Weiterentwicklung eines selbstfahrenden Fahrzeuges mit Lidar und anderen Sensoren

Studienarbeit

über die ersten drei Quartale des 3. Studienjahres

an der Fakultät für Technik
im Studiengang Informationstechnik

an der DHBW Ravensburg
Campus Friedrichshafen

von
Justin Serrer - 5577068 - TIT21
Timo Waibel - 8161449 - TIT21
Janik Frick - 4268671 - TIT21

Sperrvermerk

gemäß Ziffer 1.1.13 der Anlage 1 zu §§ 3, 4 und 5 der Studien- und Prüfungsordnung für die Bachelorstudiengänge im Studienbereich Technik der Dualen Hochschule Baden-Württemberg vom 29.09.2017 in der Fassung vom 25.07.2018.

„Der Inhalt dieser Arbeit darf weder als Ganzes noch in Auszügen Personen außerhalb des Prüfungsprozesses und des Evaluationsverfahrens zugänglich gemacht werden, sofern keine anders lautende Genehmigung vom Dualen Partner vorliegt."

Ort, Datum	Unterschrift

Selbständigkeitserklärung

gemäß Ziffer 1.1.13 der Anlage 1 zu §§ 3, 4 und 5 der Studien- und Prüfungsordnung für die Bachelorstudiengänge im Studienbereich Technik der Dualen Hochschule Baden-Württemberg vom 29.09.2017 in der Fassung vom 25.07.2018.
Ich versichere hiermit, dass ich meine Hausarbeit mit dem Thema

Weiterentwicklung eines selbstfahrenden Fahrzeuges mit Lidar und anderen Sensoren

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ort, Datum	Unterschrift

Selbständigkeitserklärung

gemäß Ziffer 1.1.13 der Anlage 1 zu §§ 3, 4 und 5 der Studien- und Prüfungsordnung für die Bachelorstudiengänge im Studienbereich Technik der Dualen Hochschule Baden-Württemberg vom 29.09.2017 in der Fassung vom 25.07.2018.
Ich versichere hiermit, dass ich meine Hausarbeit mit dem Thema

Weiterentwicklung eines selbstfahrenden Fahrzeuges mit Lidar und anderen Sensoren

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ort, Datum	Unterschrift

Selbständigkeitserklärung

gemäß Ziffer 1.1.13 der Anlage 1 zu §§ 3, 4 und 5 der Studien- und Prüfungsordnung für die Bachelorstudiengänge im Studienbereich Technik der Dualen Hochschule Baden-Württemberg vom 29.09.2017 in der Fassung vom 25.07.2018.
Ich versichere hiermit, dass ich meine Hausarbeit mit dem Thema

Weiterentwicklung eines selbstfahrenden Fahrzeuges mit Lidar und anderen Sensoren

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ort, Datum	Unterschrift

Gender-Erklärung

Das in dieser Arbeit gewählte generische Maskulinum bezieht sich zugleich auf die männliche, die weibliche und andere Geschlechteridentitäten. Zur besseren Lesbarkeit wird auf die Verwendung männlicher und weiblicher Sprachformen verzichtet. Alle Geschlechteridentitäten werden ausdrücklich mitgemeint, soweit die Aussagen dies erfordern.

Inhaltsverzeichnis

Abkürzungsverzeichnis

GPIO general-purpose input/output

LiDAR Light Detection and Ranging

ROS Robot Operating System

SLAM Simultanous Localization and Mapping

GPS Global Positioning System

ICP Iterative Closest Point

PCL Point Cloud Library

SDK Software Development Kit

Abbildungsverzeichnis

Listings

1 Einleitung

Die Automatisierung im Straßenverkehr befindet sich in stetigem Wachstum, wobei selbstfahrende Fahrzeuge zunehmend an Bedeutung gewinnen.
Auch im Bereich der Modell-Autos und Roboter ist Automatisierung ein präsentes Thema.
Zwar ist die Umsetzung eines selbstfahrenden Modell-Autos oder Roboters, aufgrund der vorhersehbaren und weniger komplexen Umgebung, einfacher als die Umsetzung eines selbstfahrenden PKW, jedoch stellt sie trotzdem eine Herausforderung dar.

2 Problemstellung, Ziel und Umsetzung

In diesem Abschnitt wird auf die Problemstellung, das generelle Ziel und die geplante Umsetzung der Arbeit eingegangen. Des Weiteren wird erläutert, weshalb das Ziel der Arbeit wichtig ist, wie die Arbeit aufgebaut ist und welche Probleme und Schwierigkeiten durch bereits getätigte Versuche einer Umsetzung des Arbeits-Ziels bereits bekannt sind.

2.1 Problemstellung

Bisherige Versuche, ein selbstfahrendes Auto, im Rahmen einer Studienarbeit zu entwerfen und einen entsprechenden Algorithmus zu programmieren, sind gescheitert. Die beiden Hauptprobleme der bisherigen Arbeiten, war zum einen der Bau eines geeigneten Fahrzeugs und zum anderen die Entwicklung einer Software, welche es dem Auto ermöglicht, ohne manuelle Steuerung, Hindernisse zu erkennen und um diese herum zu navigieren. Da es sich bei den Studenten der bisherigen Gruppen ausschließlich um Studenten mit einem Schwerpunkt in Elektrotechnik handelte, war die Entwicklung und Implementierung der Software zum autonomen Fahren die größere Herausforderung.
Da keine der bisherigen Gruppen, das Problem der Software lösen konnte, wurde sich dazu entschieden, die Aufgaben aufzuteilen. Die Aufgabe, der Erstellung eines funktionsfähigen Modell-Autos und einer Schnittstelle, zur Steuerung des Autos, wurde einer Gruppe von Studenten mit einem Schwerpunkt in Elektrotechnik zugeteilt. Somit ist das Hauptproblem dieser Arbeit, die Entwicklung und Implementierung eines Ausweichalgorithmus, welcher das Auto, über die, von der anderen Gruppe zur Verfügung gestellten Schnittstelle, steuern soll und so eine autonome Hindernis-Detektierung und Vermeidung ermöglicht.

Da die Aufgabe auf mehrere Gruppen aufgeteilt wurde, ist ein weiteres Problem die Kommunikation zwischen den Gruppen. Um einen reibungslosen und effizienten Ablauf gewährleisten zu können, sollte diese möglichst Umfangreich sein.
Durch die Aufteilung auf verschiedene Gruppen entsteht zusätzlich das Problem, dass die Hardware nur eingeschränkt verfügbar ist. Da die Gruppe, welche den Hardware-Teil der Aufgabe übernimmt, diese erst bauen und anschließend auch testen muss, ist die Hardware, vor allem zu Beginn der Arbeit, kaum verfügbar. Daher ist zur Entwicklung der Software, eine Abstrahierung der Hardware notwendig. Konkret bedeutet das, dass die Schnittstelle zur Steuerung, sowie die Daten der Sensoren simuliert werden müssen, um ein Testen des Algorithmus auch ohne Verfügbarkeit der Hardware zu ermöglichen.

2.2 Ziele der Arbeit

In diesem Abschnitt werden die Ziele der Arbeit und deren Metriken beschrieben.
Das Ziel auf das hingearbeitet wird, ist die Implementierung eines Algorithmus für ein Modell-Fahrzeug. Durch den Algorithmus soll es möglich sein, das Fahrzeug autonom durch die Umgebung zu einem Zielpunkt zu steuern. Da das Auto selbst parallel von einer anderen Gruppe gebaut wird, steht es erst zu einem späteren Zeitpunkt zur Verfügung. Daher ist die Entwicklung einer Simulation ein erster Zwischenschritt.
Um den Fortschritt des Projekts und die Ergebnisse messen zu können, werden Metriken für verschiedene Bereiche definiert. Die Erfüllung von Zielmetriken ist teilweise abhängig von der Wahl der Hard- und Softwarekomponenten. Aus diesem Grund werden an dieser Stelle zunächst die Metriken definiert, die unabhängig von externen Faktoren erfüllt werden können.

1. **Simulation einer Umgebung**
   In der Simulation sollen Hindernisse generiert werden können, die von einem simulierten Sensor erfasst werden können.

   Metrik:

   - Kann eine Umgebung generiert werden? → Ja oder Nein

2. **Erkennung von Hindernissen**
   Die Software muss in der Lage sein alle Hindernisse erkennen die den Spezifikationen, die für die Anpassung an die technischen Gegebenheiten getroffen werden, entsprechen.

Metrik:

- Werden alle Hindernisse erkannt? → Ja oder Nein

3. **Festlegen eines Zielpunktes**
   Um einen Ausweichalgorithmus zu testen, ist die Festlegung eines Zielpunktes unumgänglich. Denn durch den Zielpunkt ist klar definiert, wo das Auto ankommen soll und welche Hindernisse auf dem Weg zwischen Fahrzeug und Ziel liegen.

   Metrik:

   - Kann ein Zielpunkt definiert werden? → Ja oder Nein

4. **Simulation eines Umgebungssensors**
   Um die Hindernisse in der Umgebung zu erkennen, soll ein Sensor simuliert werden, der die Hindernisse erfasst und für die weitere Verarbeitung verfügbar macht.

   Metrik:

   - Kann ein Sensor simuliert werden? → Ja oder Nein

5. **Lokalisierung des Fahrzeugs**
   Um eine Route für das Fahrzeug berechnen zu können, muss eine Algorithmik zur Lokalisierung des Fahrzeugs implementiert werden.

   Metrik:

   - Für die Lokalisierung bezieht sich die Metrik auf Präzision und Geschwindigkeit. Beide Elemente sind abhängig von den gewählten Hard- und Softwarekomponenten.

6. **Berechnung der Route**
   Damit das Fahrzeug den festgelegten Zielpunkt erreichen kann, muss eine Route berechnet werden, die um alle erkannten Hindernisse herum führt.

   Metriken:

   - Führt die Route vom Fahrzeug zum Zielpunkt → Ja oder Nein
   - Werden alle, in der Karte des Fahrzeuges bekannten, Hindernisse umfahren → Ja oder Nein
   - Kann die gesamte Route mit dem Fahrzeug abgefahren werden? → Ja oder Nein
   - Wie lange dauert die Berechnung der Route. Berechnungsdauer $<$ 100 ms

7. **Steuerung des Fahrzeuges**
   Um das Auto entlang der berechneten Route zu bewegen, muss es von der Software angesteuert werden können.

   Metrik:

   - Kann das Auto angesteuert werden? → Ja oder Nein

2.3 Umsetzung

Um eine umfangreiche Kommunikation zwischen den Gruppen zu ermöglichen, müssen Kommunikationswege so früh wie möglich erstellt werden. Des Weiteren sollten Termine für regelmäßige Meetings festgelegt werden, um einen konstanten Austausch von Informationen zwischen den Gruppen zu gewährleisten.
Zur Umsetzung der Aufgabe selbst, stehen, neben einem RPLiDAR A1M8-R6 der Firma Slamtec, auch weitere Sensoren, wie Ultraschall-, Lenkwinkel- und Geschwindigkeits-Sensoren, sowie ein Raspberry PI 4 zur Verfügung. Bevor die eigentliche Arbeit an einem Algorithmus beginnen kann, muss die gegebene Hardware getestet werden. Zudem ist es, um das weitere Vorgehen planen zu können, notwendig, sich mit der Hardware vertraut zu machen. Zu wissen, welche Daten von den Sensoren, wann gesendet werden, ermöglicht es, präziser zu planen, wodurch die Entwicklung des Algorithmus effizienter wird.
Nachdem verstanden wurde, wie die Hardware funktioniert, muss eine Möglichkeit, diese zu simulieren, entwickelt werden. Hierbei ist es wichtig, die, für den Algorithmus notwendige Hardware, so genau wie möglich zu simulieren. Je genauer die Simulation ist, desto unwahrscheinlicher treten Probleme bei der Zusammenführung von Hard- und Software auf.
Die Simulation dient jedoch nur zum Testen des Algorithmus. Im späteren Betrieb sollen, anstelle der Daten der Simulation, die Daten der vorhandenen Sensorik verwendet werden. Hierzu ist die Entwicklung eines Interface, durch welches mit der Sensorik kommuniziert werden kann, notwendig.
Nachdem nun realitätsnahe, simulierte Daten, sowie echte Daten, eingelesen werden können, kann die Umsetzung eines Simultanous Localization and Mapping (SLAM)-Algorithmus begonnen werden. Hierzu muss eine Möglichkeit entwickelt werden, mit der die vorhandenen Daten zur Erstellung einer Karte genutzt werden können. Des Weiteren muss das Auto innerhalb der Karte lokalisiert werden können.
Als Nächstes muss der Ausweichalgorithmus entwickelt werden. Dieser muss in der Lage sein, die vorhandenen Daten zu nutzen und so das Auto um Hindernisse herum zu einem gewünschten Zielort zu navigieren.

Abschließend wird die Hardware und die Software vereint und getestet.

3 Überblick Hardware und Software

In diesem Kapitel wird die Hardware und Software beschrieben, welche zur Bearbeitung der Studienarbeit zur Verfügung stehen. Zusätzlich wird die Technologie-Entscheidung erläutert und begründet.

3.1 Hardware

Dieser Abschnitt beschreibt die Hardware, welche für die Entwicklung des Ausweichalgorithmus relevant ist.

1. **Raspberry Pi 4 Model B**
   Der Raspberry Pi 4 ist ein Single-Board-Computer, welcher im Jahr 2019 auf dem Markt erschien. Er besitzt eine ARM-basierte 64-Bit CPU, welche mit 1.5GHz getaktet ist. Das Modell, welches im Rahmen unserer Studienarbeit genutzt wird, besitzt 8 GB Arbeitsspeicher. Außerdem verfügt der Raspberry Pi 4 über 40 general-purpose input/output (GPIO) Pins, welche zur Kommunikation mit den Sensoren und der Steuerungs-Schnittstelle genutzt werden können. [16]

2. **Slamtec RPLiDAR A1M8-R6**
   Der RPLiDAR A1M8-R6 von Slamtec ist ein zweidimensionaler Laser-Scanner, welcher mittels Light Detection and Ranging (LiDAR), ein 360° Scan der Umgebung erstellen kann. [19, p. 3] Er hat eine effektive Reichweite von 0.15 bis 12 Meter und bei einer Scan-Rate von 5.5 Scans pro Sekunde, sowie eine Scan-Frequenz von 8000 Hz, eine Auflösung von weniger als einem Grad. [19, p. 8]

3. **Weitere Sensoren**
   Da der LiDAR-Sensor nur zweidimensionale Scans macht, können Hindernisse, welche kleiner als die Scan-Höhe des LiDAR sind, von diesem nicht erfasst werden. Daher sind weitere Sensoren, wie z.B. Ultraschall-Sensoren notwendig, um auch niedrige Hindernisse erkennen zu können. Außerdem wäre ein Sensor zur Bestimmung des aktuellen Lenkwinkels und ein weiterer Sensor zum Bestimmen der aktuellen Geschwindigkeit sinnvoll. Die Daten dieser Sensoren könnten bei der Ermittlung der Position im Raum von Nutzen sein.

3.2 Software

In diesem Abschnitt wird auf die Software eingegangen, welche zur Entwicklung des Algorithmus zur Verfügung steht.

1. **Robot Operating System (ROS)**
   ROS ist eine Ansammlung von Werkzeugen und Bibliotheken, wie Treiber und Algorithmen, welche bei der Entwicklung von Roboter-Anwendungen helfen sollen. Hierbei ist ROS vollständig Open-Source und bietet zudem eine ausführliche Dokumentation, Foren und eine große Community. [17] Des Weiteren bietet Slamtec, der Hersteller des zur Verfügung stehenden LiDAR-Sensors, eine Bibliothek, zur Nutzung des LiDAR-Sensors, in Kombination mit verschiedenen Versionen des ROS an. [6]

2. **Slamtec RPLiDAR Public SDK**
   Slamtec bietet, neben der ROS-Bibliothek, auch eine öffentlich zugängliches Software Development Kit (SDK) für sämtliche RPLiDAR-Produkte an. Das SDK ist in C++ geschrieben und unter der BSD 2-clause Lizenz lizenziert. [7]

3. **Slamtec RPLiDAR SDK Python-Ports**
   Die Slamtec RPLidar Sensoren sind, aufgrund des erschwinglichen Preises, vor allem bei Einsteigern sehr beliebt. Auch die Programmiersprache Python ist in den letzten Jahren immer beliebter geworden. Da Slamtec selbst kein Python-SDK anbietet, entstanden über die Jahre diverse Ports des C++-SDK.

3.3 Technologie-Entscheidung

Dieser Abschnitt erläutert die Entscheidungen für die diversen Technologien, welche zur Bearbeitung des praktischen Teils der Arbeit gewählt wurden.

Hardware
Die Auswahlmöglichkeiten der Hardware sind sehr beschränkt. Da die notwendigen Berechnungen einiges an Leistung benötigen, wird sich für das leistungsstärkste, verfügbare Modell des Raspberry Pi entschieden. Der Raspberry Pi 4 Model B bietet neben einer 64-Bit CPU mit ausreichender Leistung, auch 8 GB Arbeitsspeicher und einen Formfaktor der klein genug ist, um eine Integration in das Fahrzeug zu ermöglichen. Zusätzlich bietet er ausreichend Schnittstellen um mit den diversen Sensoren und Motoren kommunizieren zu können.
Als Hauptsensor für das Messen der Umgebung steht nur der RPLiDAR A1M8-R6 von Slamtec zur Verfügung. Weitere Sensorik soll zwar auf dem Auto verbaut werden, jedoch erstmal nicht von der Software berücksichtigt werden. Der Grund hierfür ist der Zeitpunkt, zu dem die Sensorik verbaut werden kann. Die Verarbeitung der Daten des LiDAR kann ohne Auto oder

per Simulation getestet werden, wohingegen es bei der anderen Sensorik sehr stark auf die Integration in dem Fahrzeug ankommt. Da diese jedoch erst zum Schluss in dem Fahrzeug verbaut werden, wäre eine Software-Integration dieser Sensorik nur schwer rechtzeitig umzusetzen.

Software

Maßgeblich verantwortlich für die Auswahl der Programmiersprache, ist die Auswahl der Software, welche verwendet wird um den LiDAR anzusteuern.
Die eine Möglichkeit wäre das Nutzen des offiziellen Slamtec RPLLiDAR ROS-Paket. Bei ROS handelt es sich jedoch um eine sehr umfangreiche Software. Diese kommt mit vielen, für die Studienarbeit nicht relevanten, Komponenten daher. Des Weiteren benötigt ROS ein anderes, nicht für den Raspberry Pi optimiertes, Betriebssystem wie Ubuntu. Das hat zu Folge, das weitere Ressourcen für das Betriebssystem benötigt werden und nicht für die notwendigen Berechnungen zur Verfügung stehen.
Die Alternative zur Verwendung von ROS, ist das Nutzen eines SDK. Die Python-Ports des SDK sind alle schon einige Jahre alt und haben teilweise keine wirklich übersichtliche Struktur. Das offizielle C++-SDK hingegen wird immer noch regelmäßige upgedatet und bietet eine umfangreiche Dokumentation sowie einige Beispielprogramme.
Um die begrenzt vorhandenen Ressourcen des Raspberry Pi optimal nutzen zu können, ist daher die Nutzung des C++-SDK und einem entsprechenden Interface die beste Lösung. Da das SDK in C++ geschrieben ist, ergibt es Sinn, die restliche Software ebenfalls in C++ zu implementieren. Zusätzlich verbessert die Nutzung einer kompilierten Programmiersprache wie C++ die Laufzeit der Software was die Reaktionszeit des Autos verbessert.

4 Systemvoraussetzungen und Einschränkungen

In diesem Kapitel wird definiert welche Voraussetzungen erfüllt sein müssen, um eine korrekte Funktion der Software sicherzustellen. Die Algorithmik für die Ortung des Fahrzeuges befindet sich in einem frühen Entwicklungsstadium. Aus diesem Grund müssen einige Bedingungen eingehalten werden.

4.1 Umweltvoraussetzungen

In diesem Abschnitt werden die Voraussetzungen an das Einsatzgebiet des Autos definiert.

1. **Statische Umgebung**
   Das Fahrzeug darf nur in einer statischen Umgebung autonom gefahren werden. Die Objekte in der Umgebung dürfen während der Fahrt nicht bewegt werden. Grund dafür ist, dass nicht bekannt ist, wie sich eine dynamische Umgebung auf die Präzision der Lokalisierungsalgorithmik auswirkt.

2. **Hindernisse**
   Laut Datenblatt [19] liegen die Distanzen, die der Sensor erfassen kann, zwischen 0.15 - 12 Metern. Um die Versorgung mit validen Daten sicherzustellen, muss die Umgebung so gebaut sein, dass zu jedem Zeitpunkt, sowohl in x-Richtung als auch in y-Richtung, Objekte mit einem maximalen Abstand von maximal 10 Metern vorhanden sind. Zudem müssen die Hindernisse die Scan-Ebene des LiDAR-Sensor schneiden, da sie ansonsten nicht erkannt werden. Auch sollte eine Mindestanzahl von 100 Punkten pro Scan geliefert werden. Bei Nichteinhaltung der genannten Bedingungen ist eine Lokalisierung mittels des implementierten Algorithmus ungenau.

3. **Trockene Umgebung**
   Das Fahrzeug darf nur in einem vor Wasser geschützten Bereich verwendet werden. Hintergrund ist, dass die Elektronik nicht vor eindringendem Wasser geschützt ist. Eindringendes Wasser könnte das Fahrzeug so beschädigen, dass es nicht mehr funktioniert.

4. **Höhennnterschiede der Umgebung**
   Das Fahrzeug darf nur in ebenen Umgebungen verwendet werden. Steigungen sorgen dafür, dass Hindernisse oberhalb oder unterhalb des Fahrzeuges nicht erkannt werden. Dadurch ist eine Vermeidung von Kollisionen und eine korrekte Berechnung der Route nicht zu jedem Zeitpunkt gegeben.

4.2 Einschränkungen

In diesem Abschnitt werden die Voraussetzungen und Einschränkungen beschrieben, die berücksichtigt werden müssen, wenn die Algorithmik weiterentwickelt wird.

