

Bachelorarbeit

Video-basierte Fahrspurerkennung von mobilen Robotern

Jan Wille 1535115

09.05.2022 – 04.09.2022

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

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 Operating System	2
2.5 Der JetBot Roboter	3
2.5.1 Performance Baseline	3
2.6 Aufgebaute Anlage	3
3 Kamera Kalibrierung	5
3.1 Intrinsische Kalibrierung	5
3.1.1 Radiale Verzerrung	5
3.1.2 Tangentiale Verzerrung	6
3.2 Durchführung der intrinsischen Kalibrierung	7
3.2.1 Python Script zur Durchführung der Kalibrierung	7
3.2.2 Anwenden der Kalibrierung in einer ROS Node	10
3.3 Extrinsische Kalibrierung	13
4 Fahrspurerkennung	15
4.1 Konzeptionierung in Python	15
4.1.1 Kantenerkennung mittels Canny-Edge-Detektor	17
4.1.2 Klassifizierung der Kantenpixel	18
4.1.3 Linienbildung	21
4.1.4 Performance Betrachtung	21
4.2 Implementierung in eine ROS Node	21
4.2.1 Performance Betrachtung	21
4.3 Optimierung durch eigene Implementierung des Canny-Edge-Detectors	21
4.3.1 Performance Betrachtung	21
5 Ausblick	22
Abbildungsverzeichnis	23
Tabellenverzeichnis	24

Codeverzeichniss

25

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 klas-
sisch 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. Für mehr Informationen kann die Webseite des Projektes [[OpenCV:homepage](#)] besucht werden.

Vergleich mit eigener Implementierung
evtl. Performance Vergleich

2.4 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 verschiedenste bestehende Bibliotheksfunktionen für die Verwendung heruntergeladen werden können. Dabei handelte es sich um verschiedenste Anwendungen,

angefangen Treiben, über fertige, direkt anwendbare Algorithmen bis zu Nutzer nahen Steueroberflächen und sogar (Lern-)Spiele. Die Webseite des Projektes **[ROS:homepage]** 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. **[ROS:whatsROS]** Für diese Arbeit wird die aktuelle Distribution ROS Noetic verwendet.

beschreiben:

wie Arberitet ROS

Was sind Nodes, Topics, Messages

was nutzen wir hier?

2.5 Der JetBot Roboter

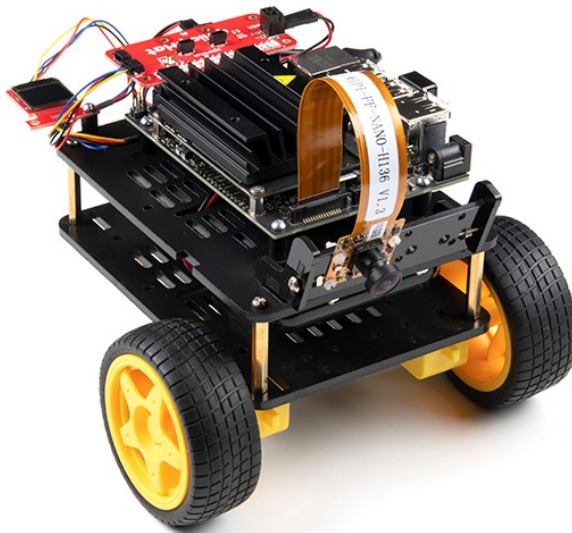


Abb. 2.1: SparkFun JetBot AI Kit V2.1 **[jetbot:Sparkfun]**

2.5.1 Performance Baseline

Baseline Auslastung ohne irgendwelche laufenden Prozesse $\approx 8\%$.

