AI-Night 2022 | Track „Swarm Robotics”

STUDIENARBEIT

T3_3101

des Studienganges Angewandte Informatik

an der Dualen Hochschule Baden-Württemberg Mosbach
Campus Bad Mergentheim

von

Marie Wierhake

Corbinian Büttner

Abgabedatum: 15.06.2022

Matrikelnummer, Kurs	1959397 | 7631677, TINF19
Gutachter*in der Dualen Hochschule	Prof. Dr. Carsten Müller

Erklärung

Wir versichern hiermit, dass wir unsere Projektarbeit mit dem Thema: „*AI-Night 2022* | *Track ‚Swarm Robotics'*“ selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben. Wir versichern zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Bad Mergentheim, 14.06.2022	Marie Wierhake
Ort Datum	Unterschrift | Marie Wierhake

Laudenbach, 14.06.2022	Büttner
Ort Datum	Unterschrift | Corbinian Büttner

Zusammenfassung

Diese Studienarbeit beschreibt die Arbeiten an den Lego®-Schwarmrobotern des Swarmlabs der DHBW Mosbach am Campus Bad Mergentheim. Während dieser Arbeit wurden die Vorgänge “Linienverfolgung”, “Erkennung von Schildern” und “Navigation durch einen Parcours mittels maschinenlesbaren Codes” umgesetzt. Dabei ermöglicht die Linienverfolgung das Fahren des Schwarmroboters entlang einer schwarzen Linie. Die Schildererkennung findet und reagiert distanzbasiert auf Stoppschilder und Einfahr-Verboten-Schilder. Hierbei wird entsprechend mit dem Anhalten vor dem Stoppschild beziehungsweise mit einer Wendung vor dem Einfahrt-Verboten-Schild reagiert. Bei der Navigation durch einen Parcours reagiert der Roboter auf Kreuzungen. An diesen orientiert er sich mittels maschinenlesbaren Codes und biegt entsprechend seinem Ziel (Parkplatz) ab. Beim Erreichen seines Ziels wird ein Parkmanöver eingeleitet.

Dabei wird auf die notwendigen Vorarbeiten im Sinne von Installation der verwendeten Programme näher eingegangen und die Vorgänge genau definiert. Ebenso werden Probleme und Herausforderungen, die bei der Bearbeitung entstanden sind, beschrieben, sowie Verbesserungen, die in zukünftigen Arbeiten erledigt werden können, aufgezeigt.

Die in dieser Studienarbeit entstanden Programme werden hinsichtlich ihrer Verwendung und Funktionsweise eingehend beschrieben, um Dritten deren Nutzung zu ermöglichen.

Abstract

This student research project describes the work on the Lego® swarm robots of the Swarmlab of the DHBW Mosbach at the Bad Mergentheim campus. During this project, the operations "line tracking", "recognition of signs" and "navigation through a parcours by means of machine-readable codes" were implemented. The line tracking enables the robot to drive along a black line. The sign recognition finds and reacts distance-based to stop signs and no entry signs. In this case, the robot reacts accordingly by stopping in front of the stop sign or by turning around in front of the no entry sign. When navigating through a parcours, the robot reacts to intersections. It orients itself at these by using machine-readable codes and turns according to its destination (parking lot). When it reaches its destination, a parking maneuver is initiated.

The necessary preparatory work in the sense of installing the programs used is discussed in more detail, and the procedures are defined precisely. Likewise, problems and challenges, which arose during the process, are described, as well as improvements, which can be dealt with in future projects, are listed.

The programs created in this study project are described in detail in terms of their use and functionality, in order to allow third parties to use them.

Vorwort

Aus Gründen der besseren Lesbarkeit wird in dieser Arbeit die männliche Sprachform verwendet, es sind aber stets Personen aller Geschlechter gemeint.

Falls wir in der Studienarbeit auf Seiten im Internet verweisen, wurden diese nach sorgfältigen Erwägungen ausgewählt. Auf die zukünftige Gestaltung und den Inhalt der Seiten haben wir jedoch keinen Einfluss. Wir distanzieren uns daher ausdrücklich von diesen Seiten, soweit darin rechtswidrige, insbesondere jugendgefährdende oder verfassungsfeindliche Inhalte zutage treten.

Inhaltsverzeichnis

Abkürzungsverzeichnis

CSV	Comma Separated Values
DHBW	Duale Hochschule Baden-Württemberg
FTP	File Transfer Protocol
GTSRB	German Traffic Sign Recognition Benchmark
QR-Code	Quick-Response-Code
RGB	Rot Grün Blau
SSH	Secure Shell

Abbildungsverzeichnis

1 Einleitung

1.1 Aufgabenstellung und Ziele

Die Weiterentwicklung des Swarmlabs der Dualen Hochschule Baden-Württemberg (DHBW) Mosbach am Campus Bad Mergentheim erfordert die Überarbeitung bereits realisierter Szenarien. Diese sollen anschließend eine solide Grundlage, für weitere Studienarbeiten im Bereich der Forschung und Lehre im Kontext der künstlichen Intelligenz und schwarmbasierter Logistik, bieten.

Diese Studienarbeit hat zum Ziel, dass ein Schwarmroboter einer Linie zuverlässig folgt. Darauf aufbauen sollen ein Stoppschild sowie ein Einfahr-Verboten-Schild eindeutig klassifiziert werden. Sobald der Schwarmroboter einen angemessenen Abstand zum jeweiligen Schild erreicht hat, soll er beim Stoppschild für drei Sekunden halten und anschließend die Linienverfolgung fortsetzen. Wird ein Einfahrt-Verboten-Schild in angemessenem Abstand erkannt, soll der Schwarmroboter um 180° wenden und anschließend die Linienverfolgung fortsetzen.

Ein weiteres Ziel ist die Erstellung eines Custom Classifier für das Einfahrt-Verboten-Schild auf Basis öffentlichen Bildmaterials. Dieser soll eine Genauigkeit von mindestens 90% aufweisen. Die Vorgehensweise soll dabei nachvollziehbar dokumentiert werden, sodass diese Dokumentation als Grundlage für Laborübungen dienen kann.

Weitergehend soll, unter Verwendung der Linienverfolgung, eine Navigation über maschinenlesbare Codes (beispielsweise Strichcodes) zwischen zwei Parkplätzen realisiert werden. Dabei müssen mehrere Kreuzungen zwischen den Parkplätzen liegen. Ein Parkplatz ist mit einer roten Linie umrandet. Der Schwarmroboter soll in der Lage sein, einen Parkplatz zu erkennen und anschließend dort einzuparken.

1.2 Vorgehensweise

Im ersten Schritt wird der bestehende Softwarestand analysiert und getestet. Dabei wird ein besonderer Fokus auf enthaltene Schwachstellen gelegt, um diese in der Weiterentwicklung zu minimieren. Zuerst wird die Linienverfolgung überarbeitet, da diese die Grundlage aller Szenarien darstellt. Daraufhin werden die einzelnen Szenarien

weiterentwickelt und erforderliche Verbesserungen implementiert. Abschließend wird die erstellte Implementierung nachvollziehbar dokumentiert.

Abbildung 1.1 Lego®-Schwarmroboter
Quelle: Unternehmenskommunikation DHBW Mosbach

2 Systemumgebung und technische Grundlagen

In diesem Kapitel werden die technischen Grundlagen, welche für die Studienarbeit erforderlich sind, beschrieben. Auch enthalten sind Hinweise für eine erfolgreiche Installation mit Verweis auf eingesetzte Anleitungen.

2.1 Verwendete Software und Versionen

Der Schwarmroboter SR14 wurde als Master-Schwarmroboter für die „old Generation“ der Schwarmroboter eingerichtet. Dieser ist mit einem Raspberry Pi und der Raspberry Pi HQ Camera ausgestattet.

2.1.1 Raspbian OS

Raspbian ist die Standardwahl für das Betriebssystem auf einem Raspberry Pi. Aufbauend auf Debian ist es Linux-basierend und speziell für den Raspberry Pi optimiert.

Verwendet wird hier Version 11 “Bullseye”. Diese ist zum aktuellen Zeitpunkt die neuste Version von Raspbian.

Diese kann mittels Raspberry Pi Imager, einem Hilfsprogramm zum Flashen von SD-Karten, direkt auf die die SD-Karte des Raspberry Pis installiert werden. Die Installation benötigt circa ein bis vier Stunden und es gilt zu beachten, dass immer wieder Optionen über die Kommandozeile eingegeben werden müssen. Eine erprobte Anleitung ist unter Quelle [1] zu finden.

Für die Benutzung des Pis mussten noch einige Programme und Bibliotheken nachinstalliert werden. Zusätzlich wurden Features wie der Remotezugriff per Secure Shell (SSH) und der Zugriff über das File Transfer Protocol (FTP) (siehe Quelle [2]) aktiviert, sowie die Netzwerkeinstellungen angepasst.

2.1.2 Git

Git ist eine Versionsverwaltung, die es ermöglicht Quellcode alleine oder im Team zu verwalten. Auf dem Raspberry Pi wurde Git deshalb installiert, um den aktuellen Stand

der Software im Team zu teilen und auf verschiedene Stände zurückzugreifen. Hierbei wird auf ein gemeinsames Repository zurückgegriffen, welches auf github.com zur Verfügung steht. Einer erprobte Installationsanleitung kann unter Quelle [3] eingesehen werden. Zu beachten gilt, dass die Authentifizierung über die Kommandozeile nicht mehr mit dem Passwort des Git-Accounts, sondern nur mit beispielsweise Token möglich ist. Eine Anleitung für die Erstellung und Verwendung der Token ist unter Quelle [4] verlinkt.

Folgende sind die wichtigsten Befehle, um mit GitHub auf dem Raspberry Pi zu arbeiten. Vorausgesetzt wird, dass der gewünschte Ordner in der Kommandozeile geöffnet ist [5].

- Status prüfen: `git status`
- Datei hinzufügen: `git add *.py` / `git add .` / inkl. subdirs `git add -A`
- Committen: `git commit -m „Comment”`
- Push: `git push / git push origin master`
- aktuellen Stand holen: `git fetch / git fetch origin master` und `git merge origin/master`

2.1.3 Python

Python ist die Programmiersprache, in der die Programmierungen dieser Studienarbeit erarbeitet wurden. Standardmäßig wird mit der oben beschriebenen Raspbian Version Python 3.9.2 mitinstalliert. Diese ist auch der aktuell verwendete Standard.

Die aktuellste, verfügbare Python-Version ist 3.10.4. Aus Kompatibilitätsgründen wurde auf ein Upgrade verzichtet.

2.1.4 OpenCV

Bei OpenCV handelt es sich um die Standardbildverarbeitungsbibliothek. Diese ermöglicht Bilder, wie z.B. von der Kamera der Raspberry Pis, auszuwerten und den Inhalt zu erkennen. Die aktuelle Version ist hier 4.5.5, welche für dieses Projekt verwendet wurde. Bei der Installation von OpenCV sind zwischenzeitlich Schwierigkeiten

aufgetreten. Mit der Anleitung aus Quelle [6] konnte die Installation erfolgreich abgeschlossen werden.

Eine Ausnahme hierbei bildet das Erstellen der Haar Cascade Classifiers. Hierfür wurde die Version 3.4.16 verwendet. Dies ist erforderlich, da einige Kommandozeilenprogramme von OpenCV eingesetzt werden, welche in der Version 4.x.x nicht enthalten sind. Es wurde die aktuellste OpenCV 3.4.x Version eingesetzt. Der Vorgang fand auf einem externen Windowsrechner statt, da die Rechenleistung eines Raspberry Pis für diesen Vorgang unzureichend ist. Die erstellten Classifier können anschließend trotzdem mit anderen Versionen der Software OpenCV genutzt werden.

2.1.5 BrickPi

BrickPi ist die Software, die es ermöglicht Lego® Mindstorm Motoren anzusprechen. Hierfür wird zusätzlich ein Pi Hat benötigt. Dabei handelt es sich um eine Platine, die auf den Raspberry Pi aufgesetzt wird. Verbunden ist das Ganze über die GPIO Pins des Pis. Die aktuelle Version dieser Software ist 1.4.8. Auch hier gestaltete sich die Installation schwierig. Mit der Anleitung aus dem zugehörigen GitHub Repository für einen „Quick Install“ verlief sie erfolgreich. Der zugehörige Befehl lautet: `curl -kL dexterindustries.com/update_brickpi3 | bash` [7].