1. **Steuerung**
   Der Algorithmus muss auf die Steuerungsmöglichkeiten des Autos angepasst sein. Das bedeutet, dass der Algorithmus vor allem den Lenkwinkel und die Breite des Autos berücksichtigen muss. Die Steuerung selbst soll über eine klar definierte Schnittstelle erfolgen. Eine Nutzung des Algorithmus für ein anderes Auto wird daher nur nach einer Anpassung der Algorithmik möglich sein.

2. **Simulation**
   Die Simulation soll einen ersten Ansatz für das Testen und die Visualisierung bieten. Deshalb ist die Implementierung eine vereinfachte Darstellung eine Umgebung. Die Simulation soll eine einfache Top-Down Perspektive auf das Auto und die Umgebung bieten. Das virtuelle Auto soll manuell und mittels Algorithmus gesteuert werden können. Auch hier gilt zu beachten, dass die Steuerung des simulierten Autos möglichst identisch mit der des eigentlichen Autos ist. Das gilt auch für die Schnittstelle zur Steuerung, welche der Algorithmus nutzen wird. Aufgrund der Abhängigkeit des Algorithmus von den konkreten Werten des Autos, wird die Simulation bezogen auf die Ansteuerung des Fahrzeuges und der Visualisierung der Umgebung keine exakte, detaillierte Abbildung der Realität bieten. Das tatsächliche Verhalten eines verwendeten Fahrzeuges kann daher von der Simulation abweichen.

3. **Laufzeit**
   Der Laufzeit des Algorithmus soll ausreichend kurz sein. Das bedeutet, dass Hindernisse in unter einer halben Sekunde erkannt werden und entsprechend reagiert werden soll. Dieser Wert ist kein empirisch, oder anders wissenschaftlich validierter Wert. Für den Einsatz in der Realität muss ein fundierter Wert ermittelt werden, sodass die tatsächlich mögliche Geschwindigkeit berücksichtigt werden kann.

5 SLAM

SLAM ist ein bekanntes Problem in der Robotertechnik. Im Folgenden wird das Problem selbst erläutert und näher auf die Umsetzung von SLAM im Rahmen dieser Studienarbeit eingegangen.

5.1 Was ist SLAM?

Bei dem SLAM-Problem handelt es sich um das Problem, eine Karte einer unbekannten Umgebung zu erstellen. Gleichzeitig soll die aktuelle Position des Roboters in dieser Karte ermittelt und dargestellt werden. Hierzu wird ausschließlich die Sensorik des Roboters genutzt.

5.2 Mapping

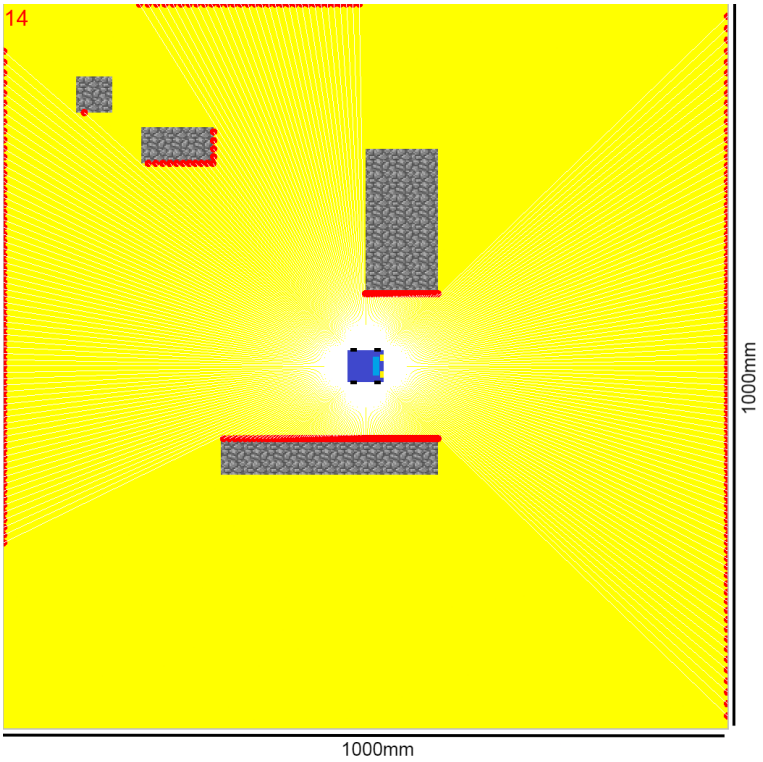
Damit das Fahrzeug Hindernisse umfahren kann, muss es die Position der Hindernisse relativ zu seiner eigenen kennen. Hierzu können aktuelle Daten der Sensorik verwendet werden. Durch das ausschließliche Nutzen der Daten in Echtzeit, wird der Bereich, in dem ein Pfad berechnet werden kann, jedoch stark eingeschränkt. Eine sinnvolle Lösung zur Vergrößerung des Radius der bekannten Umgebung ist die Konstruktion einer Karte. Dies ermöglicht das Speichern von Informationen aus vorherigen Scans.
Für den Aufbau einer solchen Karte gibt es verschiedene Ansätze, welche im Folgenden näher beleuchtet werden.

Matrix
Da es sich, aufgrund des verwendeten LiDAR-Sensors, in dieser Arbeit um eine 2D-Karte handelt, ist der simpelste Weg eine Karte umzusetzen eine einfache Matrix. In dieser steht für jede Koordinate der Karte eine 1, -1 oder 0.
Die Matrix wird mit 0 gefüllt. Erkennt die Sensorik ein Hindernis an einer Koordinate (X,Y), wird der Wert an der Stelle in der Matrix auf 1 gesetzt. Die Werte zwischen dem Hindernis und der aktuellen Position des Sensors können auf -1 gesetzt werden, da keine Hindernisse in diesem Bereich erkannt wurden. Somit kann, solange die Position des Fahrzeugs bekannt ist, die Karte Scan für Scan gefüllt werden.

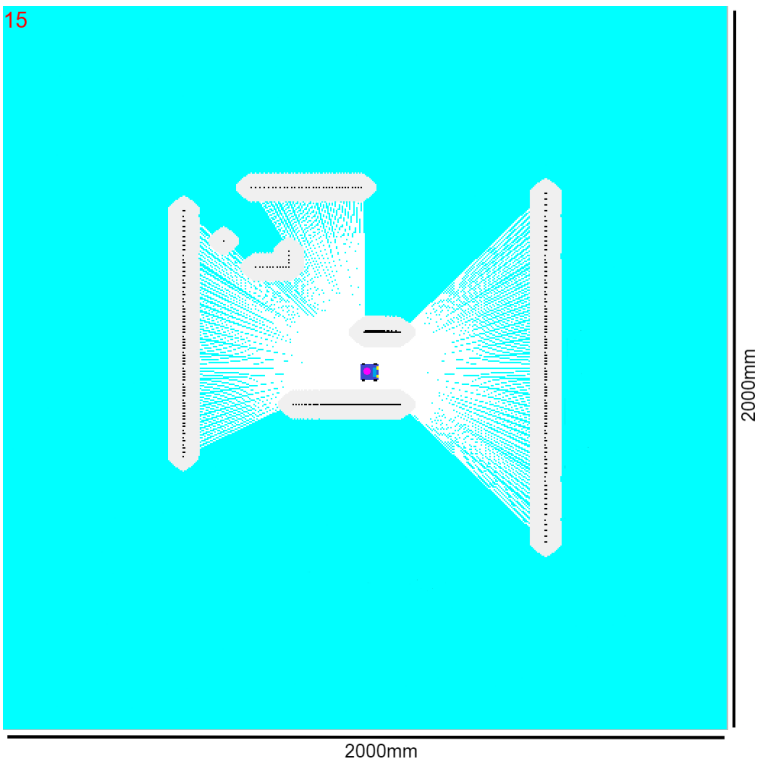
Grid Map
Im Falle dieser Arbeit werden sämtliche Distanzen in Millimetern angegeben. Jedes Feld der Matrix entspricht also einem 1x1 Millimeter großem Quadrat der Umgebung. Jedoch hat die Sensorik eine begrenzte Auflösung. Die

gescannten Punkte können also, vor allem bei größeren Entfernungen, viele Millimeter oder Zentimeter voneinander entfernt sein. Infolgedessen kann es dazu führen, dass eine Wand als viele einzelne Punkte erkannt wird.
Zur Lösung dieses Problems kann ein Grid mit niedrigerer Auflösung verwendet werden. Die Berechnungen finden somit weiter in Millimetern statt, die Karte selbst wird jedoch als Grid in einer niedrigeren Auflösung gespeichert. Hat das Grid zum Beispiel eine Auflösung von 1x1 Zentimeter, werden gescannte Punkte auf Zentimeter gerundet bevor die Information in der Karte gespeichert wird.

Wahrscheinlichkeiten
Eine weitere sinnvolle Ergänzung ist die Verwendung von Wahrscheinlichkeiten. Anstelle von 1, -1 und 0 werden auch sämtliche Werte dazwischen genutzt.
Wird ein Hindernis erkannt, wird der entsprechende Wert im Grid nicht auf 1 gesetzt. Stattdessen wird der Wert mit jedem mal wo ein Hindernis in dem Grid erkannt wird, leicht erhöht. Gleichzeitig wird der Wert leicht verringert, wenn ein Punkt in dem Grid als frei erkannt wird. Zusätzlich werden Schwellenwerte definiert, die festlegen, ab wann ein Feld des Grids als belegt oder frei gilt.
Das Ergebnis ist eine Occupancy Grid Map, welche eine geringere Auflösung als die gesamte Karte hat. In ihr werden Informationen über den Status des Teils der Karte gespeichert, welche von dem Feld der Grid Map repräsentiert wird.
Aufgrund der Simplizität und einfachen Umsetzung wurde sich dazu entschieden, eine solche Occupancy Grid zum Mapping zu verwenden. Siehe Abbildung 5.1



(a) Simulierte Umgebung



(b) Occupancy Grid Map

Abbildung 5.1: Umgebung auf Occupancy Grid Map abgebildet

Wie in der Abbildung 5.1 zu sehen ist, wird die Umgebung (a) gescannt und in eine Grid Map (b) abgebildet. Die Map selbst ist mit 2000mm x 2000mm doppelt so groß wie die Umgebung, das Grid hat jedoch nur eine Auflösung von 500px x 500px. Das bedeutet, dass jeder Pixel des Grid ein 4mm x 4mm Quadrat der Map darstellt.

5.3 Lokalisierung

Das selbstfahrende Fahrzeug soll in der Lage sein ein vorgegebenes Ziel zu erreichen. Daher ist neben dem Mapping auch die Lokalisierung des Fahrzeuges eine zentrale Aufgabe. Denn ohne das Wissen über die aktuelle Position auf der Karte kann kein Weg zum Ziel berechnet werden und es kann auch nicht bestimmt werden, ob das Ziel erreicht ist.
In Folgenden werden verschiedene Ansätze zur Lösung des Lokalisierungsproblems mit den Vor- und Nachteilen beschrieben.

5.3.1 Datengenerierung

In diesem Abschnitt werden verschiedene Möglichkeiten zur Generierung von Daten, welche zur Lokalisierung eines Fahrzeugs genutzt werden können, näher betrachtet.

Datenquellen zur Erzeugung globaler Bewegungsdaten
Globale Verfahren zur Datenerzeugung für Lokalisierungsalgorithmen basieren darauf, dass die Umgebung dafür präpariert ist. Das Fahrzeug kommuniziert mit Sendern, die in der Umgebung verfügbar sind. Mit Hilfe dieser Sender wird ein globales Netz erstellt, dass dem Fahrzeug die Berechnung der absoluten Position im globalen Koordinatensystem ermöglicht.

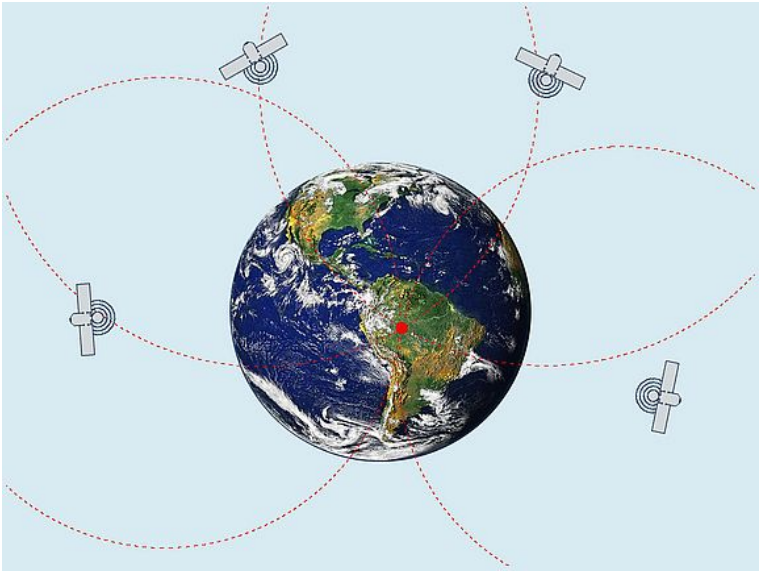


Abbildung 5.2: Visualisierung von GPS als Beispiel einer Datenquelle zur Erzeugung globaler Bewegungsdaten

1. **Global Positioning System (GPS)**
   Für die Ortung eines Fahrzeugs kommt in der Praxis das GPS zum Einsatz. GPS arbeitet mit Satelliten, die die Position des Benutzers, in diesem Fall des Fahrzeugs, bestimmen und übermitteln [1] Abbildung 5.2. Die Genauigkeit des GPS beträgt ca. 5-10 cm [1].

   Der Vorteil der Nutzung eines GPS-Sensors ist eine globale Verfügbarkeit.

   Im Einsatz für Fahrzeuge auf der Straße ist diese Genauigkeit ausreichend, da die lokalisierten Objekte deutlich größer sind und dadurch trotz der Toleranzen der richtige Ort gefunden werden kann. Relativ zur Fahrzeuggröße sind 5-10 cm bei einem kleinen Modellfahrzeug eine deutliche Abweichung, die abhängig von der Umgebung des Fahrzeugs ernsthafte Konsequenzen haben kann.

   Eine weitere Problematik, die die Verwendung von GPS-Daten mit sich bringt, ist die Abhängigkeit von der Signalstärke und -verfügbarkeit. Ist das Signal schwach, kann die Abweichung noch größer werden. Ist kein Signal verfügbar, ist gar keine Ortung möglich.

2. **Eigenes GPS**
   Um das Problem der Signalverfügbarkeit zu lösen, könnte man auf die Idee kommen ein eigenes Global Positioning System (GPS) aufzubauen, dass kleine Sender statt Satelliten verwendet. Diese Sender werden in der Umgebung platziert. Auf dem Fahrzeug ist ein Empfänger montiert, der

die Entfernungen zu den Sendern erfasst. Mit dieser Technologie kann dann über Triangulation die Position des Fahrzeugs bestimmt werden. Dadurch wäre je nach Qualität von Sender und Empfänger eine höhere Präzision als 5-10 cm möglich.

Damit wären also beide Probleme von GPS in diesem Kontext gelöst. Aber es gibt auch einen deutlichen Nachteil. Denn vor der Verwendung des Fahrzeugs muss die Umgebung zunächst mit den Sendern präpariert werden. Ein Einsatz in unbekannten Gebieten ist dadurch nicht möglich. Je nach Einsatzzweck des Fahrzeuges ist das ein großes Problem.

Quellen für die Erzeugung relativer Bewegungsdaten
Globale Ortungsverfahren haben den Nachteil abhängig von den Gegebenheiten der Umgebung zu sein. Ist in der Umgebung keine Kommunikation mit den Sendern möglich, so ist keine Lokalisierung möglich. Dabei ist es nicht von Bedeutung, ob das Signal von Satelliten oder von selbst angebrachten Sendern in der Umgebung stammt. Aus diesem Grund gibt es auch relative Verfahren, die die Positionsänderung anhand von Differenzen in den gesammelten Daten von verbauten Sensoren berechnen.

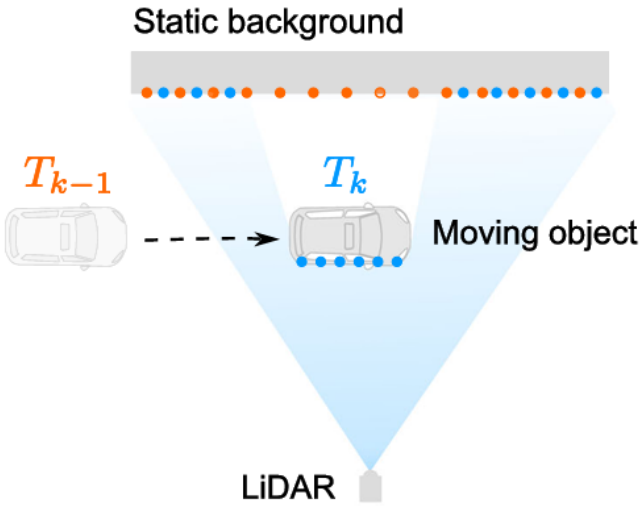


Abbildung 5.3: Visualisierung von LiDAR als Beispiel einer Datenquelle zur Erzeugung relativer Bewegungsdaten (entnommen aus [24])

In Abbildung 5.3 ist die Punktewolke eines LiDAR-Sensors zu zwei Zeitpunkten abgebildet. Die orangen Punkte gehören zu der Situation, in der das Auto noch nicht vom LiDAR erkannt wird. Die blauen Punkte gehören

zu der Situation, in der das Auto vom LiDAR erkannt wird. In diesem Fall ist die Position des Sensors statisch, während im Rahmen dieser Arbeit die Umgebung statisch ist. Die Funktionsweise bleibt aber die gleiche. Anhand der Differenz zwischen zwei Scans kann die Bewegung von Objekten ermittelt werden.

1. **LiDAR-Sensor**

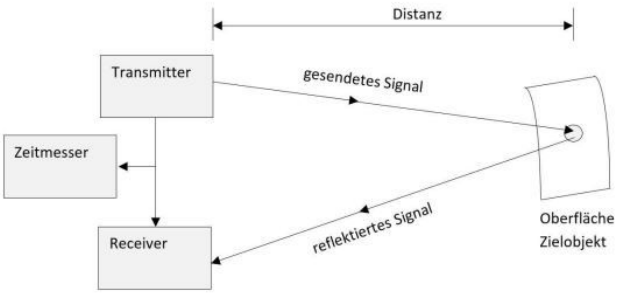


Abbildung 5.4: Visualisierung der Funktionsweise eines LiDAR-Sensors

Bei einem LiDAR-Sensor wird die Umgebung mit Hilfe von Laserstrahlen erfasst. Dabei werden Werte generiert, die die Distanz und Richtung des Objektes beinhalten. LiDAR-Sensoren bieten den Vorteil, dass sie unabhängig von den Lichtverhältnissen der Umgebung sind [11]. Bei LiDAR-Sensoren muss zwischen 2D- und 3D-Sensoren differenziert werden. 3D-Sensoren erfassen eine deutlich größere Datenmenge, wodurch die Verlässlichkeit der Daten erhöht wird. Dabei spielt es keine Rolle, ob der Sensor im Innen- oder Außenbereich zum Einsatz kommt [11].

Ein weiterer Vorteil von LiDAR-Sensoren ist die Datenrepräsentation. Die Repräsentation mit Abstand und Winkel ermöglicht eine Verarbeitung ohne aufwendige Vorbereitung und Anpassung der Daten.

Die Präzision von LiDAR-Sensoren kann aber zum Beispiel unter Wettereinflüssen leiden. Zum Beispiel können Wasserteilchen in der Luft die Reflexion der Laserstrahlen so verändern, dass die empfangenen Werte des Sensors nicht mit der Realität übereinstimmen. Auch die eingeschränkte Reichweite kann abhängig von der Umgebung ein Problem darstellen.

Bei 2D-Sensoren kann auch die feste Höhe ein Problem sein. Ist ein Hindernis unter- oder oberhalb des Sensors, aber auf Höhe anderer Fahrzeugteile, werden diese nicht erkannt und eine Kollision mit solchen Objekten kann nicht verhindert werden.

2. **Kamera**
   Kameras haben den Vorteil, dass alle Elemente, unabhängig von der Höhe, und auch in größeren Distanzen erkannt werden können.

   Der Nachteil von Kameras ist die Abhängigkeit von der Beleuchtung der Umgebung, da diese maßgeblich den erkennbaren Detailgrad beeinflusst. Auch Wetterfaktoren wie Nebel oder Niederschlag können die Qualität der Daten negativ beeinflussen.

   Auch die Verarbeitung der Daten ist ein Nachteil. Um Kameradaten automatisiert auszuwerten, müssen zunächst verschiedene Algorithmen zur Vorbereitung ausgeführt werden. Die Vorbereitung benötigt Zeit, die bei anderen Verfahren bereits zur Berechnung der Position genutzt werden kann. Außerdem sind die Vorbereitungen und die Auswertung der Bilder rechenintensiv, wodurch diese als primäre Datenquelle zur Lokalisierung eher für Fahrzeuge mit hoher Rechenleistung geeignet sind.

   Ein weiterer Nachteil von Kameras ist die eingeschränkte Sichtweite. Das Bild kann nur einen gewissen Teil der Umgebung aufnehmen und bietet nur durch die Kombination mehrerer Kameras eine vollständige Wahrnehmung der Umgebung.

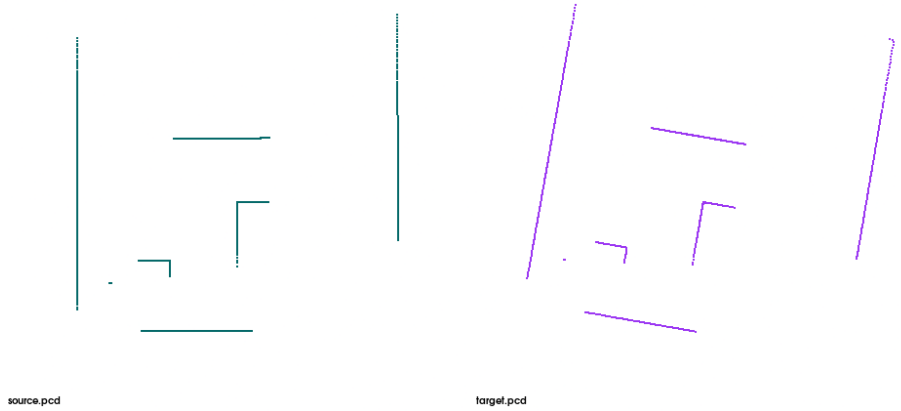
3. **Ultraschall**
   Ultraschallsensoren erfassen die Umgebung mit Hilfe von Schallwellen und deren Reflexionen. Diese Methode ist sehr einfach in der Implementierung und sehr sparsam im Energieverbrauch. Daher ist ein Einsatz auch in Fahrzeugen mit geringer Batteriekapazität möglich. Das hat aber auch Nachteile. Die Reichweite von Ultraschallsensoren ist geringer als die anderer Sensoren. Außerdem ist auch die Auflösung der erfassten Daten geringer als die anderer Sensoren.

