

Bachelorarbeit

Video-basierte Erkennung von Fahrspurmarkierungen auf mobilen Robotern

Jan Wille 1535115

12.09.2022

Erstprüfer: Prof. Dr.-Ing. Hanno Homann
Zweitprüfer: Prof. Dr.-Ing. Martin Mutz

Selbstständigkeitserklärung

Hiermit bestätige ich, dass die folgende Arbeit eigenständig von mir allein erstellt und unter Berücksichtigung der zur Verfügung gestellten Aufgabenstellung sowie dem Arbeitsmaterial unter Angabe aller verwendeten Quellen erarbeitet wurde. Die Regelungen und Konsequenzen eines Plagiats, inklusive disziplinarischer Maßnahmen, sind mir bewusst. Insbesondere wurden alle Zitate und gedanklichen Übernahmen als solche kenntlich gemacht.

Jan Wille

Abstract

Inhaltsverzeichnis

Abstract	IV
Inhaltsverzeichnis	V
Glossar	VII
1 Einleitung	1
1.1 Problemstellung	1
1.2 Aufgabenstellung	1
1.3 Inhalt der Arbeit	1
2 Stand der Technik	2
2.1 Techniken zur Fahrspurerkennung	2
2.1.1 Geometrische und algorithmische Ansätze	2
2.1.2 Deep-Learning Ansätze	3
2.2 OpenCV	4
2.3 Das Robot Operating System	4
2.4 Der JetBot Roboter	5
2.4.1 Performance Baseline	6
2.5 Aufgebaute Fahrbahnfläche	7
3 Kamera-Kalibrierung	8
3.1 Intrinsische Kalibrierung	8
3.1.1 Radiale Verzerrung	8
3.1.2 Tangentiale Verzerrung	9
3.2 Durchführung der intrinsischen Kalibrierung	10
3.2.1 Python Script zur Durchführung der Kalibrierung	10
3.2.2 Anwenden der Kalibrierung in einer ROS Node	13
4 Erkennung von Fahrspurmarkierungen	17
4.1 Konzeptionierung in Python	17
4.1.1 Kantenerkennung mittels Canny-Edge-Detektor	19
4.1.2 Klassifizierung der Kantenpixel	20
4.1.3 Linienbildung	23
4.2 Implementierung in einer ROS Node	26
4.2.1 Erstellung des Quellcodes	26
4.2.2 Performance Betrachtung	30
4.3 Optimierung durch eigene Implementierung des Canny-Edge-Detectors	31
4.3.1 Anpassung des Quellcodes	31
4.3.2 Performance Betrachtung	34
5 Fazit	35
6 Ausblick	36

Literatur	37
Abbildungsverzeichnis	39
Tabellenverzeichnis	40
Codeverzeichnis	41

Glossar

C++

Eine hardwarenahe, performante Programmiersprache.

Callback-Funktion

Eine Funktion, die unter bestimmten Bedingungen automatisch aufgerufen wird. Im Bezug auf ROS geht es meistens um Funktionen die für jede Message auf einem abonnierten Topic mit deren Inhalt aufgerufen werden.

Canny-Edge-Detektor

Ein Algorithmus zur Erkennung von Kanten in Bildern, entwickelt von John Canny [Can86].

Gaußscher Filter

Eine Filterfunktion um Bilder zu glätten. Sie bildet den mittels einer 2D-Gaußfunktion gewichteten Mittelwert einer Pixelnachbarschaft.

Gradientenorientierung

Richtung des Farbgradienten in einer Pixelnachbarschaft. Verläuft von dunklen zu hellen Bildbereichen.

JetBot

Der in dieser Arbeit verwendete Roboter. Gesteuert wird er von einem NVIDIA Jetson Nano Entwicklerboard, auf welchem die erstellten Programme laufen.

Kernel

Eine Matrix die beim Filtern und Verarbeiten von Bildern verwendet wird. Mit ihr werden Pixelnachbarschaften gewichtet.

Kamerakoordinaten

Ein 3D-Koordinatensystem, welches die Sicht der Kamera widerspiegelt. x und y -Koordinaten korrespondieren dabei zu Pixelkoordinaten und die z -Achse stellt die Entfernung zur Kamera dar.

Pixelkoordinaten (Bildkoordinaten)

Ein 2D-Koordinatensystem, welches die Position der Bild-Pixel abbildet. Es liegt in der x/y -Ebene des Kamerakoordinaten, der Ursprung ist aber um die halbe Bildgröße nach links-oben verschoben.

Weltkoordinaten

Ein 3D-Koordinatensystem, das die gesamte Szene/Welt des derzeitigen Systems umfasst.

OpenCV

OpenCV ist eine Open Source Software Bibliothek mit typischen Algorithmen und Funktionen für die Bildverarbeitung, *Computer Vision* und maschinelles Lernen.

Pixel

Ein einzelner Bildpunkt. Er hat einen Wert zwischen Schwarz (0) und Weiß (255).

Pixelnachbarschaft

Ein Bereich festgelegter Größe um einen relevanten Hauptpixel.

Python

Eine höhere, einfach zu benutzende Programmiersprache.

ROI, kurz für Region of Interest

Ein Bildbereich, der für die derzeitige Anwendung relevant ist. Das restliche Bild wird vernachlässigt.

ROS, kurz für Robot Operating System

Das Robot Operating System ist eine Sammlung von Softwarebibliotheken und Werkzeugen, die hilfreich beim Erstellen von Roboter Applikationen sind. Es enthält Treiber, fertig implementierte Standardalgorithmen und andere hilfreiche Funktionen. Es ist vollständig Open Source.

ROS Message

Ein Datensatz, der von ROS von einer Node an eine andere übertragen wird. Sie wird unter einem Topic veröffentlicht und alle Abonenten diese Topics erhalten die Message. ROS definiert eigene Datentypen denen Nachrichten entsprechen müssen.

ROS Node

Eine ROS Node ist ein Teilprogramm, welches von ROS verwaltet wird. Es kann Informationen als Topic veröffentlichen und Topics abonnieren, um die dort veröffentlichten Informationen weiter zu verarbeiten.

ROS Nodelet

Ein ROS Nodelet ist ein Programm, welches in einem Verbund mit mehreren anderen Nodelets mittels ROS gestartet wird. Alle Nodelets einer Gruppe haben die Möglichkeit auf geteilten Anwendungsspeicher zuzugreifen.

ROS Topic

Über ein Topic stellt eine Node Messages bereit. Es ist eine Bekanntmachung, das andere Nodes hier Informationen erhalten können.

1 Einleitung

1.1 Problemstellung

Auf der Projektfläche *Autonomes Fahren* des Instituts für Konstruktionselemente, Mechatronik und Elektromobilität (IKME) der Hochschule Hannover ist eine große urbane Kreuzung im Maßstab 1:18 nachgebildet. Hier sollen in Zukunft automatisierte Logistikkonzepte mit mobilen Roboterfahrzeugen entwickelt und getestet werden. Die Roboter sind jeweils mit einer nach vorne gerichteten Videokamera ausgerüstet. Um die Fahrzeuge damit sicher steuern zu können, wird eine zuverlässige Fahrspurerkennung benötigt.

1.2 Aufgabenstellung

Ziel der Arbeit ist es, eine echtzeitfähige Erkennung der Fahrspurmarkierungen aus dem Video-Bilddatenstrom zu realisieren und die Position der Markierungen relativ zum Fahrzeug anzugeben. Um eine geometrisch richtige Darstellung zu erhalten, soll zunächst eine Bestimmung der intrinsischen und extrinsischen Kamera-Kalibrierung durchgeführt werden. Mit den so bestimmten intrinsischen Parametern soll dann eine Rektifizierung der Bilder durchgeführt werden.

Auf den rektifizierten Bildern soll dann die eigentliche Erkennung der Spurmarkierungen erfolgen. Dies kann entweder kanten-basiert oder mit tiefen neuronalen Netzen erfolgen. Die extrinsische Kalibrierung soll dann genutzt werden, um die Position der Markierungen in Fahrzeug-Koordinaten umzurechnen. Zusätzlich kann die Farbinformation des Bildes genutzt werden, um zwischen weißen und gelben Linien zu unterscheiden. Gegebenenfalls kann auch das zeitliche Tracking eines Spurmodells umgesetzt werden.

Die Bildverarbeitung sollte auf der Jetson-nano Hardware unter ROS in Echtzeit lauffähig sein. Eine erste Implementierung kann mit Python erfolgen. Für den längerfristigen Einsatz wäre eine Umsetzung in C++ mit ROS Nodelets wünschenswert.

1.3 Inhalt der Arbeit

Der Inhalt der Arbeit gliedert sich in mehrere Kapitel, die hier kurz vorgestellt werden. Begonnen wird mit einem Kapitel über die theoretischen Grundlagen und existierenden Vorgehensweisen in anderen Arbeiten. Außerdem werden die technischen Gegebenheiten definiert.

Danach wird die Kalibrierung der Kamera beschrieben. Dabei geht es um die Aufbereitung des Kamerabildes, dass einige Verzerrungen aufweist. Diese werden durch eine intrinsische Kalibrierung korrigiert.

Im darauffolgenden Kapitel geht es um die eigentliche Erkennung der Fahrspurmarkierungen. Hier wird der entwickelte Algorithmus und die Umsetzung als ROS Node erläutert.

Die letzten beiden Kapitel ziehen ein Fazit und Fassen die Ergebnisse zusammen. Außerdem wird ein Ausblick auf Weiterentwicklungs-Möglichkeiten gegeben.

2 Stand der Technik

2.1 Techniken zur Fahrspurerkennung

Das Thema Fahrspurerkennung beschäftigt die Wissenschaft und auch die Automobilindustrie bereits seit einigen Jahren. Im Folgenden werden daher die existierenden üblichen Ansätze zu diesem Thema erläutern. Dabei ist besonders interessant, wie diese in diese Arbeit einfließen.

2.1.1 Geometrische und algorithmische Ansätze

Bei den klassischen und ältesten Methoden wird an die Thematik mit mathematisch-geometrischen Ansätzen herangegangen. Diese werden zu Algorithmen verknüpft, um unterschiedliche Informationen herauszuarbeiten, zu verknüpfen und das Ergebnis zu verfeinern.

Grundlage bilden hierbei verschiedene Operationen, mit welchen sich Bilder verändern lassen. Solche Operationen verknüpfen eine bestimmte Menge an Pixeln eines Ursprungsbildes mittels einer mathematischen Operation, um ein neues Pixel für das Zielbild zu ermitteln. Ein relativ simples Beispiel hierfür ist das Bilden eines Mittelwertes von jeweils drei Farbpixeln, um ein Grauwert-Bild zu erzeugen. [Sze11]

Der sehr grobe Ablauf, welcher solche Operationen zu einem Algorithmus verknüpft, ist in Abbildung 2.1 skizziert. Dabei sind die Einzelschritte in der Realität jedoch häufig sehr kompliziert. Begonnen wird eigentlich immer mit einem Vorbereitungs-Schritt, da die Bilder einer Kamera nur selten direkt verwendet werden können. Teilweise ist eine solche Vorverarbeitung aber auch hardwareseitig oder in vorgelagerten Programmteilen umgesetzt. Meisten wird das Bild außerdem in ein Grauwert-Bild umgewandelt, da so nur ein Drittel der Pixel untersucht werden müssen, was die Performance verbessert.

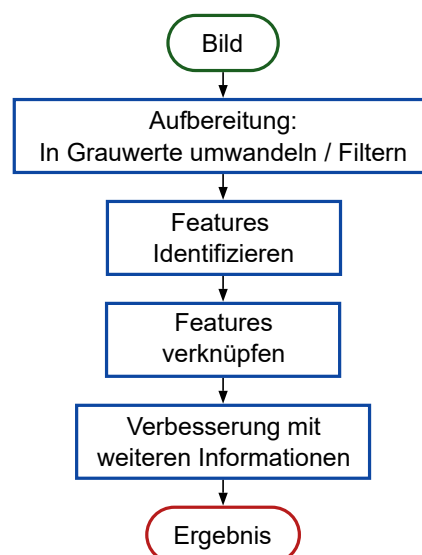


Abb. 2.1: Generischer Ablauf von Fahrspurerkennung (nach [KK15])

Bei den sogenannten Features handelt es sich um spezifische, möglichst eindeutige Muster im Bild. Im Fall von Fahrspurmarkierungen sind dies meist Kannten und Ecken ebendieser. Häufig wird hier der Canny-Edge-Detektor eingesetzt, der von John Canny [Can86] entwickelt wurde. Dieser Algorithmus ist sehr gut zum Identifizieren von Kantenpixeln geeignet und erzeugt ein Binärbild, in dem nur noch ein Pixel dicke Umrisse verbleiben.

Das genaue Vorgehen um Features zu finden und zu verknüpfen, ist Gegenstand von Forschung und Entwicklung. In [KK15] werden Beispielsweise verschiedenen Möglichkeiten die Hough Transformation zu verwenden miteinander verglichen und [WHF05] verwendet zusätzlich eine Methode zum Kombinieren von einzelnen Liniensegmenten. [Lop+05] demonstriert einen Ansatz, um direkt Mittellinien von Fahrspurmarkierungen anhand von bekannten Größenparametern abzuleiten.

Das Ergebnis kann durch das Einbeziehen weiterer Informationen noch weiter verbessert werden. Oft wird hier das originale Farbbild mit einbezogen (siehe [KK15]), aber auch das Zurückgreifen auf das vorherige Bild, wie in [Lu+19] eingesetzt, ist möglich.

Da diese Techniken bereits langfristig erprobt und daher sehr stark optimiert sind, eignen sie sich besonders gut für den Einsatz in dieser Arbeit. Auch der gezeigte Ablauf wird in dieser Arbeit so angewendet.

Insbesondere die gezeigten Methoden des Canny-Edge-Detektor werden später noch einmal aufgegriffen. Die in vielen Quellen erwähnte Hough Transformation wurde aber bereits im Voraus als zu rechenintensiv ausgeschlossen.

2.1.2 Deep-Learning Ansätze

Alternativ zu den traditionellen Ansätzen wird in den letzten Jahren vor allem an den sogenannten Deep-Learning-Methoden geforscht. Hier werden Neuronale Netze verwendet, um Lösungen für besonders schwierigen Situationen wie extremen Lichtverhältnissen oder stark verwitterte Spurmarkierungen zu finden.

Diese Netzwerke können mittels eines großen Datensatzes an Bildern trainiert werden, um unter unterschiedlichsten, unvorhersehbaren Bedingungen noch Ergebnisse zu erhalten. Dabei gilt, je größer und vielfältiger der Trainingsdatensatz, umso wahrscheinlicher erzielt das trainierte Netzwerk gute Ergebnisse. [Zha+21] vergleicht frei verfügbare Datensätze und liefert eine Übersicht für welche Anwendungen diese sich eignen.

Der generelle Detektierungsablauf ist auch bei Deep-Learning-Methoden der in Abbildung 2.1 dargestellte. Je nach Methode und Netzwerk können aber mehrere Schritte vom selben Netzwerk erledigt oder mehrere Netzwerke für die einzelnen Schritte verwendet werden.

In den letzten Jahren wurden unterschiedlichste Arten von Neuronale Netzen für die Detektion von Spurmarkierungen entwickelt und erprobt. Vier unterschiedliche Netzwerkarten, deren Verwendungszweck und Vorteile, sowie ein Vergleich der Ergebnisse ist in [TLL21] nachzulesen. Dort wurden Genauigkeiten von 97 % erreicht, allerdings war die Generalisierung der getesteten Modelle nicht immer zuverlässig.

Der in [Zha+21] erstellte Vergleich kommt zu sehr ähnlichen Ergebnissen. Zusätzlich wurde hier die Performance der Netzwerke untersucht und angegeben, wie viele Bilder pro Sekunde verarbeitet werden können. Hier haben sich einige Methoden als sehr viel schneller herausgestellt.