Mit ROS-Core und laufendem Kameratreiber $\approx 38\%$

2.6 Aufgebaute Anlage

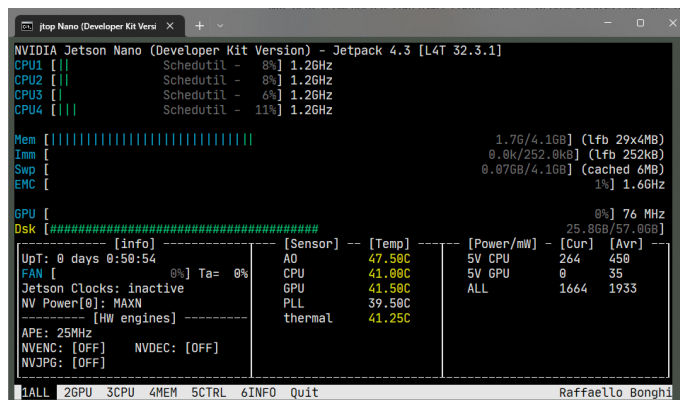


Abb. 2.2: CPU Auslastung des JetBots ohne ROS

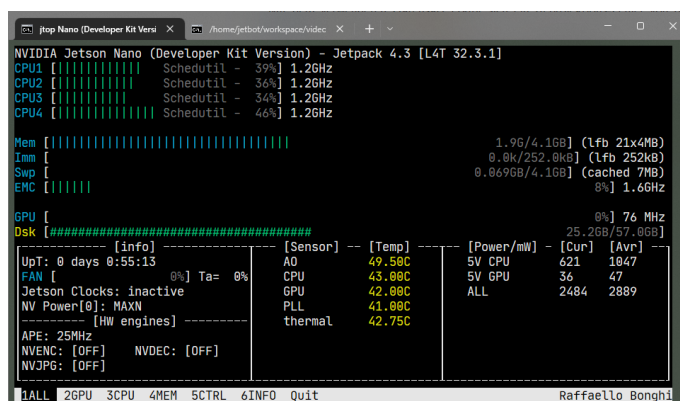


Abb. 2.3: CPU Auslastung mit laufender Kamera und ROS-Core

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.

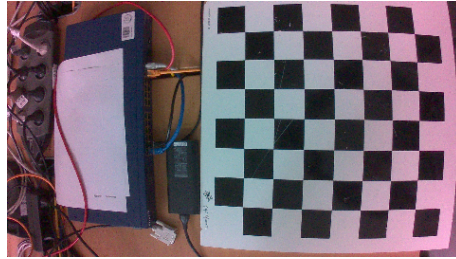


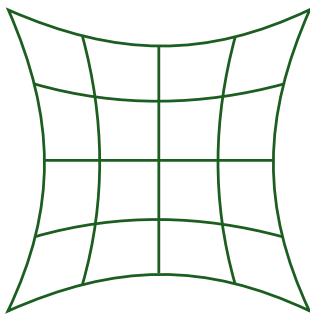
Abb. 3.1: Unkalibriertes Kamerabild mit tonnenförmiger Verzeichnung

3.1.1 Radiale Verzerrung

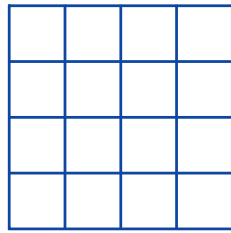
Die erste mögliche Art der Verzerrung ist die radiale Verzerrung. Diese ist die auffälligste Art der Verzerrung und wird häufig auch *Fischaugen Effekt* genannt. Bedingt durch die Brechung des Lichtes an den Kanten der Blende und der Linse entsteht eine Ablenkung der Lichtstrahlen in der Kamera, die 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]

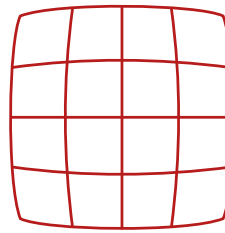
$$\begin{aligned}x_{\text{distorted}} &= x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \\y_{\text{distorted}} &= y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)\end{aligned}\tag{3.1}$$



(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.

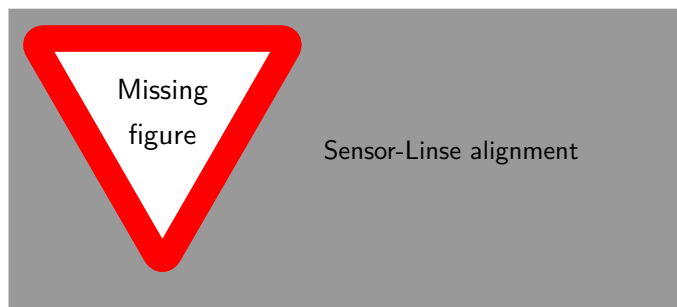


Abb. 3.3: Probleme in der Ausrichtung von Sensor und Linse (nach [Matlab:CameraCalibration])

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 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 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.

