

Bachelorarbeit

Video-basierte Fahrspurerkennung von mobilen Robotern

Jan Wille 1535115

09.05.2022 – 10.08.2022

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
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 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
2.5.1 Performance Baseline	2
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	14
4 Fahrspurerkennung	15
4.1 Bild Vorbereitung	15
4.1.1 Bildausschnitt auf relevanten Bereich reduzieren	15
4.1.2 Verbesserung der Bildqualität	16
4.2 Canny-Edge-Detector	16
4.3 Orientierungserfassung mittels Sobel	16
4.4 Linienbildung	16
5 Ausblick	17
Literatur	18
Abbildungsverzeichnis	19
Tabellenverzeichnis	20

Codeverzeichniss

21

Glossar

C++

Eine relativ hardwarenahe Programmiersprache

Callback-Funktion

Eine Funktion die unter bestimmten Bedingungen automatisch aufgerufen wird. Im Bezug aus ROS geht es meistens um Funktion die für jede Nachricht auf einem abonierten Topic mit deren Inhalt aufgerufen werden.

OpenCV

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

Python

Eine abstrakte, sehr 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. Von Treibern bis moderne Algorithmen, ROS birgt alles was für das nächste Robotic Projekt benötigt wird. Dabei ist es vollständig Open Source.

ROS Node

Eine ROS Node ist ein Teilprogramm welches von ROS verwaltet wird. Es kann Informationen als Topic veröffentlichen und Topics Abonieren 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

wie beschreiben??

Weltkoordinaten

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

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 Spruererkennung? 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

2.5.1 Performance Baseline

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

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

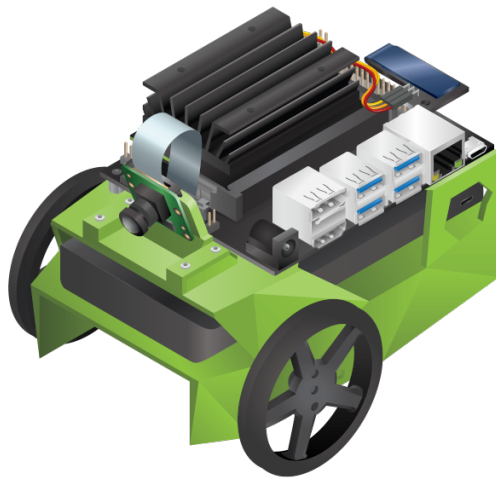


Abb. 2.1: Render eines möglichen Aufbaus [Nvi22]



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

2.6 Aufgebaute Anlage

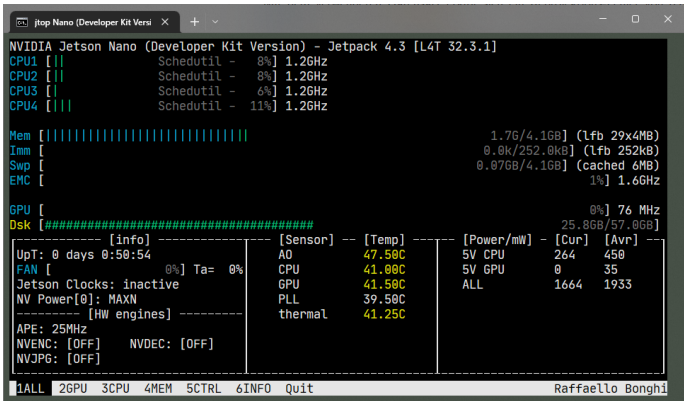


Abb. 2.3: CPU Auslastung des JetBots ohne ROS

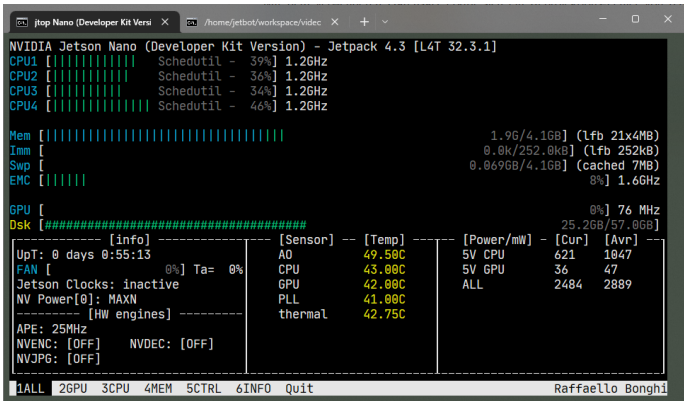


Abb. 2.4: CPU Auslastung mit laufender Kamera und ROS-Core

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.

