HAW HAMBURG

BACHELORTHESIS
Luk Jonas Schwalb

ROS-Architektur mit Autonomiefunktionen für ein Miniaturfahrzeug

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Luk Jonas Schwalb

ROS-Architektur mit Autonomiefunktionen für ein Miniaturfahrzeug

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Technische Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Tim Tiedemann
Zweitgutachter: Prof. Dr. Stephan Pareigis

Eingereicht am: 4. September 2021

Luk Jonas Schwalb

Thema der Arbeit

ROS-Architektur mit Autonomiefunktionen für ein Miniaturfahrzeug

Stichworte

Autonomes Fahren, Carla, Miniaturfahrzeug, ROS, H0

Kurzzusammenfassung

Der Miniaturmaßstab 1:87 bietet die Möglichkeit Forschenden den Einstieg in den Bereich des autonomen Fahrens zu erleichtern. Es wird eine auf ROS basierende Architektur für ein Miniaturfahrzeug entwickelt, die das Fundament für weitere Arbeiten legen soll. Die Architektur beinhaltet Wege zur Lokalisierung, Kontrolle der Hardware und Regler für das Fahrverhalten. Die Sensorik des Autos wird genutzt um die Straße zu erkennen und Navigationsziele, auch mit ungenauen oder spärlich vorhandenen Karten, zu erreichen. Die Evaluation des Systems erfolgte auf einer echten Teststrecke und im Simulator.

Luk Jonas Schwalb

Title of Thesis

ROS-Architectur with autonomous functions for a miniature vehicle

Keywords

Autonomous driving, Carla, Miniature vehicle, ROS, H0

Abstract

The miniature scale 1:87 offers opportunities for researchers to get started in the field of autonomous driving. This work developed a ROS architecture for a miniature vehicle, which is intended to lay a foundation for future works. The architecture offers ways for localization, control of the hardware and control over the driving behaviour. The car's sensors are used to detect the street and thus reach navigation goals on sparse or incomplete maps. The system was evaluated on a real test track and in a simulator.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungen

BiSeNet V2 Bilateral Network with Guided Aggregation for Real-time Semantic Segmentation.

LSE Least square error.

mIoU mean intersection over union.

MRPT Mobile Robot Programming Toolkit.

ROS Robot Operating System.

TEB Timed-Elastic-Band.

UWB Ultra-wideband.

1 Einleitung

1.1 Motivation

In einem autonomen Auto interagieren eine Vielzahl an verschiedenen Sensoren und Aktoren, um die komplexe reale Welt digital wahrzunehmen. Softwaresysteme verarbeiten die generierten Daten und treffen Entscheidungen, um ein sicheres und korrektes Fahren zu gewährleisten. Fehler in der Software können jedoch fatale Folgen haben und sogar das Leben der Fahrer oder anderer Verkehrsteilnehmer gefährden, weshalb intensives Testen zwingend notwendig ist. Doch rechtliche Vorlagen, hohe Kosten sowie hohe Risiken erschweren das Testen. Darum wurde in [13] ein Miniaturauto im Maßstab 1:87 entwickelt. Ein Miniaturauto umgeht die genannten Probleme, denn durch die geringe Größe verursacht es bei Kollisionen nur marginale Schäden, ist nicht in der Lage Menschen zu verletzen und kostet ein Bruchteil eines vollwertigen Autos. Somit ist ein risikofreies Testen und Forschen außerhalb von Simulatoren gewährleistet.

Ebenso wichtig wie eine solide Hardwareplattform ist ihre zugrunde liegende Softwarearchitektur. Wenn verschiedene Aspekte des autonomen Fahrens erforscht werden sollen, ist es hinderlich, wenn viel Vorarbeit außerhalb des Zielbereiches geleistet werden muss, um diese zugänglich zu machen.

Es gilt also Arbeitsschritte zu minimieren, die bereits ein anderer Forscher durchgeführt hat. Deshalb ist eine modulare Softwarearchitektur von Vorteil, die es ermöglicht nur einzelne Komponenten des Gesamtsystems auszutauschen. Schnittstellen müssen bereitgestellt gestellt und der Einstieg dadurch weitestgehend vereinfacht werden.

1.2 Ziel der Arbeit

Die Entwicklung einer Hardwareplattform durch [13] ist ein großer Schritt um kleinen Forschungsgruppen den Einstieg in den Bereich des autonomen Fahrens zu ermöglichen.

Im Verlauf dieser Arbeit soll zusätzlich ein softwareseitiges Fundament für zukünftige Arbeiten gelegt werden. Denn der komplexe Bereich des autonomen Fahrens lässt sich nicht durch einzelne Personen lösen, sondern bedarf ein breites Spektrum verschiedener Ansätze aus verschiedenen Bereichen. Es sollen Schnittstellen für die verbaute Sensorik und Aktorik entwickelt werden sowie Softwarebausteine die eine leichte Nutzung ermöglichen. Folgende Merkmale stehen im Fokus:

1. Leichte Benutzung
2. Viele Möglichkeiten der Wiederverwendbarkeit
3. Erweiterbar
4. Offen
5. Schlankes Design

Zusätzlich soll ein Fahrsystem implementiert werden, das nach Angabe eines Zielpunktes auf einer Karte einen Pfad berechnet und diesen abfährt. Dabei soll das Auto die Straße dynamisch erkennen und nicht verlassen. Das ermöglicht die Nutzung von ungenauen Karten mit unbekannter Straßenbreite.

1.3 Aufbau der Arbeit

Zu Beginn werden in Kapitel 2 einige Grundlagen für diese Arbeit erläutert. Darauf folgend werden die gewählte Architektur und ihre Merkmale beschrieben.

Kapitel 4 beschäftigt sich mit der Anbindung verschiedener, im Auto verbauter Hardware Komponenten, auf denen die weiteren Kapitel aufbauen.

In Kapitel 5 werden verschiedene Mechanismen für die Lokalisierung vorgestellt und ihre Stärken und Schwächen miteinander verglichen.

Das Kapitel 6 befasst sich mit der Entwicklung der Teilautonomiefunktionen, die das Auto eine Zielposition erreichen lassen sollen. Dafür wird die Wahrnehmung der Straße und das Planen einer Route unterschieden.

Zuletzt werden in den Kapiteln 7 und 8 die entwickelten Komponenten evaluiert und und in einem Fazit zusammengefasst. Anschließend erfolgt ein Ausblick für zukünftige Verbesserungen.

2 Grundlagen

2.1 Autonomes Fahren

Ein autonomes Auto sollte in der Lage sein ohne menschlichen Einfluss selber navigieren und fahren zu können. Dafür ist es notwenig die reale Umgebung wahrzunehmen, als digitale Umgebung zu rekonstruieren und daraufhin Entscheidungen zu treffen. Autonome Autos nutzen dafür ein breites Spektrum an verschiedenen Sensoren, wie zum Beispiel Kamera, LiDAR, IMU, GPS und Rad-Encoder.

Mehrere Kernmodule müssen eng zusammenarbeiten um die komplexe Aufgabe in einzelne Bereiche zu unterteilen und zu realisieren. Dazu gehören Pfadplanung, Lokalisierung, Wahrnehmung und Kartierung der Umgebung sowie die Umsetzung der Steuerbefehle. Für die Nutzung der Module in dieser zeitkritischen Umgebung muss das Auto mit entsprechender Rechenleistung ausgestattet sein.

2.2 TinyCar

Das TinyCar ist ein im Maßstab 1:87 gebautes Miniaturauto, das für den autonomen Einsatz vorgesehen wurde. Mit Maßen von 11,5x3 cm ist es das Kleinste, uns bekannte Auto, dass sich für die Forschung einsetzen lässt.

Als zentrale Recheneinheit dient ein Raspberry Pi 4 Compute Module, auf Basis eines Broadcom BCM2711, Quad core Cortex-A72 @ 1.5Ghz [19]. Über Steckverbinder wird das Compute Module an die Trägerplatine angebunden, die alle verwendeten Sensoren und Aktoren beinhaltet.

Eine 18650-Zelle wird für die Stromversorgung genutzt und reicht unter Volllast für durchschnittlich zwei Stunden Akkulaufzeit. Die Steuerung basiert auf einem Servomotor, der

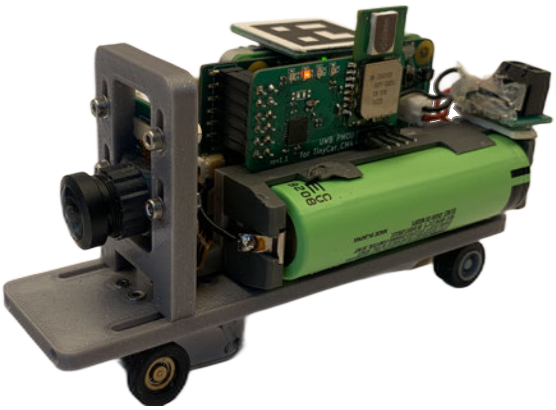


Abbildung 2.1: Das in dieser Arbeit verwendete TinyCar.

über einen Draht eine Ackermann Lenkung bewegt. Ein festes Verhältnis vom Einstellwinkel des Servos zu dem Lenkwinkel des Autos ist nicht gegeben und muss nachträglich kalibriert werden.

Das verbaute Getriebe lässt sich über einen Motor Treiber steuern, der über 2 GPIOs angesprochen wird. Ein Hall-Sensor unterhalb der Achse erkennt die Stärke des Magnetfelds zweier um die Achse rotierender Magneten, mit umgekehrter Polung. Somit lässt sich eine Odometrie ermitteln.

Um rechenintensive neuronale Netze nutzen zu können, sind zwei Google Coral TPUs verbaut [10], von denen eine aufgrund Treiber Problemen seitens Google zum Zeitpunkt dieser Arbeit noch nicht einsetzbar war. Da GNSS für den Einsatz in geschlossenen Räumen nicht geeignet ist, wurde auf das Ultra-wideband Modul DWM1000 [6] des Herstellers Decawave zurückgegriffen. Laut Angaben des Herstellers lässt sich eine Positionsbestimmung mit 10 cm Genauigkeit erreichen.

Ein LiDAR war zum Zeitpunkt dieser Arbeit nicht ausreichend klein erwerblich. Doch Solid-State LiDARs wie das Vision Mini des Herstellers Blickfeld (5x5x5 cm), kommen den nötigen Maßen nahe. Eine Verwendung von LiDAR Technologie im Miniaturmaßstab könnten bei weiteren Fortschritten möglich sein.

Abbildung 2.2: Bild des Mikrowunderland von oben.

2.3 Mikrowunderland

Das Mikrowunderland ist eine komplette Welt im Miniaturmaßstab 1:87. Inspiriert wurde das Projekt durch das Miniatur Wunderland [4], welches ebenfalls im Maßstab 1:87 gebaut ist und eine Gesamtfläche von 1500 m^2 besitzt. Das Miniatur Wunderland eignet sich für das Sammeln von Daten, da Autos durch einen in der Straße verlegten Draht komplett mechanisch fahren können.

Das Mikrowunderland misst ungefähr 2x1,5 m und bietet viele Funktionalitäten, die ein umfangreiches Testen ermöglichen. Dazu gehören verschiedene Farben des Straßenbelags, Straßenlaternen, Ampeln und abgenutzte Straßenmarkierungen. Vorgesehen ist es, das Miniatur Wunderland für das Sammeln von Daten zu verwenden und die damit trainierten Verfahren im Mikrowunderland zu testen.

Durch die hohe Verfügbarkeit an Modellen im Maßstab 1:87 kann die Umgebung schnell umgebaut und angepasst werden. Das Straßennetz besitzt Links- und Rechtskurven, einen Inner- und Außerortsbereich und Fußgängerüberwege.

2.4 CARLA Simulator

Carla [8] ist ein Open-Source Simulator, der für das Testen von autonomen Straßenfahrzeugen in einer simulierten Welt gedacht ist. Carla wird als Plugin für die Spiel-Engine Unreal Engine entwickelt und nutzt auch dessen Physikengine um Sensoren wie Kamera, IMU oder GNSS zu emulieren.

Der CARLA Simulator wird als Server gestartet und kommuniziert mit einem oder mehreren Clients. Die Clients erhalten von dem Server Sensordaten und können Fahrzeuge steuern, Objekte kreieren oder das Wetter verändern.

Derzeit existieren acht Karten, die alle der amerikanischen Straßenverkehrsordnung entsprechen. Die Karten sind mit Gebäuden, Pflanzen, parkenden Autos und weiteren dekorativen Objekten ausgestattet.

2.5 Robot Operating System

Widersprüchlich zum Namen handelt es sich bei dem Robot Operating System (ROS) nicht um ein vollwertiges Betriebssystem, sondern um einen Zusammenschluss aus Programmen und Tools für die Verwendung auf Robotern. Doch die Nutzungsmöglichkeiten von ROS begrenzen sich nicht nur auf den Einsatz von Robotern, da die Hardwareebene flexibel ausgetauscht werden kann. ROS stellt ähnlich wie ein Betriebssystem Wege für den Austausch von Nachrichten innerhalb seiner Pakete bereit, bietet eine Abstraktion der Hardware und kümmert sich um die Kontrolle der Prozesse.

Kerngedanke von ROS ist es, eine Open-Source Umgebung zu schaffen, die die Entwicklung komplexer Roboter vereinfacht, indem einzelne Komponenten leicht wiederverwendbar sind. Somit können Gruppen an Wissenschaftlern ihre Lösungen für Teilprobleme veröffentlichen und anderen Forschungsgruppen zugänglich machen. Durch standardisierte Schnittstellen und Nachrichten wird die En3twicklung erleichtert und ermöglicht hohe Kompatibilität.

Die Ziele von ROS definieren sich laut der Veröffentlichung [18] wie folgt:

Peer-to-peer: ROS Prozesse kommunizieren direkt oder über ein Publisher/Subscriber System miteinander.

Multi-lingual: ROS unterstützt Python, C++, Lisp und Octave.

Tools-based: ROS hat ein Mikrokernel Design, in dem einzelne Tools die Komponenten kompilieren und ausführen.

Thin: ROS erstellt kleine ausführbare Dateien. Die Komplexität steckt in wiederverwendbaren Bibliotheken.

Free and Open-Source: ROS ist frei verfügbar und unter der BSD Lizenz veröffentlicht.

Da ROS alle erwünschten Anforderungen erfüllt, wird es für die Entwicklung der Software verwendet. Eine Brücke, welche die Nutzung von Sensordaten und die Steuerung im Carla Simulator ermöglicht, existiert ebenfalls.

2.6 Bestehende Projekte

Das Nutzen von ROS für den Einsatz in autonomen Autos ist keine neue Idee. Das liegt vor allem an der Vielseitigkeit und den vielen bereits vorhandenen Algorithmen, die zur direkten Nutzung bereitstehen.

