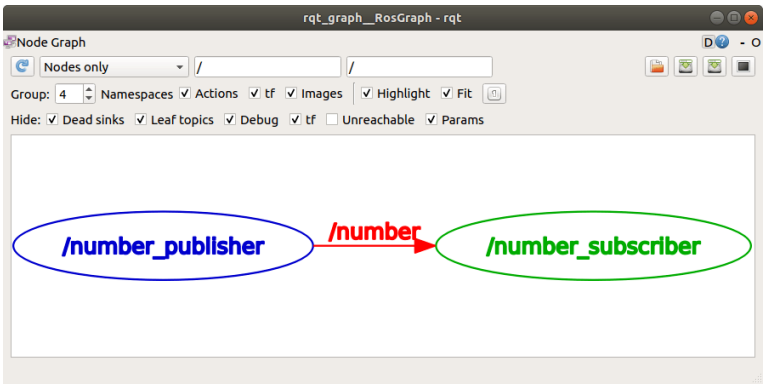


**Abbildung 1:** Ein vereinfachter Graph
[0]

## 1.4 Visualisierung des Datenaustausches

Der Datenaustausch zwischen verschiedenen Nodes kann mit Hilfe einfacher Tools visualisiert werden. Ein sehr wichtiges Tool ist *rqt_Graph*, welches alle Knoten und deren Zusammenhänge als Graphen anzeigt. Dabei werden Nodes als Punkte und deren Verbindungen als Linien angezeigt. Verbindungen werden mit dem Namen der Topic an Daten, die Übertrieben werden, beschriftet. Dies ist vereinfacht in Abbildung 1 dargestellt. Hierbei veröffentlicht die Node *number_publisher* Daten der Topic */number* an die Node *number_subscriber* (rechts), welche die linke Node abonniert hat.

## 1.5 Drahtlose Übertragung

Wie bereits erwähnt, besitzt der Turtlebot einen Raspberry Pi. Dieser besitzt zwar für seine Größe vergleichsweise viel Leistung, jedoch wäre es natürlich vorteilhaft, die Rechenleistung eines herkömmlichen Desktop-PCs zu nutzen. Genau deshalb ermöglicht es das ROS, über mehrere Geräte hinweg zu kommunizieren. Das Herzstück bildet hierbei das ROS-Core Programm, welches alle Vorgänge innerhalb des Systems steuert. Dieses Programm muss auf einem der miteinander kommunizierenden Geräte aktiv sein. Mit Hilfe IP-Adressen [1] der Geräte kommunizieren die Teilnehmer untereinander in einem Netzwerk, als ob alle Pakete (Programme) auf einem Geräte ausgeführt werden würden. Dies ermöglicht das ausführen von Rechenintensiven Programmen auf einem Computer, während kleinere Programme, sowie der ROS-Core auf dem Turtlebot laufen. Das Gerät, auf welchem der Ros-Core läuft, wird auch ROS-Master genannt. Um beide Geräte zu verbinden, benötigt es nur wenige Einstellungen. Nachdem beide Geräte sich im gleichen Netzwerk befinden müssen die Parameter: **ROS_IP** und **ROS_Master _URI** eingestellt werden. Ersterer beschreibt die IP-Adresse des Gerätes (für beide Geräte unterschiedlich) und letzterer Parameter beschreibt die IP-Adresse des ROS-Masters (für beide gleich, in meinem Fall ist es die Ip-Adresse des Turtlebot). Durch das ROS bauen beide Geräte eine Verbindung miteinander auf und kommunizieren miteinander.

## 1.6 Simulation und Visualisierung

Gazebo ist ein Open-Source-Programm [2], welches eine vollständige Simulation von Robotern, Umgebungen und Messungen ermöglicht. Robotermodelle können dazu modelliert werden, ebenso wie Umgebungen. Der Roboter kann sich daraufhin, wie in der realen Welt, bewegen und Messungen durchführen. Durch eine realistische Physik-Engine[3] können Messungen wie in der

[1] Ähnlich einer Adressanschrift: Zahlenkombination zur eindeutigen Identifikation eines Gerätes in einem Netzwerk
[2] Frei zur Verfügung gestellt
[3] Virtuelle Simulation physikalischer Kräfte

echten Welt simuliert werden. D.h. der simulierte Roboter, in meinem Fall der Turtlebot, erhält Messdaten durch seinen Sensor, welche basierend auf der virtuellen Welt errechnet werden. Da ich einen größeren Teil meiner Arbeit von zuhause verrichtet habe, wurde dieses Programm oft genutzt, weshalb auch immer Bilder aus diesem Programm, anstelle von Bildern aus der echten Welt, gezeigt werden.