Ein großer Nachteil aller Deep-Learning-Methoden ist die benötigte hohe Rechenleistung. Bereits das Training der Netzwerke erfordert einen existierenden, ausreichen großen Datensatz und viel Zeit und Rechenleistung. Aber auch zur Nutzung der fertig trainierten Netzwerke ist wieder einiges an Rechenleistung notwendig. Hier kann zwar die GPU zu Hilfe genommen werden, es ist aber ein Nachteil gegenüber den traditionellen Ansätzen.

Dies ist insbesondere für den Anwendungsfall dieser Arbeit ein Problem. Die Ressourcen des JetBots sollen so wenig wie möglich belastet werden, damit dieses mit weiten Anwendungen

geteilt werden können. Da einige potenzielle Anwendungen auf die Verwendung von Neuralen Netzen angewiesen sind, ist insbesondere die GPU sehr sparsam zu benutzen. Daher ist für diese Arbeit die Verwendung von Deep-Learning nicht sinnvoll und findet keine Anwendung.

2.2 OpenCV

Das Open-Source-Projekt OpenCV (kurz für *Open Source Computer Vision Library*) ist eine Sammlung von Softwaremodulen, die der Bildverarbeitung und dem maschinellen Lernen dienen. Sie verfügt über mehr als 2500 optimierte Algorithmen mit denen Anwendungen wie Objekterkennung, Bewegungserkennung und 3D-Modell Extraktion erstellt werden können. Daher ist sie eine der Standardbibliotheken, wenn es um digitale Bildverarbeitung geht und wird fast immer zur Demonstration neuer Konzepte benutzt. Da sie sowohl in C/C++, Java und Python genutzt werden kann, ist sie außerdem sehr vielseitig und hat den Vorteil, dass Konzepte in einer abstrakten Sprache wie Python getestet werden und später relativ simpel in eine hardwarenahe Programmiersprache übersetzt werden können. Weitere Informationen sind in der Dokumentation des Projektes [Ope22a] zu finden.

In dieser Arbeit wird diese Bibliothek daher insbesondere für die Entwicklungsphase verwendet. Da der Bibliothekscode jedoch auch viele potenziell nicht benötigte Zusatzfunktionen mitbringt, wird auch ein Wechsel auf eine eigene Implementierung mit besserer Performance in Betracht gezogen.

2.3 Das Robot Operating System

Das Robot Operating System (kurz: ROS) ist eine Sammlung von Software Bibliotheken und Werkzeugen, die zum Erstellen von Roboter Applikationen dienen. Es bietet eine eigene Paketverwaltung über die bestehende Bibliotheksfunktionen für die Verwendung heruntergeladen werden können. Dabei handelt es sich um verschiedenste Anwendungen, angefangen Treibern, über fertige, direkt anwendbare Algorithmen, bis zu nutzernahen Steueroberflächen und sogar (Lern-)Spiele. Die Webseite des Projektes [Rob22] bietet hierzu weitere Informationen. Außerdem bietet ROS Integrationen mit anderen bestehenden Projekten, wie zum Beispiel OpenCV.

Auch wenn es sich bei ROS genommen um kein vollständiges Betriebssystem handelt, stellt es für ein solches typische Funktionalitäten zur Verfügung. Beispiele hierfür sind Hardware-Abstraktion, tiefgehende Geräteverwaltung, Verwaltung von Prozessen sowie Informationsweitergabe zwischen diesen und die eben genannte Paketverwaltung und damit verbundene Abstraktion von generischen, allgemein benötigten Funktionen. [Rob18]

Für diese Arbeit ist ROS deshalb interessant, da sich die Ergebnisse so modular an potenzielle weitere Prozesse weitergeben lassen. Dies wird durch ROS Fähigkeit möglich, Einzelprozesse als sogenannte ROS Nodes zu erstellen. Jede Node kann eigene Informationen als sogenannte Topics veröffentlichen und andere, parallel laufende Nodes können diese abonnieren.

Sobald eine Information in einem Prozess bereit ist, verpackt dieser sie in einem der definierten Datentypen als sogenannte Message. Diese wird dann veröffentlicht und an alle Abonnenten des Topics verschickt. Diese können dann bei Erhalt der Message auf diese reagieren.

So lassen sich einzelne Komponenten dieser Arbeit abgekapselt voneinander umsetzen und stellen ihre Ergebnisse auf potenziellen, später noch entwickelten Prozessen zur Verfügung.

2.4 Der JetBot Roboter

Beim in für diese Arbeit verwendeten Roboter handelt es sich um einen JetBot v2.1 von der Firma Sparkfun. Im Folgenden werden seine Komponenten und Einsatzmöglichkeiten für diese Arbeit näher beschrieben.

Der Roboter verwendet das von Nvidia produzierte Jetson Nano Entwicklerboard [Nvi22]. Hierbei handelt es sich um einen Mini-Computer der speziell für Bildverarbeitung und Selbstlernende Algorithmen entwickelt wurde. Es verfügt über einen 4-Kern ARM Prozessor als CPU sowie, neben herkömmlichen Anschlüssen für PC-Peripherie, über Anschlüsse für 2 Kameras und mehrere GPIO Pins. Dadurch eignet es sich bereits sehr gut für eingebettet Anwendungen.

Was dieses Board für die Anwendung in der Bildverarbeitung aber besonders interessant macht, ist die integrierte, für die Größe leistungsstarke GPU. Diese kann für grafikintensive Anwendungen, aber vor allem für das Arbeiten mit Neuralen Netzen genutzt werden. Da für diese Arbeit explizit nicht mit Deep Learning gearbeitet wird, um diese Ressourcen für spätere Weiterentwicklung freizuhalten, ist dieses Feature aber hier uninteressant.

Die Firma Sparkfun bietet für dieses Board den Bausatz [Spa22] an, um einen selbstfahrenden Roboter zu bauen. Dieser wird für diese Arbeit verwendet. Er stattet das Board mit einer Kamera, zwei steuerbaren Motoren, einem LCD-Display und einer Batterie aus. Der fertig aufgebaute Roboter ist in Abbildung 2.2 zu sehen.



Abb. 2.2: SparkFun JetBot AI Kit V2.1 [Spa22]

Für diese Arbeit wird der Roboter als funktionstüchtig und einsatzbereit vorausgesetzt. Die Aufbauanleitung ist aber unter [Spa22] und die Anleitung zum Einrichten unter [Jet22] zu finden.

Zusätzlich wird ein fertiger Kamertreiber vorausgesetzt. Dieser wurde als Node mit dem Namen `camera_driver` unter ROS implementiert und stellt alle 0,2s ein aktuelles Bild auf dem Topic `/img/raw` zur Verfügung.

2.4.1 Performance Baseline

Da die Leistungsfähigkeit des JetBots relativ eingeschränkt ist und eines der Ziele dieser Arbeit lautet, parallel zu anderen, zukünftigen Prozessen laufen zu können, ist es wichtig möglichst Performant zu arbeiten. Daher wird ermittelt, wie sehr der JetBot vor Beginn dieser Arbeit bereits ausgelastet ist.

Begonnen wird mit der Grundleistung ohne irgendwelche laufenden Prozesse unter ROS. Das bedeutet, das nur das Betriebssystem und seine Standard-Applikationen, wie zum Beispiel der SSH-Server, laufen. Dazu wird das Terminalprogramm `jtop` verwendet, welches viele Systeminformationen und Performance-Messwerte gesammelt anzeigt. Ein Screenshot ist in Abbildung 2.3 gezeigt.

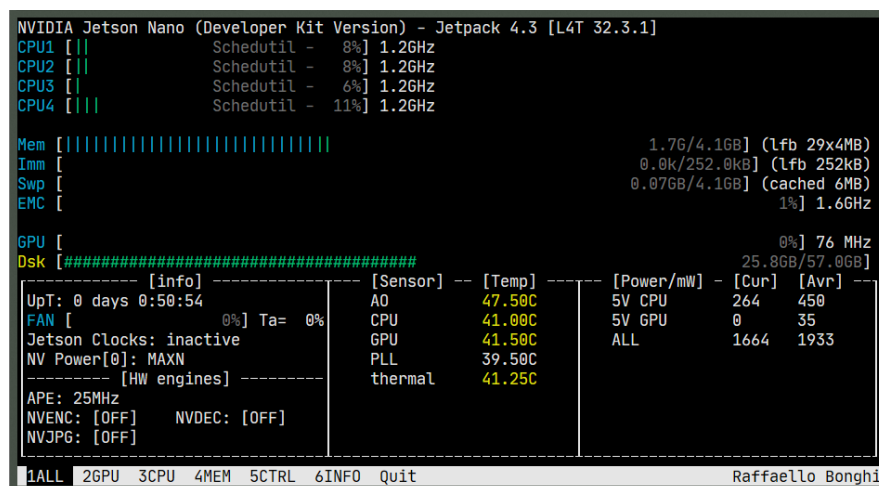


Abb. 2.3: CPU Auslastung des JetBots ohne ROS

Da alle in dieser Arbeit entwickelten Programme lediglich die CPU benutzen, ist dies der einzige relevante Messwerte. Trotz sind die anderen Werte interessant und werden mit dokumentiert. Wie man im Screenshot sieht, ist das System in diesem Fall kaum belastet. Die Auslastung ohne irgendwelche laufenden Prozesse liegt bei $\approx 8-11\%$.

Relevanter ist aber der Fall mit laufender Kamera, da dies die Voraussetzung für diese Arbeit ist. Wird die Kamera-Node gestartet und erneut `jtop` überprüft, ergibt sich die in Abbildung 2.4 gezeigte Screenshot.

Dort ist die CPU-Auslastung deutlich höher und liegt bei $\approx 38\%$. Es ist allerdings zu beachten, dass dies nicht ausschließlich auf die Kamera-Node zurückzuführen ist. Um überhaupt ROS Nodes benutzen zu können, muss das sogenannte ROS Core gestartet sein. Diese geschieht automatisch beim Starten der ersten Node. Es ist für den Großteil der zusätzlichen Auslastung verantwortlich, sodass zusätzliche Nodes die Auslastung nur geringfügig erhöhen werden.

```

NVIDIA Jetson Nano (Developer Kit Version) - Jetpack 4.3 [L4T 32.3.1]
CPU1 [|||||] Schedutil - 39% 1.2GHz
CPU2 [|||||] Schedutil - 36% 1.2GHz
CPU3 [|||||] Schedutil - 34% 1.2GHz
CPU4 [|||||] Schedutil - 46% 1.2GHz

Mem [|||||] 1.96/4.16B (lfb 21x4MB)
Imm [|||||] 0.0k/252.0kB (lfb 252kB)
Swp [|||||] 0.069GB/4.16B (cached 7MB)
EMC [|||||] 8% 1.6GHz

GPU [|||||] 0% 76 MHz
Dsk [|||||] 25.2GB/57.0GB

----- [info] ----- [Sensor] -- [Temp] ----- [Power/mW] - [Cur] [Avr] -----
UpT: 0 days 0:55:13      AO      49.50C      5V CPU      621      1047
FAN [|||||] 0% Ta= 0%    CPU      43.00C      5V GPU      36       47
Jetson Clocks: inactive GPU      42.00C      ALL      2484     2889
NV Power[0]: MAXN      PLL      41.00C
----- [HW engines] ----- thermal 42.75C
APE: 25MHz
NVENC: [OFF] NVDEC: [OFF]
NVJPG: [OFF]

iALL 2GPU 3CPU 4MEM 5CTRL 6INFO Quit Raffaele Bonghi

```

Abb. 2.4: CPU Auslastung mit laufender Kamera und ROS-Core

2.5 Aufgebaute Fahrbahnfläche

Zum Testen des JetBots für diese Arbeit und andere Projekt wurde eine die Projektfläche *Autonomes Fahren* aufgebaut. Diese besteht aus einer im Maßstab 1:18 herunterskalierten Fläche mit aufgedruckter Fahrbahn. Die folgende Abbildung zeigt die Projektfläche. Diese hat eine Größe von 200 m² und Abmessungen von 20 m×10 m. Weitere Informationen können dem Projektbericht [GW+21] entnommen werden.

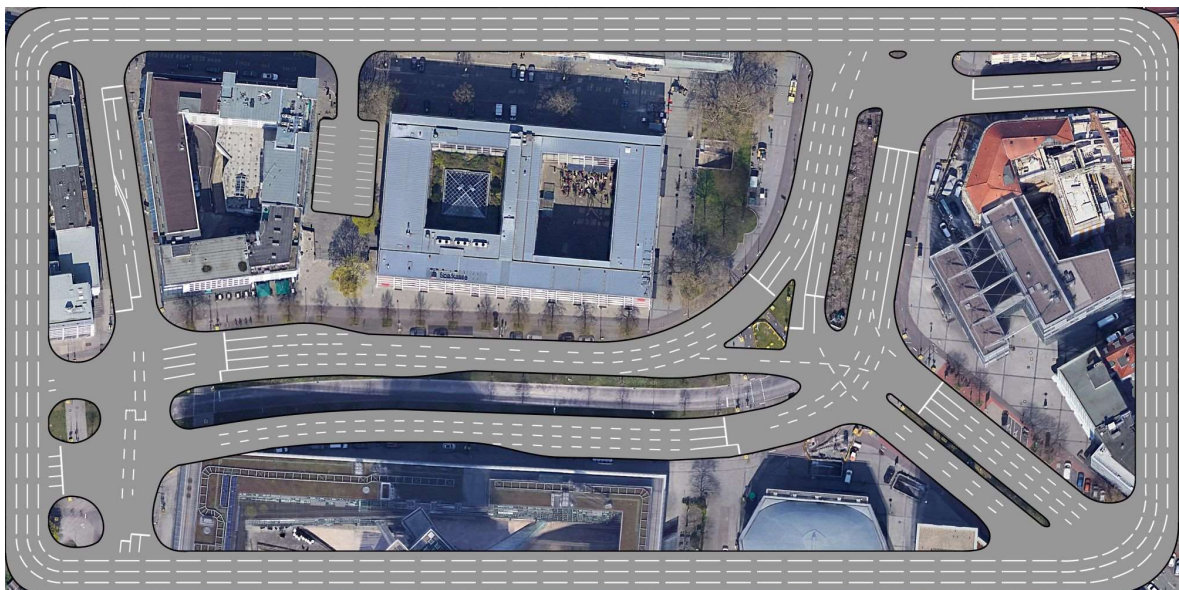


Abb. 2.5: Ansicht von oben auf die Fahrbahnfläche [GW+21]

3 Kamera-Kalibrierung

Damit die später beschriebene Fahrspurerkennung zuverlässig funktioniert, wird eine Kalibrierung der Kamera vorgenommen. Das Vorgehen dazu und die Ergebnisse sind im folgenden Kapitel dokumentiert.

3.1 Intrinsische Kalibrierung

Bedingt durch den technischen Aufbau des Linsensystems und Ungenauigkeiten bei der Herstellung sind die von der Kamera gelieferten Bilder merklich verzerrt. In Abbildung 3.1 ist dies gut anhand der Linien des Schachbrettes zu erkennen. Sie verlaufen in der Realität alle parallel, im Bild sehen sie aber gekrümmt aus.

Es können unterschiedliche Arten von Verzerrung in einem Kamerabild auftreten, die in den folgenden Unterkapiteln einzeln erläutert werden.

