

Don't code and drive

Abschlussarbeit Projektseminar Echtzeitsysteme

Proseminar eingereicht von

Josef Kinold, Dennis Kraus, Garwin Lechner, Blandine Riviere, Robin Scheich

am 8. April 2018



TECHNISCHE
UNIVERSITÄT
DARMSTADT



Fachgebiet Echtzeitsysteme

Elektrotechnik und
Informationstechnik (FB18)

Zweitmitglied Informatik (FB20)

Prof. Dr. rer. nat. A. Schürr
Merckstraße 25
64283 Darmstadt

www.es.tu-darmstadt.de

Gutachter: Geza Kulcsar

Betreuer: Geza Kulcsar

Inhaltsverzeichnis

| | | |
|-------|--|----|
| 1 | Einleitung | 1 |
| 2 | Trajektorienplanung | 2 |
| 2.1 | Modellbildung | 2 |
| 2.2 | PD Regler | 3 |
| 2.3 | Lokalisierung | 5 |
| 2.3.1 | Odometrie | 5 |
| 2.3.2 | AMCL | 6 |
| 2.4 | SLAM | 7 |
| 2.5 | Implementierung und Umsetzung | 7 |
| 2.5.1 | Navigation Stack | 7 |
| 2.5.2 | Base Controller | 12 |
| 2.6 | Ergebnisse und Probleme | 13 |
| 3 | Personenverfolgung | 14 |
| 3.1 | Modulbeschreibung | 14 |
| 3.1.1 | Detektion | 14 |
| 3.1.2 | Tracking | 16 |
| 3.1.3 | Clustering | 18 |
| 3.1.4 | Regler | 19 |
| 3.2 | Implementierung und Umsetzung | 20 |
| 3.3 | Ergebnisse und Probleme | 20 |
| 4 | Fazit und Ausblick | 22 |
| A | Anhang | 23 |
| A.1 | Launch und Config Dateien des Navigation Stack | 23 |



1 Einleitung

Im Projektseminar Echtzeitsysteme werden die Teilnehmer in Gruppen á 5 Leute aufgeteilt. Jede Gruppe bekommt ein Modellfahrzeug, welches mit Sensorik und Aktorik so ausgestattet ist, dass das Fahrzeug autonom fahren kann.

Das Ziel der Veranstaltung ist es, mithilfe des Autos drei verschiedene Aufgaben zu lösen. Die erste Aufgabe wurde dabei vom Veranstalter vorgegeben, die beiden anderen durften die Teams selbst auswählen.

Die vorgegebene Aufgabe war es, einen Rundkurs ohne Hindernisse entlang einer Wand absolvieren zu können. Wir haben dazu zwei verschiedene Lösungen entwickelt, die beide im Kapitel Trajektorienplanung beschrieben werden (siehe Kapitel 2).

Als zweite Aufgabe haben wir einen Rundkurs mit Hindernissen ausgewählt. Dieser sollte sich nur anhand der Hindernisse vom ersten Rundkurs unterscheiden, deshalb haben wir für beide Aufgaben den Navigation Stack verwendet (siehe Kapitel 2.5.1).

Unser drittes Ziel war es, eine Personenverfolgung zu implementieren (siehe Kapitel 3). Diese soll eine Person mithilfe des Kamerabildes detektieren und sie anschließend verfolgen, sobald sie sich bewegt.

Dieses Dokument beschreibt unsere Lösungen, unsere Probleme während der Umsetzung und unser Fazit. Zudem gibt es einen Ausblick auf mögliche Weiterentwicklungen für die jeweiligen Aufgabenstellungen.

2 Trajektorienplanung

Das Auto hat vorne sowie hinten jeweils zwei Räder, die durch einen Elektromotor ansteuerbar sind. Die Lenkung der Vorderachse ist durch einen Servomotor realisiert. Das Fahrzeug ist vorne und an den Seiten mit Ultraschallsensoren zur Abstandmessung ausgestattet und besitzt eine nach vorne ausgerichtete Kinect2 Kamera, die sowohl Farb- als auch Tiefenbild liefert. Für die Bestimmung der Orientierung des Fahrzeugs sind verschiedene Sensoren verbaut.

Um einen Rundkurs mit oder ohne Hindernissen zu bewältigen, muss man die gegebenen Werkzeuge nutzen und entsprechend kombinieren um das Ziel zu erreichen. Die beiden Ansätzen über einen PD-Abstandregler (vergleiche Abschnitt 2.2) und über eine Trajektorienplanung (vergleiche Abschnitt 2.5) werden im Folgenden vorgestellt.

2.1 Modellbildung

Um eine Beziehung zwischen physikalischer Größe für Geschwindigkeit und Lenkwinkel und der entsprechenden Stellgröße herzustellen, wird jeweils ein Modell benötigt.

Geschwindigkeitsmodell

Die Daten aus Abbildung 2.1 wurden bei einer Messfahrt mit Hilfe eines rosbag aufgezeichnet und im Anschluss mit dem Tool **plotjuggler** ausgewertet. Dazu wurde das Fahrzeug mit Geschwindigkeitsstellgrößen im Intervall von -500 bis 1000 angesteuert und die jeweilige Geschwindigkeit aus der Odometrie ausgelesen. Auffällig ist der Sprung am Achsenursprung, der dadurch entsteht, dass das Fahrzeug eine bestimmte Stellgröße benötigt, um seine Trägheit zu überwinden. Die gemessenen Daten lassen sich durch zwei Geraden approximieren:

$$f(x) = \begin{cases} 439.8038 \cdot x + 116.136943 & \text{für } x < 0 \\ 0 & \text{für } x = 0 \\ 477.518514 \cdot x - 104.623299 & \text{für } x > 0 \end{cases}$$

Lenkwinkelmodell

Die Bestimmung des Lenkwinkelmodells (Abbildung 2.2) erfolgte nach dem Schema:

1. Auto anheben und Stellgröße für den Lenkwinkel auf 0 setzen
2. Mit Geschwindigkeit 300 kurz geradeaus fahren und dann Lenkwinkel einstellen
3. Geschwindigkeit auf 0 setzen und den Lenkwinkel mit einem Geodreieck messen

Dazu war das Paket `rqt_reconfigure` sehr hilfreich. Die gemessenen Daten wurden im Anschluss durch das Polynom sechsten Grades

$$196814 \cdot x^6 - 50518 \cdot x^5 - 47550 \cdot x^4 + 5979.7 \cdot x^3 + 2459.5 \cdot x^2 - 2442.1 \cdot x + 143.78$$

approximiert. Auffällig ist die Verschiebung auf der x-Achse, die durch einen Offset in der Lenkung entsteht. Es sei angemerkt, dass die Messungen mit Fehlern behaftet waren und somit eine ungenaues Lenkwinkelmodell entstanden ist. Genauere Ergebnisse könnte durch Verwendung der Odometrie bei einer Fahrt im Kreis erzielt werden.

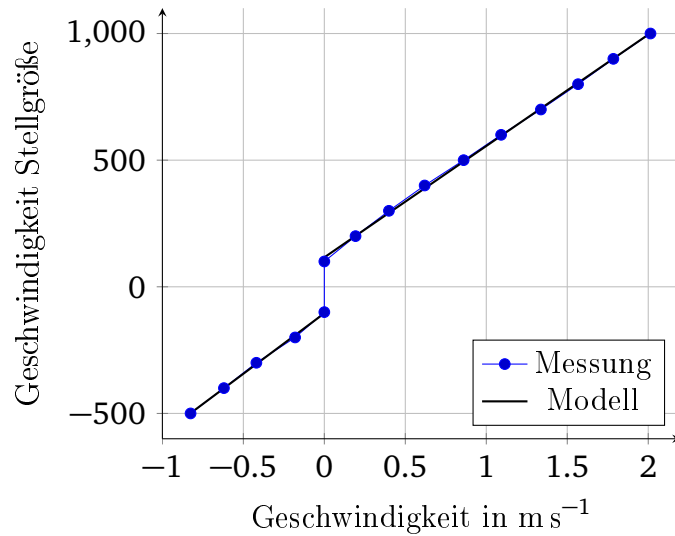


Abbildung 2.1: Messung und Geschwindigkeitsmodell.

2.2 PD Regler

Ein einfacher Ansatz für den Rundkurs ohne Hindernisse ist die Abstandregelung mit Hilfe der Ultraschallsensoren und einem PID Regler. Die zu regelnde Größe ist der Abstand zur Wand, als Stellgröße dient der Lenkwinkel. Dazu kann entweder die Stellgröße für den Lenkwinkel geregelt werden oder der Lenkwinkel selbst, welcher dann auf die Stellgröße aus dem Lenkwinkelmodell umgerechnet wird. Die Reglerauslegung ist sowohl per Simulation in Simulink als auch experimentell durchgeführt worden.

Die Simulation beruht auf dem Ackermann-Modell, einem Einspurmodell für Fahrzeuge. Es lautet

$$\dot{\varphi}_K = \frac{v}{l} \cdot \tan \varphi_L$$

$$\dot{y} = v \cdot \sin \varphi_K + v \cdot \frac{l_H}{l} \cdot \cos \varphi_K \cdot \tan \varphi_L.$$

Durch Linearisierung erhält man

$$\dot{\varphi}_K = \frac{v}{l} \cdot \varphi_L$$

$$\dot{y} = v \cdot \varphi_K + v \cdot \frac{l_H}{l} \cdot \varphi_L.$$

Dabei ist φ_K der Kurswinkel, φ_L der Lenkwinkel, v die Geschwindigkeit des Fahrzeugs, l der Achsabstand, l_H der Abstand von der Hinterachse zu einem Referenzpunkt und y der Abstand zur Wand. Mit beiden Systemen wurde in Simulink eine PD-Reglerauslegung durchgeführt. Es wurden diskrete Systeme betrachtet (vergleiche Abbildung 2.3), die Geschwindigkeit war $0.5 \frac{\text{m}}{\text{s}}$. Der simulierte Verlauf der Sprungantwort ist in Abbildung 4(a) dargestellt, der Stellgrößenverlauf in Abbildung 4(b).