Rviz ist ebenso ein Open-Source-Tool, welches es ermöglicht, Daten, ob gemessen oder errechnet, zu visualisieren. Dafür empfängt ROS Daten der Nodes und präsentiert diese auf geeigneter Weise. So kann beispielsweise die Momentane Bewegungsrichtung als Pfeil, Messdaten des LiDAR-Sensors als Punkte oder eine Errechnete Karte als Hintergrundbild angezeigt werden

# 2 Turtlebot

## 2.1 Überblick

Der Turtlebot ist ein beliebtes Robotermodell der Firma Robotis und kommt in verschiedenen Varianten. Das gute Preis-Leistungs-Verhältnis, sowie die Einsteigerfreundlichkeit, kombiniert mit vielen Integration in andere Programme machen in zu einer guten Wahl für viele Projekte Der Turtlebot 3 besteht aus vier Basisplatten, auf welchen die gesamte Technik des Roboters untergebracht ist. Auf der untersten Basisplatte findet man zwei Servomotoren der X-Series von Dynamixel vor, welche dem Turtlebot seine Mobilität verleihen, auf den anderen Plattformen ist ein Raspberry PI 3, welcher unter Umständen auch durch einen Raspberry Pi 4 ersetzt werden kann, sowie ein Einplatinencomputer vorzufinden. Bei Letzterem handelt es sich standardmäßig um einen Intel Joule 570x. Auf der obersten Basisplatte findet man Platz für einen Sensor vor, dort kann eine Kamera, allerdings auch ein Lasersensor oder ähnliches angebracht werden. Auf unserem Turtlebot ist ein LiDAR-Sensor montiert. Dieser Sensor sendet Laserimpulse aus und kann durch das zurückgestrahlte Licht den Raum exakt vermessen. Der Roboter tastet die Umgebung mit den Laserimpulsen ab, was dazu führt, dass er Messpunkte erhält.

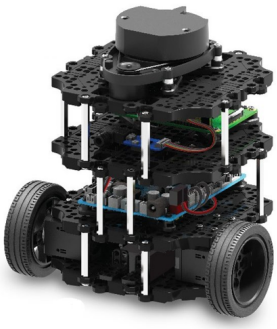

**Abbildung 2:** Der Turtlebot 3 in der „Burger“ Variante
[0]

## 2.2 LiDAR-Sensor

Dieser Sensor sendet Laserimpulse aus und kann durch das zurückgestrahlte Licht den Raum exakt vermessen. Der Roboter tastet die Umgebung mit den Laserimpulsen ab, was dazu führt, dass er Messpunkte erhält. Diese Messpunkte kann der Roboter als Wände oder andere Formen interpretieren. Allerdings handelt es sich dabei um eine 2D Messung, welche in unserem Fall die Umgebung des Roboters auf Höhe des Sensors scannt. Der Sensor kann sich unabhängig vom Roboter um 360° drehen. Die Distanz eines Punktes ergibt sich aus der Zeitdifferenz zwischen Aussendung und Empfangen einer elektromagnetischen Welle (Licht) im infraroten Bereich. Eine einfache Formel lautet wie folt:

**Abbildung 3:** Der im Turtlebot 3 verbaute LiDAR-Sensor

$$s = c \cdot \frac{t}{2} \tag{1}$$

wobei $s$ die Entfernung, $t$ die Zeit zwischen Aussendung und Empfangen, sowie $c$ die Lichtgeschwindigkeit ist.

Ein Problem für Roboters ist, dass mit Hilfe des Standard Sensor keine Hindernisse erkannt werden, welche nicht auf der Höhe des Sensors sind. Dies ist vor allem für Größere Roboter ein Problem, kann aber aufgrund der Umgebung des Roboters, sowie dessen Größen in der Labyrinth-Umgebung vernachlässigt werden.

### 2.3 Raspberrry Pi

Auf dem Turtlebot ist ein kleiner Computer, ein der Marke Raspberry Pi"eingebaut. Standardmäßig handelt es sich dabei um das Modell der Reihe 3, welches jedoch aus Leistungsgründen mit einem Modell der 4. Generation ausgetauscht worden ist. Dieser Mikrocomputer stellt das Herzstück des Roboters dar und steuert alle Vorgänge. Wie ein jeder Computer besteht auch der Raspberry Pi aus einem Prozessor, Arbeitsspeicher, einer SD-Karte als Festplattenersatz und vielem mehr. Zwar stellt die Firma Raspberry auch eine graphische Benutzeroberfläche bereit, jedoch ist diese zur Programmierung nicht zweckführend, da diese Leistung verbraucht, welche für Berechnungen verwendet werden könnte. Aus diesem Grund verwende ich den Raspberry Pi mit einem einfachen Betriebssystem, welches nur eine Konsole anzeigt, um Leistung zu sparen.

