

Entwicklung einer Wegpunktnavigation auf der Roboterplattform TurtleBot3

Hausarbeit im Modul Robotik und Embedded Systems

VON

CHIARA TERESA GRUß, ROBIN SCHEEL, JAN-LUCA REGENHARDT

IM STUDIENGANG ELEKTRO- UND INFORMATIONSTECHNIK (MA)

AN DER

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFT UND KUNST
FAKULTÄT INGENIEURWISSENSCHAFTEN UND GESUNDHEIT



PRÜFER: PROF. DR. RER. NAT. THOMAS LINKUGEL

07. JANUAR 2021

Inhaltsverzeichnis

1 Einleitung	1
1.1 Aufbau der Arbeit	2
2 Technische Grundlagen	3
2.1 Robot Operating System	3
2.2 TurtleBot3	3
2.3 Simulationsumgebung Gazebo	5
2.3.1 Modellbildung	6
2.4 Visualisierungstool RViz	7
2.4.1 Mapping	8
2.5 Global- und Local-Planner	10
2.5.1 Pfadplanung mit Roadmap	11
2.5.2 Pfadplanung mit Zellzerlegung	12
2.5.3 Pfadplanung mit Potentialfeld	13
2.6 Koordinatensystem und Wegpunkt	14
2.7 Topic und Nodes im Robot State Maschine	15
3 Programmaufbau	19
3.1 Initialisierung	19
3.2 Erläuterung der Funktionen	19
3.2.1 Wegpunkte einlesen	20
3.2.2 Wegpunkte publishen	20
3.2.3 Wegpunktstatus	21
3.2.4 Aktion am Wegpunkt	22
4 Fazit	23
Abbildungsverzeichnis	A
Tabellenverzeichnis	C
Quelltextverzeichnis	C

Literaturverzeichnis**E****5 Anhang****G**

5.1 Aufteilung der Arbeit	G
5.2 Programmcode	H
5.3 Umgebungs-Karte mit SLAM	M

1 Einleitung

Die Entwicklung neuer Konzepte und Technologien in der Robotik ist in den letzten Jahren rasant gewachsen und hat ein großes Spektrum an Anwendungsbereichen. Die erweiterten Fähigkeiten hinsichtlich der Umgebungswahrnehmung lassen es zu, dass sensitive Roboter äußere Einflüsse Wahrnehmen. Sie können sich frei in einem Raum bewegen und autonom navigieren. Zu den Beispielen für diese Hausarbeit zählen u.a. die Automobilbranche mit der Verbesserung des autonomen Fahrens oder dem Vernetzen von Fahrzeugen untereinander, die Industrie und die Haushalte mit den Staubsaug- und Mährobotern [Lin20]. Bei der Weiterentwicklung der Technologien soll es in erster Linie nicht darum gehen, den Menschen zu ersetzen, eher soll eine bessere Interaktion zwischen dem Menschen und dem Roboter möglich sein. Dies kann jedoch nur durch softwareseitige Funktionen und Sensoren an Board des Roboters erreicht werden. Durch diese Sensoren erfassst der Roboter seine dynamische Umgebung und reagiert nach vorher festgelegten Standards. All dies muss vorher simulativ getestet werden, um festzulegen, ob das System allen Sicherheitsstandards genügt.

Eine Möglichkeit zu sehen, wie sicher eine vom TurtleBot3 gewählte Trajektorie ist, ist ein Programm zu schreiben, welches dem Versuchsobjekt Wegpunkte über gibt, um zu kontrollieren, ob dieses feste Hindernisse umfährt, als auch die Wegpunkte ohne Umwege erreicht.

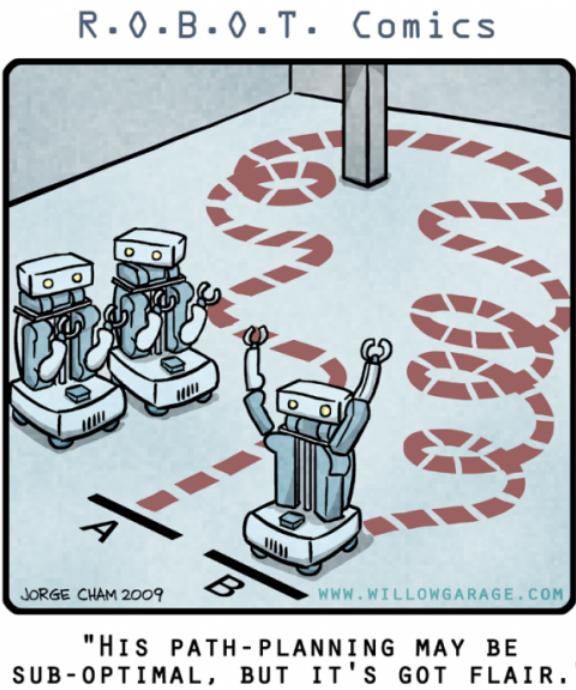


Abbildung 1.1: Path-Planning [Cha09]

In diesem *Projekt 3* werden dem TurtleBot3 Wegpunkte übergeben. Diese fährt er ab und gibt nach Erreichen des jeweiligen Punktes eine Bestätigung zurück. Um die Trajektorie zu bestimmen wird eine realistische Umgebung mit der Simulationsumgebung Gazebo erstellt, in der sich der TurtleBot3 bewegt.

1.1 Aufbau der Arbeit

Der Abschnitt 1.1 gibt einen Überblick über die gesamte Hausarbeit. Er erläutert die Teilaufgaben wie die Navigation durch Wegpunkte erreicht wird.

Das Kapitel 2 setzt sich mit den technischen Grundlagen wie der Simulationsumgebung Gazebo, dem mobilen Roboter TurtleBot3, der Pfadplannung und den Topics auseinander.

Das darauf folgende Kapitel 3 erklärt das Wegpunktprogramm, welches im Rahmen des Projektes programmiert wird.

Den Abschluss der Hausarbeit bildet das Kapitel 4 mit dem Fazit.

2 Technische Grundlagen

Das Kapitel 2 fasst die technischen Grundlagen zusammen mit dem sich im Rahmen des Projektes beschäftigt wird.

In Abschnitt 2.1 wird kurz das verwendete Betriebssystem vorgestellt. Die Abschnitte 2.3 und 2.3.1 behandeln die Simulationsumgebung Gazebo und das Modell, in dem sich der mobile Roboter bewegen soll. In Abschnitt 2.2 wird der genutzte mobile Roboter TurtleBot3 und seine implementierte Hardware thematisiert.

Den Global- und Local-Planner differenziert der Abschnitt 2.5 genauer.

Abgeschlossen wird das Kapitel 2 mit den genutzten Topics, wie *move_base_simple/goal*, in Abschnitt 2.7.

2.1 Robot Operating System

Das *Roboter Operating System* (ROS) wurde 2007 im Rahmen eines Projekts an der Standort Artificial Intelligence Laboratory entwickelt. Erst seit 2012 wurde ROS, wie auch Gazebo, durch die *Open Source Robotics Foundation* (OSRF) weiterentwickelt und steht als Open Source Software zur Verfügung. ROS stellt ein umfangreiches Betriebssystem zur Programmierung, Kommunikation und Steuerung von Robotern [Foo13].

2.2 TurtleBot3

Der TurtleBot3 ist ein fahrender, mobiler Roboter von u.a. Open Robotics, ROBOTIS und vielen weiteren Kooperationspartnern aus Industrie und Forschung. Das Modell "Burger" (Abbildung 2.1), welches in dieser Arbeit genutzt wird, verfügt über einen 360° LiDAR (engl. *Light Detection And Ranging*) Laserscanner, leistungsstarke XL430 Schrittmotoren der Firma Dynamixel und einen Raspberry Pi3 B+ Einplatinencomputer.

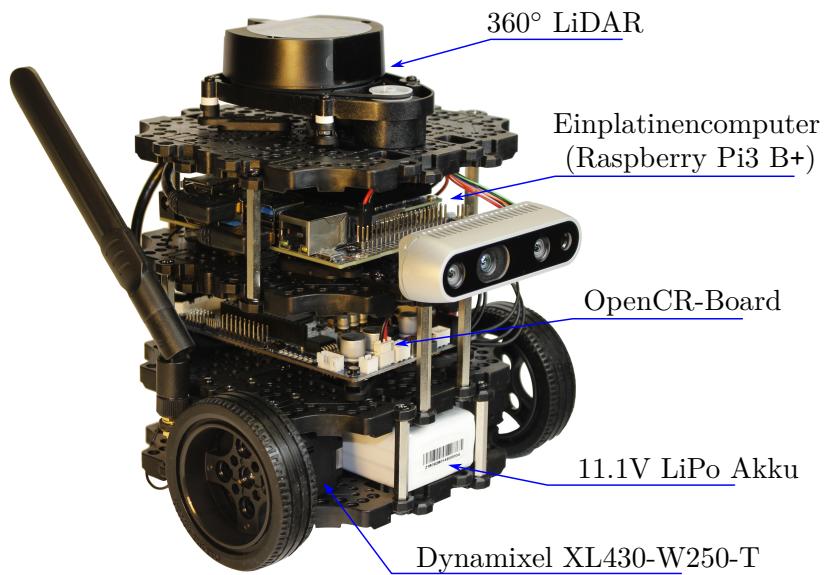


Abbildung 2.1: TurtleBot3 Burger

Tabelle 2.1: Technische Daten zum TurtleBot3 [Rob20]

Kenndaten	Wert
max. transl. Geschwindigkeit	0.22 m/s
max. rotatorische Geschwindigkeit	2.84 rad/s (162.72 deg/s)
max. Tragkraft	15 kg (1 kg Eigengewicht)
Maße (L x B x H)	138 mm x 178 mm x 192 mm
Nutzungsdauer	2 h 30 m
Mikroprozessor	32-bit ARM STM32F746ZGT6 mit Floating Point Unit
Sensorik	LDS-01 LiDAR
	MPU9250 (ICM-20648 ^{new}) IMU ¹
Akkumulator	Lithium Polymer 11.1 V 1800 mA 5C

¹Inertiale Messeinheit (engl. *Inertial Measurement Unit*)

2.3 Simulationsumgebung Gazebo

Gazebo² ist eine Simulationsumgebung, dessen Entwicklung 2002 an der *University of Southern California* begonnen hat. Obwohl Gazebo heutzutage von den meisten Nutzern für *Indoor* - Umgebungen genutzt wird, war sie zunächst für *Outdoor* - Umgebungen von Dr. Andrew Howard und seinem Studenten Nate Koenig konzipiert. Daher auch der Name "Gazebo", was einen Pavillon im Freien (draußen) bezeichnet. Seit 2012 wird Gazebo unter der OSRF weiterentwickelt.

