

Bachelorarbeit

Video-basierte Fahrspurerkennung von mobilen Robotern

Jan Wille 1535115

09.05.2022 – 08.08.2022

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

kommt als letztes

Schlüsselwörter: lane-detection

Inhaltsverzeichnis

Abstract	IV
Inhaltsverzeichnis	V
1 Einleitung	1
1.1 Problemstellung	1
1.2 Aufgabenstellung	1
1.3 Inhalt der Arbeit	1
2 Stand der Technik	2
2.1 Lochkamera Modell	2
2.2 Deep learning	2
2.3 OpenCV	2
2.4 Das Robot Operation System	2
2.5 Der JetBot Roboter	2
2.6 Aufgebaute Anlage	2
3 Kamera Kalibrierung	4
3.1 Intrinsische Kalibrierung	4
3.1.1 Radiale Verzerrung	4
3.1.2 Tangentiale Verzerrung	5
3.2 Durchführung der intrinsischen Kalibrierung	6
3.2.1 Python Script zur Durchführung der Kalibrierung	6
3.2.2 Anwenden der Kalibrierung in einem ROS Node	9
3.3 Extrinsische Kalibrierung	10
4 Fahrspurerkennung	11
4.1 Bild Vorbereitung	11
4.1.1 Bildausschnitt auf relevanten Bereich reduzieren	11
4.1.2 Verbesserung der Bildqualität	12
4.2 Canny-Edge-Detector	12
4.3 Orientierungserfassung mittels Sobel	12
4.4 Linienbildung	12
5 Ausblick	13
Abbildungsverzeichnis	14
Tabellenverzeichnis	15
Codeverzeichniss	16

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, soll damit 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 so 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 unter ROS 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

Überblick, jedes Kapitel vorstellen

2 Stand der Technik

2.1 Lochkamera Modell

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \quad (2.1)$$

$$p = K \cdot T[RT] \quad (2.2)$$

Stand der Technik: Was ist Spruerkennung? wie wird das zurzeit immer gemacht?? ein absatz klassisch kantenbasiert
ein absatz Kategorien von Deep learning ansätzen

2.2 Deep learning

Was ist das?
Warum hier nicht?
Was ist ungeeignet?

2.3 OpenCV

Das Open-Source-Projekt OpenCV (kurz für *Open Source Computer Vision Library*) ist eine Sammlung von Softwaremodule 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 standard Bibliotheken, 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 simple in eine Hardwarenahe Programmiersprache übersetzt werden können.

Quelle?? Reicht das About von opencv.org?

Vergleich mit eigener Implementierung
evtl. Performance Vergleich

2.4 Das Robot Operation System

2.5 Der JetBot Roboter

2.6 Aufgebaute Anlage

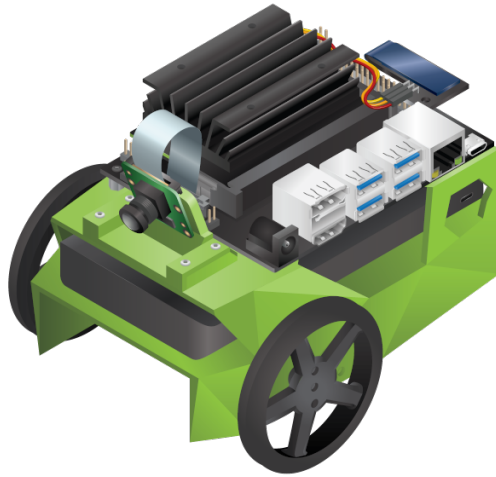


Abb. 2.1: Render eines möglichen Aufbaus [jetbot:github-doc]



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

3 Kamera Kalibrierung

Damit der später beschriebene Fahrspurerkennung möglichst zuverlässig funktioniert und möglichst reproduzierbar ist, wird eine Kalibrierung 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, die in der Realität natürlich alle parallel verlaufen, im Bild aber gekrümmt aussehen.