   Daher ist der Einsatz als primäre Datenquelle für die Lokalisierung des Fahrzeuges nur bedingt geeignet. Die Stärken von Ultraschall liegen eher im Nahbereich. Ein Einsatz dieser Technologie wäre also als Zusatz zu einer anderen Quelle denkbar. Das Ziel wäre dann durch die Ultraschall-Daten die Präzision der berechneten Position durch die primäre Datenquelle zu erhöhen.

5.3.2 Point Cloud Registration

Point Cloud Registration beschreibt ein Problem zur Schätzung der Transformation zwischen mehreren Punktewolken. Siehe Abbildung 5.5
Mit Hilfe von Punktewolken können beliebig große Mengen an Punkten dargestellt werden. Die Wolke beinhaltet Informationen zu jedem der Punkte.

Hierzu gehören zumindest die Koordinaten. Eine solche Wolke kann jedoch auch weitere Informationen wie Farbe oder Krümmung beinhalten. Die Gesamtheit der Punkte innerhalb einer solchen Wolke beschreibt die Form und Oberfläche eines Objektes [10, ch. 2.2]
Die Registration selbst lässt sich in verschiedene Unterkategorien einteilen. Beschränkt sich die Transformation auf Rotation und Translation, spricht man von einer steifen Transformation bzw. Registration. Des Weiteren wird anhand der Quelle und Anzahl der Datensätze unterschieden.
Die Punktewolken, welche Teil dieser Arbeit sind, werden mit Hilfe eines einzelnen, zwei-dimensionalen LiDAR-Sensor erstellt. Sie enthalten ausschließlich Informationen über die X und Y Koordinaten der einzelnen Punkte.
Somit spricht man bei der, im Kontext dieser Arbeit durchgeführten Point Cloud Registration, von einer paarweisen und steifen 2D Registration. Diese Art der Registration ist eine der simpelste, da lediglich zwei Punktewolken desselben Sensors miteinander verglichen werden und ausschließlich die Translation auf den zwei Achsen sowie die Rotation berechnet werden muss.



(a) Source und Target Point Cloud

(b) Transformierte Source Point Cloud über die Target Point Cloud gelegt

Abbildung 5.5: Beispiel einer starren 2D Point Cloud Registration

Ablauf

Der Ablauf einer solche Registration beinhaltet, wie in [23] beschrieben, typischerweise sechs Schritte. Siehe Abbildung 5.6

1. Datenerfassung
2. Schätzung der Keypoints
3. Schätzung der Feature-Deskriptoren

4. Schätzung der Korrespondenzen (Matching)
5. Ablehnung von Korrespondenzen
6. Schätzung der Transformation

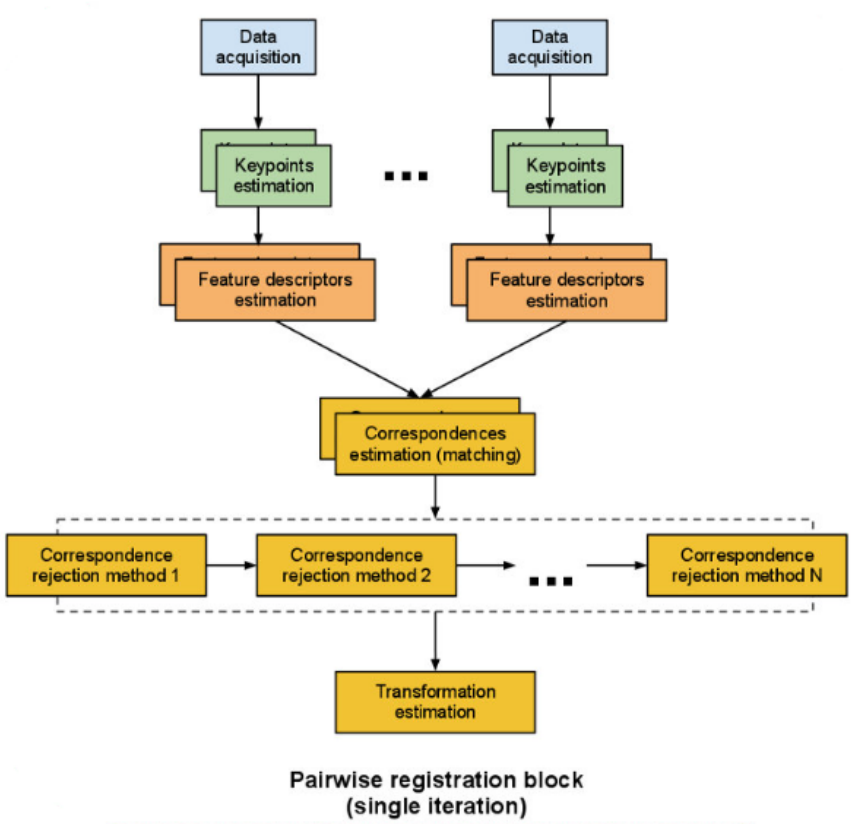


Abbildung 5.6: Typischer Ablauf einer paarweisen Registration. Quelle: [23]

Datenerfassung:
Die Datenerfassung kann auf unterschiedlichste Arten erfolgen. Hierbei werden zwei Sets an Daten, gesammelt. Die Daten werden verarbeitet und in einer Punktewolke gespeichert. Siehe Abbildung 5.5
Durch ein einheitliches Format wird sichergestellt, dass die Punkte korrekt weiterverarbeitet werden.

Schätzung der Keypoints:
Die Schätzung von Keypoints ist von enormer Wichtigkeit. Sie dient der Verringerung notwendiger Rechenleistung.

Möchte man zwei Scans mit jeweils 100 Tausend Punkten vergleichen, gibt es 10 Milliarden mögliche Korrespondenzen. Um die Zahl der Korrespondenzen zu verringern, werden Keypoints in des Scans gesucht.
Ein Keypoint beschreibt einen Punkt, welcher spezielle Eigenschaften innerhalb der Szene haben. Ein Beispiel hierfür wäre eine Ecke.

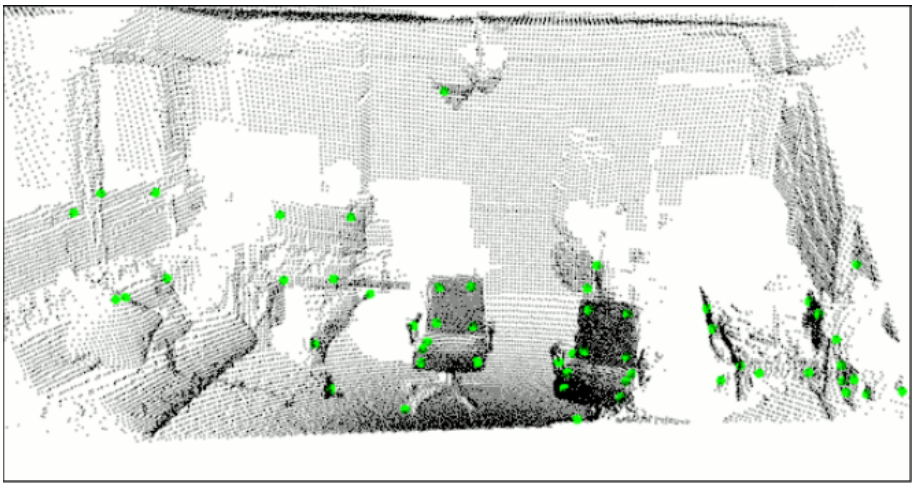


Abbildung 5.7: Keypoint Extraktion mittels Normal Aligned Radial Feature Algorithmus. Quelle: [14]

Die Keypoints werden im weiteren Verlauf für die Berechnungen genutzt, wodurch sich die Anzahl an Punkten drastisch senkt. Im Optimalfall ist das Ergebnis genau dasselbe, benötigt aber deutlich weniger Rechenleistung und somit Zeit.

Schätzung der Feature-Deskriptoren:
Je nach Anwendungszweck sind Koordinaten nicht ausreichend um einen Punkt zu beschreiben. Feature-Deskriptoren oder Point-Feature Repräsentationen sind eine Form der erweiterten Beschreibung eines Punktes.
Durch miteinbeziehen der umliegenden Punkte, können Informationen über die Form und Beschaffenheit der Fläche gesammelt werden. Diese können wiederum in den Feature-Deskriptoren gespeichert werden. Die simpelste Form eines solchen Feature-Deskriptor wäre die Normale der Fläche unter dem Punkt. Siehe Abbildung 5.8

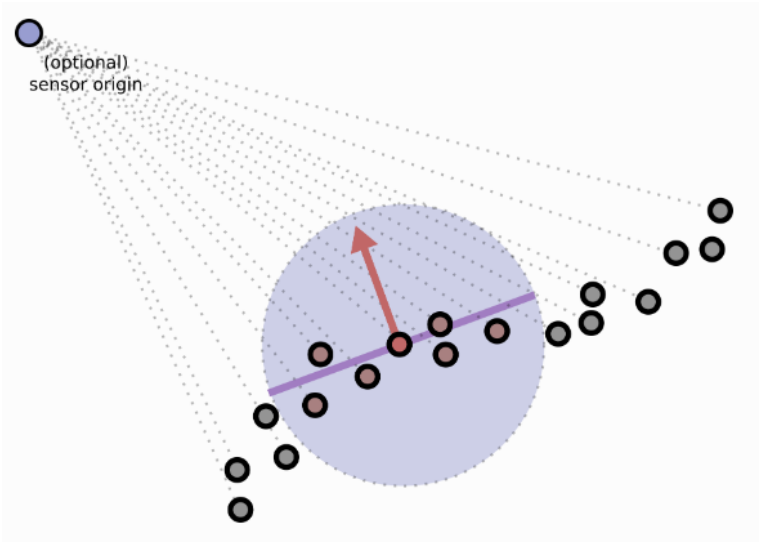


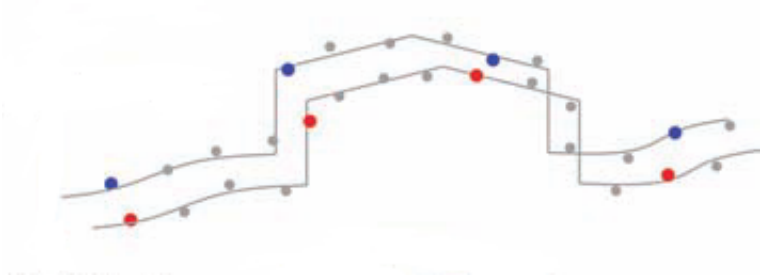
Abbildung 5.8: Beispiel für eine 2D Normalen Extraktion. Die Normale der Fläche unterhalb des roten Punktes wird mittels Nachbarpunkten bestimmt. Quelle: [4]

Schätzung der Korrespondenzen:
Die zwei vorhandenen Sets von Feature-Deskriptoren, welche aus den Keypoints berechnet wurden, können nun verwendet werden, um Korrespondenzen zu schätzen. Bei kleineren Datensets kann es auch Sinn ergeben, die Keypoint- und Feature-Deskriptor-Schätzung auszulassen und die Korrespondenzen nur mittels Koordinaten der Punkte zu schätzen. Siehe Abbildung 5.9a
Eine Korrespondenz beschreibt zwei Punkte oder Feature-Deskriptoren aus verschiedenen Datensätzen, welche den gleichen Punkt im Raum repräsentieren.

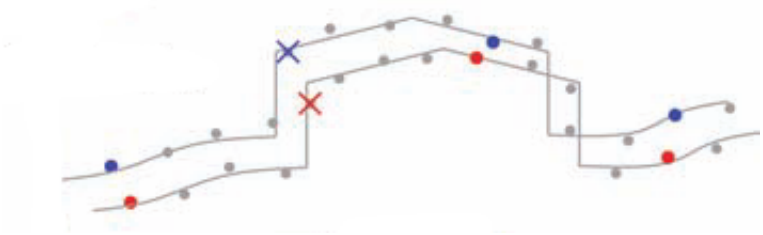
Ablehnung von Korrespondenzen:
Nachdem die Korrespondenzen geschätzt wurden, müssen schlechte Korrespondenzen verworfen werden. Siehe Abbildung 5.9b
Hierzu gibt es diverse Algorithmen, worunter der RANSAC Algorithmus am weitesten verbreitet ist. Eine weitere Möglichkeit die Anzahl an Korrespondenzen zu senken, ist das Filtern von Korrespondenzen die zwar den gleichen Source-Punkt, aber unterschiedlichen Punkten im Ziel-Datensatz korrespondieren. Hierbei kann die Korrespondenz mit der kleinsten Distanz gewählt werden. Die anderen Korrespondenzen werden verworfen.

Schätzung der Transformation:

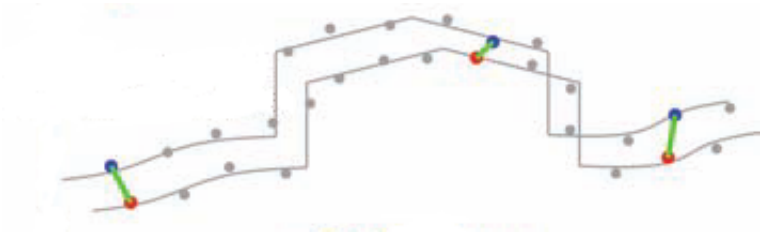
In einem Finalen Schritt wird die Transformationsmatrix geschätzt. Diese Matrix beschreibt die Translation und Rotation, welche notwendig ist um die Punktewolke A zur Punktewolke B zu transformieren.
Die Schätzung der Transformation passiert mittels, auf den Korrespondenzen basierenden, Metriken. Ein Beispiel für eine solche Metrik ist die mittlere quadratische Abweichung der Korrespondenzen. Die Punktewolke wird transformiert und die Metriken ausgewertet. Siehe Abbildung 5.9c
Dieser Vorgang wird, solange wiederholt bis ein Abbruch-Kriterium erfüllt ist. Ein solches Abbruch-Kriterium ist z.B. das Unterschreiten eines Grenzwertes für die mittlere quadratische Abweichung. Auch eine Überschreitung einer bestimmten Anzahl an Iterationen kann zum Abbruch führen.



(a) Schätzung von Korrespondenzen



(b) Ablehnung schlechter Korrespondenzen



(c) Transformieren der Source-Wolke

Abbildung 5.9: Beispiel für den Ablauf einer Iterative Closest Point (ICP)-Iteration. Quelle: [8, ch. 3]

Point Cloud Registration im Rahmen dieser Arbeit
Durch die Möglichkeit, Bewegung ausschließlich anhand zweier Scans zu berechnen, bietet sich die Nutzung der Point Cloud Registration im Rahmen dieser Arbeit an.
Die Datensätze dieser Arbeit sind mit unter 1000 Punkten recht klein. Somit ist eine Schätzung von Keypoints wenig Sinnvoll und teilweise auch nicht umsetzbar.
In der Theorie verringert das Berechnen von Feature-Deskriptoren die Laufzeit und verbessert das Ergebnis. Selbst durchgeführten Tests ergaben jedoch, dass die Laufzeit sich verschlechterte und das Ergebnis auch ohne die Nutzung von Feature-Deskriptoren ausreichend genau ist. Das lässt sich durch die geringe Anzahl an Punkten innerhalb unserer Datensätze und der Nutzung von lediglich zwei Dimensionen erklären. Eine solche Berechnung ist also, im Rahmen der Arbeit, ebenfalls wenig Sinnvoll.
Für die Schätzung der Korrespondenzen sowie der Ablehnung schlechter Korrespondenzen und der Schätzung der Transformationsmatrix wird eine Implementierung des ICP-Algorithmus der Point Cloud Library (PCL) verwendet. Die PCL ist eine C++ Bibliothek, die ein effizientes Arbeiten mit Punktewolken ermöglicht und diverse Algorithmen für unterschiedlichste Operationen mitbringt. Genauere Informationen zur Implementierung des ICP-Algorithmus Kapitel 7.2.3.

6 Simulation des Ausweichalgorithmus

In diesem Kapitel wird beschrieben, warum eine Simulation hilfreich für die Entwicklung eines sicherheitskritischen Algorithmus, wie zum Beispiel ein Ausweichalgorithmus, sein kann. Außerdem wird beschrieben, welche Aspekte der Simulation relevant für den Übertrag der Ergebnisse auf die Realität sind.

6.1 Was ist eine Simulation?

Bevor damit begonnen werden kann die verschiedenen Aspekte einer Simulation zu beleuchten, ist zu klären, was eine Simulation ist. Nach der Aussage von A. Maria ist eine Simulation eine Ausführung eines Modells eines Systems [12][p. 1, ch. 2]. Der Begriff des Modells wird ebenfalls in der Arbeit beschrieben. Ein Modell ist eine vereinfachte, funktionierende Repräsentation des Systems, das betrachtet werden soll [12][p. 1, ch. 1].
In der Simulationstechnik gibt es unterschiedliche Arten von Simulation. In diesem Kontext von Bedeutung ist die Unterscheidung zwischen realer Simulation und Computersimulation. Reale Simulationen kommen zum Einsatz, wenn durch einen Fehler keine Gefahr für Personen und Umwelt besteht. Außerdem kann es sein, dass ein Nachstellen der Umweltbedingungen so komplex ist, dass eine Nachbildung am Computer nicht ausreichend möglich oder zeitlich zu aufwendig ist. Computersimulationen kommen dann zum Einsatz, wenn ein Fehler schädliche Folgen für Personen und Umwelt herbeiführen könnten und sich die Einflussfaktoren auf das System am Computer nachahmen lassen [3]. Eine Computersimulation kann auch dann genutzt werden, wenn das Erstellen eines realen Modells nicht möglich oder nicht rentabel ist. Ein weiterer Anwendungsfall einer Computersimulation tritt ein, wenn das reale Modell noch in der Entwicklungsphase ist. In diesen Fällen stellt die Simulation sicher, dass erste Versuche mit Algorithmen, die unabhängig vom Modell funktionieren, möglich sind. Dadurch kann damit begonnen werden an Technologien und Methodiken zu arbeiten, ohne auf eine reale Umsetzung warten zu müssen.

6.2 Simulation im Kontext eines Ausweichalgorithmus

Im Rahmen dieser Arbeit hat die Implementierung einer Simulation mehrere Vorteile. Es wird parallel an der Entwicklung des Autos und der zugehörigen Software gearbeitet.
Da die Implementierung der Software von Grund auf neu gestartet wird, wäre ein Warten auf die Fertigstellung des Autos aud zeitlichen Gründen nicht

möglich. Die Situation ist also eine der Situationen die in 6.1 beschrieben sind, in denen der Einsatz einer Simulation sinnvoll ist.
Ein weiterer Grund für den Einsatz einer Simulation ist ebenfalls in 6.1 beschrieben ist, ist ein möglicher Schadensfall. Das Auto ist von den Dimensionen nicht ausreichend groß, um einem Menschen zu verletzten, daher wären bei der Nutzung des realen Fahrzeuges kein Personenschaden zu befürchten. Die Problematik in diesem Kontext ist die Anfälligkeit des Autos. Das Auto ist so konstruiert, das es mit wenig Leistung auskommt und nur die Elemente verbaut sind, dass es fahren kann. Aus diesem Grund wurden auf schützende Anbauteile wie Stoßdämpfer oder ähnliches verzichtet. Deshalb könnte eine Kollision des Fahrzeuges mit einem Hindernis problematisch. Eine Kollision könnte Schäden am Fahrzeug verursachen, die nur aufwendig, oder eventuell gar nicht repariert werden können. Da die Algorithmik in frühen Entwicklungsstufen kritische, noch unerkannte Fehler beinhalten kann, wäre es ein unnötiges Risiko die Software direkt auf dem Fahrzeuge auszuprobieren.
Ein weiterer Grund, der für den Einsatz einer Simulation spricht, ist die Abhängigkeit vom Entwicklungsfortschritt des Autos. Sollte es dazu kommen, dass das Auto nicht rechtzeitig zur Verfügung steht, kann die Algorithmik zumindest mit Hilfe der Simulation ausprobiert werden.

6.3 Aufbau der Simulation

Um die Ergebnisse und Erfahrungen der Simulation nutzen zu können, ist es wichtig, dass die Inhalte der Simulation möglichst nah an die Realität herankommen. Jede vorhandene Abweichung resultiert in erhöhtem Risiko. Ziel ist es eine Umgebung und ein Fahrzeug zu simulieren, um so die notwendigen Daten zu erhalten, die der Ausweichalgorithmus benötigt. Dabei sollen Eigenschaften des Fahrzeuges, wie der maximale Lenkwinkel, berücksichtigt werden, um eine möglichst realitätsnahe Simulation zu erhalten.
Um eine Simulation zu implementieren, ist zu klären, ob diese Funktionalität in einer Simulation realitätsnah möglich ist, oder ob eine vereinfachte Version simuliert werden muss.

6.3.1 Kritische Funktionalitäten

Zunächst müssend die Funktionalitäten identifiziert werden, die in der Simulation Probleme verursachen könnten.