Die Struktur von Gazebo bildet zum einen der Gazebo Server mit der Physik- und Sensor-Engine. Um die Umgebung zum anderen visuell darzustellen und grafisch bearbeiten zu können, verbindet sich der sogenannte *Graphical Client* mit dem Gazebo Server. Bei einem normalen Start geschieht dies jedoch automatisch mit folgendem Befehl:

```
$ gazebo worlds/empty.world
```

Mit dem folgenden ROSLAUNCH-Befehl ist es möglich, den TurtleBot3 in der Umgebung zu simulieren:

```
$ roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch
```

Die Abbildung 2.2 zeigt die Gazebo Umgebung in der Version 9.0.0 unter Ubuntu 18.04 LTS. Das **Feld A** gibt eine Übersicht über die Bestandteile der simulierten Welt. Dazu gehören unter anderem die Physik, das GUI (engl. *Graphical User Interface*), verwendete Modelle und die Beleuchtung. Diese Eigenschaften können hier angesehen und bearbeitet werden.

Unter dem Reiter "*insert*" in **Feld B** verbirgt sich die Modelldatenbank, aus welcher verschiedenste Modelle in die Welt hinzugefügt werden können. Dazu zählen einfache Geometrien, komplexe und selbst erstellte Modelle sowie komplette Topografische Karten.

Die grafische Visualisierung der Simulation gibt das **Feld C** wieder. Dort ist der TurtleBot3 und das Umgebungssmodell abgebildet. Über die Maus lässt sich die Ausrichtung der Kamera optimal einstellen, um einen guten Überblick über den aktuellen Stand der Simulation zu erhalten.

Über **Feld D** lässt sich die Simulation pausieren und der *Real Time Factor* (Simulationsgenauigkeit zur Echtzeitsimulation) einstellen. Das Feld für die Bilder pro Sekunde (engl. *Frames Per Second FPS*) ist ein Indikator dafür, wie anspruchsvoll eine Simulation für den Computer ist.

²<http://gazebosim.org/>

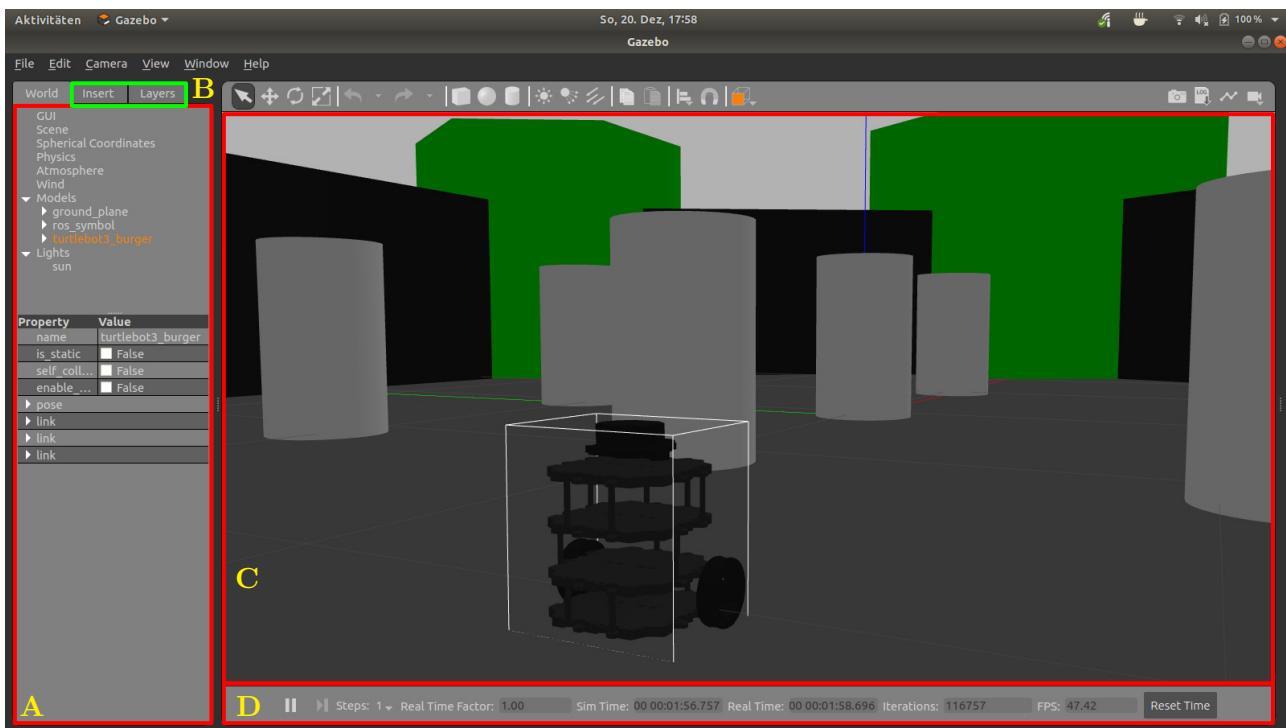


Abbildung 2.2: Gazebo Simulationsumgebung

2.3.1 Modellbildung

Als realistische Umgebung wird das erste Stockwerk des C-Gebäudes an der Hochschule für angewandte Wissenschaft und Kunst (HAWK) Fakultät [i] nach modelliert. Diese Umgebung soll der TurtleBot3 in Wegpunkten abfahren. Mit der 2D/3D-CAD (engl. *Computer-Aided Design*) Software Solid Edge³ von Siemens PLM Software in der Studierenden-Version wird das Stockwerk als Modell aus einzelnen Blöcken und Einrichtungsobjekten als .asm (Assembly) zusammengebaut und als .stl (engl. *Standard Triangle Language*) abgespeichert. Dieses stl-Format ermöglicht als Standardschnittstelle unter CAD-Systemen das Einbinden von Objekten in Gazebo. Aus dem Modell wird dabei ein Masche (engl. *Mesh*) aus vielen Dreiecken erzeugt. Diese kann nun im *Modell-Builder* der Gazebo-Umgebung eingebunden werden. Für eine realitätsnahe Umgebung sind Flure und Räume mit Tischen, Schränken und Boxen versehen. Die Abbildung 2.3 zeigt ein realistisches Abbild dieses Modells.

³<https://solidedge.siemens.com/de/>

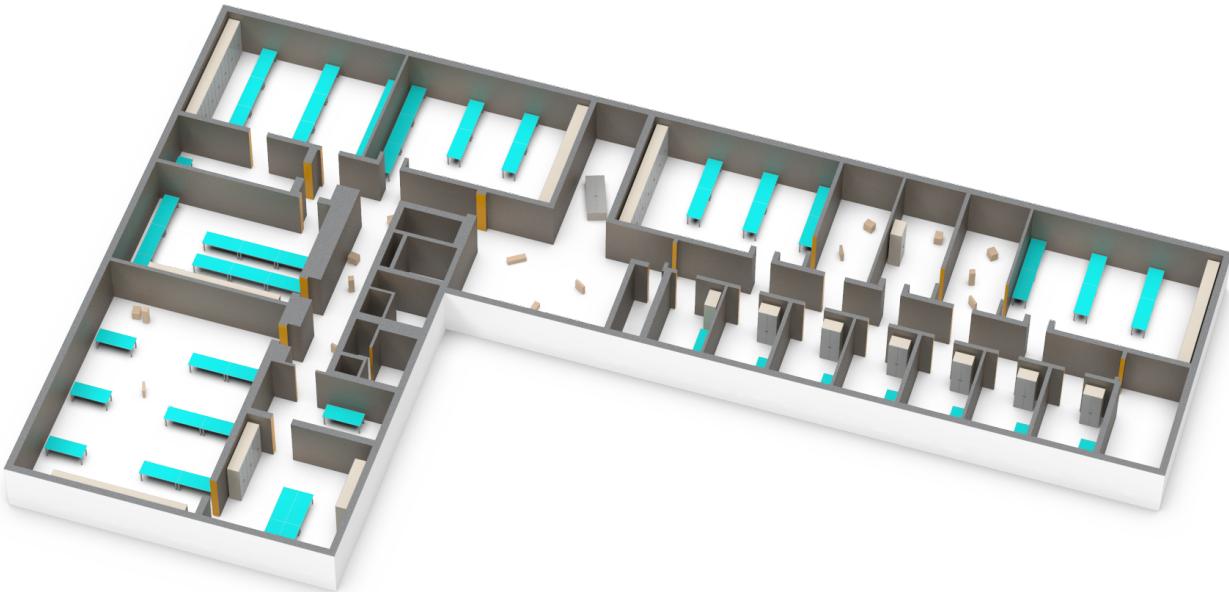


Abbildung 2.3: Modell HAWK Fak [I], Gebäude C, 1. Obergeschoss

Folgende Dateien aus dem GitHub-Repository⁴ müssen hinzugefügt werden:

1. **HAWKC_Ebene1.world** und **HAWKC_Ebene1.stl** abspeichern in
`~/catkin_ws/src/turtlebot3_simulations/turtlebot3_gazebo/worlds`
2. **turtlebot3_HAWK1.launch** abspeichern in
`~/catkin_ws/src/turtlebot3_simulations/turtlebot3_gazebo/launch`

2.4 Visualisierungstool RViz

Die visuelle Überwachung des TurtleBot3 erfolgt durch das Programm RViz (Robot Visualization). Hiermit kann sowohl auf physische als auch simulierte Roboter zugegriffen werden. Durch die visuelle Darstellung des Roboters mit den zusätzlichen Informationen wie Sensordaten, Datenströme und die erstellte Umgebung, können Aktionen und Reaktionen des TurtleBot3 aktiv beobachtet werden.