## 3 SLAM

### 3.1 Überblick

SLAM (*Simultaneous Localization and Mapping, dt.: Simultane Positionsbestimmung und Kartierung*) beschreibt den Prozess, in welchem eine Karte durch Messdaten erstellt wird, wobei der Roboter selbst auf der Karte verordnet wird. Im Fall des Turtlebot geschieht dies mithilfe der Messdaten des LiDAR-Sensors, welcher die Entfernung von Objekten durch Laserstrahlen misst (*siehe LiDAR-Sensor*)

### 3.2 Positionierung

Der Roboter wird anfangs auf den Mittelpunkt der, zu Beginn noch nicht vorhandenen Karte, welche erst durch Messdaten entsteht, platziert. Die Position des Roboters wird entsprechend der Bewegung seiner Motoren kalkuliert. Mithilfe des Umfangs der Reifen kann die bewegte Distanz errechnet werden, da dem Roboter die Winkeländerung der Motoren, z.B. eine Drehung dieser um 180°, bekannt ist. Diese werden an die Motoren übermittelt, wobei die Information noch an andere Teile weitergeleitet werden kann, hier der Microchip auf der Hauptplatine des

Raspberry Pi 3 in der Mitte der Roboters, welcher damit Rechnungen, eben zur Positionskorrektur durchführen kann. Vereinfacht gilt daher bei einer geraden Bewegung

$$D = R \cdot U \tag{2}$$

Hierbei steht $D$ für die zurückgelegte Distanz, $R$ für die Umdrehungen der Motoren und $U$ für den Umfang der Reifen. Die Rotation, also die Drehung des Roboters, wird aus der Rotationsdifferenz beider Motoren berechnet. Dreht sich der rechte Motor weiter (größerer Winkel), so dreht sich der Roboter nach links, für eine Rechtsdrehung gilt das Gegenteil. Auf diese Weise kann nur aus den Daten, welche an der Motoren gesendet wird, errechnet werden wie weit und in welche Richtung der Roboter sich bewegt hat. Die benötigten Daten werden aus Odometrie-Daten gelesen. Dies sind interne Daten des Roboters, welche durch Messungen oder (wie oben genannt) Rechnungen erhält. Diese weichen durch Umwelteinflüsse und Messungenauigkeiten leicht von der Realität ab.

## 3.3 Kartierung

Die Kartierung erfolgt mithilfe der durch den LiDAR-Sensor gemessenen Daten. Dieser misst den Abstand des Roboters in verschiedene Richtungen punktartig.

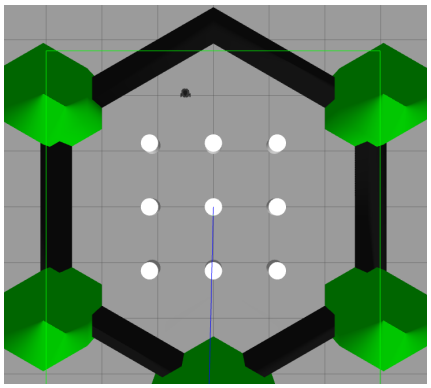

**Abbildung 4:** Roboter (zentriert im oberen Drittel) in einer virtuellen Umgebung

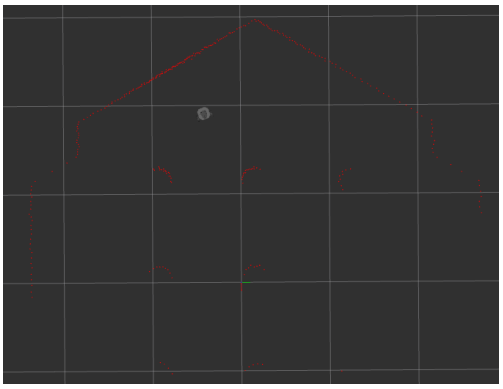

**Abbildung 5:** Momentaufnahme der des LiDAR-Sensors gemessenen Punkte (Rot)

Links erkennt man eine mit Gazebo simulierte Welt, in welcher sich der Roboter befindet, wobei er von Objekten (Säulen, Mauern) umgeben ist. Das rechte Bild ist ein Ausschnitt aus dem vorher erwähnte Programm Rviz, welches die durch den LiDAR-Sensor gemessene Punkte rot darstellt. Es ist zu erkennen, dass das Verbinden der Roten Punkte auf die Ursprüngliche Umgebung schließen lässt. Je näher der Roboter an einem Objekt ist, desto mehr Punkte befinden sich an dessen Stelle

Zwar gibt eine Messung von einem Standpunkt einen akzeptablen Überblick, so werden jedoch viele Messungen von verschiedenen Stellen benötigt. Durch die Positionsänderung (*siehe Positionierung*) kann errechnet werden wo neue Messpunkte im Vergleich zu alten Messpunkten liegen. Diese Positionierung der Punkte relativ zueinander wird schlussendlich zu einer Karte, wobei viele Messungen nötig sind. Da die Karte jedoch nur aus Punkten besteht, muss der Roboter diese noch interpretieren, sodass viele linear angeordnete Punkte z.B. als Wand interpretiert werden.

Um eine komplette Karte zu erhalten Bewegt sich der Roboter durch die ihm noch unbekannte Welt, bis er eine Region durch genug Messpunkte Kartographiert hat und sich nun in Richtung einer ihm unbekannteren Region bewegen kann. Da der Roboter durch den LiDAR-Sensor konstant seine Umgebung misst, kann die Karte konstant verändert werden, vor allem, wenn der Roboter sich in Richtung weniger erforschte Gebiete bewegt.