2.1.6 BrickPi Motorenverkabelung

Für die Verwendung der bereitgestellten Implementierung muss vorher die Verkabelung der Motoren mit dem BrickPi geprüft werden. Ansonsten werden unter Umständen die falschen Motoren angesprochen und dies kann zu Schäden am Schwarmroboter führen. Folgende Zuordnung ist sicherzustellen:

- Port A: Gabel neigen
- Port B: Antrieb
- Port C: Gabel anheben
- Port D: Lenken

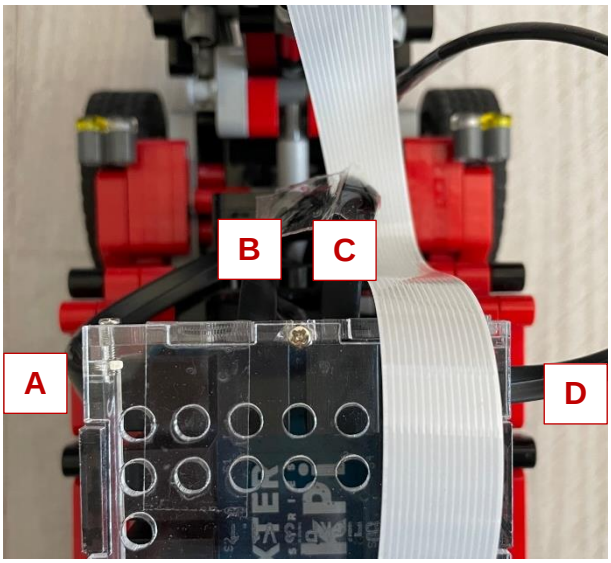

Abbildung 2.1: Motorenverkabelung
Quelle: eigene Darstellung

2.2 Namenskonventionen

2.2.1 einheitliche Benennung und Passwörter

Für eine Vereinheitlichung hat jeder Pi, der uns zur Verfügung stand folgende Namen und Passwörter und Einstellungen bekommen:

- Benennung des Schwarmroboters, beispielsweise „SR01“ (siehe Aufkleber auf dem Pi)
- Benutzername: pi
  Passwort: raspberry
  Dies ist noch das Standardlogin, gegebenenfalls könnte für weitere Entwicklungen ein DHBW Login definiert und eingesetzt werden.
- Sprache und Tastatur: deutsch

2.2.2 Python Naming Conventions

Python Projekte nutzen in der Regel das PEP 8 Namensschema, um den Code lesbarer zu machen und das Fehlerrisiko durch Einheitlichkeit zu verringern. Der bereitge-

stellte Code nutzt dieses Namensschema in den meisten Fällen. Für zukünftige Weiterentwicklungen mit der bereitgestellten Codebasis, wird empfohlen dieses Namensschema weiter einzusetzen [8].

Typ	Namenskonvention	Beispiele
Funktion / Methode	kleingeschriebene Worte, mit Unterstrichen getrennt	detect_barcode, distance_finder
Variable	einzelner Kleinbuchstabe, Wort oder Wörter mit Unterstrichen getrennt	image, sign_name
Klasse	Camel Case Schreibweise, jedes Wort mit einem Großbuchstabe beginnen	Navigator, SignDetector,
Konstante	einzelner Großbuchstabe, Wort oder mehrere Wörter mit Unterstrichen getrennt	M, PORT_1
Modul	kurzes, klein geschriebenes Wort oder Wörter mit Unterstrichen getrennt	line_tracking.py, swarm-robot_sign_program.py
Paket	kurzes, kleingeschriebenes Wort oder Wörter, keine Trennung durch Unterstriche	utility, detectionsign

3 Ausarbeitung

Dieses Kapitel beschreibt die einzelnen Teilaufgaben genauer. Es wird jeweils auf den Stand vorheriger Arbeiten an der DHBW eingegangen und anschießend die Überarbeitung dessen beschrieben. Manche Teilaufgaben erfordern das Hinzufügen zusätzlicher Funktionalitäten, welche nachfolgend auch näher beschreiben werden.

3.1 Linienverfolgung

3.1.1 Ausgangspunkt

Der Softwarestand, der an uns weitergegeben wurde, wurde eingehend analysiert und getestet. Aus diesen Tests ergab sich, dass die Linienverfolgung bei geraden Linien gut funktioniert, jedoch bei gebogenen Linien fälschlicherweise Kreuzungen erkennt. Das hat zur Folge, dass der Roboter bei einer Kurve, statt den Einschlagwinkel seiner Räder anzupassen, anhält. Dieses falsche Verhalten konnte generell abgestellt werden. Die Linienverfolgung behandelte Kurven danach korrekt, jedoch konnten keine engeren Kurven gefahren werden. Der große Wendekreis des Roboters hat zu Folge, dass nur sehr langgezogene Kurven erfolgreich durchfahren werden können. Aus diesem Grund muss die Linienverfolgung überarbeitet und speziell die Erkennung von Kreuzungen neu programmiert werden.

3.1.2 Erste Überarbeitung

In einer ersten Überarbeitung wurde die Kreuzungserkennung umgebaut. Bei der bisherigen Vorgehensweise wurde versucht über die Neigung der erkannten Linie eine Kreuzung zu erkennen. Diese Taktik funktioniert jedoch maximal auf einer geraden Strecke. Auf einer Strecke mit Kurven erzeugt sie zu viele Probleme und Fehler, da eine Kreuzung eben nicht mit einer geneigten Linie gleichgesetzt werden kann. Nach der Überarbeitung wird eine Kreuzung nun erkannt, indem alle Linien im Kamerabild gescannt werden. Diese werden dann nach ihrer Ausrichtung, horizontal oder vertikal, sortiert und die Schnittpunkte dieser Linien untereinander werden berechnet. Diese Schnittpunkte stellen potenzielle Kreuzungen dar. Mit dieser Technik kann garantiert werden, dass Kurven nur als solche erkannt werden.

3.1.3 Zweite Überarbeitung

Nach einer zweiten Überarbeitung der Linienverfolgung wurde der Prozess ausgegliedert und verbessert, sodass die Linienverfolgung parallel in einem eigenen Thread gestartet werden kann. Dabei wurde zum Beispiel die Berechnung des Einschlagwinkels der Räder verfeinert. Zusätzlich wurde die Möglichkeit geschaffen die Linienverfolgung zu pausieren, um andere Aktionen, wie z.B. einen Abbiegevorgang an einer Kreuzung, zu ermöglichen. Dieser Vorgang wird über Events gesteuert, die zwischen den einzelnen Threads ausgetauscht werden. Da beim Navigieren durch den Parcours die schlechten Kurveneigenschaften besonders herausstachen, wurde für zu scharfe Kurven ein Feature implementiert, dass bei einem Verlassen der Linie den Roboter zurücksetzt. So kann dieser den zu scharfen Teil einer Kurve in einem weniger steilen Winkel erneut ansteuern. Hierzu wird kontrolliert, ob die Linie sich in den jeweils äußeren 10% des Bildrandes befindet, um vorbeugend den Roboter vor dem Verlassen der Linie zu stoppen. Der Roboter schlägt dann leicht in die entgegengesetzte Richtung ein und fährt eine Fahrzeuglänge rückwärts. Danach wird die Linienverfolgung wieder aufgenommen. Dieser Ablauf sorgt dafür, dass verhältnismäßig enge Kurven gefahren werden können.

3.1.4 Probleme der Korrekturmaßnahme

Nach zahlreichen Tests der Korrekturmaßnahme, die das Verlassen der Linie verhindern soll, stellt sich damit ein Problem ein. Während der Navigation durch einen Parcours funktioniert das Hilfsmittel ohne nennenswerte Probleme. Wenn nur die Linienverfolgung, mit oder ohne der Schildererkennung, gestartet wird, aktualisiert die Kamera nach dem Korrekturvorgang das Bild nicht korrekt. Faktisch wird in beiden Fällen der identische Code mit gleichen Einstellungen genutzt. Der einzige Unterschied besteht darin, dass bei der Navigation die Kamera von einem parallelen Thread ebenfalls genutzt wird und sich deswegen dauerhaft aktualisiert. Selbst mit einer Simulierung eines anderen Threads, dessen einzige Aufgabe das Abfragen der Kamera ist, konnte kein zu 100% zufriedenstellendes Ergebnis erzielt werden. Hier scheint es Kommunikationsprobleme zwischen Hard- und Software zu geben, die nicht generell nachvollzogen und behoben werden konnten. Da die Linienverfolgung jedoch grundsätzlich trotzdem sehr gut funktioniert und unter der Voraussetzung, dass der Parcours auf den

Wendekreis des Roboter abgestimmt ist, wird das Feature nicht benötigt, daher ist die Korrektur nur über eine Parametereinstellung (`bot.set_leave_line_reverse(True)`) zuschaltbar.

3.2 Klassifizierung von Verkehrsschildern

3.2.1 Ausgangspunkt

Zu Beginn der Studienarbeit lagen zwei verschiedene Softwarestände für die Klassifizierung von Verkehrsschildern vor.

In dem ersten Softwarestand wird die vorbestehende botlib nicht verwendet. Die botlib ist eine Bibliothek, welche in vorherigen Projekten von vielen Studierenden für die Schwarmroboter entwickelt wurde. Zu dieser besteht kaum eine Dokumentation und es sind einige Work-in-Progress-Stände enthalten. In dem ersten Softwarestand zur Klassifizierung von Verkehrsschildern werden einzelnen Bilder aus dem Videostream der Raspberry Pi-Kamera in eine Grauskala umgewandelt und anschließend eine Bilderkennung mit einem entsprechenden Cascade Classifier vollzogen. Wenn ein Stoppschild erkannt wurde, werden die Motoren des Antriebs für drei Sekunden gestoppt und wieder gestartet. Die Ansprache der Motoren erfolgt direkt, ohne Verwendung einer Abstraktionsebene. Eine Linienerkennung ist nicht implementiert oder eingesetzt.

Der zweite Softwarestand verwendet die vorbestehende botlib. Die Klassifizierung wird ebenfalls mit Cascade Classifiern umgesetzt (in der Datei programs/botcamera/objectDetection.py des alten Softwarestands). Dort stehen Cascade Classifier für Stoppschilder, Einfahrt-Verboten-Schilder sowie verschiedene Geschwindigkeitsbegrenzungen zur Verfügung. Auch hier wird das Bild zuerst in eine Grauskala gewandelt und anschließend die Klassifizierung mittels den Cascade Classifiern vollzogen. Darauffolgend gibt die Funktion eine Liste mit den klassifizierten Schildern zurück. Die Liste enthält den Schildnamen sowie Positionsdaten. Aufrufbar ist dies über das Programm in program/follow_line.py im vorher bestehenden Softwarestand. Hier muss der Parameter doTrafficSigns=True gesetzt werden. Die Klassifizierung wird mit der vorher beschriebenen objectDetection-Klasse ausgeführt. Die klassifizierten Schilder werden anschließend im Vorschaubild mit einem Rechteck gekennzeichnet und der Schildname angezeigt. Eine Reaktion des Schwarmroboters auf die erkannten Schilder ist

nicht ersichtlich implementiert. Während der Schilderkennung wird auch eine Linienverfolgung ausgeführt. Im Test zeigte sich diese als unzuverlässig. Der Schwarmroboter verfolgte die Linie nicht flüssig und musste häufig korrigieren. Die Korrekturen schienen meistens zu stark, wodurch fortlaufend weitere Korrekturen erforderlich sind.

3.2.2 Erstellung eines Cascade Classifiers

Der Custom Haar Cascade Classifier für das Einfahrt-Verboten-Schild wird mit OpenCV auf einem Windows Computer erstellt. Ein laut Beschreibung angedachter NVIDIA Jetson AGX stand hierfür nicht zur Verfügung. Es werden Kommandozeilen-Programme von OpenCV eingesetzt, welche für die OpenCV 4.x.x Versionen nicht verfügbar sind. Daher muss die aktuellste Version von OpenCV 3.4 eingesetzt werden, siehe [9]. Die Verwendung einer IDE wie PyCharm und einer Versionsverwaltung ist empfehlenswert. Die Anleitung ist angelehnt an Quelle [10].

Als Codebasis kann das erarbeitete Projekt „create_classifier“ verwendet oder ein neues erstellt werden. In dieser Beschreibung werden entwickelte Funktionen für die Erstellung der Ausgangsbasis für das Generieren und Bewerten der Classifier verwendet. Daher empfiehlt es sich, das bereitgestellte Projekt einzusetzen oder die entsprechenden Funktionen daraus in das neue Projekt zu übernehmen. Das verwendete Projekt sollte ein eigenes virtual Environment besitzen. Außerdem wird die Installation einer aktuellen Python-Version, matplotlib und numpy vorausgesetzt. Anschließend muss OpenCV für Python installiert werden.

1) Projekt mit venv erstellen
    a. Prerequisites
        i. `pip install matplotlib`
        ii. `pip install numpy`
    b. final step
        i. `pip install opencv-python`

Im Anschluss werden negative und positive Beispielbilder gesammelt. Für die **Negativbilder** kann ein Video verwendet werden, bestenfalls aus der Umgebung, in welcher der Schwarmroboter eingesetzt werden soll. Mit dem VLC Media Player können daraus einzelne Bilder erzeugt werden. Siehe dafür Werkzeuge > Einstellungen > Video > Videoschnappschüsse, dort kann das Verzeichnis, der Name und das Format gewählt und das Tastenkürzel Shift + s ausgewählt werden [11]. Die Benennung der Bilder mit negative_<Nummer> hat sich als sinnvoll erwiesen. In diesem Videostream dürfen keine positiv-Beispiele, das heißt in diesem Fall keine Einfahrt-Verboten-Schilder, enthalten sein. Mindestens 100 Negativbilder sind empfehlenswert.