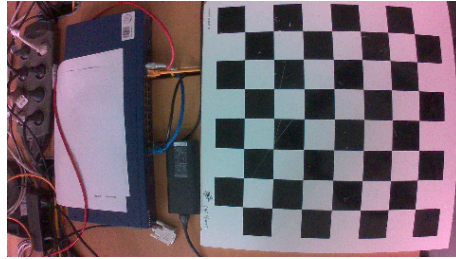


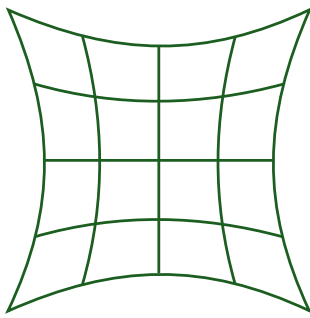
Abb. 3.1: Unkalibriertes Kamerabild mit tonnenförmiger Verzeichnung

3.1.1 Radiale Verzerrung

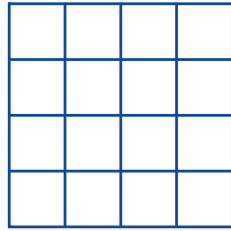
Die erste mögliche Art der Verzerrung ist die radiale Verzerrung. Diese ist die auffälligste Art der Verzerrung und wird häufig auch *Fischaugen Effekt* genannt. Bedingt durch die Brechung des Lichtes an den Kanten der Blende und der Linse entsteht eine Ablenkung der Lichtstrahlen in der Kamera, die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

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_2 und k_3 die Verzerrungskoeffizienten. Theoretisch existieren noch weitere Koeffizienten, aber in der Praxis haben sich die ersten drei als ausreichend herausgestellt. [Hanning:highPrecisionCamCalibration]

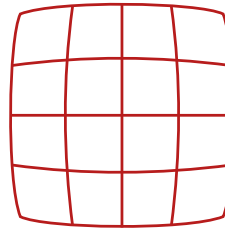
$$\begin{aligned}x_{\text{distorted}} &= x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \\y_{\text{distorted}} &= y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)\end{aligned}\tag{3.1}$$



(a) Kissenförmige Verzerrung



(b) Verzerrungsfreies Bild



(c) Tonnenförmige Verzerrung

Abb. 3.2: Darstellung der optischen Verzerrung (nach [wiki:LinsenVerzerrung])

3.1.2 Tangentiale Verzerrung

Die tangentiale Verzerrung entsteht durch kleine Ausrichtungsfehler im Linsensystem. Dadurch liegt die Linse nicht perfekt in der Bildebene und der Bildmittelpunkt sowie die Bildausrichtung können leicht verschoben sein.

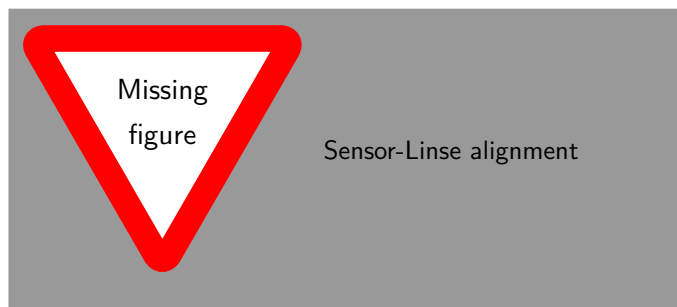


Abb. 3.3: Probleme in der Ausrichtung von Sensor und Linse (nach [Matlab:CameraCalibration])

Mathematisch wird diese Verzerrung durch den folgenden Zusammenhang beschrieben. [Hanning:highPrecisionCamCalibration]

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

Durch beide Verzerrungsarten zusammen werden also durch fünf Parameter beschrieben, die sogenannten Verzerrungskoeffizienten. Historisch begründet wird dabei k_3 an das Ende geschrieben, da dieses 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 also mindestens fünf Punkte gefunden werden, von denen die Weltkoordinaten und die Bildkoordinaten 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. [OpenCV:CameraCalibration]

In der Praxis werden 2D-Muster verwendet, um Punktpaare zu bestimmen. Da sich alle Punkte dieser Muster in einer Ebene befinden, kann der Ursprung der Weltkoordinaten in eine Ecke des Musters gelegt werden, sodass die Z-Koordinate keine Relevanz mehr hat und wegfällt. [uniFreiburg:rob2-CamCalibration]

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 die den Vorgang einfach und wiederholbar zu machen. Als Vorlage für dieses dient die Anleitung zur Kamera Kalibrierung aus der OpenCV Dokumentation [OpenCV:CameraCalibration].