Während des Anfangs des Projektes wurde der Roboter noch mit Hilfe der Tastatur an einem Rechner oder über einen Logitech-Controller Gesteuert. Danach wurde daraus jedoch eine autonome Navigation, meist in Richtung von Gebieten, welche weniger erforscht sind.

Die Kartierung erfolgt mit Hilfe des *gmapping*-Algorithmus. Dieser schaut sich die gescannten Laserdaten der *sensor_msgs/LaserScan*-Topic an und veröffentlicht drei Datenarten.

1. ein "Occupancy-Grid": 2D-Karte. Jeder Ort (Pixel) hat einen Ja/Nein Wert, welcher beschreibt, ob sich an der Position ein Objekt befindet oder nicht
2. Entropie (Sicherheit über die Erkenntnis verschiedener Punkte: unsichere Punkte werden wahrscheinlicher geändert)
3. Metadaten über die Karte.

Durch die Kartierung dieses Algorithmus wird aus den einzelnen Laserscan-Daten nun eine Karte. Dies wird in den unteren Abbildungen dargestellt. Hierbei handelt es sich um die Karterung der in Abbildung 4 gezeigten Umgebung. Schwarze Punkte wurden als Hindernis erkannt, graue Stellen sind hindernisfreie Punkte. Die durch den LiDAR-Sensor gemessenen Punkte werden in grüner Farbe über die Karte gelegt. In Abbildung 7 und 8 ist zu erkenne, dass SChwarze Punkte nicht direkt mit Gemessenen Punkten des Lasers (grün) übereinstimmen. Dies liegt zum einen darin, dass die Berechnung der Karte Zeit benötigt, weshalb die Punkte noch verarbeitet werden, sowie daran, dass die Aktualisierungsrate der Karte eingestellt werden kann. Während intuitiv eine schnelle Aktualisierungsrate sinnvoll wirkt, ist eine zu schnelle Rate nachteilig, da viel Leistung benötigt wird und mehr Fehler gemacht werden, da ungenaue Messpunkte zu schnellen Schlüssen auf der Karte führen. Bei einer längeren Zeitdauer zwischen Kartenaktualisierungen können nämlich mehr gemessene Punkte in Betracht gemessen werden.

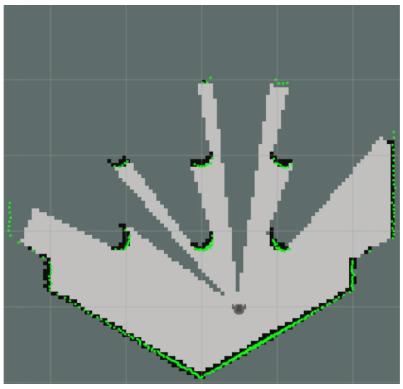

**Abbildung 6:** Kartierung am Anfangspunkt

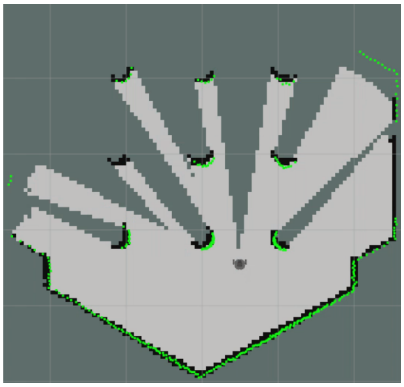

**Abbildung 7:** Kartierung nach erster Bewegung (nach oben)

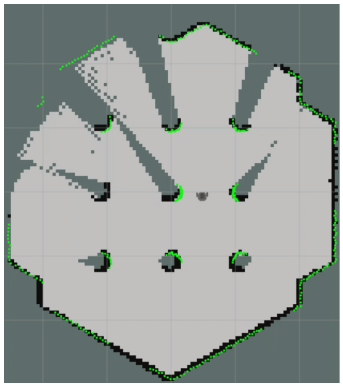

**Abbildung 8:** Kartierung nach weiterer Bewegung

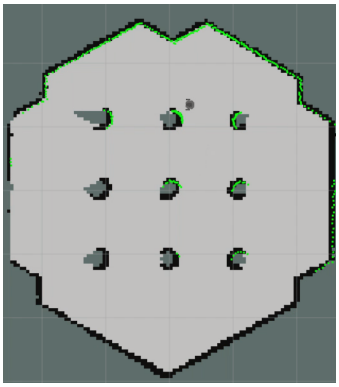

**Abbildung 9:** Kartierung nach Erreichen des oberen Endes

# 4 Pathfinding

## 4.1 Definition

Pathfinding (*engl. Wegfindung*) beschreibt den Prozess der Wegfindung, wobei der kürzestmöglichen Weg von einem Startpunkt zu einem Zielpunkt gefunden werden soll. Dieser Prozess setzt eine Karte der Umgebung (*siehe SLAM*), in welcher ein Weg gefunden werden soll, voraus.

