



hochschule mannheim

Untersuchung von Kartierungsalgorithmen unter ROS mit dem Pioneer 3-DX

Daniel Koch

Studienarbeit
Studiengang Informatik

Fakultät für Informatik
Hochschule Mannheim

23.10.2019

Betreuer
Prof. Dr. Thomas Ihme, Hochschule Mannheim

Koch, Daniel:

Untersuchung von Kartierungsalgorithmen unter ROS mit dem Pioneer 3-DX / Daniel Koch.

–

Studienarbeit, Mannheim: Hochschule Mannheim, 2019. 43 Seiten.

Koch, Daniel:

Analysis of cartographing algorithms in ROS using the Pioneer 3-DX / Daniel Koch. –
Study Project, Mannheim: University of Applied Sciences Mannheim, 2019. 43 pages.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, d. h. dass die Arbeit elektronisch gespeichert, in andere Formate konvertiert, auf den Servern der Hochschule Mannheim öffentlich zugänglich gemacht und über das Internet verbreitet werden darf.

Mannheim, 23.10.2019

Daniel Koch

Abstract

Untersuchung von Kartierungsalgorithmen unter ROS mit dem Pioneer 3-DX

Diese Studienarbeit untersucht Kartierungsalgorithmen wie Google Cartographer, gmapping oder hector_mapping im Robot Operating System (ROS), einem open-source Meta-Betriebssystem für die Entwicklung von Robotern. Es wird die Funktionsweise der genannten Algorithmen analysiert und ein Package aufgesetzt, welches mit dem Pioneer 3-DX kompatibel ist. Dieses Package wird genutzt, um die Fähigkeit der Algorithmen unter bestimmten Bedingungen in einer 2D-Umgebung zu testen. In diesen Tests schneidet der Google Cartographer auf Grund von genauen Kartierungen und einer Vielzahl an Konfigurationsmöglichkeiten am besten ab.

Analysis of cartographing algorithms in ROS using the Pioneer 3-DX

This study evaluates cartographing algorithms like Google Cartographer, gmapping or hector_mapping within the Robot Operating System (ROS). ROS is an open-source meta operating system for the development of robots. The study analyzes the functionality of these algorithms and sets up a package which is compatible with the Pioneer 3-DX. The package is used then to evaluate the capability of the algorithms in a defined 2D environment. In these tests the Google Cartographer performs best since it creates the most accurate maps and had the most configuration options.

Inhaltsverzeichnis

1 Einleitung	1
1.1 Motivation	1
1.2 Ziel der Arbeit	1
1.3 Aufbau der Arbeit	2
2 Grundlagen	3
2.1 Robot Operating System	3
2.1.1 Einführung	3
2.1.2 Technischer Aufbau	5
2.2 Pioneer 3-DX	8
3 Kartierungsalgorithmen	11
3.1 cartographer	11
3.1.1 Algorithmus	11
3.1.2 Verwendung im Robot Operating System	13
3.2 hector_mapping	14
3.2.1 Algorithmus	14
3.2.2 Verwendung im Robot Operating System	15
3.3 gmapping	16
3.3.1 Algorithmus	16
3.3.2 Verwendung im Robot Operating System	17
4 Integration der Algorithmen	19
4.1 Aufsetzen des eigenen Packages	19
4.2 Installation der Kartierungsalgorithmen	21
4.2.1 cartographer	21
4.2.2 hector_mapping	21
4.2.3 gmapping	22
4.3 Aufsetzen der Launch-Files und der Konfiguration	22
4.3.1 cartographer	23
4.3.2 hector_mapping	23
4.3.3 gmapping	24
4.4 Starten der Kartierung	24
4.4.1 Starten der Online-Kartierung	24

Inhaltsverzeichnis

4.4.2 Starten der Offline-Kartierung	25
5 Testen der Algorithmen	33
5.1 cartographer	34
5.2 gmapping	35
5.3 hector_mapping	35
5.4 Vergleich	36
6 Zusammenfassung	41
7 Ausblick	43
Abkürzungsverzeichnis	vii
Tabellenverzeichnis	ix
Abbildungsverzeichnis	xi
Quellcodeverzeichnis	xiii
Literatur	xv

Kapitel 1

Einleitung

1.1 Motivation

Für mobile Roboter ist es eine wichtige Aufgabe, sich in einer unbekannten Umgebung zurechtzufinden. Dazu ist ein nötiger erster Schritt die Kartierung der Umgebung. Diese Kartierung bringt einige neue Herausforderungen mit sich, die jeder Algorithmus unterschiedlich löst. Die Algorithmen bieten Implementierungen für das Robot Operating System (ROS). Da jeder Roboter andere Sensordaten liefern, muss auch der Algorithmus so dynamisch gestaltet sein, dass er mit verschiedenen Daten umgehen kann. Dementsprechend muss der Anwender spezielle Launch-Files und Konfigurationen definieren, um den Algorithmus für den jeweiligen Roboter möglichst gut abzubilden.

1.2 Ziel der Arbeit

Ziel dieser Arbeit ist es deshalb darzustellen, welche Kartierungsalgorithmen existieren und wie diese sich für den Pioneer 3-DX Roboter in das ROS integrieren lassen. Im Detail soll ein ROS-Package entwickelt werden, welches die optimierten Launch-Files und Konfigurationen für die drei Kartierungsalgorithmen *Cartographer*, *gmapping* und *hector_mapping* enthält. Die Arbeit soll außerdem einen Überblick über die Installation dieser Packages geben sowie eine Anleitung, wie die Online- oder Offline-Kartierung gestartet werden kann. Des Weiteren ist es Ziel dieser Arbeit die Algorithmen auszuführen und einige Szenarien mit diesen zu durchlaufen und auszuwerten.

1.3 Aufbau der Arbeit

Zuerst werden die Grundlagen in Kapitel 2 behandelt. Diese sind die Grundlagen über das ROS sowie die des Roboters Pioneer 3-DX.

Der Hauptteil befasst sich in Kapitel 3 zunächst mit den drei Kartierungsalgorithmen *Cartographer*, *gmapping* und *hector_mapping*. Dabei wird beschrieben, wie die Algorithmen technisch funktionieren und wie sie sich in das ROS integrieren lassen. Kapitel 4 implementiert diese Algorithmen durch Launch- und Konfigurationsfiles in einem eigenen ROS-Package für den Pioneer 3-DX. In Kapitel 5 werden die Implementierungen in zwei Indoor-Szenarien mit diesem Roboter getestet.

Zum Abschluss bietet Kapitel 6 eine Zusammenfassung und Kapitel 7 einen Ausblick, wie die Kartierung sonst noch hätte verbessert werden können.

Kapitel 2

Grundlagen

Das folgende Kapitel beschreibt die Grundlagen dieser Arbeit. Diese bestehen aus dem Robot Operating System, welches die Basis des zu entwickelnden Roboters darstellt. Danach wird der Roboter selbst beschrieben.

2.1 Robot Operating System

Zunächst geht diese Arbeit auf das ROS ein. Dazu wird das ROS zunächst allgemein vorgestellt. Im zweiten Teil geht es dann um die technische Seite von ROS.

2.1.1 Einführung

Das Robot Operating System ist ein Framework zur Entwicklung von Robotern. Allgemein bietet es Bausteine und Werkzeuge zur einfachen Entwicklung von Robotern. Die Wichtigsten davon sind Erweiterungen der Kommandozeilen, fertige Implementierungen von Software-Algorithmen, Hardware-Abstraktionen und Oberflächen zur Visualisierung und zum Testen von Robotern.

Beim Entwickeln von Robotern sowie bei der Entwicklung von Software treten häufig die gleichen Herausforderungen auf. Diese werden im folgenden dargestellt und es wird beschrieben, wie ROS diese Herausforderungen handhabt:

- *Netzwerk-Transparenz*: Ein Roboter besteht in der Regel nicht nur aus einem einzigen Computer. Öfters werden für mehrere Sensoren und Aktoren auch mehrere Computer zur Berechnung genutzt. Daher muss ein Framework zur

Entwicklung von Robotern die Fähigkeit besitzen, zwischen Prozessen und Netzwerken zu kommunizieren. Das Robot Operating System bietet diese Fähigkeit durch den ROS-Master, der Nachrichten sowohl lokal als auch über das Netzwerk verteilen kann.

- *Große Anzahl an verfügbaren Packages:* Beim Entwickeln von Robotern tritt der Entwickler häufig auf die gleichen algorithmischen Probleme, die der Entwickler nicht erneut entwerfen möchte. Daher bietet das Robot Operating System durch die Community viele bekannte Algorithmen in Form von Packages, die direkt installiert, konfiguriert und genutzt werden können. Der Entwickler kann sich mehr darauf fokussieren, neue Ideen auszuprobieren, statt Algorithmen zu entwerfen. Dies vereinfacht die Entwicklung von Roboterapplikationen deutlich.
- *Testprozess:* Das Testen von Robotern kann auf Grund des Ablaufs auf der Hardware sehr zeitintensiv sein. Außerdem ist es oft auch hilfreich, die Hardware erst einmal wegzulassen und ein Modell zu entwerfen. Daher gilt im Robot Operating System der Grundsatz, die Hardware- und die Software-Implementierung strikt zu trennen. Des Weiteren gibt es die Möglichkeit Sensordaten aufzuzeichnen und später erneut zum Testen und zur Fehleranalyse abzuspielen.
- *Mehrsprachigkeit:* Jede Programmiersprache hat in Bezug auf Fehleranalyse, Syntax oder Effizienz Vor- und Nachteile. Des Weiteren hat jeder Entwickler nochmals eigene Präferenzen. Daher ist das Robot Operating System gegenüber der Programmiersprache neutral. Dies bedeutet, dass im ROS auf dem Messaging Layer entwickelt wird. Es existiert eine sogenannte Interface Definition Language, um Nachrichten zwischen den Nodes zu beschreiben und auszutauschen. Code-Generatoren für jede Programmiersprache, erstellen daraus native Implementierungen. Das Serialisieren und Deserialisieren der Nachrichten zu der jeweiligen Programmiersprache erledigt das Robot Operating System automatisch.
- *Flexibles Bausteinsystem:* Oftmals ist Software, die als Monolith aufgebaut ist, nicht so flexibel wie ein modulares System. Das Robot Operating System besteht aus vielen kleinen Paketen, die der Entwickler je nach Aufgabe flexibel verbinden kann.

- *Open Source:* Zuletzt ist das Robot Operating System kostenlos und wird unter der BSD-Lizenz zur Verfügung gestellt. Dies ermöglicht eine große Gemeinschaft und das Erfassen von Fehlern auf allen Ebenen der Software.