Mit dem folgenden ROS-Befehl wird in Abbildung 2.4 die Visualisierung der Umgebung gezeigt.

```
$ rosrun turtlebot3_gazebo turtlebot3_gazebo_rviz.launch
```

⁴<https://github.com/regenhardthawk/Robotik-Projekt-WiSe2020>

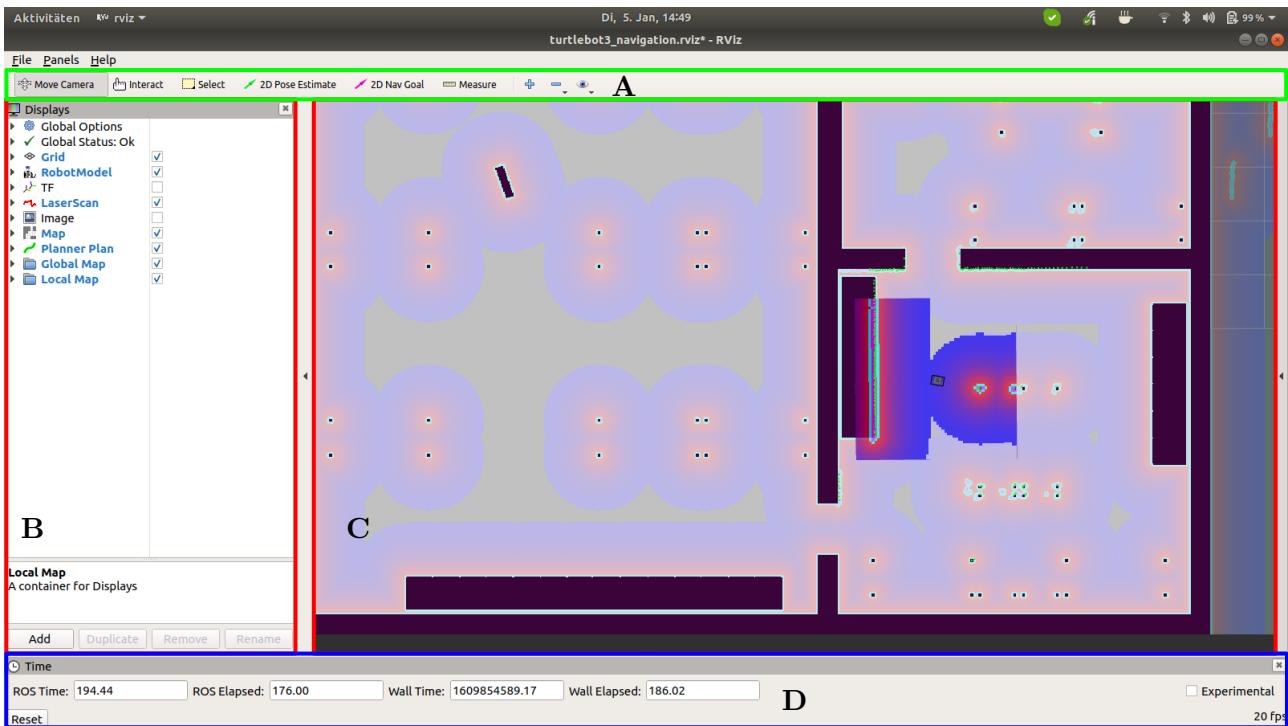


Abbildung 2.4: Visualisierungstool RViz

Die Abbildung 2.4 gibt einen Überblick über die RViz Benutzeroberfläche. Im **Feld A** ist die Kamerasteuerung, ein Messlineal und Schaltflächen für die Navigation untergebracht. *2D Pose Estimate* lässt den Benutzer die Positionen in der zuvor hinterlegten Karte mit der Simulations-Umgebung übereinander legen. Dies ist nach jedem Programmstart nötig, damit der Global-Planner (Abschnitt 2.5) einen Pfad berechnen kann. Über *2D Nav Goal* wird dem ROS-Master ein Wegpunkt auf dem Topic `/move_base_simple/goal` übergeben und angefahren.

Im **Feld B** werden einzelne Topics abonniert. Über verschiedene Parameter können diese nun angepasst und im **Feld C**, der Visualisierungs-Umgebung, grafisch angezeigt werden. Ebenfalls in **Feld C** sind der Global- und Local-Planner zu sehen, auf welche in Abschnitt 2.5 genauer eingegangen wird. Vergleichbar zur Gazebo-Umgebung in Abbildung 2.2 zeigt das **Feld D** die aktuelle ROS-Zeit und den FPS-Zähler.

2.4.1 Mapping

Damit der TurtleBot3 über RViz die Pfadplanung ausführen kann, benötigt dieser eine Karte seiner Umgebung. Die erste Möglichkeit eine solche Karte zu erstellen ist über einen SLAM-Algorithmus

(engl. *Simultaneous Localization and Mapping Algorithm*), wobei aus der Gazebo-Umgebung iterativ eine Karte zusammengebaut wird. Das Ergebnis dieses sogenannten *gmapping*-Algorithmus⁵ für das Modell ist im Anhang unter Abschnitt 5.3 zu sehen.

Aufgrund von Sensorfehlern und Überlagerungen ist diese Karte jedoch zu ungenau, weshalb eine andere Herangehensweise genutzt werden muss. Dafür wird das Modell aus Solid Edge mit einem planaren Schnitt 70 cm über dem Boden in eine technische Zeichnung überführt. Dabei muss auf die richtige Skalierung der Abbildung geachtet werden. Die resultierende Karte wird mit dem Grafikprogramm GIMP (engl. *GNU Image Manipulation Program*) zugeschnitten und Kanten mit Graustufen-Verlauf auf Schwarz verbessert. Somit wird die erzeugte *Collision Map* viel genauer.

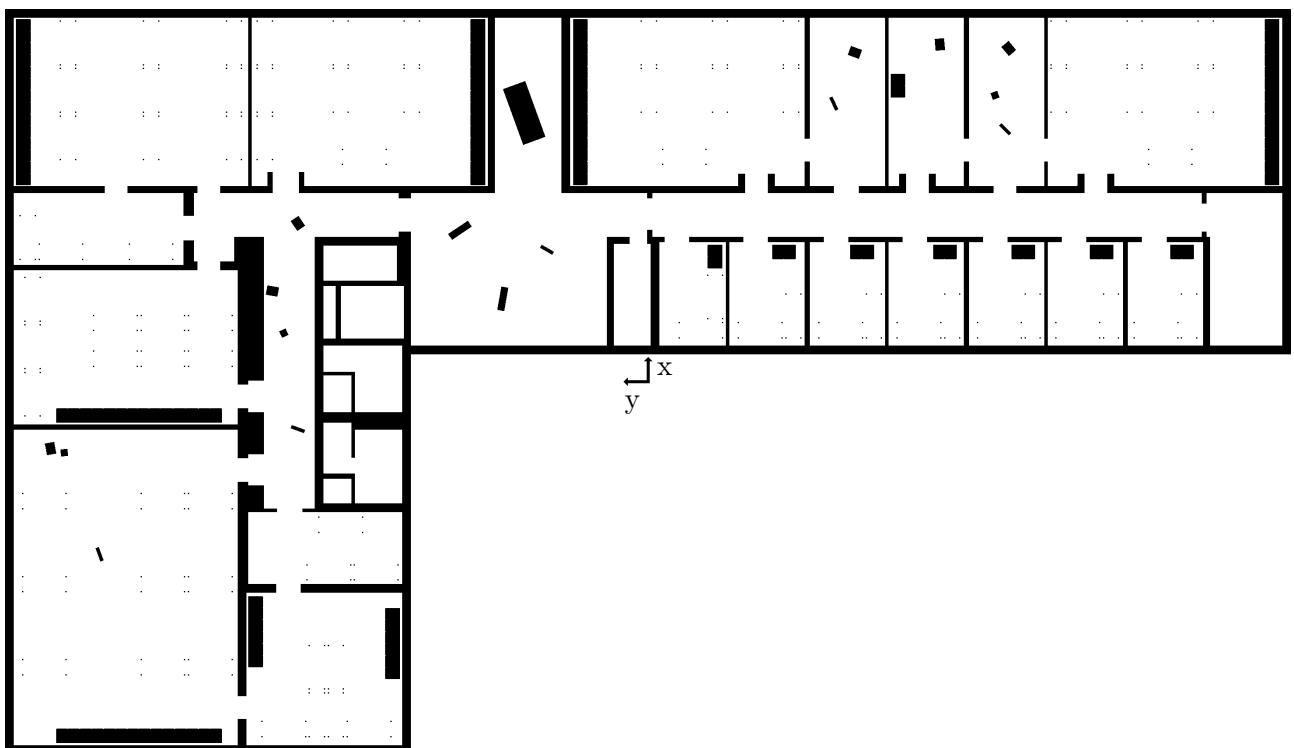


Abbildung 2.5: Map HAWK_Modell

Die fertige Karte in Abbildung 2.5 hat folgende Eigenschaften:

Abmessung: 2940 x 1700 pixel

Maßstab⁶: $0.01845 \frac{m}{px}$

⁵<https://openslam-org.github.io/gmapping.html>

⁶Zur Verfügung gestellt von Prof. Dr. rer. nat. Thomas Linkugel, *Autonomous Mobile Robotics Lab (AMRL)*, HAWK Göttingen