1. Sensordaten
2. Umgebung

3. Ausweichen
4. Lokalisierung
5. Fahrzeug

Sensordaten
Die Sensordaten bilden die Grundlage für die gesamte Simulation. Auf den Sensordaten basiert die Lokalisierung im Raum und das Erkennen und Ausweichen eines Hindernisses. Die Simulation dieser Daten stellt damit die größte Herausforderung in der Simulation dar, da die Daten in Scan-Frequenz und Aufbau den realen Daten möglichst genau entsprechen sollten. Vor allem der Aufbau der Daten sollte den realen Daten so nahe wie möglich kommen, da zusätzliche oder fehlende Daten in der Qualität der Auswertung deutlich zu erkennen sein könnten. Außerdem bedeutet eine Abweichung in der Datenstruktur eine notwendige Anpassung der Implementierung bei einem Umstieg auf reale Daten, die nicht notwendig wäre, wenn die Datenstruktur übereinstimmen würde. Gibt es Unterschiede in der Scan-Frequenz sind die Auswirkungen weniger problematisch. Ist die Scan-Frequenz langsamer als in der Simulation, kann dies durch eine langsamere Geschwindigkeit des Fahrzeuges kompensiert werden. Eine Scan-Frequenz, die die Geschwindigkeit der Berechnungen überschreitet, kann durch das Auslassen von einzelnen Scans kompensiert werden. Die Bewegung des Fahrzeuges zwischen zwei Scans ist so gering, dass das Ignorieren von zum Beispiel jedem zweiten Scan, kaum einen Einfluss auf das Ergebnis des Algorithmus haben sollte.
Die genauen Auswirkungen von Abweichungen der Scan-Frequenz sind nicht bekannt, weswegen hier eine genaue Analyse notwendig wäre.

Umgebung
Die Umgebung ist ebenfalls ein essenzieller Bestandteil. Denn die Umgebung muss so simuliert werden, dass diese von den Sensoren erkannt werden kann. Ist das nicht der Fall, ist jede Simulation der Sensorik unbrauchbar, da dann keine Daten für den Ausweichalgorithmus zur Verfügung stehen und dann auch der simulierte LiDAR keine validen Daten liefert. Eine 2D-Simulation der Umgebung ist ausreichend, da der LiDAR 2D-Daten liefert. Unter der Voraussetzung der validen Datenerzeugung auf Basis der simulierten Umgebung, ist die genaue Implementierung der Umgebung nicht von Bedeutung.

Ausweichen
Das Ausweichen ist der zentrale Bestandteil der Software. Die Implementierung in der Simulation soll auch in der Implementierung für die Steuerung

des realen Autos zum Einsatz kommen. Der Algorithmus selbst wird nicht simuliert, aber die verwendeten Daten kommen aus der Simulation. Außerdem wird der Output des Algorithmus in der Simulation visualisiert. Daher ist es notwendig, den Output in einer Form zu generieren, dass er in der Simulation visualisiert werden kann. Zum Output gehört die aktuelle Fahrzeugposition und der berechnete Weg zum Ziel.
Das Ziel der Visualisierung ist eine optische Validierung, ob der berechnete Weg tatsächlich um die Hindernisse führt. Neben der Validierung des Weges kann auch die Berechnung der Fahrzeugposition in Ansätzen validiert werden, da erkennbar wird, ob die neue Position ungefähr dem erwarteten Wert entspricht. Die genaue Position kann durch die Visualisierung alleine nicht validiert werden.

Lokalisierung
Die Simulation der Lokalisierung ist vor allem zum Testen der Lokalisierungs-Algorithmik wichtig. Der Ausweichalgorithmus kann nur korrekt arbeiten, wenn die aktuelle Position des Fahrzeuges ausreichend genau bestimmt werden kann. Da die Lokalisierung in einer unbekannten Umgebung eine große Herausforderung ist, kann die Simulation genutzt werden, um zuverlässige Daten für den Algorithmus zu bekommen. Dies ist möglich, da die genaue Fahrzeugposition anhand der erfassten Steuerbefehle für das simulierte Fahrzeug bestimmt werden kann. Die Positions-Daten des simulierten Fahrzeugs können zudem als Richtwert genutzt werden um die Funktionalität und Genauigkeit der implementierten Lokalisierungs-Algorithmik zu testen.

Fahrzeug
Das Fahrzeug ist der wichtigste Teil der Simulation. Es beinhaltet Daten wir Position und Rotation. Außerdem ist es möglich das Fahrzeug manuell oder per Algorithmus zu steuern. Dadurch wird sowohl das präzise Erstellen von Testdaten, wie auch das Testen des Ausweichalgorithmus ermöglicht. Die visuelle Darstellung des Fahrzeuges ist rein kosmetisch und kann daher stark vereinfacht werden.

6.3.2 Prüfung der kritischen Funktionalitäten

In diesem Abschnitt werden die einzelnen Funktionalitäten auf Umsetzbarkeit geprüft. Ist eine Umsetzung möglich, kann diese Funktionalität so in der Simulation implementiert werden, andernfalls muss eine Alternative erarbeitet werden.

Sensordaten
Da aktuell nur die LiDAR-Daten genutzt werden, müssen auch nur die Daten dieses Sensors simuliert werden. Der LiDAR rotiert um 360° und sendet in bestimmten Abständen Lichtstrahlen aus. Die Höhe der gesendeten Strahlen entspricht der Höhe des LiDAR-Sensors. Wie in 3.1 beschrieben, können Hindernisse mit einer Distanz zwischen 0.15 - 12 Metern akkurat identifiziert werden. Das Datenformat und die Frequenz, mit welcher die Datensätze generiert werden, können dem Datenblatt [19] entnommen werden.
Die Frequenz kann entsprechend simuliert werden und die Daten entsprechend dem Datenblatt generiert werden. Um den Distanzbereich des realen Sensors zu simulieren ist eine entsprechende Skalierung der Simulation notwendig. Die Distanzen vom Fahrzeug bis zu den Hindernissen können über etablierte Algorithmen, wie zum Beispiel Ray-Casting, oder eine angepasste Version dieser Algorithmen realisiert werden.
Die Sensordaten können also gut simuliert werden, sodass keine Probleme entstehen sollten.

Umgebung
Die Komplexität einer simulierten Umgebung ist als gering einzuschätzen. Die Umgebung muss lediglich so implementiert werden, dass basierend darauf korrekte Sensor-Daten generiert werden können. Alle anderen Details der Implementierung für die Umgebung können stark vereinfacht werden, sodass eine Umsetzung problemlos möglich sein sollte.

Ausweichen
Das Ausweichen um Hindernisse lässt sich in einer bekannten Umgebung mit gegebenem Ziel abstrahieren. Die Abstraktion an dieser Stelle ergibt einen Path-Finding Algorithmus. Für diese Art von Algorithmen gibt es bereits viele Lösungen die unterschiedlichen Stärken haben. Basierend auf den bereits existierenden Lösungen kann ein Ausweichalgorithmus mit den genannten Einschränkungen ohne Probleme implementiert werden.

Lokalisierung
Die Simulation der Daten, welche für die Lokalisierung genutzt werden können, ist recht simpel. Das liegt daran, dass es sich bei den Daten nur um Positionsdifferenz und Rotationsdifferenz handelt. Diese sind einfach zu ermitteln, da die Position und Rotation des simulierten Fahrzeugs einfach ausgelesen werden. Somit stellt die Generierung der Bewegungsdaten kein Problem dar. Diese Daten können jedoch nur im Rahmen der Simulation verwendet werden. Aufgrund der fehlender Bewegungsdaten und begrenzter Performance,ist es

außerhalb der Simulation nicht möglich zu jeder Zeit die genaue Bewegung zu berechnen. Deshalb ist mit Abweichungen zu rechnen, die einen additiven Fehler in der berechneten Position verursachen. Bei der Bewegungsberechnung mittels Lokalisierungs-Algorithmus können die simulierten Daten als Referenzwert dienen.

Fahrzeug
Da die Simulation des Fahrzeuges ist weniger komplex und kann problemlos umgesetzt werden. Es muss lediglich beachtet werden, dass die Ansteuerung des simulierten Fahrzeugs äquivalent zu der Ansteuerung des echten Fahrzeugs ist. Außerdem ist auf eine korrekte Simulation des Lenkwinkels und somit der Kurvenfahrt zu achten.

6.3.3 Auswertung der Prüfung

Basierend auf den einzelnen Teilbereichen der Simulation ergibt sich die Einschätzung, dass die Simulation ein sinnvolles und umsetzbares Mittel in der Entwicklung einer solchen Algorithmik ist. Sie ermöglicht nicht nur das Testen der Algorithmik ohne Zugriff auf physische Hardware, sondern auch eine optimale Referenz um die Genauigkeit der Lokalisierung zu überprüfen. Außerdem bietet sie eine Möglichkeit vorhandene Daten, wie z.B. die erstellte Map und der berechnete Pfad, zu visualisieren.

7 Implementierung

In dem folgenden Kapitel wird beschrieben, wie die Theorie aus den vorherigen Kapiteln in der Implementierung umgesetzt wurden und die Implementierung an sich beschrieben. Dabei wird zunächst der oberflächliche Aufbau der Implementierung erklärt und dann auf die einzelnen Bestandteile der Implementierung eingegangen. Zur Verdeutlichung werden außerdem noch Besonderheiten aus der erstellten Implementierung aufgezeigt und deren Umsetzung beschrieben.

7.1 Aufbau der Implementierung

Der gesamte Aufbau der Implementierung ist in drei Projekte aufgeteilt: Core, Simulation und LiDAR. Das Core-Projekt ist eine Library, welche keine ausführbare Datei und lediglich die Implementierungen der Algorithmen bzw. die Logik für das Steuern und Ausweichen des Fahrzeugs enthält. Das Simulation-Projekt dient für die Simulation des autonomen Fahrzeugs und zum Testen der implementierten Algorithmen. Das LiDAR-Projekt enthält den Code, welcher auf das eigentliche Fahrzeug, bzw. den Raspberry Pi des Fahrzeugs, geladen und auf diesem ausgeführt wird. Das Simulations- und LiDAR-Projekt benutzen das Core-Projekt, um die Logik zur Steuerung des autonomen Fahrzeugs auszuführen und verwenden dazu Schnittstellen in Form von Implementierungen mehrerer Interfaces, wodurch unter anderem das Fahrzeug gesteuert und LiDAR Daten ausgelesen werden können. Nachfolgend werden die einzelnen Projekte und die Schnittstellen, in Form der Interfaces, zwischen den Projekten näher beschrieben.

7.1.1 Core-Projekt

Wie bereits beschrieben enthält das Core-Projekt die Implementierungen der verwendeten Algorithmen und die Logik zum Steuern des Fahrzeugs. Damit die Algorithmen auf Daten von Sensoren, sowie die Steuerung des Autos zuzugreifen kann, werden Interfaces als Schnittstellen verwendet, welche die Funktionen für die Algorithmen bereitstellen. Diese Interfaces werden im Core-Projekt lediglich definiert und nicht implementiert. Die Implementierung der Interfaces erfolgt in den Simulations- und LiDAR-Projekt. Dort können die Interfaces so implementiert werden, dass durch Verwendung des Interfaces die richtige Aktion im jeweiligen Projekt ausgeführt wird. So stellt z. B. das Interface zum Auslesen der LiDAR Daten im Simulations-Projekt Daten, welche den aktuellen Stand der Simulation widerspiegeln, und im LiDAR-Projekt Daten, welche über das RPLiDAR-SDK aus dem verbauten

LiDAR ausgelesen wurden, zurückgegeben werden. Damit die Implementierungen der Interfaces an das Core-Projekt übergeben werden können und abhängig von der aktuellen Verwendung des Core-Projektes die richtige Implementierung verwendet wird, wird das Dependency Injection Design Pattern angewandt. Dieses Design Pattern besagt, dass Abhängigkeiten, wie z. B. die Logik für das Auslesen der LiDAR Daten, ausgelagert und über festgelegt Schnittstellen von einem Injector zur Verfügung gestellt werden. In diesem Projekt werden die Schnittstellen in Form der Implementierungen der Interfaces bei der Initialisierung der Klasse, welche die Abhängigkeiten, also die Interfaces, benutzt von dem aufrufendem Code übergeben, welcher in diesem Fall als Injector fungiert [20].
Neben den Interfaces für das Auslesen der LiDAR Daten und der Steuerung des Fahrzeugs befinden sich außerdem noch der Ausweichalgorithmus und der SLAM-Algorithmus, welcher ebenfalls durch Interfaces abstrahiert sind, damit auch diese einfach ausgetauscht werden können. Die gesamte Logik für das autonome Fahrzeug ist in der Klasse *SelfdrivingVehicle* gebündelt. Diese besitzt eine Methode *update*, welche in der Hauptschleife des jeweiligen Programms aufgerufen wird. Der Ablauf in der *update* Methode, ist:

1. **Auslesen der LiDAR Daten**

   Hier wird über das LiDAR Interface die aktuellen Daten des LiDARs ausgelesen und für die nächsten Schritte gespeichert.

2. **Ausführen des SLAM-Algorithmus**

   Hierfür werden die ausgelesenen LiDAR Daten, sowie die Odometrie Daten übergeben. Je nach Implementierung der Interfaces sind die Odometrie Daten Leer, da sie nicht vorhanden sind, oder werden nicht für die Ausführung des SLAM-Algorithmus verwendet.

3. **Ausführen des Ausweichalgorithmus**

   Dem Algorithmus wird die Karte, die Position und die Rotation des Fahrzeugs übergeben, welche im Schritt davor durch den SLAM-Algorithmus berechnet wurde. Daraus wird mit einem zuvor definierten Ziel ein Pfad berechnet, welcher um die erkannten Hindernisse fährt. Mit dieser wird dann berechnet, wie der Motor und die Lenkung gesetzt werden, damit das Fahrzeug auf diesem Pfad fährt.

4. **Updaten der Motor- und Lenksteuerung**

   In diesem Schritt werden mit den Werten aus dem vorherigen Schritt die Steuerung für den Motor und die Lenkung geändert. Dies wird über das Interface zur Motor- und Lenksteuerung gemacht.

Da die gesamte Ausführung alle Schritte nicht jeden Durchlauf der Schleife nicht Nötig und mit der vorhandenen Hardware nicht effizient wäre, ist zusätzlich noch eine Überprüfung der Zeit, seit der letzten Ausführung, vorhanden. Die Zeitdifferenz, nach welcher die nächste Ausführung startet, kann variabel gesetzt werden, sollte aber zwischen 500 und 1000 ms betragen [src/core/src/selfdrivingVehicle.cpp].

7.1.2 Simulation-Projekt

Die Simulation dient dazu, die Logik aus dem Core-Projekt zu testen, ohne dass das eigentliche Fahrzeug benötigt wird. Damit das Fahrzeug sowohl gesteuert als auch gesehen werden kann, was durch die Algorithmen berechnet wurde. Deshalb wurde die Simulation in zwei Fenster aufgeteilt, welche bei Ausführung des Programms zusammen geöffnet werden. Das erst Fenster, nachfolgend Control Window genannt, dient zur Steuerung des Fahrzeugs und der Simulation im allgemein. Es stellt in Bezug auf das autonome Fahrzeug die Realität dar. Das zweite Fenster zeigt die Ergebnisse der Algorithmen. Dabei wird die Karte, die Position und Rotation des Fahrzeugs auf der Karte, sowie der berechnete Pfad mit dem aktuellen Ziel angezeigt. In Bezug auf das autonome Fahrzeug stellt dieses Fenster die Sicht des Fahrzeugs dar. Die Hauptschleife der Simulation, welche das Control und Visualize Window aufruft, ist in einem Simulation-Manager enthalten. Dieser dient zusätzlich auch als Einstiegs- und Endpunkt der gesamten Simulation [src/simulation/src/simulationManager.cpp]. Nachfolgend wird die Funktionen und die Logik der beiden Fenster genauer erläutert. Außerdem wird noch SFML vorgestellt, was für die Umsetzung der grafischen Benutzeroberflächliche benutzt wurde. Der gesamte Ablauf der Simulation, mit Verwendung des Core-Projektes kann im Anhang A.1 eingesehen werden.

SFML

SFML (Simple and Fast Multimedia Library) ist eine Library, mit welcher grafische Anwendungen erstellt werden können. Die Anwendung können dabei auf den gängigen Plattformen, wie Windows, Linux und MacOS laufen und können in unterschiedlichen Programmiersprachen erstellt werden, darunter auch C++ [21].

Control Window

In dem Control Window kann das simulierte Fahrzeug direkt über die *WASD*-Tasten gesteuert werden. Außerdem können weitere Hindernisse platziert und das Ziel geändert werden, zu welchem das Fahrzeug fahren soll. Die Hindernisse können durch gedrückthalten der linken Maustaste und ziehen der Maus

in der gewünschten Größe an dem gewünschten Ort platziert werden. Das Ziel kann ebenfalls durch das Benutzen der linken Maustaste an eine neue Postion platziert werden. Die beiden Funktionen können nicht gleichzeitig gemacht werden, weshalb durch Drücken der Leertaste zwischen den beiden Funktionen gewechselt werden muss.
Das Control Window enthält zwei Methoden *update* und *render*, welche in der Hauptschleife des Simulation-Managers aufgerufen werden. In der *update* Methode werden zunächst Events des Benutzers, wie die Steuerung des Fahrzeugs oder Schließen des Fensters, angerufen und verarbeitet. Danach wird die *update* Methode des *SelfdrivingVehicle* aus dem Core-Projekt aufgerufen, wodurch die Logik des autonomen Fahrzeugs ausgeführt wird. Da aktuell die Simulation ausgeführt wird, wird hierbei der simulierte LiDAR verwendet und das simulierte Fahrzeug gesteuert. In der *render* Methode werden die visuellen Komponenten, wie das Fahrzeug, die Hindernisse oder die Rays des simulierten Lidars, auf dem Fenster, über SFML, dargestellt [src/simulation/src/controlWindow.cpp].

Visualize Window
Über das Visualizer Window werden die Ergebnisse ausgeführten Logik des autonomen Fahrzeugs angezeigt. Dazu gehören die Karte, die Position und Rotion des Fahrzeugs, welche durch den SLAM-Algorithmus berechnet wurden, und das Ziel mit dem berechneten Pfad, welcher durch den Ausweichalgorithmus anhand der aktuellen Karte, Position und Rotion berechnet wurden.
Wie auch das Control Window, besitzt das Visualize Window eine *update* und *render* Methode, welche auch in der Hauptschleife des Simulation-Managers aufgerufen werden. In der *update* Methode werden lediglich die Events des Benutzers abgerufen und verarbeitet und in der *render* Methode werden wieder die visuellen Komponenten dargestellt [src/simulation/src/visualizeWindow.cpp].

7.1.3 LiDAR-Projekt

Das LiDAR-Projekt enthält das Programm, welches auf dem Raspberry Pi des autonomen Fahrzeugs ausgeführt wird und mit diesem das Fahrzeug steuert. Damit dies möglich ist, werden Implementierungen für das LiDAR Interface und das Interface zur Steuerung des Fahrzeugs benötigt. Da zum Endzeitpunkt dieser Arbeit die genau Schnittstelle mit dem Motor und der Lenkung des Fahrzeugs feststeht, ist nur die Implementierung des LiDAR Interfaces vorhanden, welche in Kapitel 7.2.2 näher beschrieben wird. Voraussichtlich wäre die Schnittstelle mit dem Motor und der Lenkung durch

das Setzen von Spannungen über die GPIO Pins des Raspberry Pis gelaufen. Hierfür könnte die Library pigpio verwendet werden. Mit den Implementierungen der Interfaces könnten daraufhin das *SelfdrivingVehicle* verwendet werden, um die Logik des Fahrzeugs auszuführen. Allerdings wurde für das Programm noch keine Lösung gefunden, wie das Ziel des Fahrzeugs an den Raspberry Pi übergeben werden kann. Deshalb würde aktuell das Ziel fest im Code stehen oder als Commandline Argument beim Ausführen des Programms übergeben werden.

7.2 Erläuterung der Implementierung

Nachdem nun der Aufbau der Implementierung erläutert wurde, werden nun einige Besonderheiten aus der Implementierung näher erläutert.