ROS ist ein Meta-Betriebssystem, das auf einem bestehenden Betriebssystem läuft. Die Software ist für die Linux Distribution Ubuntu und Debian als stabile Version verfügbar und läuft auch unter Windows Services für Linux. Experimentelle Versionen sind allerdings für Betriebssysteme wie zum Beispiel OS X vorhanden. [1] [2] [3]

ROS bietet verschiedene Releases an. Tabelle 2.1 zeigt eine Auflistung dieser Releases, deren Veröffentlichungsdatum sowie dem jeweiligen Datum des sogenannten End-of-Life (EOL) Punktes:

Aktuell wird das Release ROS Melodic Morenia empfohlen. Ein ROS-Release ist immer an eine bestimmte Ubuntu LTS Version geknüpft. [4]

2.1.2 Technischer Aufbau

Das Robot Operating System ist modular aufgebaut. Zentral stehen die sogenannten Packages. Jedes Package hat seine eigenen Funktionalitäten. Packages definieren Nodes, in denen die eigentlichen Berechnungen ablaufen. Nodes können paketübergreifend über Topics oder Services kommunizieren. Der ROS-Master übernimmt die Kommunikation über die Topics und Services.

Zu jedem Package gehört ein Manifest. Dieses wird in der Datei package.xml definiert. Es beinhaltet den Namen des Pakets, die Version, den sogenannten Maintainer

Tabelle 2.1: ROS-Versionen [4]

Name der Version	Veröffentlichungsdatum	EOL-Datum
ROS Melodic Morenia	23.05.2018	05.2023
ROS Lunar Loggerhead	23.05.2017	05.2019
ROS Kinetic Kame	23.05.2016	04.2021
ROS Jade Turtle	23.05.2015	05.2017
ROS Indigo Igloo	22.07.2014	04.2019
ROS Hydro Medusa	04.09.2013	05.2015
ROS Groovy Galapagos	31.12.2012	07.2014
ROS Fuerte Turtle	23.04.2012	-
ROS Electric Emys	30.08.2011	-
ROS Diamondback	02.03.2011	-
ROS C Turtle	02.08.2010	-
ROS Box Turtle	02.03.2010	-



Abbildung 2.1: Beziehung zwischen Topics und Nodes

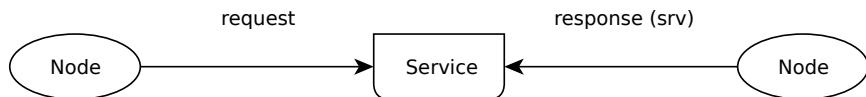


Abbildung 2.2: Beziehung zwischen Services und Nodes

sowie die Abhängigkeiten zu anderen Paketen. Des Weiteren beinhaltet ein Package die Definitionen für Nachrichten und Services. Weiterhin definiert es Nodes in Form von Python- oder C++ Programmen.

Nodes können Nachrichten austauschen. Diese Nachrichtenübertragung erfolgt über den ROS-Master. Üblich ist es, den ROS-Master vor dem Aufruf weiterer Nodes zu starten. Außerdem ist der ROS-Master für die Namensregistrierung sowie für jegliche Lookups zuständig. Es gibt zwei standardisierte Optionen der Kommunikation zwischen Nodes.

Die erste Möglichkeit ist die Kommunikation über sogenannte Topics. Diese Art der Kommunikation findet nach dem Publisher-Subscriber Prinzip statt. Demnach können, wie in Abbildung 2.1 dargestellt, Nodes in Topics veröffentlichen oder diese abonnieren. Es handelt sich dabei um eine „many-to-many“-Verbindung. Das bedeutet, dass mehrere Nodes in einen Topic veröffentlichen können und auch mehrere Nodes einen Topic abonnieren können. Das Nachrichtenformat ist das Format „.msg“. Definitionen von Messages werden in der Regel im Ordner „./msg“ definiert. Das Kommando „rqt_graph“ bietet die Möglichkeit, die Verbindungen zwischen Nodes und Topics grafisch darzustellen.

Die zweite Möglichkeit ist die Kommunikation über sogenannte Services. Diese Art der Kommunikation findet nach dem Client-Server-Prinzip statt. Demnach können Nodes, wie in Abbildung 2.2 dargestellt, Services anbieten oder diese abfragen. Es handelt sich dabei um eine „one-to-one“-Abfrage. Das bedeutet, dass eine Node eine Anfrage an einen Service stellt und nur diese Node antwortet. Das Nachrichtenformat ist das Format „.srv“. Definitionen von Services werden in der Regel im Ordner „./msg“ definiert.

```
<launch>
  <node pkg="rostopic" name="rostopic" type="rostopic" args="pub -1 /
    cmd_motor_state p2os_msgs/MotorState 1"/>
  <node pkg="p2os_driver" name="p2os_driver" type="p2os_driver"/>

  <include file="$(find p2os_launch)/launch/teleop_joy.launch"/>
</launch>
```

Listing 2.1: Beispiel eines Launch-Files in XML

Im Allgemeinen beinhaltet jede Nachricht zusätzlich zu den Daten einen sogenannten Header. Dieser enthält eine eindeutige Identifikationsnummer, einen Zeitstempel und eine Frame-ID.

Ein weiterer Bestandteil von ROS sind die sogenannten Launch-Files. Diese sollen das Ausführen von mehreren Nodes vereinfachen, da am Ende nur noch ein einziges Launch-File gestartet werden muss. Ein Launch-File wird in XML definiert. Üblich ist es, für alle Launch-Files einen Ordner mit dem Namen „launch“ zu erstellen. Neben dem Starten von Nodes können Launch-Files auch andere Launch-Files inkludieren. Außerdem kann eine Namensänderung von Topic-Namen definiert werden, um die Kompatibilität zwischen mehreren Packages zu gewährleisten. Dies nennt sich auch Remapping. Zuletzt bieten Launch-Files noch die Option Nodes bei Absturz neu zu starten, die Ausgabe auf dem Bildschirm auszugeben und die Option Konsolenargumente zu verarbeiten.

Das Launch-File in Listing 2.1 startet zwei Nodes. Die erste Node veröffentlicht über das ROStopic-Node die Message, welche den Befehl zum Motorstart ausführt. Die zweite Node startet die Node, welche Kommandos aufnehmen kann, um den Roboter zu bewegen. Der dritte Block startet ein externes Launch-File, welches wiederum Kommandos für einen Joystick beinhaltet, der Befehle an den p2os_driver geben kann. Launch-Files sind hilfreich, damit alle Nodes nicht einzeln gestartet werden müssen.

Des Weiteren bietet das System mit den sogenannten Bags die Möglichkeit zum Aufzeichnen und Wiedergeben von Nachrichten bzw. spezifischer Sensordaten. Damit ist es möglich, eine Aufnahme mit der Roboter immer wieder abzuspielen, um einen Fehler zu analysieren, ohne den Roboter erneut durchlaufen lassen zu müssen. Bag-Aufnahmen oder Bag-Wiedergaben können auch in Launch-Files definiert werden. Außerdem lassen sich Aufnahmen sowohl über das Terminal als auch über

2 Grundlagen

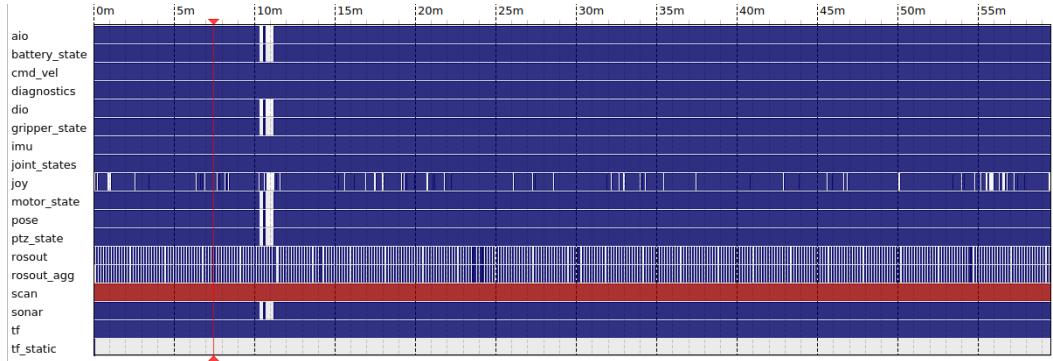


Abbildung 2.3: Abspielen eines ROS-Bags mit RQT

eine grafische Oberfläche starten und modifizieren, wie in Abbildung 2.3 zu sehen ist.

Das Robot Operating System bietet zuletzt noch ein mächtiges Kommandozeilen-Tool, mit dem Aufgaben leichter erledigt werden können. Die in Tabelle 2.2 gezeigten Kommandos, dienen als Überblick über die erläuterten Themen. [1] [2]

2.2 Pioneer 3-DX

Im nächsten Schritt wird der zu verwendende Roboter für die Kartierung vorgestellt und dargelegt, wie dieser mit dem Robot Operating System interagiert.

Der Pioneer 3-DX ist ein mobiler Roboter, der in verschiedenen Bereichen der Robotik eingesetzt werden kann, da er beliebig erweiterbar ist. Beispielsweise lässt dieser sich für die Kartierung einsetzen, wenn er um einen Laserscanner erweitert würde. Eine weitere Möglichkeit ist, dass der Roboter alltägliche Aufgaben erfüllen kann, in dem man ihn mit einem Greifarm ausstattet.

Das Modell 3-DX an sich ist 44,5 cm lang, 39,3 cm breit und 23,7 cm hoch, wiegt 9 Kilogramm und kann maximal 25kg tragen. Des Weiteren besteht der Roboter aus zwei Rädern und einem zusätzlichen Stützrad. Die Art des Fahrens ist das sogenannte Differential Drive. Außerdem besitzt der Pioneer 3-DX Ultraschallsensoren sowie Stoßfänger, falls alle anderen Mechanismen zur Verhinderung eines Aufpralls nicht greifen. [5]

Für diese Arbeit ist ein Umbau des Pioneer 3-DX eingesetzt worden. Zusätzlich zu den genannten Bauteilen kommt noch ein Computer, ein Laserscanner sowie eine Intertial Measurement Unit (IMU) zum Einsatz. Beim Computer handelt es sich



Abbildung 2.4: Pioneer 3-DX mit Lasercanner, Kinect und Computer

um die Linux Distribution Ubuntu 18.04.2 LTS. Dieser ist auf der Roboter-Basis angebracht. Der Computer hat 4 GB Arbeitsspeicher sowie 2 GB an zusätzlichem SWAP. Der Laserscanner ist der SICK LMS, der ebenfalls auf der Roboter-Basis angebracht ist. Die IMU ist die Tinkerforge Brick 2.0.