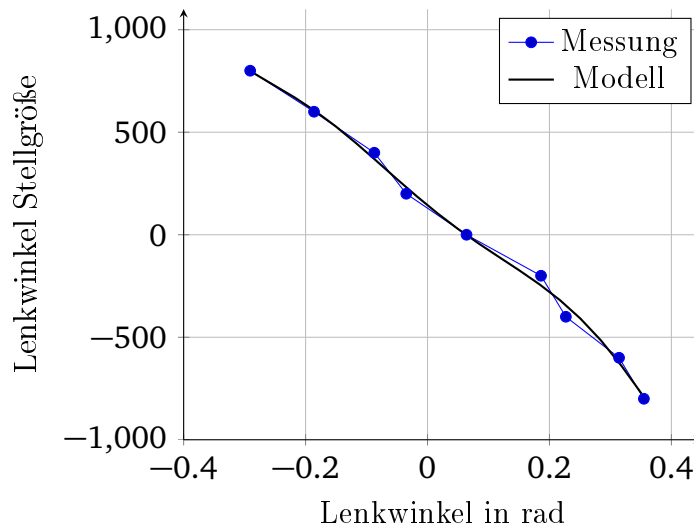


Abbildung 2.2: Messung und Lenkwinkelmodell.

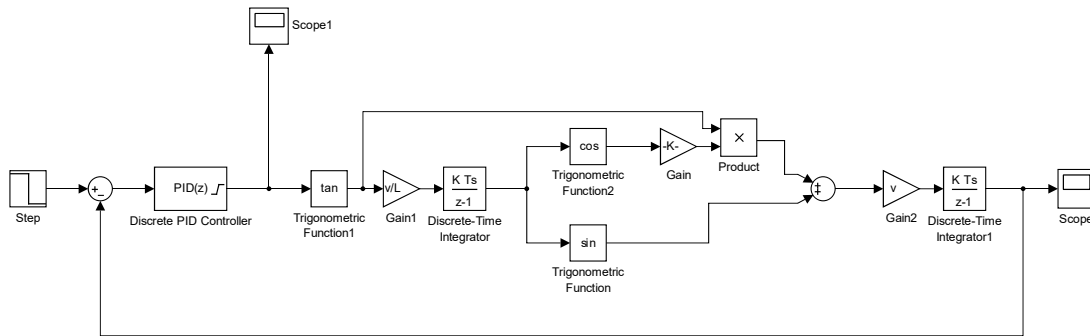


Abbildung 2.3: Diskreter Regelkreis mit nicht linearisierten Ackermann-Modell.

Mit den gefundenen Parametern konnten keine guten Ergebnisse erzielt werden, das Fahrzeug hat den gewünschten Abstand zur Wand nicht eingehalten und ist stark geschwungen. Das ist auf die Annäherung über das Ackermann-Modell und den betrachteten Abstand zu Wand zurückzuführen. Angenommen wird der Abstand senkrecht zur Wand, der in Wirklichkeit vom Fahrzeug gemessene ist bei schrägem Stand jedoch größer. Die Parameter wurden manuell modifiziert um eine weniger schwingende Fahrweise einzustellen (vergleiche Tabelle 2.1).

| | K_p | K_d |
|---------------|-------|-------|
| Simulink | 10 | 3 |
| Experimentell | 0.48 | 1.5 |

Tabelle 2.1: Werte für PD Regler aus Simulation und experimenteller Bestimmung.

Da die Ergebnisse dennoch nicht zufriedenstellend waren und die Lösung für den Rundkurs mit Hindernis nicht praktikabel ist, wurde dieser Ansatz verworfen und am Navigation Stack gearbeitet.

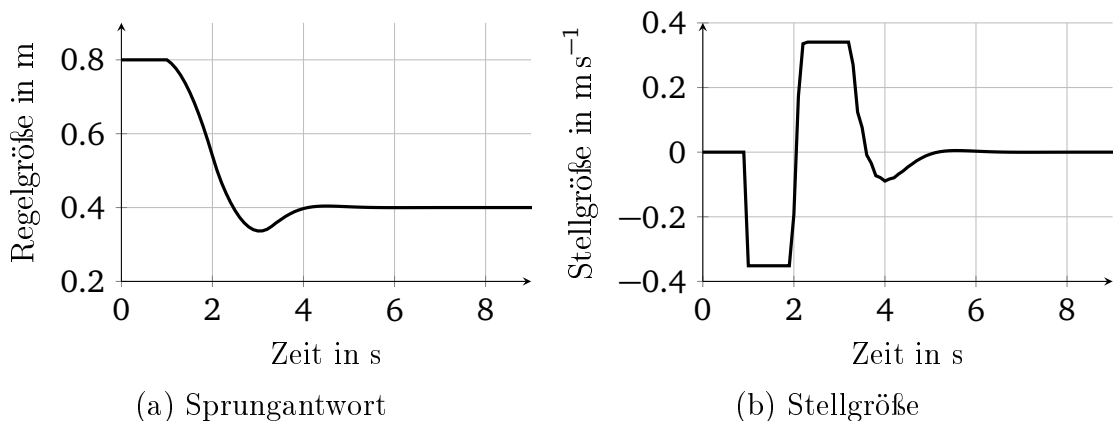


Abbildung 2.4: Simulierte Verläufe des Systems mit $K_p = 10$ und $K_d = 3$.

2.3 Lokalisierung

Die Trajektorienplanung des Roboters erfolgt anhand seiner Umgebungskarte, die mit den Sensordaten regelmäßig aktualisiert wird. Damit eine gute Trajektorie berechnet werden kann, muss das Auto in der Lage sein, sich zuverlässig in der Karte zu lokalisieren. Dies erfordert unter anderem eine verlässliche Odometrie, die im Abschnitt 2.3.1 vorgestellt wird. Um die Position des Autos in der Karte genauer zu schätzen, wurde außerdem das AMCL-Paket verwendet, wie im Abschnitt 2.3.2 beschrieben wird.

2.3.1 Odometrie

Auf dem Fahrzeug befinden sich ein Drei-Achs-Geschwindigkeitssensor und ein Drei-Achs-Gyroskop, welche etwa alle **0.5ms** neue Werte zur Verfügung stellen. Diese werden im vorhandenen Odometriepaket **pses_odometrie** genutzt um eine Position und die Ausrichtung des Fahrzeugs anzugeben. Durch das Rauschen der Sensoren unterscheidet sich die tatsächliche Position nach einiger Zeit von der angegebenen Fahrzeugposition. Die IMU (inertial measurement unit) wird im Odometriepaket zwar zu Beginn kalibriert, dies bedeutet jedoch nur, dass ein festgelegter, im Stand gemittelter Rauschwert berücksichtigt und von den IMU Daten subtrahiert wird.

Die Verwendung eines Extended Kalmanfilters (EKF) ermöglicht die Verwendung verschiedener Sensoren als Eingangsgrößen für die Schätzung der Fahrzeugposition und der Ausrichtung. Der EKF berechnet aus dem aktuellen Zustand anhand eines nicht-linearen Zustandsübergangsmodells eine Abschätzung für den nächsten Fahrzeugzustand. Durch die Verwendung mehrerer Sensoren kann die Genauigkeit und Zuverlässigkeit der Sensoren mithilfe einer Gewichtung der Eingangsgrößen des EKF berücksichtigt werden. Die Zustandsschätzung wird mit der nächsten Messung verglichen und die Gewichtung der Sensoren dynamisch angepasst.

Wir haben uns für das **robot_localization** Paket entschieden. Dies enthält einen EKF, das es ermöglicht die Sensoren modular einzubinden und sensorspezifisch zu entscheiden, welche Daten weiterverarbeitet werden sollen. Das Paket kann vier verschiedene Topic-Typen verarbeiten:

-
- `nav_msgs/Odometry` enthält Positions- und Ausrichtungsdaten
 - `geometry_msgs/PoseWithCovarianceStamped` enthält Positionsdaten
 - `geometry_msgs/TwistWithCovarianceStamped` enthält Ausrichtungsdaten
 - `sensor_msgs/IMU` enthält die Rohdaten aus der Inertialen Messeinheit (IMU)

Um die Schätzung zu vereinfachen wird angenommen, dass sich das Fahrzeug nur im zwei-dimensionalen Raum befindet und nur in X-Richtung fortbewegen kann. Z-Koordinate, Roll- und Pitch Winkel sind daher konstant Null. Die Beschleunigung in Y-Richtung wird als Eingangsgröße vernachlässigt.

Das `robot_localization` Paket ermöglicht es, die Kamera in Verbindung mit AMCL, dem Hall-Sensor und der IMU als Eingangsgrößen für das EKF zu verwenden. Aus Zeitgründen haben wir das EKF nicht fertig eingestellt, stattdessen verwenden wir für die Lokalisierung die Odometriedaten in Verbindung mit AMCL, wie im Abschnitt 2.3.2 beschrieben wird. Dies liefert uns eine ausreichend genaue Fahrzeugposition und Ausrichtung.