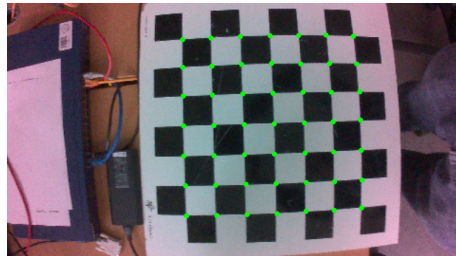


Abb. 3.4: Schachbrett Kalibriermuster mit markierten inneren Kreuzungen

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,
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
8   images.
9 objpoints = [] # 3d point in real world space
10 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 Kreuzungspunkte 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,
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 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.

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(corners2)
```

Jetzt kann die eigentliche Kalibrierung mittels der OpenCV Funktion `calibrateCamera()` durchgeführt werden. Diese nimmt die zuvor erstellten Listen von Objektkoordinaten und Bild-

punkten und löst damit die in Abschnitt 3.1 beschriebenen Gleichungen. Als Ergebnis liefert sie die Kameramatrix K und die Verzerrungskoeffizienten D_{coeff} zurück. **[OpenCV:CameraCalibration]**

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)
```

Der gesamte Code wird nun auf einen Datensatz von Bilder angewandt, um die Ergebnisse für den vorliegenden Roboter zu erhalten. Der Datensatz ist auf dem GitLab Server unter der **[git:dataset-kalibrierung]** abgelegt. 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 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.

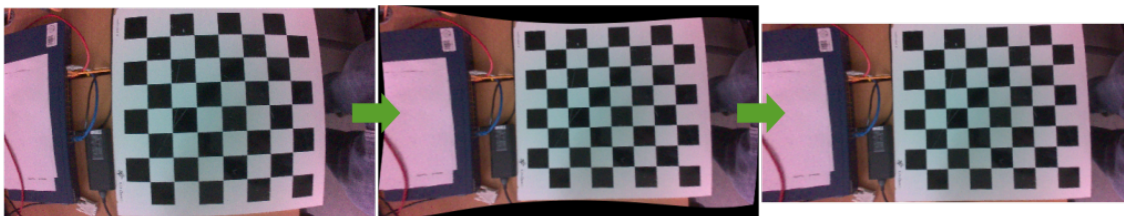


Abb. 3.5: Schritte der intrinsischen Kalibrierung

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.

Der Code 3.6 zeigt die Berechnung mittels von OpenCV zur Verfügung gestellten Funktionen und den zuvor ermittelten Kalibrierdaten. Für jeden Satz an theoretischen Weltkoordinaten des Kalibrierusters 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 einer 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.

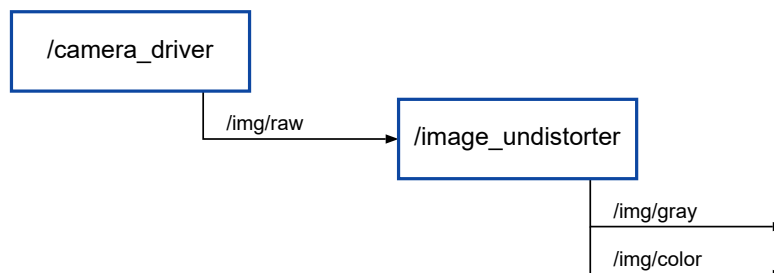


Abb. 3.6: Beziehungen der entzerrer Node zu bestehenden Nodes

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.

Code hierzu?

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 aus der Konfigurationsdatei gelesen. Code 3.7 zeigt den 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"]
   .as<std::vector<double>>();
8 cv::Mat distortion ( distortion_YAML );
9
10 // read camera matrix and convert to OpenCV matrix
11 auto cameraMatrix_YAML = camera_config["intrinsic"]["matrix"]
   .as<std::vector<std::vector<double>>>();
12 cv::Mat cameraMatrix = toMat( cameraMatrix_YAML );
13
14 // read image size
15 cv::Size imageSize(
16     full_config["images"]["size"]["width"].as<int>(),
17     full_config["images"]["size"]["height"].as<int>()
18 );

```

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 zusätzlich 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 vorhandenen Informationen werden grundsätzlich erhalten und bei Bedarf kann das Programm einfach angepasst werden, um die vollständigen Bilder zu veröffentlichen.

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 Co-

Code 3.8: Bestimmen der Pixel-Mappings zu Entzerrung

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

de 3.9 zeigt eine vereinfachte Version der Implementierung, ohne Umwandlung in ein Schwarz-weißbild 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 wird linear interpoliert.

Das Erhalten Bild wird auf die ROI reduziert und unter dem Topic `/img/color` veröffentlicht. Außerdem wird ein Schwarz-Weiß 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 undistoredImage;
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, undistoredImage, rectifyMapX,
        rectifyMapY, cv::INTER_LINEAR);
9
10    // crop relevant section from image
11    undistoredImage = undistoredImage(ROI);
12
13    // publish images
14    cv_bridge::CvImage colorImage(std_msgs::Header(), "rgb8",
        undistoredImage);
15    pub_colorImage->publish(colorImage.toImageMsg());
16 }
```