Der Pioneer 3-DX lässt sich im Netzbetrieb oder im Akkubetrieb starten. Dabei sollte mindestens eine Spannung von 12 Volt anliegen. Der Roboter besitzt drei Plätze für Akkus, die gleichzeitig genutzt werden können. [5]

Tabelle 2.2: ROS-Kommandozeile

Kommando	Erklärung
	<i>Allgemein</i>
roscore	Startet den ROS-Master
rospack list	Gibt alle Packages aus
roscd package-name	Wechselt das Verzeichnis zu einem Package
rosls	Zeigt die Inhalte eines Packages an
rospack find package-name	Findet das Verzeichnis eines Packages
	<i>Nodes</i>
rosrun package-name exec-name	Startet eine Node eines Packages
rosnode list	Listet alle Nodes auf
rosnode info node-name	Gibt Informationen zu einer Node aus
rosnode kill node-name	Beendet eine Node
	<i>Topics</i>
rostopic list	Listet alle Topics auf
rostopic echo topic-name	Gibt die Inhalte eines Topics aus
rostopic info topic-name	Gibt Informationen zu einem Topic aus
rosmsg show message-type-name	Zeigt den Aufbau der Message an
	<i>Services</i>
rosservice list	Listet alle Services auf
rosservice node service-name	Gibt die Node aus, die den Service anbietet
rosservice info service-name	Gibt Informationen zu einem Service aus
rossrv show service-data-type-name	Zeigt den Aufbau der Service-Message an
	<i>Launch-Files</i>
roslaunch filename	Führt das Launch-File aus
	<i>Bags</i>
rosbag record -a	Zeichnet alle Topics auf
rosbag play filename	Spielt das ROS-Bag wieder ab
rosbag info filename	Gibt Informationen zu einem ROS-Bag aus

Kapitel 3

Kartierungsalgorithmen

In diesem Kapitel wird auf die ROS-Packages für die Kartierung eingegangen und deren Unterschiede dargestellt. Dazu wird zunächst immer der Algorithmus und dann das Konzept der Integration in das Robot Operating System dargestellt.

Der Anwender sieht während der Algorithmus ausgeführt wird, die Karte, während diese erstellt wird, sofern nicht einfach nur ein Bag-File aufgezeichnet wird. Die Ausgabe ist eine 2D Grid Karte mit jeweils unterschiedlicher Auflösung.

3.1 cartographer

3.1.1 Algorithmus

Der Algorithmus basiert auf einem lokalen und einem globalen Ansatz. Beide Ansätze optimieren die Position der Scans, welche als Formel $\xi = (\xi_x, \xi_y, \xi_0)$ bestehend aus einer Translation mit ξ_x und ξ_y sowie einer Rotation ξ_0 , definiert sind. Es besteht die Möglichkeit, eine Inertial Measurement Unit (IMU) zu nutzen, welche hilft, die Scans besser auf die 2D-Welt zu projizieren.

Der lokale Ansatz nimmt die Aufzeichnungen des Laser-Scanners entgegen. Diese werden zu einer sogenannten Submap an der bestmöglichen geschätzten Position hinzugefügt. Eine Submap ist ein Teil der gesamten aufgenommenen Karte. Dieser Prozess wird Scan Matching genannt, und basiert darauf die Lage des Scans zu finden, der die Wahrscheinlichkeitsfunktion der Scanpunkte in der Submap maximiert.

3 Kartierungsalgorithmen

Es wird als Lösungsmethode die Methode der nichtlinearen kleinsten Quadrate genutzt.

$$\underset{\xi}{\operatorname{argmin}} \sum_{k=1}^K (1 - M_{smooth}(T_\xi h_k))^2 \quad (3.1)$$

T_ξ transformiert h_k vom Scan-Frame zum Sumap-Frame. $M_{smooth} : R^2 \rightarrow R$ bildet die Werte in der lokalen Submap ab. Es wird eine bikubische Interpolation genutzt. Diese sorgt dafür, dass hauptsächlich Werte im Bereich 0 bis 1 vorkommen. Werte außerhalb können auftreten, schaden aber dem Algorithmus nicht.

Das Scan Matching produziert kleine Fehler, die in der Summe große Auswirkungen haben können. Daher besteht das System nicht nur aus einem lokalen Ansatz, welcher auch Frontend genannt wird, sondern auch aus einem globalen Ansatz, der auch Backend genannt wird. Dieser optimiert die lokalen Berechnungen.

Der globale Ansatz wird in regelmäßigen Zeitabständen unabhängig vom lokalen Ansatz durchgeführt. Wenn eine Submap fertig ist, läuft ein sogenanntes Loop Closing ab. Dieser Prozess fügt die einzelnen Submaps zu einer großen Karte zusammen. Dies erfolgt dadurch, dass nahe und passende Scans gesucht werden. Der Algorithmus hinter der Optimierung der Scans und Submaps nennt sich Sparse Pose Adjustment. Es handelt sich dabei wieder um ein Optimierungsproblem. Dieses ist wie folgt definiert:

$$\underset{\Xi^m, \Xi^s}{\operatorname{argmin}} \sum_{ij} \rho(\xi_i^m, \xi_j^s; \Sigma_{ij}, \xi_{ij}) \quad (3.2)$$

Dabei wird die Lage der Submaps $\Xi^m = \{\xi_i^m\}_{i=1,\dots,m}$, sowie die Lage der Scans $\Xi^s = \{\xi_j^s\}_{j=1,\dots,n}$ mit einigen Randbedingungen wie Σ_{ij} und ξ_{ij} optimiert. Die Verlustfunktion ρ soll den Einfluss von Ausreißern verringern.

Insgesamt muss das Loop Closing schneller sein als neue Scans hinzugefügt werden. Durch diesen Prozess werden Loops geschlossen, wenn man einen Ort wiederbesucht. Dazu wird unter anderem ein Branch-and-bound Scan Matching verwendet. [6]

Tabelle 3.1: Topic-Abonnements des Packages Cartographer

Topic	Topic-Typ	Optional
scan	sensor_msgs/LaserScan	Nein
echoes	sensor_msgs/MultiEchoLaserScan	Nein
points2	sensor_msgs/PointCloud2	Nein
imu	sensor_msgs/Imu	Ja
odom	nav_msgs/Odometry	Ja

Tabelle 3.2: Topic-Veröffentlichungen des Packages Cartographer

Topic	Topic-Typ
scan_matched_points2	sensor_msgs/PointCloud2
submap_list	cartographer_ros_msgs/SubmapList

3.1.2 Verwendung im Robot Operating System

Damit der Cartographer eine Karte im ROS erstellen kann, müssen zunächst einmal zwei Nodes gestartet werden. Die erste Node ist die Cartographer Node, welche für die simultane Positions- und Kartenerstellung genutzt wird. Dieser Node erwartet eine Liste von Topics, die sie abonniert. (siehe Tabelle 3.1). Der Cartographer publiziert das Ergebnis seiner Berechnungen dann in weiteren Topics (siehe Tabelle 3.2).

Die zweite Node ist die Occupancy grid Node, welche die Submaps über das Topic submap_list verarbeitet und daraus eine Karte erstellt. Das Format in ROS ist das nav_msgs/OccupancyGrid-Format.

Des weiteren benötigt dieser Prozess für jede Sensorkomponente eine Transformation des TF2-Packages. Für jede Komponente sollte entweder ein robot_state_publisher oder ein static_transform_publisher auf das tracking_frame sowie das published_frame, das in der Konfigurationsdatei festgelegt wurde, gesetzt werden. Ein Beispiel dafür ist eine Transformation von der Basis (alias base_link) zum Laserscanner (alias scan).

Zusätzlich stellt der Cartographer auch Transformationen zur Verfügung. Diese sind je nach Konfiguration die Transformationen von der Karte (alias map) zur Odometrie (alias odom) und zur Basis (alias base_link). [7]

3.2 hector_mapping

3.2.1 Algorithmus

Im Gegensatz zum Cartographer basiert der SLAM-Algorithmus hector_mapping nicht auf der gängigen Konstellation eines Frontends und eines Backends. Das hector_mapping bietet nur ein Frontend an, welches die typischen Aufgaben ausführt. Dem entsprechend ist der Algorithmus auch nicht für große Karten ausgelegt, bei denen große Loops geschlossen werden müssen. Das Loop-Closing ist nicht explizit implementiert. Besser ist daher die Anwendung in kleinen Karten wie beispielsweise dem RoboCup Rescue, bei dem bestimmte Ziele gerettet werden müssen. Dabei ist die hohe Frequenz des Laserscanners wichtig, damit trotz der Bewegung Ziele auch auf unebenen Ebenen schnell erkannt werden können.

Das hector_mapping nutzt ausschließlich die Daten des Laser-Scanners als Eingabequelle. Es werden keine zusätzlichen Odometrie-Daten oder IMU-Daten genutzt. Der Prozess besteht aus drei Schritten:

1. Preprocessing
2. Scan Matching
3. Mapping

Zunächst kommen die Scans im System an und werden zu Point-Clouds weiterverarbeitet, in dem die aktuelle Lage des Roboters und die Daten des Laser-Scanners gemeinsam verarbeitet werden. Es besteht die Möglichkeit, die Liste der Scans vorher zu filtern, um beispielsweise Ausreißer zu entfernen oder um auf Grund der Menge der Datensätze nur einige davon zu betrachten.

Beim Scan Matching werden dann die Scans der Map zugeordnet. Dabei werden beim hector_mapping die Endpunkte eines Scans mit Hilfe der bisher aufgenommenen Karte optimiert. Dabei kommt das Gauß-Newton-Verfahren inspiriert aus der Computer Vision zum Einsatz. Es wird eine Transformation für den Scan gesucht, die am besten zur aktuellen Karte passt. Es handelt sich wie auch beim Cartographer um ein Minimierungsproblem:

$$\underset{\xi}{\operatorname{argmin}} \sum_{i=1}^n [1 - M(S_i(\xi))]^2 \quad (3.3)$$

Tabelle 3.3: Topic-Abonnements des Packages hector_mapping

Topic	Topic-Typ	Optional
scan	sensor_msgs/LaserScan	Nein
syscommand	std_msgs/String	Ja

Tabelle 3.4: Topic-Veröffentlichungen des Packages hector_mapping

Topic	Topic-Typ
map_metadata	nav_msgs/MapMetaData
map	nav_msgs/OccupancyGrid
slam_out_pose	geometry_msgs/PoseStamped
poseupdate	geometry_msgs/PoseWithCovarianceStamped

Die Funktion M gibt die Koordinate auf der Karte zurück. Die innere Funktion $S_i(\xi)$ gibt die Koordinaten der Scans. Dieser haben als Parameter die Variable ξ , welche die Lage des Roboters ist.

$$S_i(\xi) = \begin{pmatrix} \cos(\psi) & -\sin(\psi) \\ \sin(\psi) & \cos(\psi) \end{pmatrix} \begin{pmatrix} s_{i,x} \\ s_{i,y} \end{pmatrix} + \begin{pmatrix} p_x \\ p_y \end{pmatrix} \quad (3.4)$$

Das Ziel ist es nun zu berechnen, was passiert, wenn diese 2D-Rotationsmatrix gegen null geht, sprich das globale Minimum zu finden. Des Weiteren wird ein Startwert für ξ dazu gegeben.

$$\sum_{i=1}^n [1 - M(S_i(\xi + \Delta\xi))]^2 \rightarrow 0 \quad (3.5)$$

Jeder Bergsteigeralgorithmus sowie jedes Gradientenabstiegsverfahren kann in einem lokalen Minimum hängen bleiben. Daher wird beim hector_mapping, um dem entgegenzuwirken, eine Multi-Resolution Map Representation genutzt. [8]

3.2.2 Verwendung im Robot Operating System

Damit das Package hector_mapping eine Karte im ROS erstellen kann, muss ebenfalls nur eine einzige Node, die hector_mapping-Node, gestartet werden. Diese benötigt einige Topics (siehe Tabelle 3.3). Das Ergebnis der Berechnungen publiziert die Node dann in den folgenden Topics (siehe Tabelle 3.4).