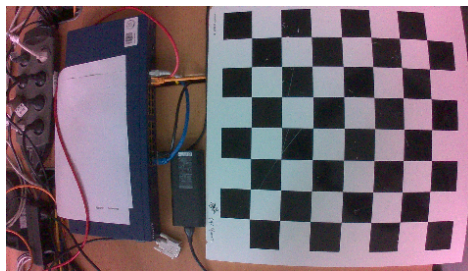


Abb. 3.1: Unkalibriertes Kamerabild mit tonnenförmiger Verzerrung

3.1.1 Radiale Verzerrung

Die erste mögliche Art der Verzerrung ist die radiale Verzerrung. Diese ist die auffälligste Art und wird häufig auch *Fischaugen Effekt* genannt. Bedingt durch die Brechung des Lichtes an den Kanten der Blende und der Linse entsteht eine Ablenkung der Lichtstrahlen in der Kamera, die mit der Entfernung vom Mittelpunkt immer stärker wird. Nimmt die Ablenkung mit der Entfernung zu, spricht man von positiver, kissenförmiger Verzerrung, den umgekehrte Fall nennt man negative, tonnenförmige Verzerrung. Zur Verdeutlichung ist in Abbildung 3.2 die Auswirkung dieser Verzerrungen auf ein Rechteckmuster skizziert.

Mathematisch lässt sich die Veränderung eines Punktes durch die Verzerrung wie in Gleichung 3.1 beschrieben berechnen. Dabei beschreiben x und y die unverzerrten Pixelkoordinaten, k_1 , k_3 und k_5 die Verzerrungs-Koeffizienten. Theoretisch existieren noch weitere Koeffizienten, aber in der Praxis haben sich die ersten drei als ausreichend herausgestellt. [Han11]

$$\begin{aligned}x_{\text{distorted}} &= x(1 + k_1r^2 + k_3r^4 + k_5r^6) \\y_{\text{distorted}} &= y(1 + k_1r^2 + k_3r^4 + k_5r^6)\end{aligned}\tag{3.1}$$

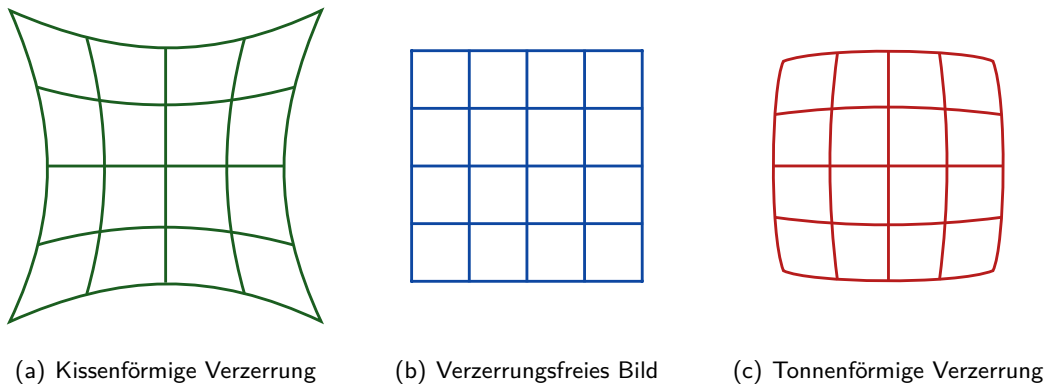


Abb. 3.2: Darstellung der optischen Verzerrungen (nach [Wik22])

3.1.2 Tangentiale Verzerrung

Die tangentiale Verzerrung entsteht durch kleine Ausrichtungsfehler im Linsensystem. Dadurch liegt die Linse oder der Sensor nicht ideal in der Bildebene und der Bildmittelpunkt sowie die Bildausrichtung können leicht verschoben sein. Das einfallende Licht wird dadurch nicht mehr an dieselbe Stelle gebündelt wie im Idealfall und das Bild wird verzerrt. Dies ist in Abbildung 3.3 skizziert.

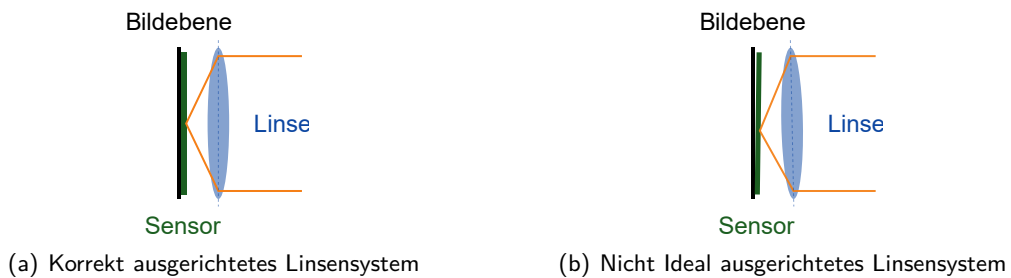


Abb. 3.3: Probleme in der Ausrichtung von Sensor und Linse (nach [Mat22])

Mathematisch wird diese Verzerrung durch den folgenden Zusammenhang beschrieben. [Han11]

$$\begin{aligned} x_{\text{distorted}} &= x + \left[2p_1xy + p_2(r^2 + 2x^2) \right] \\ y_{\text{distorted}} &= y + \left[p_1(r^2 + 2y^2) + 2p_2xy \right] \end{aligned} \quad (3.2)$$

Somit werden beiden Verzerrungsarten zusammen durch fünf Parameter beschrieben, die sogenannten Verzerrungs-Koeffizienten. Historisch begründet wird dabei k_3 an das Ende geschrieben, da dieser Parameter früher kaum berücksichtigt wurde.

$$D_{\text{coeff}} = (k_1, k_2, p_1, p_2, k_3) \quad (3.3)$$

Um die Parameter bestimmen zu können, müssen folglich mindestens fünf Punkte gefunden werden, von denen die Weltkoordinaten und die Kamerakoordinaten bekannt sind. Da sich die Punktpaare aber nur schwer mathematisch perfekt bestimmen lassen, werden mehr Paare benötigt, um ein überbestimmtes Gleichungssystem zu erhalten und dieses nach dem geringsten Fehler zu lösen. [Ope22b]

In der Praxis werden 2D-Muster verwendet, um Punktepaaire zu bestimmen. Da sich alle Punkte dieser Muster in einer Ebene befinden, kann der Ursprung der Weltkoordinate in eine Ecke des Musters gelegt werden, sodass die z -Koordinate keine Relevanz mehr hat und wegfällt. [Fra+09] Dabei werden Muster so gewählt, dass es möglichst einfach fällt die Weltkoordinaten der Punkte zu bestimmen. Beispielsweise sind bei einem Schachbrettmuster die Entfernungen alle identisch und können als 1 angenommen werden, wodurch die Koordinaten der Punkte direkt ihrer Position im Muster entsprechen.

3.2 Durchführung der intrinsischen Kalibrierung

Zur Durchführung der Kalibrierung wird ein Python-Script erstellt, um den Vorgang einfach und wiederholbar zu machen. Als Vorlage für dieses dient die Anleitung zur Kamera Kalibrierung aus der OpenCV Dokumentation [Ope22b].

Außerdem wird eine ROS Nodelet erstellt, welches die Kalibrierung auf den Video-Stream anwendet und korrigierte Bilder veröffentlicht.

3.2.1 Python Script zur Durchführung der Kalibrierung

Grundlage für die Kalibrierung ist es, eine Reihe von Bildern mit der zu kalibrierenden Kamera aufzunehmen, auf denen sich ein Schachbrettartiges Kalibriermuster befindet. Wichtig ist es, dasselbe Muster und dieselbe Auflösung für alle Bilder zu verwenden. Es muss sich dabei nicht um eine quadratische Anordnung handeln, jedoch muss die Anzahl der Zeilen und Spalten im Code angegeben werden. Dabei ist allerdings nicht die Anzahl der Felder gemeint, sondern die Anzahl der inneren Kreuzungspunkte. Ein normales Schachbrett hat beispielsweise 8×8 Felder, aber nur 7×7 interne Kreuzungen. Zur Verdeutlichung sind die Kreuzungspunkte des verwendeten Kalibrierusters in Abbildung 3.4 grün markiert.

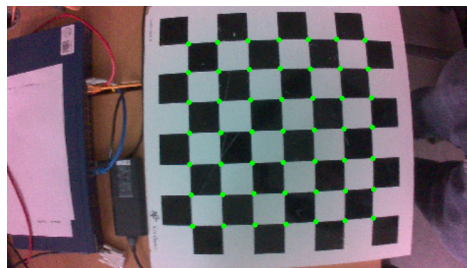


Abb. 3.4: Schachbrett-Kalibriermuster mit markierten inneren Kreuzungen

Es wird nun ein standard Schachbrett als Kalibriermuster verwendet, wie es bereits in Abbildung 3.4 zu sehen ist. Dessen Kalibriermustergröße von 7×7 wird im Code als Konstante definiert:

Code 3.1: Definition der Größe des Kalibrierusters

```
1 # define the grid pattern to look for
2 PATTERN = (7,7)
```

Entsprechend der Anleitung [Ope22b] werden benötigte Variablen initialisiert (siehe Code 3.2). Nun werden alle im aktuellen Ordner befindlichen Bilder eingelesen und in einer Liste abgespeichert. Jedes Listenelement wird eingelesen und in ein Grauwert-Bild umgewandelt. Dieses wird dann an die OpenCV Funktion `findChessboardCorners()` übergeben, welche die Kreuzungspunkte findet und zurückgibt.

Code 3.2: Initialisierung von Variablen für die Kalibrierung

```

1 # termination criteria
2 criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30,
3             0.001)
4 # prepare object points, like (0,0,0), (1,0,0),..., (6,5,0)
5 objp = np.zeros((PATTERN[0]*PATTERN[1],3), np.float32)
6 objp[:, :2] = np.mgrid[0:PATTERN[0], 0:PATTERN[1]].T.reshape(-1,2)
7 # Arrays to store object points and image points from all the images.
8 objpoints = [] # 3d point in real world space
9 imgpoints = [] # 2d points in image plane.

```

Code 3.3: Finden und Verarbeiten der Kalibrierbilder

```

1 # get all images in current directory
2 folder = pathlib.Path(__file__).parent.resolve()
3 images = glob.glob(f'{folder}/*.png')
4
5 # loop over all images:
6 for fname in images:
7     img = cv.imread(fname)
8     gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
9
10    # Find the chess board corners
11    ret, corners = cv.findChessboardCorners(gray, PATTERN, None,
12                                           flags=cv.CALIB_CB_ADAPTIVE_THRESH)

```

Dabei ist es gar kein Problem, wenn nicht in jedem Bild das Kalibriermuster gefunden werden kann, solange insgesamt ausreichend nutzbare Bilder vorhanden sind. Bei nicht nutzbaren Bildern gibt `findChessboardCorners()` `None` zurück und das Bild wird übersprungen.

Für alle nutzbaren Bilder werden die in Code 3.2 erstellten Punktbezeichnungen zur Liste der gefundenen Objekte hinzugefügt. Die Genauigkeit der gefundenen Eckkoordinaten wird über die Funktion `cornerSubPix()` erhöht und diese werden an die Liste der gefundenen Bildpunkte angehängt.

Code 3.4: Abspeichern der gefundenen Bildpunkte

```

1 # If found, add object points, image points
2 if ret == True:
3     objpoints.append(objp)
4     corners2 = cv.cornerSubPix(gray, corners, (11,11), (-1,-1),
5                                criteria)
6     imgpoints.append(corners)

```

Jetzt kann die eigentliche Kalibrierung mittels der OpenCV Funktion `calibrateCamera()` durchgeführt werden. Diese nimmt die zuvor erstellten Listen von Objektkoordinaten und Bildpunkten und löst damit die in Abschnitt 3.1 beschriebenen Gleichungen. Als Ergebnis liefert sie die Kameramatrix K und die Verzerrungs-Koeffizienten D_{coeff} zurück. [Ope22b]

Der gesamte Code wird nun auf einen Datensatz von Bildern angewandt, um die Ergebnisse für den vorliegenden Roboter zu erhalten. Der Datensatz ist auf dem GitLab Server unter der [Wil22a] abgelegt. Damit ergeben sich die folgenden Kalibrierungsergebnisse.

Code 3.5: Ermitteln der Kalibrierwerte mittels OpenCV

```

1 # get calibration parameters:
2 ret, K, D_coeff, rvecs, tvecs = cv.calibrateCamera(objpoints,
    imgpoints, gray.shape[:,-1], None, None)

```

$$k_1 = -0,42049309612684654$$

$$k_2 = 0,3811654512587829$$

$$p_1 = -0,0018273837466050299$$

$$p_2 = -0,006355252159438178$$

$$k_3 = -0,26963105010742416$$

$$K = \begin{pmatrix} 384,65 & 0 & 243,413 \\ 0 & 384,31 & 139,017 \\ 0 & 0 & 1 \end{pmatrix}$$

Um zu zeigen, wie sich das Bild damit verbessern lässt, werden die Ergebnisse auf eines der Bilder angewandt. Da sich die Abmessungen des entzerrten Bildes, von denen des verzerrten unterscheiden, wird zuerst die OpenCV Funktion `getOptimalNewCameraMatrix()` verwendet, welche eine weiter skalierte Kameramatrix ermittelt, mit der die Abmessungen zueinander passen. Diese liefert außerdem eine ROI, also den Bildbereich der nur relevante (nicht leere) Pixel enthält.

Mit dieser zusätzlichen Matrix kann nun die OpenCV Funktion `undistort()` auf das Bild angewandt werden. Diese produziert das entzerrte Bild mit leeren Pixeln in den Bereichen, wo keine Informationen im Originalbild vorlagen. Um diese leeren Pixel zu entfernen wird das Bild auf die ROI reduziert.

In Abbildung 3.5 ist die Entzerrung des Beispielbildes mit dem Zwischenschritt mit Leerpixeln gezeigt.

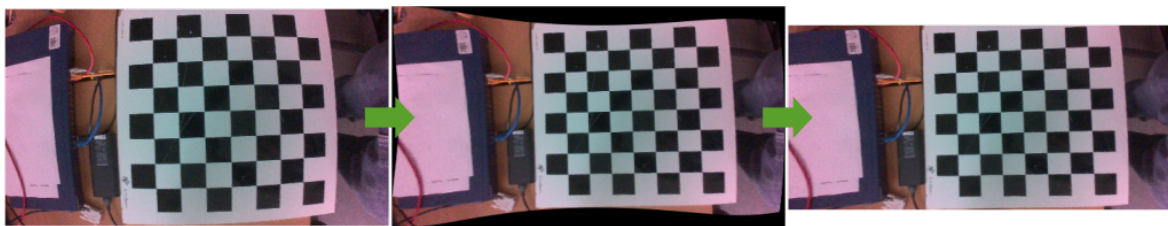


Abb. 3.5: Schritte der Bild-Entzerrung

Reprojektions-Fehler

Um eine Aussage über die Genauigkeit der gefundenen Kalibrierungs-Parameter treffen zu können, wird der Reprojektions-Fehler bestimmt. Dieser gibt den Abstand zwischen einem im Kalibriermuster gefundenen Kreuzungspunkt und den mittels der Kalibrierungsergebnisse berechneten Weltkoordinate. Der Mittelwert aller Abweichungen in allen verwendeten Bilder gibt den Reprojektions-Fehler für den ganzen Kalibriervorgang an.

Der Code 3.6 zeigt die Berechnung mittels von OpenCV zur Verfügung gestellten Funktionen und den zuvor ermittelten Kalibrierdaten. Für jeden Satz an theoretischen Weltkoordinate des Kalibriermusters in `objpoints` werden die Punkte im Bild mit der OpenCV Funktion `projectPoints()`

bestimmt und mit den gefundenen Punkten verglichen. Dazu wird die OpenCV Funktion `norm()` verwendet, die direkt die Summe aller Differenzen zwischen den beiden Punktelisten liefert. Das Ergebnis wird auf dem Bildschirm ausgegeben.