7.2.1 Simulation des LiDARs

Ein Teil der Simulation des autonomen Fahrzeugs ist die Simulation des LiDARs. Hierfür wird die Methode des Raycasting verwendet, bei welchem Strahlen, also Vektoren, von dem Zentrum des LiDARs in alle Richtungen ausgestrahlt werden und die Schnittpunkte mit den Hindernissen gespeichert werden. Umgesetzt wurde dies, indem der gesamte Umfang des LiDARs in 360 bzw. 720 Winkel unterteilt wurde, welche die Rays darstellen, und dann für jeden Ray die Schnittpunkte für alle Hindernisse berechnet wurden. Zusätzliche wurde auch der Rahmen des Fensters als hinzugefügt, da dieser die Wände des Raums und somit auch ein Hindernis darstellt. Da alle Hindernisse, eingeschlossen Fensterrahmen, Rechtecke sind, müssen also lediglich die Schnittpunkte zwischen einem Rechteck und einem Ray, also einer Geraden, berechnet werden. Dies kann weiter unterteilt werden in den Schnittpunkt zweier Geraden, da ein Rechteck aus vier Geraden besteht. Für diesen Zweck wurde eine Funktion erstellt, welche für diesen Fall einen Schnittpunkt berechnet, falls dieser existiert. In dieser wird zunächst überprüft, ob die beiden Geraden überhaupt einen Schnittpunkt besitzen oder ob diese parallel sind. Dies kann einfach über das Berechnen des Kreuzproduktes gemacht werden. Ist das Ergebnis des Kreuzproduktes gleich 0, sind die beiden Geraden parallel und es gibt keinen Schnittpunkt. Als Grundlage für eigentliche Berechnung des Schnittpunktes wurde die Formel zur Berechnung des Schnittpunktes zweier Geraden benutzt, in welcher lediglich zwei Geraden in Parameterform ($Stuetzvektor + Parameter * Richtungsvektor$) gleichgestellt werden. Da für die Berechnung der Schnittpunkte angenommen wird, dass der LiDAR bei $(0, 0)$ liegt, kann dieser in der Gleichung weggelassen werden.

$$s * \begin{pmatrix} rayDirection_x \\ rayDirection_y \end{pmatrix} = \begin{pmatrix} v1_x \\ v1_y \end{pmatrix} + t * \begin{pmatrix} v12_x \\ v12_y \end{pmatrix}$$

Aus dieser Formel wird ein lineares Gleichungssystem gemacht, welches anschließend nach dem Parameter t aufgelöst wird.

$$t = \frac{v1_x * rayDirection_x - v1_y * rayDirection_y}{v12_x * rayDirection_x - v12_y * rayDirection_y}$$

Durch Einsetzen in eine der beiden Gleichungen des LGS kann auch der Parameter s berechnet werden.

$$s = \frac{v1_x + t * v12_x}{rayDirection_x}; s = \frac{v1_y + t * v12_y}{rayDirection_y}$$

Nun muss überprüft werden, ob der Punkt nicht nur auf beiden Geraden, sondern auch zwischen den beiden Punkten des Rechtecks, sowie vor dem LiDAR liegt. Dafür wird geschaut, ob der Wert von t (Parameter für die Kante des Rechtecks) zwischen 0 und 1 liegt. Außerdem wird geschaut, ob der Wert von s (Parameter für den Ray) größer oder gleich 0 ist. Sollte beides gegeben sein, wird der Schnittpunkt zur Liste aller Schnittpunkte des aktuellen Rays hinzugefügt [src/simulation/src/intersection.cpp].

```
bool intersects(const sf::Vector2f &rayOrigin, const sf::Vector2f &
    rayDirection, const sf::Vector2f &p1, const sf::Vector2f &p2, std::
    vector<sf::Vector2f> &intersectionPoints)
{
    sf::Vector2f v1 = p1 - rayOrigin;
    sf::Vector2f v2 = p2 - rayOrigin;
    sf::Vector2f v12 = v2 - v1;

    const float cross = crossProduct(rayDirection, v12);

    if (cross == 0)
    {
        return false; // ray and edge are parallel
    }

    const float t = crossProduct(v1, rayDirection) / cross; // Parameter
        for Edge
    const float s = rayDirection.x != 0 ? ((v1.x + t * v12.x) /
        rayDirection.x) : ((v1.y + t * v12.y) / rayDirection.y); //
        Parameter for Ray

    if (t >= 0 && t <= 1 && s >= 0) // Is between points and in positive
        direction of Ray
    {
        const sf::Vector2f intersectionPoint = v1 + rayOrigin + t * v12;
        intersectionPoints.push_back(intersectionPoint);

        return true;
    }
```

```
    return false;
}
```

Listing 7.1: Berechnung des Schnittpunktes zweier Geraden

Nachdem alle Schnittpunkte berechnet wurden, muss anschließend der Schnittpunkt bestimmt werden, der am nächsten am LiDAR, also dem Stützvektor des Rays, ist. Dafür wird die Liste mit den Schnittpunkten nach dem Abstand zum LiDAR sortiert, indem der quadrierte Abstand zwischen Schnittpunkt und LiDAR berechnet. Der Vorteil des quadrierten Abstandes gegenüber dem normalen Abstand ist, dass die Reihenfolge einzelner Punkte gleich bleibt, aber die Berechnung über die Wurzel vermieden wird, welche vergleichsweise aufwendig ist [22]. Nach der Sortierung kann dann das erste Element der Liste als Schnittpunkt für diesen Ray verwendet werden. Da aktuell nur die Koordinaten des Schnittpunktes bekannt sind, ein LiDAR aber nur den Winkel und die Entfernung eines Punktes kennt, müssen für die Simulation diese Werte noch berechnet werden [src/simulation/src/intersection.cpp, src/simulation/src/lidarSensorSim.cpp].

7.2.2 Implementierung des LiDARs mit der RPLIDAR SDK

Wie bereits in der Technologie-Entscheidung beschrieben, wird für die Ansteuerung des LiDARs das RPLIDAR-SDK verwendet. Zusätzlich wird die Library pigpio benutzt, um die GPIO Pins des Raspberry Pis zu verwenden. Dies wird benötigt, da der Motor des LiDAR über ein PWM-Signal gesteuert wird. Nach dem Auslesen müssen die erhaltenen Werte vor der Weiterverwendung noch zu den richtigen Werten konvertiert werden. Zur einfacheren Übertragung werden die Werte für den Winkel und den Abstand bei das SDK als unsigned 16 bit bzw. 32 bit Integer gespeichert. Um die Werte wieder in eine Kommazahl zu konvertieren, wird der Abstand durch 4000 und der Winkel mal 90 und durch 16384 (eine 1 um 14 Stellen nach links geshifted) geteilt [7]. Mit den konvertierten Werten kann nun auch der x und y Wert des Punktes über *cos* und *sin* gerechnet werden [src/lidar/src/a1lidarSensor.cpp].

```
void A1LidarSensor::getScanData(lidar_point_t *data, size_t count)
{
    rplidar_response_measurement_node_hq_t scanData[count];
    u_result res = drv->grabScanDataHq(scanData, count);

    if (res == RESULT_OK)
    {
        printf("Grabbed scan data\n");
    }
    else
    {
        printf("Failed to grab scan data\n");
        printf("Error code: %d\n", res);
```

```
        return;
    }

    for (int i = 0; i < count; i++)
    {
        const double angle = scanData[i].angle_z_q14 * (90.f / 16384.f);
        const double distance = scanData[i].dist_mm_q2 / 4000.0f;
        data[i].radius = distance;
        data[i].angle = angle;
        data[i].x = distance * cos(angle);
        data[i].y = distance * sin(angle);
        data[i].quality = scanData[i].quality;
        data[i].valid = scanData[i].quality > 7;
    }
}
```

Listing 7.2: Auslesen der LiDAR Daten

7.2.3 Umsetzung von SLAM

Zur Umsetzung einer Lösung des SLAM Problems, welches in Kapitel 5 beschrieben wurde, sind zwei Implementierungen notwendig. Zum einen die Implementierung einer Möglichkeit eine Karte der unbekannten Umgebung aufzubauen und zum anderen die Implementierung einer Möglichkeit die Position des Fahrzeugs innerhalb dieser Karte zu bestimmen.
Die gesamte Implementierung wird durch den *SlamHandler* vom Rest des Codes abstrahiert. Somit wird für jedes Update lediglich die update-Funktion des *SlamHandler* durch das *SelfdrivingVehicle* aufgerufen. Das entsprechende Sequenzdiagramm befindet sich im Anhang. A.2

***Particle*-Klasse** Ursprünglich war geplant SLAM mittels eines Partikel Filter umzusetzen. Ein Beispiel für einen solchen Filter ist der FastSLAM-Algorithmus [13]. Dieser Plan wurde jedoch, aufgrund der hohen Komplexität eines solchen Filters und der begrenzten Rechenleistung und Zeit, verworfen. Im aktuellen Stand wird somit nur ein einzelnes Partikel bei der Initialisierung des *SlamHandler* erstellt. Das *Particle*-Objekt repräsentiert den aktuellen Zustand des Roboters. In ihm werden Position, Rotation und Karte, sowie eine Instanz des *PclHandler* gespeichert. [src/core/src/particle.cpp]
Neben Get-Methoden für die Karte, Position und Rotation gibt es Methoden zum Aktualisieren des Partikels. Diese aktualisieren sowohl die Karte, als auch die Position und Rotation des Partikels. Auf den Ablauf wird in dem Kapitel 7.2.3 eingegangen.

Erstellung einer Karte Wie bereits in Kapitel 5.2 beschrieben, wird für den Aufbau der Karte ein Occupancy Grid verwendet. Hierzu gibt es die entsprechende Klasse *OccupancyGrid*. [src/core/src/occupancyGrid.cpp]

Eine wichtige Unterscheidung ist die zwischen Karte und Grid. Die Karte selbst wird lediglich für die Berechnungen benutzt. Sie beschreibt nur Grenzwerte für Koordinaten und somit einen Bereich in dem der Roboter und die gescannten Punkte sein können.
Sämtliche Informationen über den Status der Karte an sämtlichen Koordinaten werden in dem Occupancy Grid gespeichert. Man kann sich das Grid als Etwas, dass auf der virtuellen Karte darauf liegt vorstellen. Sowohl die Maße der Karte, als auch die Maße des Grid werden mittels Konstanten in einer Settings-Datei festgelegt. [src/core/include/settings.h]
Das Grid selbst, sowie eine Kopie des Grids, welcher per shared Pointer über die Get-Methode erfragt werden kann, werden als Matrix gespeichert. Da es sich hier um sehr große Matrizen handelt, wurde für die Handhabung dieser die Bibliothek Eigen3 verwendet. Eigen ist eine vielseitig einsetzbare und schnelle Bibliothek für die Handhabung von Vektoren und Matritzen.
Die beiden Matrizen werden im Konstruktor des *OccupancyGrid* als *probMap* und *probMapCpy* deklariert und mit Nullen initialisiert.

Jedes mal wenn neue Daten mittels Scan erstellt werden, wird die Funktion *updateProbMap* aufgerufen. Hierbei werden sowohl die Scan-Daten, in Form einer Matrix mit einem Punkt pro Zeile, als auch die aktuelle Position des Roboters in der Karte und die Rotation des Roboters in Grad übergeben.
Für die Aktualisierung des Grid werden die Scan-Daten genutzt um zwei Arten von Punkten zu berechnen. Punkte an denen ein Hindernis ist und Punkte welche frei befahrbar sind. Hierzu wird die Methode *getPoints* genutzt. 7.3
Zuerst werden zwei RowMajor Matritzen deklariert. Die Option RowMajor sorgt dafür, dass die Matrix Zeilenweise in den Speicher geschrieben wird. Da jede Zeile einem Punkt entspricht, verkürzt das die Laufzeit beim Auslesen der Punkte aus der Matrix enorm.
Danach wird jeder Punkt des Scans von den Polarkoordinaten, welche der LiDAR-Scan zurückgibt, in kartesische Koordinaten umgerechnet. Hierbei werden die Punkte zusätzlich in ein globales Koordinatensystem übertragen. Dies geschieht durch die Addition der Rotation des Roboters auf den Winkel der Polarkoordinate und der Addition der Position des Roboters auf die resultierenden kartesischen Koordinaten. Die gescannten Punkte werden dann an die Matrix für die belegten Punkte angehangen.
Die frei befahrbaren Punkte sind alle die Punkte, welche sich zwischen dem Sensor und dem gescannten Hindernis befinden. Zur Berechnung dieser Punkte wird der Bresenham-Algorithmus verwendet. Der entsprechenden Methode werden die X- und Y-Koordinaten des Roboters, sowie die Koordinaten des Zielpunktes übergeben. Zurückgegeben wird eine Matrix, welche Koordinaten sämtlicher Punkte enthält, die sich auf einer Linie zwischen den übergebe-

nen Punkten befinden. Diese werden der Matrix für frei befahrbare Punkte angehangen.
Am Ende werden beide Matrizen als Pair zurückgegeben.

```
std::pair<Eigen::MatrixX2d, Eigen::MatrixX2d> OccupancyGrid::getPoints(
    const Eigen::MatrixX2d &scan, const Eigen::RowVector2d &robPos, double
    robRotAngle)
{
    // Define X,2 Matrices for occupied and free points.
    // RowMajor stores the matrix row wise in memory and is used since each
         point is represented by one row in the matrix.
    Eigen::Matrix<double, -1, 2, Eigen::RowMajor> occPoints;
    Eigen::Matrix<double, -1, 2, Eigen::RowMajor> freePoints;

    Eigen::Matrix<double, -1, 2, Eigen::RowMajor> data = scan;

    for (int i = 0; i < data.rows(); i++)
    {
        // Converts each of the polar points from the scan data into
            cartesian and stores it in occPoints
        Eigen::RowVector2d polarPoint = data.block<1, 2>(i, 0);
        Eigen::RowVector2d cartPoint = polarToCartesian(polarPoint, robPos,
             robRotAngle);
        occPoints.conservativeResize(occPoints.rows() + 1, Eigen::NoChange)
            ;
        occPoints.row(occPoints.rows() - 1) = cartPoint;

        // Calculates free points using bresenham and stores them into
            freePoints
        Eigen::MatrixX2d bresenhamPoints = bresenham(robPos[0], robPos[1],
            cartPoint[0], cartPoint[1]);
        for (int j = 0; j < bresenhamPoints.rows(); j++)
        {
            freePoints.conservativeResize(freePoints.rows() + 1, Eigen::
                NoChange);
            freePoints.row(freePoints.rows() - 1) = bresenhamPoints.block
                <1, 2>(j, 0);
        }
    }
    // Returns both matrices
    return std::make_pair(occPoints, freePoints);
}
```

Listing 7.3: Berechnung occPoints und freePoints

Die berechneten Punkte werden dann verwendet um die *probMap* bzw. das Grid zu aktualisieren. 7.4
Für jeden Punkt der Matrix mit belegten Punkten, werden X- und Y-Koordinate ausgelesen. Da die Koordinaten in der Karte liegen, das Grid jedoch kleiner ist, müssen die Koordinaten der Punkte angepasst werden. Daher werden diese durch den Faktor, um den die Map größer ist als das Grid, geteilt. Zusätzlich ist darauf zu achten, dass die Matrix per (Zeile,Spalte) adressiert wird. Da die Zeilen der y-Achse entsprechen und die Spalten der x-Achse muss die Adressierung entsprechen per (Y,X) erfolgen. Der Wert in der Matrix wird, solange er unter einem Grenzwert liegt, um ein Delta erhöht. Wird

der Grenzwert erreicht, wird der Punkt im Grid als belegt angesehen. Auch der Grenzwert und das Delta können in der oben erwähnten Settings-Datei festgelegt werden.
Derselbe Vorgang wird für die freien Punkte wiederholt. Einziger Unterschied ist, dass der Wert im Grid gesenkt statt erhöht wird. Auch hier können Grenzwert und Delta per Settings-Datei festgelegt werden.

```
// Updates values in the probMap for occupied points
for (int i = 0; i < occPoints->rows(); i++)
{
    // Gets x and y for each point from the matrix
    // Must be divided by the ratio between map and grid since the grind is
        a fraction of the size to allow for some error
    // (e.g. 1x1 in the grid is 10x10 in the map so all the points on the
        map that lay in this 10x10 area will change the value of
        probability at that single point in the grid)
    int x = occPoints->coeff(i, 0) / (MAP_WIDTH / GRID_WIDTH);
    int y = occPoints->coeff(i, 1) / (MAP_WIDTH / GRID_WIDTH);

    if ((*probMap)(y, x) < PROB_OCC_THRES)
        (*probMap)(y, x) += DELTA_OCC;
}

// Updates values in the probMap for free points
for (int i = 0; i < freePoints->rows(); i++)
{
    // Gets x and y for each point from the matrix
    // Must be divided by the ratio between map and grid since the grind is
        a fraction of the size to allow for some error
    int x = freePoints->coeff(i, 0) / (MAP_WIDTH / GRID_WIDTH);
    int y = freePoints->coeff(i, 1) / (MAP_WIDTH / GRID_WIDTH);

    if ((*probMap)(y, x) > PROB_FREE_THRES)
        (*probMap)(y, x) += DELTA_FREE;
}
```

Listing 7.4: Auschnitt aus updateProbMap

Aktualisierung der Position und Rotation Zur Aktualisierung der Position wird die Methode *updatePosition* des *Particle* aufgerufen. Dieser werden die notwendigen Werte für Positionsänderung und Rotationsänderung übergeben, welche dann auf die aktuellen Werte addiert werden.
Die Berechnung der Werte kann auf zwei Arten erfolgen. 7.5

1. **Auslesen aus der Simulation**
   Die erste Möglichkeit an die Werte zu kommen ist das Auslesen dieser aus der Simulation. Hierzu wird der Methode *update* des *Particle* nur der aktuelle Scan, sowie die Werte für Positionsänderung und Rotationsänderung übergeben. Diese wurden vorher aus der Simulation ausgelesen und über die Update-Funktion des *SlamHandler* an diesen

übergeben. Der Scan wird für das Updaten der Karte benötigt. Die Werte für Positionsänderung und Rotationsänderung werden einfach an die Methode *updatePosition* des *Particle* weitergegeben.

Diese Methode ist ausschließlich für Debugzwecke gedacht und kann nur in Kombination mit der Simulation verwendet werden.

2. **Berechnung mittels ICP**
   Die zweite Möglichkeit ist das Berechnen der Transformationsmatrix zwischen vorherigem Scan und dem Aktuellen. Hierzu wird ein ICP-Algorithmus der PCL-Bibliothek verwendet. Auch hier wird die Methode *update* des *Particle* aufgerufen. Neben dem aktuellen Scan wird diesmal jedoch auch der vorherige Scan mit übergeben. Die Werte für Positionsänderung und Rotationsänderung werden zwar ebenfalls übergeben, allerdings geschieht das nur für Debugzwecke. Die übergebenen Werte haben auf die weitere Berechnung und die Werte welche letztendlich an die *updatePosition* Methode übergeben werden keinen Einfluss.

Die Auswahl der genutzten Möglichkeit wird über ein Flag in der Header-File des *SlamHandler* ermöglicht. [src/core/include/slamHandler.h]

```
void SlamHandler::update(lidar_point_t *data, const Eigen::RowVector2d &
    positionDiff, double rotationDiff)
{
    [...]

    // If update is called the first time no lastScan exists so update of
        the particle gets skipped
    if (this->initial)
    {
        this->initial = false;
    }
    else
    {
        if (this->useOdometry)
        {
            this->particle.update(this->currentScan, positionDiff,
                rotationDiff); // Only Ododmetry, no ICP
        }
        else
        {
            this->particle.update(this->lastScan, this->currentScan,
                positionDiff, rotationDiff); // Uses ICP. Odometry data
                gets used for debug output
        }
    }

    [...]
}
```

Listing 7.5: Auschnitt aus der Methode *update* des *SlamHandler*

Umsetzung mittels ICP Die Methode zur Berechnung der Werte für Positionsänderung und Rotationsänderung, welche bei Verwendung des echten LiDAR zum Einsatz kommen soll, verwendet einen ICP-Algorithmus.
Als Interface zur Kommunikation zwischen *Particle* und der PCL-Bibliothek, gibt es die Klasse *pclHandler*. [src/core/src/pclHandler.cpp]
Durch Aufruf der Methode *computeTransformation* werden die Werte für Positionsänderung und Rotationsänderung, basierend auf den letzten beiden Scans, berechnet und als *TransformationComponents* 7.6 zurückgegeben.

```
typedef struct
{
    /// @brief A vector composed of global x and y values
    Eigen::RowVector2d translation_vector;
    /// @brief A rotation angle in deg
    double rotation_angle;
} TransformationComponents;
```

Listing 7.6: Aufbau *TransformationComponents*

Die, für den ICP-Algorithmus verwendeten Parameter, sowie typedefs, welche den Code übersichtlicher machen, befinden sich in der Header-Datei. [src/core/include/pclHandler.h]
Beim Aufruf der Methode *computeTransformation* werden die Scans, in Form zweier Matrizen mit Punkten in Polarkoordinaten, sowie die Rotation des Roboters zum Zeitpunkt des ersten Scans übergeben. Da PCL Punktewolken für sämtliche Berechnungen verwendet, müssen diese Matrizen zuerst in Punktewolken umgewandelt werden.
Hierzu werden die Punkte zuerst in ein kartesisches Koordinatensystem überführt. Dabei gilt zu beachten, dass die Rotation des Roboters auf den Winkel der Punkte addiert wird. Damit wird sichergestellt, dass die resultierenden Wolken nach den globalen Koordinatenachsen der Karte ausgerichtet sind. Genaueres hierzu in Kapitel 4.
Nachdem die Scan-Daten korrekt ausgerichtet und in ein Punktewolken-Format gebracht wurden, wird der ICP-Algorithmus durchgeführt. 7.7
Zuerst wird ein Pointer auf eine PointCloud erzeugt, in welchen die vom Algorithmus transformierte Punktewolke geschrieben wird. Dieser wird mit dem Pointer der PointCloud des ersten Scans gleichgesetzt. Das sorgt dafür, dass das Ergebnis des Algorithmus die Punktewolke des ersten Scans überschreibt. Somit wird bei der Durchführung einer weiteren Iteration des Algorithmus, das Ergebnis der vorherigen Iteration als Input genommen.
Außerdem wird die Transformationsmatrix und ein Counter initialisiert.
Danach beginnt eine do-while Schleife. So wird sichergestellt, dass der Algorithmus mindestens einmal durchlaufen wird. Sollte das Ergebnis zu ungenau sein, wird der Algorithmus erneut durchgeführt. Das geschieht so lange, bis

das Ergebnis eine gewünschte Genauigkeit erreicht hat oder die maximale Anzahl an Iterationen erreicht wurde.
Der Ablauf einer Iteration ist recht simpel. Die, in der Header-Datei festgelegten Parameter werden gesetzt. Hierbei wird die *MAX_ CORRESPONDENCE_ DISTNACE_ ICP* mit jeder Iteration verringert. Dies geschieht weil davon ausgegangen wird, dass nach jeder Iteration der Abstand zwischen den beiden Punktewolken verringert wurde.
Danach werden Input und Target des Algorithmus gesetzt und der Algorithmus wird ausgeführt. Genauere Informationen über den Ablauf des Algorithmus und die Möglichkeiten finden sich in der PCL-Dokumentation [15]. Das Ergebnis ist eine transformierte Punktewolke, welche an den dafür erstellten Pointer geschrieben wird. Im Optimalfall stimmt diese mit der Target PointCloud überein.
Als Nächstes wird die Transformationsmatrix ausgelesen. Diese beschreibt die Transformation die nötig ist um die Source Punktewolke in die, vom Algorithmus berechnete Punktewolke, zu transformieren. Sie wird mit der Transformationsmatrix der vorherigen Iteration multipliziert. Das Resultat ist eine Kombination beider Matrizen. Diese kann auf die ursprüngliche Punktewolke des ersten Scans angewendet werden, um das Ergebnis der aktuellen Iteration zu erhalten.
Zusätzlich wird noch der Counter für die Iterationen erhöht.