Performance Betrachtung

Da diese Node eine Grundlagenfunktion darstellt und parallel zu jeder anderen Anwendungen 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 ja für jedes Bild durchlaufen wird, gemessen.

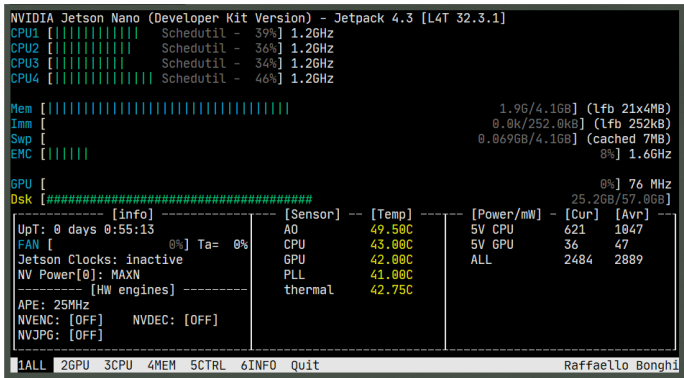


Abb. 3.7: CPU Auslastung des JetBots mit laufender Kamera und entzerrer Node

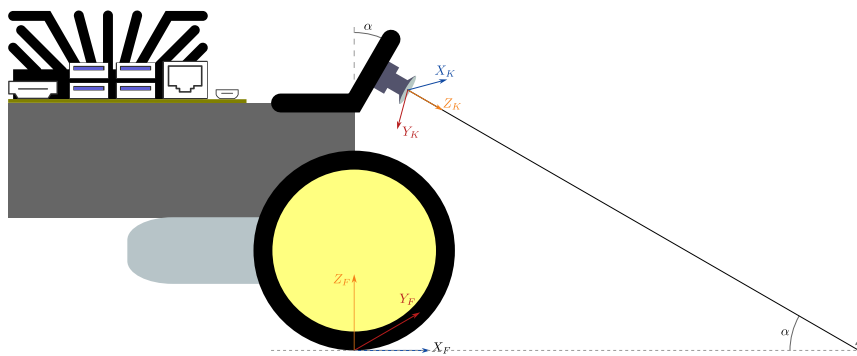
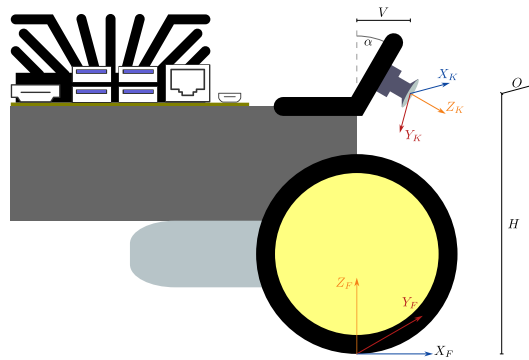
Der jtop Screenshot in Abbildung 3.7 zeigt die CPU Nutzung bei aktivem ROS-Core, Kamera-treiber und der neu erstellten Entzerrer Node. Die durchschnittliche CPU Auslastung liegt bei ungefähr 38,75 %, ist also nur sehr geringfügig höher als die in Unterabschnitt 2.5.1 gemessene Grundauslastung ohne die neue Node. Um die Laufzeit der Node zu bestimmen wird die aktuelle Zeit wie sie von der Funktion `ros::Time::now()` zurückgegeben wird verwendet. Die aktuelle Zeit beim Start der Callback-Funktion wird abgespeichert. Nach Durchlauf der Funktion wird erneut die aktuelle Zeit bestimmt und die Differenz in Sekunden als Debug-Nachricht ausgegeben. Die Laufzeit der Node wird über einige Zeit gemittelt. Dabei ergibt sich eine Laufzeit von $\approx 3,9$ 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	3,970 ms
6	4,086 ms
7	4,025 ms
8	3,897 ms
9	3,753 ms
10	4,096 ms

ist die Tabelle überhaupt sinnvoll?