Das `hector_mapping` benötigt exakt eine TF-Transformation. Diese ist die Transformation vom Frame, welches zu den eingehenden Scan gehört, zur Basis (alias `base_link`).

So wie alle anderen Packages stellt das `hector_mapping` ebenfalls eine TF-Transformation zur Verfügung. Dies ist die Transformationen von der Karte (alias `map`) zur Odometrie (alias `odom`). [9]

3.3 gmapping

3.3.1 Algorithmus

Gmapping ist ein weiterer Kartierungsalgorithmus. Im Gegensatz zum Cartographer und zum `hector_mapping` nutzt Gmapping einen Partikelfilter für das Simultaneous Localization and Mapping (SLAM)-Problem. Dieser ist der Rao Blackwellized Particle Filter.

Dieser Filter basiert darauf die A-posteriori-Wahrscheinlichkeit $p(x_{1:t}|z_{1:t}, u_{0:t})$ zu berechnen. Dabei ist p die aktualisierte Wahrscheinlichkeit, ob das Feld auf der Karte gesetzt ist, z sind die Scans und u sind die Odometrie-Daten. Dies wird genutzt, um die A-posteriori-Wahrscheinlichkeit über die Karte und die Trajektorie zu berechnen.

$$p(x_{1:t}, m|z_{1:t}, u_{0:t}) = p(m|x_{1:t}, z_{1:t})p(x_{1:t}|z_{1:t}, u_{0:t}) \quad (3.6)$$

Dies lässt sich für die A-posteriori-Wahrscheinlichkeit über die Karte $p(m|x_{1:t}, z_{1:t})$ berechnen, wenn $x_{1:t}$ und $z_{1:t}$ gegeben sind. Um die A-posteriori-Wahrscheinlichkeit für die Trajektorie $p(x_{1:t}|z_{1:t}, u_{0:t})$ zu berechnen, nutzt der Algorithmus einen weiteren Partikelfilter.

Die Karte wird durch die Scans $z_{1:t}$, sowie die Trajektorie $x_{1:t}$ zusammengestellt. Es wird ein Rao-Blackwellized Sampling Importance Resampling (SIR)-Filter genutzt, der die Scans und die Odometrie-Daten inkrementell zu einer Karte zusammenfügt. Die Odometrie-Daten werden genutzt, sofern sie vorhanden sind. Der Prozess läuft im Allgemeinen wie folgt ab:

1. *Sampling*: Über die Verteilung $\pi(x_t|z_{1:t}, u_{0:t})$ werden über die aktuellen Partikel die neuen Partikel entnommen.

Tabelle 3.5: Topic-Abonnements des Packages gmapping

Topic	Topic-Typ	Optional
scan	sensor_msgs/LaserScan	Nein

Tabelle 3.6: Topic-Veröffentlichungen des Packages gmapping

Topic	Topic-Typ
map_metadata	nav_msgs/MapMetaData
map	nav_msgs/OccupancyGrid
entropy	std_msgs/Float64

2. *Importance Weighting*: Jeder Partikel erhält eine Gewichtung.
3. *Resampling*: Partikel mit einer geringen Gewichtung werden durch Partikel mit einer höheren Gewichtung ausgetauscht.
4. *Map Estimation*: Für jeden Partikel wird die dazugehörige Schätzung auf der Karte berechnet.

Für den gmapping-Algorithmus wird zusätzlich zu dem genannten Ablauf noch eine verbesserte Verteilung berechnet. Außerdem wird ein selektives Resampling genutzt. [10] [11]

3.3.2 Verwendung im Robot Operating System

Damit das Package gmapping eine Karte im ROS erstellen kann, muss nur eine einzige Node, die `slam_gmapping`-Node, gestartet werden. Diese benötigt einige Topics (siehe Tabelle 3.5). Das Ergebnis der Berechnungen publiziert die Node dann in den folgenden Topics (siehe Tabelle 3.6).

Gmapping benötigt ebenfalls noch einige TF-Transformationen. Dieses ist einmal eine Transformation vom Frame, welches zu den eingehenden Scan gehört, zur Basis (alias `base_link`). Des weiteren wird eine Transformation von der Basis (`base_link`) zur Odometrie (alias `odom`) benötigt.

Auch Gmapping stellt eine TF-Transformation zur Verfügung. Dies ist die Transformationen von der Karte (alias `map`) zur Odometrie (alias `odom`). [12]

Kapitel 4

Integration der Algorithmen

In diesem Kapitel werden die zuvor beschriebenen Kartierungsalgorithmen in das eigene Package integriert. Dazu werden die Algorithmen als ROS-Packages installiert, durch das eigene Package konfiguriert und mittels eines Launch-Files gestartet.

Das Ziel jeder Integration in das Package ist das Online-SLAM, bei dem die Karte während der Aufnahme erstellt wird, als auch das Offline-SLAM, bei dem die Karte erst nach der Aufnahme berechnet wird.

4.1 Aufsetzen des eigenen Packages

Um den Roboter nun nutzen zu können, muss dieser mit dem Robot Operating System verbunden werden. Dazu werden einige Packages benötigt, die in einem selbst entwickelten Package zusammengefasst werden.

- *p2os*: Dieses Paket stellt die Schnittstelle zur Hardware des Pioneer 3-DX zur Verfügung. Das Package bietet daneben noch die Steuerung mittels Tastatur oder Joystick.
- *sicktoolbox_wrapper*: Dieses Paket startet den Laserscanner Sick LMS und sendet die Scan-Daten an einen Topic.
- *ROS-tinkerforge_sensors*: Dieses Paket startet die Inertial Measurement Unit Tinkerforge Brick 2.0 und sendet die Beschleunigungsdaten an einen Topic.

Das eigene Package hat den Namen `pioneer3dx_cartographer` und ist in folgende Ordner eingeteilt:

- *launch*: Dieser Ordner beinhaltet alle Launch-Files.
- *defs*: Dieser Ordner stellt das Xacro-Modell des Pioneers zur Verfügung.
- *meshes*: Dieser Order stellt die geometrischen Informationen des Pioneers als STL-Dateien zur Verfügung.
- *configuration_files*: Dieser Order speichert Konfigurationen für den RVIZ oder für Algorithmen.

Im ersten Schritt ist im Launch-Ordner die Datei `pioneer.launch` von Bedeutung, da dieses den Motor, den Laserscanner und die Intertial Measurement Unit des Pioneers, sowie das Xacro-Modell und die benötigten Transformationen startet bzw. lädt. Im weiteren Verlauf des Kapitels wird der Ordner um weitere Launch-Files erweitert, die die einzelnen Kartierungsalgorithmen starten.

Im nächsten Ordner liegt das Modell des Pioneers im Format XML Macros (xacro). Dieses wird zur Laufzeit in das Unified Robot Description Format (URDF) umgewandelt. Ein Modell besteht immer aus mehreren Links und Joints. Ein Link beschreibt einen Teil des Roboters, der Informationen über seine Trägheit und über das Aussehen definiert. Ebenfalls wird in einem Link ein Kollisionsmodell definiert. Links werden über Joints miteinander verbunden. Beispiele für einen Link sind die Räder oder die Basis des Roboters. Das Aussehen eines Links wird über geometrische Informationen im STL-Format definiert. Die STL-Dateien eines jeden Links liegen im Folder `meshes`. [13] [14]

Das Modell im xacro-Format lässt sich manuell in das urdf-Format umwandeln. Danach kann eine Grafik der Links und Joints per Kommando erstellt werden.

Für den Pioneer 3-DX ergibt sich dann die folgende Baumansicht in Abbildung 4.1 nach dem Ausführen von Listing 4.1.

```
rosrun xacro xacro -o pioneer3d.urdf pioneer3dx.xacro  
urdf_to_graphviz pioneer3d.urdf
```

Listing 4.1: Umwandlung von xacro in urdf und grafische Darstellung

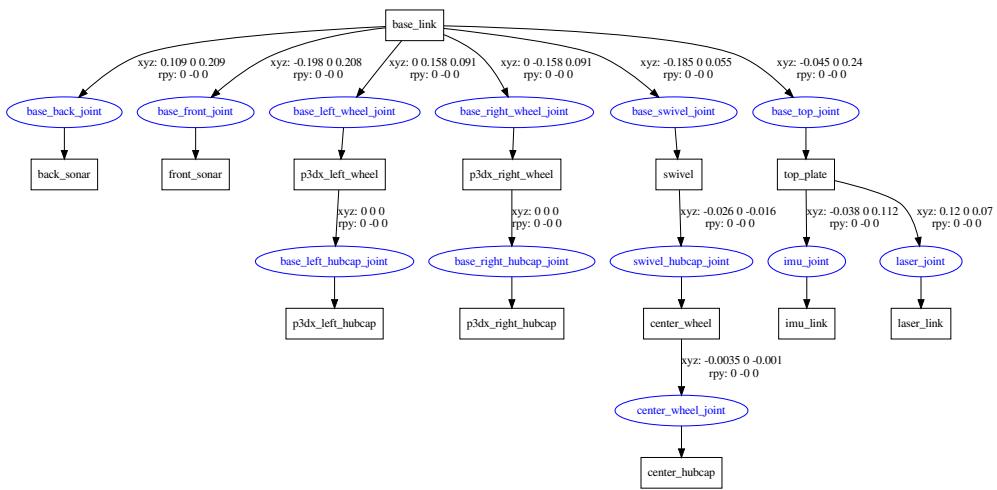


Abbildung 4.1: Pioneer 3-DX URDF-Modell

4.2 Installation der Kartierungsalgorithmen

Um das eigenen Package um die Kartierungsalgorithmen zu erweitern, müssen die Packages für die Kartierung installiert werden.

4.2.1 cartographer

Für die Installation des Cartographers müssen zunächst einmal die Paketquellen aktualisiert werden. Dann wird das Tool wstool sowie rosdep und Ninja installiert. Danach wird zum Catkin-Workspace gewechselt und einige weitere Installations-schritte mittels dem Tool wstool eingeleitet. Ist dieser Installationsschritt ausge-führt, muss jetzt mittels rosdep noch die Abhängigkeit hinzugefügt werden. Zuletzt muss der Build gestartet werden und der Cartographer installiert werden. Dabei wird das Tool Ninja genutzt, um den Build-Prozess zu verschnellern. [15]

4.2.2 hector_mapping

Die Installation des Packages hector_mapping gestaltet sich wesentlich einfacher als die des Packages cartographer, da dieses manuell gebaut werden musste. Bei diesem Package wird lediglich der ROS-Befehl zur Installation ausgeführt, wie in Listing 4.3 dargestellt.

```
sudo apt-get update
sudo apt-get install -y python-wstool python-rosdep ninja-build

wstool init src
wstool merge -t src https://raw.githubusercontent.com/googlecartographer/
    cartographer_ros/master/cartographer_ros.rosinstall
wstool update -t src

src/cartographer/scripts/install_proto3.sh
sudo rosdep init
rosdep update
rosdep install --from-paths src --ignore-src --rosdistro=${ROS_DISTRO} -y

catkin_make_isolated --install --use-ninja
```

Listing 4.2: Installation des Packages Cartographer

```
sudo apt-get install ros-indigo-hector-slam
```

Listing 4.3: Installation des Packages hector_mapping

4.2.3 gmapping

Die Installation des Packages gmapping gestaltet sich ebenso einfach wie die des Packages hector_mapping. Bei diesem Package wird wieder lediglich der ROS-Befehl zur Installation ausgeführt, wie in Listing 4.4 dargestellt.