## 4.2 Anwendung in der heutigen Welt

Pathfinding-Prozesse sind von heutiger Technologie nicht mehr zu trennen. Sie sind überall präsent und helfen uns, auch wenn wir es manchmal nicht bemerken . Navigationssysteme müssen die kürzeste Route zu einem Zielpunkt finden, wohingegen Staubsaugroboter Orte in einem Haus erreichen müssen. Signale über Satelliten hinweg werden auch über den kürzesten Weg geleitet. Diesen zu finden bedeutet immer, dass für den Transport weniger Aufwand und Zeit benötigt wird

## 4.3 Algorithmen zur Umsetzung

Für die Wegfindung gibt es viele etablierte Algorithmen, jedoch stellt sich nun die Frage, welcher am Nützlichsten ist. Diese Nützlichkeit wird meist an zwei Faktoren gemessen. Diese sind zum einem die Genauigkeit, welche beschreibt ob ein Algorithmus den besten Weg findet, und zum anderen die Geschwindigkeit. Diese beschreibt, wie effizient ein Algorithmus den idealen Weg findet. Mit Geschwindigkeit ist also nicht die Zeit der Ausführung gemeint, welche mit der Effizienz jedoch stark zusammenhängt, da diese je nach genutzten Bauteilen in Computern variiert. Vielmehr wird, wie erwähnt, die Rechenintensität als Maß genutzt. Diese beschreibt, wie viele Schritte durchgeführt werden müssen, bis der ideale Weg gefunden wurde. Je geringer, desto besser.

Viele Wegfindungsalgorithmen nutzen das gleiche Grundprinzip. Die Karte wird in kleine Vierecke unterteilt. Vom Startpunkt aus wird nun in eine Richtung gegangen und zwar einen Schritt weit. Dabei wird diese Stelle als ”besucht”angesehen, wobei von ihr aus nun in weitere Richtungen gelaufen werden kann (Abbildung 10). Grenzt eine besuchte Fläche an das Ziel an, so wurde ein Weg gefunden. Nachdem anfangs in jede Richtung ein Schritt in die Tiefe gegangen wurde, wird nun erneut in eine andere Richtung gegangen, bis in jede Richtung ein Schritt gegangen bzw. abgesucht worden ist (Abbildung 13). Eine Tiefe (Längeneinheit) wurde somit vollständig abgedeckt. Daraufhin wird nun das Gleiche mit einer Tiefe von zwei zum Startpunkt hin wiederholt wird.

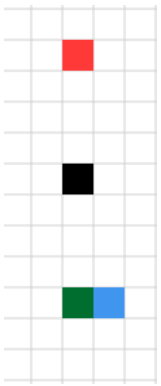

**Abbildung 10:**
1. Feld wird durchlaufen

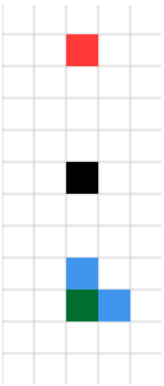

**Abbildung 11:**
2. Feld wird durchlaufen

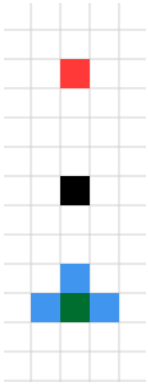

**Abbildung 12:**
3. Feld wird durchlaufen

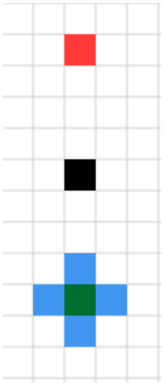

**Abbildung 13:**
4. Feld wird durchlaufen

Dies wird in der unteren Abbildung verdeutlicht. Grün repräsentiert den Startpunkt, Rot den Endpunkt. Schwarz repräsentiert ein nicht passierbares Feld und blaue Felder sind von dem Suchalgorithmus bereits besucht worden. Die dargestellte Suchart stellt den **Breadth-first Search Algorithmus** dar, welche zur oben gegebenen Beschreibung passt. Dabei handelt es sich um einen einfachen Wegfindungsalgorithmus.

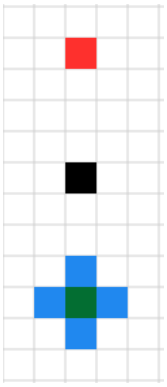

**Abbildung 14:** Besuchte Felder nachdem die Tiefe 1 durchlaufen wurde

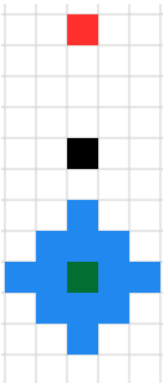

**Abbildung 15:** Besuchte Felder nachdem die Tiefe 2 durchlaufen wurde

Zur Verbesserung der Effizienz nutzen manche Algorithmen weitere Algorithmen, die Schätzalgorithmen, welche man auch Heuristik nennt, die Einfluss darauf nehmen, welche Felder als nächstes besucht werden. Felder, welche in Richtung des Ziels zeigen bzw. sehr wahrscheinlich dorthin führen, werden bevorzugt durchlaufen, Felder, welche vom Ziel weg zeigen, werden hingegen seltener Besucht. Dazu misst der Algorithmus nicht nur die Distanz(Kosten) bis zu einem Punkt, um zu berechnen, ob es sich lohnt, von diesem aus weiter zu suchen, sondern auch die