3.3 Extrinsische Kalibrierung



4 Fahrspurerkennung

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 best möglichst Performance zu erhalten.

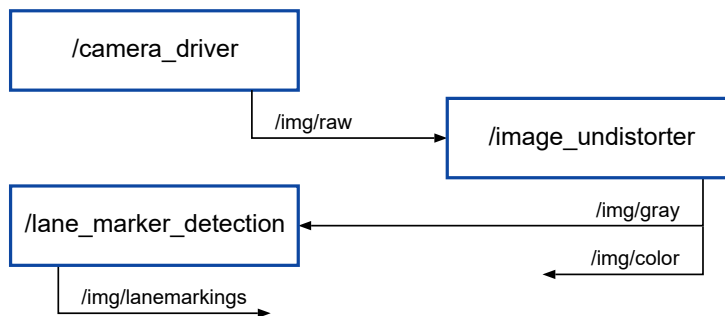


Abb. 4.1: Zusammenhang der Fahrspurmarkierung-Erkennungs Node mit den bestehenden Nodes

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

4.1 Konzeptionierung in Python

Die Entwicklung und Konzeptionierung des Algorithmus Erfolg in Python, da diese Sprache nicht kompiliert werden muss, was das Testen beschleunigt, und generell einfacher zu verwenden ist.

Der Algorithmus lässt sich in mehrer Einzelschritte aufteilen und wird daher in den folgen Unterkapitel beschreiben. Zur Übersicht ist aber der gesamte Ablauf in Abbildung 4.2 vereinfacht skizziert. Angefangen wird dort mit dem Erhalten des Bildes, womit sowohl manuelles laden eines Beispieldes, 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 **[git:dataset-strassen]** abgelegt. In Abbildung 4.3 ist eines dieser Bilder gezeigt, mit dem im Folgenden die Einzelschritte demonstriert werden.