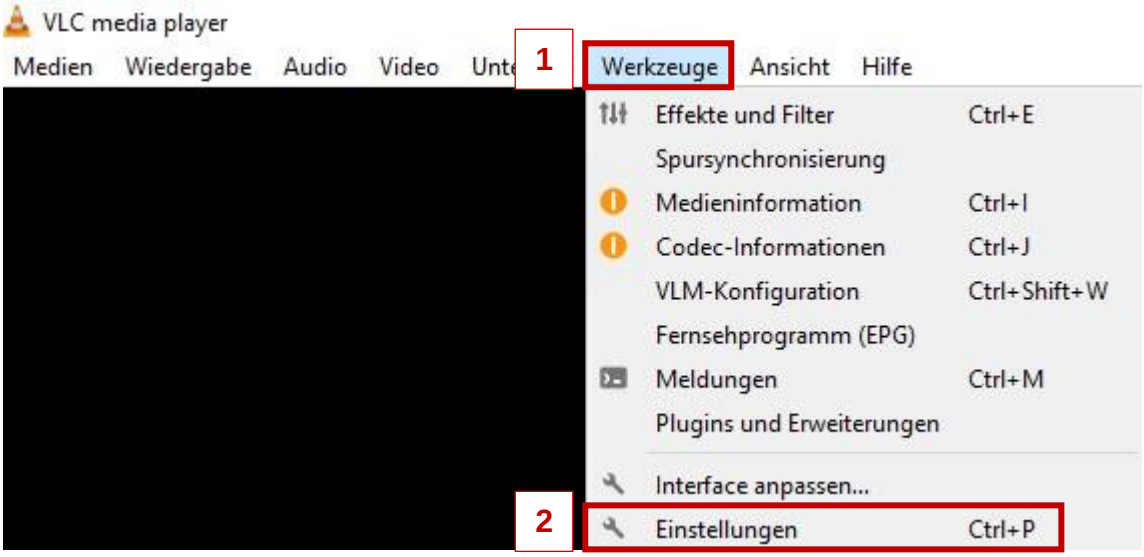

Abbildung 3.1: VLC Media Player – Einstellungen
Quelle: eigene Darstellung

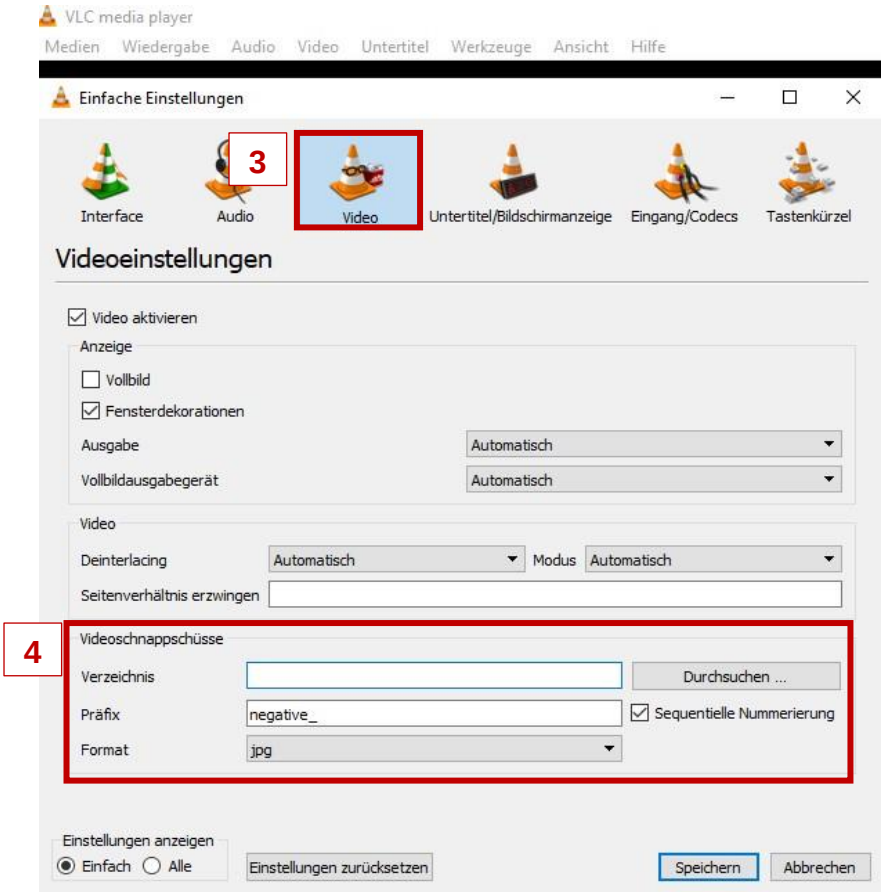

Abbildung 3.2: VLC Media Player – Videoschnappschüsse
Quelle: eigene Darstellung

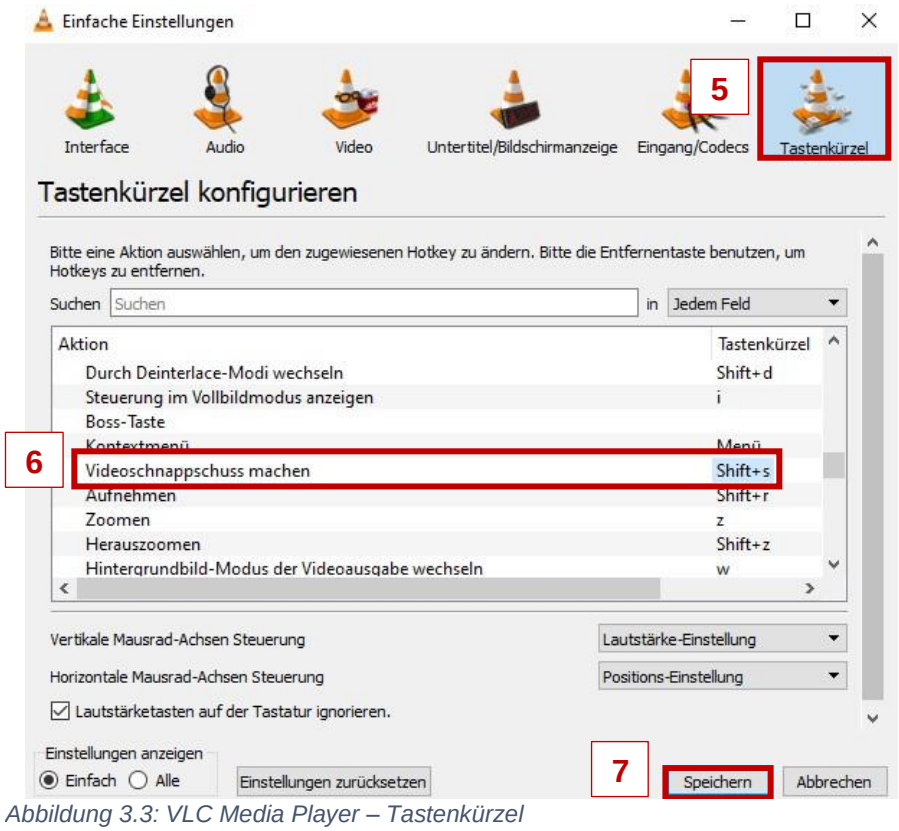

Abbildung 3.3: VLC Media Player – Tastenkürzel
Quelle: eigene Darstellung

Für die **Positivbilder** wird der Datensatz „German Traffic Sign Recognition Benchmark“ (GTSRB) vom Institut für Neuroinformatik eingesetzt, Download über Quelle [12] beziehungsweise direkt über Quelle [13], verwende GTSRB_Final_Training_Images.zip. Der Datensatz enthält verschiedene Ordner mit den jeweiligen Schildern und Beschreibungsdateien. Das nachfolgende Bild enthält die Schilderreihenfolge. Die passende Nummer kann von links nach rechts, startend bei 0 abgezählt werden.

Abbildung 3.4: GTSRB – Bildreihenfolge
Quelle: [17]

Ordner 14 enthält Bilder der Stoppschilder, Ordner 17 Bilder der Einfahrt-Verboten-Schilder.

Hierbei ist zu beachten, dass die Bilder nach dem Download im ppm-Format vorliegen. Dieses kann beispielsweise mit Gimp geöffnet werden. Der Algorithmus für das Training kann mit ppm-Dateien umgehen. Alternativ können die Bilder auch mithilfe von Pythoncode zu jpg-Dateien umgewandelt werden. Dafür steht im bereitgestellten Projekt die Funktion convert_ppm_to_jpg() unter cascadeUtils.py bereit.

Anschließend folgt die Erstellung eines **Descriptionfiles für die Negativbilder**. Dafür kann die Methode generate_negative_description_file() unter cascadeUtility.py verwendet werden.

```
def generate_negative_description_file():
    # open the output file for writing. will overwrite all existing data in there
    with open('neg.txt', 'w') as f:
        # loop over all the filenames
        for filename in os.listdir('negative'):
            f.write('negative/' + filename + '\n')
```

Abbildung 3.5: Code - Descriptionfile für Negativbilder
Quelle: eigene Darstellung

Die entstandene Datei ‚neg.txt' enthält den Ordnername und die Dateinamen der einzelnen Negativbilder in jeweils einer Zeile.

```
1  negative/negative_00001.jpg
2  negative/negative_00002.jpg
3  negative/negative_00003.jpg
4  negative/negative_00004.jpg
```

Abbildung 3.6: Beispiel für neg.txt
Quelle: eigene Darstellung

Danach wird ein **Descriptionfile für die Positivbilder** erstellt werden. Die Methode generate_positive_description_file() unter cascadeUtilty.py kann hierfür eingesetzt werden. Das Descriptionfile enthält den Dateipfad der Bilder sowie Positionsdaten der erkannten Stoppschilder. Dabei werden erkannte Schilder, welche kleiner als 24x24 Pixel sind, aussortiert, um ein besseres Ergebnis zu erzielen. Zudem muss beachtet werden, dass das weiterführend eingesetzte Kommandozeilenprogramm von OpenCV mit roi.x | roi.y | roi.width | roi.height arbeitet. Der Datensatz stellt allerdings statt roi.width | roi.height die x- und y-Werte, also roi.x2 | roi.y2 in der beschreibenden CSV-Datei (Comma Separated Values) zum Datensatz bereit. Diese Werte werden in der Funktion automatisch umgerechnet.

```python
def generate_positive_description_file(sign_pos_txt, sign_pos_csv, sign_dir):
    pos_file = open(sign_pos_txt, 'w')

    csv_string = 'positive/' + sign_dir + '/' + sign_pos_csv
    with open(csv_string, 'r') as desc_file:
        csv_reader = csv.reader(desc_file, delimiter=';')

        # skip first
        next(csv_reader)

        for line in csv_reader:
            # line[0] = Filename
            # line[1] = width
            # line[2] = height
            # line[3] = roi.x1
            # line[4] = roi.y1
            # line[5] = roi.x2 (useful for calculation of roi.width)
            # line[6] = roi.y2 (useful for calculation of roi.height)
            # line[7] = classId
            if abs(int(line[3]) - int(line[5])) < 20:
                next(csv_reader)
            elif abs(int(line[4]) - int(line[6])) < 20:
                next(csv_reader)
            else:
                # structure: path signCount(here always one) roiValues
                pos_file.write('positive/' + sign_dir + '/' + line[0] + ' 1 ' + line[3] + ' ' + line[4] + ' ' +
                               str(abs(int(line[5]) - int(line[3]))) + ' ' +
                               str(abs(int(line[6]) - int(line[4]))) + '\n')

    pos_file.close()
    print('cascadeUtils | generated positive description file for ' + sign_dir)
```

Abbildung 3.7: Descriptionfile für Positivbeispiele
Quelle: eigene Darstellung

Im nächsten Schritt werden **Samples** erstellt. Hierfür kann das Terminal von beispielsweise PyCharm verwendet werden. Es wird das Kommandozeilenprogramm createsamples.exe von OpenCV 3.4.x eingesetzt. Der Befehl ist mit folgendem Raster aufgebaut:

```
C:\Users\<vollständiger_Dateipfad_zu_OpenCV_3_4_x>\
opencv\build\x64\vc15\bin\opencv_createsamples.exe
-info positive_no_entry.txt
-w 20 -h 20 -num 1800
-vec pos_no_entry_01.vec
```

Parameteroptionen

- vollständiger Pfad zur Installation von OpenCV 3.4.x
- -info = Datei mit Beschreibung der positiven Bilder
- -w | -h = kleinster markierter Bereich der erkannt wird
  Empfehlung: 20x20 oder 24x24

- -num = Anzahl der Vektoren, die erstellt werden sollen
  Empfehlung: größer als Anzahl der beschriebenen positiven Schilder
- -vec = Datei, um Output zu speichern, vec-Dateiformat

Im nächsten Schritt wird das **Modell trainiert**. Hierfür wird das Kommandozeilenprogramm tarincascade.exe von OpenCV 3.4.x verwendet. Der Vorgang kann mehrere Stunden in Anspruch nehmen. Es empfiehlt sich den eingesetzten Befehl an einem anderen Ort nochmals zu speichern, um bei wiederholtem Erstellen von Classifiern die vorangegangenen Parameteroptionen nachschauen und optimieren zu können. Der Befehl ist mit dem nachfolgenden Raster aufzubauen:

```
C:\Users\<vollständiger_Dateipfad_zu_OpenCV_3_4_x>\
opencv\build\x64\vc15\bin\opencv_traincascade.exe -data cascade/
-vec pos_no_entry_01.vec -bg neg.txt -w 20 -h 20 -precalValBufSize
4000 -precalcIdxBufSize 4000 -numPos 550 -numNeg 1100 -numStages 16
-maxFalseAlarmRate 0.4 -minHitRate 0.999
```

Parameteroptionen

- data = Speicherort der erstellten Classifier, Ordner ‚cascade' ist dafür in der Implementierung vorgesehen, Hinweise: darin sollten keine Daten enthalten sein, da es sonst zu einem Fehler kommt
- -bg = Background/ Negativbeispiele, Negativ-Descriptionfile
- -w | -h = kleinste erkannte Region,
  Empfehlung: 20x20 oder 24x24
- -precalValBufSize | - precalcIdxBufSize = Rechnerkapazität für Berechnung
- -numPos = Anzahl Positivbeispiele für das Training
  Empfehlung: kleiner als Anzahl der Positivbeispiele
- -numNeg = Anzahl Negativbeispiele für das Training
  Empfehlung: verschieden, teils ½x positiv oder auch 2x positiv
- -numStage = max. Trainingstages
  Achtung! Overtraining, nicht zu viele!
  Bereich zwischen 15 und 20 anvisieren