4.3 Aufsetzen der Launch-Files und der Konfiguration

Im nächsten Schritt werden die Launch-Files sowie damit auch die Konfiguration der Kartierungsalgorithmen aufgesetzt. Dabei muss berücksichtigt werden, dass für jeden Algorithmus andere ROS-Nodes gestartet und andere Anforderungen an das Transformations-Modul TF erfüllt werden müssen, wie in Unterabschnitt 3.1.2, Unterabschnitt 3.2.2 sowie in Unterabschnitt 3.3.2 beschrieben. Pro Algorithmus wird sowohl ein Launch-File für die Online-Kartierung als auch eines für die Offline-Kartierung präsentiert. Bei ersterem wird die Karte während der Aufnahme erstellt. Bei zweiterem wird erst nach der Aufnahme das Bag-File an das Launch-File gegeben und somit wird erst im Nachhinein die Karte erstellt.

```
sudo apt-get install ros-melodic-slam-gmapping
```

Listing 4.4: Installation des Packages gmapping

Allgemein enthalten alle Launch-Files zusätzlich den Startbefehl für den RVIZ, welcher als Argument eine Konfigurationsdatei enthält. Die Konfigurationsdatei beschreibt die Oberfläche des RVIZ.

4.3.1 cartographer

Der Cartographer definiert für das Online-Kartieren in Listing 4.8 eine Node, welche den Pioneer startet sowie zwei spezifische Nodes für den Cartographer. Die erste Node erhält als Argument eine Konfigurationsdatei, siehe Listing 4.9, mit Optionen zum Customizing der Cartographer-Node. Der zweiten Node wird lediglich die Auflösung der resultierenden Karte mitgegeben.

Die Konfigurationsdatei in Listing 4.9 definiert die genutzten Frames und Nodes, die Frequenz sowie das Sampling von Daten. Dabei waren einige vom Standard abweichende Anpassungen nötig. Wenn eine IMU genutzt werden soll, muss das Attribut TRAJECTORY_BUILDER_2D.use_imu_data gesetzt werden. Die Anzahl der Laserscanner ist durch das Attribut num_laser_scans auf 1 gesetzt. Da 2D-Karten erstellt werden, muss das Attribut publish_frame_projected_to_2d aktiviert sein.

Das Launch-File für das Offline-Kartieren wird in Listing 4.10 gezeigt. Unterschiedlich zu Listing 4.8 ist, dass der Pioneer nicht mehr gestartet werden muss, sondern stattdessen beim Starten an das ROS-Bagfile als Argument an das Launch-File übergeben wird, welches beim Start abgespielt wird.

4.3.2 hector_mapping

Auch beim hector_mapping in Listing 4.11 wird zunächst die selbe Node wie beim Cartographer definiert, die den Pioneer startet. In den weiteren Zeilen werden einige Konfigurationswerte definiert. Danach wird nur eine Node gestartet, welche diese Werte als Parameter nutzt. Die Konfiguration erfolgt also innerhalb des Launch-Files. In der Konfiguration werden unter anderem die Frames, die Auflösung, Größe und Position der Karte, einige Intervalle sowie der Topic für den Scan definiert.

Das Offline-Kartieren in Listing 4.12 erfolgt ähnlich wie das Online-Kartieren. Wie auch beim Cartographer muss das Launch-File des Pioneers nicht mehr gestartet werden. Stattdessen werden wieder die nötigen Nodes für das Einlesen des ROS-Bagfiles genutzt.

4.3.3 gmapping

Auch beim gmapping in Listing 4.13 wird zunächst die selbe Node wie beim Cartographer und beim hector_mapping definiert, die den Pioneer startet. In den weiteren Zeilen werden ebenfalls wie beim hector_mapping einige Konfigurationswerte definiert. Danach wird nur eine Node gestartet, welche diese Werte als Parameter nutzt. Die Konfiguration erfolgt also wieder innerhalb des Launch-Files. In der Konfiguration wird der Scan-Topic angegeben, sowie Parameter für die Maße und für das Erstellen der Karte.

Das Offline-Kartieren in Listing 4.14 erfolgt wieder ähnlich wie das Online-Kartieren. Wie auch bei den anderen Kartierungsalgorithmen muss das Launch-File des Pioneers nicht mehr gestartet werden, sondern stattdessen nur die nötigen Nodes für das Einlesen des ROS-Bagfiles.

4.4 Starten der Kartierung

Zum Starten der beschriebenen Online- oder Offline-Kartierungen, werden die gängigen ROS-Befehle genutzt. Um diese folgenden Befehle ausführen zu können, muss der Anwender sich im Ordner des ROS-Packages befinden.

4.4.1 Starten der Online-Kartierung

Die Online-Kartierung erfolgt immer über einen einzigen roslaunch-Befehl, welcher in Listing 4.5 dargestellt wird.

```
roslaunch launch/cartographer/online.launch  
roslaunch launch/hector_mapping/online.launch  
roslaunch launch/gmapping/online.launch
```

Listing 4.5: Online-Kartierung

```
roslaunch launch/pioneer.launch  
rosbag record scan -a
```

Listing 4.6: Aufnahme eines Bag-Files

4.4.2 Starten der Offline-Kartierung

Bei der Offline-Kartierung muss vorher ein Bag-File mit den gewünschten Topics aufgenommen werden, welches später einem Launch-File als Parameter übergeben wird. Dieses läuft wie in Listing 4.6 beschrieben ab. Dieses Beispiel nimmt alle verfügbaren Topics auf. Es muss darauf geachtet werden, dass in einem zweiten Terminal das Launch-File des Pioneer vorher gestartet ist, damit Aktionen mit dem Roboter ausgeführt werden können.

Nach der Aufnahme des Bag-Files, kann nun das eigentliche Erstellen der Karte ausgeführt werden. Dies läuft wieder über einen gängigen ROS-Launch-Befehl ab, welcher als Parameter den Dateinamen des Bags enthält, wie in Listing 4.7 beschrieben.

```
roslaunch launch/cartographer/offline.launch bag_filename:=/path/to/the.bag  
roslaunch launch/hector_mapping/offline.launch bag_filename:=/path/to/the.bag  
roslaunch launch/gmapping/offline.launch bag_filename:=/path/to/the.bag
```

Listing 4.7: Offline-Kartierung

```
<launch>
  <include file="$(find pioneer3dx_explorer)/launch/pioneer.launch"/>

  <node name="cartographer_node" pkg="cartographer_ros" type="cartographer_node"
    args="-configuration_directory $(find pioneer3dx_explorer)/
    configuration_files -configuration_basename pioneer.lua" output="screen" />

  <node name="cartographer_occupancy_grid_node" pkg="cartographer_ros" type="
    cartographer_occupancy_grid_node" args="-resolution 0.05" />

  <node name="rviz" pkg="rviz" type="rviz" required="true" args="-d $(find
    pioneer3dx_explorer)/configuration_files/pioneer.rviz" />
</launch>
```

Listing 4.8: Online Launch-File für das Package Cartographer

```
include "map_builder.lua"
include "trajectory_builder.lua"

options = {
    map_builder = MAP_BUILDER,
    trajectory_builder = TRAJECTORY_BUILDER,
    map_frame = "map",
    tracking_frame = "base_link",
    published_frame = "base_link",
    odom_frame = "odom",
    provide_odom_frame = true,
    publish_frame_projected_to_2d = true,
    use_odometry = false,
    use_nav_sat = false,
    use_landmarks = false,
    num_laser_scans = 1,
    num_multi_echo_laser_scans = 0,
    num_subdivisions_per_laser_scan = 10,
    num_point_clouds = 0,
    lookup_transform_timeout_sec = 0.2,
    submap_publish_period_sec = 0.3,
    pose_publish_period_sec = 5e-3,
    trajectory_publish_period_sec = 30e-3,
    rangefinder_sampling_ratio = 1.,
    odometry_sampling_ratio = 1.,
    fixed_frame_pose_sampling_ratio = 1.,
    imu_sampling_ratio = 1.,
    landmarks_sampling_ratio = 1.,
}
MAP_BUILDER.use_trajectory_builder_2d = true
TRAJECTORY_BUILDER_2D.num_accumulated_range_data = 10
TRAJECTORY_BUILDER_2D.use_imu_data = true

return options
```

Listing 4.9: Konfigurationsdatei für den Cartographer

```
<launch>
  <node name="cartographer_node" pkg="cartographer_ros" type="cartographer_node"
    args="-configuration_directory $(find pioneer3dx_explorer)/
      configuration_files -configuration_basename pioneer.lua" output="screen" />

  <node name="cartographer_occupancy_grid_node" pkg="cartographer_ros" type="
    cartographer_occupancy_grid_node" args="-resolution 0.05" />

  <node name="rviz" pkg="rviz" type="rviz" required="true" args="-d $(find
    pioneer3dx_explorer)/configuration_files/pioneer.rviz" />

  <param name="/use_sim_time" value="true" />

  <node name="playbag" pkg="rosbag" type="play" args="--clock $(arg bag_filename)"
    />
</launch>
```

Listing 4.10: Offline Launch-File für das Package Cartographer

```

<launch>
  <include file="$(find pioneer3dx_explorer)/launch/pioneer.launch"/>

  <arg name="tf_map_scanmatch_transform_frame_name" default="scanmatcher_frame"/>
  <arg name="base_frame" default="base_link"/>
  <arg name="odom_frame" default="odom"/>
  <arg name="pub_map_odom_transform" default="true"/>
  <arg name="scan_subscriber_queue_size" default="5"/>
  <arg name="scan_topic" default="scan"/>
  <arg name="map_size" default="2048"/>

  <node pkg="hector_mapping" type="hector_mapping" name="hector_mapping" output="screen">
    <param name="map_frame" value="map" />
    <param name="base_frame" value="$(arg base_frame)" />
    <param name="odom_frame" value="$(arg odom_frame)" />

    <param name="use_tf_scan_transformation" value="true"/>
    <param name="use_tf_pose_start_estimate" value="false"/>
    <param name="pub_map_odom_transform" value="$(arg pub_map_odom_transform)"/>

    <param name="map_resolution" value="0.050"/>
    <param name="map_size" value="$(arg map_size)"/>
    <param name="map_start_x" value="0.5"/>
    <param name="map_start_y" value="0.5" />
    <param name="map_multi_res_levels" value="2" />

    <param name="update_factor_free" value="0.4"/>
    <param name="update_factor_occupied" value="0.9" />
    <param name="map_update_distance_thresh" value="0.4"/>
    <param name="map_update_angle_thresh" value="0.06" />
    <param name="laser_z_min_value" value = "-1.0" />
    <param name="laser_z_max_value" value = "1.0" />

    <param name="advertise_map_service" value="true"/>

    <param name="scan_subscriber_queue_size" value="$(arg scan_subscriber_queue_size)"/>
    <param name="scan_topic" value="$(arg scan_topic)"/>

    <param name="tf_map_scanmatch_transform_frame_name" value="$(arg tf_map_scanmatch_transform_frame_name)" />
  </node>

  <node name="rviz" pkg="rviz" type="rviz" required="true" args="-d $(find pioneer3dx_explorer)/configuration_files/pioneer.rviz"/>
</launch>
```