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

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 verwendet werden. 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 Kreuzungspunkten. Ein normales Schachbrett hat beispielsweise 8×8 Felder, aber nur 7×7 interne Kreuzungen. Zur Verdeutlichung sind die Kreuzungspunkte des Verwendeten Kalibriermuster in Abbildung 3.4 grün markiert.

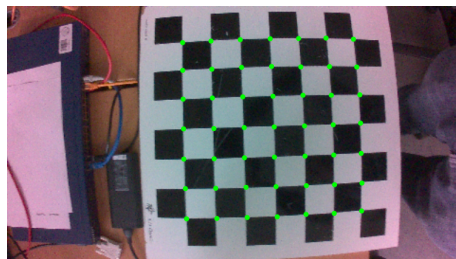


Abb. 3.4: Schachbrett Kalibriermuster mit markierten inneren Kreuzungen

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 Kalibriermuster

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

Entsprechend der Anleitung [OpenCV:CameraCalibration] werden benötigte Variablen initialisiert (siehe Code 3.2).

Code 3.2: Initialisierung von Variablen für die Kalibrierung

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

Nun werden alle im aktuellen Ordner befindlichen Bilder eingelesen und in einer Liste abgespeichert. Jedes Listenelement wird eingelesen und in ein Schwarzweißbild umgewandelt. Dieses wird dann an die OpenCV Funktion `findChessboardCorners()` übergeben, welche die Kreuzungspunkten findet und zurückgibt.

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

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 Verzerrungskoeffizienten D_{coeff} zurück. **[OpenCV: Camera Calibration]** 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 URL https://lab.it.hs-hannover.de/p9r-rxm-u1/videodrive_ws/-/wikis/uploads/6c853b3f41964eccd6671954a07ad5ed/intrinsicCalibration_down4.zip ab-

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),
5         (-1,-1), criteria)
6     imgpoints.append(corners)
```

Code 3.5: Ermitteln der Kalibrierwerte mittels OpenCV

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

gelegt. Damit ergeben sich die folgenden Kalibrierungsergebnisse.

$$\begin{aligned}k_1 &= -0,42049309612684654 \\k_2 &= 0,3811654512587829 \\p_1 &= -0,0018273837466050299 \\p_2 &= -0,006355252159438178 \\k_3 &= -0,26963105010742416 \\K &= \begin{pmatrix} 384,65 & 0 & 243,413 \\ 0 & 384,31 & 139,017 \\ 0 & 0 & 1 \end{pmatrix}\end{aligned}$$

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 *Region of interes*, 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.

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 Kalibrierung Ergebnisse berechneten Weltkoordinaten. Der Mittelwert aller Abweichungen in allen verwendeten Bilder gibt den Reprojektions-Fehler für den ganzen Kalibriervorgang an.

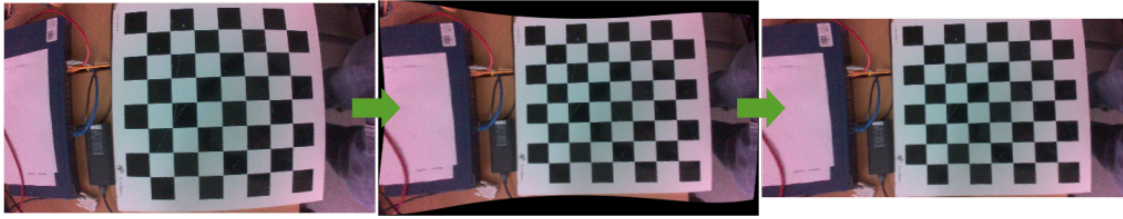


Abb. 3.5: Schritte der intrinsischen Kalibrierung

Der Code 3.7 zeigt die Berechnung mittels von OpenCV zur Verfügung gestellten Funktionen und den zuvor ermittelten Kalibrierdaten. Für jeden Satz an theoretischen Weltkoordinaten des Kalibrieramusters 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, was genau genug für diesen Anwendungsfall ist.

3.2.2 Anwenden der Kalibrierung in einem ROS Node

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

Code 3.7: Initialisierung der Entzerrer Node