- -maxFalseAlarmRate = max. Fehlerrate, es werden Stages hinzugefügt bis diese erreicht ist / anderer Parameter greift
  Empfehlung: 0.4 oder kleiner
- -minHitRate = wenn diese unterschritten wird, werden keine weiteren Layer mehr hinzugefügt
  Empfehlung: 0.99 oder größer

Die ersten Stages werden sehr schnell trainiert, alle weiteren nehmen mehr Zeit in Anspruch. Nach starten des Trainings sollte im Terminal ein ähnliches Bild wie folgt zu sehen sein:

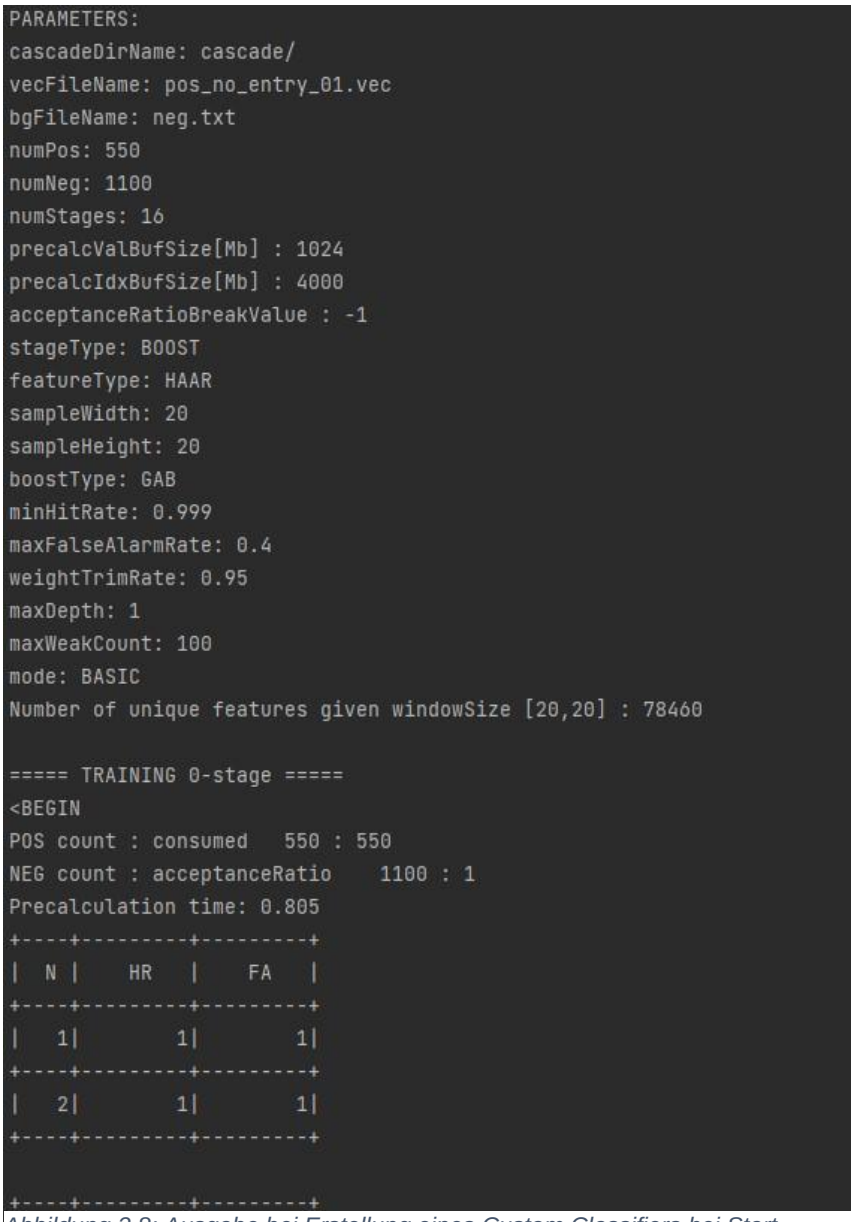

Abbildung 3.8: Ausgabe bei Erstellung eines Custom Classifiers bei Start
Quelle: eigene Darstellung

```
===== TRAINING 1-stage =====
<BEGIN
POS count : consumed   550 : 550
NEG count : acceptanceRatio    1100 : 0.224673
Precalculation time: 0.812
+----+---------+---------+
|  N |    HR   |    FA   |
+----+---------+---------+
|   1|        1|        1|
+----+---------+---------+
|   2|        1|        1|
+----+---------+---------+
|   3|        1|     0.26|
+----+---------+---------+
END>
Training until now has taken 0 days 0 hours 0 minutes 6 seconds.

===== TRAINING 2-stage =====
<BEGIN
POS count : consumed   550 : 550
NEG count : acceptanceRatio    1100 : 0.0614766
Precalculation time: 0.769
+----+---------+---------+
|  N |    HR   |    FA   |
+----+---------+---------+
|   1|        1|        1|
+----+---------+---------+
|   2|        1|        1|
+----+---------+---------+
|   3|        1| 0.693636|
+----+---------+---------+
|   4|        1| 0.294545|
+----+---------+---------+
END>
Training until now has taken 0 days 0 hours 0 minutes 10 seconds.
```

Abbildung 3.9: Ausgabe bei Erstellung eines Custom Classifiers erster Durchgang
Quelle: eigene Darstellung

Finaler Schritt eines neuen Classifiers:

```
===== TRAINING 14-stage =====
<BEGIN
POS count : consumed   550 : 550
NEG count : acceptanceRatio    1100 : 6.19009e-07
Precalculation time: 0.803
+----+---------+---------+
|  N |    HR   |    FA   |
+----+---------+---------+
|   1|        1|        1|
+----+---------+---------+
|   2|        1|        1|
+----+---------+---------+
|   3|        1|        1|
+----+---------+---------+
|   4|        1| 0.866364|
+----+---------+---------+
|   5|        1| 0.800909|
+----+---------+---------+
|   6|        1| 0.802727|
+----+---------+---------+
|   7|        1| 0.673636|
+----+---------+---------+
|   8|        1| 0.549091|
+----+---------+---------+
|   9|        1| 0.525455|
+----+---------+---------+
|  10|        1|      0.4|
+----+---------+---------+
END>
Training until now has taken 0 days 1 hours 7 minutes 44 seconds.

===== TRAINING 15-stage =====
<BEGIN
POS count : consumed   550 : 550
NEG count : acceptanceRatio    1 : 2.14748e-07
Required leaf false alarm rate achieved. Branch training terminated.
```

Abbildung 3.10: Ausgabe bei Erstellung eines Custom Classifiers Ende
Quelle: eigene Darstellung

Der fertige Classifier ist anschließend im Ordner cascade/ unter dem Namen cascade.xml abgelegt. In der Datei kann die Anzahl der Stanges aus den Kommentaren entnommen werden. Falls der Prozess zwischenzeitig abbricht, jedoch bereits im cascade-Ordner Dateien mit dem Namen stagex.xml enthalten sind, wird der abgebrochene Classifier bei erneutem Aufruf des vorherigen Befehls weiter trainiert. Bevor ein neuer Classifier in dem Projekt erstellt werden soll, muss der cascade-Ordner geleert und die Dateien gegebenenfalls an einem anderen Ort gespeichert werden. Für die Verwendung des Classifiers ist lediglich die Datei cascade.xml erforderlich, die stage-Dateien können verworfen werden. In der params.xml können bestimmte Parameter ausgelesen werden.

Anschließend wird der erstellte **Classifier ausgewertet**. Dafür steht die Methode get_accuracy() unter calculateAccuracy.py zur Verfügung. Der Aufruf kann über die main.py wie folgt aussehen:
Zuerst muss eine Datei mit den Daten für die Positivbeispiele erstellt werden, die für die Berechnung der Genauigkeit in Betracht gezogen werden. Dafür ist eine Funktion in cascadeUtils.py bereit.

```
cascadeUtils.generate_positive_description_file_accuracy(
      'positive_no_entry_accuracy.txt',
      'GT-00017.csv',
      'no_entry')
```

Parameteroptionen

1) Beschreibungsdatei der Positivbeispiele, die für das Training eingesetzt wurde
2) Name der CSV-Datei des GTSRB-Datensatzes
3) Ordnername der Positivbilder

Anschließend folgt die Berechnung der Genauigkeit:

```
calculateAccuracy.get_accuracy('positive_no_entry_accuracy.txt',
                               'neg.txt',
                               'test_no_entry.txt',
                               'no_entry/classifier_no_entry_07.xml')
```

Parameteroptionen

1. Beschreibungsdatei der Positivbeispiele, welche für die Berechnung der Genauigkeit infrage kommen
2. Beschreibungsdatei der Negativbilder
3. Datei in welche die erkannten Schilder mit Position geschrieben werden
4. Dateipfad des Cascade Classifiers

Für die Berechnung der Genauigkeit sollten die gleichen Positiv- und Negativbilder eingesetzt werden, wie zum Training verwendet wurden.

Die eingesetzten Cascade Classifier für das Stoppschild und das Einfahrt-Verboten-Schild erfüllen eine Genauigkeit über 90%.

```
*** Calculate accuracy for best stop classifier ***
cascadeUtils | generated positive accuracy description file for stop
get test pictures | finished
load pictures | finished
               _^
              / _)
     _/\/\/\_/ /
   _|         /
 _|  (  | (  |
/__.-'|_|--|_|
accuracy for the classifier stop/classifier_stop_01.xml is:
0.9086021505376344
```

```
*** Calculate accuracy for best no entry classifier ***
cascadeUtils | generated positive accuracy description file for no_entry
get test pictures | finished
load pictures | finished
               _^
              / _)
     _/\/\/\_/ /
   _|         /
 _|  (  | (  |
/__.-'|_|--|_|
accuracy for the classifier no_entry/classifier_no_entry_07.xml is:
0.9310344827586207
```

3.2.3 Cascade Classifer erstellen | Kurzanleitung

Siehe README vom Projekt “create_classifier“

1) use the project “create_classifier” or create new one with following structure and programs:

   cascade
   generated_classifier

   negative

   positive

   calculateAccuracy.py

   cascadeUtils.py

   main.py

2) gather negative pictures (or use existing ones) – at least 100
3) download positive pictures with description file from https://sid.erda.dk/public/archives/daaeac0d7ce1152aea9b61d9f1e19370/published-archive.html
   => GTSRB_Final_Training_Images.zip and choose the corresponding folder. Add the ppm-pictures and CSV to a new folder under positive/<new folder>
4) generate negative description file with "`cascadeUtils.generate_negative_description_file()`" (if not exists or you changed the negative pictures)
5) generate positive description file with e.g. "`cascadeUtils.generate_positive_description_file('positive_stop.txt','GT-00014.csv','stop')`" (if not exists or you changed the positive pictures)
6) generate samples with command line program of OpenCV 3.4.x with e.g. `C:\Users\<full_path_to_OpenCV_3_4_x>\opencv\build\x64\vc15\bin\opencv_createsamples.exe`

   ```
   -info positive_no_entry.txt

   -w 20 -h 20

   -num 1800
   -vec pos_no_entry.vec
   ```

   variate with the parameters - read more about them in the documentation

7) train model with command line program of OpenCV 3.4.x | info! could need some hours, use e.g.

```
C:\Users\<full_path_to_OpenCV_3_4_x>\opencv\build\x64\vc15\bin\
opencv_traincascade.exe
   -data cascade/
   -vec pos_no_entry.vec
   -bg neg.txt
   -w 20 -h 20
   -precalValBufSize 4000 -precalcIdxBufSize 4000
   -numPos 550
   -numNeg 1100
   -numStages 16
   -maxFalseAlarmRate 0.4
   -minHitRate 0.999
```

variate with the parameters - read more about them in the documentation

8) calculate accuracy of the classifier

a. generate test-description-file with e.g. “`cascadeUtils.generate_positive_description_file_accuracy('positive_no_entry_accuracy.txt', 'GT-00017.csv', 'no_entry')`“

b. calculate accuracy with e.g. “`calculateAccuracy.get_accuracy('positive_no_entry_accuracy.txt', 'neg.txt', 'test_no_entry.txt', 'no_entry/classifier_no_entry_07.xml')`“

3.2.4 Distanzbestimmung der klassifizierten Verkehrsschilder

Eine große Schwachstelle des vorherigen Softwarestandes ist, dass lediglich Schilder erkannt werden können, aber kein weiterer Parameter erfasst wird. Eine Reaktion auf ein Verkehrsschild soll jedoch in angemessenem Abstand zu dem Schild erfolgen und

nicht sobald das Schild im Bild erkennbar ist. Daher wurde die Klassifizierung der Verkehrsschilder mit einer Distanzbestimmung erweitert. Es wurde folgende Anleitung dafür adaptiert und für den Schwarmroboter angepasst, siehe . Die Implementierung ist dem bereitgestellten Projekt im Rahmen dieser Studienarbeit zu entnehmen.

Voraussetzung für diese Art der Distanzbestimmung ist die Verwendung von Python, OpenCV und einem Haar Cascade Classifier.

Begonnen wird mit der Aufnahme eines **Referenzbild**s für das jeweilige Verkehrsschild. Dafür kann ***take_picture.py*** verwendet werden. Der Dateipfad zur Speicherung des muss entsprechend angepasst werden. Gespeichert wird das Referenzbild im Ordner detectionsign/referencepictures. Als Name bietet sich die Art des Schildes und die Distanz in Zentimetern an, beispielsweise stop_30.png. Das Referenzbild wird mit dem Schwarmroboter aufgenommen und die Distanz zum Schild sowie die Breie des Schilds müssen bekannt sein. Eine genaue Distanzbestimmung ist stark von der Einstellung des Kamerawinkels und der Position des Schildes abhängig. Daher handelt es sich um grob gerundete Werte. Die Implementierung verwendet jeweils 4cm große Schilder und einen Abstand von 30cm.