Code 3.6: Berechnen des Reprojektions-Fehlers

```

1 # calculate re-projection error
2 mean_error = 0
3 for i in range(len(objpoints)):
4     imgpoints2, _ = cv.projectPoints(objpoints[i], rvecs[i],
5                                     tvecs[i], mtx, dist)
6     error = cv.norm(imgpoints[i], imgpoints2,
7                     cv.NORM_L2)/len(imgpoints2)
8     mean_error += error
9 print(f"total error: {mean_error/len(objpoints)}")

```

Mit dem verwendeten Datensatz ergibt sich ein Reprojektions-Fehler von 0,049. Dies ist ausreichend für diesen Anwendungsfall.

3.2.2 Anwenden der Kalibrierung in einer ROS Node

Um die Kalibrierungsergebnisse auf jedes Bild, das vom Kamera-Treiber veröffentlicht wird, anzuwenden, wird eine weitere Node erstellt. Diese entzerrt jedes erhaltene Bild und veröffentlicht die korrigierte Version als eigenes Topic. Das korrigierte Bild wird sowohl in Farbe als auch in Graustufen veröffentlicht. Die Beziehung der Topics ist in Abbildung 3.6 grafisch dargestellt.

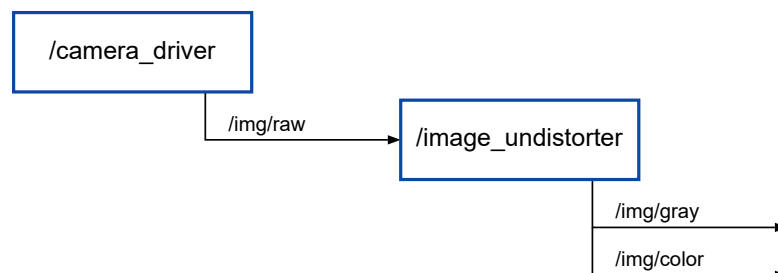


Abb. 3.6: Beziehungen der Entzerrer-Node zu bestehenden Nodes

Initialisieren der Node

Beim Start der Node wird die `main()` Funktion aufgerufen, welche die notwendigen ROS Funktionen zur Initialisierung aufruft, das benötigte Topic abonniert, eine Callback-Funktion anhängt und die eigenen Topics veröffentlicht.

Außerdem werden die Kalibrierdaten aus einer Konfigurationsdatei im YAML-Format eingelesen und in Variablen übernommen. Die Verzerrungsparameter werden als Vektor eingelesen und die Kameramatrix wird in eine OpenCV Matrix umgewandelt. Außerdem wird die Bildgröße benötigt und daher aus der Konfigurationsdatei gelesen. Code 3.7 zeigt diesen Ablauf. Es ist sinnvoll, dies bereits in der `main()` Funktion durchzuführen, um die Callback-Funktion zu entlasten und dort Rechenzeit einzusparen.

Code 3.7: Einlesen der Kalibrierungsergebnisse aus einer YAML-Datei

```
1 // open YAML-file and get config
2 std::string configFilePath = "../tools/calibration/calibration.yaml";
3 YAML::Node full_config = YAML::LoadFile(configFilePath);
4 YAML::Node camera_config = full_config["cameras"]["default"];
5
6 // read distortion coefficients and convert to OpenCV vector
7 auto distortion_YAML = camera_config["intrinsic"]["distortion"]
8   .as<std::vector<double>>();
9 cv::Mat distortion ( distortion_YAML );
10
11 // read camera matrix and convert to OpenCV matrix
12 auto cameraMatrix_YAML = camera_config["intrinsic"]["matrix"]
13   .as<std::vector<std::vector<double>>>();
14 cv::Mat cameraMatrix = toMat( cameraMatrix_YAML );
15
16 // read image size
17 cv::Size imageSize(
18   full_config["images"]["size"]["width"].as<int>(),
19   full_config["images"]["size"]["height"].as<int>()
20 );
```

Mit diesen Werten können nun *Mappings* erzeugt werden, welche die geometrische Beziehung zwischen einem Pixel im Originalbild und einem Pixel im entzerrten Bild abspeichern. Es werden zwei *Mappings* für die x und die y -Koordinate erzeugt, welche in globalen Variablen abgelegt werden. Das ist notwendig damit die Informationen der Callback-Funktion zur Verfügung stehen. Zuvor ist es aber noch sinnvoll, eine umskalierte, optimierte Kameramatrix zu erzeugen. OpenCV stellt hierzu die Funktion `getOptimalNewCameraMatrix()` zur Verfügung. Diese erstellt die neue Matrix abhängig von einem freien Skalierungsparameter α . Für $\alpha = 0$ ist die zurückgegebene Matrix so gewählt, dass das entzerrte Bild möglichst wenig unbekannte Pixel enthält. Das bedeutet aber, dass einige Pixel des Originalbildes außerhalb des neuen Bildbereiches liegen und vernachlässigt werden. Mit $\alpha = 1$ enthält das entzerrte Bild alle Pixel des Originalbildes, allerdings bleiben einige Pixel schwarz. Da die Funktion zusätzlichen eine ROI liefert, welches den Bildausschnitt ohne schwarze Pixel beschreibt, wird hier $\alpha = 1$ verwendet. Die veröffentlichten Bilder werden zwar auf die ROI reduziert, aber die zusätzlichen Informationen sind grundsätzlich vorhanden und bei Bedarf kann das Programm angepasst werden, um die vollständigen Bilder zu veröffentlichen.

Code 3.8: Bestimmen der Pixel-Mappings zu Entzerrung

```
1 // get scaled camera matrix
2 auto scaledCameraMatrix = cv::getOptimalNewCameraMatrix(cameraMatrix,
3   distortion, imageSize, 1, imageSize, &ROI);
4
5 // calculate undistortion mappings
6 cv::initUndistortRectifyMap(cameraMatrix, distortion, cv::Mat(),
7   scaledCameraMatrix, imageSize, CV_16SC2, rectifyMapX,
8   rectifyMapY);
```

Callback-Funktion zur Handhabung der Einzelbilder

Die Callback-Funktion `callback_undistort_image()` wurde während der Initialisierung an das Topic `/img/raw` angehängt und wird nun für jedes dort veröffentlichte Bild aufgerufen. Der Code 3.9 zeigt eine vereinfachte Version der Implementierung, ohne Umwandlung in ein Grauwert-Bild und ohne Laufzeitmessung.

Da das Bild als ROS eigener Datentyp übergeben wird, muss es zuerst in ein mit OpenCV kompatibles Format umgewandelt werden. Die dazu notwendigen Funktionen sind im ROS-Paket `cv_bridge` zur Verfügung gestellt. Dessen Funktion `toCvCopy()` kopiert die Daten des Originalbildes in eine OpenCV Matrix, welche weiter verwendet werden kann.

Das Bild kann nun mit der OpenCV Funktion `remap()` entzerrt werden. Diese benutzt die zuvor bestimmten *Mappings*, um jeden Pixel des Originalbildes an die korrekte Position im entzerrten Bild zu übertragen. Dabei erfolgt eine lineare Interpolation.

Das erhaltene Bild wird auf die ROI reduziert und unter dem Topic `/img/color` veröffentlicht. Außerdem wird eine Grauwert-Version erzeugt und diese als `/img/gray` veröffentlicht, was hier aber nicht gezeigt ist.

Code 3.9: Vereinfachte Version der Callback-Funktion zur Durchführung der Entzerrung

```

1 void callback_undistort_image(sensor_msgs::Image original) {
2     cv::Mat undistortedImage;
3
4     // convert from ROS msg-type to opencv matrix
5     cv_bridge::CvImagePtr imagePtr = cv_bridge::toCvCopy(original);
6
7     // apply the calculated maps to undistort the image
8     cv::remap(imagePtr->image, undistortedImage, rectifyMapX,
9               rectifyMapY, cv::INTER_LINEAR);
10
11    // crop relevant section from image
12    undistortedImage = undistortedImage(ROI);
13
14    // publish images
15    cv_bridge::CvImage colorImage(std_msgs::Header(), "rgb8",
16    undistortedImage);
17    pub_colorImage->publish(colorImage.toImageMsg());
18 }

```

Performance Betrachtung

Da diese Node eine Grundlagenfunktion darstellt und parallel zu jeder anderen Anwendung laufen muss, ist es wichtig, dass sie möglichst performant ist und wenig Ressourcen des JetBots verbraucht.

Daher wurde die mittlere CPU Auslastung und die durchschnittliche Laufzeit der Callback-Funktion, welche für jedes Bild durchlaufen wird, gemessen.

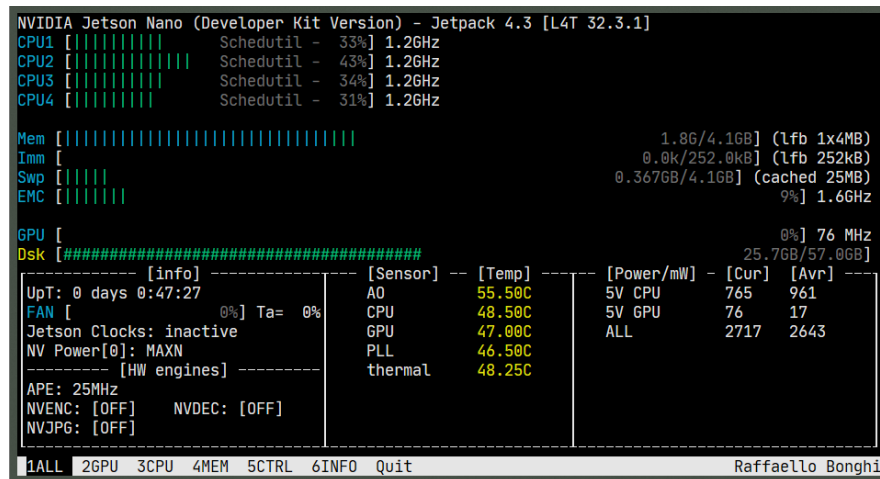


Abb. 3.7: CPU Auslastung des JetBots mit laufender Kamera und Entzerrer-Node

Der jtop Screenshot in Abbildung 3.7 zeigt die CPU Nutzung bei aktivem ROS-Core, Kamera-Treiber und der neu erstellten Entzerrer-Node. Die durchschnittliche CPU-Auslastung liegt bei ungefähr 35,25 %, ist also sogar sehr geringfügig niedriger als die in Unterabschnitt 2.4.1 gemessene Grundauslastung ohne die neue Node. Das ist aber auf die starke Fluktuation in der CPU-Auslastung und die daher ungenauen Messung zurückzuführen. Die Auslastung wird daher als identisch betrachtet.

Um die Laufzeit der Node zu bestimmen, wird die aktuelle Zeit, wie sie von der Funktion `ros::Time::now()` zurückgegeben wird, verwendet. Die Zeit beim Start der Callback-Funktion wird abgespeichert. Nach Durchlauf der Funktion wird erneut die aktuelle Zeit bestimmt und die Differenz in Millisekunden als Debug-Nachricht ausgegeben. Die Laufzeit der Node wird über einige Durchläufe gemittelt. Es ergibt sich ein Mittelwert von $\approx 4,07$ ms.

Tab. 3.1: Gemessene Laufzeit bei 10 Durchläufen der Callback-Funktion

Durchlauf Nr.	gemessene Laufzeit
1	3,885 ms
2	4,068 ms
3	3,969 ms
4	3,712 ms
5	5,231 ms
6	4,086 ms
7	4,025 ms
8	3,897 ms
9	3,753 ms
10	4,096 ms

4 Erkennung von Fahrspurmarkierungen

Dieses Kapitel thematisiert, wie die Erkennung der Fahrspurmarkierungen umgesetzt wird. Begonnen wird mit einer konzeptionellen Version in Python, mit der der Ablauf des Algorithmus geplant und getestet wird. Danach wird die Logik in einer C++ Node umgesetzt, um die bestmögliche Performance zu erhalten.

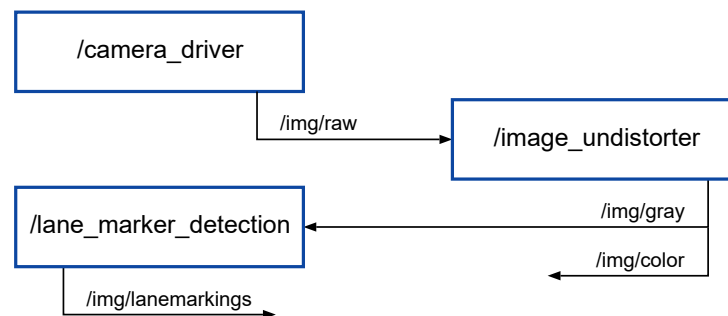


Abb. 4.1: Zusammenhang der Fahrspurmarkierung-Erkennungs-Node mit den bestehenden Nodes

Wie diese neuen Node mit den bestehenden Nodes in Beziehung stehen soll, ist in Abbildung 4.1 grafisch dargestellt. Neu ist dabei, dass diese Node das korrigierte Grauwert-Bild von der in Unterabschnitt 3.2.2 beschriebenen Entzerrer-Node abonniert und das eigene Ergebnis als neues Topic zur Verfügung stellt.

4.1 Konzeptionierung in Python

Die Entwicklung und Konzeptionierung des Algorithmus erfolgt in Python. Diese Sprache muss nicht kompiliert werden und hat eine einfache Syntax, wodurch das Testen beschleunigt wird und sie generell einfacher zu verwenden ist.

Der Algorithmus lässt sich in mehrere Einzelschritte aufteilen, welche in den folgenden Unterkapiteln beschreiben im Einzelnen beschrieben sind. Zur Übersicht ist aber der gesamte Ablauf in Abbildung 4.2 skizziert. Angefangen wird dort mit dem Erhalten des Bildes, womit sowohl manuelles Laden eines Beispiel-Bildes, als auch das Erhalten des Bildes über ein Topic gemeint ist.

Während einer Testfahrt des JetBots wurden von der Entzerrer-Node veröffentlichte Bilder abgespeichert, sodass sie zum lokalen Testen zur Verfügung stehen. Diese wurden unter [Wil22b] abgelegt. In Abbildung 4.3 ist eines dieser Bilder gezeigt, mit dem im Folgenden die Einzelschritte demonstriert werden.