Autoware [14]: Am weitesten verbreitet ist die Autoware Umgebung, die fertige Lösungen für Lokalisierung, Objekterkennung, Pfadplanung und Steuerung bietet. Autoware wird ebenfalls Open Source entwickelt und unterteilt sich in die ROS1 Version *Autoware.ai* sowie in die ROS2 Version *Autoware.auto*.

In Autoware basieren viele der grundlegenden Funktionen, wie zum Beispiel Lokalisierung, Objekterkennung und Straßenerkennung auf der Auswertung von LiDAR Informationen, die das TinyCar nicht besitzt. Außerdem arbeitet Autoware mit hochaufgelösten Karten, die in dieser Arbeit nicht verwendet werden.

A Self-Driving Car Architecture in ROS2 [20]: stellt eine Architektur vor, die auf ROS2 basiert und einen starken Fokus auf Echtzeitperformance legt. Eine konkrete Umsetzung dieser Architektur existiert nur in Teilen und wurde nicht veröffentlicht. Ziel der Architektur ist es, ein schlankes, erweiterbares und vollautonomes System zu schaffen.

Ebenfalls wird hier auf die Nutzung von hochaufgelösten Karten und LiDAR Informationen für die Lokalisierung gesetzt. Die Auswertung der Architektur erfolgt auf einem umgebauten KIA Niros.

Da keine öffentliche Implementierung der Architektur existiert, ist eine komplette Eigenentwicklung erforderlich.

3 Architektur

3.1 Anforderungen

Viele der Anforderungen, die für ein reguläres autonomes Auto gelten, können im Maßstab 1:87 vorübergehend vernachlässigt werden. Da mehrere Tonnen bewegte Masse schwere Sachschäden und den Tod von Menschen verursachen können, wird in echten Autos auf Echtzeitsysteme gesetzt. Fahrzeuge im Miniaturmaßstab können keine Schäden verursachen, Echtzeitsysteme und extrem hohe Sicherheitsvorkehrungen sind deshalb nicht zwingend notwendig.

Daraus ergeben sich abweichende Anforderungen gegenüber den meisten anderen Systemen. Die Steuerungsschleife kann mit einstelligen Hertz Werten arbeiten, statt ungefähr 100Hz wie es bei vollwertigen autonomen Autos gängig ist. Falls es die Gegebenheiten erforderlich machen, könnte sogar die Geschwindigkeit aller Fahrzeuge innerhalb der Testumgebung reduziert werden, um mehr Zeit für Berechnungen zu gewinnen.

Doch die begrenzte Rechenleistung des TinyCars macht eine stark optimierte Software dennoch erforderlich. In regulären Autos werden leistungsstarke Computer verbaut, um den hohen Ansprüchen der Software gerecht zu werden. Außerdem kann nicht auf die Verwendung von LiDAR Informationen zurückgegriffen werden, da in dem Maßstab des TinyCars keine LiDAR Technologien existieren.

Das entwickelte System soll durch stetige Erweiterung ein komplett autonomes Fahren ermöglichen. Die einzelnen Komponenten dafür müssen jedoch nicht sofort verfügbar sein.

Jene Nodes, die mit der Hardware kommunizieren, sollen durch andere Nodes ausgetauscht werden können, die ihr Verhalten im Carla Simulator simulieren. Damit lässt sich die Software auch ohne Zugriff zur echten Hardware testen.

3.2 Architektur Design

Eine grafische Übersicht der Architektur ist in Abbildung 3.1 zu sehen.

Aufgrund des hohen Entwicklungsaufwandes wurde in dieser Arbeit davon abgesehen eine eigene Pfadplanung und die zugehörigen Komponenten zur Generierung von Steuerkommandos zu erstellen. Stattdessen wurde auf das ROS Package *move_ base* zurückgegriffen, das die gängige Lösung für die Navigation von Fahrzeugen, basierend auf ROS, ist. Das Package startet eine lokale und eine globale Costmap und bietet Schnittstellen für die Einbindung einer lokalen Pfadplanung sowie einer globalen Pfadplanung.

Dadurch kann leicht auf existierende Algorithmen für die Pfadplanung zurückgegriffen werden, denn die einzelnen Plugins für die lokale und globale Pfadplanung können leicht ausgetauscht werden.

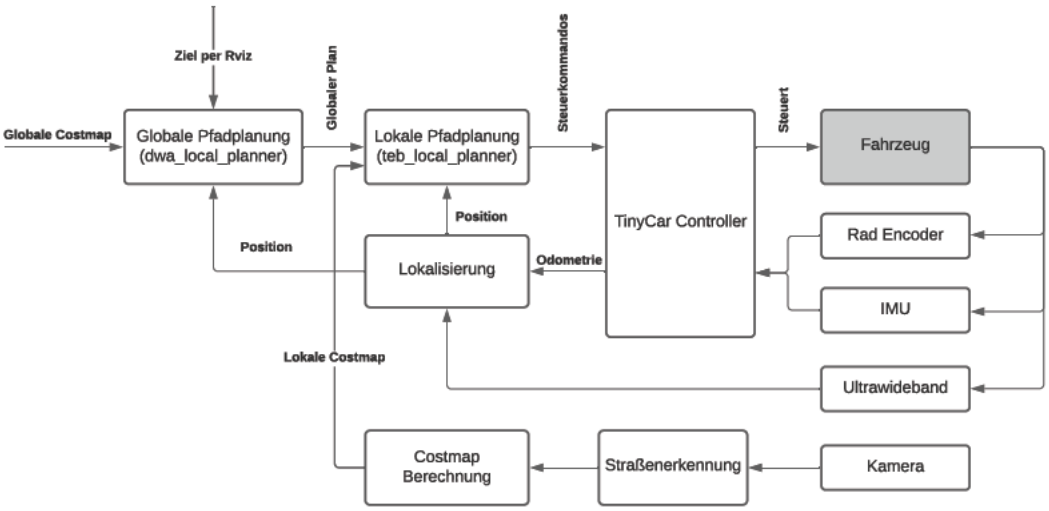


Abbildung 3.1: Blockdiagramm der entwickelten Systemarchitektur.

In Abbildung 3.1 arbeiten die Nodes von links nach rechts wie folgt:

Globale Costmap: Die globale Costmap wird mittels einer statischen Karte generiert.

Globale Pfadplanung: Ein globaler Pfad wird durch das *move_ base* Plugin *dwa_ - global_ planner* erstellt.

Lokale Pfadplanung: Eine leicht modifizierte Version des *teb_ local_ planner* generiert den lokalen Pfad und sendet Steuerkommandos.

TinyCar Controller: Der TinyCar Controller empfängt Steuerkommandos und berechnet daraus Signale für die Lenkung und die Geschwindigkeitsregelung. Außerdem wertet der TinyCar Controller die Daten des Rad-Encoders aus und generiert in Kombination mit den aktuellen Steuerkommandos eine Odometrie.

Lokalisierung: Die Lokalisierung wird durch *robot_ localization* realisiert. Ein Extended-Kalman-Filter fusioniert die Werte der Odometrie mit denen der Ultra-wideband Lokalisierung und der IMU.

Straßenerkennung: Ein Bilateral Network with Guided Aggregation for Real-time Semantic Segmentation (BiSeNet V2) [21] erkennt die Straße anhand des Kamerabildes.

Costmap Berechnung: Die erkannte Straße wird in die Vogelperspektive umgerechnet und in der lokalen Costmap eingetragen.

4 TinyCar Hardware und Controller

4.1 Hardware

Zur Verwendung der Hardware des TinyCars sind einige Nodes nötig. Im Falle von Sensoren müssen die gemessenen Werte in entsprechende Nachrichten verpackt und versendet werden, damit andere Nodes diese Werte verwenden können.

Aktoren hingegen empfangen Einstellwerte und richten sich nach diesen aus.

4.1.1 Kamera

Zur Verwendung der Kamera kann der fertige *raspicam_node* genutzt werden.

Er sendet das Kamerabild als *CameraInfo* Nachricht, in der auch die Verzerrungsmatrix enthalten ist. Ein weiterer Node *image_proc* empfängt das Bild und führt die Entzerrung durch.

4.1.2 IMU

Ein fertiger Node für die Nutzung der verbauten Bosch IMU BNO055 [2] existiert bereits und bedarf keiner Eigenentwicklung.

Ausgelesen werden die Werte der IMU über die UART Schnittstelle /dev/ttyAMA1 des Compute Module. Der Node sendet periodisch eine *Imu* Nachricht, die Geschwindigkeit, Beschleunigung und Ausrichtung enthält.

Außerdem können die Daten des beinhalteten geomagnetischen Sensors und die aktuelle Temperatur ausgegeben werden. Davon wird in dieser Arbeit jedoch kein Gebrauch gemacht.

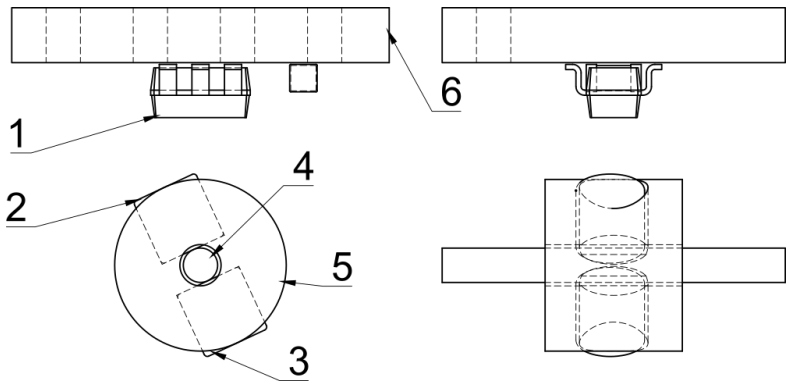


Abbildung 4.1: Aufbau des Rad-Encoder durch Hall-Sensor. 1: TLV493D Hall-Sensor 2: Magnet (Nordpol) 3: Magnet (Südpol) 4: Achse 5: Aufnahme für Magnete 6: Trägerplatine für TLV493D

Quelle: Markus Kasten

4.1.3 Hall-Sensor

In dem TinyCar ist zur Auswertung der an der Achse montierten Magneten der Hall-Sensor TLV493D [11] von Infineon verbaut. Zum Auslesen des Hall-Sensors wurde ein eigener Node entwickelt, da bislang keine fertigen Lösungen existieren.

Die Kommunikation mit dem Chip findet mittels des I2C Bus /dev/i2c-6 statt. Der Node liest ebenfalls periodisch die aktuellen Messwerte des TLV493D aus und publiziert daraufhin einen *Vector3Stamped*. Der Vektor beschreibt die Flussdichte des Magnetfeldes in den jeweiligen Achsen X, Y und Z.

4.1.4 Rad-Encoder

Der Rad-Encoder wird durch zwei Magneten realisiert, die mit umgekehrten Magnetfeld um die Hinterachse des TinyCars rotieren. Oberhalb der Achse ist der TLV493D Sensor positioniert und misst die sich ändernde Magnetische Flussdichte. Ein schematischer Aufbau des Rad-Encoders wird in 4.1 dargestellt.

Bevor die Daten des Hall-Sensors verwendet werden können, müssen sie in ein Format gebracht werden, das es erlaubt die Geschwindigkeit des Schaftes zur ermitteln. Für die weitere Interpretation der Daten wird der in [1] präsentierte Ansatz angewandt.

Der Magnet rotiert sich um die Y-Achse des vom Sensor gemessenen Magnetfeldes. Änderungen sind dementsprechend nur in der X- und Z-Achse messbar. Die Erkennung des

aktuellen Winkels erfolgt durch die Berechnung von Alpha in einem imaginären Einheitskreis.

Werte, die von dem Hall-Sensor ausgelesen werden, dienen als X- und Y-Positionen eines Punktes auf einem Einheitskreis. Die seitliche Verschiebung des Einheitskreises wird davor durch das Messen der minimalen und maximalen X- und Y-Werte abgezogen. Normalisiert werden die Werte durch das Teilen der Differenz zwischen minimaler und maximaler Werte.

Die Formeln dafür lauten wie folgt

$$X_1 = X - \frac{X_{max} + X_{min}}{2}$$

$$Y_1 = Y - \frac{Y_{max} + Y_{min}}{2}$$

$$X_2 = \frac{X_1}{\frac{X_{max} - X_{min}}{2}}$$

$$Y_2 = \frac{Y_1}{\frac{Y_{max} - Y_{min}}{2}}$$

$$\alpha = \arctan\left(\frac{Y_2}{X_2}\right)$$

Das Ergebnis wird in Form einer *WheelState* Nachricht publiziert. Die *WheelState* Nachricht enthält den aktuellen Wert für Alpha, die Änderung von Alpha seit dem letzten Update und die vergangene Zeit.

4.1.5 Getriebemotor und Lenkservo

Um die Stellwerte zu erhalten empfängt der Node Nachrichten vom Typ *Float32*. Die Nachrichten haben einen Wertebereich von -1 bis 1 und werden jeweils einzeln für Getriebemotor und Lenkservo versandt.

Der Getriebemotor wird durch einen Motortreiber mit interner H-Brücke gesteuert. Je nach Fahrtrichtung muss an Pin18 beziehungsweise Pin19 des Compute Modules ein PWM-Signal mit einem Tastgrad von 0 bis 100% anliegen. Die Fahrtrichtung ergibt sich durch das Vorzeichen des Stellwertes. Die Höhe des Wertes bestimmt dabei die Intensität.

Der Lenkservo wird direkt mit einem PWM-Signal angesprochen. Je nach Einstellwinkel muss ein Signal mit einer Pulsbreite von 500 µs to 2500 µs anliegen. Negative Stellwerte entsprechen einer Ausrichtung nach rechts und positive Werte einer Ausrichtung nach links. Die Höhe des Wertes bestimmt wie stark der Servomotor in die jeweilige Richtung auslenkt.

Pro Auto muss die Einstellvariable *servo_trim*, die den Einstellwinkel bei einem Wert von 1 bestimmt, kalibriert werden. Das ist nötig, da der Servomotor die maximal mögliche Auslenkung der Lenkung überschreiten und sie somit beschädigen kann.

Ein PWM-Signal wird mittels der C-Bibliothek [12] generiert, die mit in den Node eingebunden ist.

4.1.6 GNSS

Die absolute Positionsbestimmung auf dem TinyCar erfolgt über Ultra-wideband. Ein *uwb_localize* Node wertet die Sensordaten aus und generiert daraus eine *Pose* Nachricht mit der Position des Autos in der Welt.

Der genaue Ablauf wird in Kapitel 5 beschrieben.