Der Abstand in Zentimeter und die Breite des Schilds in Zentimetern werden in detectionsign/sign_detection.py in der Methode init_distance_to_signs() eingetragen. Hier wird auch der Dateipfad zum Referenzbild sowie der Dateipfad zum eingesetzten Classifier festgehalten.

Anschließend wird die Brennweite (focal length) über die Breite des Schildes im Frame ermittelt und gespeichert. Dafür wird zuerst im Referenzbild das Schild mit den Cascade Classifier klassifiziert und anschließend die ermittelten Parameter für die Berechnung verwendet. Ein Fehler kann hierbei auftreten, wenn das Schild im Referenzbild mit dem Cascade Classifier nicht erkannt wird. Es ist wichtig ein geeignetes Bild in der gleichen Breite und Höhe zu verwenden, wie im Videostream eingesetzt wird. Mit der implementierten Methode in detectionsign/sign_detection.py/distance_finder() kann die **Distanz** des klassifizierten Verkehrsschildes zum Schwarmroboter mithilfe des aktuellen Kamerabilds ermittelt werden.

Weitergehend ist zu erwähnen, dass die angezeigte Distanz im Videostream bei der aktuellen Implementierung auf rechtsstehende Schilder optimiert ist. Daher kann es

bei Schildern an anderen Positionen und vor allem liegenden Schildern zu Abweichungen bei der Distanz kommen. Jedoch ist in der Implementierung beachtet, dass die Klassifizierung und Reaktion auf Verkehrsschilder sowohl mit stehenden als auch auf dem Boden liegenden Schildern zuverlässig funktioniert.

3.2.5 Erste Überarbeitung der Klassifizierung von Verkehrsschildern

Im ersten Schritt wird auf den vorherigen Softwarestand und die Funktion in programs/follow_line.py aufgebaut. Wenn dort doTrafficSigns=True gesetzt ist, ist die Klassifizierung von Schildern aktiviert. Diese wird um die Distanzbestimmung erweitert. Das bedeutet, für alle erkannten Schilder, die wie beschrieben als Liste zurückgegeben werden, wird die Distanz bestimmt. Wenn die Distanz zu einem Stoppschild 30cm unterschreitet, werden die Antriebsmotoren gestoppt und nach 3 Sekunden wieder gestartet. Die vorherige Linienverfolgung wird fortgesetzt. Um dasselbe Stoppschild nicht mehrfach zu erkennen und wiederholt zu stoppen, wird jeweils ein Zeitstempel gesetzt, wenn ein Stoppschild erkannt und an diesem gehalten wurde. In den 3 Sekunden nach dem Halt an einem Stoppschild wird ein wiederholtes Anhalten unterdrückt. Hierfür ist es sinnvoll mit einem Zeitstempel zu arbeiten und nicht mit der Funktion time bzw. sleep() von Python. Die Funktion sleep stoppt alle weiteren Prozesse in der eingetragenen Zeit, sodass beispielsweise die Linienverfolgung nicht möglich ist. Da in diesem Fall jedoch nur vergangene Zeit gemessen werden soll, ist das Setzten eines Zeitstempels und das abfragen des Zeitstempels die bessere Wahl um die vergangene Zeit zu bestimmen. Wenn ein Einfahrt-Verboten-Schild in 30cm Distanz oder geringer erkannt wird, wird das Wenden um 180° eingeleitet und anschließend die Linienverfolgung fortgesetzt.

Bei dieser Lösung ist aufgefallen, dass die Linienverfolgung nicht flüssig erfolgt und dies häufig zu großen Korrekturen vom Schwarmroboter führt. Daher wurde die Linie nahezu im Zick-zack verfolgt.

Umfangreiche Suchen nach der Ursache haben ergeben, dass der Roboter aufgrund der aufwendigen Bildverarbeitung lediglich ein bis zwei Bilder pro Sekunde verarbeitet. Dementsprechend ist auch die Linienverfolgung nur mit wenigen Bilder versorgt,

wodurch heftige Korrekturen eingeleitet werden und der Roboter einer Linie nicht sauber folgen kann. Da die Bildverarbeitung bereits auf die minimal erforderlichen Maßnahmen optimiert ist, wird das Problem mit der Implementierung von parallelisierten Prozessen gelöst.

3.2.6 Parallelisierung der Linienverfolgung und Klassifizierung von Verkehrsschildern

Wie bereits in der Linienerkennung (Kapitel 3.1.3) beschreiben, wird im wiederholt überarbeiteten Stand die Linienerkennung in einem eigenen Thread gestartet. Die Linienerkennung ist in der Methode linetracking/line_tracking.py/track_line() enthalten. Aus dem Neigungswinkel der Konturen der Linie im Bild wird der erforderliche Einschlagwinkel der Hinterräder des Schwarmroboters bestimmt (siehe linetracking/pidcontroller.py). Die Linienerkennung wird im swarmrobot/swarmrobot.py mit der Methode _setup_autopilot() als Thread gestartet. Diese Methode wird nicht direkt aufgerufen, sondern über das Setzen des Parameters _track_active = True gestartet.

Für die Schilderkennung wird ein weiterer Thread ebenfalls über das Setzen von Parametern erstellt. Der Parameter _sign_detection_active = True, aktiviert die Klassifizierung von Schildern. Mit dem Parameter _show_only = True wird nur das Kamerabild mit den klassifizierten Schildern sowie der Distanz angezeigt. Wird der Parameter _drive_and_show = True gesetzt, wird während der Fahrt und der Reaktion auf die jeweiligen Schilder, das Kamerabild angezeigt. Wenn keiner der beiden zuletzt genannten Parameter gesetzt ist, wird die Klassifizierung sowie Reaktion auf Schilder ausgeführt, ohne die Anzeige des Videostreams der Raspberry Pi-Kamera in einem Fenster.

Die Klasse SignDetector ist ausschließlich für die Erkennung von Verkehrsschildern sowie die Auswertung der Distanz zum jeweiligen Schild und das Labeln eines klassifizierten Bildes zuständig. Alle Reaktionen des Schwarmroboters werden in der Klasse SignReactor implementiert. Hier wird das Stoppen am Stoppschild über ein Event getriggert und die Linienverfolgung anschließend wieder aktiviert. Sobald der Thread für die Erkennung von Verkehrsschildern ein Event setzt, unterbricht der Thread der Linienverfolgung seine Tätigkeit, bis das Event zurückgezogen wird. Für das Wenden bei

einem Einfahrt-Verboten-Schild wird ebenfalls ein Event zum Stoppen der Linienverfolgung gesetzt. Dann wird der TurnAssistant in navigation/turnassistant.py aufgerufen, um eine 180° Wendung zu vollziehen. Anschließend wird das Event „gecleart“ und die Linienverfolgung wieder aufgenommen. Besonders zuverlässig ist das Wenden auf einer geraden Linie. Wenn das Einfahrt-Verboten-Schild in eine Kurve gestellt wird, kann es Schwierigkeiten bei Wiederaufnahme der Linienverfolgung nach der 180°-Wendung kommen.

Mit der parallelisierten Vorgehensweise ist eine maßgebliche Verbesserung der Linienverfolgung im Test ersichtlich. Auch die Klassifizierung von Schildern und entsprechende Reaktion wird zuverlässig umgesetzt. Häufige Fehlerquelle ist, dass vor allem das stehende Schild nicht ausreichend im gegebenen Kamerawinkel erfasst wird. Daher ist zwingend erforderlich, dass der Kamerawinkel und die Positionierung des Schildes an der Fahrstrecke aufeinander abgestimmt werden.

3.3 Navigation über maschinenlesbare Codes

3.3.1 Ausgangspunkt

Beim übergebenen Stand findet die Navigation über Kommandos wie z.B. „rechts“, die in einem Quick-Response-Code (QR-Code) gespeichert sind, statt. Wenn der Roboter einen QR-Code erkennt, wird die Anweisung, die in ihm gespeichert ist, ausgeführt. Der Roboter verarbeitet diese Anweisungen, indem er die Räder in die entsprechende Richtung einschlägt und losfährt. Aus diesen Kommandos ergibt sich in einem Parcours eine eindeutige, aber auch immer gleiche Strecke, die gefahren wird. Die Erkennung der QR-Codes funktioniert grundlegend, die Zuverlässigkeit ist allerdings nicht zu 100% befriedigend.

3.3.2 Überarbeitung der QR-Codes

Die bisherige Navigation ist sehr trivial und in der Praxis eher unbrauchbar. Durch die festgelegten Kommandos, die fest auf dem Parcours verteilt werden, entsteht keine echte Navigation. Der Roboter kann hier nicht selbstständig entscheiden, in welche

Richtung sein Ziel liegt und an welcher Stelle er abbiegen muss, um dieses zu erreichen. Für die “echte” Navigation muss der Roboter, genauso auch wie im Straßenverkehr, Richtungsanweisungen mit dem entsprechenden Ziel, das in diese Richtung erreicht wird, bekommen.

Diese Idee wurde zunächst mit Statements wie “l:1;g:0;r:2” dargestellt, welche statt dem Kommando “rechts” im QR-Code gespeichert wurden. Hier gibt es die Informationen, dass es nach links zu Parkplatz 1, nach rechts zu Parkplatz 2 geht und geradeaus der Rundkurs liegt. Mittels dieser Richtungsanweisungen kann der Roboter zu einem definierten Ziel finden.

Nach mehreren Tests war jedoch festzustellen, dass QR-Codes als Mittel der Wahl nicht zuverlässig genug waren. Die Kombination aus den kleinen Pixeln eines QR-Codes, der relativ gering auflösenden Kamera des Raspberry Pis, sowie deren Neigungswinkel und den nicht immer zu beeinflussenden Lichtverhältnissen macht ein garantiertes Erkennen des QR-Codes nicht immer möglich. Je nach Lichtverhältnissen und Kamerawinkel kann hier die Erfolgsquote zwischen 70% und 0% schwanken. Dies schafft keine solide Grundlage, auf der auch noch zu einem späteren Zeitpunkt aufgebaut werden kann.

3.3.3 Änderung der Code Art

Da ein QR-Code bei einem Kamerawinkel von annähernd 45° optisch sehr verzerrt wird und das die Lesbarkeit stark beeinflusst, wurde nach einem Code gesucht, für den der Neigungswinkel kein Problem darstellt.

Die Wahl fiel hier auf einen Barcode der Art Code-128. Diese Barcodeart entspricht dem aktuellen Standard, der in allen Bereichen des Lebens Verwendung findet. Beispiele hierfür sind die Etikettierung von Lebensmitteln oder auch die Beschriftung von Paletten oder Gütern in einem Lager.

Die vorherig beschriebene Codierung kann auch hier benutzt werden. Ein Problem, das hieraus entsteht ist jedoch, dass die Barcodes sehr lang werden. Um diese in ausreichender Größe auf dem Parcours anbringen zu können, müssen diese kürzer werden. Der von den Barcodes behobene Nachteil der optischen Verzerrung, wird durch eine sehr längliche Form wieder zerstört. Bei sehr langen Barcodes reicht der

Blickwinkel der Kamera nicht aus, damit diese vollständig im Bild zu sehen sind. Die quadratischen QR-Codes hatten hier wiederum einen Vorteil.

Um die Barcodes so kurz wie möglich zu halten, muss die Codierung verkürzt werden. Hierbei kann bei den Richtungsangaben auf die Richtung verzichtet werden, indem ein Standard in der Reihenfolge festgelegt wird. So kann definiert werden, dass die erste Zahl das Ziel in linke Richtung, die zweite das Ziel in gerader Richtung und die dritte das Ziel in rechter Richtung angibt. Ebenso kann auf Trennzeichen verzichtet werden, wenn jedes Ziel mit einem einzelnen Zeichen dargestellt wird. So wird eine Länge von 3 Zeichen erreicht, die einen fast quadratischen Barcode ergibt.

Diese Implementierung ist schon fast zufriedenstellend. Weitere Tests haben allerdings gezeigt, dass der Barcode noch etwas größer sein müsste, um immer zuverlässig erkannt zu werden. Jedoch wird damit der Barcode wieder etwas zu lang, um ebenfalls immer zuverlässig komplett im Bild zu sein. Die Lösung dieses Problems wird durch eine weitere Verkürzung des Barcodes erreicht, um ihn damit wirklich quadratisch zu machen und ihn auf die Maße des QR-Codes zu bekommen. Mit einer Barcodegröße von circa 10,5 cm wurden sehr zuverlässige Ergebnisse erzielt.

Bei der Auswertung der Barcodes wurde schon eine Standardrichtung festgelegt, um definitiv immer eine Richtung zu haben, in die der Roboter weiterfahren kann. Falls bei der Auswertung der Barcodes keine passende Richtung ermittelt werden kann, fährt der Roboter geradeaus weiter. Das bedeutet, dass das zweite Zeichen im Code weggelassen werden kann, wenn davon ausgegangen werden kann, dass wenn in linke und rechte Richtung nicht das richtige Ziele liegt, dieses in gerader Richtung liegen muss. Damit ist der Barcode auf zwei Zeichen verkürzt und hat das richtige Format.

3.4 Erkennung der Barcodes

Ein Schwarmroboter kann an eine Kreuzung grundsätzlich aus jeder Richtung heranfahren. Das bedeutet, dass er aus jeder Richtung kommend wissen muss, in welche Richtungen welches Ziel liegt. Das hat zur Folge, dass an einer Kreuzung vier Barcodes platziert werden müssen. Per Definition scannt der Roboter immer den Barcode im linken, oberen Eck der Kreuzung. Um dabei nicht von den anderen Barcodes verwirrt zu werden, müssen diese aus dem zu analysierenden Bild entfernt werden. Dazu

wird die Funktion get_left_upper_corner_intersection() in detection/intersection_detection.py aufgerufen, die anhand von der ermittelten Kreuzung das Bild passend zuschneidet. Der Kreuzungspunkt spiegelt dabei die untere, rechte Ecke des Bildausschnittes wider.