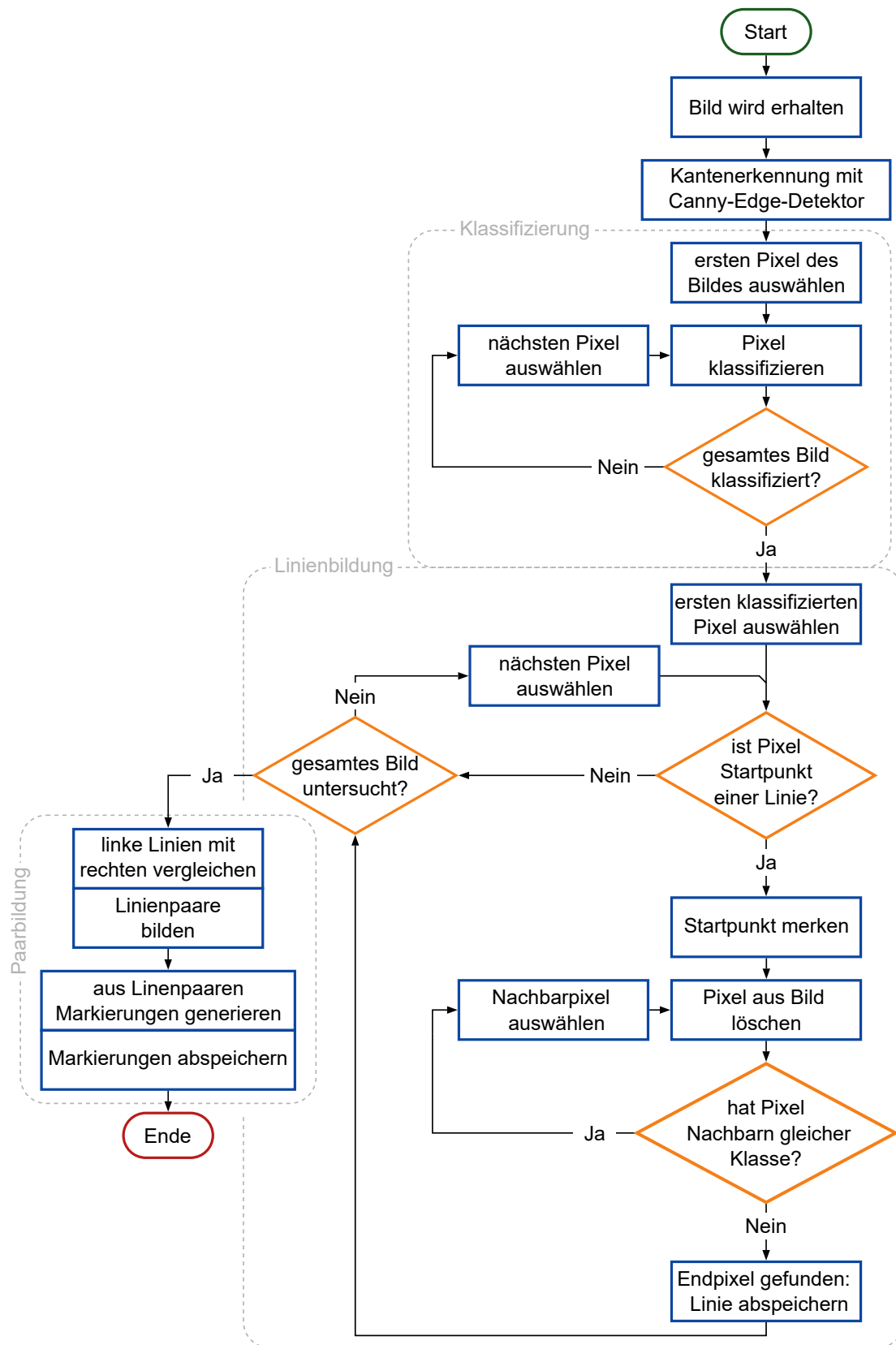


Abb. 4.2: Ablauf des Algorithmus zur Erkennung von Fahrspurmarkierungen

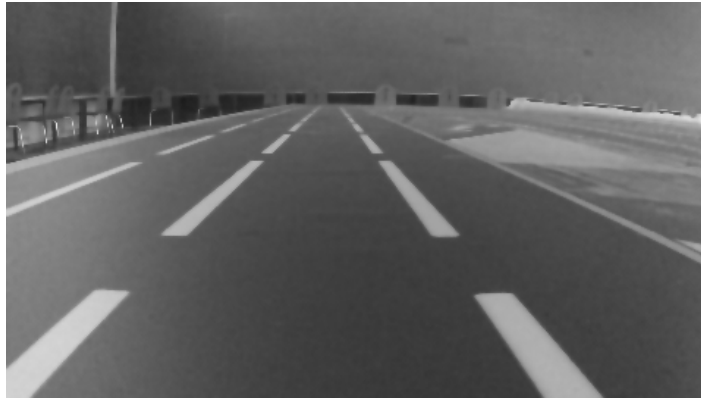


Abb. 4.3: Beispiel-Bild an dem der Ablauf demonstriert wird

4.1.1 Kantenerkennung mittels Canny-Edge-Detektor

Begonnen wird mit der Detektion von Kanten im Bild. Dazu wird das Bild zuerst mit OpenCV geladen. Um kleine Störungen im Bild, welche bestehende Kanten verzerren oder als falsche Kante detektiert werden könnten, zu reduzieren, wird das Bild mit einem Gaußschen Filter geglättet. Es wird ein 3×3 Kernel mit einer Normalverteilung von $\sigma = 1,5$ verwendet. OpenCV stellt hierzu die Funktion `GaussianBlur()` zur Verfügung, der das geladene Bild, die Kernelgröße und der Wert für σ übergeben wird.

Die eigentliche Kantenerkennung wird mittels eines Canny-Edge-Detektors durchgeführt. Dabei handelt es sich um einen von John Canny 1983 entwickelten und in [Can86] veröffentlichten Algorithmus. Dieser bestimmt für jeden Pixel den Gradientenbetrag in x und y -Richtung. Dann werden diejenigen Pixel unterdrückt, welche entlang der Gradientenrichtung kein Maximum darstellen. Zum Abschluss wird das Bild mit einem Hysterese-Schwellwert binarisiert. Das bedeutet, dass alle Pixel über einem initialen, oberen Schwellwert als Kanten gesetzt werden und mittels eines zweiten, niedrigeren Schwellwerts, Lücken zwischen diesen Pixeln geschlossen werden. [Nis+12] Auch dieser Algorithmus ist in OpenCV bereits implementiert und wird für den ersten Entwurf verwendet. Die Funktion bekommt das geladene und geglättete Bild sowie die beiden Hysterese-Schwellwerte übergeben. Diese ist auch in Code 4.1 gezeigt.

Code 4.1: Laden, Glätten eines Bildes und Durchführen der Kantenerkennung mit OpenCV

```

1 # load image (should be gray, so convert)
2 img = cv2.imread("./image.png")
3 img = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
4
5 # edge detection
6 img = cv2.GaussianBlur(img, (3,3), 1.5)
7 canny = cv2.Canny(img, 180, 40)

```

Wird dieser Code auf das Beispiel-Bild 4.3 angewendet und das Ergebnis des Canny-Edge-Detektors ausgegeben, ergibt sich Abbildung 4.4. Im Gegensatz zu Alternativen, wie einer reinen Gradientenbetrachtung, liefert der Canny-Edge-Detektor Kantenmarkierungen, hier in weiß, die nur einen Pixel breit sind. Dies ermöglicht die in den folgenden Unterkapiteln beschriebenen Schritte.

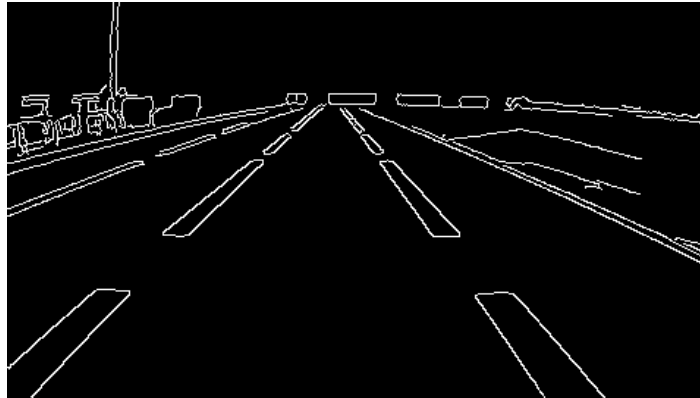


Abb. 4.4: Vom Canny-Edge-Detector gefundene Kanten

4.1.2 Klassifizierung der Kantenpixel

Nur die Identifikation von Pixeln als Kantenpixeln reicht nicht aus, um Linienmarkierungen zu erkennen. Menschen fällt es zwar leicht in Abbildung 4.4 die gesuchten Linien zu identifizieren, für den Algorithmus handelt es sich aber nur um eine scheinbar zufällige Ansammlung von weißen Pixeln. Es werden weitere Informationen benötigt.

Deshalb wird jedem Kantenpixel eine Klasse entsprechend seiner Orientierung zugeordnet. Um die Datenmenge gering und die Laufzeit schnell zu halten, werden lediglich die vier Klassen *Vertikal*, *Horizontal*, *Diagonal 1* und *Diagonal 2* verwendet. Zusätzlich wird noch die Richtungsinformation als Vorzeichen abgespeichert.

Die Klassifizierung erfolgt anhand der Gradientenorientierung eines Pixels. Dazu werden mit 3×3 Sobel-Kerneln die Gradienten d_x und d_y bestimmt. Mit der $\text{atan2}()$ Funktion kann aus diesen beiden Größen der Winkel des Gradientenvektors \vec{G} berechnet werden. Mit diesem Winkel kann nun entsprechend der Abbildung 4.5 die Klasse bestimmt werden. Dabei ist zu beachten, dass \vec{G} immer orthogonal auf der eigentlichen Kante steht. Deshalb ist die Klasse *Vertikal* auch auf der links-rechts Achse der Abbildung zu finden.

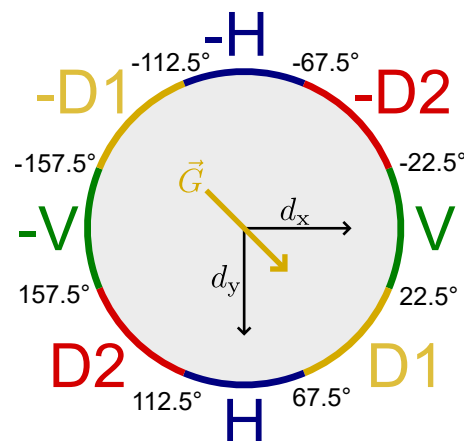


Abb. 4.5: Klassifizierung der Gradientenorientierung (nach [Hom21])

Die Klassifizierung wird in einer 8-Bit Variable abgespeichert, da so ein normales Graustufen-Bild als Datenstruktur verwendet werden kann. Jeder Klasse wird dabei ein Bit wie folgt zugeordnet:

Tab. 4.1: Zuordnung der Klassen zu Bits

Bit	Klasse
1	<i>Vertikal</i>
2	<i>Diagonal 1</i>
3	<i>Diagonal 2</i>
4	<i>Horizontal</i>
5	Vorzeichen-Bit

Um die Klassifizierung in Python durchzuführen, wird zuerst ein weiteres, leeres 8-Bit Bild mit identischer Größe angelegt. Dann wird erneut über alle Pixel des Bildes iteriert. Da allerdings die meisten Pixel schwarz und damit uninteressant sind, können diese direkt verworfen werden. Für alle verbleibenden, weißen Pixel wird die Klassifizierung durchgeführt.

Code 4.2: Schleife über das vom Canny-Edge-Detektor gelieferte Bild

```

1 for (u, v), e in np.ndenumerate(canny[1:-1, 1:-1]):
2     if not e:
3         continue
4     u += 1
5     v += 1

```

$$\begin{aligned}
 d_x &= \begin{bmatrix} p_{-1-1} & p_{-10} & p_{-11} \\ p_{0-1} & p_{00} & p_{01} \\ p_{1-1} & p_{10} & p_{11} \end{bmatrix} \circ \begin{bmatrix} 0 & 0 & 0 \\ -1 & 0 & +1 \\ 0 & 0 & 0 \end{bmatrix} \\
 d_y &= \begin{bmatrix} p_{-1-1} & p_{-10} & p_{-11} \\ p_{0-1} & p_{00} & p_{01} \\ p_{1-1} & p_{10} & p_{11} \end{bmatrix} \circ \begin{bmatrix} 0 & -1 & 0 \\ 0 & 0 & 0 \\ 0 & +1 & 0 \end{bmatrix}
 \end{aligned} \tag{4.1}$$

Dazu werden zuerst die Gradienten d_x und d_y ermittelt. Die 3×3 Pixelnachbarschaft des aktuellen Pixels wird dabei elementweise mit dem jeweiligen Sobel-Kernel multipliziert und die Summe der Ergebnismatrix gebildet (siehe Gleichung 4.1). Das Pythonpaket `numpy` stellt hierfür sehr hilfreiche Funktion zum Arbeiten mit Matrizen zur Verfügung. Dadurch lässt sich diese Operation in wenigen Zeilen durchführen, wie Code 4.3 zeigt.

Code 4.3: Bestimmung der Gradienten d_x und d_y

```

1 nh = img[u-1:u+2, v-1:v+2]
2 dx = np.sum(nh * SOBEL_X)
3 dy = np.sum(nh * SOBEL_Y)

```

Mit diesen wird nun die `atan2(dy, dx)` Funktion aufgerufen. Diese gibt einen Winkel in rad zurück, welcher zur besseren Nachvollziehbarkeit in Grad umgerechnet wird.

Durch eine Folge von Bedingungen wird nun die Klasse des aktuellen Pixels bestimmt. Zuerst wird das Vorzeichen ermittelt und im 5. Bit abgespeichert. Dies vereinfacht die folgenden Abfragen, da für die *Vertikal* und *Horizontal* Klasse der Betrag des Winkels ausreicht.

Ist die Klasse bestimmt, wird das entsprechende Bit des Pixels gesetzt. Die Umsetzung in Python ist in Code 4.4 gezeigt.

Code 4.4: Durchführen der Klassifizierung mittels des bestimmten Winkels

```
1 arc = atan2(dy, dx) / pi * 180
2
3 if arc < 0:
4     pixel_info[u, v] |= 0x10
5 arc = abs(arc)
6 if arc >= 157.5 or 22.5 > arc:
7     pixel_info[u, v] |= V
8 elif 22.5 <= arc < 67.5:
9     pixel_info[u, v] |= D1 if not pixel_info[u, v] else D2
10 elif 67.5 <= arc < 112.5:
11     pixel_info[u, v] |= H
12 elif 112.5 <= arc < 157.5:
13     pixel_info[u, v] |= D2 if not pixel_info[u, v] else D1
```

Wurde jeder Kantenpixel klassifiziert, ist der Vorgang beendet. Zur Veranschaulichung wurde ein Bild erstellt, in dem jeder Klasse und Vorzeichen eine eindeutige Farbe zugeordnet ist. So ist genau zu erkennen, welche Kanten derselben Klasse zugeordnet wurden. Dieses Bild ist in Abbildung 4.6 gezeigt.

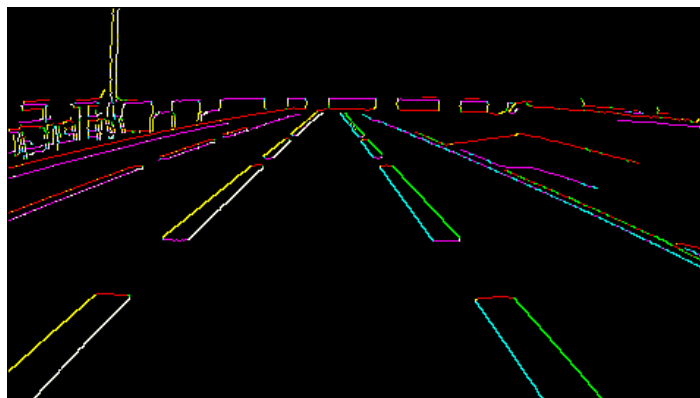


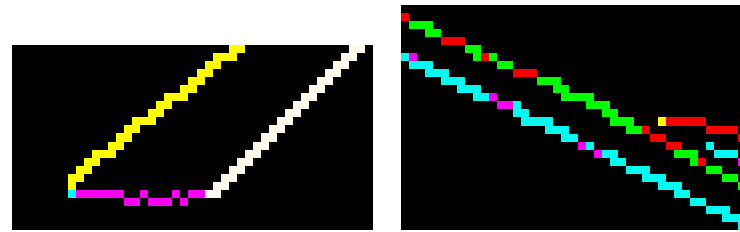
Abb. 4.6: Klassifizierte Kanten mit farblicher Markierung der unterschiedlichen Klassen (Farben sind nicht identisch mit Abbildung 4.5)

Genauigkeit der Klassifizierung

Die Klassifizierung erfolgt nicht immer völlig zuverlässig. Gut klassifizierte Kanten haben für die gesamte Länge der Linie dieselbe Klasse erhalten, wie es im vergrößerten Bildausschnitt Abbildung 4.7(a) zu sehen ist. Im Gegensatz dazu haben unzuverlässig klassifizierte Kanten mehrere Klassen in einem engen Bereich und wechseln häufig sogar mehrfach zwischen mehreren Klassen, wie das Beispiel in Abbildung 4.7(b) zeigt.

Durch Störungen und generell schlechtere Bildqualität in weiter von der Kamera entfernten Bildbereichen weisen vor allem die äußeren Linien viele dieser Ungenauigkeiten auf. Das führt dazu, dass die nachfolgende Logik viele einzelne, kleine Linien anstatt der vollständigen, durchgängigen Linie, erkennt.