2.3.2 AMCL

Zur Lokalisierung des Autos wird das von ROS bereitgestellte AMCL-Paket verwendet. Dieses Paket implementiert den probabilistischen Adaptive Monte Carlo Localization - Algorithmus und ermöglicht dadurch die 2D-Lokalisierung von Robotern in einer vorgegebenen Karte. Die Daten aus dem Laserscan werden auf die Karte projiziert, woraus die wahrscheinlichste Position und Ausrichtung des Fahrzeugs bestimmt wird.

Zu diesem Zweck benötigt es mehrere Informationen, nämlich :

- einen Laserscan, der aus den Kameradaten erzeugt wird
- die Transformationen zwischen den verschiedenen Koordinatensystemen des Roboters
- die initiale Position des Roboters, die in unserem Fall manuell durch Rviz übergeben wird
- die Karte, in der der Roboter sich orten soll

Diese Topics werden gleichermaßen vom Navigation Stack benötigt und werden deswegen im entsprechenden Kapitel 2.5 näher beschrieben.

Allerdings ist darauf hinzuweisen, dass AMCL eine explizite Transformation vom Laserscan-Frame (hier, `base_laser`) zum Odometrie-Frame (hier, `odom`) erfordert.

Das AMCL-Paket stellt ebenso eine ganze Menge von einstellbaren Optimierungsparemtern bereit. Es ist zum Beispiel möglich das Odometriemodell (`omni` bzw. `diff` für Roboter mit omnidirektionalem bzw. Differentialantrieb) auszuwählen, sowie die entsprechenden Parameter einzustellen. Aus zeitlichen Gründen wurde jedoch eine im Navigation Stack bereits vorhandene Launchdatei für differentialgetriebene Fahrzeuge (`amcl_diff.launch`) verwendet.

2.4 SLAM

Die Abkürzung SLAM steht für Simultaneous Localization And Mapping und besteht für einen Roboter darin, gleichzeitig eine Karte seiner Umgebung zu erstellen bzw. zu verbessern und sich darin zu lokalisieren. In ROS stehen mehrere Pakete zur Verfügung, welche SLAM implementieren, nämlich **gmapping** und **hector_slam**. Letzteres hat den Vorteil, dass es keine Odometriedaten benötigt, und nur anhand des Laserscans die Karte erstellt. Um eine zuverlässige Karte aufzubauen sind allerdings folgende Punkte zu beachten. Erstens muss das Auto langsam und möglichst geradeaus (ohne Schwingungen) fahren, um zu vermeiden, Artefakten in die Karte einzubauen. Die Verwendung der vom Fachgebiet bereitgestellten Pakete **pses_dashboard** und **CarControl-App** kann sich beispielsweise zur Steuerung des Autos als nützlich erweisen. Falls **gmapping** benutzt wird, ist die Qualität der Karte und der Lokalisierung ebenso mit der Genauigkeit der Odometriedaten eng verbunden.

Das **hector_slam**-Paket wurde zum Aufbau einer Karte des Stocks des Fachgebiets verwendet. Da die Qualität der Karte jedoch nicht zufriedenstellend war, wurde zur Lokalisierung die von den Tutoren vorgelegte Karte verwendet.

2.5 Implementierung und Umsetzung

Die praktische Umsetzung besteht aus zwei Teilen. Zum einen dem Navigation Stack (siehe 2.5.1), der die aktuelle Position des Fahrzeugs mit den Umgebungsdaten verarbeitet und daraus eine Trajektorie plant. Zum anderen gibt es einen Knoten (siehe 2.5.2), der die geplante Geschwindigkeit und den Lenkwinkel aus dem Navigation Stack zu Stellgrößen umrechnet und am Fahrzeug setzt.

2.5.1 Navigation Stack

Der Navigation Stack ist ein Überbegriff für die Verbindung mehrerer Pakete. Aus den Odometrie- und Sensordaten wird eine Trajektorie zu einem vorher definierten Ziel geplant. Es werden Geschwindigkeits- und Lenkwinkeldaten ausgegeben, die den Roboter zu dem gegebenen Ziel führen.

Costmap

Für die Navigation wird eine Karte genutzt. Es wird unterschieden zwischen globaler und lokaler Costmap¹, zusätzlich liegen Informationen auf unterschiedlichen Schichten, sogenannten Layers.

- Static Map Layer: diese Schicht beinhaltet Kartendaten aus bereits gemappten Umgebungen, falls vorhanden. Sie sind Teil der globalen Costmap und bleiben während der gesamten Laufzeit bestehen.
- Obstacle Map Layer: diese Schicht enthält die Daten aus dem Laserscan, der mit dem Package **depthimage_to_laserscan** aus dem mit **pses_kinect_utilities** Median-gefilterten Tiefenbilder der Kinect2 Kamera erstellt wird. Hier werden also

¹ http://wiki.ros.org/costmap_2d

lokale Hindernisse gespeichert und bei Bedarf wieder entfernt. Diese Schicht ist Teil der lokalen Costmap.

- Inflation Layer: diese Schicht definiert einen Bereich um jedes erkannte Hindernis und überlagert diesen mit einer Kostenfunktion (nah am Objekt hohe Bestrafung, weiter entfernt weniger Bestrafung). Die wichtigen Parameter sind `inflation_radius` (1 m) und `cost_scaling_factor` (0.7 / 5). Diese Schicht ist ebenfalls Teil der lokalen Costmap.

Globaler Planer

Anhand der Position des Roboters, der Costmap und des Zieles wird eine einfache Trajektorie geplant. Dabei wird weder die Dynamik des Roboters betrachtet noch der Mindestabstand zu Hindernisse eingehalten. Es wurde mit dem `navfn`² Planer gearbeitet.

Lokaler Planer

Ausgehen von dem globalen Pfad wird ein lokaler Pfad geplant, der auf Hindernisse reagiert und die Ausmaße und Dynamik des Roboters bei der Planung mit in Betracht zieht. Es wurde sich für den `TEB local planner`³ entschieden, da er auch für nicht holonome Fahrzeuge gut geeignet ist. Die Qualität der geplanten Trajektorie und des realen Fahrverhaltens wird maßgeblich von der Konfiguration der Parameter des Planers beeinflusst. Parameter mit sehr großem Einfluss sind in Tabelle 2.2 zusammengefasst.

TF

Jeder veröffentlichte Topic wird mit einem Frame markiert. Um die Beziehung zwischen den dazugehörigen Koordinatensystemen herzustellen, kann das Package `TF`⁴ genutzt werden. Darin wird jeweils zwischen zwei Systemen eine Beziehung definiert und somit ein TF-Baum erstellt. Eine Transformation definiert sich durch

```
<node pkg="tf" type="static_transform_publisher" name="
  Select_a_name" args="x y z qx qy qz qw PARENT CHILD 20" />.
```

Der Baum setzt sich, wie in Tabelle 2.3 gezeigt, zusammen.

Debugging

Beim Einrichten des Navigation Stack treten hin und wieder Probleme und Fehler auf. Die Korrektheit des TF-Baum lässt sich mit `roslaunch rqt_tf_tree rqt_tf_tree` überprüfen. Die Informationen in einem Topic lassen sich mit `rostopic echo TOPIC` auslesen und Publisher sowie Subscriber mit `rostopic info TOPIC` ausgeben. Die Verbindungen zwischen laufenden Nodes wird mit `roslaunch rqt_graph rqt_graph` dargestellt. Sobald die grundlegende Struktur funktioniert, hilft das grafische Programm `RViz`⁵ weiter. Ob die Transformationen zwischen den Koordinatensystemen korrekt sind lässt sich

² <http://wiki.ros.org/navfn>

³ http://wiki.ros.org/teb_local_planner

⁴ http://wiki.ros.org/tf#static_transform_publisher

⁵ <http://wiki.ros.org/rviz>

| Parameter | Wert | Auswirkung |
|--------------------------------|----------|---|
| dt_ref | 0.5-0.75 | Besseres folgen der Trajektorie, weniger Überschwingen (sehr wichtig) |
| max_vel_x | 0.8 | Geschwindigkeitsbegrenzung vorwärts |
| max_vel_x_backwards | 0.5 | Geschwindigkeitsbegrenzung rückwärts |
| max_vel_y | 0.0 | Planer darf keine Geschwindigkeit in y-Richtung planen |
| acc_lim_y | 0.0 | Planer darf nicht in y-Richtung beschleunigen |
| max_vel_theta | 3.1 | Winkelgeschwindigkeit experimentell bestimmt für optimale Performance (gemessen ca. $1.1 \frac{\text{rad}}{\text{s}}$) |
| acc_lim_x | 2 | x-Beschleunigung experimentell bestimmt für optimale Performance (gemessen ca. $0.6 \frac{\text{m}}{\text{s}^2}$) |
| acc_lim_theta | 1.12 | Winkelbeschleunigung gemessen |
| cmd_angle_instead_rotvel | True | Planer gibt Lenkwinkel und x-Geschwindigkeit aus (für nicht holonome Roboter) |
| enable_homotopy_class_planning | False | Schnellere Planung, da nicht mehrere Möglichkeiten betrachtet werden |

Tabelle 2.2: Parameter mit hohem Einfluss auf die Performance des TEB local planner.

damit überprüfen. Außerdem lässt sich damit die globale Karte, Costmap und der Laserscan darstellen und auf Korrektheit prüfen. Zusätzlich ist die Position und der Footprint (digitale Repräsentation des Fahrzeugs) visualisierbar, welcher sich beim Fahren auch verschieben. Dabei sollte beachtet werden, dass sich die dargestellte Position des Fahrzeugs nur verschiebt, wenn der Motor angesteuert wird, beim Drehen des Hallsensors ohne Ansteuerung wird vom **pses_odometrie** Package keine Positionsänderung gepublisiert. Sobald die Karte und Position des Fahrzeugs korrekt wiedergegeben werden, ist es sinnvoll in RViz über den Button **2D Nav Goal** ein Ziel zu setzen und die geplante Trajektorie ausgeben zu lassen.