Distanz zum Ziel. Mathematisch kann man dies als zusammengesetzte Funktion verstehen.

$$f(x) = g(x) + h(x) \tag{3}$$

Hierbei stellt $g(x)$ die Funktion dar, welche die Kosten(Distanz) vom Startpunkt bis zu einem Punkt $x$ berechnet. $h(x)$ beschreibt die vermutete Distanz von $x$ bis zum Zielpunkt. Beide Ergebnisse zusammen ergeben einen Wert, welcher die Kosten über einen Punkt $x$ zum Zielpunkt errechnet. Dies wird für viele Punkte $x$ berechnet, wobei der nächste Schritt von dem Punkt ausgeht, welcher den niedrigsten Wert (also die geringste, errechnete Distanz) hat. Diese Algorithmen sind für viele Verwendungszwecke sehr effizient. Heuristik nutzende Wegfindungsalgorithmen können unter Umständen nachteilig sein, wenn diese in Labyrinthen eingesetzt werden, welche eine hohe Komplexität aufweisen, da Sackgassen, welche nur durch eine Wand vom Ziel getrennt werden, meist durchsucht werden, obwohl sie im Endeffekt nicht Zielführend sind.

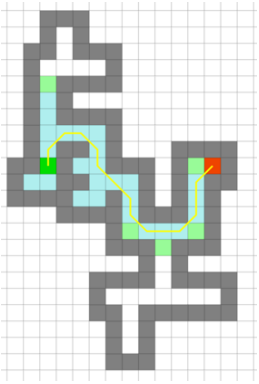

**Abbildung 16:** Wegfindung eines Algorithmus mit Heuristik

## 4.4 Coastmap

Eine gemessene Karte wird zur Navigation zu einer Coastmap umgewandelt. Diese enthält Daten, welche während der Kartenerstellung nicht angezeigt, trotzdem aber gesammelt werden. Im Falle einer Coastmap wird nicht nur betrachtet, ob ein Punkt besetzt oder frei ist, sondern auch, wie schwierig es ist, gewisses Terrain zu überqueren. Zwar sind diese Daten in einer ebenen Fläche relativ irrelevant, vor allem aber bei hügeligem Boden sind diese interessanter, weshalb ich dies hier erwähnen wollte. Dies weiteren erhalten Hindernisse einen Radius, welcher von dem Roboter tendenziell vermieden werden sollte, da Kollisionen, u.a. aufgrund der Objektform oder ungenauer Messdaten möglich sein Könnte.

## 4.5 Positionierung auf einer Karte

Zwar hat der Roboter nun eine Karte seiner Umgebung, jedoch fehlt dem Roboter noch sein genauer Standpunkt, da er sich auf irgendeinem Punkt in der Karte befinden könnte. Um nun den genauen Standpunkt zu finden, nutzt man die Monte-Carlo-Lokalisation. Dafür wird die Karte der Umgebung, sowie die Sensordaten, benötigt. Anfangs ist es gleich wahrscheinlich, dass der Roboter sich in jeglicher Stelle auf der Karte befindet. Nun werden die gemessenen Sensorpunkte und die Karte der Umgebung übereinander gelegt. Stimmen Hindernisse, welche durch die Karte in der Nähe sind, sowie gemessene Sensordaten der realen Welt überein, so ist es wahrscheinlich, dass sich der Roboter an dieser Stelle in der Karte befindet. Misst der Turtlebot beispielsweise einen Runden Gegenstand vor sich und in einer Position auf der Karte ist in gleicher Distanz eine Runde Säule vor dem Roboter, so ist es wahrscheinlich, dass der Roboter sich in dieser Position befindet. In der beigefügten Abbildung, wird das vergleichen von Sensordaten und Karte deutlich.