Listing 4.11: Online Launch-File für das Package hector_mapping

```

<launch>
  <arg name="tf_map_scanmatch_transform_frame_name" default="scanmatcher_frame" />
  <arg name="base_frame" default="base_link" />
  <arg name="odom_frame" default="odom" />
  <arg name="pub_map_odom_transform" default="true" />
  <arg name="scan_subscriber_queue_size" default="5" />
  <arg name="scan_topic" default="scan" />
  <arg name="map_size" default="2048" />

  <node pkg="hector_mapping" type="hector_mapping" name="hector_mapping" output="screen">
    <param name="map_frame" value="map" />
    <param name="base_frame" value="$(arg base_frame)" />
    <param name="odom_frame" value="$(arg odom_frame)" />

    <param name="use_tf_scan_transformation" value="true" />
    <param name="use_tf_pose_start_estimate" value="false" />
    <param name="pub_map_odom_transform" value="$(arg pub_map_odom_transform)" />

    <param name="map_resolution" value="0.050" />
    <param name="map_size" value="$(arg map_size)" />
    <param name="map_start_x" value="0.5" />
    <param name="map_start_y" value="0.5" />
    <param name="map_multi_res_levels" value="2" />

    <param name="update_factor_free" value="0.4" />
    <param name="update_factor_occupied" value="0.9" />
    <param name="map_update_distance_thresh" value="0.4" />
    <param name="map_update_angle_thresh" value="0.06" />
    <param name="laser_z_min_value" value="-1.0" />
    <param name="laser_z_max_value" value="1.0" />

    <param name="advertise_map_service" value="true" />

    <param name="scan_subscriber_queue_size" value="$(arg scan_subscriber_queue_size)" />
    <param name="scan_topic" value="$(arg scan_topic)" />

    <param name="tf_map_scanmatch_transform_frame_name" value="$(arg tf_map_scanmatch_transform_frame_name)" />
  </node>

  <node name="rviz" pkg="rviz" type="rviz" required="true" args="-d $(find pioneer3dx_explorer)/configuration_files/pioneer.rviz" />

  <param name="/use_sim_time" value="true" />

  <node name="playbag" pkg="rosbag" type="play" args="--clock $(arg bag_filename)" />
</launch>
```

Listing 4.12: Offline Launch-File für das Package hector_mapping

```
<launch>
  <include file="$(find pioneer3dx_explorer)/launch/pioneer.launch"/>

  <node pkg="gmapping" type="slam_gmapping" name="slam_gmapping" output="screen"
    args="/scan">
    <param name="delta" type="double" value="0.05" />
    <param name="temporalUpdate" type="double" value="2.5" />
    <param name="xmin" type="double" value="-2" />
    <param name="xmax" type="double" value="2" />
    <param name="ymin" type="double" value="-2" />
    <param name="ymax" type="double" value="2" />
  </node>

  <node name="rviz" pkg="rviz" type="rviz" required="true" args="-d $(find
    pioneer3dx_explorer)/configuration_files/pioneer.rviz"/>
</launch>
```

Listing 4.13: Online Launch-File für das Package gmapping

```
<launch>
  <node pkg="gmapping" type="slam_gmapping" name="slam_gmapping" output="screen"
    args="/scan">
    <param name="delta" type="double" value="0.05" />
    <param name="temporalUpdate" type="double" value="2.5" />
    <param name="xmin" type="double" value="-2" />
    <param name="xmax" type="double" value="2" />
    <param name="ymin" type="double" value="-2" />
    <param name="ymax" type="double" value="2" />
  </node>

  <node name="rviz" pkg="rviz" type="rviz" required="true" args="-d $(find
    pioneer3dx_explorer)/configuration_files/pioneer.rviz" />

  <param name="/use_sim_time" value="true" />

  <node name="playbag" pkg="rosbag" type="play" args="--clock $(arg bag_filename)"
    />
</launch>
```

Listing 4.14: Offline Launch-File für das Package gmapping

Kapitel 5

Testen der Algorithmen

Im letzten Schritt werden die Kartierungsalgorithmen getestet. Dazu werden verschiedene Szenarien dargestellt. Für jedes Szenario werden Karten durch das Ausführen der Kartierungsalgorithmen erstellt.

Im ersten Szenario wurde aus Tischen ein Rechteck aufgebaut. Ein Tisch ist 1,75m lang. Demnach ist die Breite des Aufbaus 1,75m und die Länge 3,90m, da es auf beiden Seiten noch einen Offset von 20cm gibt. Der Pioneer soll in verschiedenen Arten dort außen herum fahren.

Die erste Aufgabe besteht darin, um das aufgestellte Rechteck zu fahren. Dabei wird möglichst gerade und in einer durchschnittlichen Geschwindigkeit gefahren. Die nächste Aufgabe baut auf der ersten Aufgabe auf. Der Unterschied ist, dass diesmal versucht wird, so schnell wie der Pioneer fahren kann, zu fahren. Die dritte Aufgabe baut ebenfalls auf der ersten Aufgabe auf. Diesmal darf nur rückwärts gefahren werden. In der vierten Aufgabe darf nur in Schlangenlinien gefahren werden.

Im zweiten Szenario soll der Pioneer durch die Vordertür einen Raum betreten und diesen durch die Hintertür wieder verlassen. Das Ziel ist es, auf einer größeren Fläche den Kreis von Scans wieder zu schließen. Die Aufgabe besteht darin, in einer durchschnittlichen Geschwindigkeit durchzufahren und wieder am Endpunkt anzukommen.

Für die Durchführung wird für jede Aufnahme ein Bag-File aufgenommen. Dieses wird beim Auswerten für jeden Algorithmus abgespielt. Dafür dienen die Offline-Launch-Files, welche in Kapitel 4 definiert sind.



Abbildung 5.1: Szenario-Aufbau: Um ein Rechteck fahren

5.1 cartographer

Der Cartographer konnte die Karten zunächst nur ziemlich ungenau zeichnen, da es Probleme mit der Positions- und Drehungsbestimmung des Roboters gab. Der Algorithmus hat diesen Wert falsch geschätzt.

Durch das Aktivieren des *use_online_correlative_scan_matching* und dem Setzen von *TRAJECTORY_BUILDER_2D.ceres_scan_matcher.translation_weight* auf 10 sowie *TRAJECTORY_BUILDER_2D.ceres_scan_matcher.rotation_weight* auf 2 konnten gute Ergebnisse erzielt werden.

Das Resultat des ersten Szenarios findet man in Abbildung 5.2. Die Teilabbildung Abbildung 5.2a zeigt die normale Auswertung, bei der das Rechteck ziemlich genau dargestellt wird. Auch die Aufgabe in Abbildung 5.2b, rückwärts um das Rechteck zu fahren, wird sehr präzise dargestellt. Bei der schnellen Fahrt in Abbildung 5.2c ist das Rechteck zu erkennen. Dieses ist allerdings nicht exakt gerade gezeichnet. Gründe dafür könnten sein, dass der Algorithmus zu wenige Daten erhalten hat oder durch die schnellen Bewegungen Ungenauigkeiten in der Odometrie entstanden sind. In Abbildung 5.2d wurden Schlangenlinien um das Rechteck gefahren. Dennoch ist das Rechteck gut zu erkennen. Lediglich hat im unteren Teil ein Scan das Rechteck geschnitten. Da dort die Aufnahme geendet hat, hatte der Cartographer keine Zeit mehr, dieses durch weitere Scans zu optimieren. Insgesamt haben

alle Auswertungen bis auf die Auswertung Abbildung 5.2c, bei der schnell gefahren wurde, den gesamten Umriss des Raums gut erkannt.

In Abbildung 5.3 wurde eine IMU verwendet, um die Auswertung zu verbessern. Dabei gab es Probleme mit der Kalibrierung und der Integration in das eigene ROS-Modul, so dass die Auswertung dadurch eher ungenauer wurde. Die IMU wurde durch die Konfiguration *TRAJECTORY_BUILDER_2D.use_imu_data* aktiviert.

Im zweiten Szenario in Abbildung 5.4 konnte der Cartographer ohne IMU die Scans wieder zusammenfinden und eine ziemlich genaue Karte erstellen.

5.2 gmapping

Bei gmapping waren keine Anpassungen nötig, um vernünftige Karten erstellen zu können. In Abbildung 5.5 werden die Ergebnisse präsentiert. Die Auswertung Abbildung 5.5a ist gut gelungen, da das Rechteck gut erkennbar ist. Beim Rückwärtsfahren in Auswertung Abbildung 5.5b ist sowohl der Raum als auch das Rechteck ungenau gezeichnet. Beim Schnellfahren in Abbildung 5.5c ist das Rechteck wieder präzise gezeichnet. Es fehlen dort lediglich einige Messwerte, um den Raum auszufüllen. Auch beim Fahren in Schlangenlinien in Abbildung 5.5d ist gmapping stabil und produziert ein präzises Resultat. Das gmapping hat außer beim Rückwärtsfahren immer präzise funktioniert.

Auch beim zweiten Szenario in Abbildung 5.6 produziert der Algorithmus ein gutes Ergebnis, da die Scans nach dem Zurückkehren guten zusammengefunden wurden.

5.3 hector_mapping

Auch beim hector_mapping waren keine Anpassungen nötig, um vernünftige Karten erstellen zu können. Die Karte in Abbildung 5.7a hat wie in den vorherigen Algorithmen ebenfalls gute Ergebnisse erzielt. Beim Rückwärtsfahren in Karte Abbildung 5.7b ist der linke Teil des Rechtecks verrutscht. Ansonsten ist diese Karte gut gelungen. Kaum gut gelungen sind die Karten, die durch Schnellfahren oder durch Schlangenlinien in Abbildung 5.7c und Abbildung 5.7d erzeugt wurden. Bei diesen ist gar kein Rechteck erkennbar und diese sind ziemlich verrutscht. Das hector_mapping hat hier also nur gut funktioniert, wenn vorwärts oder rückwärts in

einer normalen Geschwindigkeit gefahren wurde. Zu viele oder zu schnelle Kurven haben dagegen nicht funktioniert.

Auch im zweiten Szenario in Abbildung 5.8 haben zu viele Kurven das Ergebnis ungenau gemacht. Der Rest der Karte ist ansonsten präzise.

5.4 Vergleich

Tabelle 5.1 stellt dar, wie die einzelnen Kartierungsalgorithmen nach einigen Kriterien abgeschlossen haben.