Die Karte im PNG-Format (engl. *Portable Network Graphics*) wird in ein PGM-Format (engl. *Portable Graymap*) um konvertiert, damit sie in ROS verwendet werden kann. Zu dieser gehört eine Datei im YAML-Format (engl. *Markup Language*) mit folgendem Inhalt:

```

1      image: /home/<username>/Fak_I_Map2.pgm
2      resolution: 0.018450
3      origin: [-15.682500, -27.121500, 0.000000]
4      negate: 0
5      occupied_thresh: 0.65
6      free_thresh: 0.196

```

Um zu erreichen, dass RViz beim Programmstart mit der Karte startet, müssen die PGM- und YAML-Datei im Heimverzeichnis (`~/home/<username>`) liegen, und folgender Befehl ausgeführt werden:

```
$ roslaunch turtlebot3_navigation turtlebot3_navigation.launch map_file:=$HOME/Fak_I_Map2.yaml
```

2.5 Global- und Local-Planner

Um einen TurtleBot3 in einer 2D-Ebene zu navigieren, muss eine optimale Trajektorie entwickelt werden, die den Start- und Zielpunkt durch einen kollisionsfreien Pfad verbindet. Die optimale Route ist nicht immer die kürzeste, sondern berücksichtigt alle Randfaktoren der Umgebung und des Fahrzeugs. Diese Generierung der Trajektorie wird mittels zwei Planner erstellt:

- Global-Planner
- Local-Planner

Um einen optimalen Pfad für den TurtleBot3 zu planen, dürfen die durch den Local - und Global- Planner visualisierten Hindernisse nicht außer Acht gelassen werden. Dies ist beispielhaft in Abbildung 2.4 abgebildet. Diese können auf unterschiedliche Weise durch Algorithmen in der Karte angezeigt werden [TBF05]. Die erste Möglichkeit ist, den Raum in einheitliche und quadratische Felder zu unterteilen (*Gitter*). Für ein jeweiliges Feld gibt es zwei Zustände. Ein dunkel markiertes Feld zeigt ein Hindernis an, wohingegen ein nicht markiertes Feld (helles Feld) eine mögliche Route aufzeigt. Das Positive an dieser Darstellung sind die klaren Kanten der definierten Hindernisse. Die zweite Möglichkeit ist die

Polyedrische Repräsentation bei der anhand eines Polyeder das Hindernis modelliert wird. So können Hindernisse präziser erstellt werden, so dass im Umkehrschluss eine bessere Pfadplanung erzeugt wird [Mic13].

Im Folgenden wird im Einzelnen auf die beiden Planner eingegangen.

Die Aufgabe des Global-Planner ist die Berechnung der optimalen Route. Für diese Planung wird eine Karte der Umgebung (Abbildung 2.5) erstellt, die über das Topic `/map` in ROS geladen wird. Mittels dieser Karte erzeugt ein Algorithmus des Global-Planners mit unterschiedlichen Methoden den optimalen Pfad. Es gibt drei Methoden durch einen Algorithmus den Pfad zu ermitteln [TBF05].

- Roadmap
- Zellzerlegung
- Potenzialfeld

2.5.1 Pfadplanung mit Roadmap

Bei diesem Pfadplanungsverfahren werden freie Punkte miteinander verbunden. Hierdurch entsteht ein Netz von Pfaden. Dieses Netz wird als *Roadmap* bezeichnet. Für die Erstellung eines Pfades werden die Linien vom Startpunkt bis zum Zielpunkt mittels der vorgefertigten Netzlinien verbunden. Auch hier gibt es unterschiedliche Ansätze diese Netzstruktur zu erzeugen. Beim Sichtbarkeitsgraph (Abbildung 2.6a) werden alle Hindernisse n-eckig modelliert, so dass der TurtleBot3 einen kollisionsfreien Pfad erstellen kann. Für den Pfad werden nur die Strecken erfasst, welche um die Hindernisse herumführen [Lat90].

Das Voronoi-Diagramm (Abbildung 2.6b) erzeugt in der Karte viele Punkte um die Hindernisse herum. Für eine Pfadplanung werden diese dann zu einer vorgefertigten Route verbunden [Lat90].

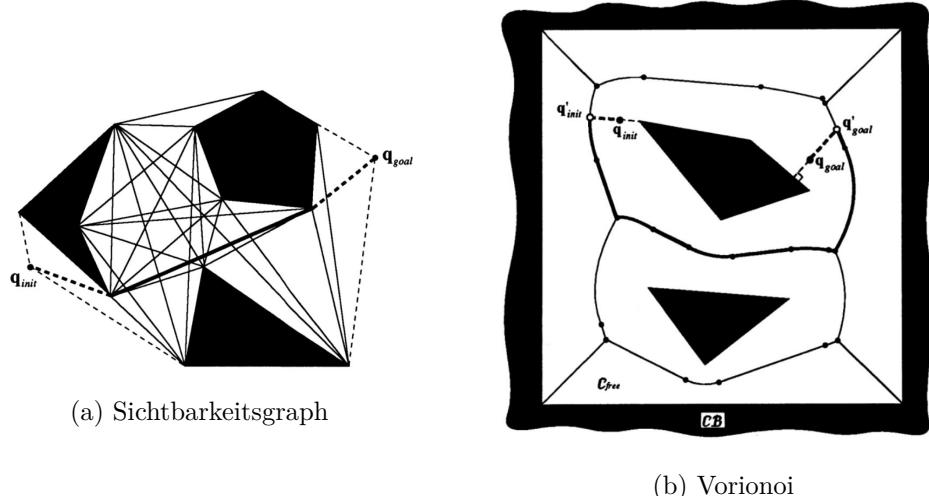


Abbildung 2.6: Roadmap-Pfadplanung [Lat90]

2.5.2 Pfadplanung mit Zellzerlegung

In dem *Zellzerlegungsverfahren* wird der Arbeitsraum segmentiert. Nach der Aufteilung der Karte in kleine und gleich große Zellen werden die benachbarten Segmente miteinander verbunden. Es entsteht ein sogenannter Knotengraph (engl. *connectivity graph*). In dieser Darstellung sind alle Knoten unter- und miteinander verbunden. Eine Möglichkeit aus dem Knotengraph einen Pfad zu erzeugen, erfolgt mit dem Dijkstra- Algorithmus oder dem A*-Algorithmus (Abbildung 2.7). Der Rechenaufwand für den Dijkstra-Algorithmus ist um einiges höher, denn dieser fragt jeden Weg einzeln ab. Der A*-Algorithmus verwendet eine Schätzfunktion (Heuristik), um zielgerichteter zu suchen und somit die Rechenzeit zu verringern. Dabei werden zuerst die Knoten untersucht, welche dem Ziel am nächsten sind [Lat90].

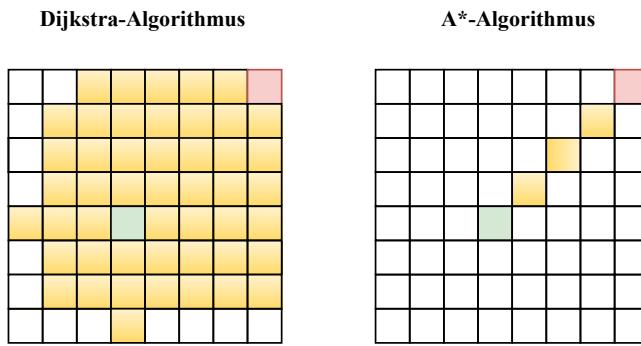


Abbildung 2.7: Vergleich Algorithmus

Bei diesen Algorithmen wird jedem Pfad zur nächsten Zelle eine Gewichtung zugewiesen. Durch das Iterationsverfahren wird der Pfad mit der geringsten Kostensumme berechnet.

2.5.3 Pfadplanung mit Potentialfeld

Bei der *Potenzialfeld* Pfadplanung erzeugt der TurtleBot3 ein künstliches Potenzialfeld (z.B ist dieser positiv geladen). Für die Planung einer Route wird das Ziel negativ gepolt, so dass der TurtleBot3 von diesem Ziel magnetisch angezogen wird. Der Raum hat ein somit gegengepoltes Potenzialfeld zu dem des Roboters. Die Hindernisse hingegen haben ein gleichgepoltes Potenzialfeld, was dazu führt, dass der Roboter von ihnen abgestoßen wird. So ist eine Lokalisierung der Hindernisse im Raum möglich und eine kollisionsfreie Route kann erstellt werden (Abbildung 2.8) [Lat90].

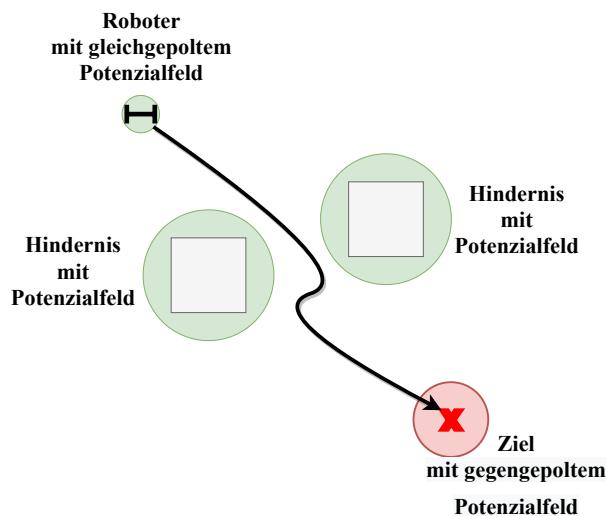


Abbildung 2.8: Pfadplanung Potentialfeld

Der durch den Global-Planner generierte Pfad wird nun in geeignete Wegpunkte durch den Local-Planner umgewandelt. Die Erstellung erfolgt auf der Grundlage der dynamischen Hindernisse und der Fahrzeugbeschränkungen . Dafür wird die Umgebung auf den unmittelbaren Bereich des TurtleBot3 reduziert. Dies ist notwendig, da die Sensoren die ganze Karte nicht ohne Erzeugung eines großen Rechenaufwandes erfassen können. Somit induziert der Local-Planner eine lokale Karte mit den globalen Wegpunkten. Hierfür stehen unterschiedliche Methoden der Trajektorien Bildung zur Verfügung. Alle Ansätze (z.B Splines Bogen oder Bezier Line) der Trajektorien Bildung erstellen Zwischenpunkte auf der schon erzeugten globalen Trajektorie [Mar+17].