Nachdem eines der Abbruchkriterien der Schleife erfüllt ist, werden zusätzliche Informationen der finalen Iteration ausgelesen und ausgegeben. Einmal die Information, ob der Algorithmus überhaupt erfolgreich abgeschlossen hat. Dann der Fitness-Score des Ergebnisses. Dieser ist nichts Anderes als die mittlere, quadratische Abweichung der Konvergenzen. Außerdem wird noch die Information ausgegeben, aus welchem Grund der Algorithmus beendet hat. Mögliche Gründe sind z.B. das Erreichen der maximalen Anzahl an internen Iterationen oder das Unterschreiten eines Schwellwerts, welche über die Parameter festgelegt wurden.
Abschließend wird die finale Transformationsmatrix zurückgegeben.

```
Eigen::Matrix4f PclHandler::computeAlignment(const PointCloud::Ptr &
    sourcePoints, const PointCloud::Ptr &targetPoints)
{
    // Create ICP instance
    pcl::IterativeClosestPoint<PointT, PointT> icp;

    // Set registration output to the pointer of the source cloud
    // This overwrites the source cloud with the result of the icp so it
        can be used for another iteration
    PointCloud::Ptr registrationOutput = sourcePoints;
```

```
    // Create initial transformation matrix that does nothing
    Eigen::Matrix4f tfMatrix;
    tfMatrix << 1, 0, 0, 0,
        0, 1, 0, 0,
        0, 0, 1, 0,
        0, 0, 0, 1;

    int iterations = 0;

    // Loop ICP until fitness score is under a threshhold or other criteria
         is met
    do
    {
        // Set ICP parameter
        icp.setMaximumIterations(MAX_ITERATIONS_ICP);
        icp.setMaxCorrespondenceDistance(MAX_CORRESPONDENCE_DISTANCE_ICP -
            (iterations * 20));
        icp.setRANSACOutlierRejectionThreshold(
            OUTLIER_REJECTION_THRESHOLD_ICP);
        icp.setTransformationEpsilon(TRANSFORMATION_EPSILON_ICP);
        icp.setTransformationRotationEpsilon(ROTATION_EPSILON_ICP);
        icp.setEuclideanFitnessEpsilon(EUCLIDEAN_FITNESS_EPSILON_ICP);

        // Set ICP source and target point cloud
        icp.setInputSource(registrationOutput);
        icp.setInputTarget(targetPoints);

        // Run ICP
        icp.align(*registrationOutput);

        // Get transformation matrix and store it to tfMatrix
        // If icp is not run for the first time the result gets added onto
            the results from previous iterations
        tfMatrix = icp.getFinalTransformation() * tfMatrix;

        iterations++;

    } while (icp.getFitnessScore() > ICP_FITNESS_THRESHOLD && iterations <
        ICP_MAX_NR_CORRECTIONS);

    // Output of ICP results
    std::cout << "Converged: " << (bool)icp.hasConverged() << std::endl;
    std::cout << "Fitness: " << icp.getFitnessScore() << std::endl;
    std::cout << "Reason: " << convCrit[icp.getConvergeCriteria()->
        getConvergenceState()] << std::endl;

    return tfMatrix;
}
```

Listing 7.7: Implementierung des ICP-Algorithmus

Mittels der Methode *extractTransformationComponents* können die relevanten Werte aus der berechnete Transformationsmatrix ausgelesen und in *TransformationComponents* umgewandelt werden.
Diese werden an das *Particle* zurückgegeben und dort verwendet, um die Position zu aktualisieren.

7.2.4 Umsetzung des Ausweichalgorithmus

Ein wichtiger Teil des autonomen Fahrzeugs ist Umfahren oder Ausweichen von Hindernissen. In dieser Arbeit wird hierfür ein Pathfinding-Algorithmus verwendet, welcher mit der Karte, Position und Rotation des Fahrzeugs, welche durch SLAM mit den LiDAR Daten generiert wurde, sowie einem Zielpunkt, einem Pfad um die erkannten Hindernisse findet. Aktuell wird hierfür der A*-Pathfinding-Algorithmus verwendet. Dieser arbeitet mit einer Liste an Punkten, welche die abgelaufenen Punkte beinhaltet. Außerdem wird mit drei Arten von Kosten gearbeitet, welche einem Punkt zugewiesen werden und den Aufwand für den Pfad widerspiegeln, wenn dieser Punkt im Pfad enthalten ist:

1. **g-Cost: die bisherigen Kosten**

   Die Kosten, die benötigt werden, um zu diesem Punkt zu gelangen.

2. **h-Cost: die geschätzten Kosten**

   Die Kosten, die voraussichtlich noch zum Ziel auftreten. Hierfür wird eine Heuristik-Funktion zur Berechnung der Kosten verwendet. Diese rechnet den Abstand zwischen dem aktuellen Punkt und dem Ziel aus. Hierfür wird in der Regel die Euklidische Distanz (Luftlinie) oder die Manhattan Distanz (Summe der absoluten Differenzen) verwendet.

3. **f-Cost: die gesamten Kosten**

   Die Kosten, die entstehen, wenn g-Cost und h-Cost summiert werden.

Bei einer Schleife wird der Punkt mit dem niedrigsten f-Cost aus der Liste genommen und überprüft, ob dieser dem Ziel entspricht. Sollte dies der Fall sein, kann der Algorithmus abbrechen und den Pfad rekonstruieren. Entspricht der Punkt nicht dem Ziel, werden die Nachbarn des Punktes zur Liste hinzugefügt, falls an diese keine Hindernisse sind. Für die Nachbarn werde die Kosten berechnet, wobei für g-Cost die g-Cost des aktuellen Punktes genommen und um eins erhöht wird. Außerdem wird der aktuelle Punkt als Parent-Punkt alle Nachbarn gesetzt. Die Schleife wird nun so lange wiederholt, bis keine Einträge mehr in der Liste sind oder der Punkt für das Ziel gefunden ist. Um den Pfad zu rekonstruieren wir von dem Ziel die jeweiligen Parent-Punkte nachgefahren, bis der Startpunkt erreicht wurde [9].
Umgesetzt wurde der Algorithmus mit einer Priority Queue, welche immer das Element mit der geringsten f-Cost zurückgibt. Außerdem wird jeweils eine Matrix für f-Cost, g-Cost und das Festhalten der Parent-Punkten benutzt. In der Schleife werden Nachbarn nur zur Priority Queue hinzugefügt, wenn dies

innerhalb der Karte und der Wert an diesem Punkt kleiner als $PROB\_OCC$ ist, was bedeutet, dass dort wahrscheinlich kein Hindernis vorhanden ist. Da ein Punkt auch über mehrere Punkte angefahren werden kann, wird der Punkt nur in die Priority Queue hinzugefügt, wenn der g-Cost größer ist, als der aktuell in der Matrix hinterlegte Wert [src/core/src/evasionAStar.cpp].

```
void EvasionAStar::execute()
{
    size_t rows = this->map->rows();
    size_t cols = this->map->cols();
    auto cmp = [](std::pair<double, Eigen::RowVector2d> left, std::pair<
        double, Eigen::RowVector2d> right)
    { return left.first > right.first; };
    std::priority_queue<std::pair<double, Eigen::RowVector2d>, std::vector<
        std::pair<double, Eigen::RowVector2d>>, decltype(cmp)> openSet(cmp)
        ;

    Eigen::MatrixXd gScore = Eigen::MatrixXd::Constant(rows, cols, INF);
    Eigen::MatrixXd fScore = Eigen::MatrixXd::Constant(rows, cols, INF);
    Eigen::Matrix<Eigen::RowVector2d, Eigen::Dynamic, Eigen::Dynamic>
        parent(rows, cols);

    openSet.push({0, this->origin});
    gScore(ROUND(this->origin.x()), ROUND(this->origin.y())) = 0;
    fScore(ROUND(this->origin.x()), ROUND(this->origin.y())) = this->
        heuristic(this->origin, this->destination);
    parent(ROUND(this->origin.x()), ROUND(this->origin.y())) = {-1, -1};

    while (!openSet.empty())
    {
        Eigen::RowVector2d current = openSet.top().second;
        openSet.pop();

        if (current == this->destination)
        {
            while (current != Eigen::RowVector2d(-1, -1))
            {
                this->path.push_back(current);
                current = parent(ROUND(current.x()), ROUND(current.y()));
            }

            std::reverse(this->path.begin(), this->path.end());

            return;
        }

        std::vector<Eigen::RowVector2d> neighbors = {
            current + Eigen::RowVector2d(-1, 0),
            current + Eigen::RowVector2d(1, 0),
            current + Eigen::RowVector2d(0, -1),
            current + Eigen::RowVector2d(0, 1),
        };

        for (const auto &neighbor : neighbors)
        {
            size_t x = ROUND(neighbor.x());
            size_t y = ROUND(neighbor.y());

            if (x >= 0 && x < rows && y >= 0 && y < cols && this->isFree(x,
```

```
                y))
            {
                double tentativeGCost = gScore(ROUND(current.x()), ROUND(
                    current.y())) + 1;

                if (tentativeGCost < gScore(x, y))
                {
                    parent(x, y) = current;
                    gScore(x, y) = tentativeGCost;
                    fScore(x, y) = gScore(x, y) + this->heuristic(neighbor,
                        this->destination);
                    openSet.push({fScore(x, y), neighbor});
                }
            }
        }
    }
}
```

Listing 7.8: Pathfinding mit A*

Obstacle Inflation

Damit das autonome Fahrzeug auch wirklich den Weg fahren kann und z. B. nicht bei einer Lücke zwischen zwei Hindernissen stecken bleibt, wird eine Methode benutzt, bei welcher die Hindernisse künstlich vergrößert werden. Diese Methode wird auch Obstacle Inflation genannt. Angewendet wird diese Methode, indem vor dem eigentlich Pathfinding-Algorithmus durch dir Karte iteriert wird und bei allen Punkten in der Karte, die einen Wert größer als $PROB\_OCC$ haben, also an welchen sich wahrscheinlich ein Hindernis befindet, die umliegenden Punkte den künstlichen Wert $INFLATED$ bekommen. Dabei gilt $PROB\_OCC > INFLATED > 0$. Dieser Wert darf aber nur für Punkte gegeben werden, die einen Wert kleiner als $PROB\_OCC$ haben, also an welchen sich wahrscheinlich kein Hindernis befindet. Ansonsten würde die Information über das Vorkommen er richtigen Hindernisse verloren gehen [5]. Damit nun der Pfad nicht durch die künstlichen Hindernisse geht, muss bei der Ausführung des Pathfinding-Algorithmus in der Überprüfung, ob der Punkt belegt ist, geschaut werden, ob der Wert an dem Punkt größer $INFLATED$ und nicht mehr größer $PROB\_OCC$ ist [src/core/src/evasionControl.cpp].

```
void EvasionControl::inflateObstacles()
{
    const size_t rows = this->map->rows();
    const size_t cols = this->map->cols();

    for (size_t i = 0; i < rows; ++i)
    {
        for (size_t j = 0; j < cols; ++j)
        {
            if ((*this->map)(i, j) >= PROB_OCC)
            {
```

```
                for (int x = -VEHICLE_RADIUS; x <= VEHICLE_RADIUS; ++x)
                {
                    const int circleValue = std::round(VEHICLE_RADIUS * std
                        ::cos(std::abs(x) * M_PI / (2 * VEHICLE_RADIUS)));
                    for (int y = -circleValue; y <= circleValue; ++y)
                    {
                        int nx = i + x;
                        int ny = j + y;
                        if (nx >= 0 && ny >= 0 && nx < rows && ny < cols &&
                            (*this->map)(nx, ny) < PROB_OCC)
                        {
                            (*this->map)(nx, ny) = INFLATED;
                        }
                    }
                }
            }
        }
    }
}
```

Listing 7.9: Obstacles Inflation

Natürliches Fahrtverhalten

Die aktuelle Implementierung des A*-Algorithmus, ergibt zwar einen Pfad, welcher nicht über Hindernisse und, durch Ostacle Inflation, zu nah an einem Hindernis vorbeigeht, jedoch ist der resultierende Pfad für das autonome Fahrzeug dennoch nicht befahrbar. Das liegt daran, dass die jeweilige aktuelle Ausrichtung des Fahrzeugs nicht beachtet wird und dadurch ein Pfad resultiert, bei welchem das Fahrzeug sich auf der Stelle drehen müsste. Da dies nicht möglich ist, muss eine Optimierung des Algorithmus vorgenommen werden. Diese Optimierung kann gemacht werden, indem neben den x und y Koordinaten noch eine weitere Dimension hinzugefügt wird, welche die aktuelle Rotation enthält, die das Fahrzeug an diesem Punkt hätte. Außerdem werden nicht alle umliegenden Nachbarn zu der Liste hinzugefügt, sondern Punkte vor und hinter dem Fahrzeug, welche mit dem maximalen Lenkwinkel erreicht werden könnten. Da dies unendlich viele Punkte sind, sollten hier die Punkte ohne Lenkung und mit voller Lenkung nach rechts bzw. links gewählt werden. Für die Nachbarn wird anschließend die Rotation berechnet, die das Fahrzeug hätte, wenn es zu diesem Punkt gefahren wäre [2]. Aus zeitlichen Gründen konnte dieses Konzept nicht funktionieren und effizient umgesetzt werden [src/core/src/evasionAdvancedAStar.cpp (tree advanced-pathfinding)].

Ansteuerung des autonomen Fahrzeugs

Nachdem ein valider Pfad erstellt wurde, muss dieser mit Fahrzeug abgefahren wurde. Dafür kann das *VehicleActuator* Interface benutzt werden, mit

welchen das echte bzw. simulierte Fahrtzeug gesteuert werden kann. Mit diesem kann zunächst ein Wert für die Lenkung bzw. den Motor gesetzt werden, welcher signalisiert, wie stark der Motor oder die Lenkung angesteuert werden soll. Anschließend wird in der *update* Methode des *SelfdrivingVehicle* die *update* Methode des *VehicleActuator* aufgerufen, in welcher anhand der gesetzten Werte, die Spannung oder das PWM Signal an den Pins setzt. In der Simulation wird hier lediglich der Punkt bzw. die Rotation des simulierten Fahrzeugs verändert [src/core/include/vehicleActuator.h]. Da über den aktuellen Ausweichalgorithmus, ein Pfad berechnet wird, welcher keinem natürlichen Fahrtverhalten entspricht, ist die Konvertierung von berechnetem Pfad zu Ansteuerungsanweisung noch nicht vorhanden.

8 Fazit

Im Folgenden werden die Ergebnisse der Arbeit nochmal aufgezählt, erläutert und evaluiert. Des Weiteren wird auf Möglichkeiten zur Optimierung und sinnvolle Änderungen eingegangen.

8.1 Rückblick

In diesem Kapitel wird auf die diversen Probleme eingegangen, welche während der Bearbeitung der Studienarbeit aufkamen. Zusätzlich wird auch darauf eingegangen, was für Ergebnisse erzielt wurden und ob die Ziele der Arbeit erreicht sind.

Probleme
In sämtlichen Bereichen der Studienarbeit kam es zu Problemen, welche versucht wurden zu lösen.

1. **Kommunikation mit dem LiDAR**
   Das erste größere Problem welches gelöst werden musste, war die Kommunikation mit dem LiDAR. Der LiDAR kommuniziert über eine serielle Schnittstelle. Zusätzlich gibt es einen Pin zur Steuerung der Motordrehzahl. Siehe Abbildung 8.1

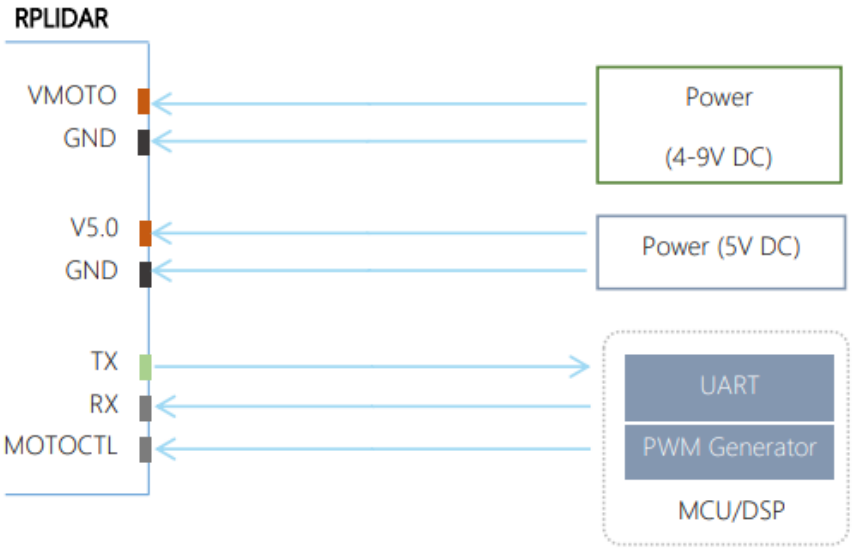


Abbildung 8.1: RPLiDAR A1 Pins Reference Design. Quelle: [18, p. 12]

Standardmäßig liegt dem Slamtec RPLiDAR A1M8-R6 ein Adapter bei. Dieser ermöglicht es, über Standard USB-A, mittels SDK mit dem LiDAR zu kommunizieren und auch den Motor des LiDAR anzusteuern. Bei unserem Exemplar des LiDAR lag ein solcher Adapter jedoch nicht bei. Anstelle des Adapters stand lediglich ein Kabel zur Verfügung, dass an den LiDAR angeschlossen werden konnte. Das andere Ende des Kabels war nicht terminiert, sodass die einzelnen Kabel direkt an den Pins des Raspberry Pi angeschlossen werden mussten.

Zu Beginn funktionierte die serielle Kommunikation überhaupt nicht. Es stellte sich jedoch raus, dass die Kabel lediglich an den falschen Pins des Raspberry Pi angeschlossen wurden. Durch korrektes Anschließen der Kabel zur seriellen Kommunikation wurde das Problem gelöst.

Eine weitere Folge des fehlenden Adapters war, dass die Motorsteuerung mittels SDK nicht länger möglich war. Um den Motor zu steuern, musste ein entsprechendes PWM-Signal an dem korrekten Pin des Raspberry Pi manuell gesetzt werden. Zur Vereinfachung dieses Prozesses wurde ein Interface implementiert, welches den Motor bei Bedarf automatisch startet und stoppt.

2. **Simulation des LiDAR**
   Das größte Problem, welches bei der Entwicklung der Simulation auftrat, war die Umsetzung der LiDAR Simulation. Da ein solcher Sensor mit Lichtstrahlen arbeitet, erschien eine Umsetzung mittels Ray Castig am sinnvollsten. Die Simulation der Umgebung und den darin enthaltenen Hindernissen funktionierte bereits. Jedoch gab es Probleme bei der Berechnung der Schnittpunkte der simulierten Lichtstrahlen und den Hindernissen. Durch eine fehlerhafte Umstellung der Formel entstand ein Bug, wodurch manche Schnittpunkte nicht korrekt berechnet wurden. Durch umfangreiches Debugging konnte der Fehler in der Formel gefunden und behoben werden.

3. **ICP Performance**
   Ein weiteres Problem war die schlechte Performance der zuerst verwendeten Implementierung eines Iterative Closest Point (ICP)-Algorithmus. Die verwendete Implementation hatte seit sieben Jahren kein Update erhalten und entstand vermutlich als Projekt einzelner Personen. Trotz der geringen Anzahl an Punkten, aus denen die einzelnen Scans bestanden, brauchte der Algorithmus teilweise mehrere Sekunden zur Berechnung der Transformationsmatrix. Da der gesamte Ausweichalgorithmus eine Laufzeit von unter 500ms haben sollte, war die Laufzeit des ICP-Algorithmus zu hoch. Ein erster Versuch die Laufzeit zu verbessern bestand darin, die

Anzahl an Punkten zu verringern. Hierzu wurde sich dazu entscheiden die n Punkte zu verwenden, die am nächsten sind. Das hatte jedoch zur Folge, dass das Ergebnis des Algorithmus deutlich ungenauer wurde. Zudem entstand durch die geringe Anzahl an Punkten, welche sich zudem meistens nur in einem kleinen Bereich des 360° Sichtfeld des Sensors befanden, eine hohe Fehleranfälligkeit bei Bewegungen nahe am Sensor. Diese entstanden vor allem dann, wenn das Fahrzeug sich einem Hindernis näherte.

Um das Problem zu lösen wurde sich dazu entschieden die Implementation der Point Cloud Library zu nutzen. 5.3.2 Da diese open-source Bibliothek für die Arbeit mit Punktewolken gedacht ist, bietet sie verschiedene ICP-Algorithmen, welche nicht nur optimiert sind, sondern auch deutlich mehr Konfigurationsmöglichkeiten als die ursprünglich verwendete Implementation bieten.

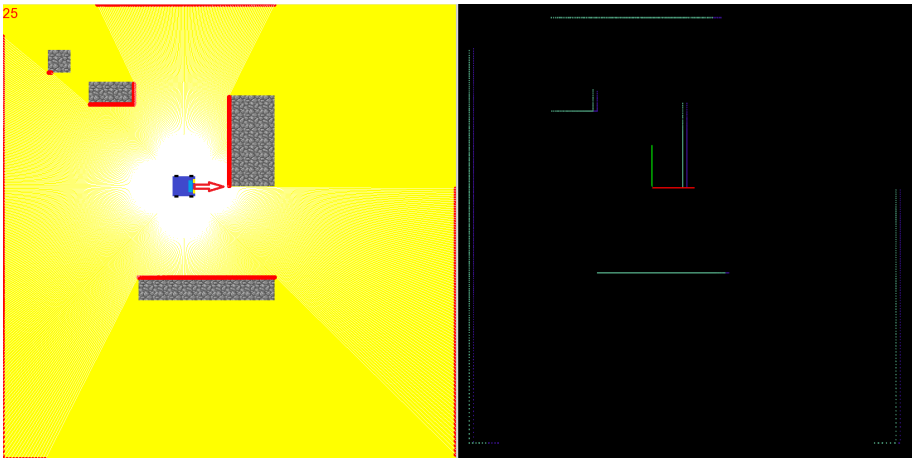
Durch die Nutzung der PCL-Implementation konnte die Laufzeit erheblich verbessert werden, somit konnte das Ziel von einer Laufzeit unter 500ms, selbst bei Nutzung von 720 Punkten pro Scan, erreicht werden.

4. **Koordinatensysteme**
   Zusätzlich zu dem Problem der Laufzeit, gab es das Problem, das die Werte für die X und Y Translation, welche der ICP-Algorithmus berechnete, ungenau waren. Vor allem bei Bewegungen des Fahrzeug auf der Y-Achse unterschieden sich die berechneten Werte deutlich von den tatsächlichen Werten.