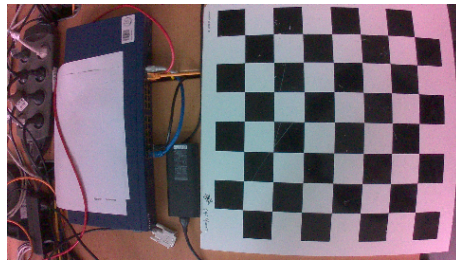


Abb. 3.1: Unkalibriertes Kamerabild mit tonnenförmiger Verzeichnung

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_3 und k_5 die Verzerrungskoeffizienten. Theoretisch existieren noch weitere Koeffizienten, aber in der Praxis haben sich die ersten drei als ausreichend herausgestellt. [Han11]

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

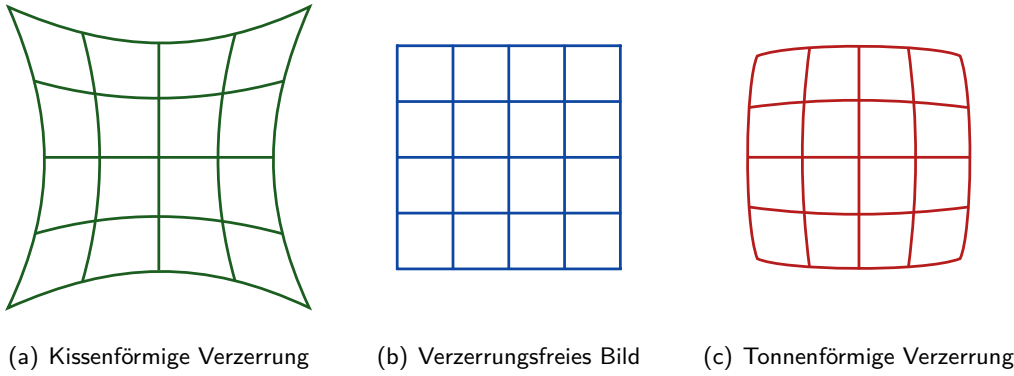


Abb. 3.2: Darstellung der optischen Verzerrung (nach [Wik22])

3.1.2 Tangentielle Verzerrung

Die tangentielle 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.

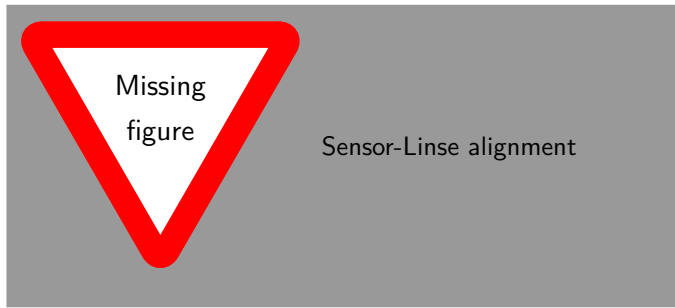


Abb. 3.3: Probleme in der Ausrichtung von Sensor und Linse (nach [Mat22])

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

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

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. [Ope22]

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. [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 die den Vorgang einfach und wiederholbar zu machen. Als Vorlage für dieses dient die Anleitung zur Kamera Kalibrierung aus der OpenCV Dokumentation [Ope22].

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.

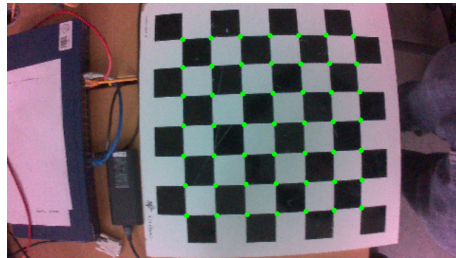


Abb. 3.4: Schachbrett Kalibriermuster mit markierten inneren Kreuzungen

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 [Ope22] 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 Schwarzweißbild umgewandelt.