Es sei angemerkt, dass ein positiver Wert für die Stellgröße des Lenkwinkels einen Einschlag nach rechts bedeutet, der TEB local planner allerdings mit positiven Werten einen Lenkeinschlag nach links meint.

Hilfreiche Tools

Für Testfahren ist es hilfreich das Fahrzeug und einen Laptop mit dem selben WLAN-Netzwerk zu verbinden und per SSH auf das System des Autos zuzugreifen. Dazu wird die IP-Adresse des Fahrzeugs benötigt (**ifconfig**) und die Verbindung vom Laptop zum Fahrzeug geschieht dann über **ssh user@IP**. Dafür ist es ratsam einen WLAN-Hotspot, beispielsweise über ein Handy, einzurichten.

Mit dem Tool **dynamic_reconfigure**⁶ lassen sich Parameter während der Laufzeit

⁶ http://wiki.ros.org/dynamic_reconfigure

| Frame | Ursprung |
|--------------------|------------------|
| map | map_server |
| odom | pses_odometry |
| base_footprint | pses_odometry |
| base_link | Navigation Stack |
| base_laser | Navigation Stack |
| camera_depth_frame | LaserScan |

Tabelle 2.3: TF-Baum mit dem Ursprung oben.

verändern. Wurde der Knoten entsprechend konfiguriert, wird der Parameter auf der per SSH verbundenen Konsole mit **roslaunch dynamic_reconfigure dynparam set NODE PARAMETER VALUE** geändert. Somit kann während einer Testfahrt an der Performance des Fahrzeugs gearbeitet werden, ohne an den Arbeitsplatz zurückkehren zu müssen.

Eine weitere Möglichkeit zur Kontrolle über das Netzwerk bietet die Konfiguration verteilter Systeme⁷. Da ROS ohnehin über Netzwerkstandards kommuniziert, bietet es die Option Topics im Netzwerk zu verteilen. Das bedeutet, dass die Nodes auf dem Fahrzeug gestartet und die Topics auf einem Laptop, der im selben Netzwerk hängt, durch RViz visualisiert werden können. Dadurch kann während einer Testfahrt mit dem Navigation Stack die Karte und Trajektorie kontrolliert werden. Zudem ist es möglich Ziele zu setzen und die aktuelle Position des Fahrzeugs vorzugeben. Dies geschieht über den Button **2D Pose Estimate** und dient dazu den Roboter in der Karte zu positionieren. Die Konfiguration auf dem Fahrzeug, sowie auf dem Laptop folgt diesem Schema:

Auf Fahrzeug:

IP herausfinden (ifconfig)

In jedem Terminal, in dem ein Topic am Laptop braucht wird:

```
export ROS_IP=machine_ip_addr <- IP des Fahrzeugs
node/launch-file starten
```

Auf Laptop:

Terminal:

```
export ROS_MASTER_URI=http://172. . . :11311 <- IP Fahrzeug
IP herausfinden (ifconfig)
export ROS_IP=machine_ip_addr (Laptop IP)
rviz
```

Freier Rundkurs und Hindernisparcours

Die beiden Aufgaben Rundkurs ohne sowie mit Hindernissen sind vom Grundprinzip sehr ähnlich und werden beide mit dem Navigation Stack bewältigt. Die Planung des zu fahrenden Pfades bleibt gleich, nur einige Parameter für die Fahrt machen einen Unterschied für die beiden Aufgaben. Die Dateien zum Starten und Konfigurieren des Navigation Stack befinden sich in Anhang A.1. Die sich für den freien Rundkurs und Hinder-

⁷ <http://wiki.ros.org/ROS/Tutorials/MultipleMachines>

nisparscours unterscheidenden Parameter sind jeweils mit dem Kommentar **for obstacle** **parcour** vermerkt.

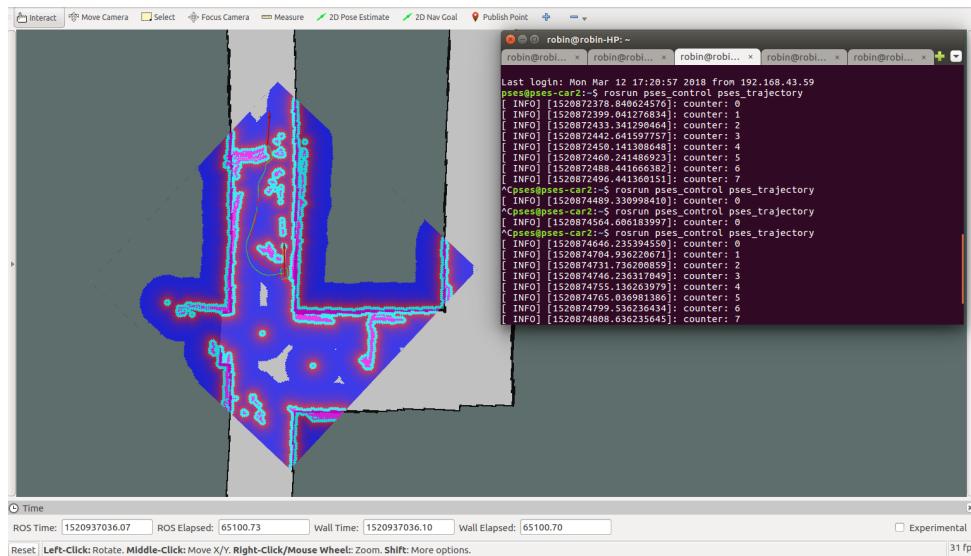


Abbildung 2.5: Ausschnitt aus rviz mit der Costmap, dem Footprint des Fahrzeugs und der Zielführung sowie das Terminal, in dem der Fahrknoten gestartet ist.

Probleme

Obwohl der Navigation Stack und hauptsächlich der TEB local planner insgesamt gut eingestellt sind, gibt es einige Probleme, die es in Zukunft zu lösen gilt.

- **Artefakte Laserscan:** Das Package **depthimage_to_laserscan** liest eine Zeile aus dem Tiefenbild, welches mit Rauschen behaftet ist. Dadurch entstehen Artefakt-Punkte auf der Costmap, die der Planer als Hindernis erkennt, die allerdings nicht real sind. Obwohl das Tiefenbild bereits durch einen Medianfilter (Filtergröße 5) bearbeitet wird, tritt dieser Effekt immer noch auf. Ein Lösungsansatz wäre die Größe des Filters zu erhöhen, allerdings ist der verwendete Filter aus dem Package **pscs_kinect_utilities** mit Größe 5 begrenzt. Ein weiterer Lösungsansatz wäre den Laserscan aus mehreren Zeilen des Tiefenbildes auszulesen und von jeder Spalte den maximalen Wert herauszuschreiben.
- **Recovery:** Da die lokale Costmap nicht nur die aktuellen Daten aus dem Laserscan beinhaltet, sondern auch mit den vorangegangenen Daten eine Karte zusammenbaut, können Artefakte mit in die Karte eingetragen werden. Passiert das häufiger an einer Stelle, kann es den Weg für den Trajektorienplaner versperren. Um diese falsch detektierten Punkte wieder loszuwerden, gibt es den Parameter **recovery_behaviors**, der verschiedene Szenarien zum Leeren des Costmap beinhaltet. Es gibt einige Parameter⁸, die den Zeitpunkt des Ausführens des Recovery Programms beeinflussen, diese sollten eingestellt werden. Es sei angemerkt, dass

⁸ http://wiki.ros.org/move_base#Parameters

der Parameter `clearing_rotation_allowed` auf false gesetzt sein sollte, da das Fahrzeug eine solche Bewegung nicht vollziehen kann.

- Rückwärts fahren: Sobald das Fahrzeug nah an einem Hindernis fährt, kommt es in einen Bereich in dem der Wert der Straffunktion hoch ist. Das bedeutet, dass das Fahrzeug sich in diesem Bereich nur langsam bewegen darf. Es hat sich gezeigt, dass in diesem Fall der TEB local planner nur noch vereinzelt Geschwindigkeitswerte ausgegeben hat und im Hindernis stecken geblieben ist.

Der Navigation Stack sollte entsprechend eingestellt werden, dass dieser in solchen Situationen rückwärts fährt und sich somit Platz verschafft, um am Hindernis vorbei zu fahren.

- AMCL Fehldetektion: AMCL dient dazu sich besser in einer Karte zu orientieren, indem es den Laserscan auf die Karte projiziert und damit die aktuelle Position bestimmt. Während der Testfahrten kam es gelegentlich vor, dass AMCL in einer Kurve die hintere Wand auf die vordere Wand gemapt hat und somit den Weg verschlossen hat.