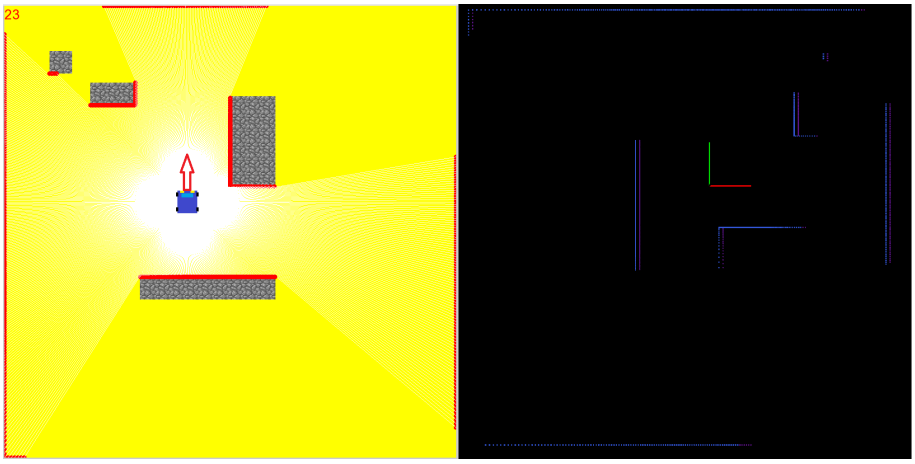
   Ausführliches Debugging ergab, dass der ICP-Algorithmus die Punktewolken nahezu perfekt übereinander legte. Das Problem war jedoch, das die resultierenden Werte für X- und Y-Translation basierten auf dem lokalen Koordinatensystem des Autos. Das kommt daher, dass das Auto die Scans erzeugt, diese jedoch keine Informationen über die aktuelle Rotation des Fahrzeugs, relativ zum globalen Koordinatensystem der Karte, enthalten.

   So lange sich das Auto nicht drehte und nur vorwärts oder rückwärts fuhr, stimmten die Koordinaten-Achsen des Autos mit den gloablen Koordinaten-Achsen überein. Somit stimmte auch das Ergebnis des ICP-Algorithmus. Siehe Abbildung 8.2a

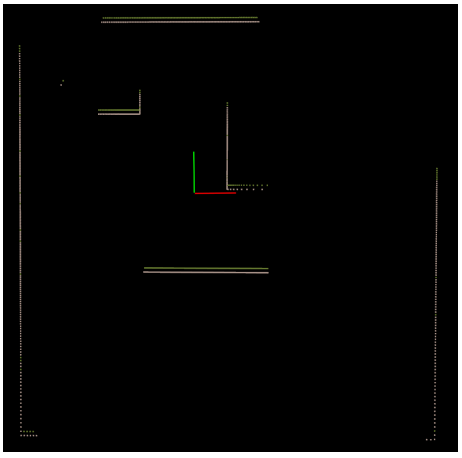
   Fuhr das Auto jetzt aber vorwärts entlang der y-Achse, also um 90 Grad verdreht nahm der ICP-Algorithmus das als Bewegung auf der x-Achse war, siehe Abbildung 8.2b. Das lag daran, dass er das lokale Koordinatensystem des Autos nutzte, bei der die x-Achse nach vorne (0 Grad) gerichtet ist. Sämtliche, nach vorne gerichtete Bewegung, wurde also als Bewegung in X-Richtung interpretiert.



(a) Umgebung und Source- und Target-Wolke bei Bewegung vorwärts entlang der x-Achse



(b) Umgebung und Source- und Target-Wolke bei Bewegung vorwärts entlang der y-Achse



(c) Source und Target Punktewolke - Bewegung entlang der y-Achse nach Implementierung des Fix

Abbildung 8.2: Vergleich der Bewegung in X und Y Richtung vor und nach dem Fix.

Das Problem wurde gelöst, indem die Punktewolken, um die Rotation des Autos zum Zeitpunkt der Erstellung der ersten Punktewolke, gedreht wurden. Dies hatte zur Folge, dass das Koordinatensystem des Scans auf das globale Koordinatensystem gelegt wurde. Dadurch stimmte die berechnete Translation wieder mit den realen Werten überein. Siehe Abbildung 8.2c

5. **ICP Genauigkeit**
   Ein weiteres Problem bei der Implementierung des ICP-Algorithmus war die Genauigkeit des Ergebnisses. Die anfänglich eher schlechte Genauigkeit, konnte durch Anpassen der Konfigurations-Parameter erheblich verbessert werden. Hierzu wurden verschiedene Werte in unterschiedliche Situationen getestet und ausgewertet. Hierbei war es wichtig die Laufzeit des Algorithmus im Auge zu behalten. Je enger die Parameter gesetzt werden, desto genauer ist das Ergebnis des Algorithmus, jedoch steigt dadurch auch die Menge an Iterationen und somit die Laufzeit.

   Es fiel auf, das der ICP-Algorithmus, selbst mit etwas lockerer gesetzten Parametern ausreichend gute Ergebnisse liefert. Nur in seltenen Fällen war ein Ergebnis suboptimal, was sich durch einen hohen Fittness-Wert des Ergebnisses auszeichnete. Um die Laufzeit weiter zu verbessern wurde eine zusätzliche Schleifen, um den ICP-Algorithmus herum implementiert 7.2.3. Diese hat den Fitness-Score des Iterations-Ergebnises als Abbruchkriterium. Somit wird der ICP-Algorithmus mindestens einmal durchgeführt. Sollte das Ergebnis nicht genau genug sein, wird der Algorithmus ein weiteres Mal aufgerufen. Diesmal mit dem vorherigen Ergebnis als Input. Des Weiteren werden mit jeder Iteration die Parameter enger gesetzt.

   Leider kam es in seltenen Fällen dazu, dass die Berechnung auf diese Art und Weise, mehrere Sekunden dauerte. Daher wurde eine maximale Anzahl an ICP-Iterationen festgelegt.

   Somit ist die Laufzeit des Algorithmus weiterhin optimal, jedoch kann es dazu kommen, das einige Ergebnisse einen nicht zu vernachlässigbaren Fehler haben. Auch die Ergebnisse, welche eine ausreichende Genauigkeit haben, sind nicht zu 100 Prozent genau. Da mehrere Scans pro Sekunde verglichen werden, addiert sich dieser Fehler. Zudem resultiert ein Fehler in der berechneten Rotation des Fahrzeugs dazu, dass das oben beschriebene Problem mit den Koordinaten-Achsen nicht mehr so gut ausgeglichen werden kann. Das hat zur Folge, dass sich der Fehler exponentiell erhöht.

   Eine mögliche Lösung für dieses Problem wären die Verwendung weiterer Sensoren zum Messen der Bewegung des Fahrzeugs. Durch zusätzliche Odometrie-Daten und einer Gewichtung der Ergebnisse kann die Lokalisierung des Fahrzeugs erheblich verbessert werden. Eine weitere Lösung wäre

die Integration von Filtern, welche die Ergebnisse des ICP-Algorithmus filtern.

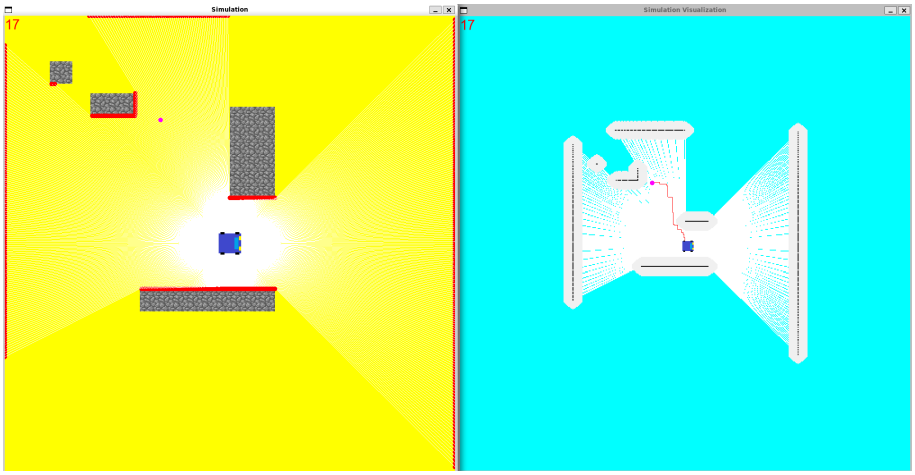
Zum jetzigen Zeitpunkt sind jedoch keine Lösungen für das Problem implementiert. Der Fehler ist, vor allem zu Beginn, vernachlässigbar und beschränkt sich auf die Position und Rotation des Fahrzeugs. Da sich der LiDAR auf dem Fahrzeug befindet, ist die Erkennung von Hindernissen unabhängig von der Position des Fahrzeugs. Lediglich die Erstellung der Karte und somit auch die Navigation zum Zielort wird mit der Zeit ungenauer.

6. **Ausweichalgorithmus**
   Die Implementierung des Ausweichalgorithmus war ebenfalls herausfordernd.

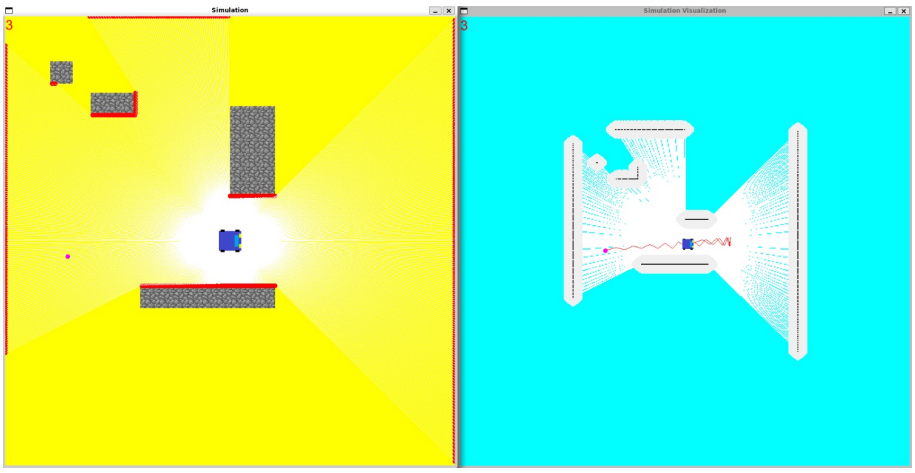
   Zur Umsetzung wird der A-Stern Path-Finding-Algorithmus verwendet. Anfänglich nahm dieser keine Rücksicht auf die eingeschränkten Bewegungsmöglichkeiten des Autos. Somit wurden Pfade berechnet, welche z.B. Zick-Zack-Bewegungen enthielten. Da sich das Auto nicht auf der Stelle drehen kann, waren diese Pfade nicht für das Auto abfahrbar. Siehe Abbildung 8.3a.

   Dieses Problem konnte nicht vollständig behoben werden. Es wurde versucht, den Winkel, in dem sich die Punkte befinden, welche der Algorithmus als nächsten Schritt in Erwägung zieht, einzuschränken. Das resultierte darin, dass der Pfad nahe am Auto, den Lenkwinkel berücksichtigt. Durch Fehler in der Logik verhält sich der Pfad jedoch nach kurzer Zeit unvorhersehbar. Siehe Abbildung 8.3b.

   Dadurch entstehen unmögliche Pfade weshalb die neue Logik, zum aktuellen Zeitpunkt, noch rein experimentell und für die Anwendung ungeeignet ist.



(a) Berechneter Pfad mit der alten Logik



(b) Berechneter Pfad mit der neuen Logik

Abbildung 8.3: Vergleich der Pfadberechnung vor und nach dem Versuch eines Fix

Des Weiteren neigte der Algorithmus dazu, Pfade zu generieren, welche durch Wände führten. Wie sich herausstellte, war das Problem die Darstellung der Koordinaten. Die Implementierung des Algorithmus repräsentierte Punkte in der Form (x,y). Die Werte, welche zeigen, ob sich an einem Punkt ein Hindernis befindet, werden jedoch in einer Matrix gespeichert. Diese wird in der Form (Zeile, Spalte) angesprochen. Sieht man die Matrix nun als Koordinaten-System entsprechen die Zeilen der, nach unten gerichteten y-Achse. Die Spalten entsprechen der, nach rechts gerichteten x-Achse. Somit muss, um den korrekten Wert aus der Matrix auslesen zu können, diese in der Form (y,y) angesprochen werden. Ein

einfaches Tauschen der Koordinaten bei der Abfrage des Matrix-Wertes löste das Problem.

Ergebnisse
Die Umsetzung der Studienarbeit erzielte diverse Ergebnisse auf welche im Folgenden genauer eingegangen werden soll.

1. **Simulation**
   Das Fahrzeug selbst wird so simuliert, dass es über entsprechende Steuerungsbefehle gesteuert werden kann. Des Weiteren wird auf den maximalen Lenkwinkel Rücksicht genommen.

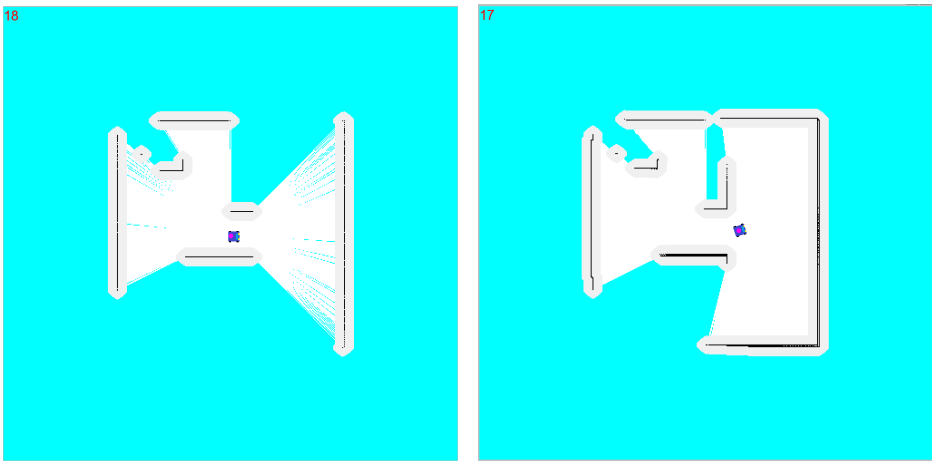
   Ein Kollisionserkennung sowie manuelle Steuerung wurde ebenfalls, zu Debug-Zwecken, implementiert. Eine statische Umgebung kann ebenfalls erfolgreich simuliert werden. Neben den zum Start bereits vorhandenen Hindernissen können weitere Hindernisse manuell hinzugefügt werden.

   Der LiDAR-Sensor kann ebenfalls realitätsnah simuliert werden. Die Anzahl an Punkten pro Scan lässt sich variabel einstellen. Die Scan-Daten des Sensors lassen sich entweder manuell per Knopfdruck in einer Datei speichern, oder über eine Schnittstelle in Form einer Matrix auslesen. Die hierbei ausgelesene Matrix hat die gleiche Datenstruktur wie die Matrix, die als Ergebnis eines Scans mit dem realen Sensor über dessen Schnittstelle zurückgegeben wird.

2. **Map-Erstellung**
   Die Sensordaten können genutzt werden, um eine Map aufzubauen. Basierend auf einkommenden Daten, werden Werte innerhalb der Felder eines Grids angepasst. Somit kann erfasst werden, an welchen Stellen der Karte ein Hindernis erkannt wurde und an welchen Stellen frei ist.

   Die Größe der Karte, sowie die Größe des Grids können beliebig angepasst werden. Eine zu große Karte hat jedoch deutlich längere Laufzeiten zu Folge.

   Sofern die Fahrzeug-Position zum Zeitpunkt des Scans bekannt ist, kann die Karte beliebig erweitert werden. Hierbei ist darauf zu achten, dass die Fahrzeug-Position nicht außerhalb der Karte liegt. Siehe Abbildung 8.4



(a) Map zum Zeitpunkt t1 (b) Map zu einem späteren Zeitpunkt t2

Abbildung 8.4: Beispiel einer Erweiterungen der Karte durch Bewegung des Fahrzeuges

3. **Lokalisierung**
   Zur Lokalisierung des Fahrzeugs wird ein ICP-Algorithmus der Point Cloud Library verwendet. Dieser berechnet, basierend auf zwei aufeinanderfolgenden Scans, die Positionsdifferenz des Fahrzeuges zwischen den Scans. Somit kann, sofern der Startpunkt bekannt ist, die Fahrzeug-Position nach jedem Scan angepasst werden.

   Aufgrund der Frequenz, mit welcher die Scans erstellt werden, ist die Differenz recht gering. Das hat zur Folge, dass der ICP-Algorithmus sehr schnell und sehr präzise arbeiten kann.

   Wie bereits in 8.1 beschrieben ist der Algorithmus nicht zu 100 Prozent akkurat. Aufgrund fehlender Filter und Bewegungsdaten des Fahrzeugs kann dieser Fehler nicht korrigiert werden. Somit addiert er sich immer weiter auf. Das Ergebnis ist eine, immer weiter von der Realität abweichende, berechnete Position und Rotation des Fahrzeugs.

   Da die Hindernisse basierend auf der berechneten Position und Rotation des Fahrzeuges in die Map eingetragen werden, sind die Abstände zu den Hindernissen weiterhin akkurat. Die resultierende Map ist jedoch verzerrt was zu Problemen bei der Navigation führen kann.

4. **Ausweichen**
   Die Möglichkeit auszuweichen wurde mittels Path-Finding und der generierten Karten umgesetzt.

Dem Path-Finding-Algorithmus wird die aktuelle Position des Roboters, sowie die Zielkoordinate übergeben. Daraufhin wird, mit Hilfe der Map-Daten, ein Pfad um die Hindernisse herum berechnet. Siehe Abbildung 8.5

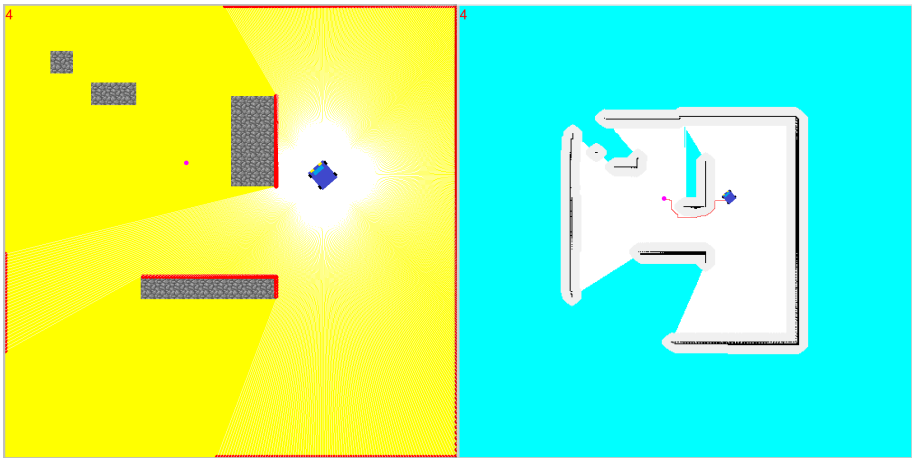
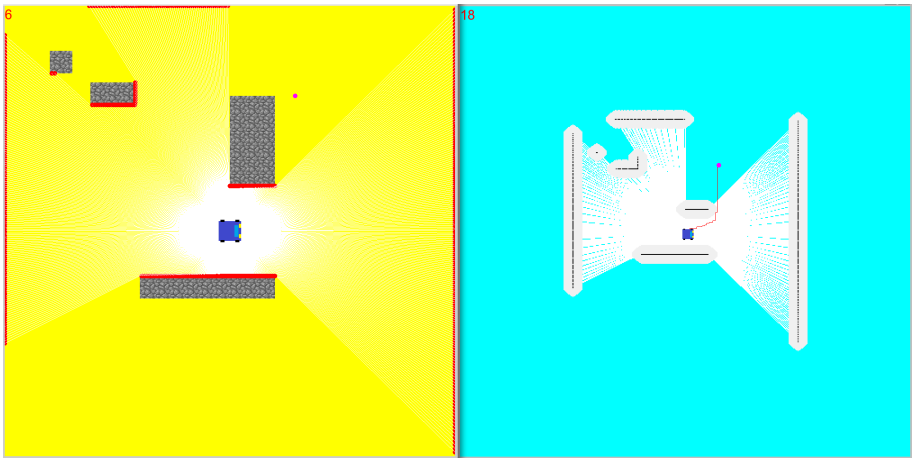


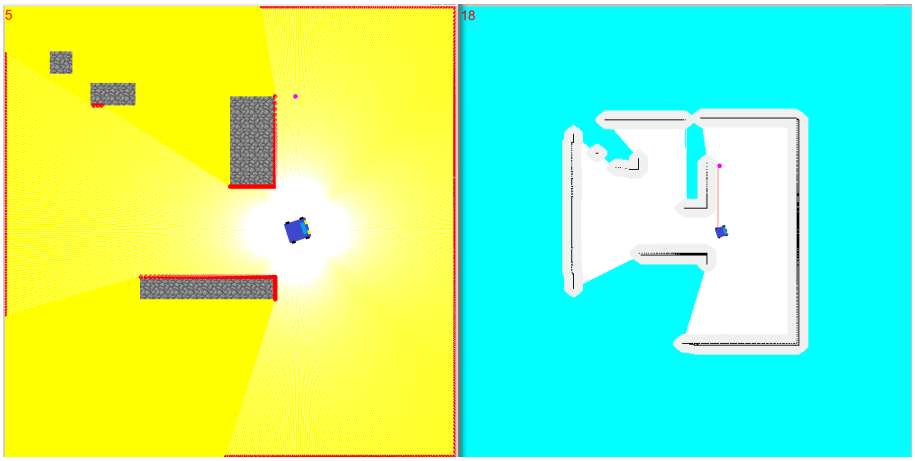
Abbildung 8.5: Darstellung eines Pfades um ein Hindernis herum

Bei der Berechnung des Pfades wird nicht auf den maximalen Lenkwinkel des Autos Rücksicht genommen. Eine autonome Steuerung ist noch nicht möglich, da der Algorithmus weder mit dem simulierten noch dem realen Auto kommunizieren kann.