Um zusätzlich schlechte Lichtverhältnisse auszugleichen, werden einige Operationen auf dem erzeugten Bildausschnitt durchgeführt. Dabei wird das Bild geschärft und die Farben intensiviert. Heraus kommt dabei ein Bild, dass den passenden Barcode mit bestmöglichem Kontrast beinhaltet. Dieser wird dann mittels “pyzbar”-Bibliothek in der Funktion detect_barcode() in detection/barcode_detection.py ausgelesen. Diese Bibliothek ist in Python hierfür der allgemeine Standard und funktioniert auch recht zuverlässig, solange das Bild einen gut lesbaren Barcode enthält.

Der Inhalt des Barcodes wird anschließen ausgewertet, um die möglichen Richtungen zu ermitteln.

3.5 Fahren durch den Parcours

3.5.1 Abbiegen, Wenden und Parken

Ein Roboter, der durch den Parcours fährt und an eine Kreuzung kommt, scannt den dort angebrachten Barcode, um sich für seine Weiterfahrt zu orientieren. Wenn ein Barcode erkannt wurde und die optimale Richtung feststeht, muss in zwei von drei Fällen abgebogen werden. Der Schwarmroboter hat dabei allerdings nicht die Möglichkeit in einem Zug seine Richtung zu ändern. Auf Grund des zu geringen Einschlagwinkels der Räder wird in drei Zügen abgebogen. Die Funktionalität hierzu ist im TurnAssistant in navigation/turnassistant.py zu finden.

Zum Abbiegen wird zuerst in Abbiegerichtung eingeschlagen und ein Stück nach vorne gefahren, dann wird entgegengesetzt zur Abbiegerichtung eingeschlagen und zurückgesetzt und im Anschluss nochmal in Abbiegerichtung eingeschlagen und wieder ein Stück nach vorne gefahren. Nach dieser Prozedur steht der Roboter passend auf der Linie und setzt die Linienverfolgung fort.

Bei einer Wendung um 180° wird das gleiche Prinzip angewendet, allerdings doppelt so oft, da der Roboter sich auch, im Vergleich um abbiegen um 90°, doppelt so weit drehen muss.

Der Parkvorgang ist dabei sehr ähnlich, da der Roboter rückwärtseinparkt. Hierbei wird erst eine 180° Wendung durchgeführt und im Anschluss nochmal eine Fahrzeuglänge zurückgesetzt, um auf dem Parkplatz zu stehen.

Ein Parkplatz ist dabei als rotes Quadrat gekennzeichnet. Der Roboter findet diesen sehr ähnlich zur Kreuzung, da sich auch hier die Fahrtlinie mit der Begrenzung des Parkplatztes kreuzt. Wenn das Bild, das normalerweise für die Barcodeermittlung genutzt wird, eine rote Linie enthält, dann ist das das Kommando zum Einparken. Beim Erkennen des Parkplatzes kann leider nicht auf eine Implementierung zurückgegriffen werden, bei der ein komplettes, rotes Viereck erkannt wird. Da auf Grund des Kamerawinkels nicht garantiert werden kann, dass dieses Viereck komplett zu sehen ist. Um eine Erkennung zu ermöglichen, muss eine einzelne rote Linie ausreichen. Das ist generell weniger zuverlässig. Dieser Nachteil muss dann über die Richtlinien zum Aufbau eines Parcours ausgeglichen werden, welcher den Aufbau festlegt. Rote Linien müssen darin für Parkplätze vorbehalten sein.

Die rote Linie wird dabei über eine Maske erkannt. Dabei wird ein unterer und oberer Farbschwellwert definiert. Wenn die Maske über das Bild gelegt wird, dann bleiben die Pixel erhalten, die zwischen diesen Farbwerten liegen. Wenn hier nun eine Linie erkannt wird, dann handelt es sich um einen Parkplatz.

3.5.2 Allgemeine Empfehlungen für den Aufbau

Der Parcours sollte generell eher großzügig geplant werden, damit der Schwarmroboter immer genug Platz zum Manövrieren hat. Grundsätzlich sollte für eine Kreisbahn wenigstens 80cm Durchmesser gewählt werden, damit der Roboter diese ohne größere Korrekturen befahren kann.

Von rechtwinkligen Kurven sollte generell Abstand genommen werden. Ohne Korrektur können diese überhaupt nicht gefahren werden. Hier wäre höchstens eine Behand-

lung als Kreuzung möglich, allerdings müssten sich hierfür die Linien tatsächlich kreuzen. Diese Streckenplanung bedeutet mehr Nachteile, als dass sie Vorteile bringt und sollte deshalb vermieden werden.

Ebenso sollte vor Kreuzungen wenigstens eine Fahrzeuglänge gerade Strecke existieren. So ist garantiert, dass der Roboter an der Kreuzung passend ausgerichtet ist und dieser, wenn er über die Kreuzung fährt, die Fahrlinie nicht direkt verliert.

Zwischen Kreuzungen oder Parkplätzen sollte ebenfalls wenigstens ein bis zwei Fahrzeuglängen Platz sein. Dieser Platz wird benötigt, um dem Roboter genug Zeit zu geben wieder auf eine neue Situation zu reagieren.

Wenn diese Vorgaben eingehalten werden, steht einer generellen Verwendung des Programms nichts im Weg.

3.5.3 Testaufbau für maschinenlesbare Codes und Parkplätze

Für den Testaufbau wurde ein Rundkurs mit zwei Schleifen, zwei Kreuzungen und zwei Parkplätzen gewählt (Abbildung 3.11). Dieser Kurs ermöglicht ein generelles Befahren des Roboters in einer Schleife, sowie dem Parken an den Parkplätzen. Dazwischen werden regelmäßig die Kreuzungen befahren, um das Abbiegeverhalten zu demonstrieren.

Die Barcodes werden so platziert, dass aus jeder Richtung kommend, immer der linke, obere Barcode gelesen werden kann. Der Barcode sollte dabei mindestens zwei bis drei Centimeter von der Linie rechts und ein bis zwei Centimeter oberhalb der unteren Linie angebracht werden. Kleber oder Klebeband sind dabei ratsam, da der Roboter diese überfährt und sonst verschiebt. An einer Kreuzung ergeben sich somit vier Barcodes, die in jeweils unterschiedliche Richtungen zeigen.

Nach der in Kapitel 3.3.3 Logik beschreibt die erste Ziffer des Barcodes welches Ziel in linke Richtung erreicht werden kann, die zweite Ziffer beschreibt die rechte Richtung.

Auf diese Funktionsweise ist bei der Anbringung der Codes zu achten. Die Barcodes für den Testparcours sind im Hauptordner des Projektes zu finden.

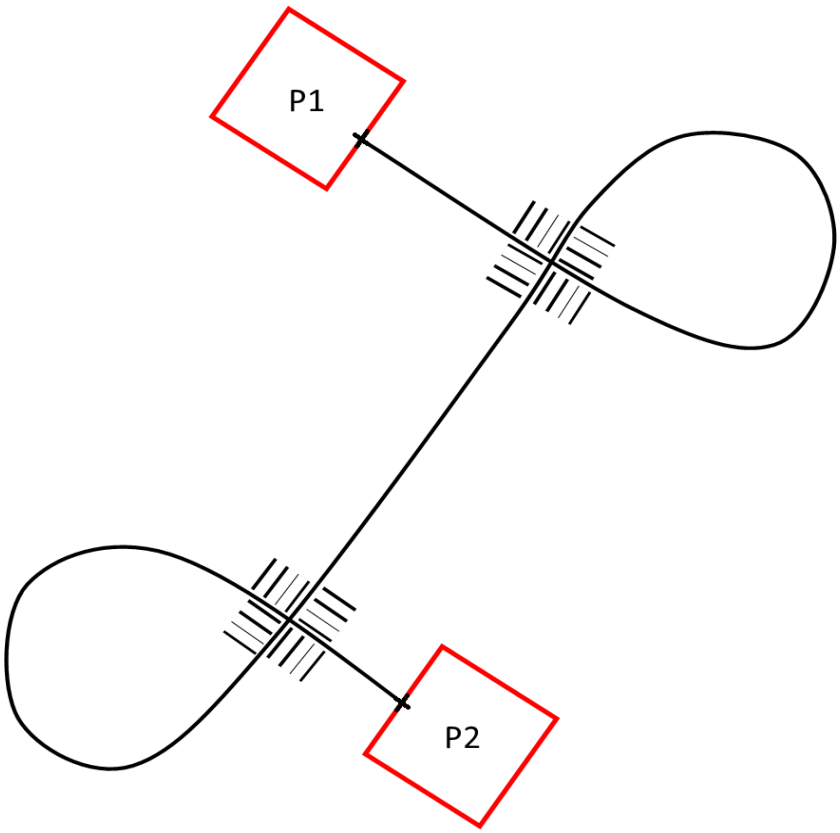

Abbildung 3.11: Parcours für Navigation mit maschinenlesbaren Codes
Quelle: eigene Darstellung

3.5.4 Testaufbau für Linienverfolgung und Verkehrsschilderkennung

Um sowohl die Linienverfolgung auf geraden und kurvigen Strecken darzustellen, kann beispielsweise ein Rundkurs, mit der im Bild dargestellten Form, eingesetzt werden. Hierbei ist, wie vorangegangen erwähnt, zu beachten, dass die Kurven einen ausreichend großen Radius haben.

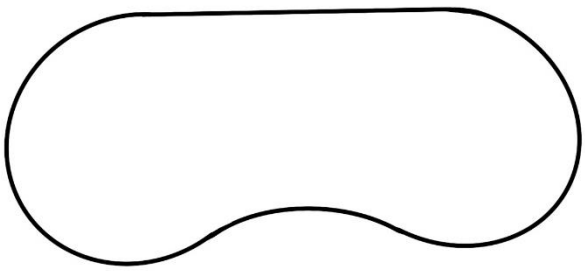

Abbildung 3.12: Rundkurs für Linienverfolgung und Verkehrsschilderkennung
Quelle: eigene Darstellung

Auf der gerade Strecke können die Schilder stehend oder auf dem Boden liegend positioniert werden. Auch eine Positionierung in den Kurven ins möglich. Beim Einfahrt-Verboten-Schild kann dies jedoch zu Schwierigkeiten bei der erneuten Aufnahme der Linienverfolgung nach dem Wenden führen. Generell ist die Positionierung auf den Kamerawinkel des Schwarmroboters abzustimmen. Dem Schwarmroboter muss die Möglichkeit geboten werden, dass Schild vollständig im Bild erfassen zu können. Es ist möglich auch andere Routenführungen als die beispielhafte Darstellung im Bild zu verwenden. Die eingesetzten Schilder sollten für eine akkurate Distanzerkennung jeweils einen Durchmesser von vier Zentimetern haben.

3.6 Verwendung der Programme

In diesem Abschnitt wird der Aufbau der Implementierung sowie die konkrete Verwendung der bereitgestellten Programme genauer beschreiben. Zusätzlich werden die, im Hintergrund ablaufenden Funktionen genauer beleuchtet, um das Verständnis der Programme zu vertiefen.

3.6.1 Aufbau der Ordnerstruktur

Den Ausgangspunkt der Ordnerstruktur stellt die “Studienarbeit/Swarmrobotlib_oldGen” in „home/pi“ dar. Hier finden sich alle notwendigen Dateien, um mit dem Roboter zu arbeiten. Ausführbare Dateien, die den Roboter starten sind direkt auf der obersten Ebene abgelegt. Diese tragen Titel wie beispielsweise “swarmrobot_navigation_program.py”.

Im Ordner “classifiers” sind die trainierten Classifier gespeichert. Diese beinhalten die Daten, um beispielsweise Schilder zu erkennen. Nach aktuellem Stand handelt es sich um Haar Cascade Classifier für Stoppschilder und Einfahrt-Verboten-Schilder.

Der Ordner “detection” beinhaltet die Programmteile, die für die Erkennung von verschiedenen Elementen genutzt werden. So zum Beispiel die Erkennung der Barcodes.

Dem ähnlich enthält der Ordner “detectionsign” die Funktionen, um Schilder zu erkennen und zu verarbeiten.

Im Ordner “linetracking” ist die Logik der Linienverfolgung enthalten.

Das Programm für die Navigation ist im Ordner “navigation” zu finden.

Das Herzstück der Anwendung, in dem alles zusammenläuft befindet sich im Ordner “swarmrobot”.

Und zuletzt der Ordner “utility”, der einige Hilfsprogramme, wie z.B. Funktionen für die Motoransteuerung oder den BrickPi3.

3.6.2 Linienverfolgung

Die Linienverfolgung besteht aus zwei Teilen. Der eine ist der LineTracker (linetracking.py) und der andere der PIDController (pidcontroller.py). Der LineTracker wird dabei mit der Größe des Kamerabildes initialisiert und hat die Optionen für eine Preview und das Debugging.

Die Funktion für die Linienverfolgung benötigt als Übergabeparameter das aktuelle Kamerabild, das Event und den Swarmrobot. Die Linienverfolgung wird dabei nur ausgeführt, wenn das Event nicht gesetzt ist. Das aktuelle Kamerabild wird in eine Grauskala umgewandelt und ein Blurr hinzugefügt, bevor es auf Linien untersucht wird. Die Grautöne verringern dabei die Datendichte und beschleunigen die Verarbeitung. Ein leicht verschwommenes Bild, erzeugt durch den Blurr, beseitigt Details, die zu fehlerhaftem Erkennen von Linien führen können. Im Bild werden mittels OpenCV die Linien in Form von Kontouren erkannt. Für diese werden danach der jeweilige Schwerpunkt ermittelt. Anhand dieses Schwerpunktes wird auch bestimmt, ob die Linie in den äußeren zehn Prozent des Bildes liegt, um das Verlassen der Linie vorzeitig zu erkennen.