Die zentralen Linienmarkierungen der eigenen Spur werden aber zuverlässig genug klassifiziert.



(a) Beispiel guter Kantenklassifizierung (b) Beispiel schlechter Kantenklassifizierung

Abb. 4.7: Vergleich von gut und schlecht klassifizierten Bildbereichen

4.1.3 Linienbildung

Mit den Informationen über die Kantenklasse kann nun der eigentliche Prozess der Linienerkennung erfolgen. Dieser gliedert sich in zwei Schritte, das Zusammenfassen von gleich klassifizierten Kantenpixeln zu durchgängigen Linien und das Zusammenfassen von Linien zu einer Fahrspurmarkierung.

Für den ersten Schritt ist es ein weiteres Mal nötig, über das gesamte Bild zu iterieren. Auch diesmal können wieder alle schwarzen Pixel übersprungen werden. Wird ein klassifiziertes Pixel gefunden, muss überprüft werden, ob es sich um ein Startpixel handelt. Startpixel sind Pixel, die keine Nachbarn in der ihrer Klasse entsprechenden Ursprungsrichtung haben. Zum Beispiel wäre ein Pixel der *Vertikal* Klasse ein Startpixel, wenn sich direkt oder diagonal über ihm keine weiteren Pixel derselben Klasse befinden.

Ist ein Startpixel gefunden, wird es für später abgespeichert. Nun wird der Linie zu ihrem Ende gefolgt. Dazu wird der nächste Nachbarpixel gesucht. Da die grobe Richtung entsprechend der Klasse bekannt ist, müssen hier nicht alle Nachbarpixel überprüft werden. Beim Beispiel mit der *Vertikal* Klasse müssten die Pixel direkt und diagonal unterhalb betrachtet werden. Existiert ein Nachbar wird dieser ausgewählt und der Prozess wiederholt. Dabei werden alle bereits besuchten Pixel aus dem Bild gelöscht, damit sich nicht noch einmal untersucht werden.

Code 4.5: Verfolgen einer Linie vom Start- zum Endpunkt

```

1 start = (u, v)
2 relevant_nh *= -1
3
4 while True:
5     pixel_info[u, v] = 0
6     for x, y in relevant_nh:
7         if e == pixel_info[u+x, v+y] & 0x0f:
8             u, v = u+x, v+y
9             break
10    else: # no more neighbours
11        break
12
13 l = Line(start, (u, v), info)
14 if l.length > 5:
15     lines.append(l)

```

Hat ein Pixel keine weiteren Nachbarn, ist er der Endpunkt dieser Linie. Start- und Endpunkt werden in ein Linienobjekt zusammengefasst und abgespeichert. Zusätzlich wird ebenfalls die Orientierungsklasse mit abgespeichert.

Mittels Start- und Endpunkt kann außerdem die Länge der Linie bestimmt werden. Da durch die in Unterunterabschnitt 4.1.2 beschriebenen Störungen viele kurze Linien gefunden werden, deren Berücksichtigung zu viel Rechenzeit in Anspruch nehmen würde, werden Linien unter einer Minimallänge von 5 Pixeln vernachlässigt.

Akzeptierte Linien werden ihrer Orientierung entsprechend in einzelnen Listen abgespeichert, so dass am Ende eine Liste für jede Orientierungsklasse entstanden ist.

Da ein Linienmarker immer aus einer linken und einer rechten Kante besteht, kann dieser durch Bilden von Linienpaaren gleicher Orientierung, aber unterschiedlichem Vorzeichen, die in geringem Abstand zueinander liegen, identifiziert werden.

Dazu werden die Elemente der entsprechenden Listen nacheinander miteinander verglichen, bis ein passendes Paar gefunden wurde. Ein Beispiel ist in Code 4.6 für Linienmarkierungen der Orientierung *Diagonal 1* gezeigt. Es wird für jeden Kandidaten aus der ersten Liste ein Partner in der zweiten gesucht. Ein solcher ist gefunden, wenn die Start- und Endpunkte beider Linien innerhalb bestimmter Bereiche liegen.

Code 4.6: Finden von Linienpaaren in Python

```
1 for a in left_D1_edges:
2     for b in right_D1_edges:
3         if (
4             (a.start[0] - 20) < b.start[0] < (a.start[0] + 20) and
5             a.start[1] < b.start[1] < (a.start[1] + 30) and
6             (a.end[0] - 20) < b.end[0] < (a.end[0] + 20) and
7             a.end[1] < b.end[1] < (a.end[1] + 20)
8         ):
9             markings_found.append(LineMarking(a,b, "left"))
10            right_D1_edges.remove(b)
11            break
```

Die gefundenen Linienpaare werden zu einem Linienmarker Objekt zusammengefasst. In diesem wird der Umriss bestehend aus den vier Linienpunkten abgespeichert. Außerdem wird der Mittelwert der beiden Start- und Endpunkte gebildet und somit die Mittellinie des Linienmarkers angenähert.

So gefundene Linienmarker lassen sich wieder im Beispiel-Bild markieren, wodurch sich Abbildung 4.8 ergibt. Wie man dort sehen kann, wurden die Linienmarker der eigenen linken und rechten Fahrspurbegrenzung erfolgreich identifiziert. Weitere Markierungen anderer Spuren konnten aufgrund der unzuverlässigen Klassifizierung nicht erkannt werden.

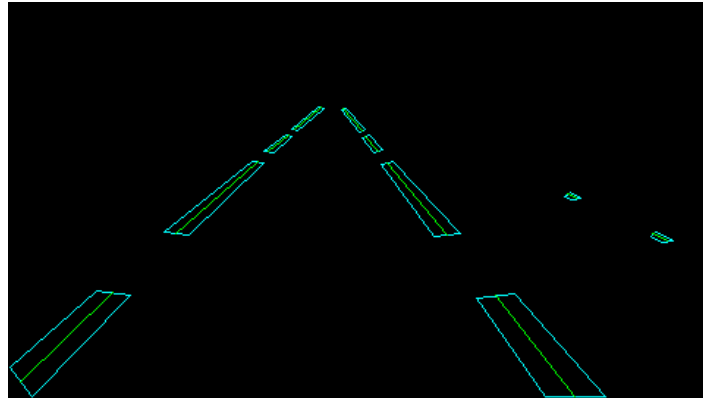


Abb. 4.8: Umrisse und Mittellinien der gefundenen Fahrspurmarkierungen

Komplexere Szenen

Nicht jede Situation, auf die der Roboter treffen kann, führt zu so guten Ergebnissen wie das gezeigte Beispiel. Daher sind in Abbildung 4.9 einige weitere Beispiele und die darin detektierten Markierungen im direkten Vergleich gezeigt.

In 4.9(a) sind die Linienbegrenzungen nahe am Fahrzeug durchgezogen und werden erst in größerer Entfernung gestrichelt. Hier war die Erkennung der durchgezogenen Teile sehr gut möglich, jedoch kam es zu einer unpräzisen Identifizierung im oberen Teil der rechten Linie, da sich zwei Liniensegmente unterschiedlicher Linien zu nahe aneinander befanden.

Bei der Szene 4.9(b) handelt es sich um eine Kurve. Dies erschwert die Erkennung deutlich, da eine durchgängige Klassifizierung einer Kante nicht garantiert ist. Daher sind auch nicht alle Spurmarkierungen identifiziert.

Der letzte Vergleich 4.9(c) zeigt eine Szene mit vollständig durchgezogener Linie. Dies ist aus dem Grund schwierig, dass die Wahrscheinlichkeit einer Störung durch die hohe Pixelanzahl sehr groß ist. Daher wurde die rechte Kante der Linie auch nicht durchgängig erkannt und der gefundenen Marker wirkt verzerrt.

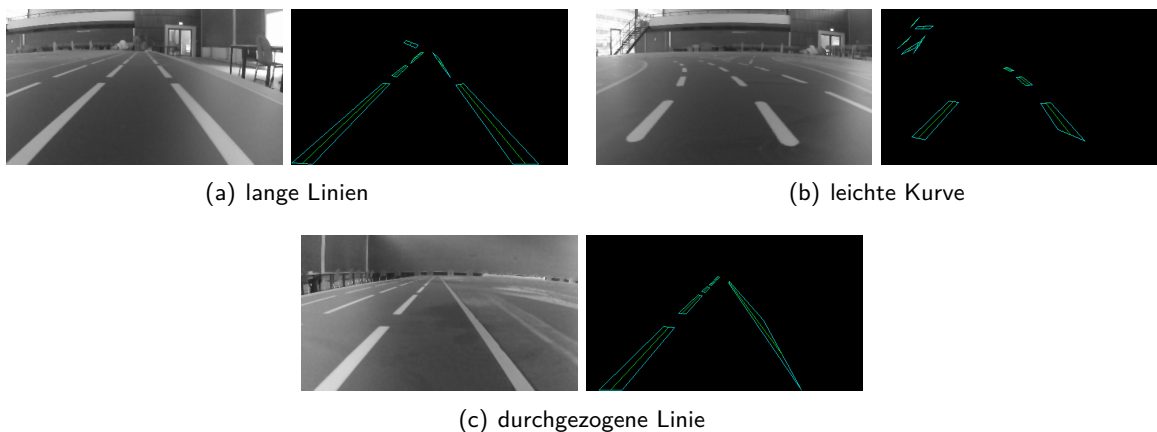


Abb. 4.9: Ergebnisse bei komplexeren Szenen im Vergleich

4.2 Implementierung in einer ROS Node

Um den Algorithmus zur Erkennung von Spurmarkierungen auf jedes Kamerabild anwenden zu können, wird er in einer Node umgesetzt. Da der Python-Code bereits auf einem leistungsfähigen Entwickler-PC Laufzeiten von $> 0,25\text{s}$ hat, wird die Programmiersprache C++ für die Node gewählt. Dadurch sollte sich die Performance deutlich verbessern.

4.2.1 Erstellung des Quellcodes

Die Beziehung der neuen Node zu den bestehenden Nodes wurde bereits in Abbildung 4.1 skizziert. Dort sieht man, dass für diese Node der Namen `lane_marker_detection` gewählt wird. Außerdem wird das Topic `/img/gray` von der Entzerrer-Node abonniert, um jedes Grauwert-Bild zu bekommen. Das Bild mit den eingezeichneten, detektierten Spurmarkierungen wird nach Durchlauf des Algorithmus auf dem eigenen Topic `/img/lanemarkings` veröffentlicht.

Beim Abonnieren des `/img/gray` Topics wird die Callback-Funktion `callback_image()` angehängt, sodass diese von ROS für jedes Bild aufgerufen und das Bild an sie übergeben wird. Da dieses in einem ROS eigenen Bild-Datentyp übergeben wird, OpenCV diesen aber nicht verwenden kann, ist es nötig das Bild zuerst einmal in einen anderen Datentyp umzuwandeln. Hierzu wird wieder das ROS-Paket `cv_bridge` und dessen Funktion `toCvCopy()` verwendet.

Kantenerkennung und Klassifizierung

Das so konvertierte Bild kann nun an die Funktion `edgeDetectionClassification()` übergeben werden, welche die Erkennung und Klassifizierung der Kanten durchführt. Wie bereits in der Python-Version wird wieder der Canny-Edge-Detektor aus OpenCV verwendet. Dadurch können die Parameter einfach übernommen werden. Der Code 4.7 zeigt den Aufruf. Das Ergebnis mit den detektierten Kanten wird in der Variable `canny` abgespeichert.

Code 4.7: Aufruf des Canny-Edge-Detektor in C++

```
1 cv::Mat canny;  
2 cv::Canny(image, canny, 180, 50);
```

Die Implementierung der Kantenklassifizierung in C++ läuft im Allgemeinen sehr ähnlich zur Python-Version, die wichtigsten Unterschiede sind die For-Schleifen für beide Dimensionen des Bildes, welche explizit einzeln verwendet werden müssen, und die Beachtung von Datentypen.

Die Erstellung des benötigten, leeren Bildes und die For-Schleifen sind in Code 4.8 zu sehen. Auch hier werden wieder aller leeren Pixel übersprungen.

Code 4.8: Initialisieren des leeren Bildes und iterieren über jenes.

```
1 cv::Mat classified_edges =  
    cv::Mat::zeros(cv::Size(image.cols, image.rows), CV_8U);  
2 for (int u=1; u < image.rows-1; u++) {  
3     for (int v=1; v < image.cols-1; v++) {  
4         if ( ! canny.at<uint8_t>(u,v) )  
5             continue;
```

Mittels Sobel die Gradienten zu bestimmen, wirkt in C++ deutlich komplizierter, da hier vieles manuell gemacht werden muss, was in Python von `numpy` erledigt wurde. Mit zwei For-Schleifen wird über die 3×2 Pixelnachbarschaft iteriert, die Elemente mit dem Kernel multipliziert und aufsummiert, wie in Code 4.9 zu sehen.

Code 4.9: Bestimmung der Gradienten mittels Sobel

```

1 uint8_t e;
2 int dx=0, dy=0;
3 for (int y=0; y<3; y++) {
4     for (int x=0; x<3; x++) {
5         e = image.at<uint8_t>(u+y-1,v+x-1);
6         dx += SOBEL_X[y*3+x] * e;
7         dy += SOBEL_Y[y*3+x] * e;
8     }
9 }

```

Die eigentliche Klassifizierung ist praktisch identisch zur Python-Version. Lediglich die Überprüfung der Winkelbereiche ist etwas langwieriger, da nicht wie in Python auf einen Wertebereich überprüft werden kann. Die Codierung der Klassen Erfolg wieder über die einzelnen Bit des Bytes der einzelnen Pixel. Die Umsetzung ist in Code 4.10 dargestellt.

Code 4.10: Bestimmung der Gradienten mittels Sobel

```

1 double arc = atan2(dy,dx) / 3.1415 * 180.0;
2
3 uint8_t clsif = 0;
4 if (arc < 0)
5     clsif = 0x10;
6 arc = fabsf(arc);
7
8 if (arc<=22.5f || arc>157.5f ) {
9     clsif |= V;
10 } else if ( 67.5f<=arc && arc<112.5f ) {
11     clsif |= H;
12 } else if (( !clsif && arc<67.5f )||( clsif && 112.5f<=arc )) {
13     clsif |= D1;
14 } else if (( clsif && arc<67.5f )||( !clsif && 112.5f<=arc )) {
15     clsif |= D2;
16 }
17 classified_edges.at<uint8_t>(u,v) = clsif;

```

Das Ergebnisbild mit den klassifizierten Pixeln wird von der Funktion zurückgegeben und dort weiterverarbeitet.

Linienbildung

Mit dem klassifizierten Bild kann nun dieselbe Methodik zur Identifizierung zusammenhängender Linien wie in Python angewendet werden. Allerdings ist in C++ das Definieren und Testen der relevanten Pixelnachbarschaft nicht so übersichtlich möglich wie in Python. Daher müssen viele lange if-Bedingungen verwendet werden, welche in den folgenden Codebeispielen zur Übersichtlichkeit verkürzt sind.

Begonnen wird wieder mit einer doppelten for-Schleife über das gesamte Bild, wie in Code 4.11 zu sehen. Dabei werden wieder alle leeren Pixel vernachlässigt.

Code 4.11: for-Schleifen über alle klassifizierten Pixel

```
1 for (int u=1; u < classified_edges.rows-1; u++) {
2     for (int v=1; v < classified_edges.cols-1; v++) {
3         uint8_t clsif_org = classified_edges.at<uint8_t>(u,v);
4         if ( ! clsif_org )
5             continue;
```

Für jeden Pixel wird wieder überprüft, ob er ein Startpixel ist. Genau wie in Python ist hierfür wieder der Klasse entsprechend eine Pixelnachbarschaft relevant. Ist dort ein Nachbar gleicher Klasse vorhanden, wird mit dem nächsten Pixel weitergemacht. Dies ist in Code 4.12 gezeigt.