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.
```

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.

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), criteria)
5     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 Verzerrungskoeffizienten D_{coeff} zurück. [Ope22]

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 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 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.

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 Weltkoordinaten. Der Mittelwert aller Abweichungen in allen verwendeten Bildern gibt den Reprojektions-Fehler für den ganzen Kalibriervorgang an.



Abb. 3.5: Schritte der intrinsischen Kalibrierung

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 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 zweigen 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.

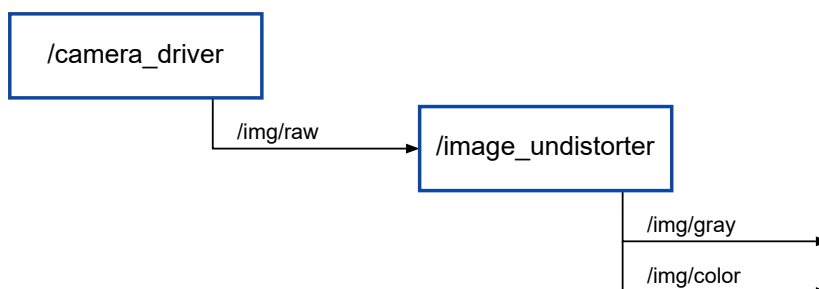


Abb. 3.6: Beziehungen der entzerrer Node zu bestehenden Nodes

Initialisieren der Node

Beim Start der Node wird die `main()` Funktion aufgerufen, welche die notwendigen ROS Funktionen zur Initialisierung aufruft, das benötigt Topic abonniert, ein 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 =
3     "./tools/calibration/calibration.yaml";
4 YAML::Node full_config = YAML::LoadFile(configFilePath);
5
6 // read distortion coefficients and convert to OpenCV vector
7 auto distortion_YAML =
8     camera_config["intrinsic"]["distortion"]
9     .as<std::vector<double>>();
10 cv::Mat distortion ( distortion_YAML );
11
12 // read camera matrix and convert to OpenCV matrix
13 auto cameraMatrix_YAML =
14     camera_config["intrinsic"]["matrix"]
15     .as<std::vector<std::vector<double>>>();
16 cv::Mat cameraMatrix = toMat( cameraMatrix_YAML );
17
18 // read image size
19 cv::Size imageSize(
20     full_config["images"]["size"]["width"].as<int>(),
21     full_config["images"]["size"]["height"].as<int>()
22 );
```

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.

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);
```

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 Schwarzweißbild 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 undistortedImage;
3
4     // convert from ROS msg-type to opencv matrix
5     cv_bridge::CvImagePtr imagePtr =
6         cv_bridge::toCvCopy(original);
7
8     // apply the calculated maps to undistort the image
9     cv::remap(imagePtr->image, undistortedImage, rectifyMapX,
10         rectifyMapY, cv::INTER_LINEAR);
11
12     // crop relevant section from image
13     undistortedImage = undistortedImage(ROI);
14
15     // publish images
16     cv_bridge::CvImage colorImage(std_msgs::Header(),
17         "rgb8", undistortedImage);
18     pub_colorImage->publish(colorImage.toImageMsg());
19 }

```

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.

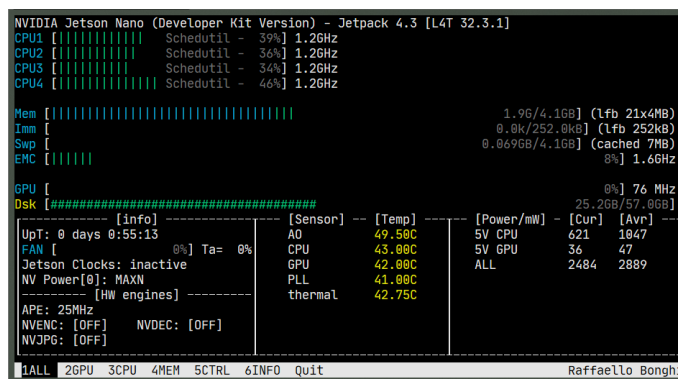


Abb. 3.7: CPU Auslastung des JetBots mit laufender Kamera und entzerrer
Node

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 ≈ 6 ms.

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.

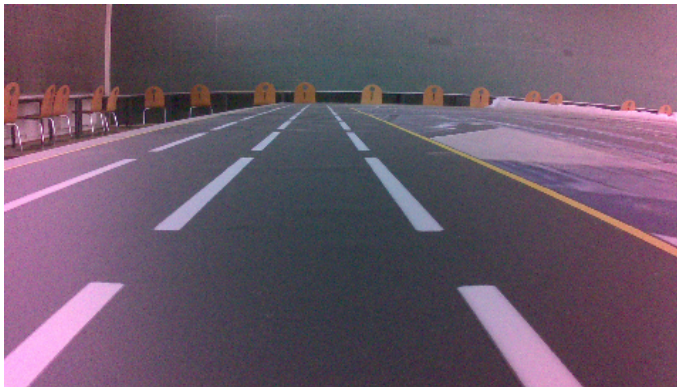


Abb. 4.1: Ein Beispiel Bild an dem der Ablauf demonstriert wird

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.