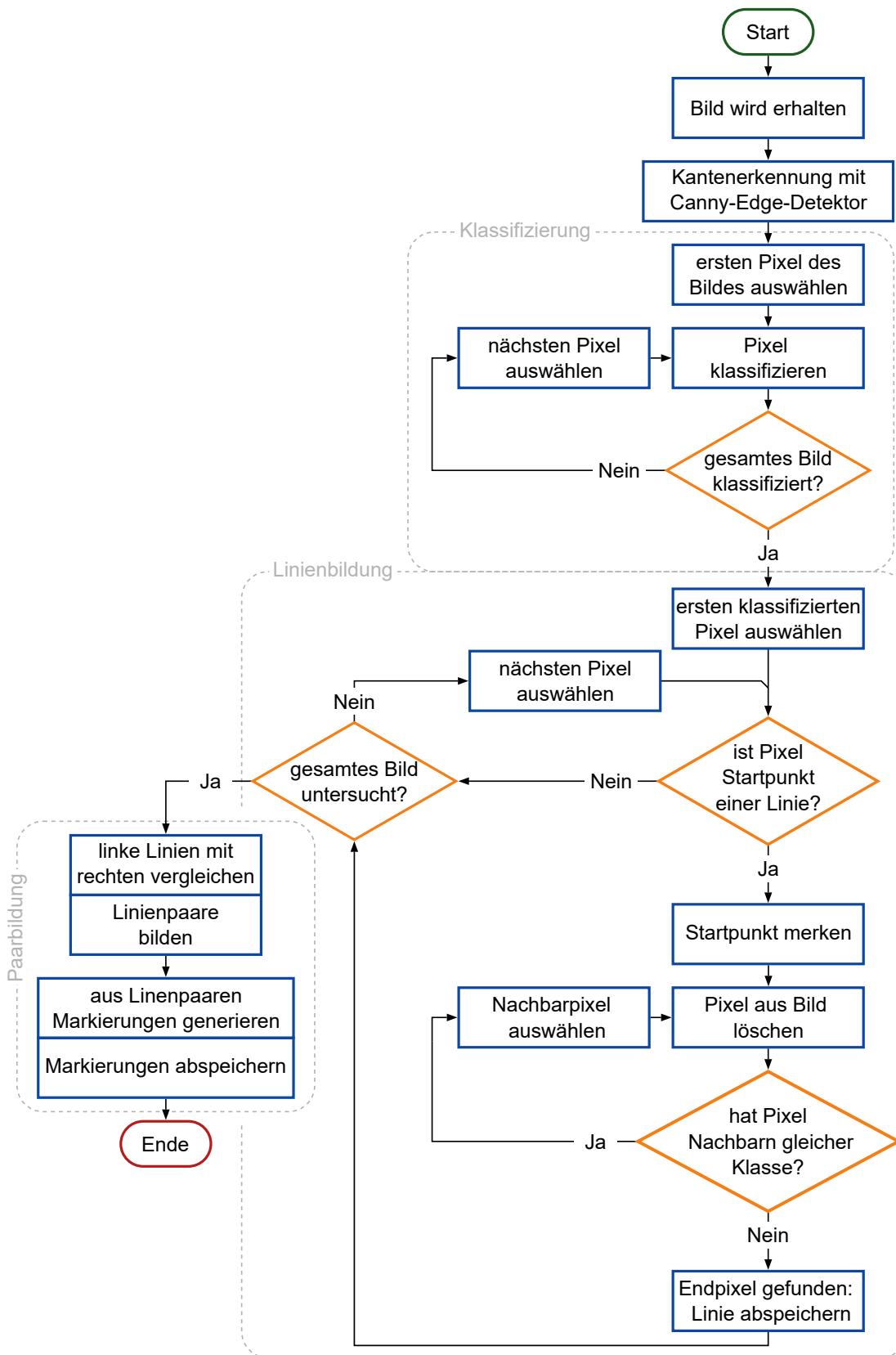


Abb. 4.2: Ablauf des Algorithmus zur Erkennung von Fahrspurmarkierungen

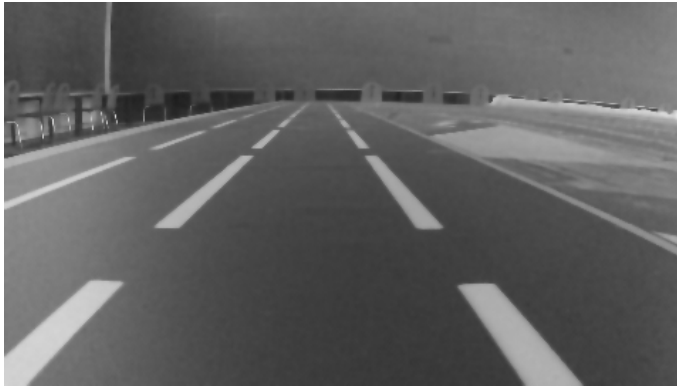


Abb. 4.3: Ein Beispiel Bild an dem der Ablauf demonstriert wird

4.1.1 Kantenerkennung mittels Canny-Edge-Detektor

Begonnen wird mit der Detektion von Kante im Bild. Dazu wird das Bild zuerst mit OpenCV geladen.

Absatz Ja/Nein?

Um kleine Störungen im Bild, welche bestehende Kanten verzerren oder als falsche Kante erkannt 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 Kegelgröße 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 19983 entwickelten und in [**Canny:computationAlapproachEdgeDetection**] veröffentlichten Algorithmus. Dieser bestimmt für jeden Pixel den Gradientenbetrag der Gradienten 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 Schwellwerte, Lücken zwischen diesen Pixeln geschlossen werden. [**Nischwitz:Computergrafik2**]

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 Beispielbild 4.3 angewendet und das Ergebnis des Canny-Edge-Detektors ausgegeben, ergibt sich Abbildung 4.4. Im Gegensatz zu alternativen, wie einer reinen Grabantenbetrachtung, liefert der Canny-Edge-Detektor Kantenmarkierungen (hier in weiß) die nur ein Pixel breit sind. Dies ermöglicht die in den folgenden Unterkapiteln beschriebenen Schritte.

einen?

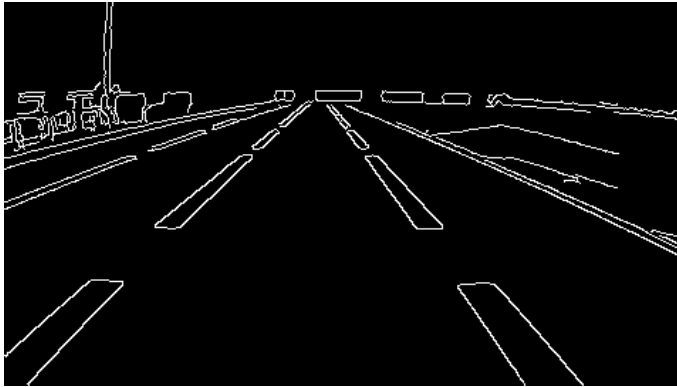


Abb. 4.4: Vom Canny-Edge-Detector gefundene Kanten

4.1.2 Klassifizierung der Kantenpixel

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

„zufällige“ OK?