2.6 Koordinatensystem und Wegpunkt

Bei dem Projekt 3 handelt es sich um einen Roboter, der zielorientiert agiert. Dabei stellt sich dieser, wie auch jeder Mensch, der sich von Punkt A nach B bewegen will, folgende Fragen:

1. Wie ist meine aktuelle Position?
2. Wo ist meine Zielposition?
3. Wie komme ich zu meiner Zielposition?

Dafür plant der TurtleBot3 einen Pfad wie im Abschnitt 2.5 beschrieben ohne mit Hindernissen zu kollidieren. Für eine Pfadplanung wird eine Karte (Abbildung 2.5) erstellt. Da der TurtleBot3 mittels kartesischer Koordinaten gesteuert wird (Abbildung 2.9), wird auch ein kartesisches Koordinatensystem in die Karte implementiert. Die Änderung der Position erfolgt durch die Übergabe von Wegpunkten in folgender Form.

$$WP = \begin{pmatrix} P_x \\ P_y \\ P_\Theta \end{pmatrix} \quad (2.1)$$

Eine Pose beschreibt die Position des TurtleBot3 in der Simulationsumgebung und dessen Orientierung. Mit ihr wird der Standort des TurtleBot3 bestehend aus x- und y- Koordinaten und der Orientierung in Grad (Θ Ausrichtung des TurtleBot3) übergeben.

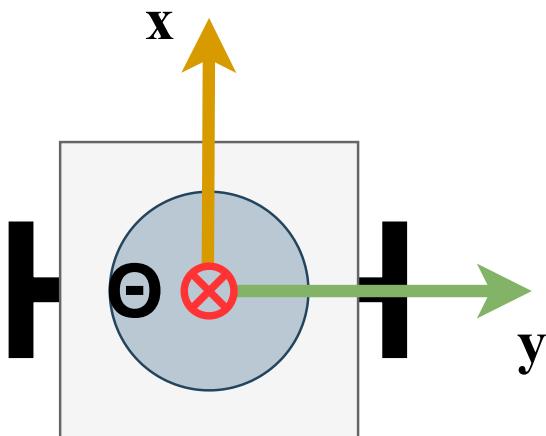


Abbildung 2.9: Roboter mit Koordinatensystem

Durch die Abfrage der Position kann eine relativ genaue Aussage über den Standort des Roboters in der Karte getroffen werden. Erst durch eine Transformation des Wegpunktes in Quaternion kann eine Aussage über die tatsächliche Ausrichtung des Roboters getroffen werden [FP14].

2.7 Topic und Nodes im Robot State Maschine

ROS ist ein quell-offenes Rahmenwerk und bietet eine Vielzahl an Bibliotheken (7,5 Mio. Zeilen an freier Software [NR20] Folie 3). Durch seine Infrastruktur, bestehend aus Subscribers und Publishers, welche miteinander kommunizieren, werden Informationen durch Netzwerke von Knoten (engl. *Nodes*) ausgetauscht. Ein Node bezeichne dabei die kleinste Einheit eines Prozessors, der in ROS läuft und empfohlen wird einen einzigen Knoten für jeden Zweck zu erstellen [Hab18].

Der ROS-Master steht dabei im Mittelpunkt und macht die Kommunikation erst möglich, indem er den Nodes die notwendigen Informationen zur Kommunikation liefert (TCP/IP). Der Master wird aufgerufen, sobald der Befehl `roscore` eingegeben wurde. Danach kann der Name von jedem Node aufgerufen werden und bei Bedarf Informationen abgerufen werden [Pyo+17].

Das Topic ist eine Themenbezeichnung und unter dessen werden Nachrichten ausgetauscht. Die Nodes (Knoten) sind Teil des Netzwerkes und abonnieren (engl. *subscribe*) oder veröffentlichen (engl. *publish*) Nachrichten unter einem Topic. [NR20].

Die Übertragung von Informationen kann über drei Arten erfolgen:

1. unidirektionale Übertragung/Empfang (beidseitiger Austausch)
2. bidirektional Anfrage-Antwort
3. Aktion die bidirektionale Nachricht Ziel/Ergebnis/Rückmeldung bereitstellt

Außerdem können Parameter, die in einem Node benutzt werden, von außerhalb des Knotens geändert werden, was wiederum eine weitere Art der Kommunikation darstellt [Pyo+17].

Die Robot State Machine (RSM) ist ein Paket, welches die grundlegenden Funktionen für eine Wegpunkt Navigation bietet. Es ist so aufgebaut, dass benutzerdefinierte Navigations-/Kartierungsverfahren genutzt werden können. Gesteuert über das Visualisierungstool RViz (Abschnitt 2.4) beinhaltet es u.a. das Node `move_base` [Mar19]. Dieses Node liefert Topic's für eine einfache Navigation und ist ein Base State der RSM. Das `move_base`-Paket implementiert die Aktion, dass der Turtlebot ein Ziel in der Welt bekommt und versucht dieses zu erreichen. Dabei verbindet der Knoten einen Global- und Lokal-Planner miteinander, um das Ziel zu erreichen. Abbildung 2.10 zeigt eine Übersicht über den `move_base`-Knoten und seine Interaktion mit anderen Komponenten [nic20].

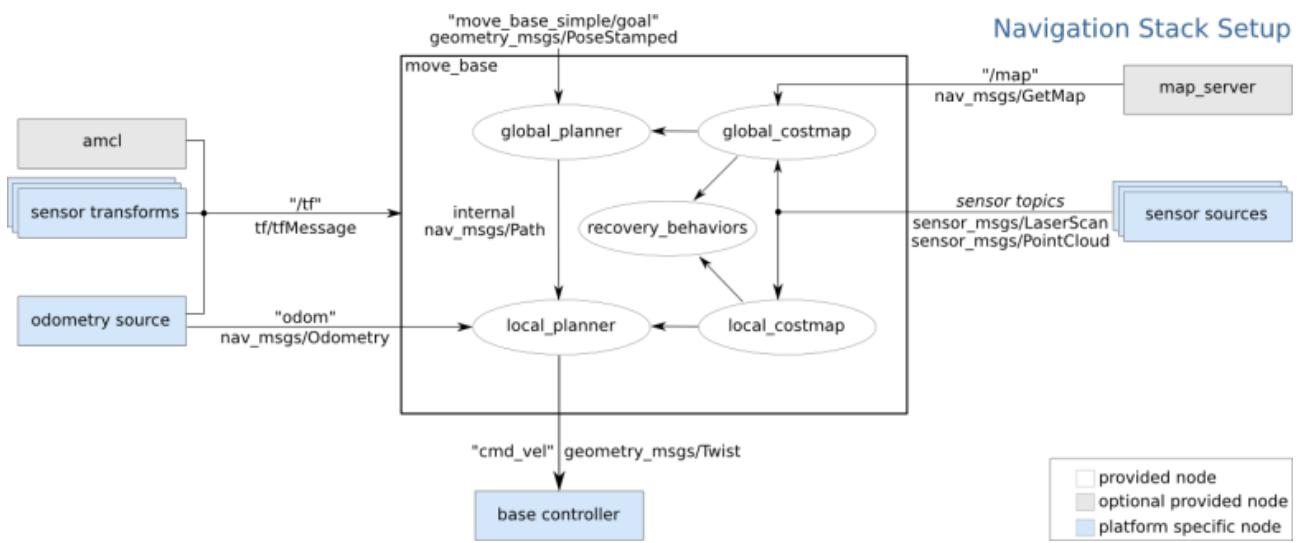


Abbildung 2.10: Navigation Stack Setup `move_base` [nic20]

Für die Wegpunkt-Navigation in dieser Arbeit wurde das Topic `/move_base_simple/goal` abonniert, welche das Ziel in Form einer `geometry_msgs/PoseStamped`-Nachricht sendet. Dieses Ziel soll dann vom TurtleBot3 angefahren werden. Über das veröffentlichte Topic `/move_base/result` (`move_base_msgs/MoveBaseActionResult`) kann abgefragt werden, ob das Ziel erreicht wurde. Aus der `MoveBaseActionResult` Message kann die Variable `Status` abgefragt werden, welche wiedergibt ob der TurtleBot3 sein Ziel erreicht hat. Wenn die Variable `Status` den Wert drei hat, bedeutet das die Aktion war erfolgreich und der Wegpunkt wurde erreicht. Durch Abfrage von `Status` kann nun eine Zielaktion implementiert werden.

3 Programmaufbau

In diesem Kapitel werden die einzelnen Funktionen des Wegpunkt-Programms erklärt. Begonnen wird in Abschnitt 3.1 mit der Initialisierung des TurtleBot3 und der Bereitstellung aller gebrauchten Bibliotheken aus ROS. In Abschnitt 3.2 und den Unterabschnitten wird auf das Einlesen und das Publizieren der Wegpunkte eingegangen. Außerdem wird eine Callback-Funktion beschrieben, welche abfragt ob das Ziel erreicht worden ist und weitere Schritte einleitet. Abgeschlossen wird das Kapitel 3 mit der visuellen Bestätigung bei Erreichen des Wegpunktes.