Zurückgegeben wird die relative Position der Linie im Vergleich zur Mitte des Bildes. Dieser Wert wird an den PIDController übergeben.

Der PIDController ermittelt, anhand der vergangenen Lenkungsdaten und der aktuellen Position der Linie, wie der Roboter seine Räder einschlagen muss. Dieser bekommt nur diese Position übergegeben. Die restlichen Informationen, die zur Berechnung des Einschlagswinkels genutzt werden, sind im PIDController gespeichert.

Um die Linienverfolgung zu starten, kann das Programm swarmrobot_line_program.py verwendet werden. In diesem wird der Schwarmroboter zu Beginn, wie in jedem weiteren Programm auch, kalibriert. Bei der Kalibrierung ist anzumerken, dass der

Schwarmroboter das erste Mal nach dem Einschalten eine Fahrzeuglänge rückwärtsfährt. Bei allen weiteren Kalibrierungen tritt dies nicht auf.

Nach der Kalibrierung wird der Autopilot, der die Linienverfolgung widerspiegelt, aktiviert. Dieser setzt den oben beschriebenen Linetracker und PIDController ein. Zusätzlich muss die Geschwindigkeit über `bot.set_drive_power_lvl()` gesetzt und über `bot.change_drive_power_lvl()` gestartet werden. Es empfiehlt sich ein Wert zwischen 20 und 30. Die Standard-Linienverfolgung zeigt kein Videobild an. Das Videobild kann ich swarmrobot.py unter _setup_autopilot() > follow() aktiviert werden, indem der entsprechende Kommentar einkommentiert wird. Aussagekräftiger ist jedoch das Aktivieren des Previews indem der Preview-Parameter in der init-Methode auf True gesetzt wird, siehe hierfür swarmrobot.py > __init__() > self._line_tracker(…). Der Preview zeigt den gewählten Bildausschnitt inklusive der erkannten Kontouren sowie den Linienschwerpunkt an, der verfolgt wird.

3.6.3 Kreuzungserkennung

Für die Kreuzungserkennung wird die Intersection_detection.py verwendet. Der initialisierten IntersectionDetection wird in die Funktion “detect_intersection” das Kamerabild übergeben. Das Bild wird für die Weiterverarbeitung noch optisch bearbeitet. Mit OpenCV-Funktionen wie “erode” und “dilate” werden hinderliche Details entfernt. Mittels Kontourerkennung werden alle Linien im Bild gefunden und anschließend nach ihrer Ausrichtung sortiert. Die Schnittpunkte dieser Linien werden berechnet und zurückgegen, um die Kreuzungsdaten an anderer Stelle wiederverwenden zu können.

3.6.4 Barcodeerkennung

Die Barcodeerkennung besteht aus der BarCodeDetection. Diese beinhaltet die Funktion “detect_barcode”, der ein Bild übergeben wird. Dieses Bild wird mittels der Bibliothek Pyzbar auf Barcodes oder ähnliche Codes untersucht. Wenn ein solcher Code gefunden wurde, wird dieser ausgelesen und der Inhalt für die weitere Verarbeitung zurückgegeben.

3.6.5 Parkplatzerkennung

Bei der Parkplatzerkennung wird die ParkingSpaceDetection aus der parkingspace_detection.py genutzt. Hierbei wird das gleiche Bild, wie auch schon für die Suche nach Barcodes verwendet wurde, nach roten Linien untersucht. Dabei kommt eine Maske zur Verwendung, die über das Bild gelegt wird. Die untere Grenze hat dabei den Rot-Grün-Blau (RGB) -Wert (235, 36, 0) und den oberen RGB-Wert (255, 112, 67). Die Werte stehen dabei aus den Rot-, Grün-, und Blauwerten, aus der sich eine Farbe zusammensetzen lässt. Dieser definierte Bereich stellt dabei die Bildteile dar, die nach der Verwendung der Maske übrigbleiben und auf Kontouren untersucht werden. Die dabei gefunden Linien spiegeln dann einen Parkplatz wider.

3.6.6 Linienverfolgung und Verkehrsschilderkennung

Das Programm swarmrobot_sign_program.py triggert zuerst die Erstellung eines Threads für die Linienverfolgung. Anschließend wird die Erstellung eines Threads für die Klassifizierung von Verkehrsschildern sowie die angemessene Reaktion auf diese angestoßen. Mit dem Aufruf `bot.set_power_lvl(25)` wird die Geschwindigkeit gesetzt, hier ist ein Wert zwischen 20 und 30 empfehlenswert. Mit dem Aufruf `bot.change_drive_power_lvl()` startet der Schwarmroboter seine Fahrt mit der vorher gesetzten Geschwindigkeit.

Die Linienverfolgung geschieht wie bereits oben beschreiben. Für die Schilderkennung wird der aktuelle Frame mit dem SignDetector (detectioonsign/sign_detection.py) ausgewertet. Der SignDetector wandelt das Bild in eine Grauskala um. Anschließend wird die Methode detectMultiScale() von cv2.CascadeClassifier das Grau-Bild mit dem jeweils ausgewählten Classifier analysiert. Wenn ein Schild erkannt wurde, wird dessen Position zurückgegeben. Danach wird die Distanz mit der Methode distance_finder() bestimmt, wie in Kapitel 3.2.4 beschrieben. Der Prozess wird mit jedem Classifier wiederholt. Die ermittelten Schildnamen, die Positionen sowie die Distanzen werden in einer Liste an die detect()-Methode in swarmrobot.py zurückgeben. Wenn Schilder erkannt wurden, wird anschließend der SignReactor aufgerufen. Seine Aufgabe ist das Auslösen einer Reaktion, wenn das Schild in 30 cm Abstand oder weniger erkannt wurde. Die Reaktionen sind, wie in Kapitel 3.2.6 beschreiben, beim Stoppschild wird

ein Event gesetzt, um den Schwarmroboter für drei Sekunden anzuhalten. Zusätzlich wird ein Zeitstempel gesetzt und anschließend das Event gecleart, um die Linienverfolgung wieder zu starten. Wenn bereits ein Stoppschild erkannt und gehalten wurde, wird in den nächsten drei Sekunden der Fahrt kein weiteres Stoppschild erkannt, da der Schwarmroboter sonst mehrfach an einem Schild halten würde. Wird ein Einfahrt-Verboten-Schild in passendem Abstand klassifiziert, wird ebenfalls ein Event gesetzt, mithilfe des TurnAssistants gewendet und anschließend die Linienverfolgung wieder aufgenommen, siehe Kapitel 3.2.5.

Übersicht der Parameterkonfigurationen in swarmrobot_sign_programm.py bei bot.set_sign_detection_state:

- active=True => muss True sein, um die Schilderkennung zu aktivieren
- show_only=False => nur für Anzeige ohne Fahren gedacht
- drive_and_show=True/False => wahlweise, wenn False gewählt wird, wird kein Bild angezeigt, bei True wird ein Videostream angezeigt. Da die Bildanzeige von OpenCV nicht Threadsave ist, kann es in seltenen Fällen zu Fehlermeldungen kommen. Die Anzeige der Bilder ist teils geringfügig verzögert, dies fällt vor allem auf, wenn der Schwarmroboter beispielsweise auf ein Stoppschild reagiert, welches im angezeigten Bild noch eine Distanz größer 30 cm hat. Die Ausgaben auf der Kommandozeile sind in diesem Fall genauer und besser für die Echtzeitkontrolle der Reaktion des Schwarmroboters geeignet. Wie bereits erwähnt, kann die Distanzanzeige für auf dem Boden liegende Schilder abweichen. Die Funktion wird hierdurch jedoch nicht beeinträchtigt.

3.6.7 Klassifizierung von Verkehrsschildern | Anzeige

Mit dem Programm show_distance_to_sign.py wird das Kamerabild angezeigt und Stoppschilder sowie Einfahrt-Verboten-Schilder klassifiziert. Aktiviert wird dies mit dem Aufruf:

```
bot.set_sign_detection_state(active=True, show_only=True, drive_and_show=False)
```

Mit active=True wird ein Thread für die Schilderkennung über swarmrobot.py/_setup_sign_detection() gestartet. Der Parameter show_only=True sorgt dafür,

dass der Schwarmroboter das Kamerabild anzeigt und keine Reaktion auf die erkannten Verkehrsschilder startet. Eine Linienverfolgung ist in diesem Programm nicht enthalten, da es lediglich zur Demonstration der Schilderkennung gedacht ist.

Der angezeigte Videostream zeigt beispielsweise folgendes Bild:

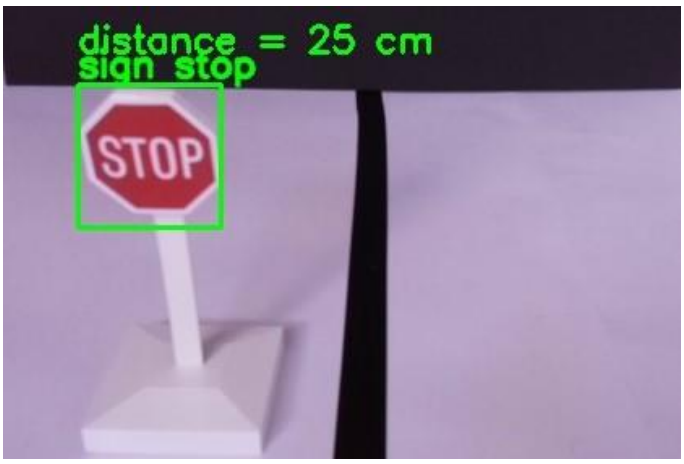

Abbildung 3.13: Anzeige der Distanz zu einem Verkehrsschild im Videostream
Quelle: eigene Darstellung

Es ist jeweils die gerundete Distanz in Zentimetern sowie der Name des klassifizierten Schilds enthalten.

3.6.8 Richtungsänderungen

Das Programm zur Richtungsänderung besteht aus dem Abbiegeassistenten (turnassistant.py). Dieser beinhaltet die Funktionen, um 90° abzubiegen, 180° zu wenden, rückwärts einzuparken oder ein Stück gerade zufahren. Dabei wird die jeweilige Funktion aufgerufen, welche direkt den, für die Fortbewegung definierten, Motor anspricht und diesen um eine bestimmte Anzahl an Grad dreht. Ebenso wird hier der Einschlagswinkel der Räder gesteuert, in dem der hierfür zuständige Motor angesprochen wird.

3.6.9 Navigation

Die Navigation findet über das Programm “navigation.py” statt. Der hier definierte Navigator wird wie auch der Linetracker initialisiert. Die Funktion “navigate” beinhaltet dabei die Logik, um sich im Parcours zurecht zu finden. Sie bekommt dafür das Kamerabild, sowie das Event übergeben. Über den Zugang zum Swarmrobot kann auf

die Kreuzungsdaten zugegriffen werden. Ist hier eine Kreuzung erkannt worden, wird der Roboter gestoppt und das Event gesetzt. Damit ist die Linienverfolgung pausiert. Das Bild der Kreuzung wird mittels Filter nachgeschärft, sowie Helligkeit und Kontrast werden angepasst. Die Kreuzung wird dann auf Barcodes, bzw. einen Parkplatz untersucht. Wenn mittels Barcodedetektor ein Barcode gefunden wird, wird der Inhalt ausgewertet, um die nächste Aktion zu bestimmen. Die Standardrichtung ist hierbei geradeaus. Wenn statt einem Barcode ein Parkplatz gefunden wurde, wird dementsprechend der Parkvorgang eingeleitet. Dabei wird auf die ParkingSpaceDetection zurückgegriffen. Nach einem erfolgreichen Abbiegevorgang wird die Linienverfolgung wieder aufgenommen. Nach dem Einparken stattdessen, endet die Fahrt. Die Logik beim Ermitteln der nächsten Richtung ist dabei folgende. Die erste Zahl der Barcodes wird mit dem festgelegten Ziel verglichen. Wenn diese übereinstimmen, wird die Fahrt in die linke Richtung fortgesetzt. Wenn nicht, wird die zweite Zahl des Barcodes betrachtet. Bei einer Übereinstimmung geht es nach rechts weiter. Falls beides nicht übereinstimmt, wird geradeaus gewählt.

3.6.10 Der Swarmrobot

Das Herzstück des ganzen ist die swarmrobot.py. Der darin definierte SwarmRobot verbindet alle anderen Funktionen miteinander. Hier werden alle wichtigen Konstanten definiert, wie beispielsweise die Belegung der Motoren. Außerdem sind hier die Funktionen implementiert, die für die Steuerung notwendig sind und die passenden Kommandos an den BrickPi weitergeben. Ebenso werden das Linetracking, die Navigation, die Kreuzungserkennung und die Schilderkennung als eigener Thread initialisiert.

Hierbei gibt es für jeden dieser Threads eine eigene Initialisierungsfunktion, über die die einzelnen Teile aktiviert werden können.