```
1 ros::init(argc, argv, "img_undistort");
2
3 ros::NodeHandle nh;
4 ros::NodeHandle private_nh("~");
```

Runtime: ≈ 6 ms

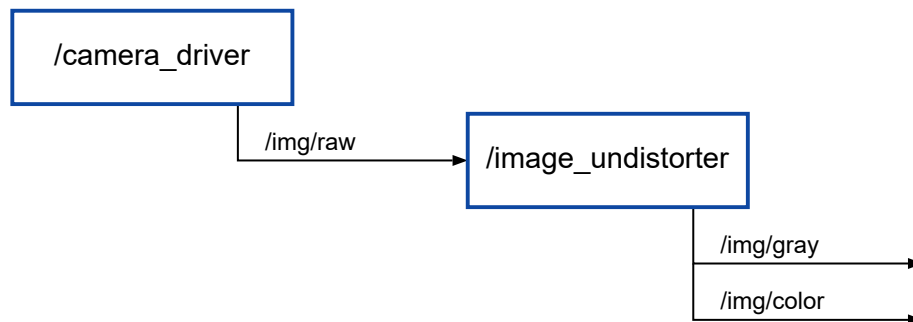


Abb. 3.6: Beziehungen der entzerrer Node zu bestehenden Nodes

3.3 Extrinsische Kalibrierung

4 Fahrspurerkennung

Das folgende Kapitel beschreibt den Ablauf und die Umsetzung des Fahrspur-Erkennungs-Prozesses.

Es werden Codesegmente zu den einzelnen Abschnitten erklärt und an einem beispielhaften Bild werden die Ergebnisse einzelnen Schritte gezeigt.

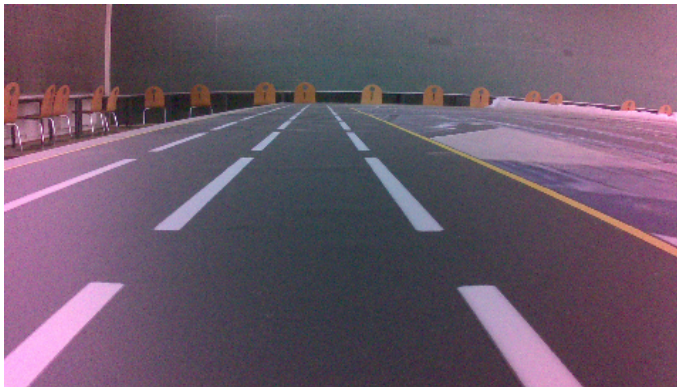


Abb. 4.1: Ein Beispiel Bild an dem der Ablauf demonstriert wird

4.1 Bild Vorbereitung

4.1.1 Bildausschnitt auf relevanten Bereich reduzieren

Der obere Bildbereich enthält wenig für die Fahrspurerkennung relevante Informationen. Hier gibt es viele Hintergrundobjekte die zu Störungen führen können und Fahrspurmarkierungen sind durch die perspektivische Verzeichnung zu nahe aneinander um erkennbar zu sein.

Daher wird das obere drittel des Bildes verworfen und der Bildausschnitt reduziert. In Python kann hierzu einfach eine neue Ansicht beginnen beim Index $\frac{1}{3} \cdot \text{Height}$ erstellt werden. Das ist auch im folgenden Codebeispiel gezeigt:

Code 4.1: Test caption