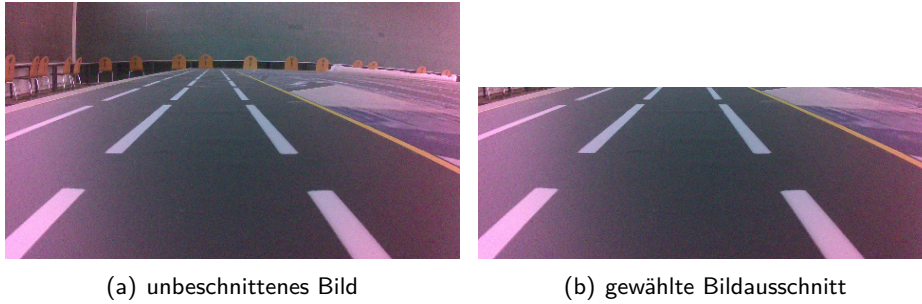


Abb. 4.2: Vergleich von Ursprungsbild und Bildausschnitt



Abb. 4.3: Durch bilaterales Filtern verbessertes Bild

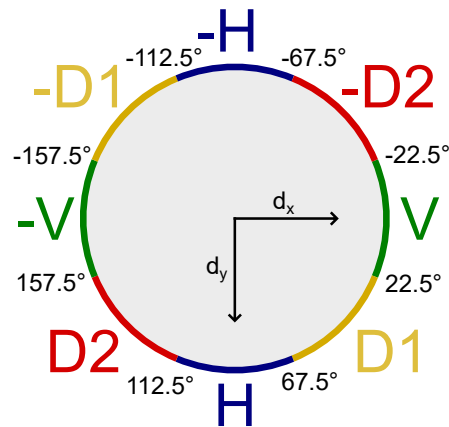


Abb. 4.4: Klassifizierung der Gradientenorientierung (nach [Hom21])

4.1.2 Verbesserung der Bildqualität

4.2 Canny-Edge-Detector

4.3 Orientierungserfassung mittels Sobel

4.4 Linienbildung

5 Ausblick

Literatur

- [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).
- [Han11] Tobias Hanning. *High Precision Camera Calibration*. eng. 1. Aufl. Vieweg+Teubner, 2011. 225 S. DOI: 10.1007/978-3-8348-9830-2. URL: <https://link.springer.com/content/pdf/10.1007/978-3-8348-9830-2.pdf> (besucht am 19.07.2022).
- [Hom21] Prof. Dr.-Ing. Hanno Homann. *Script: Vorlesung Bildverarbeitung*. ger. Moodle, Open Source Lernplattform der Hochschule Hannover. Stand: WiSe 2021/22. 2021.
- [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).
- [Nvi22] Nvidia. *GitHub Repository: NVIDIA-AI-IOT/jetbot. Documentation and sourcecode for NVIDIA Jetson Nano*. eng. 2022. URL: <https://github.com/NVIDIA-AI-IOT/jetbot> (besucht am 15.07.2022).
- [Ope22] 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 18.07.2022).
- [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 15.07.2022).
- [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).

Abbildungsverzeichnis

2.1	Render eines möglichen Aufbaus [Nvi22]	3
2.2	SparkFun JetBot AI Kit V2.1 [Spa22]	3
2.3	CPU Auslastung des JetBots ohne ROS	4
2.4	CPU Auslastung mit laufender Kamera und ROS-Core	4
3.1	Unkalibriertes Kamerabild mit tonnenförmiger Verzeichnung	5
3.2	Darstellung der optischen Verzerrung (nach [Wik22])	6
3.3	Probleme in der Ausrichtung von Sensor und Linse (nach [Mat22])	6
3.4	Schachbrett Kalibriermuster mit markierten inneren Kreuzungen	7
3.5	Schritte der intrinsischen Kalibrierung	10
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	Ein Beispiel Bild an dem der Ablauf demonstriert wird	15
4.2	Vergleich von Ursprungsbild und Bildausschnitt	16
4.3	Durch bilaterales Filtern verbessertes Bild	16
4.4	Klassifizierung der Gradientenorientierung (nach [Hom21])	16

Tabellenverzeichnis

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 .	13
4.1	Test caption	15