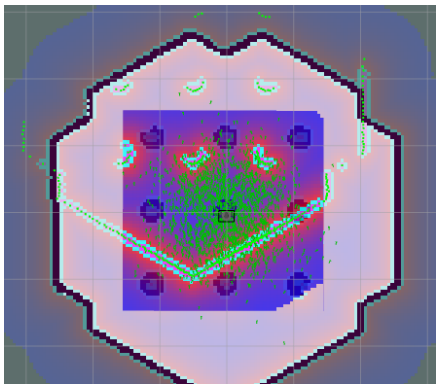

**Abbildung 17:** Coastmap mit Sensordaten

In der Abbildung wird deutlich, dass der Roboter auf der gegebenen Karte (Hintergrund) mittig platziert wird. Gleichzeitig misst der LiDAR-Sensor auch Hindernisse, welche wiederum eine neue Karte der Umgebung bilden. Diese Hindernisse sind Hell im Vordergrund zu sehen. Man erkennt, dass diese nicht übereinander liegen und der Roboter deshalb noch nicht korrekt positioniert ist. In Wirklichkeit werden Daten aus der unteren linken Ecke der Karte gemessen. Dadurch werden Positionen in dieser Umgebung sehr wahrscheinlich. Damit der Roboter bei vielen ähnlichen Stellen, was hier nicht der Fall ist, trotzdem korrekt positioniert ist, wird der Roboter zwischen verschiedenen LiDAR-Messungen bewegt. Während der Bewegungen wird errechnet, wo der Roboter sich, ausgehend von den Ursprünglichen Vermutungen der Position, befindet. Stimmen auch die neuen Messungen mit der Karte überein, so wird eine Position immer wahrscheinlich, bis der Roboter schlussendlich, sicher positioniert werden kann, da Sensordaten und Karte übereinstimmen. im Falle anderer Roboter können neben LiDAR-Sensoren auch andere Sensordaten verwendet werden.

## 4.6 Die Move-Base

Um den Roboter nun zu einem Ziel zu navigieren, wird das move_base-Package verwendet. Dieses steuert die Navigation des Roboters, sowie den Pfad, welchem der Roboter folgen soll. Es besteht aus mehreren Nodes (in der Abbildung oval dargestellt). Dazu abonniert diese Nodes verschiedene Topics. Dazu gehören Sensordaten (z.B. sensor_msgs/LaserScan für die Daten des LiDAR-Sensors), Odometriedaten und eine Karte. Die Karte wird durch einen Kartenserver bereitgestellt, welcher, als Node, diese Karte veröffentlicht, sodass jegliche Programme darauf zugreifen können. Dafür muss die Karte jedoch als Bild vorliegen, weshalb die Karte nach vollendeter Kartierung des Labyrinth auf einem Datenträger, meist die Festplatte eines über Wifi mit dem Turtlebot verbundenen Computers, gespeichert werden muss. Die gespeicherte Karte kann daraufhin von dem Karten-Server geöffnet werden. Dieses System sorgt gleichzeitig dafür, dass man eine Umgebung zur Navigation nicht notwendigerweise erneut kartieren muss, sollte schon eine Karte existieren. Im Anwendungsfall ist dies auch praktisch. Staubsageroboter müssen daher beispielsweise nicht jedes mal erneut eine komplette Karte eines Hauses erstellen.