3.1 Initialisierung

Um C++ mit ROS-Funktionen zu vereinen müssen Header-Dateien von ROS bekannt gemacht werden. Dabei werden durch `#include "ros/ros.h"` die Hauptfunktionen implementiert, welche u.a. für Subscriber, Publisher oder den ROS eigenen Timer gebraucht werden. Außerdem wurde der Header `"geometry_msgs/PoseStamped.h"` für Positionsangaben und deren Ausgabe verwendet. Für die Kommunikation mit dem o.g. Node `move_base` wurde der `move_base_msgs.h`-Header benötigt. Die Nachrichten vom `move_base` werden von der `MoveBase.action` automatisch generiert. Die "action" meint hierbei eine weitere Methode der Nachrichtenkommunikation, welche verwendet wird, wenn nach Empfang einer Anforderung eine längere Zeit für die Antwort benötigt wird und währenddessen Zwischenwerte erforderlich sind um auf das Endergebnis zu kommen [Pyo+17, S. 44]. Zusätzlich wird dann noch die `MoveBaseActionResult.h`-Headerdatei benötigt, welche die Variable `Status` enthält und im späteren dazu genutzt wird festzustellen ob das Ziel erreicht wurde. Des Weiteren werden noch Bibliotheken aus C benötigt. Zum einen `<tf/tf.h>` zur Transformation von QuaternionMsgFromYaw, `<fstream>` für das Einlesen von filestreams, `<vector>` für den Umgang mit Vektoren, sowie `<iostream>` für den In- und Output von Streams und `<string>` für Zeichenketten.

3.2 Erläuterung der Funktionen

Begonnen wurde damit in dem eigenen Namespace `rob_proj4` im Header eine Klasse `waypoint_nav` zu erstellen, um die Funktion und Variablen bekannt zu machen. Es wurden zwei `Nodehandler` und

jeweils *Publisher* und *Subscriber* implementiert, welche Daten an den TurtleBot3 senden oder erhalten. *Nodehandler* sind die Schnittstelle von `roscpp` zum Anlegen von *Subscribers*, *Publishern* und mehr ([Qui+20]).

3.2.1 Wegpunkte einlesen

Die Funktion `read_waypoints()` aus der Klasse `waypoint_nav` soll drei Zahlenwerte aus einer Textdatei(.txt) einlesen und diese einem Vektor übergeben. Der Vektor `m_wp` nehme Werte vom Datentyp `float` an und wird in der Klasse `waypoint_nav` deklariert (Codeauszug 5.3, Zeile 35). Zum Bearbeiten von Textdateien werden in C++ *Streams* benutzt. Die Standardklasse `ifstream` dient zum Lesen aus Dateien. Dabei wird Zeilenweise eingelesen und dann als zusammenhängender `string` übergeben. Die while-Schleife (Codeauszug 5.2, Zeile 46) durchläuft alle Zeilen und ersetzt Kommata und Tabulatoren durch Leerzeichen. Dadurch können die Zahlenwerte besser aus der Textdatei eingelesen werden ohne das auf Syntax zu achten ist. Zusätzlich werden Zeilen die mit einem ‘#’ beginnen nicht betrachtet und können für Kommentare verwendet werden. Die Zeilen aus der Textdatei werden dann in einen `stringstream` zeilenpuffer(zeile) umgewandelt und zwischengespeichert. In C++ ist es nun durch den `>>` Operator möglich, den String `linebuffer` einzulesen und den float-Variablen `m_x`, `m_y` und `m_w` zuzuweisen (Codeauszug 5.2, Zeile 56). Außerdem zählt die Variable `m_num_wp` die Anzahl an Zeilen und damit die Anzahl der eingelesenen Messpunkte. Über den Befehl `push_back` werden die eingelesenen Werte dann der x- und y-Koordinate, sowie der z-Achsen-Drehung des Messpunktes zugeordnet. Die z-Achsen-Drehung kann in der Textdatei in Grad angegeben werden, da sie über $m_w \cdot \frac{3.14}{180}$ in Radian umgerechnet wird. Die Werte werden dann im folgendem weiterverarbeitet.

3.2.2 Wegpunkte publishen

Die Funktion `update_waypoint()` aus der Klasse `waypoint_nav` dient dazu die eingelesenen Messpunkte dem TurtleBot3 zu übergeben. In einer if-Schleife werden für jeden eingelesenen Wert folgende Operationen durchgeführt, um dem Turtlebot eine Nachricht für den Topic zu übergeben. Die Position und die Orientierung des gewünschten Wegpunktes wird in einem Objekt Wegpunkt vom Datentyp `PoseStamped` gespeichert. Dann werden diesem Objekt die gewünschten Header (seq,stamp und frame_id) zugewiesen. Die Laufvariable seq dient dazu den aktuellen Messwert zu bestimmen und wird

am Ende der if-Schleife inkrementiert. Über die Funktion aus "ros/ros.h", `Time::now()`, wird dem Header stamp der von ROS aufgenommene Zeitstempel übergeben. Außerdem benötigt das Objekt Wegpunkt für die Visualisierung in RViz eine `frame_id`, welche ihm den Namen der Karte gibt, auf der er arbeiten soll. Die eingelesene Datei besteht nun aus einer Reihe von Zahlenwerten, wobei jeweils drei Zahlenwerte pro Wegpunkt angegeben werden. Der erste Zahlenwert ist immer die x-Komponente, abhängig von dem Durchlauf der Schleife wird also beim ersten Durchlauf (`seq=0`) der Zahlenwert an der Stelle $0 \cdot 3 + 0 = 0$ der `pose.position.x` zugewiesen, dann (`seq=1`) der Zahlenwert an der Stelle $1 \cdot 3 + 0 = 3$, also der erste Zahlenwert des zweiten Messpunktes für die x-Komponente. Für die y- und orientation(z-) Komponente gilt dasselbe Prinzip nur das dann der Zahlenwert an der Stelle $n+1$ (oder $n+2$) benutzt wird. Für die Orientierung kommt zusätzlich eine Transformation in Quaternion hinzu. Die Variable `m_update` wird zunächst *false* gesetzt, damit erst die Aktion (Unterabschnitt 3.2.4) durchgeführt wird. Im Anschluss wird das Objekt Wegpunkt dann dem `ros::Publisher m_waypoint_pub` übergeben und dem TurtleBot3 gepublished. Wurden alle Wegpunkte der Textdatei abgearbeitet, so erfolgt eine Ausgabe.

3.2.3 Wegpunktstatus

Die Funktion `mb_resultCallback` aus der Klasse `waypoint_nav` sorgt dafür, dass Daten aus `move_base_msgs` ausgegeben werden können, indem man einen Zeiger übergibt der nur lesen kann. In diesem Fall wird die Funktion jedoch nicht dazu genutzt um Daten auszugeben, sondern um zu verifizieren, dass der TurtleBot3 sein Ziel erreicht hat. Dafür wird die msg, welche ein konstanter Zeiger aus der Klasse `move_base_msgs` mit der Funktion `MoveBaseActionResult` ist (Codeauszug 5.2, Zeile 19), mit `status.status` verglichen. Ist der `status.status==3`, so bedeutet das, dass der aktuelle Messpunkt erreicht wurde. Ein switch-case entscheidet, ob ein neuer Wegpunkt angefahren, oder eine Aktion ausgeführt werden soll. Je nach Zustand der boolschen Variable `m_update` wird dann entweder die Aktion durchgeführt oder es wird über die Funktion `update_waypoints()` der nächste Messpunkt geladen und der TurtleBot3 fährt dorthin. Gibt `status.status` einen anderen Wert als 3, so hat der TurtleBot3 sein Ziel nicht erreicht ([Mar13] Zeile 248).

3.2.4 Aktion am Wegpunkt

Wenn der TurtleBot3 einen Zielwegpunkt erreicht hat wird die Funktion `action_waypoints()` aus der Klasse `waypoint_nav` aufgerufen. Ähnlich wie bei der `update_waypoints()`-Funktion wird eine Nachricht für die Ausgabe aufgestellt. Wieder werden die Sequenz, der aktueller Zeitstempel, die `frame_id` für die `map` sowie x- und y-Koordinate übergeben. Die Aktion, die der TurtleBot3 durchführen soll ist eine Drehung um 180° und wird der `pose.orientation` übergeben. Der Winkel wird dabei von Grad in Radian umgerechnet und in Quaternion transformiert (Codeauszug 5.2, Zeile 102). Der Wegpunkt, mit dem Kommando zur 180° -Drehung wird dann veröffentlicht und die Variable `m_update` wird auf `true` gesetzt, sodass der nächste Wegpunkt angefahren werden kann.

4 Fazit

Im Rahmen des Projektes 3 wurde ein Programm entwickelt, welches einen TurtleBot3 mittels eingelesenen Wegpunkten steuern kann.

Im Kapitel 2 dieser Hausarbeit wurden die technischen Grundlagen, wie das Erstellen der Umgebung in der sich der Turtlebot3 bewegt, die verwendete Software, die Pfadplanungsverfahren und die Nodes die miteinander kommunizieren, erklärt.

Der TurtleBot3 ist jetzt in der Lage beliebig viele Wegpunkte aus einer Textdatei einzulesen. Bei der Textdatei muss der Pfad allerdings absolut eingeben werden und nicht relativ zum Verzeichnis in dem sich das Programm befindet. Dann kann etwas umständlich sein kann. Demnach muss der gesamte Pfad wie hier kurz gezeigt: *home/user/catkin_ws/...* eingegeben werden, die übliche Verwendung durch die Tilde ist nicht möglich. Seine Sensoren erkennen Hindernisse und RVis erstellt eine *Costmap* mit einem Toleranzbereich um die Hindernisse, damit der TurtleBot3 mit diesen nicht kollidieren kann. Der TurtleBot3 führt, wenn er sein Ziel erreicht hat, eine Rückgabe in Form einer 180°-Drehung durch. Derzeit wird im Programm nur ein Status abgefragt. Lediglich ob der TurtleBot3 sein Ziel erreicht hat oder nicht. Eine Fehlersuche, weshalb das Ziel nicht erreicht werden konnte, fehlt. Erweitert man jedoch das Programm um weitere Abfragen, erhöhe das zwar den Umfang aber macht eine Fehlersuche möglich. Dann könne man herausfinden, weshalb ein Wegpunkt nicht erreicht werden kann. Die Eingabe der Wegpunkte ist flexibel und man braucht nur wenig bei der Syntax zu beachten, da diese bereits beim Einlesen angepasst wird. Eine Falscheingabe ist dennoch möglich und es werden nicht alle Syntax-Fehler aufgefangen.