Deshalb werden 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* verwendend. Zusätzlich wird noch die Richtungsinformation als Vorzeichen abgespeichert.

Die Klassifizierung erfolgt anhand der Gradientenorientierung eines Pixels. Dazu werden mit 3×3 Sobel-Kernen 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 entsprechen der Abbildung 4.5 die Klasse bestimmt werden. Dabei ist zu beachten das \vec{G} immer orthogonal auf der eigentlichen Kante steht, deshalb ist die Klasse *Vertikal* auch auf der links-rechts Achse der Abbildung zu finden.

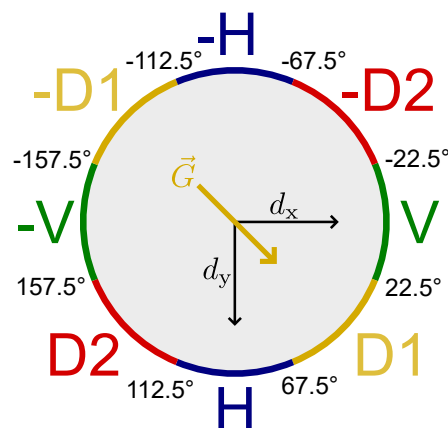


Abb. 4.5: Klassifizierung der Gradientenorientierung (nach [Homann:VorlesungBildverarbeitung])