4.2 Carla-Hardware

Mittels der Carla-ROS-Bridge können Sensordaten aus dem Simulator in die ROS-Umgebung versendet werden und umgekehrt Steuerbefehle aus ROS zurück in den Simulator gelangen.

Vor der Nutzung muss jedoch eine Instanz des Carla Simulators gestartet und das Auto mit den verwendeten Sensor erstellt werden.

Erstellt wird:

1. Frontkamera
2. Kamera Vogelperspektive (Für Debugging Zwecke)
3. IMU
4. GNSS

Eine Odometrie wird von allein versendet.

4.2.1 Frontkamera

Bei der Frontkamera handelt es sich um eine RGB Kamera mit einem Sichtfeld von 120°. Die Bilder der Kamera sind nicht verzerrt, eine Entzerrung ist daher nicht notwendig.

4.2.2 IMU

Die in Carla erstellte IMU kann direkt verwendet werden. Aus der Bridge werden Nachrichten vom Typ *IMU* gesendet.

Rad-Encoder

Der Carla-Simulator stellt zur Zeit noch keine Möglichkeit bereit die aktuelle Position des Rades auszulesen. Um einen Rad-Encoder zu simulieren wurde der Node *carla_fake_wheel_encoder* erstellt.

Der Node empfängt die aus dem Simulator gesendet Odometrie und berechnet den zurückgelegten Weg seit der letzten Nachricht. Mithilfe des Radumfanges wird anhand des zurückgelegten Weges der aktuelle Winkel vom Rad bestimmt.

Der errechnete Winkel wird ebenfalls in einer WheelState Nachricht versendet.

4.2.3 Steuerung

Fahrzeuge in Carla lassen sich aus ROS durch *CarlaEgoVehicleControl* Nachrichten steuern. Die Nachrichten beinhalten wie stark gebremst, eingelenkt und das Gaspedal betätigt werden soll und in welche Richtung es fährt.

Der Node *carla_control* wandelt die für das TinyCar bestimmten Steuerbefehle in *CarlaEgoVehicleControl* Nachrichten um. Werte für den Getriebemotor und die Steuerung werden unverändert in die *CarlaEgoVehicleControl* Nachricht eingetragen. Der Rückwärtsgang und die Bremse werden von allein betätigt. Somit verhält sich der *carla_control* Node identisch zu der TinyCar Steuerung.

4.2.4 GNSS

Der GNSS Sensor im Carla Simulator publiziert *NavSatFix* Nachrichten, welche den Längengrad, den Breitengrad und die Höhenlage enthalten. Nach Erhalt einer Nachricht werden die GNSS Daten durch den *carla_gnss_to_world* Node in das Koordinatensystem von ROS konvertiert.

Anschließend werden die umgerechneten Koordinaten als *Pose* zur Verfügung gestellt.

4.3 Controller

Der TinyCar Controller ist dafür verantwortlich Fahrkommandos zu empfangen und daraus entsprechende Signale für die Aktorik-Nodes zu generieren. Außerdem erstellt er eine Odometrie, die aus Fahrkommandos und den Daten des Rad-Encoders berechnet wird.

Empfangen werden *AckermannDriveStamped* und *WheelState* Nachrichten. Jede Ackermann Nachricht enthält Sollwerte für Beschleunigung, Geschwindigkeit und Lenkwinkel. Eine *WheelState* Nachricht enthält die aktuelle Position des Rades und führt zu einer sofortigen Berechnung der aktuellen Geschwindigkeit und Beschleunigung.

Abbildung 4.2 zeigt den Internen Aufbau des Controllers.

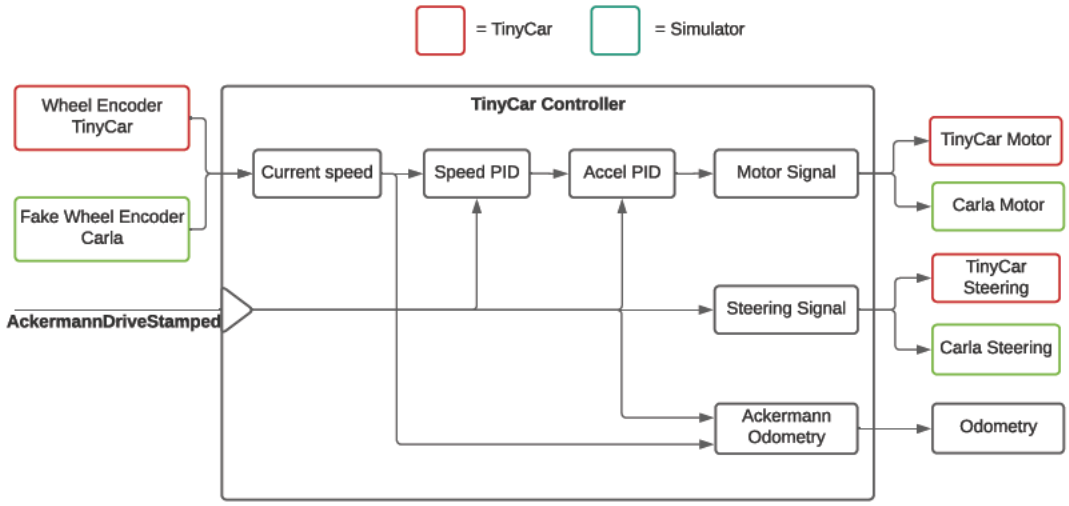


Abbildung 4.2: Aufbau des TinyCar Controllers.

4.3.1 Steuersignale

Der Controller aktualisiert die Werte für die Aktorik in einem festen Intervall. In jedem Zyklus der Hauptschleife wertet er den gemessenen Zustand aus und vergleicht ihn mit den Sollwerten. Daraufhin werden die neuen Signale an die Nodes der Aktorik versendet.

Für die Berechnung der Einstellwerte von Beschleunigung, Geschwindigkeit sowie Lenkung sind drei PID Regler vorgesehen. Aufgrund fehlender Feedbackmechanismen für den eingestellten Lenkwinkel auf dem TinyCar wird in dieser Arbeit jedoch kein PID für die Lenkung implementiert.

Um einen neuen Ausgabewert zu erstellen, nutzt der Geschwindigkeitsregler die aktuelle Geschwindigkeit sowie den Geschwindigkeitssollwert. Für die Weiterverarbeitung im Beschleunigungsregler kann der erzeugte Ausgabewert als Beschleunigungssollwert interpretiert werden. Dabei vergleicht der Beschleunigungsregler den Sollwert mit der aktuellen Beschleunigung, um den geregelten Wert zu bestimmen.

Die resultierende Beschleunigung wird zu der aktuellen Geschwindigkeit addiert. Daraus ergibt sich ein neuer Geschwindigkeitswert, der anschließend in dem Wertebereich -1 bis 1 skaliert werden muss. Dafür wird die maximal mögliche Geschwindigkeit genutzt.

Der skalierte Wert wird als *Float32* Nachricht an die Nodes der Aktorik versendet.

4.3.2 Ackermann Odometrie

Eine Odometrie aus den Daten des Rad-Encoders ist zwingend notwendig, da die weitere Aktorik nicht für eine präzise Lokalisierung reicht.

Nach jedem Erhalt einer *WheelState* oder *AckermannDriveStamped* Nachricht wird die derzeitige Position neu berechnet. Dafür wird zuerst ermittelt, welcher Teil der Geschwindigkeit in seitliche Richtung verläuft.

Der Radius des Kreises den das Auto beim Einlenken abfährt, lässt sich wie folgt berechnen:

$$r = \frac{R}{\tan(W)}$$

Wobei R den Radstand und W den Lenkwinkel beschreibt.

Die Winkelgeschwindigkeit um die Hochachse lautet:

$$\omega = \frac{v}{r}$$

v ist die Geschwindigkeit des Autos.

Und die Änderung der Ausrichtung um die Hochachse ist:

$$\Delta yaw = \omega \cdot \Delta t$$

$$yaw = yaw + \Delta yaw$$

Die Summe alle Änderungen beschreibt die aktuelle Ausrichtung um die Hochachse. Die neue Position lässt sich dann wie folgt berechnen:

$$v_{lat} = radius \cdot (1 - \cos(\Delta yaw))$$

$$v_{fwd} = radius \cdot \sin(yaw)$$

$$\Delta x = v_{fwd} \cdot \cos(yaw) - v_{lat} \cdot \sin(yaw) \cdot \Delta t$$

$$\Delta y = v_{fwd} \cdot \sin(yaw) + v_{lat} \cdot \cos(yaw) \cdot \Delta t$$

$$x_i = x_{i-1} + \Delta x$$

$$y_i = y_{i-1} + \Delta y$$

Die berechnete Odometrie wird anschließend als *Odometry* Nachricht publiziert.

5 Lokalisierung

Die Lokalisierung ist ein wichtiger Aspekt für die autonome Pfadplanung. Mit ihr ist es möglich abzuschätzen, wo sich das Auto in der Welt befindet und wie es fahren muss, um ein Ziel zu erreichen.

In ROS werden verschiedene Koordinatensysteme in Frames unterteilt. Anhand der Frames können Positionen eines Koordinatensystems direkt in die eines anderen umgerechnet werden. Es existiert ein festgesetzter Frame *map*, zu dem alle weiteren Frames relativ angegeben werden. Verschiedene Nodes können Transformationen von einem Frame zu einem anderen Frame publizieren, damit sie für alle Nodes zur Verfügung stehen.

Da einzelne Sensoren nicht perfekt sind und immer eine gewisse Ungenauigkeit vorliegt, ist eine Fusion mehrerer Sensorwerte wünschenswert. Dafür werden die Sensorwerte der IMU, des Rad-Encoders und die globale Position genutzt. Die Fusionierung der einzelnen Sensorwerte erfolgt anhand eines Extended-Kalman-Filters. Eingesetzt wurde das existierende Package *robot_localization*, welches alle Voraussetzungen erfüllt.

5.1 Robot Localization

Das Package *robot_localization* [16] wurde mit dem Ziel entwickelt einen Kalman-Filter möglichst flexibel und einfach in ROS verwenden zu können. Es lassen sich beliebig viele Sensoren über eine Konfigurationsdatei mit in die Fusionierung einbinden. Verschiedene Sensoren vom gleichen Typ können auch mehrfach mit in die Fusionierung eingebunden werden, wie zum Beispiel eine zweite IMU. Dabei sind Nachrichten vom Typ *Odometry*, *Imu*, *PoseWithCovarianceStamped* oder *TwistWithCovarianceStamped* erlaubt.

Über den Parameter *world_frame* lässt sich angeben zu welchem Frame die Position relativ abgeschätzt werden soll.

Jeder einzelne Sensor lässt sich durch eine Vielzahl an Parametern speziell an den Aufgabenbereich anpassen. Es ist ebenfalls möglich spezifisch auswählen, welche Teile der Sensordaten ausgeschlossen oder mit einbezogen werden sollen.

Aus den Werten wird dann eine Schätzung für den 15-dimensionalen Zustand des Autos errechnet. Der Zustand setzt sich wie folgt zusammen:

$(X, Y, Z, roll, pitch, yaw, \dot{X}, \dot{Y}, \dot{Z}, \dot{roll}, \dot{pitch}, \dot{yaw}, \ddot{X}, \ddot{Y}, \ddot{Z})$

Nach Erhalt der ersten Nachricht werden konstant Werte für den Zustand berechnet. Falls durch einen Ausfall temporär Sensordaten fehlen, schätzt der Filter die zu erwartenden Werte weiter ab.

Außerdem werden mit dem Filter leichte Abweichungen durch die Fusion mit den anderen Sensordaten ausgeglichen.

5.2 Konfiguration

Absolute Positionsdaten enthalten, ähnlich wie bei einem echten GPS Sensor, Sprünge in den Werten. Durch die sprunghaften Änderungen entstehen auch Sprünge in dem durch *robot_localization* generierten Zustand.

Während Sprünge im Map Frame unproblematisch sind, sollte sich der Zustand im Odom Frame stets kontinuierlich und nicht sprunghaft bewegen. Denn die Pfadplanung wird im Odom Frame ausgeführt und Sprünge in der Positionen führen ebenfalls zu sprunghaften Änderungen im geplanten Pfad.

Deshalb werden zu Beginn zwei Nodes vom Typ *ekf_localization_node* gestartet. Bei einem der Nodes wird der Parameter *world_frame* auf den Wert des Odom Frames gesetzt, bei dem anderen auf den Wert des Map Frames. Dadurch publiziert einer der Nodes den Transform *map -> odom* und der andere *odom -> base_link*.

In den Map Transform fließen die Daten der TinyCar Odometrie, der IMU und eine globale Position ein, die wie im Abschnitt 5.3 beschrieben, generiert wird.

Der Odom Transform nutzt lediglich die Daten der TinyCar Odometrie und der IMU, was zu einer kontinuierlichen Bewegung führt.

5.3 Ultra-wideband

Eine Positionsbestimmung anhand IMU und Rad-Encoder ist nur in der Lage das Auto relativ zu der Startposition zu lokalisieren. Um die absolute Position des Autos auf der Teststrecke bestimmen zu können, ist ein externes System, ähnlich wie das GPS bei einem echten Auto, nötig.

Ultra-wideband (UWB) ist eine Funktechnologie, die zur Kommunikation über kurze Distanzen vorgesehen ist. Wie der Name bereits sagt, arbeitet UWB mit sehr hohen Bandbreiten. Die Sendeleistung ist auf nur wenige Milliwatt begrenzt, wird dafür aber auf ein sehr breites Frequenzspektrum verteilt.

Statt ein kontinuierliches Signal zu senden, schickt UWB kurze Impulse mit einer Dauer von etwa 2 ns. Andere Funktechniken werden durch UWB nicht gestört, da die schwachen Impulse für sie als Rauschen erscheinen. Ebenso kann UWB selbst nicht von anderen Funktechniken gestört werden.

UWB ist besonders zu Zwecken der Lokalisierung geeignet, da die Positionsbestimmung auf dem Messen der Signallaufzeiten statt Signalstärke basiert. Ein starkes Signal ist somit nicht notwendig.

Double-Sided-Two-Way-Ranging

Für das Ranging wird das Double-Sided-Two-Way-Ranging Verfahren genutzt. Es soll die Distanz zwischen einem sich bewegenden *Tag* und einem festliegenden *Anchor* messen.

Abbildung 5.1 zeigt den Ablauf einer Ranging Anfrage des Tags an einen Anchor.

Bei jedem Empfang einer Nachricht wird der genaue Zeitpunkt des Erhalts gespeichert. Ebenso wird bei jedem Senden einer Nachricht der genaue Zeitpunkt des Absendens gespeichert.