Code 4.12: Überprüfen, ob ein Pixel ein Startpixel ist

```
1 // get only classification without direction:
2 uint8_t clsif = 0x0f & clsif_org;
3 bool has_neighbour = false;
4 switch (clsif) {
5     case V:
6         if ( /* any of the relevant neighbours */ )
7             has_neighbour = true;
8     case D1:
9         if ( /* any of the relevant neighbours */ )
10            has_neighbour = true;
11     case H:
12         if ( /* any of the relevant neighbours */ )
13            has_neighbour = true;
14     case D2:
15         if ( /* any of the relevant neighbours */ )
16            has_neighbour = true;
17 }
18 if ( has_neighbour )
19     continue;
```

Ist ein Startpixel gefunden, wird er gespeichert und wieder so lange der nächste Nachbar ausgewählt, bis kein Nachbar mehr vorhanden ist. Dann ist die gesamte Linie nachverfolgt. Dabei werden alle besuchten Pixel aus dem Bild gelöscht. Siehe dazu Code 4.13.

Code 4.13: Linie verfolgen, bis keine Nachbarn mehr existieren

```

1 pnt start(v,u);
2 // follow line to its end
3 do {
4     classified_edges.at<uint8_t>(u,v) = 0;
5     has_neighbour = false;
6     switch (clsif) {
7         case V:
8             if ( /* any neighbour exists */ )
9                 // u+-;v+-; change u,v to new coordinates
10                has_neighbour = true;
11         case D1:
12             if ( /* any neighbour exists */ )
13                 // u+-;v+-; change u,v to new coordinates
14                has_neighbour = true;
15         case D2:
16             if ( /* any neighbour exists */ )
17                 // u+-;v+-; change u,v to new coordinates
18                has_neighbour = true;
19         }
20         case H:
21             if ( /* any neighbour exists */ )
22                 // u+-;v+-; change u,v to new coordinates
23                has_neighbour = true;
24     }
25 } while ( has_neighbour );

```

Die so gefundenen Linien werden wieder auf ihre Länge überprüft und als Objekte abgespeichert. Dabei werden einzelne Listen für jede Klasse angelegt, sodass diese später verglichen werden können.

Besonders wichtig ist es, dass im Gegensatz zu Python die Schleifenvariablen *u* und *v* manuell wieder auf die Startkoordinaten zurückgesetzt werden müssen.

Paarbildung

Auch die Bildung von Linienpaaren aus einer rechten und linken Linie erfolgt analog zu Python. Hier gibt es auch keine signifikanten Unterschiede in der Umsetzung, wie in Code 4.14 zu sehen ist.

Auch hier wird mit den gefundenen Paaren wieder ein Linienmarker-Objekt erzeugt und zur oberen Nutzung abgespeichert.

Da noch nicht klar ist, wie genau die gefundenen Daten potenziell zukünftigen Prozessen zu Verfügung gestellt werden sollen, werden diese zurzeit nicht veröffentlicht. Stattdessen wird analog zur Python-Implementierung ein Bild mit eingezeichneten Spurmarkern erzeugt und als visuelles Ergebnis zur Verfügung gestellt. Dieses kann unter dem Topic `/img/temp` abgerufen und live angeschaut werden.

Code 4.14: Linie verfolgen, bis keine Nachbarn mehr existieren

```

1 for(const Line& a : left_D1_edges) {
2     for(auto it = right_D1_edges.begin(); it != right_D1_edges.end();
3         it++ ) {
4         const Line b = *it;
5         if (
6             ( a.start.y - 10 < b.start.y && b.start.y < a.start.y +
7               10 ) &&
8             ( a.start.x < b.start.x && b.start.x < a.start.x + 25 ) &&
9             ( a.end.y - 10 < b.end.y && b.end.y < a.end.y + 10 ) &&
10            ( a.end.x < b.end.x && b.end.x < a.end.x + 25 )
11        ) {
12            markings_found.push_back(LineMarking(a,b, D1));
13            left_D1_edges.erase(it);
14            break;
15        }
16    }
17 }

```

4.2.2 Performance Betrachtung

Mit dieser zusätzlichen Node ist es erneut interessant, wie sich diese auf die Performance auswirkt. Aus Abschnitt 3.1 ist bereits die Performance mit laufender Kamera- und Entzerrer-Node bekannt. Zum Vergleich wurde wieder die Systemauslastung mit dem Programm `jtop` aufgenommen und die Durchlaufzeit nach Erhalt eines Bildes gemessen.

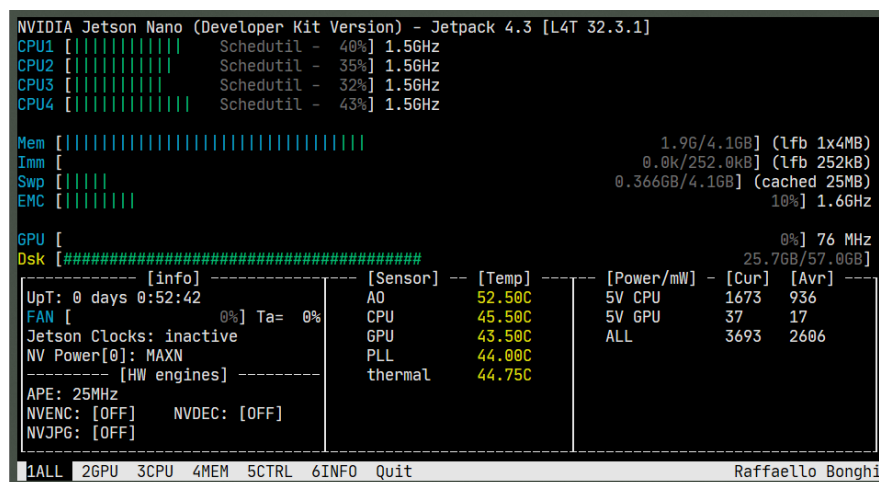


Abb. 4.10: CPU Auslastung des JetBots mit laufender Kamera, Entzerrung und Markierungserkennung

Ein Screenshot von `jtop` ist in Abbildung 4.10 abgebildet. Mit der zusätzlichen neuen Node ist die CPU Auslastung nur geringfügig auf ca. 37,5 % gestiegen, auch dies ist dank starker Fluktuation in der CPU Nutzung praktisch vernachlässigbar.

Die Laufzeit-Performance der C++ implementiert ist wie erwartet deutlich besser als bei der Python-Version. Vom Erhalt des Bildes bis zur Veröffentlichung des Ergebnisses mit den gefundenen Linienmarkierungen dauert es $\approx 3,46$ ms.

Tab. 4.2: Gemessene Laufzeit bei 10 Durchläufen der Callback-Funktion

Durchlauf Nr.	gemessene Laufzeit
1	5,050 ms
2	5,050 ms
3	5,186 ms
4	6,229 ms
5	5,329 ms
6	5,231 ms
7	6,762 ms
8	5,631 ms
9	5,309 ms
10	4,862 ms

4.3 Optimierung durch eigene Implementierung des Canny-Edge-Detectors

Im Folgenden wird untersucht, ob eine eigene Implementierung des Canny-Edge-Detektor einen Vorteil gegenüber der Verwendung von OpenCV bietet. Da die Bibliotheksfunktionen viel potenziell nicht erforderlich Zusatzfunktionen und Schutzmechanismen enthalten, welche Zeit und Performance benötigen und für deren Verwendung eine Datenumwandlung notwendig ist, kann eine eigene Implementierung hier Vorteile bieten.

Außerdem führt der Canny-Edge-Detektor intern bereits eine Sobel-Filterung durch und es ließe sich die Klassifizierung dort direkt mit durchführen. Das eliminiert die Notwendigkeit zweimal über das Bild zu iterieren.

4.3.1 Anpassung des Quellcodes

Da ohne OpenCV das Bild nicht mehr in eine praktische Matrixform umgewandelt wird, muss ein eigener Datentyp definiert werden. Dieser speichert die Daten im selben Format wie der verwendete ROS Datentyp, sodass keine Umwandlung notwendig ist. Der Datentyp ermöglicht lediglich den Zugriff auf die Daten über x und y Koordinaten.

Code 4.15 zeigt die Umsetzung mit dem Namen `Image`. Initialisiert werden kann eine Instanz entweder nur mit Höhe und Breite, wodurch ein neues, leeres Bild erzeugt wird, oder mit einem existierenden Bild, wobei dessen Daten übernommen werden.

Um auf die Bilddaten zuzugreifen, wird der Operator `[]` überladen, sodass ein Punkt bestehend aus einer x und einer y Koordinate übergeben werden kann. Das entsprechende Pixel wird dann zurückgegeben oder beschrieben.

Code 4.15: Eigener Datentyp zum Ablegen von Bildern

```

1 class Image {
2     public:
3         std::vector<uint8_t> data;
4         int width; int height;
5
6         Image(int width, int height): data(width*height, 0),
7             width(width), height(height) {};
8         Image(sensor_msgs::Image img): data(img.data), width(img.width),
9             height(img.height) {};
10
11         uint8_t& operator[](pnt const& p) { return data[p.y*width + p.x];
12             }
13         const uint8_t& operator[](pnt const& p) const { return
14             data[p.y*width + p.x]; }
15 };

```

Die Callback-Funktion `callback_image()` verläuft vollständig analog zur Implementierung mit OpenCV. Der Unterschied liegt in der Funktion `edgeDetectionClassification()`. Diese führt nun die drei Schritte des Canny-Edge-Detektor durch: Gradienten Bestimmung, unterdrücken von nicht-lokalen-Maxima und Hysterese-Grenzwertbildung (siehe [Can86]).

Zusätzlich wird außerdem die Klassifizierung durchgeführt. Da im ersten Schritt die Sobel-Gradienten ohnehin berechnet werden, können hierüber ohne viel Zusatzaufwand auch die Gradientenwinkel mit bestimmt werden. Der Code 4.16 zeigt diesen Schritt.

Code 4.16: Bestimmung der Sobel-Gradienten im Canny-Edge-Detektor

```

1 // Compute gradient magnitude and orientation
2 for (int u=1; u < image.height-1; u++) {
3     for (int v=1; v < image.width-1; v++) {
4         int dx=0, dy=0;
5         for (int y=0; y<3; y++){
6             for (int x=0; x<3; x++){
7                 dx += SOBEL_X[y*3+x] * image[pnt(v+y-1,u+x-1)];
8                 dy += SOBEL_Y[y*3+x] * image[pnt(v+y-1,u+x-1)];
9             }
10        }
11        gards[pnt(v,u)] = static_cast<int>(sqrt(dx*dx+dy*dy));
12        agls[pnt(v,u)] = static_cast<float>(atan2(dy,dx) /
13            3.1415*180.0);
14    }
15 }

```


Im zweiten Schritt werden nun zusätzlich zur Unterdrückung von Pixeln, welche kein lokales Maximum sind, auch die Klassen bestimmt und abgespeichert. Da dieser Code sehr viele verschachtelte und gedoppelte if-Abfragen aufweist, wird er in Code 4.17 vereinfacht gezeigt.

Code 4.17: Klassifizierung und Unterdrückung von Nicht-Lokalen-Maxima

```

1 // Non-maximum suppression and classification
2 for (int u=1; u < image.height-1; u++) {
3     for (int v=1; v < image.width-1; v++) {
4         int grad = gradients[pnt(v,u)];
5         int arc = angles[pnt(v,u)];
6
7         if (arc < 0)
8             uint8_t negative = 0x10;
9         arc = fabsf(arc);
10
11         if ( /* is in a specific angle range */ ) {
12             canny_edges[pnt(v,u)] = negative /* CLASSIFICATION */;
13             if ( /* is not local maximum */ )
14                 gradients[pnt(v,u)] = 0;
15         }
16     }
17 }

```

Der letzte Schritt der Hysterese-Grenzwertbildung wurde für diese Anwendung zu einem simplen Grenzwert vereinfacht. Dies erzeugt ausreichend gute Ergebnisse.

Wichtig ist außerdem, dass im Gegensatz zu einem herkömmlichen Canny-Edge-Detektor kein binarisiertes Bild, mit nur völlig weißen oder völlig schwarzen Pixelwerten, zurückgegeben wird. Stattdessen enthalten alle detektierten Kantenpixel bereits den Wert, der ihre Klasse repräsentiert.

Code 4.18: Grenzwertbildung und Ergebnissrückgabe

```

1 // thresholding
2 uint8_t T1 = 40;
3 for (int u=1; u < image.height-1; u++)
4     for (int v=1; v < image.width-1; v++) {
5         pnt p(v,u);
6         canny_edges[p] = gradients[p] < T1 ? 0 : canny_edges[p];
7     }
8 return canny_edges;

```

4.3.2 Performance Betrachtung

Die wichtigste Frage ist nun, wie sich diese Implementierung im Gegensatz zum Ansatz mit OpenCV verhält. Dazu wurden wieder die CPU-Auslastung mittels `jtop` erfasst und die Durchlaufzeit des Algorithmus für jedes Bild gemessen.

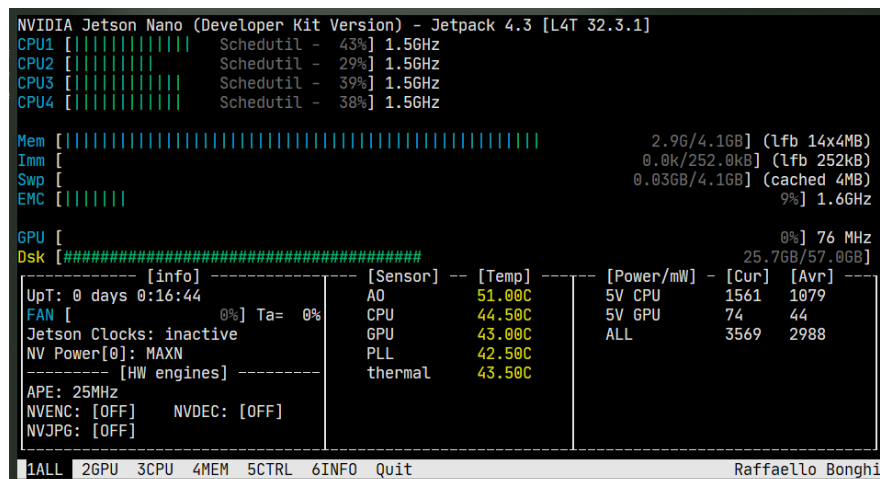


Abb. 4.11: CPU Auslastung des JetBots mit laufender Kamera, Entzerrung und Markierungserkennung mit eigener Implementierung

Wie aus Abbildung 4.11 abzulesen ist, ist die CPU-Auslastung mit durchschnittlichen 37,75% exakt identisch zur Implementierung mit OpenCV. Das ist wenig überraschend, da der Algorithmus Grunde immer noch dieselbe Arbeit verrichtet.

Die Messung der Laufzeit zeigt allerdings, dass die eigene Implementierung deutlich weniger performant ist. Mit einer Durchlaufzeit von $\approx 22,58$ ms ist diese Version um mehr als Faktor 6 größer. Hier gleichen die Vorteile durch das Kombinieren von Einzelschritten die Leistungsfähigkeit einer stark optimierten Bibliotheksfunktion also leider nicht einmal annähernd aus.

Tab. 4.3: Gemessene Laufzeit bei 10 Durchläufen der Callback-Funktion

Durchlauf Nr.	gemessene Laufzeit
1	22,266 ms
2	22,064 ms
3	22,127 ms
4	22,596 ms
5	23,370 ms
6	22,278 ms
7	24,207 ms
8	22,547 ms
9	22,097 ms
10	22,201 ms

5 Fazit

Die Arbeit hatte zum Ziel, die auf dem JetBot existierende Bilderfassung zu verbessern und mit einer Erkennung von Fahrspurmarkierungen in Echtzeit auszustatten.