Da der Pfad bei jedem Positions-Update neu berechnet wird, können so auch Ziele angefahren werden, welche sich in noch unbekanntem Gebiet der Karte befinden. In dem Fall wird die Karte erstellt, sobald der Bereich für den LiDAR sichtbar wird. Siehe Abbildung 8.6



(a) Pfad mit Ziel in unbekanntem Gebiet



(b) Pfad nachdem das Gebiet erkundet wurde

Abbildung 8.6: Berechnung eines Pfades mit Ziel in einem unbekannten Gebiet

Die Eingabe der Zielkoordinaten erfolgt über die simulierte Umgebung.

8.2 Evaluation

Im Folgenden werden wichtige Aspekte der Arbeit beleuchtet und evaluiert.

Ziele
In diesem Abschnitt wird evaluiert, ob die eingangs definierten Metriken und Ziele erreicht wurden.

1. **Simulation einer Umgebung**

   Angewandte Metrik:

   Kann eine Umgebung generiert werden? → Ja oder Nein

   Beschreibung des Ergebnisses:

   Es ist möglich eine Umgebung mit verschiedenen Hindernissen zu erstellen. Hindernisse können auch zur Laufzeit hinzugefügt werden.

   Metrik erfüllt:

   - **Simulation:** Ja
   - **Realität:** keine Angabe

   Übertrag auf die Realität

   Bei diesem Ziel macht ein Übertrag auf die Realität keinen Sinn, da die Umgebung nicht implementiert werden muss.

2. **Erkennung von Hindernissen**

   Angewandte Metrik:

   Werden alle Hindernisse erkannt? → Ja oder Nein

   Beschreibung des Ergebnisses:

   In der Simulation werden alle Hindernisse korrekt erkannt, auch wenn diese erst zur Laufzeit hinzugefügt werden.

   Metrik erfüllt:

   - **Simulation:** Ja
   - **Realität:** vermutlich Ja

   Übertrag auf die Realität:

   Der LiDAR generiert 2D-Daten. Da die Simulation ebenfalls nur 2D ist und auch nicht von äußeren Einflüssen betroffen ist, sind die Umwelteinflüsse und Begebenheiten in der Implementierung nicht von Bedeutung. In der Realität ist darauf zu achten, dass die Umgebung den in Kapitel 4 beschriebenen Voraussetzungen entspricht. Da die Implementierung des LiDAR-Sensors in der Simulation dem verwendeten Modell nachempfunden ist, sollten Hindernisse in einer korrekten Umgebung ebenfalls erkannt werden.

3. **Festlegen eines Zielpunktes**

   Angewandte Metrik:

   Kann ein Zielpunkt definiert werden? → Ja oder Nein

   Beschreibung des Ergebnisses:

   In der Simulation ist es möglich zur Laufzeit einen Zielpunkt zu setzen.

   Metrik erfüllt:

   - **Simulation:** Ja
   - **Realität:** vermutlich Ja

   Übertrag auf die Realität: Der Zielpunkt wird über Koordinaten abgebildet. Diese Koordinaten können in der Simulation auch außerhalb des Bereiches liegen, der vom LiDAR bereits erfasst wurde. Eine vorausgehende „Erkundungsfahrt“ist also nicht notwendig. Allerdings ist darauf zu achten, dass in der Simulation die Umgebung bekannt ist und darauf geachtet werden kann, das Ziel nicht innerhalb eines Hindernisses zu setzen. In der Realität sind die Koordinaten der Umgebung unbekannt. Das Ziel könnte also in einem Hindernis platziert sein. Deshalb muss noch eine Lösung implementiert werden, die erkennt, ob sich ein Ziel innerhalb eines Hindernisses befindet.

4. **Simulation eines Umgebungssensors**

   Angewandte Metrik:

   Kann ein Sensor simuliert werden? → Ja oder Nein

   Beschreibung des Ergebnisses:

   Die Verwendung von Raycasting ermöglicht die Simulation eines LiDAR-Sensors.

   Metrik erfüllt:

   - **Simulation:** Ja
   - **Realität:** Keine Angabe

   Übertrag auf die Realität:

   Bei diesem Ziel macht ein Übertrag auf die Realität keinen Sinn, da der LiDAR-Sensor nicht implementiert werden muss.

5. **Lokalisierung des Fahrzeugs**

   Angewandte Metrik:

   Für die Lokalisierung bezieht sich die Metrik auf Präzision und Geschwindigkeit. Beide Elemente sind abhängig von den gewählten Hard- und Softwarekomponenten.

   Beschreibung des Ergebnisses:

   Wie bereits beschrieben, ist die Lokalisierung des Fahrzeuges basierend auf den LiDAR-Daten noch nicht fehlerfrei. Aus diesem Grund wurde für dieses Ziel keine Metrik definiert. Die Definition einer Metrik macht Sinn, wenn sichergestellt ist, dass der Algorithmus zuverlässig arbeitet. Dann kann daran gearbeitet werden, theoretische Höchstwerte der Präzision und Geschwindigkeit der genutzten Technik zu erheben. Auf der Basis dieser Werte kann überlegt werden, welche Grenzwerte ausreichend sind, um die Zuverlässigkeit zu gewährleisten.

   Metrik erfüllt:

   - **Simulation:** Keine Angabe
   - **Realität:** Keine Angabe

   Übertrag auf die Realität:

   Da noch keine brauchbaren Ergebnisse in Bezug auf die Lokalisierung erzielt wurden, kann auch kein Übertrag von Erfahrungen aus der Simulation auf die Realität erfolgen.

   Es muss an einer Möglichkeit gearbeitet werden den additiven Fehler zu korrigieren. Danach können erste Überlegungen für passende Metriken folgen.

6. **Berechnung der Route**

   Angewandte Metriken:

   - Führt die Route vom Fahrzeug zum Zielpunkt → Ja oder Nein
   - Werden alle, in der Karte des Fahrzeuges bekannten, Hindernisse umfahren → Ja oder Nein
   - Kann die gesamte Route mit dem Fahrzeug abgefahren werden? → Ja oder Nein
   - Wie lange dauert die Berechnung der Route. Berechnungsdauer $<$ 100ms

Beschreibung des Ergebnisses:

Es wird eine Route berechnet, diese beinhaltet Start- und Zielpunkt der Route. Außerdem werden alle Hindernisse umfahren, die auf der Strecke liegen. Die Berechnung der Route dauert in der Simulation ca. 500ms

Es gibt aber ein Problem mit der Route. Nicht alle Richtungswechsel werden korrekt berechnet. Nahe am Fahrzeug sind die Richtungswechsel kurvenförmig und der Kurvenradius des Fahrzeuges wird berücksichtigt. Ab einem gewissen Abstand sind die Richtungswechsel eckig und daher nicht vollständig fahrbar.

Metrik erfüllt:

- **Simulation:** Ja - Ja - Nein - Nein
- **Realität:** Ja - Ja - Nein - keine Angabe

Übertrag auf die Realität:

Die Implementierung der Routenberechnung ist unabhängig davon, ob die Daten aus der Simulation oder aus der Realität kommen. Daher sollten die Erfahrungen und Ergebnisse der Simulation auf die Realität übertragen werden können.

Vermutung zur Problematik mit der Route:

Die Route des Fahrzeuges wird laufend neu berechnet. Daher bewegt sich auch der Teil der Route, der mit dem Auto genutzt werden kann, mit dem Auto mit. Daher sollte die Veränderung der Route hin zu einem eckigen Verlauf kein Problem darstellen. Allerdings kann das nicht garantiert werden. Daher sollte dieses Problem in der Implementierung des Algorithmus zur Routenberechnung analysiert und behoben werden, bevor damit das Modellauto gesteuert wird. Außerdem muss die Performance der Lösung noch optimiert werden, da bei laufender Neuberechnung der Route eine Dauer von 500ms nur für die Route zu lange ist.

7. **Steuerung des Fahrzeuges**

Angewandte Metrik:

Kann das Auto angesteuert werden? → Ja oder Nein

Beschreibung des Ergebnisses:

Für die Ansteuerung des Fahrzeuges gibt es eine Schnittstelle für die Simulation, die allerdings noch nicht verwendet wird. Für das reale Auto gibt es eine Python-Implementierung, die eine Steuerung mit einem Controller arbeitet. Eine Steuerung durch die Software ist noch nicht möglich.

Metrik erfüllt:

- **Simulation:** Nein
- **Realität:** Nein

Übertrag auf die Realität:

Um das Modellauto zu steuern, muss überlegt werden, ob die Performance von Python für diese Anwendung ausreichend ist, oder ob ein Übertrag der Elemente, die für eine autonome Fahrweise relevant sind, in C++ Sinn ergibt. In den Überlegungen dieser Arbeit hätten wir uns aufgrund der Performance für die C++-Implementierung entschieden.

Die maximalen Werte, die in der Python-Implementierung definiert sind und mit Skalierungsfaktoren verrechnet werden, sind empirisch ermittelt worden. Diese Werte sollten bei ersten Versuchen eventuell etwas nach unten korrigiert werden, um nicht die ersten Versuche die maximalen Fähigkeiten auszureizen.

Die Auswertung der gesetzten Ziele ergibt folgendes Gesamtbild:
In Bezug auf die Simulation wurden viele grundlegende Ziele erreicht, die es ermöglichen die Arbeiten an den noch fehlenden Stellen zu evaluieren und weiterzumachen. Die gewonnenen Erkenntnisse könne teilweise direkt auf die Realität übertragen werden, während für andere die Implementierung für das Modellauto umgesetzt werden muss, da das Verhalten der Simulation nicht für alle Bereiche auf die Realität übertragen werden können.

Kommunikation
Die Kommunikation innerhalb der Gruppe war gut. Regelmäßige Meetings und Absprachen sorgten für einen größtenteils reibungslosen Ablauf.
Die Kommunikation mit der anderen Gruppe hingegen war eher mangelhaft. Meetings wurden selten gehalten und das Kommunizieren wichtiger Informationen dauerte teilweise viel zu lange. Zudem wurden keine Protokolle geführt, was zu dem Verlust weitere Informationen führte. Deutlichere und häufigere Kommunikation wäre wichtig gewesen und hätte den Fortschritt des Projektes vorangebracht.

Aufgabenteilung
Die Aufgabenteilung war gut. Es wurde umfangreich geplant, wodurch jeder zu jedem Zeitpunkt konkrete Aufgaben hatte. Somit konnte die Zeit gut genutzt werden.

8.3 Ausblick

Das Ergebnis der Arbeit bietet eine gute Grundlage für die autonome Hinderniserkennung und Umgehung. Allerdings gibt es diverse Verbesserungsvorschläge und Erweiterungen, welche das Ergebnis verbessern.

Steuerung des Fahrzeugs
Der wohl wichtigste Punkt ist die Nutzung des Algorithmus zur Steuerung des Fahrzeugs. Zum aktuellen Zeitpunkt ist der Algorithmus lediglich in der Lage, einen Pfad zu berechnen.
Eine notwendige Ergänzung ist somit die Verbindung des simulierten und des realen Fahrzeugs mit dem Algorithmus. Die Logik des Algorithmus ist vollständig von der Quelle der Daten abstrahiert.
Die Schnittstelle für die Kommunikation mit dem simulierten Auto existiert bereits, der Algorithmus ist jedoch noch nicht in der Lage diese zur Steuerung des simulierten Fahrzeugs zu verwenden.
Die Schnittstelle für die Kommunikation mit dem realen Auto ist noch gar nicht implementiert. Der Aufbau der Schnittstelle sollte von der Schnittstelle des simulierten Autos kopiert werden. Anstelle von Methoden-Aufrufen zur Steuerung des Autos, muss die Schnittstelle für das reale Auto Pins des Raspberry Pi setzten. Die entsprechende Python-Implementierung der Hardware-Gruppe ist vorhanden und muss nur in C++ übersetzt werden.

Handy-Stuerung
Die Übermittlung der Zielkoordinaten erfolgt bei der Simulation über einen Mausklick innerhalb der simulierten Umgebung.
Eine Möglichkeit die Zielkoordinaten an das reale Fahrzeug zu übermitteln ist das Erstellen einer Handy-App. Da der Raspberry Pi über WLAN und Bluetooth verfügt, ist es möglich, eine App zu erstellen, welche die generierte Karte anzeigt. Zurzeit wird eine solche Karte, bei Nutzung der Simulation, in einem zweiten Fenster angezeigt. Die Eingabe einer Zielkoordinaten könnte dann mittels Klick auf die Karte erfolgen.

Zusätzliche Sensorik
Es wurden bereits zusätzliche Ultraschall-Sensoren an dem Fahrzeug verbaut. Eine Integration dieser in die bestehende Logik ist eine weitere, sinnvolle Ergänzung.
Da diese Sensoren unterhalb des LiDAR, vorne an dem Fahrzeug angebracht sind, können sie genutzt werden um flache Hindernisse, welche sich unterhalb des LiDAR-Sensors befinden, zu erkenne. Außerdem können sie als fail-safe dienen, falls das Fahrzeug auf ein Hindernis zusteuert, diesem aber nicht

ausweicht. Ein solcher fail-safe dient vor allem dem Schutz der Hardware, welche fragil und teuer ist.
Neben Ultraschall-Sensoren macht auch die Erweiterungen um einen Beschleunigungs-Sensor Sinn. Ein solcher Sensor kann genutzt werden, um zusätzliche Bewegungsdaten zu sammeln. Diese können verwendet werden, um z.B. durch Gewichtung, die Ergebnisse der Lokalisierung zu verbessern.

Filter
Eine weitere wichtige Ergänzung zur Verbesserung der Lokalisierung sind Filter.
Filter sind eine reine Software-Lösung und können somit ohne zusätzliche Materialkosten integriert werden. Ein solcher Filter sorgt dafür, dass schlechte Ergebnisse gefiltert werden. Hierdurch kann der durchschnittliche Fehler, welcher die Ergebnisse des ICP-Algorithmus haben, gesenkt werden.

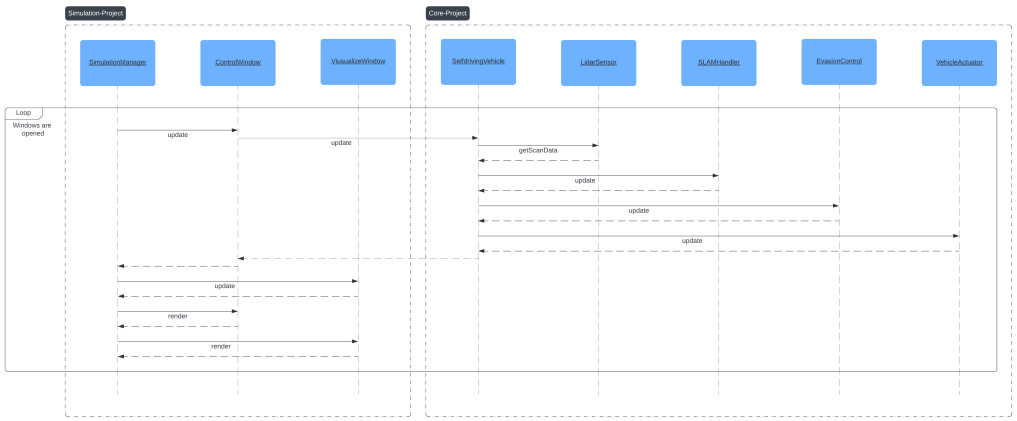


Abbildung A.1: Ablaufdiagram Simulation

A Anhang

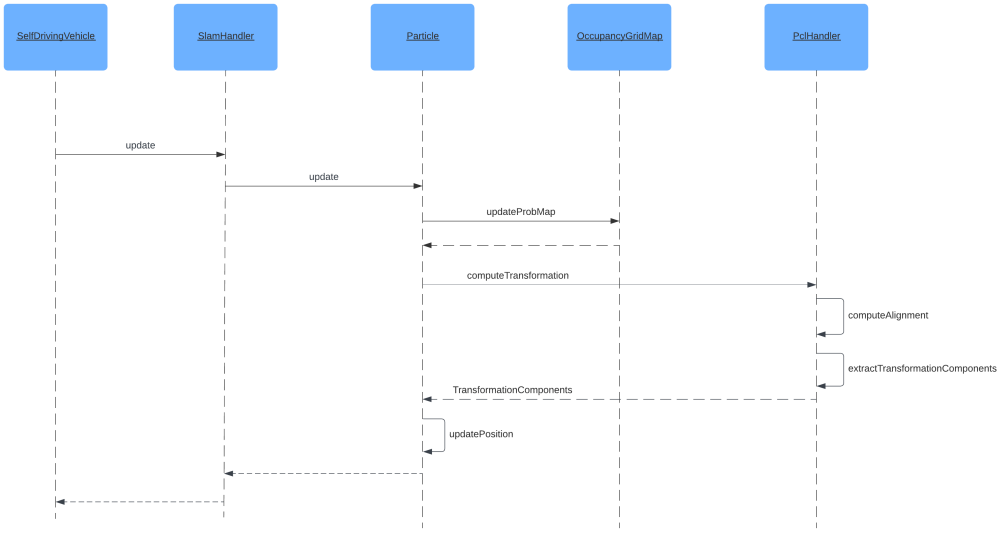


Abbildung A.2: Ablaufdiagram eines Slam Update-Zyklus

Literatur

[1] Neil Ashby. „Relativity in the global positioning system“. In: *Living Reviews in relativity* 6 (2003), S. 1–42.

[2] Weckar E. *How to adapt pathfinding algorithms to restricted movement?* Game Development Stack Exchange. 9. Nov. 2017. URL: https://gamedev.stackexchange.com/q/150605.

[3] The Editors of Encyclopaedia Britannica. *computer simulation*. 2023. URL: https://www.britannica.com/technology/computer-simulation (besucht am 23. 01. 2024).

[4] *Estimating Normals for a Point Cloud*. 2024. URL: https://www.fwilliams.info/point-cloud-utils/sections/point_cloud_normal_estimation.

[5] Elisabete Fernandes u. a. „Towards an Orientation Enhanced Astar Algorithm for Robotic Navigation“. In: *2015 IEEE International Conference on Industrial Technology (ICIT)*. Seville: IEEE, März 2015, S. 3320–3325. ISBN: 978-1-4799-7800-7. DOI: 10.1109/ICIT.2015.7125590.

[6] *GitHub: Slamtec/rplidar_ros*. 2023. URL: https://github.com/slamtec/rplidar_ros (besucht am 05. 02. 2024).

[7] *GitHub: Slamtec/rplidar_sdk*. 2023. URL: https://github.com/Slamtec/rplidar_sdk (besucht am 05.02.2024).

[8] Philipp Glira u.a. „A Correspondence Framework for ALS Strip Adjustments based on Variants of the ICP Algorithm“. In: *Photogrammetrie - Fernerkundung - Geoinformation* 2015 (Aug. 2015). DOI: 10.1127/pfg/2015/0270.

[9] Peter E. Hart, Nils J. Nilsson und Bertram Raphael. „A Formal Basis for the Heuristic Determination of Minimum Cost Paths“. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (Juli 1968), S. 100–107. ISSN: 2168-2887. DOI: 10.1109/TSSC.1968.300136.

[10] Leihui Li, Riwei Wang und Xuping Zhang. „A Tutorial Review on Point Cloud Registrations: Principle, Classification, Comparison, and Technology Challenges“. In: *Mathematical Problems in Engineering* 2021.1 (2021). DOI: https://doi.org/10.1155/2021/9953910. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1155/2021/9953910. URL: https://onlinelibrary.wiley.com/doi/abs/10.1155/2021/9953910.

[11] Yanjie Liu u.a. „Improved LiDAR Localization Method for Mobile Robots Based on Multi-Sensing“. In: *Remote Sensing* 14.23 (2022). ISSN: 2072-4292. DOI: 10.3390/rs14236133. URL: https://www.mdpi.com/2072-4292/14/23/6133.

[12] Anu Maria. *Introduction to modeling and simulation*. 1997. URL: https://dl.acm.org/doi/pdf/10.1145/268437.268440 (besucht am 23.01.2024).

[13] Michael Montemerlo u.a. „FastSLAM: A factored solution to the simultaneous localization and mapping problem“. In: *Aaai/iaai* 593598 (2002), S. 593–598.

[14] *PCL Walkthrough*. 2023. URL: https://pcl.readthedocs.io/projects/tutorials/en/latest/walkthrough.html#keypoints.

[15] *pcl::IterativeClosestPoint< PointSource, PointTarget, Scalar > Class Template Reference*. 2024. URL: https://pointclouds.org/documentation/classpcl_1_1_iterative_closest_point.html.

[16] *Raspberry Pi 4 Model B*. Techn. Ber. Raspberry Pi Ltd, 2023. URL: https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-product-brief.pdf (besucht am 31.01.2024).

[17] *ROS: Home*. 2023. URL: https://www.ros.org/ (besucht am 05.02.2024).

[18] *RPLIDAR A1 - Development Kit User Manual.* 2023. URL: https://www.slamtec.ai/wp-content/uploads/2023/11/LM108_SLAMTEC_rplidarkit_usermaunal_A1M8_v2.2_en.pdf (besucht am 08. 07. 2024).

[19] *RPLIDAR A1 - Introduction and Datasheet.* 2020. URL: https://www.slamtec.ai/wp-content/uploads/2023/11/LD108_SLAMTEC_rplidar_datasheet_A1M8_v3.0_en.pdf (besucht am 31. 01. 2024).

[20] Niko Schwarz, Mircea Lungu und Oscar Nierstrasz. „Seuss: Decoupling Responsibilities from Static Methods for Fine-Grained Configurability". In: *Journal of Object Technology* 11.1 (2012), 3:1–3:23. DOI: 10.5381/jot.2012.11.1.a3. URL: https://www.jot.fm/issues/issue_2012_04/article3.pdf.

[21] SFML. *SFML.* https://www.sfml-dev.org/index.php.

[22] Game Dev Stackexchange. *Are There Any Disadvantages of Using Distance Squared Checks Rather than Distance?* Forum Post. Feb. 2012.

[23] *The PCL Registration API.* 2023. URL: https://pcl.readthedocs.io/projects/tutorials/en/latest/registration_api.html.

[24] Huajie Wu u. a. „Moving event detection from LiDAR point streams". In: *Nature Communications* 15.1 (2024), S. 345. ISSN: 2041-1723. DOI: 10.1038/s41467-023-44554-8.