Zu Beginn sendet der Tag (Device A) eine Poll Nachricht an den Anchor (Device B). Der Anchor empfängt die Nachricht, speichert den Zeitpunkt und sendet eine Antwort, die den Ranging Vorgang bestätigt, zurück. T_{reply1} ist die Zeitspanne zwischen dem Erhalt und dem Senden der Nachricht.

Der Tag empfängt die Rückmeldung und speichert die Roundtrip Zeit T_{round2} ab. Außerdem sendet er eine weitere Nachricht ab, welche T_{round1} und T_{reply2} enthält. T_{reply2}

ist die vergangene Zeit zwischen dem Erhalt der Rückmeldung und dem Absenden der Nachricht.

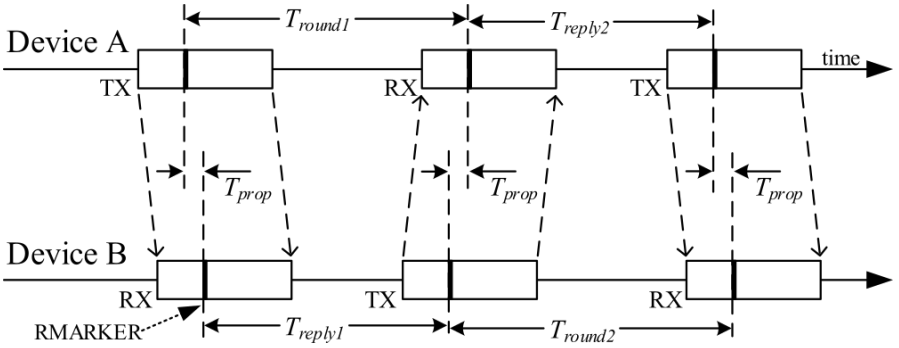
Der Anchor empfängt diese Nachricht und ermittelt die Roundtrip Zeit T_{round2}.

Mittels folgender Formel lässt sich eine Schätzung der Time-Of-Flight berechnen.

$$\hat{T} = \frac{T_{round1} \cdot T_{round2} - T_{reply1} \cdot T_{reply2}}{T_{round1} + T_{round2} + T_{reply1} + T_{reply2}}$$

Die Distanz zwischen Tag und Anchor ist das Produkt der Time-Of-Flight und der Lichtgeschwindigkeit.

Das standardisierte Double-Sided-Two-Way-Ranging Verfahren ist an dieser Stelle beendet, da die Distanz ermittelt wurde. Allerdings sollen alle weiteren Berechnungen auf dem TinyCar und nicht durch ein externes System erfolgen. Deshalb wurde das Verfahren um eine zusätzliche Nachricht erweitert, die die Distanz an den Tag zurücksendet.



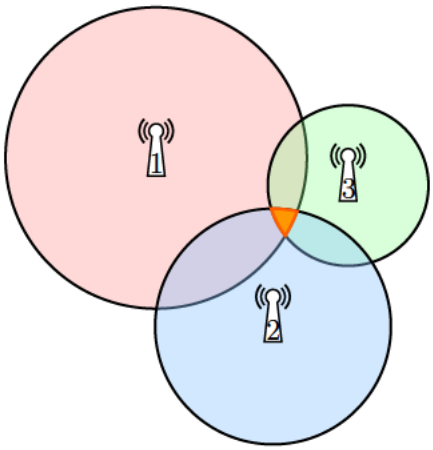
Quelle: DW1000 User Manual

Abbildung 5.1: Squenzieller Ablauf eines Double-Sided-Two-Way-Ranging Verfahrens.

5.3.1 TinyCar UWB

Um das TinyCar zu lokalisieren, besitzt es ein DWM1000 Modul des Herstellers Decawave, das als Tag konfiguriert ist. Rund um das Mikrowunderland sind weitere DWM1000 Module an bekannten Stellen verteilt, die als Anchors dienen.

Das auf dem TinyCar verbaute Modul ist mit einem STM32 ausgestattet, der die Logik für die Kommunikation mit dem DW1000 Chip enthält. In einem festen Intervall wird eine Ranging Anfrage an alle eingespeicherten Adressen gesendet.



Quelle: Wikipedia

Abbildung 5.2: Prinzip der Trilateration mit drei Anchors.

Die Antwort der einzelnen Anchors enthält die gemessene Distanz zwischen Anchor und Tag. Wenn die Distanz zu mehreren Anchors mit einer bekannten Position gemessen wurde, lässt sich daraus die Position des Tags ermitteln.

Dafür gibt es mehrere Ansätze, von denen zwei in dieser Arbeit genauer getestet wurden.

5.3.2 Trilateration

Der erste Ansatz besteht darin, durch Trilateration die Position des Autos zu berechnen. Dafür wurde ein eigener Node *uwb_localize* entwickelt, der wiederum die Python Bibliothek *localization* benutzt.

Als Eingabe empfängt der Node eine *Float32MultiArray* Nachricht, in der in fester Reihenfolge die Distanzen zu jedem Anchor stehen. Seine Ausgabe ist eine *Pose* Nachricht, welche als X- und Y-Werte die geschätzte Position des Autos in der Welt besitzt.

Dadurch, dass der Abstand von dem Tag zu dem Anchor sowie die Position des Anchors bekannt ist, lässt sich die mögliche Position des Tags als Kreis um den Anchor darstellen. Abbildung 5.2 zeigt den durch drei Anchors eingeschlossenen Lösungsbereich, in dem sich das Auto potenziell befinden kann.

Da es durch das Rauschen in den Senorwerten keine perfekte Lösung gibt, wird durch das Least square error (LSE) Verfahren die Position ermittelt, die den geringsten Fehler aufweist.

5.3.3 Partikel Filter

Der zweite Ansatz ist die Nutzung eines Partikel Filters. Partikel Filter eignen sich nicht nur für die Lokalisierung in einer bekannten Karte durch LiDAR Technologie, sondern auch für den Einsatz mit UWB.

Für die Implementierung des Partikel Filters wurde die Mobile Robot Programming Toolkit (MRPT) Bibliothek verwendet. Das *mrpt_ navigation* package wird genutzt, um innerhalb von ROS Gebrauch von den durch MRPT bereitgestellten Algorithmen zu machen.

MRPT empfängt eigene Nachrichten vom Typ *ObservationRangeBeacon* und *SingleRangeBeaconObservation*, welche die Distanzen, IDs sowie Informationen über die Sensor Charakteristiken enthalten. Der *mrpt_ conversion* Node übersetzt die vom TinyCar gemessenen Distanzen in MRPT kompatible Nachrichten.

5.4 Overhead-Kamera

Eine Alternative zu der Verwendung von UWB stellt das Nutzen eines kamerabasierten Ortungssystems dar. Das Auto kann lokalisiert werden, indem eine überhalb der Teststrecke angebrachte Kamera einen Marker erkennt, der auf dem Dach des Autos fixiert ist.

Anhand des erkannten Markers kann die Position und die Ausrichtung des Autos berechnet werden. Dafür müssen die Position der Kamera relativ zur Welt und die genauen Maße des Markers bekannt sein.

Verwendet wurde ein Aufbau mit einer Raspberry Pi HQ-Kamera in Kombination mit einem 6mm Objektiv. Wie auch bei dem Kamerabild des TinyCars wurde vor der Verwendung die Kamera für die Bildentzerrung kalibriert.

Um die Marker zu erkennen und ihre genaue Position zu bestimmen, wurde die Bibliothek OpenCV [3] genutzt. OpenCV stellt Funktionen bereit, um die Marker zu erkennen und ihre Position zu bestimmen.

Da es sich bei der kamerabasierten Lokalisierung um ein externes System handelt, wurde es in zwei Teile unterteilt. Ein Raspberry Pi 4 erstellt einen MQTT Server, in dem Geräte wie das TinyCar Nachrichten erhalten können. Für jeden Marker den der Raspberry Pi erkennt wird die ID und Position in dem MQTT Server publiziert.

Auf dem TinyCar empfängt ein ROS Node die Position, überprüft die ID des Markers und publiziert eine *Pose* Nachricht.

5.4.1 Ultra-wideband vs. Overhead-Kamera

Der entscheidende Vorteil der kamerabasierten Lokalisierung liegt in der höheren Genauigkeit und der vereinfachten Anwendung. Während bei der UWB Lokalisierung viele einzelne Komponenten zusammen agieren müssen, sind hier nur eine Kamera und ein Marker notwendig.

Je nach Kamera, Linse und Aufnahmeauflösung ist die Lokalisierung auf wenige Millimeter genau. Damit ist sie als Groundtruth oder zum Testen neuer Algorithmen geeignet. Nachteilig sind die hohen Kosten pro Kamera und die schlechte Skalierbarkeit, da eine Kamera nur wenige Quadratmeter abdecken kann. Außerdem müssen ein freies Sichtfeld und gute Lichtverhältnisse gewährleistet sein.

UWB hingegen ist stark skalierbar, da ein einzelnes Modul laut Herstellerangaben eine Reichweite von bis zu 290 m hat. Der Preis pro Modul ist deutlich geringer und die Lokalisierung ist unabhängig von Licht-, und Sichtverhältnissen.

Dafür ist die Lokalisierung ungenauer, fehleranfällig und schwer zu kalibrieren.

Im Gegensatz zu einer Overhead-Kamera handelt es sich bei UWB um ein Konzept, das sich auch auf echte Autos übertragen lässt. UWB verhält sich sehr ähnlich wie GPS, das in jedem Auto verbaut ist.

6 Teilautonomie

Ein Ziel dieser Arbeit war es, Teilautonomie Funktionen auf Basis der entwickelten ROS-Architektur zu entwickeln. Konkret soll es möglich sein, dass das Auto einen ausgewählten Zielpunkt von alleine anfährt. Der Zielpunkt kann durch das Tool RViz auf einer Karte des Straßennetzes angegeben werden.

Es wird vorausgesetzt, dass eine grobe Kartierung des Straßennetzes existiert, da diese ebenfalls zum Ermitteln des globalen Plans verwendet wird. Der lokale Plan basiert rein auf der durch die Kamera erkannten Straße.

Dadurch kann die echte Straße stark von der Karte abweichen, ohne, dass das Auto von ihr abkommt. Dieses Verhalten wurde als Anforderung bewusst gewählt, da somit die globale Karte aus einer groben Beschreibung, wie zum Beispiel OpenDrive oder OpenStreetMap, generiert werden kann.

Die Ausführung eines Fahrbefehls lässt sich grob in drei Schritte unterteilen.

1. **Straßenerkennung**

   Zuerst muss ermittelt werden, welcher Teil der Straße befahrbar ist und als freie Zone in die Costmap eingetragen werden kann. Dafür wird ein neuronales Netzwerk verwendet, das eine semantische Segmentierung durchführt.

2. **Generierung der Costmap**

   Bevor die Aufnahmen der Frontkamera in die Costmap eingetragen werden können, müssen sie zuerst gefiltert und in die Vogelperspektive umgerechnet werden. Danach wird die erkannte Straße durch eine CostmapLayer in die Costmap eingetragen.

3. **Pfadplanung**

   Um das ausgewählte Ziel zu erreichen, muss das Auto anhand der lokalen und globalen Costmap einen Weg berechnen und ihm folgen.

6.1 Straßenerkennung

Um einen Pfad planen zu können, muss der befahrbare Bereich klar von dem nicht befahrbaren Bereich unterschieden und zellenweise in die Costmap eingetragen werden. Bei einer Spur- oder Linienerkennung müssten demnach die erkannten Linien zu einem geschlossenen Bereich konvertiert werden. Auf einer geraden Straßen ist das noch leicht möglich, doch in Kurven oder Kreuzungen ist dieser Ansatz schwer umsetzbar. Außerdem sind aktuelle state of the art Ansätze in der Spurerkennung für Autobahnen und Landstraßen und nicht für Kreuzungen oder scharfe Kurven ausgelegt.

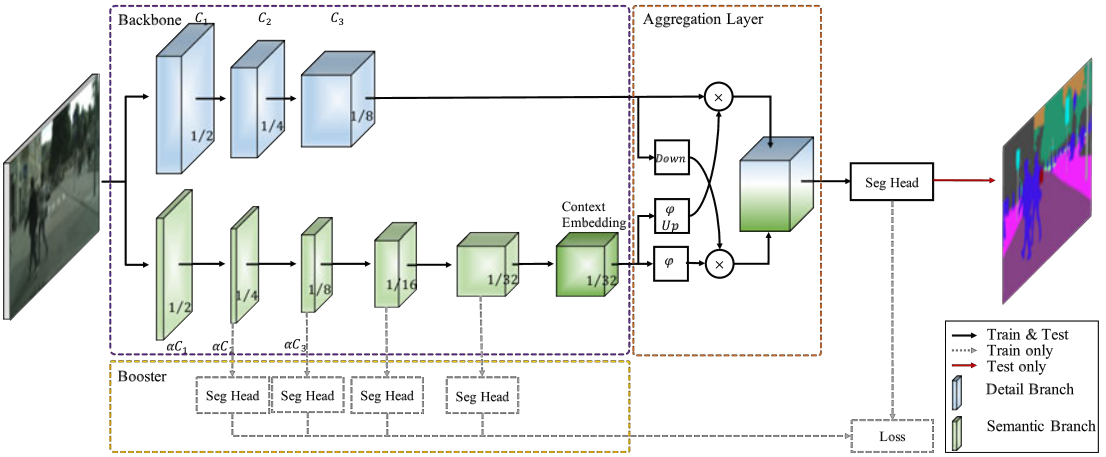
Der Vorteil einer semantischen Segmentierung ist, dass der gesamte befahrbare Bereich erkannt wird und keine weitere Konvertierung nötig ist. Auch in engen Kurven oder Kreuzungen hat die Segmentierung eine hohe Erkennungsrate, allerdings wird nicht zwischen eigener und Gegenfahrbahn unterschieden.

Ein weiterer Nachteil von Segmentierungsnetzen liegt in dem erhöhten Rechenaufwand durch größere Netzwerke, da die Output-Auflösung direkt mit der Output-Größe zusammenhängt.

Durch die Verwendung des TinyCars entstehen einige spezielle Anforderungen an das neuronale Netzwerk. Da die Rechenleistung der CPU bei weitem nicht für einen Echtzeit-Einsatz ausreicht, muss die verbaute Google EdgeTPU genutzt werden. Die EdgeTPU benötigt 8 Bit unsigned Integer Werte. Jedes Model das genutzt werden soll, muss deshalb vorher durch Quantisierung kompatibel gemacht werden.

Außerdem muss sich das Model in Googles Tensorflow Lite Format [7] befinden. Tensorflow Lite ist eine für Embedded Geräte optimierte Version des normalen Tensorflow Frameworks. Durch die Umrechnung von Float32 in Uint8 Werte kann der benötigte Rechenaufwand und Verbrauch von Arbeitsspeicher stark reduziert werden.