Durch eine intrinsische Kalibrierung der Kamera und das Erstellen einer Node, welche diese Kalibrierung anwendet, konnte das von der Kamera erhaltene Bild deutlich verbessert und vorhandene Verzerrungen korrigiert werden. Insbesondere die radiale Verzerrung, die gerade Linien wie zum Beispiel Fahrspurmarkierungen, im Bild gekrümmt erscheinen lässt, konnte annähernd vollständig entfernt werden. Dies verbesserte die Voraussetzungen für die nachfolgenden Schritte. Die dabei gemessene Durchlaufzeit pro Bild von ≈ 4 ms garantierte eine Echtzeitfähigkeit.

Außerdem wurde die Erkennung der Fahrspurmarkierungen erfolgreich implementiert. Der Algorithmus wurde in der Programmiersprache Python entwickelt und getestet, wobei insbesondere die einfache Syntax und gute Debuggmöglichkeiten das Vorgehen vereinfacht haben. Da Python aber nicht performant genug war, um die Echtzeitanforderung zu erfüllen, wurde die eigentliche Implementierung in C++ umgesetzt. Dadurch wurde die Laufzeit des Programms deutlich beschleunigt und die Echtzeitfähigkeit gewährleistet.

Die Erkennung von Fahrspurmarkierungen wurde mittels Kantenerkennung durch einen Canny-Edge-Detektor und Klassifizierung der einzelnen Kantenpixel entsprechend ihrer Orientierung umgesetzt. So ließen sich aus den Kanten erfolgreich zusammenhängende Linien einer Orientierung ableiten und deren Start- und Endpunkte abspeichern. Durch Paarbildung zwischen rechten und linken Linien konnten Rückschlüsse über einzelne Fahrspurmarkierungen gezogen werden. Die vier Punkte eines Linienpaares wurden dann als Kontur abgespeichert und die Mittellinie berechnet. Zur Veranschaulichung wurden die Konturen in ein leeres Bild eingezeichnet und dieses ebenfalls veröffentlicht. Da die gemessene Durchlaufzeit pro Bild bei $\approx 3,5$ ms lag, war ein direkter Vergleich zwischen Originalbild und gefundenen Konturen in Echtzeit möglich.

Eine weitere Optimierung durch eine eigene Implementierung des Canny-Edge-Detektor und Kombinieren von diesem mit dem Klassifizierungsschritt, war nicht möglich. Hierdurch wurden deutlich schlechtere Laufzeiten erzielt.

6 Ausblick

Die grundsätzliche Erkennung von Linienmarkierungen im Bilddatenstrom konnte erfolgreich umgesetzt werden. Jedoch bestehen in mehreren Bereichen des Programmes noch Weiterentwicklungsmöglichkeiten. Dadurch könnte die Erfassungsgenauigkeit insbesondere in schwierigen Situationen erhöht und weitere Informationen gewonnen werden.

Mehr Orientierungsklassen

Derzeit wird bei der Klassifizierung der Kantenpixel lediglich in vier unterschiedliche Klassen eingeteilt. Das kann insbesondere bei Kurven zu dem Problem führen, dass die Klassifizierung innerhalb einer Kante plötzlich wechselt und so zwei unterschiedliche Linien auf derselben Kante erfasst werden.

Durch mehr Klassen und Berücksichtigung von benachbarten Klassen in der Linienverfolgung ließen sich potenziell auch gekrümmte Linien verfolgen.

Reduzierung von Falschklassifizierungen

Derzeit werden vor allem bei weiter von der Kamera entfernten Linien, also vor allem den Spurmarkierungen benachbarter Spuren, viele Pixel fehlklassifiziert. Wie in Abbildung 4.7(b) gezeigt, wird eine Kante durch solche Pixel unterbrochen und eine durchgängige Linienverfolgung nicht möglich ist.

Hier könnten zuerst einmal Ansätze zum Verbinden von sehr eng zusammenliegenden Linien Abhilfe schaffen. Aber eine Verbesserung der Genauigkeit bei der Klassifizierung wäre ebenfalls denkbar, vor allem da dies bereits teilweise durch den vorherigen Punkt mit abgedeckt ist.

Datenübergabe an weitere Prozesse

Da diese Arbeit als Grundlage für weitere Prozesse dient, z.B. zum autonomen Einhalten der Spur beim Fahren, müsste die genaue Methode zur Datenübergabe festgelegt werden. Zurzeit wird ein weiteres Bild mit den eingezeichneten Fahrspurmarkierung veröffentlicht, weitere Prozesse benötigen aber die erzeugten Objekte mit den enthaltenen Informationen.

Diese liegen vor und können über die definierte Headerdatei genutzt werden, es findet aber noch keine Übertragung mittels ROS statt. Diese müsste bei Bedarf eingerichtet werden.

Weiters Optimierungspotenzial

Der Versuch den C++ Quellcode durch eine eigene Implementierung des Canny-Edge-Detektor zu optimieren, hat leider keine nutzbaren Ergebnisse erbracht. Grundsätzlich sind die gemachten Überlegungen aber gültig und mit tieferem Verständnis der Programmiersprache wäre eine erfolgreiche Optimierung möglich. Da OpenCV quelloffen ist, wäre es auch denkbar, deren Quellcode direkt zu verwenden.

Literatur

- [Can86] John Canny. „A Computational Approach to Edge Detection“. eng. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8.6 (1986), S. 679–698. DOI: 10.1109/TPAMI.1986.4767851.
- [Fra+09] Barbara Frank u. a. *Robotics 2 - Camera Calibration*. eng. 2009. URL: <http://ais.informatik.uni-freiburg.de/teaching/ws09/robotics2/pdfs/rob2-08-camera-calibration.pdf> (besucht am 19.07.2022).
- [GW+21] Prof. Dr. L. Gusig, Prof. Dr. N. Waldt u. a. *Projekt - Autonomes Fahren*. Techn. Ber. Hochschule Hannover, 15. Feb. 2021.
- [Han11] Tobias Hanning. *High Precision Camera Calibration*. eng. 1. Aufl. Vieweg+Teubner, 2011. 225 S. DOI: 10.1007/978-3-8348-9830-2.
- [Hom21] Prof. Dr.-Ing. Hanno Homann. *Script: Vorlesung Bildverarbeitung*. ger. Moodle, Open Source Lernplattform der Hochschule Hannover. Stand: WiSe 2021/22. 2021.
- [Jet22] JetBot-community. *JetBot - full online Documentation*. eng. 2022. URL: <https://jetbot.org/> (besucht am 01.09.2022).
- [KK15] Gurveen Kaur und Dinesh Kumar. „Lane Detection Techniques: A Review“. eng. In: *International Journal of Computer Applications* 112.10 (2015), S. 4–8. URL: <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.695.9070&rep=rep1&type=pdf> (besucht am 01.09.2022).
- [Lop+05] A. Lopez u. a. „Detection of lane markings based on ridgeness and RANSAC“. eng. In: *Proceedings. 2005 IEEE Intelligent Transportation Systems, 2005*. 2005, S. 254–259. DOI: 10.1109/ITSC.2005.1520139.
- [Lu+19] Jingyan Lu u. a. „A Lane Detection Method Based on a Ridge Detector and Regional G-RANSAC“. eng. In: *Sensors* 19 (Sep. 2019), S. 4028. DOI: 10.3390/s19184028.
- [Mat22] MathWorks. *What Is Camera Calibration? - MATLAB & Simulink - MathWorks Deutschland*. eng. 2022. URL: <https://de.mathworks.com/help/vision/ug/camera-calibration.html> (besucht am 19.07.2022).
- [Nis+12] Alfred Nischwitz u. a. *Computergrafik und Bildverarbeitung. Band I: Computergrafik*. ger. 3. Aufl. Studium. Wiesbaden: Vieweg+Teubner, 2012. 1 online resource. DOI: 10.1007/978-3-8348-8323-0.
- [Nvi22] Nvidia. *Jetson Nano Developer Kit*. eng. 2022. URL: <https://developer.nvidia.com/embedded/jetson-nano-developer-kit> (besucht am 03.09.2022).
- [Ope22a] OpenCV. *Homepage - Open Source Computer Vision Library*. eng. 27. Juli 2022. URL: <https://opencv.org/> (besucht am 12.08.2022).
- [Ope22b] OpenCV. *Python-Tutorials: Camera Calibration*. eng. 17. Juli 2022. URL: https://docs.opencv.org/4.6.0/dc/dbb/tutorial_py_calibration.html (besucht am 09.08.2022).
- [Rob18] Open Robotics. *ROS 2 Documentation - Introduction*. eng. 8. Aug. 2018. URL: <http://wiki.ros.org/ROS/Introduction> (besucht am 14.08.2022).

- [Rob22] Open Robotics. *Homepage - Robot Operating System*. eng. 2022. URL: <https://www.ros.org/> (besucht am 15.08.2022).
- [Siv+21] S.A. Sivasankari u. a. „Lane detector for driver assistance systems“. eng. In: *Materials Today: Proceedings* (2021). DOI: 10.1016/j.matpr.2021.03.649.
- [Spa22] Sparkfun. *Assembly Guide for SparkFun JetBot AI Kit V2.0*. eng. 2022. URL: <https://learn.sparkfun.com/tutorials/assembly-guide-for-sparkfun-jetbot-ai-kit-v20> (besucht am 01.09.2022).
- [Sze11] Richard Szeliski. *Computer Vision. Algorithms and Applications*. eng. SpringerLink Bücher. London: Springer, 2011. DOI: 10.1007/978-1-84882-935-0.
- [TLL21] Jigang Tang, Songbin Li und Peng Liu. „A review of lane detection methods based on deep learning“. eng. In: *Pattern Recognition* 111 (März 2021), S. 107623. DOI: 10.1016/j.patcog.2020.107623.
- [WHF05] Chun-Che Wang, Shih-Shinh Huang und Li-Chen Fu. „Driver assistance system for lane detection and vehicle recognition with night vision“. In: *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2005, S. 3530–3535. DOI: 10.1109/IR0S.2005.1545482.
- [Wik22] Wikipedia. *Verzeichnung — Wikipedia, die freie Enzyklopädie*. ger. 2022. URL: <https://de.wikipedia.org/w/index.php?title=Verzeichnung&oldid=223280815> (besucht am 12.07.2022).
- [Wil22a] Jan Wille. *VideoDriveWorkspace Wiki: example data sets - intrinsische kalibrierung down4*. 15. Juni 2022. URL: https://lab.it.hs-hannover.de/p9r-rxm-u1/videodrive_ws/-/wikis/uploads/6c853b3f41964eccd6671954a07ad5ed/intrinsicCalibration_down4.zip (besucht am 19.06.2022).
- [Wil22b] Jan Wille. *VideoDriveWorkspace Wiki: example data sets - straßen leer down4*. 15. Juni 2022. URL: https://lab.it.hs-hannover.de/p9r-rxm-u1/videodrive_ws/-/wikis/uploads/c8e09d2f28a2279b9b76cd899c383cc9/stra%C3%9Fen_leer_down4.zip (besucht am 19.06.2022).
- [Zha+21] Youcheng Zhang u. a. „Deep Learning in Lane Marking Detection: A Survey“. eng. In: *IEEE Transactions on Intelligent Transportation Systems* PP (Apr. 2021), S. 1–17. DOI: 10.1109/TITS.2021.3070111.

Abbildungsverzeichnis

2.1	Generischer Ablauf von Fahrspurerkennung (nach [KK15])	2
2.2	SparkFun JetBot AI Kit V2.1 [Spa22]	5
2.3	CPU Auslastung des JetBots ohne ROS	6
2.4	CPU Auslastung mit laufender Kamera und ROS-Core	7
2.5	Ansicht von oben auf die Fahrbahnfläche [GW+21]	7
3.1	Unkalibriertes Kamerabild mit tonnenförmiger Verzerrung	8
3.2	Darstellung der optischen Verzerrungen (nach [Wik22])	9
3.3	Probleme in der Ausrichtung von Sensor und Linse (nach [Mat22])	9
3.4	Schachbrett-Kalibriermuster mit markierten inneren Kreuzungen	10
3.5	Schritte der Bild-Entzerrung	12
3.6	Beziehungen der Entzerrer-Node zu bestehenden Nodes	13
3.7	CPU Auslastung des JetBots mit laufender Kamera und Entzerrer-Node	16
4.1	Zusammenhang der Fahrspurmarkierung-Erkennungs-Node mit den bestehenden Nodes	17
4.2	Ablauf des Algorithmus zur Erkennung von Fahrspurmarkierungen	18
4.3	Beispiel-Bild an dem der Ablauf demonstriert wird	19
4.4	Vom Canny-Edge-Detector gefundene Kanten	20
4.5	Klassifizierung der Gradientenorientierung (nach [Hom21])	20
4.6	Klassifizierte Kanten mit farblicher Markierung der unterschiedlichen Klassen (Farben sind nicht identisch mit Abbildung 4.5)	22
4.7	Vergleich von gut und schlecht klassifizierten Bildbereichen	23
4.8	Umrisse und Mittellinien der gefundenen Fahrspurmarkierungen	25
4.9	Ergebnisse bei komplexeren Szenen im Vergleich	25
4.10	CPU Auslastung des JetBots mit laufender Kamera, Entzerrung und Markierungserkennung	30
4.11	CPU Auslastung des JetBots mit laufender Kamera, Entzerrung und Markierungserkennung mit eigener Implementierung	34

Tabellenverzeichnis

3.1	Gemessene Laufzeit bei 10 Durchläufen der Callback-Funktion	16
4.1	Zuordnung der Klassen zu Bits	21
4.2	Gemessene Laufzeit bei 10 Durchläufen der Callback-Funktion	31
4.3	Gemessene Laufzeit bei 10 Durchläufen der Callback-Funktion	34

Codeverzeichnis

3.1	Definition der Größe des Kalibrierusters	10
3.2	Initialisierung von Variablen für die Kalibrierung	11
3.3	Finden und Verarbeiten der Kalibrierbilder	11
3.4	Abspeichern der gefundenen Bildpunkte	11
3.5	Ermitteln der Kalibrierwerte mittels OpenCV	12
3.6	Berechnen des Reprojektions-Fehlers	13
3.7	Einlesen der Kalibrierungsergebnisse aus einer YAML-Datei	14
3.8	Bestimmen der Pixel-Mappings zu Entzerrung	14
3.9	Vereinfachte Version der Callback-Funktion zur Durchführung der Entzerrung . .	15
4.1	Laden, Glätten eines Bildes und Durchführen der Kantenerkennung mit OpenCV	19
4.2	Schleife über das vom Canny-Edge-Detektor gelieferte Bild	21
4.3	Bestimmung der Gradienten d_x und d_y	21
4.4	Durchführen der Klassifizierung mittels des bestimmten Winkels	22
4.5	Verfolgen einer Linie vom Start- zum Endpunkt	23
4.6	Finden von Linienpaaren in Python	24
4.7	Aufruf des Canny-Edge-Detektor in C++	26
4.8	Inizialisieren des leeren Bildes und iterieren über jenes.	26
4.9	Bestimmung der Gradienten mittels Sobel	27
4.10	Bestimmung der Gradienten mittels Sobel	27
4.11	for-Schleifen über alle klassifizierten Pixel	28
4.12	Überprüfen, ob ein Pixel ein Startpixel ist	28
4.13	Linie verfolgen, bis keine Nachbarn mehr existieren	29
4.14	Linie verfolgen, bis keine Nachbarn mehr existieren	30
4.15	Eigener Datentyp zum Ablegen von Bildern	32
4.16	Bestimmung der Sobel-Gradienten im Canny-Edge-Detektor	32
4.17	Klassifizierung und Unterdrückung von Nicht-Lokalen-Maxima	33
4.18	Grenzwertbildung und Ergebnissrückgabe	33