Trotzdem ließt der TurtleBot3 die Wegpunkte erfolgreich ein, fährt diese an und führt nach Erreichen eine 180°-Drehung durch.

Abbildungsverzeichnis

1.1	Path-Planning [Cha09]	1
2.1	TurtleBot3 Burger	4
2.2	Gazebo Simulationsumgebung	6
2.3	Modell HAWK Fak [I], Gebäude C, 1. Obergeschoss	7
2.4	Visualisierungstool RViz	8
2.5	Map HAWK_Modell	9
2.6	Roadmap-Pfadplanung [Lat90]	12
2.7	Vergleich Algorithmus	13
2.8	Pfadplanung Potentialfeld	13
2.9	Roboter mit Koordinatensystem	15
2.10	Navigation Stack Setup move_base [nic20]	16
5.1	SLAM-Algorithmus gmapping	M

Tabellenverzeichnis

2.1	Technische Daten zum TurtleBot3 [Rob20]	4
5.1	Aufteilung der Arbeit	G

Quelltextverzeichnis

5.1	waypointnavigation.cpp	H
5.2	waypoint_nav.cpp	H
5.3	waypoint_nav.h	K

Literaturverzeichnis

- [Lat90] Jean-Claude Latombe. *Robot Motion Planning*. Springer, 1990.
- [TBF05] Sebastian Thrun, Wolfram Burgard und Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005. ISBN: 0262201623.
- [Cha09] Jorge Cham. *R.O.B.O.T. Comics: Path Planning*. 2009. URL: <https://www.willowgarage.com/blog/2009/09/04/robot-comics-path-planning?page=32>. (Abruf am 21.Dez. 2020).
- [Foo13] Tully Foote. *ROS/Introduction*. 2013. URL: <http://wiki.ros.org/de/ROS/Introduction>. (Abruf am 28.Dez. 2020).
- [Mar13] Eitan Marder-Eppstein. *MoveBaseActionResult.h*. 2013. URL: http://docs.ros.org/en/electric/api/move_base_msgs/html/MoveBaseActionResult_8h_source.html. (Abruf am 29.Dez. 2020).
- [Mic13] Schmidt Michael. „Evaluierung gitterbasierter Pfadplanungs-Algorithmen für die Hardwrebeschleunigung mit FPGAs“. Dissertation. Friedrich-Alexander-Universität Erlangen-Nürnberg, 2013.
- [FP14] Tully Foote und Mike Purvis. *Standard Units of Measure and Coordinate Conventions*. 2014. URL: <https://www.ros.org/reps/rep-0103.html>. (Abruf am 24.Dez. 2020).
- [Mar+17] Pablo Marin-Plaza u. a. „Global and Local Path Planning Study in a ROS-Based Research Platform for Autonomous Vehicles“. Research Article. Universidad Carlos III de Madrid, Leganes, 2017.
- [Pyo+17] YoonSeok Pyo u. a. *ROS Robot Programming*. ROBOTIS Co.,Ltd., 2017.
- [Hab18] HabibOladepo. *Nodes*. 2018. URL: <http://wiki.ros.org/Nodes>. (Abruf am 29.Dez. 2020).
- [Mar19] MarcoSteinbrink. *robot_statemachine*. 2019. URL: http://wiki.ros.org/robot_statemachine. (Abruf am 29.Dez. 2020).
- [Lin20] Thomas Linkugel. *Robotik und autonome Systeme*. Skript. HAWK Göttingen, 2020.
- [nic20] nickfragale. *move_base*. 2020. URL: http://wiki.ros.org/move_base. (Abruf am 29.Dez. 2020).
- [NR20] Niklas Noack und Jan-Luca Regenhardt. „Tutorium Robotik und autonome Systeme“. Präsentation. HAWK Göttingen, 2020.

- [Qui+20] Morgan Quigley u. a. *ros::NodeHandle Class Reference*. 2020. URL: http://docs.ros.org/en/noetic/api/roscpp/html/classros_1_1NodeHandle.html. (Abruf am 30.Dez. 2020).
- [Rob20] Robotis. *TurtleBot3 Burger Hardware*. 2020. URL: <https://emanual.robotis.com/docs/en/platform/turtlebot3/specifications/>. (Abruf am 24.Dez. 2020).

5 Anhang

5.1 Aufteilung der Arbeit

Die Karte in Abbildung 5.1 entstand in alleiniger Arbeit durch Herrn Regenhardt.

Das Programm entstand zu gleichen Teilen in Gruppenarbeit.

Kapitel/Abschnitt	Bearbeiter/-in
Einleitung	Ch.Gruß
Aufbau der Arbeit	Ch.Gruß
Technische Grundlagen	Ch.Gruß
Robot Operating System	R.Scheel
TurtleBot3	Ch.Gruß
Simulationsumgebung Gazebo	J.Regenhardt
Modellbildung	J.Regenhardt
Visualisierungstool RViz	J.Regenhardt
Mapping	J.Regenhardt
Global- und Local-Planner	Ch.Gruß
Pfadplanung mit Roadmap	Ch.Gruß
Pfadplanung mit Zellzerlegung	Ch.Gruß
Pfadplanung mit Potentialfeld	Ch.Gruß
Koordinatensystem und Wegpunkt	Ch.Gruß
Topic und Nodes im Robot State Maschine	R.Scheel
Programmaufbau	R.Scheel
Initialisierung	R.Scheel
Erläuterung der Funktionen	R.Scheel
Wegpunkt einlesen	R.Scheel
Wegpunkt publishen	R.Scheel
Wegpunktstatus	R.Scheel
Aktion am Wegpunkt	R.Scheel
Fazit	R.Scheel

Tabelle 5.1: Aufteilung der Arbeit

5.2 Programmcode

Codeauszug 5.1: waypointnavigation.cpp

```

1  /*************************************************************************/
2  /*  NodeName: Wegpunktnavigation des TurtleBot3 in ROS               */
3  /*  Bearbeiter: Chiara Teresa Gruß, Robin Scheel, Jan-Luca Regenhardt   */
4  /*          Hochschule für angewandte Wissenschaft und Kunst, Fak [i]      */
5  /*  Datum:      07.01.2021, 13:27 Uhr                                     */
6  /*************************************************************************/
7
8 #include "waypoint_nav.h" //Einbinden der Haupt-Headerdatei
9
10 int main(int argc, char **argv) //Hauptschleife
11 {
12     ros::init(argc, argv, "waypointnavigation"); //Initialisierung des Nodes "waypointnavigation" am Master
13
14     rob_proj4::waypoint_nav go; //Erzeugung eines Klassenobjektes (Enthält alle weiteren Aktionen)
15
16     ros::spin(); //Endlosschleife, in welcher der Node auf Aktionen oder ein Stop wartet.
17     return 0;
18 }
```

Codeauszug 5.2: waypoint_nav.cpp

```

1 #include "waypoint_nav.h"
2
3 namespace rob_proj4{
4
5     waypoint_nav::waypoint_nav(): //Initialisierung des Konstruktors einer Klasse
6         seq(0),
7         m_num_wp(0),
8         m_update(false)
9     {
10         //Subsriber "/move_base/result", löst "mb_resultCallback" aus
11 }
```

```
11     m_waypoint_sub = m_ns.subscribe<move_base_msgs::MoveBaseActionResult>("/move_base/
12         result", 1, &waypoint_nav::mb_resultCallback, this);
13     //Publisher auf "/move_base_simple/goal"
14     m_waypoint_pub = m_np.advertise<geometry_msgs::PoseStamped>("/move_base_simple/goal",
15         1, this);
16     read_waypoints(); //Einlesen der Wegpunkte
17     update_waypoints(); //Erstes Publishen eines Wegpunktes
18 }
19
20 /**
21 * **** Callback-Funktion: Erreichen/Abbruch der Wegpunktnavigation *****/
22 void waypoint_nav::mb_resultCallback(const move_base_msgs::MoveBaseActionResult::ConstPtr
23     & msg){
24
25     if(msg->status.status == 3){ //3: Ziel wurde vom Action Server erfolgreich angefahren.
26
27         switch(m_update){ //Entscheidet, ob ein neuer Wegpunkt angefahren, oder eine Aktion
28             //ausgeführt werden soll.
29
30             case true:
31                 update_waypoints();
32                 break;
33             case false:
34                 std::cout << "SUCCEEDED: " << seq+1 << ". Goal Reached" << std::endl;
35                 action_waypoints();
36                 break;
37             }
38         }
39     else{
40         std::cout << "NOT SUCCEEDED: Goal not Reached or Aborted" << std::endl;
41     }
42 }
43
44 /**
45 * **** Funktion: Einlesen der Wegpunkte aus einer .txt Datei *****/
46 void waypoint_nav::read_waypoints(){
47
48     float m_x, m_y, m_w; //Variablen für die Koordinatenübergabe
49
50     std::ifstream inputfile("/home/jan/catkin_ws/src/waypointnavigation/src/waypoints.txt")
51         ; //Dateipfad
52     std::string line; //Variable für die aktuelle Zeile
53
54     while (std::getline(inputfile, line)){ //Lese Zeile ein und führe Aktion bis
55         Zeilenumbruch aus.
```

```

47     for (int i = 0; i < line.size(); i++){ //Jedes einzelne Zeichen in der aktuellen
        Zeile durchgehen.
48     if (line[i] == ',' || line[i] == '\t'){ //Ersetze alle Tabulatoren und Komma mit
        Leerzeichen
49
50         line[i] = ' ';
51     }
52 }
53
54 if (line[0] != '#'){ //Wenn KEIN Kommentarzeichen "#" genutzt wurde, verarbeite
    aktuelle Zeile
55 std::stringstream linebuffer(line); //Zwischenspeichern der aktuellen Zeile
56 linebuffer >> m_num_wp >> m_x >> m_y >> m_w; //Abspeichern der Koordinaten in
    Zwischenvariablen
57 std::cout << m_num_wp << ":" << m_x << "\t" << m_y << "\t" << m_w << std::endl; //Ausgabe der Wegpunkte
58
59 m_wp.push_back(m_x); //x-Koordinate an Vektor hinten anhängen
60 m_wp.push_back(m_y); //y-Koordinate an Vektor hinten anhängen
61 m_wp.push_back((m_w*3.14)/180); //z-Achsenrotation an Vektor hinten anhängen (
    Umrechnung von Deg in Rad)
62 }
63 }
64 }
65
66 //***** Funktion: Publishen eines Wegpunktes *****/
67 void waypoint_nav::update_waypoints(){
68
69     if(seq < m_num_wp){ //Nur ausführen, wenn neue Wegpunkte vorhanden sind
70         geometry_msgs::PoseStamped Wegpunkt; //Objekt Wegpunkt vom Datentyp PoseStamped (Enthält Position und Orientierung)
71
72         //Zusammbau der Nachricht für den Topic
73         Wegpunkt.header.seq = seq; //Sequenz des Topics
74         Wegpunkt.header.stamp = ros::Time::now(); //Aktueller Zeitstempel in ROS-Zeit
75         Wegpunkt.header.frame_id = "map"; //frame_id (wichtig für RVis)
76
77         Wegpunkt.pose.position.x = m_wp[(seq)*3+0]; //Eingabe der x-Koordinate
78         Wegpunkt.pose.position.y = m_wp[(seq)*3+1]; //Eingabe der y-Koordinate
79         Wegpunkt.pose.orientation = tf::createQuaternionMsgFromYaw(m_wp[(seq)*3+2]); //Eingabe der z-Achsenrotation mit Umrechnung in Quaternionen
80
81         m_waypoint_pub.publish(Wegpunkt); //Publish des Wegpunktes