Für die Umrechnung stellt Google den EdgeTPU Compiler bereit, der allerdings noch nicht alle Operationen des normalen Tensorflow Frameworks implementiert hat. Bei der Wahl des Netzwerkes muss also auf Kompatibilität mit dem EdgeTPU Compiler geachtet werden.



Quelle: Paper BiSeNet V2

Abbildung 6.1: Die Zweige der BiSeNet V2 Architektur.

6.1.1 BiSeNet V2

Die Wahl fiel auf ein BiSeNet V2 [21], da es einen guten Kompromiss zwischen Geschwindigkeit und Performance darstellt. BiSeNet V2 wurde im April 2020 veröffentlicht und erreicht im Real-Time Semantic Segmentation on Cityscapes Test mean intersection over union (mIoU) von 75,3%.

Die Kernidee besteht darin, die semantischen Informationen sowie die Details mit in die Segmentierung einzubringen. Während sich semantische Informationen gut durch tiefe Netzwerke ermitteln lassen, gehen Details in den tiefen Schichten verloren. Details hingegen lassen sich gut in flachen Schichten erhalten in denen sich semantische Informationen nicht ausreichend erkennen lassen. Die optimale Tiefe zum Erhalten der Details und semantischen Informationen widerspricht sich.

BiSeNet V2 löst das Problem durch eine doppelzweigige Struktur, wie in Abbildung 6.1 zu sehen ist. Der Detail Zweig hat flachere Schichten mit breiten Channels während der semantische Zweig tiefe Schichten mit dünnen Channels besitzt. Außerdem ist die Auflösung im semantischen Zweig um Faktor vier reduziert, um semantische Merkmale besser zu erkennen.

Ein weiterer Abschnitt der BiSeNet V2 Architektur ist dafür verantwortlich, den semantischen Zweig mit dem Detail Zweig zu fusionieren. Durch Up- und Downsampling wird

die Auflösung der Zweige an den jeweils Anderen angepasst und elementweise mit ihm multipliziert.

Zum Trainieren wurden drei verschiedene Datensätze genutzt:

1. CityScapes:

   CityScapes [5] ist ein gängiger Datensatz um die Performance verschiedener Segmentierungsnetzwerke zu vergleichen. Der Datensatz wurde in 50 deutschen Städten aufgenommen und enthält 4000 pixelgenau segmentierte Bilder. Alle Bilder sind manuell ausgewählt, um viele dynamische Objekte und eine hohe Diversität an Hintergründen sowie Umgebungen zu garantieren.

   CityScapes enthält größtenteils innerstädtliche Verkehrssituationen, die häufig auf Kreuzungen oder engen Straßen aufgenommen wurden. Das Mikrowunderland enthält viele Kreuzungen und enge Bereiche, jedoch keine Autobahnen. Eine Vielzahl an Datensätzen wurden hingegen auf Autobahnen aufgenommen, weshalb sie für das TinyCar ungeeignet sind.

2. Mikrowunderland:

   Auf dem Mikrowunderland selbst wurde ebenfalls ein Datensatz aufgezeichnet, der etwa 1000 Bilder enthält. Verwendet wurde dafür bereits ein TinyCar, weshalb auch die Optik für den Praxiseinsatz dieselbe ist. Auf den Bildern wurden manuell der befahrbare Bereich als Straße markiert.

3. Carla:

   Der Carla Simulator kann für das Kamerabild automatisch eine Segmentierung generieren. Es wurden ca. 1500 Bilder auf Karte Town02 aufgenommen.

   Der Datensatz wurde im Synchronen Modus von Carla generiert, um zu garantieren, dass das Kamerabild und die Segmentierung zusammen passen. Durch die begrenzte Kartengröße weißt der Datensatz keine hohe Vielfalt auf, jedoch wurden verschiedene Wettersituationen wie starker Regen oder dunkles Licht simuliert.

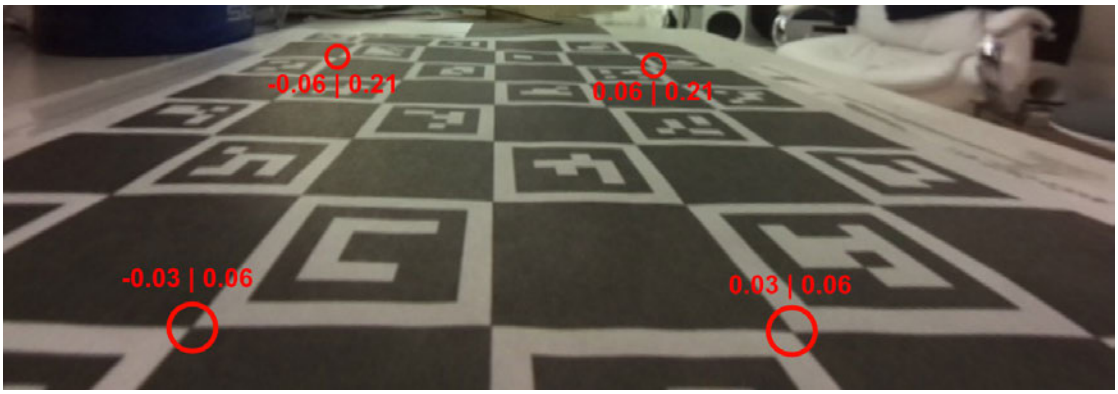


Abbildung 6.2: Referenzpunkte und zugehörige Koordinaten in Metern, aufgenommen auf der Kamera des TinyCars.

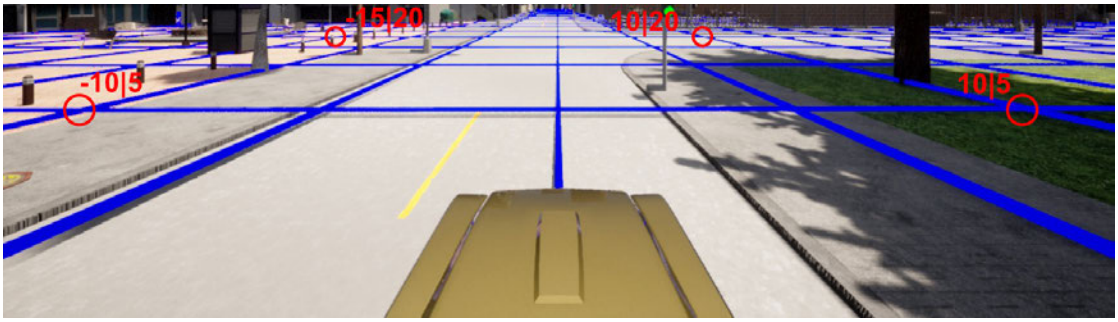


Abbildung 6.3: Referenzpunkte und zugehörige Koordinaten in Metern, Bild aus dem Carla Simulator mit Referenzstreifen.

6.1.2 Vogelperspektive

Bevor die erkannte Straße in eine Costmap eigetragen werden kann, muss das Kamerabild aus der Perspektive der Windschutzscheibe in die Vogelperspektive umgerechnet werden. Da die Straßen im Mikrowunderland keine Steigungen enthalten, wird in dieser Arbeit angenommen, dass es sich nur um zweidimensionale Straßen handelt. Durch diese Vereinfachung lässt sich eine Perspektive durch eine konstante Homografie Matrix in eine andere Perspektive umrechnen.

Für die Berechnung der Matrix müssen vier Punktpaare bekannt sein. Dafür müssen vier Punkte aus einem Quellbild und ihre erwünschten Koordinaten in der Zielperspektive ausgewählt werden. Die Zielkoordinaten können direkt in der Einheit Meter angegeben werden, die Quellkoordinaten sind Pixel im Eingabebild.

Aufgrund der abweichenden Kameraposition und Optik zwischen Carla und dem TinyCar, muss für beide eine eigene Homografie Matrix berechnet werden. Um ein möglichst genaues Ergebnis zu erreichen, muss mit der verwendeten Kamera ein Bild von Referenzpunkten aufgenommen werden. Die Art und Weise der Punktselektion unterscheidet sich zwischen Carla und dem TinyCar.

1. TinyCar:

   Für das TinyCar wurde ein Schachbrettmuster mit bekannten Abstand zwischen den Kästen verwendet. In Abbildung 6.2 sind das aufgenommene Bild und die eingezeichneten Transformationspunkte zu sehen. Durch die bekannten Abstände lässt sich für jeden Punkt die genaue Position relativ zum Auto bestimmen.

2. Carla:

   In Carla erweist sich das Problem als schwerer, denn es ist nicht möglich ein Schachbrettmuster vor die Kamera zu halten. Der Simulator stellt jedoch Funktionen bereit, um Linien, Schrift, Punkte oder Pfeile in die Welt zu zeichnen.

   Die Linien wurden genutzt, um auf Straßenhöhe ein Gittermuster mit einer Kantenlänge von 2 m zu zeichnen. Abbildung 6.3 zeigt das Muster und die verwendeten Referenzpunkte. Anhand der Linien ist es möglich genaue Abstände abzulesen.

Da die Transformation jedoch nicht auf das Eingabebild, sondern auf das Ausgabebild angewendet werden soll, müssen zuerst alle Quellkoordinaten in die Ausgabegröße des BiSeNet V2 umgerechnet werden.

Die Umrechnung erfolg mit:

$$x = P_x \cdot W_{seg}/W_{img}$$

$$y = P_y \cdot H_{seg}/H_{img}$$

W beschreibt die Breite und H die Höhe des Quellbildes (img) und des Ausgabebildes vom BiSeNet V2 (seg). P ist die Quellkoordinate die umgerechent werden soll.

Um weitere Schritte zu erleichtern und die Performance zu steigern werden außerdem die angegebenen Zielkoordinaten direkt in Zellen auf der Costmap umgerechnet. Eine Costmap hat eine feste Anzahl an Zellen in X und Y Richtung, die sich anhand Breite, Höhe und Auflösung ergeben.

Da die Zielkoordinanten bereits in Metern angegeben sind, lassen sie sich mittels der Auflösung fest einer Zelle auf der Costmap zuordnen.

$$x = O_x + P_x/R$$

$$y = O_y - P_y/R$$

O ist der Ursprung der Costmap, R die Auflösung und P die Koordinate in Metern, die umgewandelt werden soll.

Die Homografie, die aus diesen Punkten berechneten wird, ist nun in der Lage das Ausgabebild des BiSeNet V2 in die Vogelperspektive umzurechnen und dabei einen wahrheitsgemäßen Maßstab für die Costmap einzuhalten.

Abbildung 6.4: Eingabebild für die Transformation.

Abbildung 6.5: Segmentierug nach der Transformation in die Vogelperspektive.

Die Abbildungen 6.4 und 6.5 zeigen ein Eingabebild und die daraus generierte Vogelperspektive.

6.2 Costmap Berechnung

Eine Costmap ist eine simple Datenstuktur, die für jede ihrer Zellen die zugehörigen Kosten als uint8 speichern kann. Jede Zelle repräsentiert eine Fläche im zweidimensionalen Raum. Einer Costmap wird eine feste Breite, Höhe und Auflösung in Metern zugeteilt.

Auch wenn jede Zelle 256 unterschiedliche Werte haben kann, arbeitet die Costmap intern nur mit den drei Zuständen Occupied, Free Space und Unknown. Dem Routenplaner ist es jedoch selbst überlassen, wie er die Kosten pro Zelle interpretiert und behandelt.

Occupied, Free Space und Unknown sind den Werten 254, 0 und 255 zugeordnet. Wenn die Sensorik eines Fahrzeuges ein Objekt im Raum erkennt, würde in der entsprechenden Zelle der Wert von Occupied, also 254, eigetragen werden.

Eine Costmap ist in mehrere Schichten unterteilt, die in der Konfigurationsdatei beschrieben werden. Die Schichten sind für eine spezielle Aufgabe entwickelt und sind dafür verantwortlich die Costmap mit Werten zu füllen.

Periodisch werden alle Schichten in fester Reihenfolge aktualisiert und schreiben die Änderungen in die einzelnen Zellen. Zum Beispiel lädt die *static_layer* Schicht eine Karte aus einer Datei. Die *obstacle_layer* Schicht dagegen, erkennt in den Sensordaten Hindernisse. Für die Straßenerkennung wurde eine eigene Schicht entwickelt, die im Abschnitt 6.2.1 näher beschrieben wird.

6.2.1 Costmap im TinyCar

Es werden zwei Costmaps zum Planen von Routen verwendet. Die globale Costmap hat einen festen Ursprung und wird durch die *static_layer* Schicht aus einer fertigen Karte geladen. Hindernisse werden in der globalen Karte nicht eingetragen. Die Lokale Costmap ist an den *base_link* Frame des TinyCars verankert und bewegt sich immer mit dem Auto mit. In ihr wird die erkannte Straße eingetragen, die zur weiteren Navigation dient.

Üblichen Konfigurationen tragen Hindernisse sowohl in die lokale als auch in die globale Costmap ein, davon wurde aber bewusst von abgesehen. Der globale Plan soll die Richtung vorgeben, ähnlich wie bei einem Navigationssystem. Rücksicht auf die Straße nimmt nur der lokale Plan.

Auch wenn die Costmap fest an dem *base_link* Frame verankert ist, erkennt die Costmap anhand der Odometrie wie sich das Auto in der Welt bewegt hat. Alle Werte, die in die Costmap eingetragen werden, bleiben an einer absoluten Position in einer absoluten Karte, über die sich der Ausschnitt der lokalen Costmap bewegt. Deshalb bleiben bereits erkannte Hindernisse auf der Costmap stehen und verschwinden, sobald sie sich nicht mehr in dem durch die Größe angegebenen Bereich befinden.

Birdview Layer

Die Birdview Layer empfängt die Bilder, die der *BirdviewPublisher* Node nach Berechnung der Vogelperspektive verschickt um diese in der Costmap einzutragen. Ziel ist es, dass die Fläche der Straße als Free Space markiert wird und die Ränder der Straße als Occupied, da sie nicht befahren werden sollen.

Durch die Parameter *x_range* und *y_range* kann eigestellt werden, wie weit in X- beziehungsweise Y-Richtung die Werte eingetragen werden. Ein weitere Parameter *front_cutoff* gibt an, ab welcher Entfernung vor dem Auto die Ränder als Hindernis erkannt werden sollen. Diese Option ist nötig, da anderenfalls der Beginn der Straße, direkt vor dem Auto, als Hindernis erkannt wird.

Die Berechnung der Costmap auf Basis des Ursprungsbildes 6.6 lässt sich in 4 Schritte unterteilen.