Das Verhalten ist abhängig von der Qualität der Fahrt und sollte untersucht werden. AMCL⁹ bringt selbst Parameter mit, die zum Verbessern dieses Verhaltens beitragen könnten.

2.5.2 Base Controller

Das Fahrzeug wird tatsächlich von einem Knoten gesteuert, dessen Aufgabe darin besteht, die Stellgrößen für den Motor und die Lenkung zu berechnen und die verschiedenen Ziele für den Rundkurs zu setzen.

Modelle

Da die Lenkwinkel- und Geschwindigkeitsbefehle aus dem Navigation Stack in `m/s` und in `rad` gepublished werden, müssen sie zu Stellgrößen umgerechnet werden. Hierfür werden die im Abschnitt 2.1 beschriebenen Geschwindigkeits- und Lenkwinkelmodelle genutzt. Diese Stellgrößen werden dann als Topics für die `uc_bridge` gepublished.

Zielsetzung

Damit eine Trajektorie geplant wird, muss dem Navigation Stack ein Ziel übergeben werden. Es bestehen mehrere Möglichkeiten dafür. Entweder gibt man ein Ziel manuell in rviz vor, oder man publisht ein Ziel direkt in dem Topic `/move_base_simple/goal`.

Um den Rundkurs zu fahren, wurden mehrere Ziele entlang der Strecke gesetzt, wie in Abbildung 2.6 dargestellt. Mithilfe eines Zählers wird jedes Ziel automatisch aktiviert, sobald das Fahrzeug sich ausreichend dem vorherigen Ziel genähert hat, d.h. in einer Entfernung kleiner als 1,5m.

⁹ <http://wiki.ros.org/amcl>

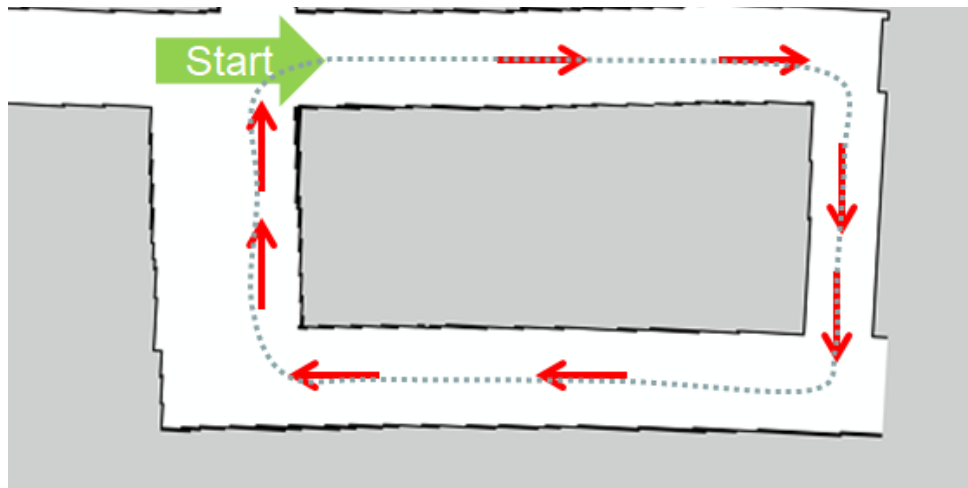


Abbildung 2.6: Aufeinanderfolgenden Ziele entlang des Parcours

Recovery Behavior

Wie bereits im Kapitel 2.5.1 (siehe Probleme) erwähnt, ist es möglich, dass der lokale Planer aufgrund der Präsenz von Artefakten in der Costmap keine vernünftige Trajektorie berechnen kann. Das Auto fährt dann abwechselnd rück- und vorwärts, ohne einen Ausweg zu finden. Um dieses Problem zu beheben, wurde ein eigenes Recovery Behavior implementiert, welches dieses problematische Verhalten erkennen und das Obstacle Layer in der Costmap leeren soll.

Diese Implementierung hat jedoch nicht funktioniert, da das richtige Layer beim Leeren der Costmap nicht gefunden werden konnte. Aus Zeitmangel konnte dieser Fehler nicht behoben werden.

2.6 Ergebnisse und Probleme

Wie im Kapitel 2.3.1 bereits erwähnt, wurde das Extended Kalmanfilter nicht fertig eingestellt. Das implementierte EKF benutzt die IMU-Daten und die Geschwindigkeit, welche durch den Hall-Sensor berechnet wird. Obwohl die Y-Geschwindigkeit des Fahrzeugs nicht als Eingabegröße für das EKF verwendet wird und nur die Steuerung der Geschwindigkeit in X-Richtung und des Yaw Winkels zugelassen sind, bewegt sich das Auto in den gefilterten Odometriedaten des EKF seitwärts. Das Einstellen der maximalen Be- und Entschleunigungswerte des Fahrzeugs hat zwar zur Verbesserung der Abschätzung beigetragen, das Problem der seitlichen Bewegung jedoch nicht behoben.

3 Personenverfolgung

In diesem Kapitel wird die Personenverfolgung erläutert. Dabei folgt das Fahrzeug einer bestimmten Person in einem gewissen Abstand. Zuerst wird auf die benötigten Module eingegangen, bevor diese zur Gesamtstruktur zusammengesetzt und ihre Beziehungen zueinander erläutert werden. Abschließend erfolgt eine Diskussion der erzielten Ergebnisse und Probleme.

3.1 Modulbeschreibung

Der Algorithmus teilt sich in die Module Detektion, Tracking, Clustering und Regler auf. Diese werden im Folgenden erläutert.

3.1.1 Detektion

Als grundlegender Schritt der Personenverfolgung muss eine Person in dem Kamerabild gefunden werden. Dieser Vorgang nennt sich (Personen-)Detektion. Herkömmliche Histogram-of-Oriented-Gradients Ansätze liefern keine zufriedenstellende Lösung bezüglich des Autos. Daher fiel die Wahl auf die Nutzung eines Convolutional Neural Networks kurz CNN. Diese Netze eignen sich sehr gut zur Klassifizierung von Objekten in Bildern, da sie die Zugehörigkeit eines Objektes zu einer Klasse mit Wahrscheinlichkeiten ausdrücken. Ein typisches CNN besteht aus drei Layern. Das erste ist das Convolutional Layer. Dazu wird das Kamerabild mit einem Kernel bzw. einer Faltungsmatrix gefaltet. Die Werte des Kernels sind dabei auf das jeweilige Problem angepasst. Es wurde somit eine sogenannte Feature-Map erzeugt. Werden nun die Ergebnisse der Faltung mit einer Aktivierungsfunktion verknüpft, bildet dies den Eingang der einzelnen Neuronen. Bei CNNs ist die Nutzung einer Rectified Linear Unit mit $f(x) = \max(0, x)$ mit x als Ergebnis der Faltung üblich. Die zweite Schicht bildet das Pooling Layer. Hier findet eine Ordnungsreduktion statt, indem überflüssige Informationen verworfen werden. Das geschieht beispielsweise mit einem Max-Pooling. Dabei wandert ein 2×2 -Quadrat über die Neuronen des Convolutional Layers und behält nur die Aktivität des aktivsten Neurons. Der Vorteil dieses Layers ist unter anderem eine erhöhte Berechnungsgeschwindigkeit. Den Abschluss bildet das Fully-connected Layer. Hier werden die Eingänge des Layers Objektklassen zugeteilt.

Als Neuronales Netz zur Bewältigung der ursprünglichen Aufgabe wird das Google MobileNet SSD[?] als Caffe Framework Modell verwendet, welches bereits vortrainiert ist. Ein weiterer Vorteil dieses Netzes ist die Optimierung auf ressourcenschonende Berechnungen, allerdings auf Kosten der Genauigkeit. Dieser einhergehende Nachteil wird im vorliegenden Anwendungsfall jedoch nicht als relevant erkannt. Diese Optimierung zieht eine Änderung der gewöhnlichen Struktur eines CNNs mit sich wie in Abbildung 3.1 dargestellt. Die grundlegende Idee hinter Depthwise Separable Convolution ist die Aufteilung der Faltung eines CNNs in eine 3×3 Depthwise Convolution gefolgt von einer 1×1 Pointwise Convolution. Dabei dient ersteres dazu, jeden Eingangskanal zu filtern und letzteres dazu, diese Ausgänge wieder zusammenzuführen. Der Vorteil davon ist die drastische Reduzierung des Berechnungsaufwands und der Modellgröße. Batch Normali-

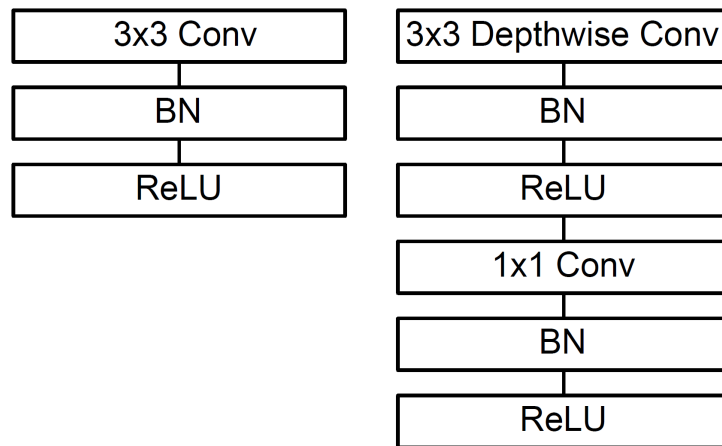


Abbildung 3.1: Links: Standard Convolutional Layer mit Batch Normalization und ReLU. Rechts: Depthwise Separable Convolutions mit Depthwise und Pointwise Layers gefolgt von Batch Normalization und ReLU. (aus [?])

zation¹⁰ bezeichnet dabei die Normalisierung der Werte der unsichtbaren Schichten des Netzes, um die Trainingsgeschwindigkeit deutlich zu erhöhen.