```
1 img = img[int(h/3):, :]
```

Abbildung 4.2 zeigt das Beispielbild vor und nach dem Beschnitt.

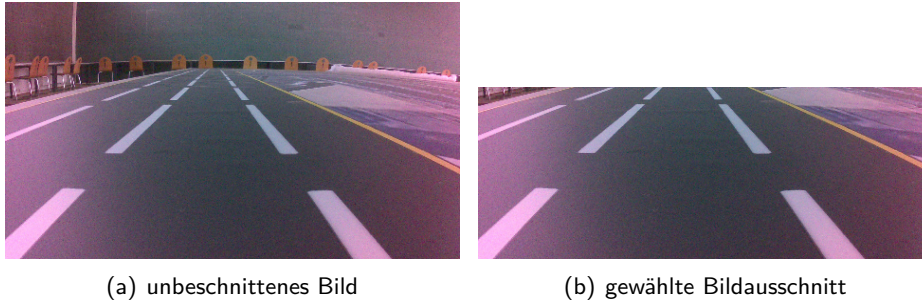


Abb. 4.2: Vergleich von Ursprungsbild und Bildausschnitt



Abb. 4.3: Durch bilaterales Filtern verbessertes Bild

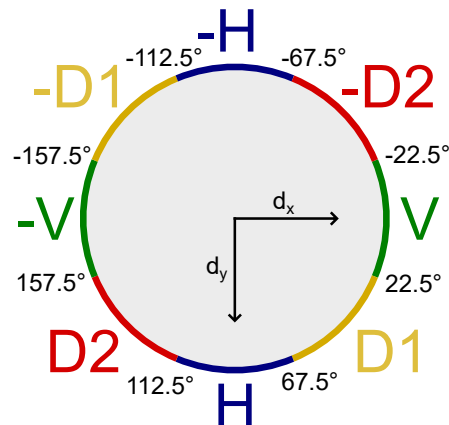


Abb. 4.4: Klassifizierung der Gradientenorientierung (nach [Homann:VorlesungBildverarbeitung])

4.1.2 Verbesserung der Bildqualität

4.2 Canny-Edge-Detector

4.3 Orientierungserfassung mittels Sobel

4.4 Linienbildung

5 Ausblick

Abbildungsverzeichnis

2.1	Render eines möglichen Aufbaus [jetbot:github-doc]	3
2.2	SparkFun JetBot AI Kit V2.1 [jetbot:Sparkfun]	3
3.1	Unkalibriertes Kamerabild mit tonnenförmiger Verzeichnung	4
3.2	Darstellung der optischen Verzerrung (nach [wiki:LinsenVerzerrung])	5
3.3	Probleme in der Ausrichtung von Sensor und Linse (nach [Matlab:CameraCalibration])	5
3.4	Schachbrett Kalibriermuster mit markierten inneren Kreuzungen	6
3.5	Schritte der intrinsischen Kalibrierung	9
3.6	Beziehungen der Entzerrer-Node zu bestehenden Nodes	10
4.1	Ein Beispielbild an dem der Ablauf demonstriert wird	11
4.2	Vergleich von Ursprungsbild und Bildausschnitt	12
4.3	Durch bilaterales Filtern verbessertes Bild	12
4.4	Klassifizierung der Gradientenorientierung (nach [Homann:VorlesungBildverarbeitung])	12

Tabellenverzeichnis

Codeverzeichnis

3.1	Definiteion der Größe des Kalibriermuster	6
3.2	Initialisierung von Variablen für die Kalibreirung	7
3.3	Finden un Verarbeiten der Kalibrierbilder	7
3.4	Abspeichern der Gefundenen Bildpunkte	8
3.5	Ermitteln der Kalibrierwerte mittels OpenCV	8
3.6	Berechnen des Reprojektions-Fehlers	9
3.7	Initialisierung der Entzerrer Node	9
4.1	Test caption	11