```

```

82     m_update = false; //Kein Publishen eines neuen Wegpunktes, zunächst die Aktion
83 }
84 else std::cout << "All Waypoints processed" << std::endl;
85 }
86
87 ***** Funktion: Aktion nach Ankunft an einem Wegpunkt *****
88 void waypoint_nav::action_waypoints(){
89
90     if(seq < m_num_wp){
91         geometry_msgs::PoseStamped Wegpunkt; //Objekt Wegpunkt vom Datentyp PoseStamped (
92             Enthält Position und Orientierung)
93
94         //Zusammebau der Nachricht für den Topic
95         Wegpunkt.header.seq = seq; //Sequenz des Topics
96         Wegpunkt.header.stamp = ros::Time::now(); //Aktueller Zeitstempel in ROS-Zeit
97         Wegpunkt.header.frame_id = "map"; //frame_id (wichtig für RViz)
98
99         Wegpunkt.pose.position.x = m_wp[(seq)*3+0]; //Eingabe der x-Koordinate
100        Wegpunkt.pose.position.y = m_wp[(seq)*3+1]; //Eingabe der y-Koordinate
101
102        //Hier die Aktion: Drehung um 180° am Wegpunkt
103        Wegpunkt.pose.orientation = tf::createQuaternionMsgFromYaw(m_wp[(seq)
104            *3+2]+(180*(3.14/180)));
105
106        m_waypoint_pub.publish(Wegpunkt); //Publish der Aktion am Wegpunkt
107        seq++; //Erhöhung des Sequenzzählers um 1
108    }
109    m_update = true; //Nächster Wegpunkt kann gepublished werden
110 }
111 } //end namespace rob_proj4

```

Codeauszug 5.3: waypoint_nav.h

```

1 #include "ros/ros.h" //ROS-Standard Bibliothek
2 #include "geometry_msgs/PoseStamped.h" //Publisher Objekt
3 #include "move_base_msgs/MoveBaseActionResult.h" //Subscriber Objekt
4 #include <tf/tf.h> //Transformation (für Quaternionen)
5 #include <vector>
6 #include <iostream> //C++ Bibliothek für Konsolenausgabe
7 #include <fstream> //Zum Auslesen der Wegpunkt-Datei

```

```
8 #include <string>
9
10 namespace rob_proj4
11 {
12     class waypoint_nav
13     {
14     public:
15         waypoint_nav(); //Standardkonstruktor
16
17     private:
18
19     //Callback wenn der Action-Server die Ankunft-Nachricht für einen Wegpunkt published
20     void mb_resultCallback(const move_base_msgs::MoveBaseActionResult::ConstPtr& msg);
21
22     //Funktionen zum Auslesen und Publishen der Wegpunkte. Und Aktion nach ankunft am
23     //Wegpunkt.
24     void read_waypoints();
25     void update_waypoints();
26     void action_waypoints();
27
28     //Handler für das abonnieren des Ergebnisses der Navigation
29     ros::NodeHandle m_ns;
30     ros::Subscriber m_waypoint_sub;
31
32     //Handler für das Publishen eines Wegpunktes
33     ros::NodeHandle m_np;
34     ros::Publisher m_waypoint_pub;
35
36     std::vector<float> m_wp; //Vektor-Objekt zum zwischenspeichern der Wegpunkte aus der
37     //Datei.
38     int m_num_wp; //Zähler, wie viele Wegpunkte geladen wurden.
39     int seq; //Aktueller Stand (Sequenz) der Wegpunkte.
40     bool m_update; //Schalter, ob entweder ein Wegpunkt angefahren oder eine Aktion ausgefü
41     //hrt werden soll.
42 };
43 }
```

5.3 Umgebungs-Karte mit SLAM

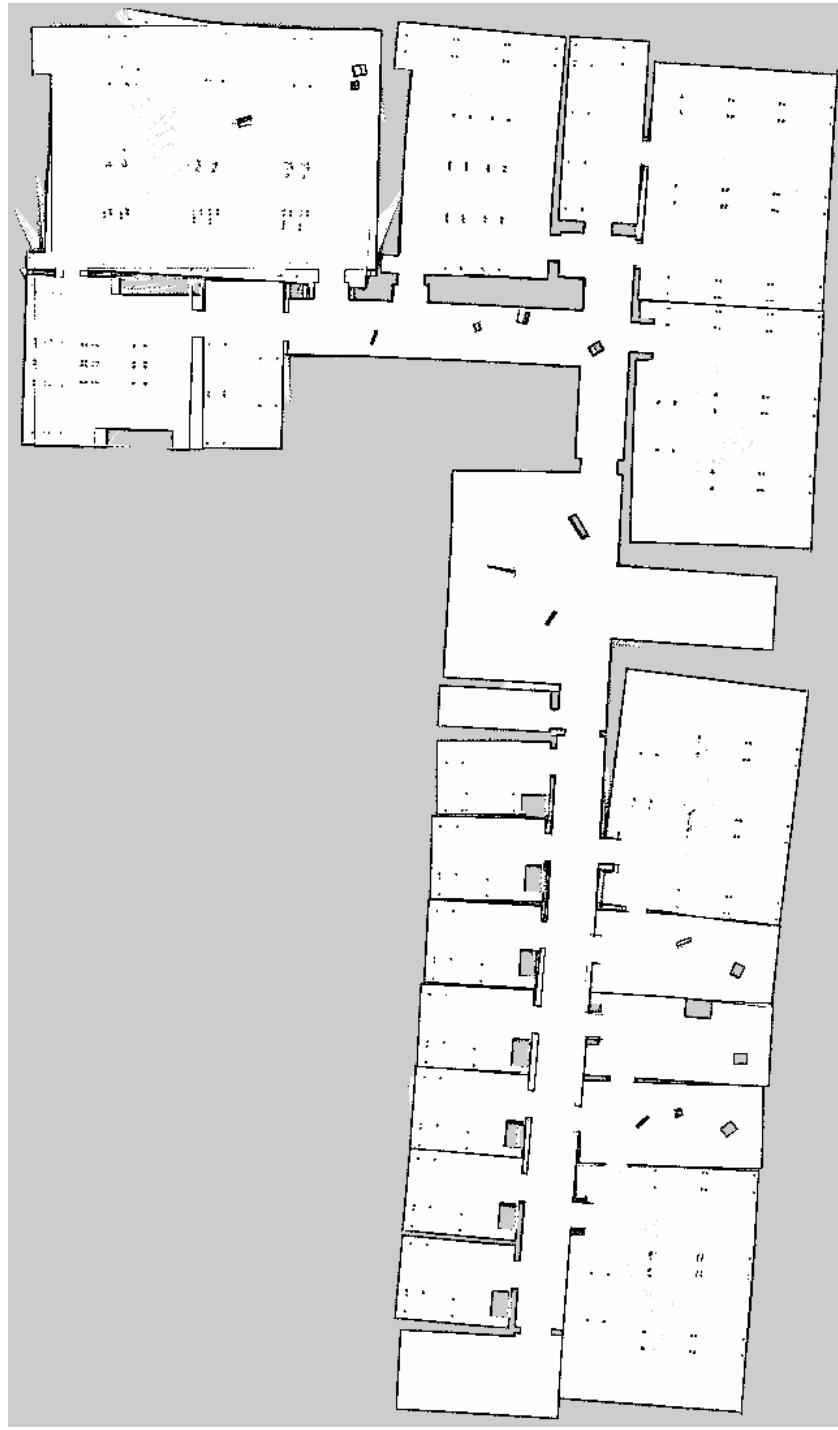


Abbildung 5.1: SLAM-Algorithmus gmapping