1. Schritt

   Zuerst wird aus dem Eingabebild nur der gewünschte Bereich, wie in *x_range* und *y_range* festgelegt, ausgeschnitten. In der weiteren Beschreibung wird dieser Bereich Region Of Interest (ROI) genannt. Das Ergebnis ist in Abbildung 6.7 dargestellt.

2. Schritt

Abbildung 6.6: Ursprungsbild in der Vogelperspektive.

Abbildung 6.7: Die relevante Region of Interest.

Abbildung 6.8: Korrektion der Orientierung.

Abbildung 6.9: Durch Laplace Filter erkannte Kanten.

Um das Bild an die Orientierung des Autos in der Welt anzupassen, muss es entsprechend rotiert werden. Mithilfe des Yaw Wertes des Autos wird eine Rotationsmatrix berechnet und auf die ROI angewandt. Das Ergebnis ist in Abbildung 6.8 dargestellt.

3. Schritt

   Um den Rand der Straße zu erkennen wird ein Laplace Filter angewendet und die Ausgabe in einem separaten Bild gespeichert. Da der Rand die komplette Straße einschließt, wird der vordere Teil bis zur erwähnten *front_cutoff* Reichweite abgeschnitten.

4. Schritt

Anschließend wird durch alle Pixel der ROI iteriert und jeweils überprüft, ob an der Position Straße, Rand oder keines von beiden gesetzt ist. Damit ein Wert in die Costmap geschrieben werden kann, muss mithilfe der Funktion worldToMap die Position in der Karte anhand der globalen Position ermittelt werden. Die globale Position errechnet sich wie folgt:

$$P_X(x) = R_X + (x - U_X) \cdot A$$

$$P_Y(y) = R_Y + (y - U_Y) \cdot A$$

R ist die Position des Roboters in der Welt, U ist der Ursprung, also die Mitte, der ROI und A die Auflösung der Karte.

Dafür ist es wichtig, dass nach der Berechnung der Vogelperspektive ein Pixel äquivalent zu der Breite einer Zelle ist, wie in Abschnitt 6.1.2 beschrieben.

Dadurch, dass sich die Costmap kontinuierlich mit einer nicht perfekten Odometrie weiterbewegt, wird die erkannte Straße teilweise nicht im richtigen Winkel oder leicht versetzt eingetragen. Abbildung 6.10 verdeutlicht das Problem. Das führt zu vielen Hindernissen, die auf der Karte gestreut eingezeichnet werden. Aufgrund der nicht perfekten Segmentierung und falsch erkannten Bereiche des Bildes, die ebenfalls als Hintergrund eingetragen werden, wird der Effekt verstärkt.

Wie in Abbildung 6.10 zu sehen ist, befinden sich in der Costmap viele lose verteilte Teile der Straße. Um diese effektiv filtern zu können, wurde die Filter Layer entwickelt. Eine weitere Aufgabe des Filters umfasst das Ausgleichen kleiner Fehler in der Segmentierung.

Sie ist ebenfalls in Form einer Costmap Layer implementiert und befindet sich in der Reihenfolge nach der Birdview Layer. Der Ablauf der Filter Layer lässt sich ebenfalls in kurze Schritte unterteilen.

1. Schritt

   Für die leichte Bearbeitung mittels OpenCV wird der Zeiger auf die aktuelle Zellbelegung ausgelesen und als OpenCV Bild interpretiert. Aus dem Bild wird der Bereich extrahiert, der als FREE_SPACE angegeben ist und anschließend in ein neues Bild kopiert.

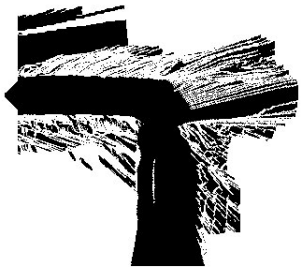


Abbildung 6.10: Costmap vor dem Durchlaufen der Filter Layer.

Abbildung 6.11: Bild nach dem Anwenden der Opening Operation.

Abbildung 6.12: Bild nach dem Anwenden der Closing Operation.

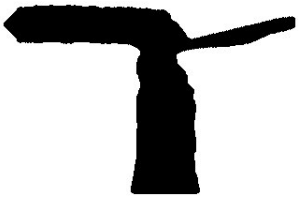


Abbildung 6.13: Kleinere Konturen wurden entfernt.

2. Schritt

   Auf den vorher erkannten Bereich wird eine Opening Operationen angewendet, die einer Erode Operation, gefolgt von einer Dilation Operation, entspricht. Durch das Erode werden die Kanten entlang der Konturen des Bildes geschrumpft. Die darauffolgende Dilation lässt die Konturen wieder wachsen und bringt sie zurück in ihre Originalgröße. Kleine, lose Teile bleiben jedoch verschwunden, da sie nach der Erode Operation vollständig wegfallen. Wie in Abbildung 6.11 zu sehen ist, werden so bereits ein Großteil der Fragmente entfernt.

3. Schritt

   Auf die Ausgabe des vorherigen Schrittes wird anschließend eine Closing Operation angewandt. Im Gegensatz zu Opening wird bei Closing zuerst eine Dilation

und dann eine Erode Operation durchgeführt. Durch die Dilation werden Flächen die nah beieinander liegen, jedoch nicht miteinander verbunden sind, zusammengebracht und kleine Löcher innerhalb von Flächen ausgefüllt. Das Erode bringt die Konturen wieder in ihre Ursprungsgröße zurück. In 6.12 ist zu sehen, dass der kleine freie Bereich innerhalb der Straße geschlossen ist.

4. Schritt

   Zuletzt wird aus allen Konturen nur die Größte herausgefiltert. Dadurch lassen sich übrig gebliebene Flächen, die nicht mit dem Hauptteil der Straße verbunden sind, entfernen. Die Ränder der größten Kontur werden in das referenzierte Bild aus Schritt 1 als LETHAL_OBSTACLE und der innere Teil als FREE_SPACE eingetragen. Es resultiert das Ausgabebild 6.13.

Das Ergebnis hat deutlich weniger Unterbrechungen und lose Bereiche, die den Planer stören können.

6.3 Pfadplanung

Wie bereits erwähnt, wird für die Pfadplanung das Package *move_base* genutzt. *move_base* kann jeweils ein Plugin für die Berechnung des lokalen Plans und ein Plugin für die Berechnung des globalen Plans laden. Da der globale Plan im Falle dieser Architektur nur die grobe Richtung vorgibt, ist die Wahl des Plugins nicht entscheidend.

Der lokale Plan hingegen, muss die Beschränkungen des Roboters einhalten um realistische Pläne zu generieren. Im Gegensatz zu den häufig verwendeten Robotern mit Zwei-Rad basierten Steuerungen, besitzt das TinyCar eine Ackermann Lenkung.

Deshalb dürfen die Steuerkommandos, die die Planung generiert, nicht voraussetzen, dass sich das Fahrzeug auf der Stelle dreht oder zu stark einlenkt.

Der häufig genutzte Dynamic Window Approach (DWA) schied aufgrund dieser Anforderungen als Möglichkeit aus. Getestet wurden zwei verschiedene Local Planner, wobei für beide eine eigens entwickelte Zielpunktwahl verwendet wurde.

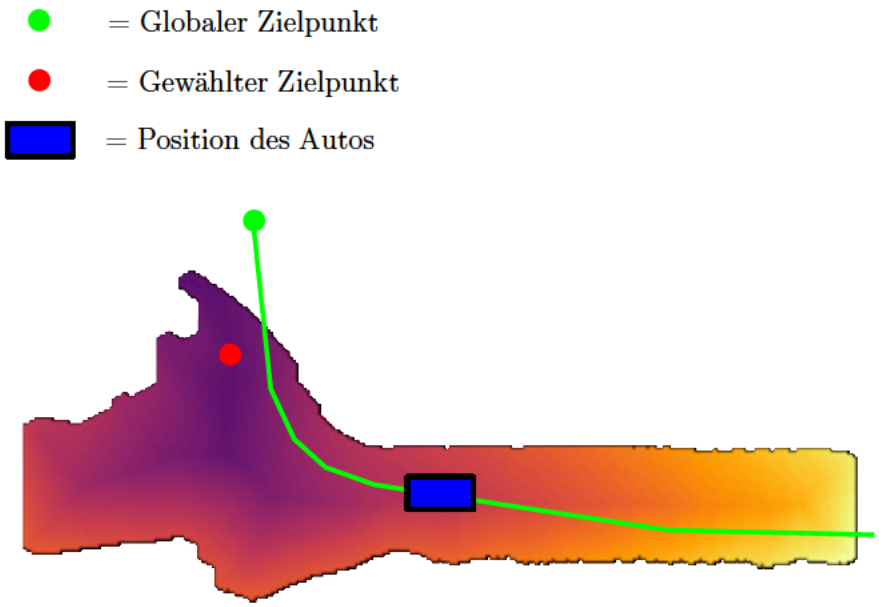


Abbildung 6.14: Visualisierung der errechneten Kosten.

6.3.1 Lokale Zielpunktwahl

Durch die Segmentierung kann nicht zwischen dem Straßenrand und dem nicht einsehbaren Teil der Straße unterschieden werden. Die lokale Pfadplanung kann keinen Pfad finden, der zu dem Ziel führt, da das Auto komplett von Hindernissen eingeschlossen ist.

Zusätzlich ist es nicht möglich von dem globalen Plan bei Ungenauigkeiten in der Straße abzuweichen, da der Planner stets den Wegpunkten des globalen Plans folgt. Deshalb ist eine Modfikation des Planners notwendig, um Wegpunkte angeben zu können, die sich innerhalb der erkannten Straße befinden und nah an dem globalen Plan liegen.

Das entwickelte System orientiert sich an der in [17] vorgestellten Zielpunktwahl.

Der globale Plan, der dem Local Planner übermittelt wird, besteht aus einer Vielzahl einzelner Posen. Zuerst muss die Pose des globalen Plans ermittelt, die am weitesten vom Auto entfernt ist, sich aber dennoch innerhalb der lokalen Costmap befindet.

Entscheidend für die Wahl eines geeigneten Zielpunktes ist ein geringer Abstand zu der ermittelten Pose und ein möglichst hoher Abstand zu dem Rand der Straße.

Dafür wird über alle in der Costmap freien Pixel iteriert und für jeden der Pixel Kosten aufgestellt.

Die Kosten werden durch die Funktion $J(x)$ bestimmt.

$$J(p) = d(p) \cdot W_d + r(p) \cdot W_r$$

$$r(p) = s_{max} - s_p$$

Dabei ist p der Punkt auf der Costmap und $d(p)$ der Abstand zwischen dem Punkt und der Vergleichspose des globalen Plan. Die Funktion $r(p)$ beschreibt den Abstand zum Straßenrand s vom Punkt p im Verhältnis zum maximalen Abstand zur Straße s_{max}. W_d und W_r sind einstellbare Faktoren für die Kosten.

In Abbildung 6.14 ist die daraus resultierende Karte der Kosten und die finale Zielposition dargestellt. Als Zielposition wird der Pixel ausgewählt, der die geringsten Kosten aufweist.

6.3.2 Timed-Elastic-Band

Der TEB Ansatz versucht die möglichst Zeit-optimierte Route entlang definierter Wegpunkte zu finden. Die Wegpunkte werden durch den globalen Plan vorgegeben. Um den Rechenaufwand einzugrenzen werden Einschränkungen und Vorgaben lediglich als Kosten berücksichtigt. Dadurch ist nicht garantiert, das Vorgaben eingehalten werden, egal wie hoch die Faktoren der jeweiligen Kosten sind.

Zu den Kosten zählen die Abstände zu Hindernissen, ein möglichst genaues Abfahren der Wegpunkte, die Begrenzungen von Beschleunigung und die Geschwindigkeit sowie die physikalischen Eigenschaften des Fahrzeuges. Da der TEB Ansatz häufig in einem lokalen Optimum hängen bleibt, wie zum Beispiel rechts von einem Hindernis, unterstützt TEB das Optimieren mehrerer Verfahrwege parallel. So kann etwa eine Route im linken und eine im rechten Bereich der Costmap untersucht und so die bessere ausgewählt werden.

TEB bietet ein breites Spektrum an Möglichkeiten. Durch über 110 Parameter ist TEB stark konfigurierbar und lässt sich an individuelle Anforderungen anpassen.

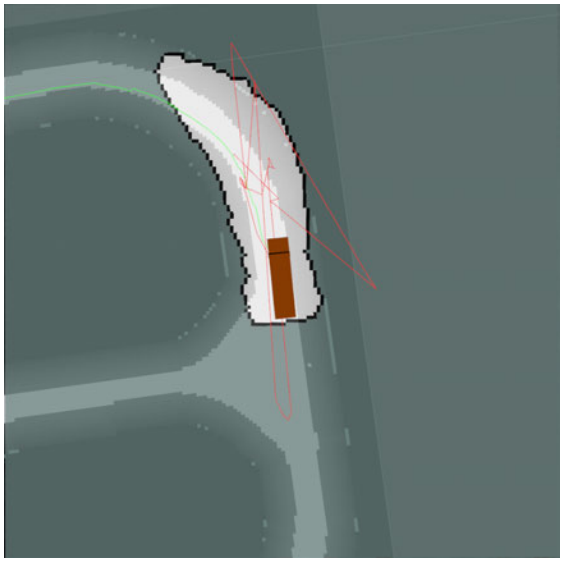


Abbildung 6.15: Bild der lokalen Costmap des TinyCars, mit dem durch TEB generierten Pfad. Der rote Pfad ist der lokale und der grüne Pfad der globale Plan.

Hindernisse in der Costmap werden für die Berechnung von Kollisionen in geometrische Formen umgewandelt und auch das Erkennen von dynamischen Hindernissen wird unterstützt.

Nachteil von TEB ist die hohe Komplexität, worauf bei der gering vorhandenen Rechenleistung auf dem TinyCar geachtet werden muss. Die schlechte Nachvollziehbarkeit der Planungsentscheidung ist ebenfalls ein Problem.

Anwendung von TEB

Während ein LiDAR Sensor Distanzwerte mit geringem Rauschen und hoher Auflösung generiert, ist die Erkennung der Straße stark fehleranfällig. Durch die Transformation in die Vogelperspektive kann ein Fehler von einigen Pixeln in der Segmentierung bereits einen Unterschied von mehreren Zentimetern verursachen.

Die schnellen Änderungen in dem freien Bereich auf der Costmap führen zu einem häufigen Umplanen der Route, was wiederum zu einem inkonsistenten Lenkwinkel führt. Besonderes auf engem Raum wird der Effekt dadurch verstärkt, dass TEB zum Rangieren Manöver mit Rückwärtsfahrten plant.