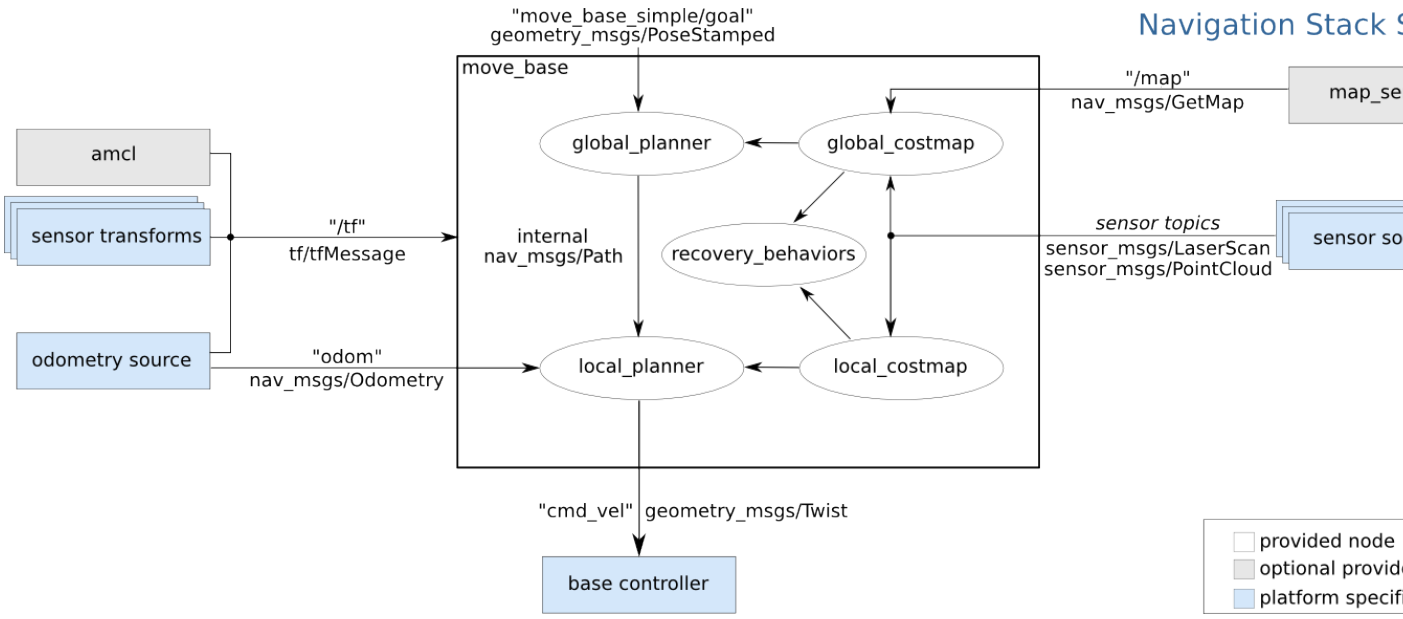


**Abbildung 18:** Das move_base-Package [0]

Nachdem der Roboter nun auf der durch den Kartenserver bereitgestellten Karte verordnet ist, muss dem Roboter ein Navigationsziel gegeben werden. Dazu gibt es mehrere Möglichkeiten. Das Programm Rviz ermöglicht es, wie bereits erwähnt, die Karte zu visualisieren. Gleichzeitig hat dies den Nebeneffekt, dass man durch das Programm bestimmen kann, welche Koordinaten ein Punkt auf der Karte hat, indem man die Maus auf diesen Bewegt. Rviz ermöglicht es sogar, direkt im Programm ein Navigationsziel festzulegen, was jedoch teilweise von dem Roboter nicht akzeptiert wird, weshalb ich ein Skript nutze, um die Koordinaten des Zielpunktes, welche ich in Rviz ablesen kann, an das move_base-Package zu übertragen. Dafür wird folgendes Skript verwendet, in welchem der Roboter sich beispielhaft zu dem Punkt $P(3|2)$ bewegen soll:

```
01 import actionlib
02 import rospy
03 from move_base_msgs.msg import MoveBaseAction,
   MoveBaseGoal, MoveBaseFeedback, MoveBaseResult

04 rospy.init_node("nav_goal_sender")

05 client = actionlib.SimpleActionClient("/move_base",
   MoveBaseAction)
06 client.wait_for_server()

07 goal = MoveBaseGoal()
08 goal.target_pose.header.frame_id = 'map'
09 goal.target_pose.pose.position.x = 3
10 goal.target_pose.pose.position.y = 2
11 goal.target_pose.pose.orientation.z = 0
12 goal.target_pose.pose.orientation.w = 0.5

13 client.send_goal(goal)
14 client.wait_for_result()
```

Nun erläutere ich das Programm kurz Stück für Stück. Mit Hilfe der **import**-Befehle wer-

den Bibliotheken eingebunden (Z. 1-3), welche Funktionalitäten für das Programm beinhalten, welche benötigt werden. Dazu zählt z.B: **rospy** (Z.2), welches eine Interaktion mit dem ROS-Betriebssystem ermöglicht, sowie verschiedene Klassen[4], welche mit dem move_base-Package zusammenhängen (Z.3).

Daraufhin wird eine Node-initiiert, deren einzige Funktion ist, ein Navigatinsziel auszusenden. Die Node erhält den Namen "*nav_goal_sender*"(Z.4).

In den nächsten zwei Zeilen wird ein "Action-Client" initiiert. Dies ist eine Klasse, welche es ermöglicht, eine Action an einen Service auszusenden. Diese Art des Datenaustausches wurde in Abschnitt 1.3.2- "Services"ausgeführt. Die Initiierung bedeutet vereinfacht gesagt, dass alles vorbereitet wird, sodass ein Befehl an den Service des move_base-Packetes gesendet werden kann (Z.5f.).

In den nachfolgenden sechs Zeilen (Z.7-12) werden die Parameter gesetzt, welche die Position, zu welcher Navigiert werden soll, bestimmen. Hier ist die z-Koordinate irrelevant, da der Roboter sich nicht nach oben/unten bewegen kann. Die x Koordinate (Z.9) wird gleich drei gesetzt. Die y-Koordinate gleich 2 (Z.10). In Zeile 11 wird die Richtung, in welche der Roboter am Ende schauen soll, festgesetzt.

In Zeile 13 wird eine Action, also der Befehl mit den eingestellten Parametern, hier sind dies die Koordinaten, an den move_base Service geschickt, welcher daraufhin zu diesem Ziel navigiert.

## Literatur


[0] Roboticsbackend. "rqt-graph Ausschnitt, Ein vereinfachter Graph, mit RQT-Graph visualisiert". Deutsch. (), Adresse: `https://roboticsbackend.com/wp-content/uploads/2019/07/rqt_graph_two_nodes_one_topic.png`.

[0] Robotis. "Turtlebot 3, ROBOTIS TurtleBot 3 Burger Plattform Roboter INTL 901-0118-200". Deutsch. (), Adresse: `https://www.zerodna.de/media/image/product/6437/md/robotis-turtlebot-3-burger-plattform-roboter-intl-901-0118-200.jpg`.

[0] OpenRobotics. "move-base-Node, Überblick über die Funktion der move-base-Node". English. (), Adresse: `http://wiki.ros.org/navigation/Tutorials/RobotSetup?action=AttachFile&do=view&target=overview.png`.


[4] **Programmierung: Ansammmlung von Funktionen (eines Aufgabenbereiches)**.