Benutzt wird das Netz mithilfe des seit OpenCV 3.3 existierenden Deep Neural Networks (dnn) Moduls. Dieses kann unter anderem Caffe Framework Modelle laden, benutzen und trainieren. Letztere Funktion wird nicht benutzt, da das Netz bereits ausreichend trainiert ist. Daher muss es lediglich einmalig zum Programmstart geladen werden. Der aktuelle Kameraframe des Farbbildes wird auf eine Größe von 300 px×300 px skaliert, da das Netz ein Eingangsbild dieser Größe erwartet. Anschließend wird der Frame vorverarbeitet. Einerseits wird mit einer so genannten Mean Subtraction vom Frame der Durchschnitt des jeweiligen Farbkanals subtrahiert, um Helligkeitsunterschieden entgegen zu wirken. Dabei wird der Durchschnitt hier auf allen Farbkanälen zu $\mu = 127,5$ gewählt. Andererseits kann diese Subtraktion mithilfe eines Skalierungsparameters σ normalisiert werden. Hier wird $\sigma = 127,502231$ gewählt. Zu beachten ist, dass der übergebene Skalierungsfaktor der blobFromImage-Funktion $1/\sigma$ entspricht. Jetzt kann der eben erstellte Blob dem Netz als Eingang übergeben werden. Die Ausgabe beinhaltet alle erkannten Objekte als Matrix zusammengefasst. Eine Zeile dieser Matrix besteht aus den Einträgen $d^T = [0, \text{ID der zugehörigen Klasse, Wahrscheinlichkeit, } x, y, \text{Breite, Höhe}]$. Dabei liegen die Werte der x - und y -Koordinate im Bereich $[0, 1]$ und müssen, um die eigentlichen Koordinaten zu erhalten, erst mit der Bildbreite bzw. Bildhöhe multipliziert werden. Da unter Umständen mehrere Personen im Bild detektiert werden können, wird diejenige ausgewählt, die die größte Klassenwahrscheinlichkeit besitzt. Eine Alternative wäre, zusätzlich die Wahl der Personen auf einen Bereich um den Bildmittelpunkt zu begrenzen. Nun ist die gewünschte Person detektiert und es kann mithilfe der angesprochenen Koordinatentransformation eine Bounding Box der Person für die nachfolgenden Module erstellt werden.

¹⁰ <https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c>

3.1.2 Tracking

Da die Detektion einen gewissen Rechenaufwand und somit Rechenzeit benötigt, kann sie aufgrund den begrenzten Hardwareressourcen nicht regelmäßig durchgeführt werden. Aus diesem Grund wird in der zweiten Phase der Personenverfolgung Tracking genutzt.

Tracking ist ein Vorgehen, um Objekte, in unserem Fall eine Person, in einer Folge von Bildern zu verfolgen. Der Vorteil von Tracking ist die rechensparsame Berechnung der nächsten Position eines Objekts, die unter gewissen Bedingungen sogar robuster als eine Detektion sein kann. Jedoch muss Tracking zu Beginn initialisiert werden, das durch die Bounding-Box der Detektion bereits gegeben ist. Zur Umsetzung des Trackings existieren verschiedene Ansätze, die einzelne Pixel, einen ganzen Bildteil oder Bewegungen im Bild nutzen.

Eine Auswahl an fünf Trackern wurden bereits in der OpenCV-Tracking API implementiert, sodass mehrere Tracker für die gegebene Anwendung getestet werden konnten. Die Tracker verwenden jeweils die interne Representation einer Bounding-Box und lernen abgesehen vom Median Flow einen Klassifizierer mithilfe von Online Learning¹¹. Als geeignetester Tracker stellte sich bei der Versuchsreihe der Median Flow heraus. Er bietet den besten Tradeoff zwischen Rechenzeit und Robustheit bei der Personenverfolgung. Die relativ langsamen und vorhersagbare Bewegung einer Person eignen sich gut für den gewählten Tracker.

Der Median Flow besteht aus zwei Hauptkomponenten. Im ersten Schritt wird die Bewegung ausgewählter Pixel mithilfe des Lucas-Kanade-Trackers berechnet und anschließend wird die berechnete Position durch den Forward/Backward-Error evaluiert. Die beiden Komponenten werden im Folgenden nochmal genauer beschrieben:

Lucas-Kanade-Tracker

Der Lucas-Kanade-Tracker basiert auf dem Prinzip des Optischen Flusses. So kann in einer Bildfolge die neue Position eines Pixels mit folgendem Zusammenhang ausgedrückt werden:

$$I(x, y, t) = I(x + dx, y + dy, t + dt) \quad (1)$$

Hierbei beschreibt $I(x, y, t)$ die Intensität eines Pixels an der Position (x, y) zum Zeitpunkt t . Durch die Linearisierung mithilfe Taylorreihenentwicklung ergibt sich die Formel des Optischen Flusses zu

$$I_x u + I_y v + I_t = 0 \quad (2)$$

mit

$$u = \frac{dx}{dt}, \quad v = \frac{dy}{dt}, \quad I_x = \frac{dI}{dx}, \quad I_y = \frac{dI}{dy}, \quad I_t = \frac{dI}{dt}. \quad (3)$$

¹¹ <https://www.learnopencv.com/object-tracking-using-opencv-cpp-python/>

Nun besteht die Aufgabe die Bewegung (u, v) zu bestimmen. Dies wird mithilfe der Erweiterung der Gleichung unter Einbeziehung der benachbarten Pixel in einem 3×3 Fenster ermöglicht:

$$\begin{pmatrix} I_x(p_1) & I_y(p_1) \\ \dots & \dots \\ I_x(p_9) & I_y(p_9) \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} + \begin{pmatrix} I_t(p_1) \\ \dots \\ I_t(p_9) \end{pmatrix} = 0 \quad (4)$$

Die überbestimmte Formel beinhaltet die 9 Pixel $p_1 \dots p_9$ aus dem 3×3 Fenster und kann mithilfe der Methode der kleinsten Quadrate gelöst. Als Ergebnis erhält man die Bewegung eines Pixels, die die Verfolgung $\tilde{A}_{\frac{1}{4}}$ über eine Folge von Bildern ermöglicht.

Forward/Backward Error

Mithilfe des Lucas-Kanade-Trackers wird ein Pixels über eine Bilderfolge von k Bildern verfolgt und so eine Trajektorie $T_f^k = (x_t, x_{t+1}, \dots, x_{t+k})$ bestimmt. Hierbei steht f für *forward* und beschreibt die Forwärtstrajektorie. Das Ziel ist nun diese Trajektorie zu validieren. Hierfür beginnen wir bei dem x_{t+k} und verfolgen diesen Pixel rückwärts über die gegebene Folge von k Bildern. So wird eine zweite Trajektorie $T_b^k = (\hat{x}_t, \hat{x}_{t+1}, \dots, \hat{x}_{t+k})$ bestimmt, die mit der ersten verglichen wird. Um die Berechnung simpel zu halten, wird die euklidische Distanz zwischen Start und Endpunkt der jeweiligen Trajektorie gewählt:

$$distance(T_f^k, T_b^k) = \|x_t - \hat{x}_t\| \quad (5)$$

In Abbildung 3.2 wird das Vorgehen zur Bestimmung des Forward/Backward Errors nochmals veranschaulicht.

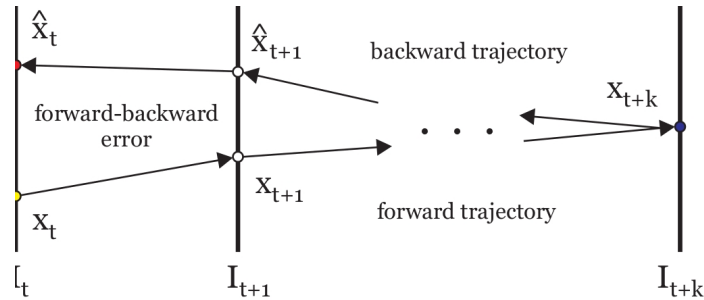


Abbildung 3.2: Das Vorgehen zur Berechnung des Forward/Backward Errors. (Quelle)

Die beschriebene Methode zur Validierung wird in der Implementierung des Median Flow zwei mal angewendet. Zu Beginn wird sie zur Auswahl der Featurpunkte aus dem Detektionfenster, die mit dem Ansatz von Lucas und Kanade getrackt werden, genutzt. So werden die Pixel, die einen hohen Forward/Backward Error aufweisen, verworfen und die signifikanten für Tracking geeignete Feature behalten. Weiterhin wird der Fehler zur Validierung des gesamten Trackingergebnisses genutzt und kann so ein fehlerhaftes Tracking erkennen und abrechnen.