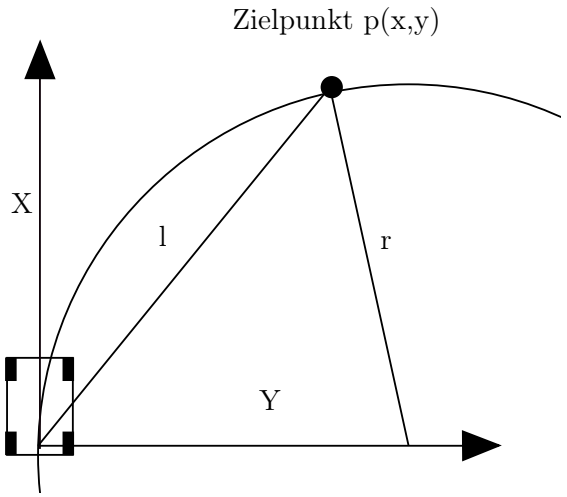


Abbildung 6.16: Geomtrie des Pure Pursuit Verfahrens.

Das Ausführen von TEB auf dem TinyCar führt allerdings zu dem Problem, dass permanent eine Kollision mit den Straßenrand erkannt wird. Wie in Abbildung 6.15 zu sehen ist, führt das Verhalten zu einem nicht nutzbaren Plan.

Es war nicht möglich den Ursprung des Problems zu ermitteln, allerdings ist eine Inkompatibilität mit dem reduzierten Maßstab nicht unwahrscheinlich. Weder ROS noch TEB wurden für den geringen Maßstab entwickelt. Die Hindernisgröße von 5 mm könnte Schwellwerte unterschreiten, welche von den Entwicklern TEBs als nicht plausibel eingeschätzt wurden.

Aufgrund dessen wurde ein weiterer Planner entworfen.

6.3.3 Pure Pursuit

Die Implementierung des Pure Pursuit Verfahrens [5] wurde als Local Planner eigens für diese Arbeit entwickelt. Denn wie die modifizierte Version von TEB, nutzt auch er der Pure Pursuit Planner die angepasste Zielpunktwahl, die in Abschnitt 6.3.1 beschrieben wurde.

Zwischen der ermittelten Zielposition und der aktuellen Position wird der benötigte Lenkeinschlag errechnet (6.16). Als Koordinatenursprung für den Zielpunkt p dient der *base_link* Frame.

Die Lookahead Distance (l) beschreibt die Entfernung von der Position des Autos zu der Zielposition. Eine hohe Lookahead Distance führt zu größeren Kurven mit geringen Oszillationen. Allerdings können Kurven bei starken Biegungen leicht geschnitten werden. Eine niedrige Lookahead Distance führt dazu, dass das Auto überlenkt und dem globalen Pfad mit Oszillationen folgt.

Die Geschwindigkeit v wird als Konstante behandelt, könnte aber je nach Lenkintensität angepasst werden.

Die Krümmung des Kreises, dem das Auto folgen soll, berechnet sich anhand:

$$\gamma = \frac{2y}{x^2 + y^2}$$

Da ein lokaler Planner als Ausgabe eine *Twist* Nachricht generieren soll, muss die Rotation als Winkelgeschwindigkeit um die Z-Achse angegeben werden.

Die Winkelgeschwindikeit berechnet sich durch:

$$\omega = \gamma \cdot v$$

7 Evaluation

In diesem Kapitel wurden Funktion und Wirksamkeit der in dieser Arbeit entwickelten Komponenten durch Tests überprüft.

7.1 Ergebnisse Rad-Encoder

Der Rad-Encoder ist sehr entscheidend für das Fahrverhalten des TinyCars, da nur durch ihn ein Feedback für den Fahrregler gewährleistet werden kann. Falsche Werte führen zu einer inkorrekten Fahrgeschwindigkeit und zu einer ungenauen Odometrie.

Um die Genauigkeit des Hall-Sensors und dessen Auswertung zu überprüfen, wurde ein Test mit einem optischen Drehgeber aufgebaut. Die Magneten der Hinterachse wurden an dem Drehgeber befestigt und dieser manuell über dem Hall-Sensor rotiert. Per I2C wurde der aktuelle Winkel des Drehgebers von einem externen Mikrocontroller an das TinyCar übertragen In der Software wurden gleichzeitig der aktuelle Winkel des Drehgebers und der durch den Hall-Sensor ermittelte Winkel aufgezeichnet.

Graph 7.1 zeigt den Unterschied zwischen dem realen Winkel des Drehgebers und den durch den Hall-Sensor gemessenen Winkel. Markante Änderungen des Winkels erscheinen bei beiden Verläufen während der gleichen Messungen.

Die Auswertung der Daten ergibt einen durchschnittlichen Fehler von etwa 6,9°.

Bei einem Reifenumfang von 1,2 cm entspricht das einem Fehler von 0,23 mm pro Umdrehung und 19,2 mm pro gefahrenem Meter.

Die Genauigkeit ist zwar verbesserungsfähig, aber ausreichend für das Einhalten der Geschwindigkeit und für die Ermittlung der Odometrie.

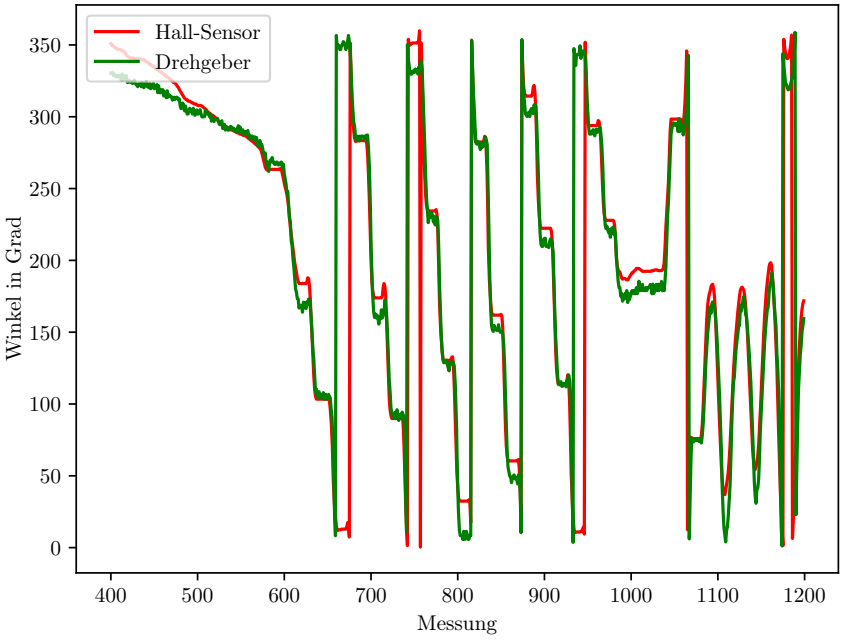


Abbildung 7.1: Vergleich zwischen dem realen Winkel und dem vom Hall-Sensor gemessenen Winkel.

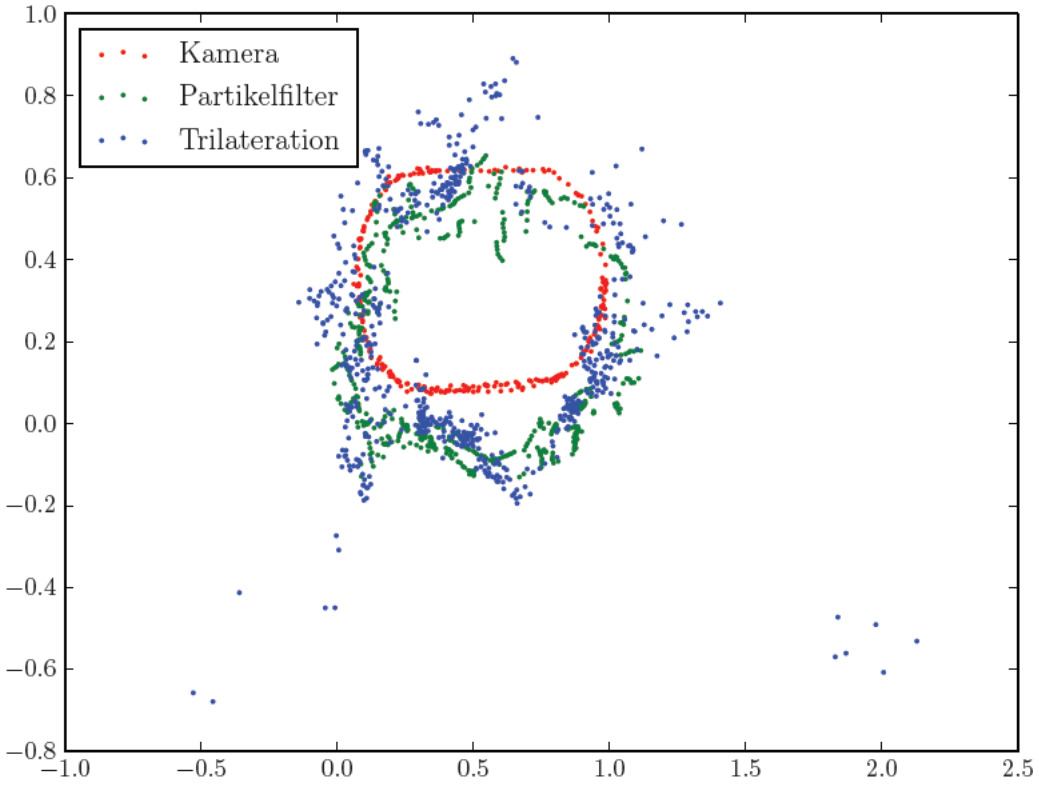


Abbildung 7.2: X/Y Graph Darstellung der Positionsbestimmung durch Kamera, Partikelfilter und Trilateration.

7.2 Ergebnisse Lokalisierung

Um die Lokalisierung zu testen wurden alle vorgestellten Lokalisierungsarten gleichzeitig aktiviert. Verglichen wurden die direkten Ausgabewerte ohne die Fusionierung durch den Extended-Kalman Filter.

Durch die hohe Genauigkeit der Kameralokalisierung können ihrer Werte als Groundtruth interpretiert werden.

In dem Graphen 7.2 sind die gesammelten Ergebnisse der Lokalisierung eingetragen. Beide UWB Verfahren weichen in Teilen stark und ähnlich von der Groundtruth ab. Das

bedeutet, dass für eine höhere Präzision die UWB Anchors besser kalibriert und/oder ihre Software verbessert werden muss.

Allerdings erzielen Partikelfilter und Trilateration auch abweichende Ergebnisse. Der durchschnittliche Fehler des Partikelfilters beträgt 28,94 cm im Vergleich zu einem Fehler von 15,4 cm des Trilateration Verfahrens.

Das Trilateration Verfahren ist im Durchschnitt zwar genauer, aber auch anfälliger für sprunghafte Änderungen sowie starke Abweichungen.

7.3 Fahrverhalten

Um das Fahrverhalten zu testen, wurden Fahrten im Carla Simulator und auf dem TinyCar durchgeführt. Auf der Karte wurde jeweils ein Ziel angegeben, welches das Auto erreichen soll.

Die Karten der globalen Costmap haben das gleiche Maß wie das Straßennetz im Simulator und in der echten Teststrecke. Daher kann als Referenzpfad der globale Plan verwendet werden.

7.3.1 Carla

Die Ergebnisse der Testfahrten sind in dem Graphen 7.3 dargestellt. Es wurde eine Testfahrt mit TEB als Local Planner und eine mit dem Pure Pursuit Planner durchgeführt.

Beide weisen ähnliche Ergebnisse auf. TEB hat eine durchschnittliche Abweichung von 2,0 m und der Pure Pursuit Ansatz eine durchschnittliche Abweichung von 3,1 m.

In dem TEB Pfad befindet sich eine starke Beule, die durch das späte Erkennen des Kreuzungsbereiches verursacht wurde und deshalb durch eine bessere Straßenerkennung nicht entstanden wäre. Aufgrund des überlegenen Funktionsumfangs und der höheren Genauigkeit von TEB, sollte TEB für Fahrten im Simulator genutzt werden.

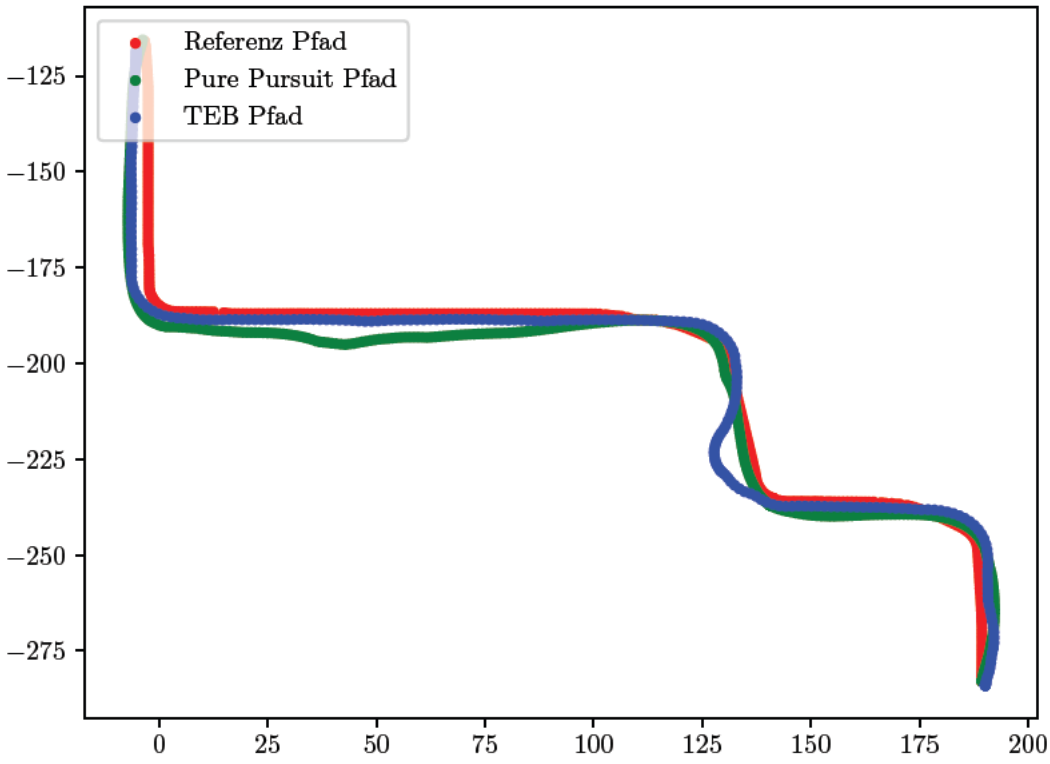


Abbildung 7.3: X/Y Graph Darstellung der im Carla Simulator gefahrenen Strecke mit dem Pure Pursuit und TEB Planner sowie der dazugehörige, ideale Referenzpfad.