Zunächst einmal sind die Konfigurationsmöglichkeiten der Algorithmen von Bedeutung. Da bietet der Cartographer eine große Auswahl mit einer guten Dokumentation. Die anderen Algorithmen lassen sich auch konfigurieren, allerdings ist mit dem Cartographer einfach viel mehr möglich, wie zum Beispiel die 3D-Kartierung oder das Verwenden einer IMU.

Alle Algorithmen konnten den ersten Test problemlos durchführen. Probleme hingegen gab es beim Rückwärts-, Schnell oder Schlangenlinien fahren. Insgesamt schnitt der Cartographer am besten ab, da dieser alle Tests nach einigen Optimierungen der Konfiguration bestanden hat. Der gmapping-Algorithmus hatte Probleme beim Rückwärtsfahren und der hector_mapping-Algorithmus konnte die Kartierung für eine schnelle Fahrt und eine Fahrt in Schlangenlinien nicht gut durchführen.

Der Test für die IMU schlug zwar auf dem Cartographer auf Grund von einer falschen Kalibrierung und einem fehlerhaften ROS-Package fehl, dennoch ist es positiv, dass der Algorithmus diese Option als einziger anbietet.

Tabelle 5.1: Vergleich der Kartierungsalgorithmen

	Cartographer	gmapping	hector_mapping
Konfigurationsmöglichkeiten	●	○	○
Normal fahren	●	●	●
Rückwärts fahren	●	○	●
Schnell fahren	●	●	○
Schlangenlinien fahren	●	●	○
IMU	○	-	-