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, leere 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

```

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

Zuerst die Gradienten d_x und d_y ermittelt. Dazu wird die 3×3 Pixelnachbarschaft des aktuellen Pixels 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 zu Verfügung. Dadurch lässt sich diese Operation in wenigen Zeilen durchführen, wie ?? gezeigt.

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

Nun werden durch eine Folge von Bedingungen die Klasse des aktuellen Pixels bestimmt. Zuerst wird das Vorzeichen bestimmt und im 5. Bit abgespeichert. Dies vereinfacht die folgenden Abfragen, da für die *Vertikale* und *Horizontale* 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 mittel 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, wo jeder Klasse und Vorzeichen eine eindeutige Farbe zugeordnet ist. So ist genau zu erkennen, welche Kanten derselben Klasse zugeordnet wurden. Diese Bild ist in Abbildung 4.6 gezeigt.

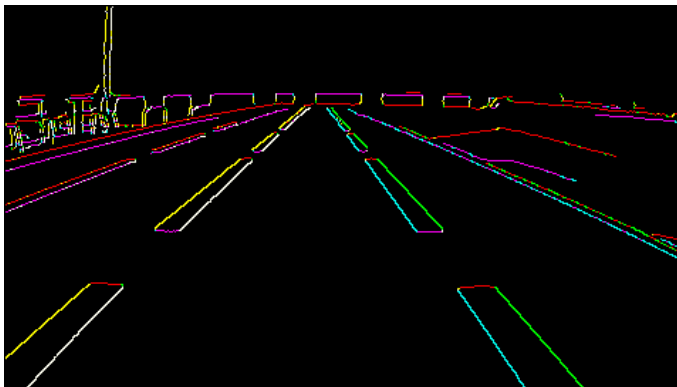


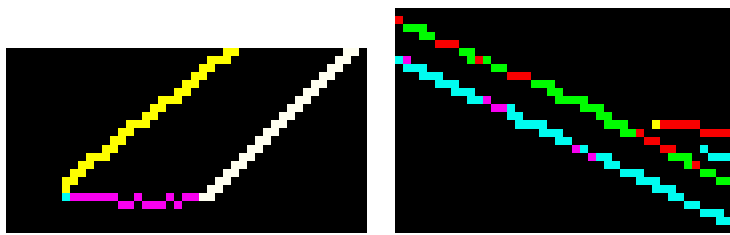
Abb. 4.6: Klassifizierte Kanten mit farblicher Markierung der unterschiedlichen Klassen (Farben sind nicht identisch mit Abbildung 4.5)

Genauigkeit der Klassifizierung

Die Klassifizierung erfolgt leider 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 gut und schlecht klassifizierte Bildbereiche

4.1.3 Linienbildung

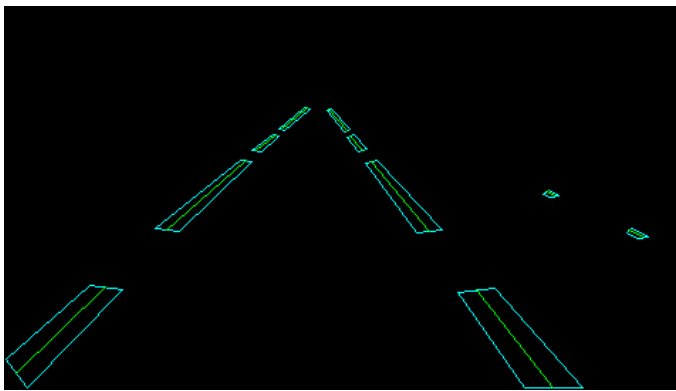


Abb. 4.8: Umrisse und Mittellinien der gefundenen Fahrspurmarkierungen

4.1.4 Performance Betrachtung

sehr schlecht offensichtlich...

4.2 Implementierung in eine ROS Node

4.2.1 Performance Betrachtung

4.3 Optimierung durch eigene Implementierung des Canny-Edge-Detectors

4.3.1 Performance Betrachtung

5 Ausblick

Abbildungsverzeichnis

2.1	SparkFun JetBot AI Kit V2.1 [jetbot:Sparkfun]	3
2.2	CPU Auslastung des JetBots ohne ROS	4
2.3	CPU Auslastung mit laufender Kamera und ROS-Core	4
3.1	Unkalibriertes Kamerabild mit tonnenförmiger Verzeichnung	5
3.2	Darstellung der optischen Verzerrung (nach [wiki:LinsenVerzerrung])	6
3.3	Probleme in der Ausrichtung von Sensor und Linse (nach [Matlab:CameraCalibration])	6
3.4	Schachbrett Kalibriermuster mit markierten inneren Kreuzungen	7
3.5	Schritte der intrinsischen Kalibrierung	9
3.6	Beziehungen der entzerrer Node zu bestehenden Nodes	10
3.7	CPU Auslastung des JetBots mit laufender Kamera und entzerrer Node	13
4.1	Zusammenhang der Fahrspurmarkierung-Erkennungs Node mit den bestehen- den Nodes	15
4.2	Ablauf des Algorithmus zur Erkennung von Fahrspurmarkierungen	16
4.3	Ein Beispiel Bild an dem der Ablauf demonstriert wird	17
4.4	Vom Canny-Edge-Detector gefundene Kanten	18
4.5	Klassifizierung der Gradientenorientierung (nach [Homann:VorlesungBildverarbeitung])	18
4.6	Klassifizierte Kanten mit farblicher Markierung der unterschiedlichen Klassen (Farben sind nicht identisch mit Abbildung 4.5)	20
4.7	Vergleich gut und schlecht klassifizierte Bildbereiche	21
4.8	Umrisse und Mittellinien der gefundenen Fahrspurmarkierungen	21

Tabellenverzeichnis

3.1	Gemessene Laufzeit bei 10 Durchläufen der Callback-Funktion	13
4.1	Zuordnung der Klassen zu Bits	19

Codeverzeichnis

3.1	Definiteion der Größe des Kalibriermuster	7
3.2	Initialisierung von Variablen für die Kalibreirung	8
3.3	Finden und Verarbeiten der Kalibrierbilder	8
3.4	Abspeichern der Gefundenen Bildpunkte	8
3.5	Ermitteln der Kalibrierwerte mittels OpenCV	9
3.6	Berechnen des Reprojektions-Fehlers	10
3.7	Einlesen der Kalibrierungsergebnisse aus einer YAML-Datei	11
3.8	Bestimmen der Pixel-Mappings zu Entzerrung	12
3.9	Vereinfachte Version der Callback-Funktion zur Durchführung der Entzerrung	12
4.1	Laden, glätten eines Bildes und durchführen der Kantenerkennung mit OpenCV	17
4.2	Schleife über das vom Canny-Edge-Detektor gelieferte Bild	19
4.3	Bestimmung der Gradienten d_x und d_y	19
4.4	Durchführen der Klassifizierung mittel des bestimmten Winkels	20