3.1.3 Clustering

Der letzte Schritt der Bildverarbeitung besteht aus dem Clustering. Das Ziel ist die Bestimmung der Tiefe der Person aus der getrackten Bounding-Box mit möglichst simplen und somit rechensparsamen Ansätzen. Um überhaupt Tiefeninformationen zu erhalten, wird ab diesem Schritt das Tiefenbild der Kinect-Kamera verwendet. Im ersten Schritt wird die Bounding-Box nach bestimmten Kriterien überprüft, um den Voraussetzungen des an das Farbbild angepasste Tiefenbild zu genügen. Die Voraussetzungen ergeben sich aus dem schwarzen Bereichen an den Rändern des Tiefenbildes, die durch dessen Verzerrung entstehen. Weiterhin werden invalide Tiefenbestimmungen aus dem Tiefenbild gefiltert, um ausschließlich brauchbare Werte in den nächsten Verarbeitungsschritten zu berücksichtigen.

Aus den Tiefenpixel, die sich innerhalb der Bounding-Box befinden, wird nun der Mittelwert gebildet und anschließend eine Maske erstellt, die ausschließlich Pixel unterhalb des Mittelwertes berücksichtigt:

$$Mask(x, y) = \begin{cases} 1 & \text{if } BBox(x, y) < \frac{1}{N_x + N_y} \sum_{x=0}^{N_x} \sum_{y=0}^{N_y} BBox(x, y) \\ 0 & \text{if } BBox(x, y) > \frac{1}{N_x + N_y} \sum_{x=0}^{N_x} \sum_{y=0}^{N_y} BBox(x, y) \end{cases} \quad (6)$$

$Mask(x, y)$ und $BBox(x, y)$ beschreiben die Pixel der Maske bzw. Bounding-Box an der Stelle (x, y) und N_x bzw. N_y die Anzahl der Pixel in der jeweiligen Richtung. Der Gedanke hinter diesem Vorgehen ist, dass die Bounding-Box im Normalfall nur die Person und den Hintergrund enthält. Sollte dennoch zusätzlich ein Hinderniss in der Bounding-Box auftauchen, würde es nur einen kleinen Teil im Ausschnitt ausmachen und somit die Maske kaum verfälschen. In Abbildung 3.3 wird die Maskierung anhand eines Beispielbildes gezeigt.

Anschließend werden die Tiefenpixel aus dem Tiefenbild, die sich innerhalb der Maske befinden, erneut gemittelt, um die gemittelte Tiefe der verfolgten Person zu erhalten. Zusammen mit dem Mittelpunkt des Detektionsfensters ergeben sich daraus die 3D-Koordinaten $p = [p_x, p_y, p_z]^T$ der Person.

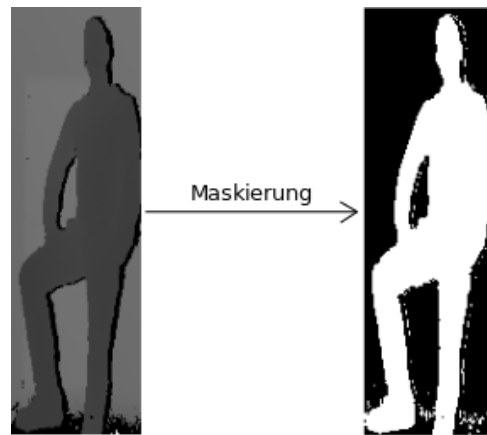


Abbildung 3.3: Die Maskierung der Bounding-Box des Tiefenbildes

3.1.4 Regler

Die jetzt bekannten Koordinaten $p = [p_x, p_y, p_z]^T$ der Person sind aus mehreren Gründen als finales Ziel ungeeignet. Einerseits ist eine Erkennung der Person schwierig, wenn das Fahrzeug nah an diese heranfährt. Andererseits fährt es bei stationärer Ausregelung stets die Person an. Daher ergibt sich das finale Ziel bzw. die Führungsgröße aus einer radialen Verschiebung der Personenkoordinaten in Richtung der Kamera mittels des Strahlensatzes zu

$$w = \begin{pmatrix} w_x \\ w_y \\ w_z \end{pmatrix} = \begin{pmatrix} p_x \cdot (1 - d_p/d_{euc}) \\ p_y \\ p_z \cdot (1 - d_p/d_{euc}) \end{pmatrix} \quad (7)$$

mit $d_{euc} = \sqrt{p_x^2 + p_z^2}$ und d_p als beliebiger Abstand zur Person. Dabei wird der Abstand zu $d_p = 2.0$ m festgelegt, allerdings bietet es sich an, diesen variabel zu gestalten, um zum Beispiel Fahrten um Ecken in schmalen Gängen zu verbessern. Bei diesem großen Wert kann es durchaus vorkommen, dass solche Szenarien nicht bewältigt werden können.

Eine Anfahrt dieser Zielkoordinate w ist auf zwei Varianten möglich. Die einfachste und ressourcenschonendste ist die Regelung mithilfe eines konventionellen Reglers. Mit der Wahl eines PD-Reglers entspricht diese Variante der hier verwendeten. Dabei muss zum einen der Lenkwinkel mit

$$\delta = K_{p,\delta} \cdot w_x(t) + K_{D,\delta} \frac{w_x(t) - w_x(t-1)}{dt} \quad (8)$$

berechnet werden, um die x -Koordinate anzufahren. Zum anderen muss die Geschwindigkeit

$$v = K_{p,v} \cdot w_z(t) + K_{D,v} \frac{w_z(t) - w_z(t-1)}{dt} \quad (9)$$

berechnet werden, um mit einem flüssigen Fahrverhalten die z -Koordinate zu erreichen. Dabei entsprechen $K_{p,\delta}$ und $K_{D,\delta}$ bzw. $K_{p,v}$ und $K_{D,v}$ den jeweiligen Verstärkungsfaktoren, t dem Abtastzeitpunkt und dt der Differenz beider Abtastzeitpunkte. Sofern stationäre Genauigkeit gewünscht ist, müssen die Regelgesetze um einen I-Anteil erweitert werden und eine Implementierung eines Anti-Windups sollte erfolgen.

Als zweite Variante ist die Kombination mit einem lokalen Mapper und Planer möglich. Hier kann der Navigation Stack aus Kapitel ?? bei gleichbleibender Konfiguration verwendet werden. Das Ziel w wird stetig als ROS-Message PoseStamped gesendet, welche direkt als Zielvorgabe dem Planer übergeben werden kann. Somit liegt ein dynamisches Ziel vor, das mit einer Erkennung von möglichen Hindernissen und einer Umgehung dieser angefahren wird. Hierbei ist allerdings zu beachten, dass diese Variante sehr rechenintensiv ist und auf der aktuellen Hardwarekonfiguration des Fahrzeugs nicht lauffähig ist.

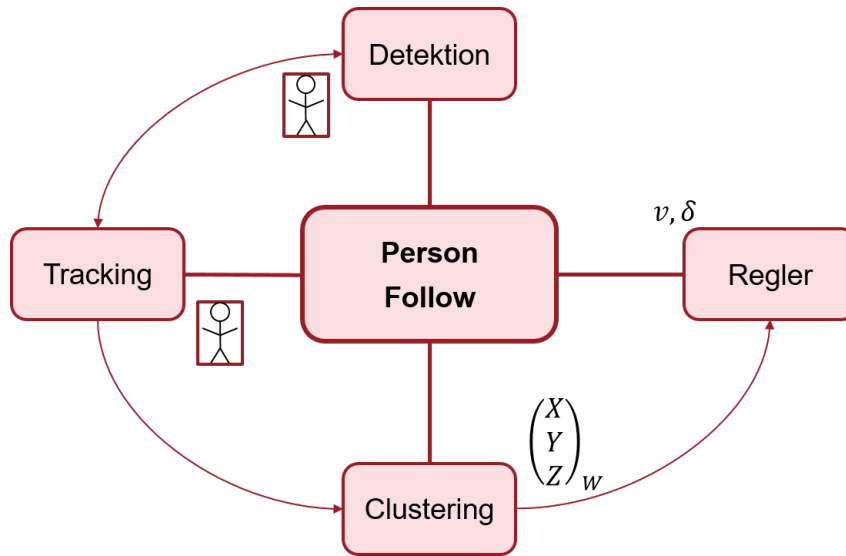


Abbildung 3.4: Struktur der Personenverfolgung mit jeweiligen Modulen.


3.2 Implementierung und Umsetzung

Die Struktur des Algorithmus der Personenverfolgung mit ihren jeweiligen Modulen ist in Abbildung 3.4 dargestellt. Dabei ist anhand der Pfeilrichtungen gut zu erkennen, wie die einzelnen Module miteinander verknüpft sind. Anfangs wird mittels der Detektion eine Person im Farbbild detektiert und als Zielperson für die Anfahrt festgelegt. Letzteres wird durch die Erstellung einer Bounding Box um die Person erreicht. Da diese Detektion stetig stattfindet, ist sie ziemlich rechenaufwändig. Die Bildrate der Kamera des Autos fällt dabei stark ab. Daher ist die Kombination mit einem Tracker notwendig. Dieser wird aktiviert, sobald eine Personendetektion erfolgt und bekommt die entsprechende Bounding Box übergeben. Da Trackeralgorithmen grundsätzlich mit einem gewissen Drift der Position der Person bzw. des Objektes einhergehen ist eine Umschaltung zur Detektion nach einer gewissen Zeit sinnvoll, um die Korrektheit der Bounding Box wiederherzustellen. Somit findet ein stetiger Wechsel beider Module statt. Im Modul Clustering erfolgt die Berechnung der Personenkoordinaten. Dazu geschieht eine Projektion der Bounding Box vom Farbbild in das Tiefenbild. Innerhalb der projizierten Bounding Box wird die Person von der Umgebung extrahiert und ihre Tiefe bestimmt. Mithilfe dieser und der Projektionsmatrix erfolgt die Berechnung der Koordinaten der Person. Diese bekommt der Regler übergeben, welcher daraus das finale Ziel berechnet und einen entsprechenden Lenkwinkel θ und eine entsprechende Geschwindigkeit v vorgibt.