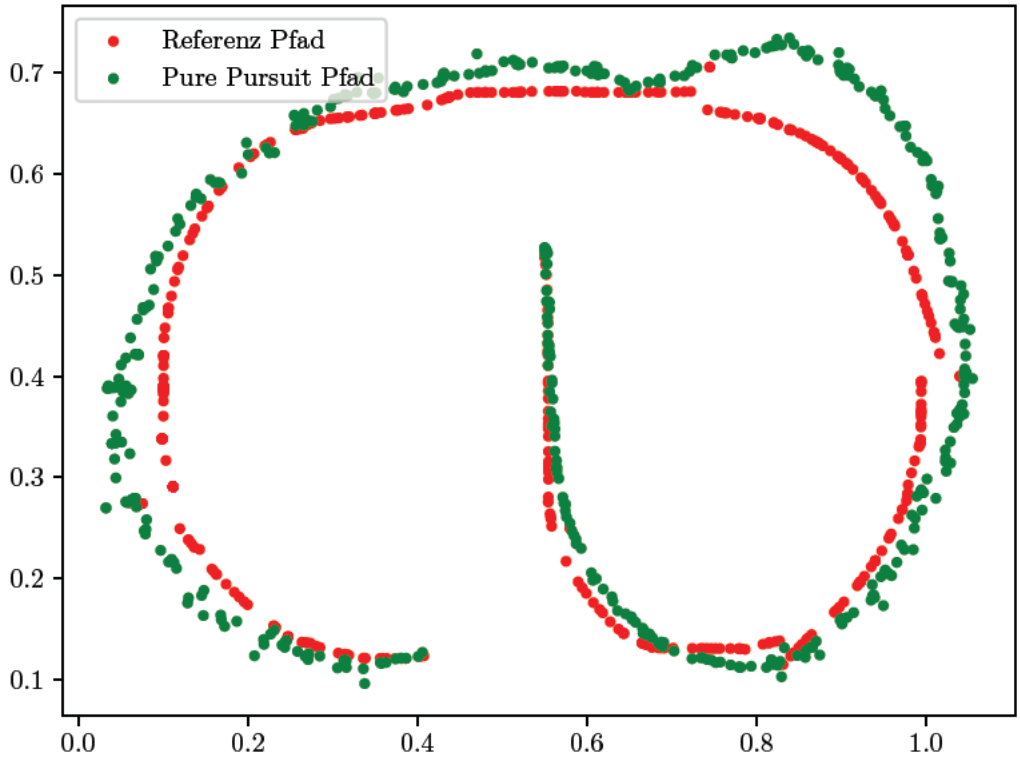


Abbildung 7.4: X/Y Graph Darstellung der auf dem TinyCar gefahrenen Strecke mit dem Pure Pursuit sowie der dazugehörige, ideale Referenzpfad.

7.3.2 TinyCar

Wie auch im Carla Simulator wurde über die Overhead-Kamera die tatsächlich gefahrene Strecke mit dem Referenzpfad verglichen. In Abbildung 7.4 sind beide Pfade dargestellt. Die Sprünge im Referenzpfad entstehen durch erneutes Setzen der Zielposition, was aufgrund der kleinen Strecke nötig ist.

Die durch Pure Pursuit gefahrene Strecke hat zu dem Referenzpfad eine durchschnittliche Abweichung von 0,029 Metern. In einem realen Maßstab (Faktor 87), entspricht das ungefähr 2.5 Metern. Das erzielte Ergebnis liegt somit nah an den Resultaten aus dem Simulator.

Es fiel jedoch auf, dass der durch Pure Pursuit ermittelte Lenkwinkel nicht genau zu dem durch das TinyCar eingestellten Lenkwinkel passt. Das ist auf Ungenauigkeiten in der Lenkung zurückzuführen, da ein lineares Ändern der Servoposition nicht zu einer linearen Änderung des Lenkwinkels führt.

8 Fazit

8.1 Fazit

In dieser Arbeit wurde ein Architektur entwickelt, die es in Zukunft erleichtern soll Miniaturfahrzeuge wie das TinyCar zu verwenden. Zuerst wurden alle Komponenten entwickelt, die nötig sind um die verbaute Hardware anzusprechen und auszulesen. Der nächste Schritt war es, die komplexeren Teilaufgaben zu realisieren, die sich die Daten der Sensoren zunutze machen.

Die Lokalisierung des TinyCars erfolgte mittels eines Extended-Kalman-Filters, welcher Werte der Funktechnologie UWB mit denen anderer Sensoren fusioniert. Zwei verschiedene Methoden zur Auswertung der UWB Distanzwerte wurden genauer untersucht. Das Verfahren der Trilateration schnitt jedoch aufgrund höherer Genauigkeit bei geringerem Rechenaufwand besser ab als der Partikelfilter.

Um das Ausführen eines Navigationszieles zu realisieren, wurde das ROS Package *move_base* mit in die Architektur einbezogen. Da die globale Karte nur eine Richtung vorgibt, darf sie nicht für die tatsächliche Ausführung des geplanten Pfades dienen. Um die Straße dynamisch zu erkennen, kommt deshalb ein BiSeNet V2 zum Einsatz. Durch eine Transformation in die Vogelperspektive lässt sich die Straße in eine Costmap eintragen, damit sie in die lokale Planung einbezogen werden kann.

Ziel war es, den Einstieg in die Arbeit mit dem TinyCar oder ähnlichen Miniaturfahrzeugen zu vereinfachen, indem eine Grundlage geschaffen wird, von der aus einzelne Komponenten ausgetauscht oder verbessert werden können. Die Evaluation hat gezeigt, dass es möglich ist das Auto autonom zu einer gewählten Zielposition fahren zu lassen. Somit ist bestätigt, dass die grundlegenden Funktionen des TinyCars funktionieren und nutzbar sind.

Durch ROS ist eine offene und flexible Plattform gewährleistet, die sich durch das Nutzen einzelner Nodes leicht erweitern lässt. Alle Kernkomponenten des Systems sind in einzelne Nodes unterteilt und lassen sich von anderen Nodes durch Nachrichten leicht steuern.

Zusammengefasst wurden die definierten Ziele erreicht, doch es bleibt Raum für Verbesserungen.

8.2 Ausblick

Im Folgenden werden mögliche Verbesserung für einige der Komponenten aufgeführt und näher erläutert.

8.2.1 Hardware

Steuerung

Die Lenkung des TinyCars sollte in einer weiteren Version überarbeitet werden, um das Spiel zu reduzieren. Zusätzlich muss die Übertragung des Servos auf die Lenkung verbessert werden, damit die eingestellte Servoposition linear zu dem realen Lenkwinkel verläuft oder eindeutig berechenbar ist.

Alternativ kann durch exaktes Messen des realen Lenkwinkels im Verhältnis zu der eingestellten Servoposition eine Kalibrierung generiert werden, die das ungleiche Übertragungsverhältnis ausgleicht. Das Spiel innerhalb der Lenkung führt dennoch zu Ungenauigkeiten des eingestellten Lenkwinkels.

8.2.2 Lokalisierung

Die Lokalisierung mittels UWB erreichte in den Testergebnissen nur eine durchschnittliche Genauigkeit von 15 cm und nicht die laut Hersteller möglichen 10 cm.

Ein besseres Ergebnis ist vermutlich durch Filtern der Distanzwerte und ein fortgeschritteneres System für die Kalibrierung möglich. Außerdem sollte die Firmware des STM32, der die Kommunikation mit dem DWM1000 durchführt, verbessert werden. Zusätzlich kann auch die Anzahl der verwendeten Anchors und ihre Position zu einem verbesserten Ergebnis führen.

Ein weiterer Aspekt, der verbessert werden sollte, ist das Tuning der Kovarianzen für den Kalman-Filter.

8.2.3 Straßenerkennung

Neuronale Netze befinden sich zur Zeit im schnellen Wandel mit vielen Fortschritten. Inzwischen wurde durch [9] eine für Echtzeit optimierte Version des BiSeNet V2 entwickelt, die eine höhere Genauigkeit und Geschwindigkeit bietet.

Außerdem ist die Straßenerkennung durch die Umwandlung in die Vogelperspektive stark begrenzt. Die Umrechnung ist fehleranfällig und für dreidimensionale Straßen nicht anwendbar.

Das Paper [15] präsentiert einen interessanten Ansatz eines Netzwerkes, das die Bildsegmentierung und die Berechnung der Vogelperspektive kombiniert. Zusätzlich werden durch das Netzwerk Annahmen für den nicht sichtbaren Bereich der Straße generiert.

Literaturverzeichnis

[1] AG, Infineon T.: Out of Shaft. (2018), 07

[2] Bosch (Veranst.): *Data sheet BNO055*. 2020. – URL https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bno055-ds000.pdf. – Rev. 1.7

[3] Bradski, G.: The OpenCV Library. In: *Dr. Dobb's Journal of Software Tools* (2000)

[4] Braun, Frederik ; Hertz, Stephan ; Braun, Gerrit: *Miniatur Wunderland Hamburg*. – URL https://www.miniatur-wunderland.de/

[5] Cordts, Marius ; Omran, Mohamed ; Ramos, Sebastian ; Rehfeld, Timo ; Enzweiler, Markus ; Benenson, Rodrigo ; Franke, Uwe ; Roth, Stefan ; Schiele, Bernt: The Cityscapes Dataset for Semantic Urban Scene Understanding. In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016

[6] Decawave (Veranst.): *DWM1000 Datasheet*. 2016. – URL https://www.decawave.com/sites/default/files/resources/DWM1000-Datasheet-V1.6.pdf. – Rev. 1.6

[7] Developers, TensorFlow: *TensorFlow*. August 2021. – URL https://doi.org/10.5281/zenodo.5189249

[8] Dosovitskiy, Alexey ; Ros, German ; Codevilla, Felipe ; Lopez, Antonio ; Koltun, Vladlen: CARLA: An Open Urban Driving Simulator. In: *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, S. 1–16

[9] Fan, Mingyuan ; Lai, Shenqi ; Huang, Junshi ; Wei, Xiaoming ; Chai, Zhenhua ; Luo, Junfeng ; Wei, Xiaolin: *Rethinking BiSeNet For Real-time Semantic Segmentation*. 2021

[10] Google LLC. (Veranst.): *Coral Accelerator Module datasheet.* 2020. – URL https://coral.ai/static/files/Coral-Accelerator-Module-datasheet.pdf. – Rev. 1.4

[11] Infineon (Veranst.): *3D Magnetic Sensor Datasheet.* 2019. – URL https://www.infineon.com/dgdl/Infineon-TLV493D-A1B6-DataSheet-v01_10-EN.pdf?fileId=5546d462525dbac40152a6b85c760e80. – Rev. 1.1

[12] JOAN2937: *The pigpio library.* https://github.com/joan2937/pigpio. 2012

[13] KASTEN, Markus: *Hardwareplattformen für autonome Straßenfahrzeuge im Maßstab 1:87.* 2021

[14] KATO, S. ; TAKEUCHI, E. ; ISHIGURO, Y. ; NINOMIYA, Y. ; TAKEDA, K. ; HAMADA, T.: An Open Approach to Autonomous Vehicles. In: *IEEE Micro* 35 (2015), Nr. 6, S. 60–68

[15] MANI, Kaustubh ; DAGA, Swapnil ; GARG, Shubhika ; SHANKAR, N. S. ; JATAVALLABHULA, Krishna M. ; KRISHNA, K. M.: *MonoLayout: Amodal scene layout from a single image.* 2020

[16] MOORE, T. ; STOUCH, D.: A Generalized Extended Kalman Filter Implementation for the Robot Operating System. In: *Proceedings of the 13th International Conference on Intelligent Autonomous Systems (IAS-13)*, Springer, July 2014

[17] ORT, Teddy ; JATAVALLABHULA, Krishna ; BANERJEE, Rohan ; GOTTIPATI, Sai ; BHATT, Dhaivat ; GILITSCHENSKI, Igor ; PAULL, Liam ; RUS, Daniela: MapLite: Autonomous Intersection Navigation Without a Detailed Prior Map. In: *IEEE Robotics and Automation Letters* 5 (2019), 12, S. 1–1

[18] QUIGLEY, Morgan ; CONLEY, Ken ; GERKEY, Brian ; FAUST, Josh ; FOOTE, Tully ; LEIBS, Jeremy ; WHEELER, Rob ; NG, Andrew: ROS: an open-source Robot Operating System, 01 2009

[19] Raspberry Pi Trading Ltd. (Veranst.): *Compute Module 4 datasheet.* 1 2021. – URL https://datasheets.raspberrypi.org/cm4/cm4-datasheet.pdf

[20] REKE, Michael ; PETER, Daniel ; SCHULTE-TIGGES, Joschua ; SCHIFFER, Stefan ; FERREIN, Alexander ; WALTER, Thomas ; MATHEIS, Dominik: A Self-Driving Car Architecture in ROS2, 01 2020, S. 1–6

[21] Yu, Changqian ; Gao, Changxin ; Wang, Jingbo ; Yu, Gang ; Shen, Chunhua ; Sang, Nong: *BiSeNet V2: Bilateral Network with Guided Aggregation for Real-time Semantic Segmentation*. 2020

Glossar

18650-Zelle Verbreitete Bauform eines Lithium-Ionen-Akkus mit einer typischen Kapazität von 0,8–3,5 Ah.

GNSS Ein globales Navigationssatellitensystem ist ein System zur Positionsbestimmung auf der Erde und in der Luft.

GPIO Ein GPIO ist ein digitaler Ein- oder Ausgang dessen Zustand per Software ausgelesen oder eingestellt werden kann.

Hall-Sensor Ein Sensor der den Hall-Effekt zum Messen von Magnetfeldern nutzt.

IMU Eine inertial measurement unit, ist ein Sensor um die Beschleunigung und Rotation zu messen.

LiDAR ist eine Technik um Distanzen zu messen. Ein Objekt wird per Laser bestrahlt und die Zeit gemessen, die das reflektierte Licht braucht um zu dem Sender zurückzukehren.

MQTT ist ein offenes Netzwerkprotokoll für Machine-to-Machine-Kommunikation.

Odometrie Eine Methode zur Abschätzung der Position und Orientierung anhand der Daten des Vortriebsystems.

PID Ein Übertragungselement der Regelungstechnik, das sich aus Anteilen eines P-Gliedes, eines I-Gliedes und eines D-Gliedes zusammensetzt.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

ROS-Architektur mit Autonomiefunktionen für ein Miniaturfahrzeug

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_____ _____ [redacted]

Ort Datum