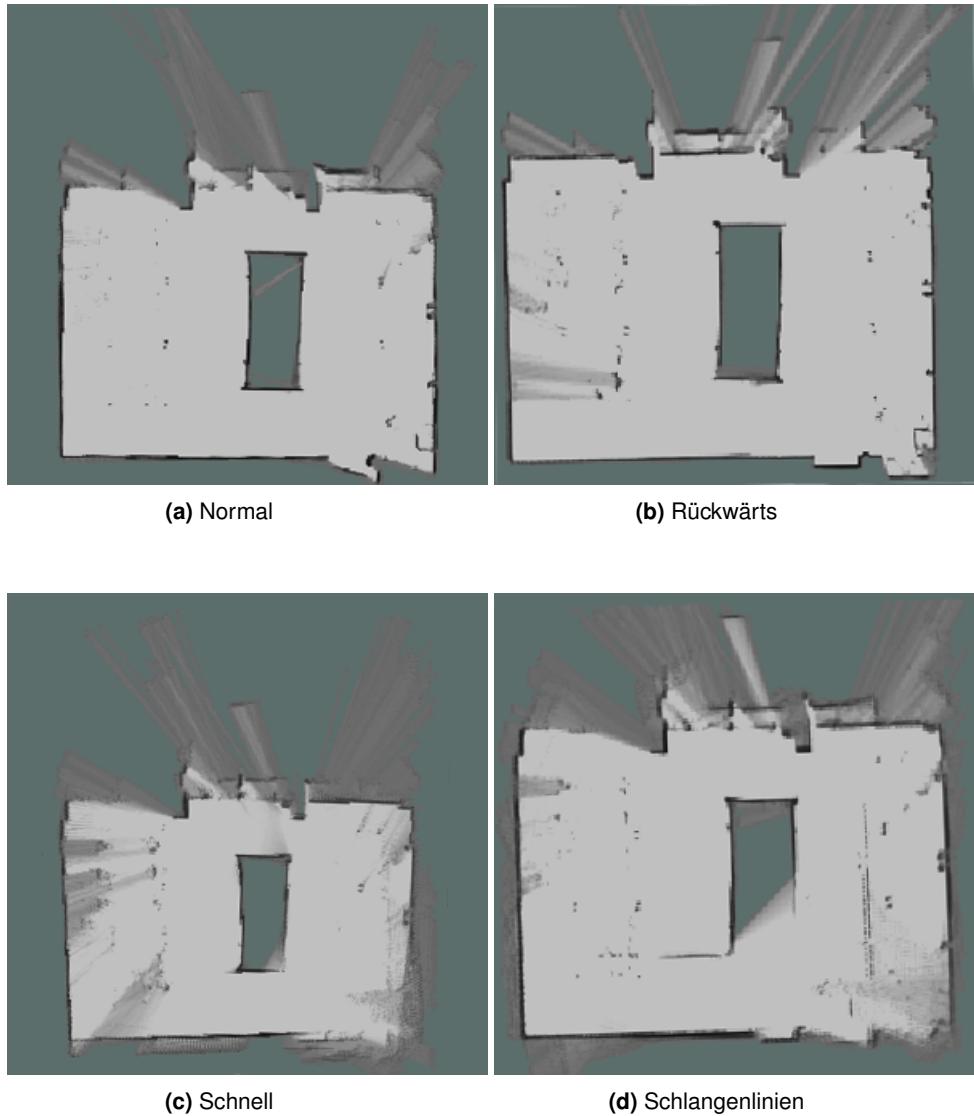


Abbildung 5.2: cartographer: Resultat der Aufnahmen des ersten Szenarios

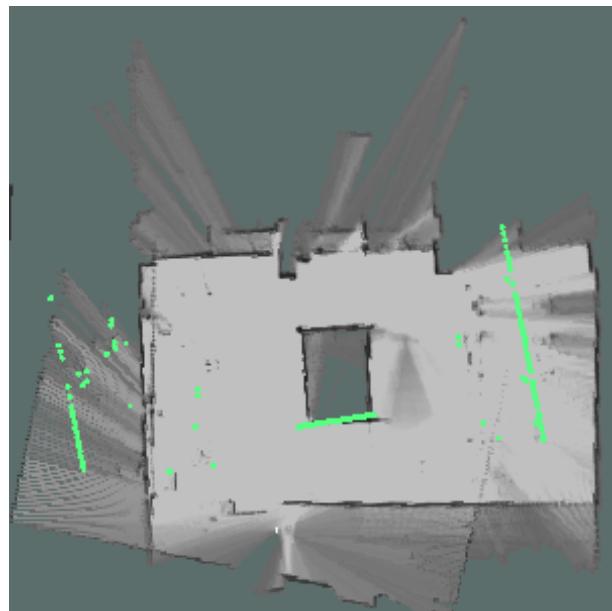


Abbildung 5.3: cartographer: Resultat der Aufnahmen des ersten Szenarios mit IMU

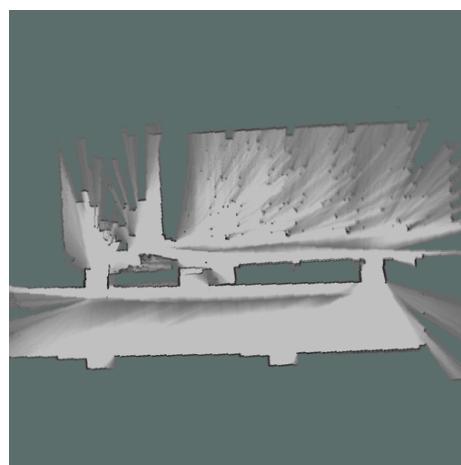


Abbildung 5.4: cartographer: Resultat der Aufnahmen des zweiten Szenarios

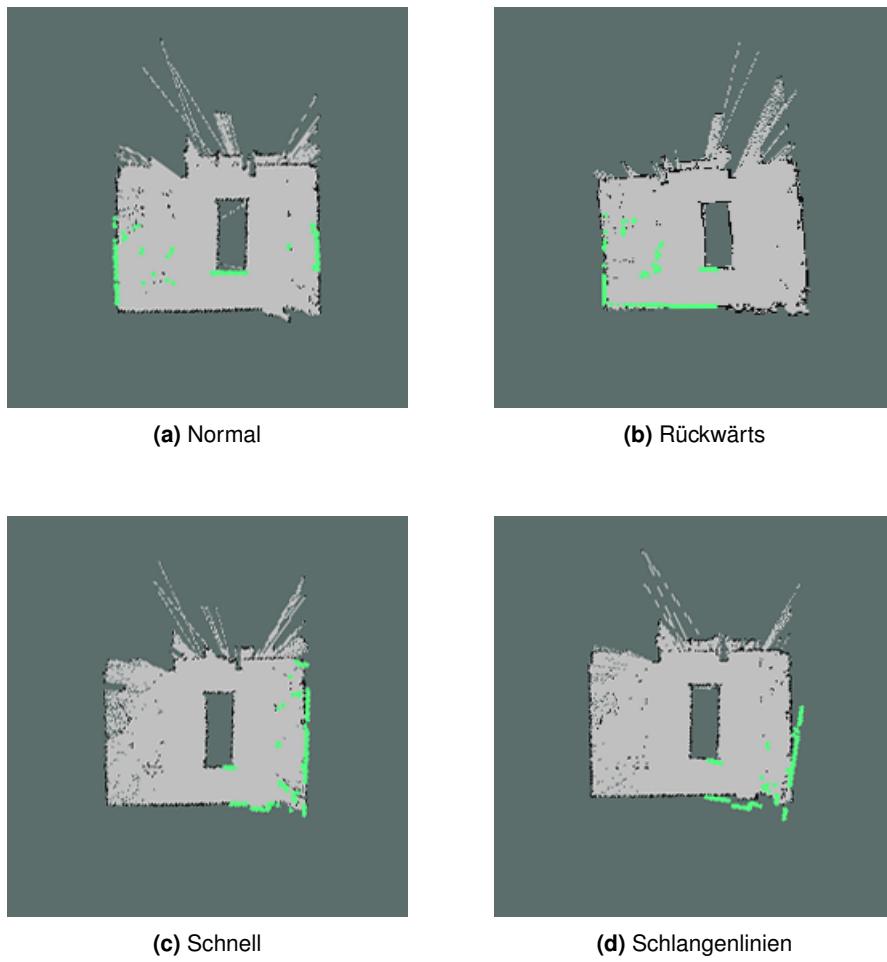


Abbildung 5.5: gmapping: Resultat der Aufnahmen des ersten Szenarios

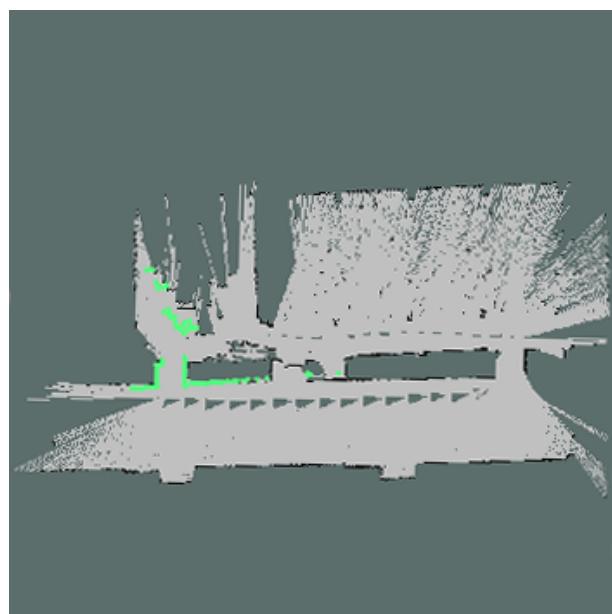


Abbildung 5.6: gmapping: Resultat der Aufnahmen des zweiten Szenarios

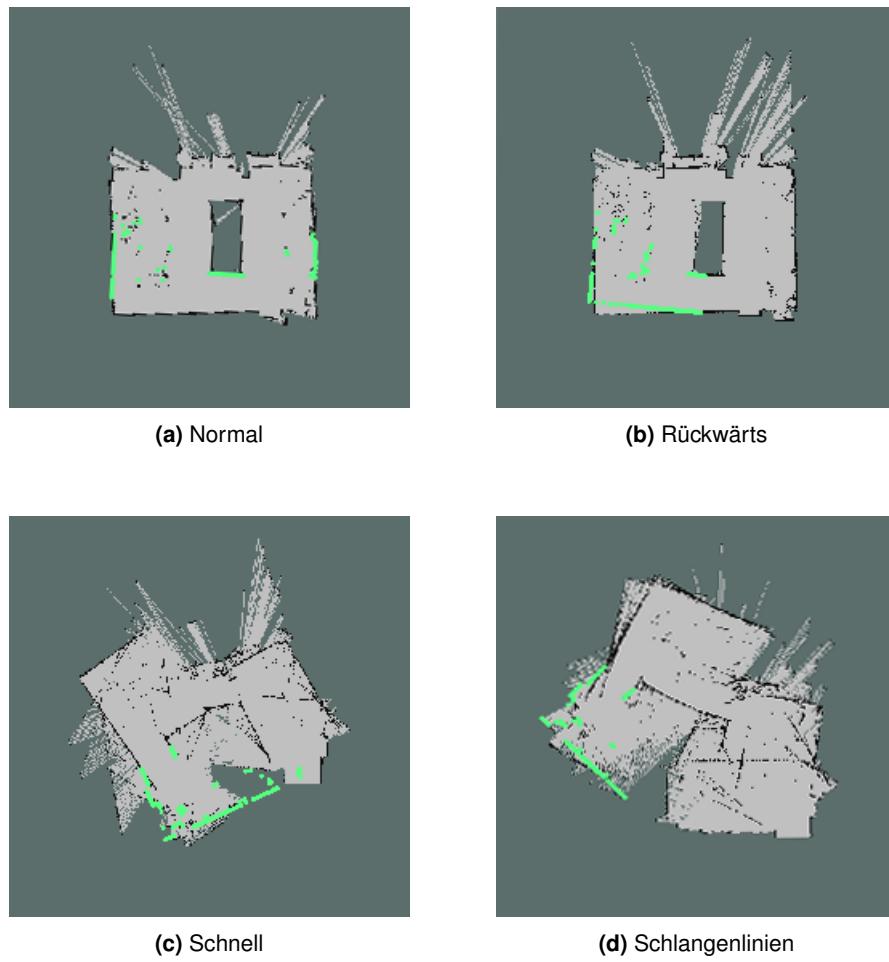


Abbildung 5.7: hector_mapping: Resultat der Aufnahmen des ersten Szenarios



Abbildung 5.8: hector_mapping: Resultat der Aufnahmen des zweiten Szenarios

Kapitel 6

Zusammenfassung

Das Ziel der Arbeit war es herauszuarbeiten, welche Kartierungsalgorithmen existieren und wie sich diese für den Pioneer 3-DX Roboter in das ROS integrieren, konfigurieren, starten und verbessern lassen.

In Kapitel 3 wurden die Funktionsweise sowie die Verwendung im ROS der einzelnen Kartierungsalgorithmen vorgestellt.

In Kapitel 4 wurden das eigene ROS-Package vorgestellt, welches die Launch-Files und Konfigurationen der drei Kartierungsalgorithmen speichert. Außerdem wurde hier dargestellt, wie die Packages installiert werden und wie die Kartierung sowohl online als auch offline gestartet werden kann.

In Kapitel 5 wurden dann Tests nach zwei Testszenarien für die Kartierungsalgorithmen Cartographer, gmapping und hector_mapping durchgeführt sowie ausgewertet. Dabei haben sich gewisse Unterschiede zwischen den Algorithmen herausarbeiten lassen. Es hat sich herausgestellt, dass der Cartographer für die ausgewählten Szenarien auf Grund der Konfigurationsmöglichkeiten und der guten Kartierung am besten abgeschnitten hat.

Kapitel 7

Ausblick

Weiterführend hätte man noch das Thema der 2½D-Kartierung aufgreifen können, bei der Elemente wie Treppenabgänge sowie Geländer berücksichtigt werden. Im nächsten Schritt wäre auch eine 3D-Kartierung von Bedeutung, welche es ermöglicht, mehr als ein Stockwerk zu kartieren. Interessant wäre es hierbei mit einem Höhensensor zu messen, in welchem Stockwerk der Roboter sich befindet oder allgemein nicht nur eine 2D-Karte aufzunehmen, sondern durch einen bewegbaren Laserscanner, welcher auch Höhendaten liefern kann, eine Karte zu erstellen. Dafür würde sich beispielsweise der Cartographer gut eignen, da dieser auch einige 3D-Optionen anbietet und grundsätzlich sehr viele Konfigurationsmöglichkeiten hat.

Des weiteren wäre es hilfreich gewesen, die IMU vollständig zu kalibrieren und konfigurieren, um zu testen, ob diese den Cartographer-Algorithmus verbessert hätten.

In einem weiterführenden Schritt hätte man noch die Navigation nach dem Erstellen der Karten ausprobieren können. Dabei hätte man vergleichend auf verschiedene Navigationsalgorithmen eingehen können und diese ebenfalls wie die Kartierungsalgorithmen in einem ROS-Package bündeln können.

Abkürzungsverzeichnis

SLAM Simultaneous Localization and Mapping

SIR Sampling Importance Resampling

IMU Inertial Measurement Unit

ROS Robot Operating System

Tabellenverzeichnis

2.1	ROS-Versionen [4]	5
2.2	ROS-Kommandozeile	10
3.1	Topic-Abonnements des Packages Cartographer	13
3.2	Topic-Veröffentlichungen des Packages Cartographer	13
3.3	Topic-Abonnements des Packages hector_mapping	15
3.4	Topic-Veröffentlichungen des Packages hector_mapping	15
3.5	Topic-Abonnements des Packages gmapping	17
3.6	Topic-Veröffentlichungen des Packages gmapping	17
5.1	Vergleich der Kartierungsalgorithmen	36

Abbildungsverzeichnis

2.1	Beziehung zwischen Topics und Nodes	6
2.2	Beziehung zwischen Services und Nodes	6
2.3	Abspielen eines ROS-Bags mit RQT	8
2.4	Pioneer 3-DX mit Lasercanner, Kinect und Computer	9
4.1	Pioneer 3-DX URDF-Modell	21
5.1	Szenario-Aufbau: Um ein Rechteck fahren	34
5.2	cartographer: Resultat der Aufnahmen des ersten Szenarios	37
5.3	cartographer: Resultat der Aufnahmen des ersten Szenarios mit IMU	38
5.4	cartographer: Resultat der Aufnahmen des zweiten Szenarios	38
5.5	gmapping: Resultat der Aufnahmen des ersten Szenarios	39
5.6	gmapping: Resultat der Aufnahmen des zweiten Szenarios	39
5.7	hector_mapping: Resultat der Aufnahmen des ersten Szenarios . . .	40
5.8	hector_mapping: Resultat der Aufnahmen des zweiten Szenarios . .	40

Listings

2.1	Beispiel eines Launch-Files in XML	7
4.1	Umwandlung von xacro in urdf und grafische Darstellung	20
4.2	Installation des Packages Cartographer	22
4.3	Installation des Packages hector_mapping	22
4.4	Installation des Packages gmapping	23
4.5	Online-Kartierung	25
4.6	Aufnahme eines Bag-Files	25
4.7	Offline-Kartierung	25
4.8	Online Launch-File für das Package Cartographer	26
4.9	Konfigurationsdatei für den Cartographer	27
4.10	Offline Launch-File für das Package Cartographer	28
4.11	Online Launch-File für das Package hector_mapping	29
4.12	Offline Launch-File für das Package hector_mapping	30
4.13	Online Launch-File für das Package gmapping	31
4.14	Offline Launch-File für das Package gmapping	31

Literatur

- [1] A. Martinez, *Learning ROS for Robotics Programming*. Packt Publishing Ltd., 2013, Bd. 2.
- [2] J. M. O’Kane, *A Gentle Introduction to ROS*. Department of Computer Science und Engineering, 2013, Bd. 2.1.2.
- [3] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler und A. Y. Ng, „ROS: an open-source Robot Operating System“, Jg. 3, Nr. 3.2, S. 5, 2009.
- [4] *Distributions - ROS Wiki*, (Accessed on 07/13/2019). Adresse: <http://wiki.ros.org/Distributions>.
- [5] *Pioneer 3 Operations Manual*. Mobile Robots Inc, 2007. Adresse: <http://vigir.missouri.edu/~gdesouza/Research/MobileRobotics/Software/P3OpMan5.pdf>.
- [6] W. Hess, D. Kohler, H. Rapp und D. Andor, „Real-time loop closure in 2D LIDAR SLAM“, in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, Mai 2016, S. 1271–1278. DOI: 10.1109/ICRA.2016.7487258.
- [7] *Cartographer - ROS Wiki*, (Accessed on 07/24/2019). Adresse: <http://wiki.ros.org/cartographer>.
- [8] S. Kohlbrecher, O. von Stryk, J. Meyer und U. Klingauf, „A flexible and scalable SLAM system with full 3D motion estimation“, in *2011 IEEE International Symposium on Safety, Security, and Rescue Robotics*, Nov. 2011, S. 155–160. DOI: 10.1109/SSRR.2011.6106777.
- [9] *Hector Mapping - ROS Wiki*, (Accessed on 07/24/2019). Adresse: http://wiki.ros.org/hector_mapping.
- [10] C. S. Giorgio Grisetti und W. Burgard, „Improved Techniques for Grid Mapping with Rao-Blackwellized Particle Filters“, in *IEEE Transactions on Robotics*, Bd. 23, 2007, S. 34–46.

- [11] G. Grisetti, C. Stachniss und W. Burgard, „Improving Grid-based SLAM with Rao-Blackwellized Particle Filters by Adaptive Proposals and Selective Resampling“, in *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, Apr. 2005, S. 2432–2437. DOI: 10.1109/ROBOT.2005.1570477.
- [12] *Gmapping - ROS Wiki*, (Accessed on 07/24/2019). Adresse: <http://wiki.ros.org/gmapping>.
- [13] *xacro - ROS Wiki*, (Accessed on 07/24/2019). Adresse: <http://wiki.ros.org/xacro>.
- [14] *urdf - ROS Wiki*, (Accessed on 07/24/2019). Adresse: <http://wiki.ros.org/urdf>.
- [15] *Compiling Cartographer ROS & Cartographer ROS documentation*, (Accessed on 15/08/2019). Adresse: <https://google-cartographer-ros.readthedocs.io/en/latest/compilation.html>.