3.3 Ergebnisse und Probleme

Die erste Versuchsreihe wird mit dem Fahrzeug auf dem Tisch platziert durchgeführt. Hierbei soll ausschließlich die Personendetektion und das Tracking ohne Bewegung des Fahrzeugs getestet werden. Die Resultat sind durchaus vielversprechend. Die Personverfolgung aus Detektion und Tracking können die Person in diesen Versuchen verlässlich verfolgen, wobei auch fehlerhaftes Tracking erkannt wurde. Das Erkennen des misslunge-

nen Trackings ist hingegen nicht besonders robust und somit noch ausbaufähig. In der zweiten Versuchsreihe wird die vollständige Personenverfolgung samt Folgeregelung getestet. Um überhaupt eine Person im Kamerabild zu erhalten, muss hierfür die Kamera stark nach oben geneigt werden. Diese veränderte Perspektive stellt sich als Problem für die Personenverfolgung dar, da das Bild nun stark verzerrt wird. Sowohl die Detektion als auch das Tracking verschlechtert sich aus dieser Perspektive und sind deutlich weniger robust. Dennoch gelingt das Folgen einer Person regelmäßig für eine längere Zeitdauer. Um die Ergebnisse in Zukunft deutlich zu verbessern, sollte eine perspektivische Transformation des Kamerabildes vor der Personenverfolgung durchgeführt werden. Ein weiteres Problem ist die Verfolgung einer Personen um enge Ecken, da der fest eingestellte Abstand von zwei Metern für diesen Fall zu groß ist. Hierfür wären zwei Lösungsansätze möglich. Einerseits könnte der Abstand variabel gestaltet werden, sodass dieser bei der Fahrt um eine Ecke verringert wird. Dabei entsteht das Problem, dass die Person nicht mehr komplett im Kamerabild erfasst werden kann. Somit wäre die optimale Lösung, die Verwendung eines Kalman Filters, der die Person mithilfe der zuvor berechneten Bewegung auch außerhalb des Kamerasichtfeldes weiter verfolgen kann. Dieses könnte in Kombination zum bereits implementierten Tracker eingesetzt werden und somit die gesamte Personenverfolgung verbessern.



4 Fazit und Ausblick

A Anhang

A.1 Launch und Config Dateien des Navigation Stack

Listing A.1: launch/kai_configuration.launch

```
<launch>
  <!-- Transform between coordinate systems -->
  <node pkg="tf" type="static_transform_publisher" name="
    base_laser_broadcaster" args="0.08 0 0.115 0 0 0 1 base_link
    base_laser 20" />
  <node pkg="tf" type="static_transform_publisher" name="
    base_link_broadcaster" args="0 0 0 0 0 0 1 base_footprint
    base_link 20" />
  <node pkg="tf" type="static_transform_publisher" name="
    base_camera_depth_frame_broadcaster" args="0 0 0 0 0 0 1
    base_laser camera_depth_frame 20" />
  <include file="$(find kinect2_bridge)/launch/kinect2_bridge.
    launch"/>

  <!-- Start odometry node -->
  <node pkg="pses_odometry" type="odometry_node" name="
    odometry_node" output="screen">
  </node>
</launch>
```

Listing A.2: launch/move_base.launch

```
<launch>
  <master auto="start"/>

  <arg name="depth_image" default="/kinect_utilities/
    depth_image_filtered"/>

  <!-- Create LaserScan -->
  <node pkg="depthimage_to_laserscan" type="
    depthimage_to_laserscan" name="depthimage_to_laserscan"
    output="screen">
    <remap from="image" to="$(arg depth_image)"/>
  </node>

  <!-- Run the map server -->
  <arg name="pses_map" default="/home/pses/catkin_ws/src/
    pses_control/pses_navigation/map2.yaml"/>
  <node name="map_server" pkg="map_server" type="map_server" args
    ="$(arg pses_map)"/>

  <!-- Run AMCL -->
  <include file="$(find amcl)/examples/amcl_diff.launch" />
```

```

<!-- Start Navigation Stack -->
<node pkg="move_base" type="move_base" respawn="false" name="
  move_base" output="screen">
  <rosparam file="$(find pses_navigation)/cfg/
    costmap_common_params.yaml" command="load" ns="
    global_costmap" />
  # for obstacle parcoure costmap_common_params_obstacle.yaml
  <rosparam file="$(find pses_navigation)/cfg/
    costmap_common_params.yaml" command="load" ns="
    local_costmap" />
  # for obstacle parcoure costmap_common_params_obstacle.yaml
  <rosparam file="$(find pses_navigation)/cfg/
    local_costmap_params.yaml" command="load" />
  <rosparam file="$(find pses_navigation)/cfg/
    global_costmap_params.yaml" command="load" />
  <rosparam file="$(find pses_navigation)/cfg/
    base_local_planner_params.yaml" command="load" />
  # for obstacle base_local_planner_params_obstacle.yaml
  <rosparam file="$(find pses_navigation)/cfg/
    base_global_planner_params.yaml" command="load" />
  <param name="base_local_planner" value="teb_local_planner /
    TebLocalPlannerROS" />
  <param name="base_global_planner" value="navfn/NavfnROS" />
  <param name="controller_frequency" value="8.0"/>
  <rosparam param="recovery_behaviors"> [{name: "
    conservative_reset", type: "clear_costmap_recovery /
    ClearCostmapRecovery"}, {name: "aggressive_reset", type: "
    clear_costmap_recovery/ClearCostmapRecovery"}] </rosparam>
  <param name="planner_patience" value="5.0"/>
  <param name="clearing_rotation_allowed" value="false"/>
</node>

</launch>

```

Listing A.3: cfg/base_global_planner_params.yaml

```

NavfnROS:
  allow_unknown: true
  cost_factor: 0.55

```

Listing A.4: cfg/base_local_planner_params.yaml

```

TebLocalPlannerROS:

```

```

  odom_topic: odom
  map_frame: /map

```

```

# Trajectory

```

```

teb_autosize: True
dt_ref: 0.75 # 0.5 for obstacle parcour
dt_hysteresis: 0.1
global_plan_overwrite_orientation: True
max_global_plan_lookahead_dist: 3.5
feasibility_check_no_poses: 5

# Robot
max_vel_x: 0.8 # 0.5 for obstacle parcour
max_vel_x_backwards: 0.5
max_vel_y: 0.0
acc_lim_y: 0.0
max_vel_theta: 3.1
acc_lim_x: 2
acc_lim_theta: 1.12
footprint_model:
  type: "polygon"
  vertices: [ [-0.18, 0.105], [0.18, 0.105], [0.18, -0.105], [-0.18,
    -0.105] ] # for type "polygon"
min_turning_radius: 1.0
wheelbase: 0.25 # Wheelbase of our robot
cmd_angle_instead_rotvel: True # stage simulator takes the angle
    instead of the rotvel as input (twist message)

# GoalTolerance

xy_goal_tolerance: 0.3
yaw_goal_tolerance: 0.5
free_goal_vel: True

# Obstacles

min_obstacle_dist: 0.2
include_costmap_obstacles: True
costmap_obstacles_behind_robot_dist: 1.0
obstacle_poses_affected: 30
costmap_converter_plugin: ""
costmap_converter_spin_thread: True
costmap_converter_rate: 5

# Optimization

weight_max_vel_x: 2.0

# Homotopy Class Planner

enable_homotopy_class_planning: False

```

```

enable_multithreading: True
simple_exploration: False
max_number_classes: 3
roadmap_graph_no_samples: 15
roadmap_graph_area_width: 5
h_signature_prescaler: 0.5
h_signature_threshold: 0.1
obstacle_keypoint_offset: 0.1
obstacle_heading_threshold: 0.45
visualize_hc_graph: False

```

Listing A.5: `cfg/costmap_common_params.yaml`

```

obstacle_range: 5
raytrace_range: 5.5
footprint: [[-0.18, 0.105], [0.18, 0.105], [0.18, -0.105], [-0.18,
    -0.105]]
map_type: costmap

plugins:
  - {name: static_map, type: "costmap_2d::StaticLayer"}
  - {name: obstacles, type: "costmap_2d::VoxelLayer"}
  - {name: inflation_layer, type: "costmap_2d::InflationLayer"}

obstacles:
  combination_method: 1
  observation_sources: laser_scan_sensor
  laser_scan_sensor: {sensor_frame: base_laser, data_type:
    LaserScan, topic: /scan, marking: true, clearing: true}

inflation_layer:
  inflation_radius: 1.0
  cost_scaling_factor: 0.7 # 5 for obstacle parcou

```

Listing A.6: `cfg/global_costmap_params.yaml`

```

global_costmap:
  global_frame: /map
  robot_base_frame: base_link
  update_frequency: 5.0
  static_map: true
  transform_tolerance: 0.5

static_map:
  enable: true
  map_topic: map

```

Listing A.7: `cfg/local_common_params.yaml`

```

local_costmap:

```

```
global_frame: odom
robot_base_frame: base_link
update_frequency: 5.0
publish_frequency: 2.0
static_map: false
transform_tolerance: 0.2
rolling_window: true
width: 10.0
height: 10.0
resolution: 0.05
```