3.6.11 Zusammenstellung eines Programms

Um ein lauffähiges Programm für den Roboter aus den bisher beschriebenen Funktionen aufzubauen, muss zuallererst ein SwarmRobot initialisiert werden. Dieser muss

sich dann mit seiner “calibrate”-Funktion kalibrieren. Danach können schon die einzelnen Threads gestartet werden. Über Funktionen wie “set_autopilot_state” oder “set_navigation_state” kann hier mittels Übergabeparameter “active=True”, der Thread für die Linienverfolgung und die Navigation gestartet werden. Dann muss noch die Geschwindigkeit des Motors mit “set_power_lvl” und der Motor mit “change_drive_power_lvl” gestartet werden. Der Roboter wird nun bis zu einem definierten Ende in Form eines Parkplatzes oder einer abgelaufenen Zeit fahren.

Der beschriebene beispielhafte Aufbau wird in folgendem Ausschnitt dargestellt:

```python
from swarmrobot.swarmrobot import SwarmRobot
from time import sleep
import sys


def main():
    bot = SwarmRobot()
    # calibrate bot
    bot.calibrate(False, True)
    # setup automatic line detection
    bot.set_autopilot_state(active=True)

    # set velocity of bot
    bot.set_power_lvl(25)
    bot.change_drive_power_lvl()

    # duration of program in sec
    sleep(10)
    bot.stop_all()
    sys.exit("finished successfully")


if __name__ == '__main__':
    main()
```

Abbildung 3.14: Codebeispiel | Erstellung eines Programms für die Linienverfolgung
Quelle: eigene Darstellung

Zusätzlich zum Softwareteil muss auf die Einstellung der Kamera geachtet werden. Für die Navigation durch einen Parcours muss die Kamera mehr nach unten geneigt sein als für die Erkennung von Schildern. Das liegt am zu geringen Sichtfeld der Kamera, um gleichzeitig ein komplettes Bild von Kreuzungen für die Barcodeerkennung, sowie ein ausreichend geradeausschauendes Bild für die Schildererkennung zu bekommen. Dafür wurde eine Fixierung am Kamerahalter angebracht, um durch Umstecken zweier Streben das Kamerabild für den jeweiligen Modus zu erhalten. Die beste Einstellung für die Linienverfolgung ist: Strebe bei Kamerahalterung in oberem Loch und an der Gabelseite auch im oberen Loch. Beste Einstellung für Linienverfolgung mit Erkennung stehender Schilder: an der Kameraseite im mittleren Loch und auf der Gabelseite auch im mittleren Loch.

3.7 Programmverwendung | Kurzanleitung

- Linie verfolgen | ***swarmrobot_sign_program.py*** starten
- Stoppschild und Einfahrt-Verboten-Schild erkennen, während eine Linie verfolgt wird, inklusive stoppen oder wenden | ***swarmrobot_sign_program.py*** starten
- Nur die Distanz zu einem Stoppschild oder Einfahrt-Verboten-Schild anzeigen, Roboter bewegt sich nicht | ***show_distance_to_sign.py*** starten
- Navigation über Barcodes und einparken | ***swarmrobot_navigation_program.py*** starten
- Schwarmroboter sofort stoppen – Emergency stop | ***stop.py*** starten

4 Kritische Reflexion und Ausblick

4.1 Kritische Reflexion und Verbesserungspotenzial

Während dem Entwickeln und Testen der Funktionalitäten kamen immer wieder Probleme und Herausforderungen auf, die es zu lösen galt. Manche größer und dringender und manche, die eher Verbesserungsvorschläge für die Zukunft darstellen.

So kommt es bei der Kreuzungserkennung noch gelegentlich zu Problemen, wenn die Lichtverhältnisse schlecht sind. Dann wird beispielsweise keine Kreuzung erkannt und der Roboter überfährt diese in gerader Richtung. Hier besteht die Möglichkeit mittels Bildbearbeitung ein besseres Ergebnis erzielen. Noch mehr Rechenaufwand für den Raspberry Pi war allerdings während dieser Studienarbeit nicht zu rechtfertigen. Die Hardware war mit drei Threads, die das Kamerabild bearbeiteten und auswerteten hoch ausgelastet.

Ebenso kann es an Kreuzungen zu Problemen kommen, wenn der Roboter die Kreuzung zu spät erkennt, weil er nicht schnell genug neue Bilder von der Kamera bekommt. Das kann dazu führen, dass er erst anhält, wenn die Kreuzung nicht mehr im Bild ist oder der Barcode nicht mehr vollständig zu sehen ist. Bei dieser Limitierung kann auch die Hardware als Verursacher gesehen werden. Es müsste die gesamte Software optimiert werden, um aus dem einfachen Pi noch mehr Leistung herauszuholen oder es könnte auf ein neueres Modell mit generell mehr Rechenleistung umgestiegen werden.

Ein weiterer Punkt ist die Anzeige des Videostreams während dem Ablauf mehrerer Threads. Da cv2.imshow() nicht Threadsave ist, kommt es je nach Art und Position des Aufrufes gelegentlich zu Exceptiones. Wenn eine anspruchsvollere Kontrollanzeige des aktuellen Kamerabilds des Pis gefordert ist, sollte in diesem Bereich nach einer anderen, threadsicheren Lösung gesucht werden.

Ein anderes Problem stellte das Gehäuse dar. Nicht nur, dass dieses sehr umständlich beim Zusammenbauen ist, es ist außerdem nicht gut belüftet. Während den zahlreichen Tests wird der passiv gekühlte Pi immer heißer, was ebenfalls die Rechenleistung und damit den Ablauf der Software, negativ beeinflusste. Hier müsste nach einem anderen Modell mit besserer Belüftung und gegebenenfalls aktiver Kühlung geschaut werden.

Eine andere Schwachstelle ist die Kamera. Die Kamerabilder sind oftmals nicht hochauflösend genug, wie beispielsweise bei den QR-Codes gesehen werden konnte, und der Blickwinkel ist generell etwas zu schmal. Egal, ob es um das Erkennen der Barcodes oder der maximalen Entfernung, die ein Schild von der Linie entfernt stehen darf, geht. Oftmals würde hier ein etwas größerer Blickwinkel eine deutliche Verbesserung erzielen.

Eine weitere Schwachstelle der Hardware ist die Wendigkeit des Roboters. Mit seinem weitläufigen Wendekreis und der schwerfälligen Steuerung, ist er eigentlich nicht für enge Parcours geeignet. Wenn der komplette Aufbau überarbeitet werden sollte, dann ist es sehr zu empfehlen die Lenkung zu verbessern. Hierbei kann sich möglicherweise an einem echten Gabelstapler orientiert werden, um das Fahrverhalten positiv zu beeinflussen.

Auch im Code gibt es die ein oder andere Schwachstelle. Vorgänge wie beispielsweise das Wenden können verbessert und dynamischer gestaltet werden. Ebenso ist die Kommunikation zwischen den Threads ein weiterer Ansatzpunkt für Optimierungen. Im Großen und Ganzen sollte vor allem das Augenmerk auf der Optimierung liegen. Hierüber kann die Rechenleistung des Raspberry Pis noch besser genutzt werden.

Zusammenfassend können aber auch einige positive Punkte benannt werden. Durch den modularen Aufbau des Programms können dynamisch neue Funktionen angeschlossen werden. Die schon bereitgestellten Funktionen können in anderen Aufgabenbereichen eingebunden und sinnvoll genutzt werden. In der aktuellen Implementierung ist eine zuverlässige Linienverfolgung bereitgestellt. Nach dem aufgezeigten Schema können weitere Verkehrsschilder in das Classifier-Repertoire des Schwarmroboters aufgenommen werden. Das erläuterte Konzept für die Erkennung von maschinenlesbaren Codes kann auf weitere Szenarien angewandt werden. Und es ist möglich, dass der Roboter automatisch an Parkplätzen einparkt. Das erarbeitete Softwarekonzept mit der Möglichkeit für Parallelisierung wird als geeignet bewertet, um als Basis für neue Aufgabenstellungen verwendet und durch diese erweitert zu werden. Daher kann diese Studienarbeit als Erfolg für die Weiterentwicklung des Swarmlabs der DHBW Mosbach am Campus Bad Mergentheim verbucht werden.

4.2 Ausblick

Mit dieser Studienarbeit wurde generell ein guter Grundstein gelegt. Die erforderlichen Funktionen wurden erfolgreich umgesetzt und dokumentiert. Wenn dieses Projekt in zukünftigen Arbeiten wieder aufgegriffen wird, dann gibt es eine solide Grundlage, auf die aufgebaut werden kann. Die vorhergehend beschrieben Schwachstellen bieten sich als Ansatzpunkte an, um Verbesserungen umzusetzen und den Schwarmroboter, sowie das gesamte Swarmlab nach vorne zu bringen.

5 Literaturverzeichnis

[1] Pi My Life Up, „How to Upgrade Raspbian Stretch to Raspbian Buster," 31 01 2022. [Online]. Available: https://pimylifeup.com/upgrade-raspbian-stretch-to-raspbian-buster/. [Zugriff am 12 05 2022].

[2] Pi My Life Up, „How to Setup FTP on the Raspberry Pi," 30 01 2022. [Online]. Available: https://pimylifeup.com/raspberry-pi-ftp/. [Zugriff am 12 05 2022].

[3] GeekTechStuff, „Introduction To GitHub (Raspberry Pi)," 09 09 2019. [Online]. Available: https://geektechstuff.com/2019/09/09/introduction-to-github-raspberry-pi/. [Zugriff am 04 05 2022].

[4] GitHub Docs, „Creating a personal access token," o.J.. [Online]. Available: https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token. [Zugriff am 04 05 2022].

[5] HU Berlin, „GitLab-Dokumentation der HU Berlin | Git-Befehle," o.J.. [Online]. Available: https://pages.cms.hu-berlin.de/cms-webtech/gitlab-documentation/docs/git-befehle/. [Zugriff am 04 05 2022].

[6] C. B. Durmus, „OpenCV Python Installation for Raspberry Pi," 27 11 2021. [Online]. Available: https://singleboardblog.com/install-python-opencv-on-raspberry-pi/. [Zugriff am 13 05 2022].

[7] o.A., „DexterInd | BrickPi3," 23 05 2018. [Online]. Available: https://github.com/DexterInd/BrickPi3. [Zugriff am 07 11 2021].

[8] J. Finer, „How to Write Beautiful Python Code With PEP 8," o.J.. [Online]. Available: https://realpython.com/python-pep8/#naming-styles. [Zugriff am 16 05 2022].

[9] OpenCV, „OpenCV | Releases," 2022. [Online]. Available: https://opencv.org/releases/. [Zugriff am 11 05 2022].

[10] Learn Code By Gaming, „Training a Cascade Classifier," 22 08 2020. [Online]. Available: https://learncodebygaming.com/blog/training-a-cascade-classifier. [Zugriff am 11 05 2022].

[11] S. Albrecht, „Screenshot im VLC Media Player machen,“ 10 10 2013. [Online]. Available: https://praxistipps.chip.de/screenshot-im-vlc-media-player-machen_12483. [Zugriff am 11 05 2022].

[12] Institut für Neuroinformatik, „GTSRB | Dataset,“ 10 05 2019. [Online]. Available: https://benchmark.ini.rub.de/gtsrb_dataset.html. [Zugriff am 11 05 2022].

[13] J. Stallkamp, M. Schlipsing, J. Salmen und C. Igel, „Archive Meta Data,“ 10 05 2019. [Online]. Available: https://sid.erda.dk/public/archives/daaeac0d7ce1152aea9b61d9f1e19370/published-archive.html. [Zugriff am 16 05 2022].

[14] Institut für Neuroinformatik, „The German Traffic Sign Recognition Benchmark,“ 2019. [Online]. Available: https://benchmark.ini.rub.de/gtsrb_news.html. [Zugriff am 11 05 2022].

[15] A. Dal, „Distance(webcam) Estimation with single-camera | OpenCV-python,“ 18 12 2021. [Online]. Available: https://medium.com/mlearning-ai/distance-estimation-with-single-camera-opencv-python-298a96383c2b. [Zugriff am 11 05 2022].

[16] SPARKLERS, „Easiest way to install OpenCV for python in Raspberry pi within few minutes,“ 29 09 2019. [Online]. Available: https://www.youtube.com/watch?v=xlmJsTeZL3w&t=41s. [Zugriff am 12 05 2022].

[17] J. de Oliveira Neto, „Representatives of the 43 traffic sign classes in the GTSRB,“ o.J.. [Online]. Available: https://www.researchgate.net/figure/Representatives-of-the-43-traffic-sign-classes-in-the-GTSRB_fig6_270273697 . [Zugriff am 11 05 2022].

6 Anhang

1 Zuordnung der Kapitel

1 Einleitung

1.1 Aufgabenstellung und Ziele | **Teamleistung**

1.2 Vorgehensweise | **Teamleistung**

2 Systemumgebung und technische Grundlagen

2.1 Verwendete Software und Versionen

2.1.1 Raspbian OS | **Teamleistung**

2.1.2 Git | **Teamleistung**

2.1.3 Python | **Teamleistung**

2.1.4 OpenCV | **Teamleistung**

2.1.5 BrickPi | **Teamleistung**

2.1.6 BrickPi Motorenverkabelung | **Teamleistung**

2.2 Namenskonventionen | **Teamleistung**

2.2.1 einheitliche Benennung und Passwörter | **Teamleistung**

2.2.2 Python Naming Conventions | **Teamleistung**

3 Ausarbeitung

3.1 Linienverfolgung | **Teamleistung**

3.1.1 Ausgangspunkt | **Teamleistung**

3.1.2 Erste Überarbeitung | **Teamleistung**

3.1.3 Zweite Überarbeitung | **Teamleistung**

3.1.4 Probleme der Korrekturmaßnahme | **Teamleistung**

3.2 Klassifizierung von Verkehrsschildern | **Wierhake**

3.2.1 Ausgangspunkt | **Wierhake**

3.2.2 Erstellung eines Cascade Classifiers | **Wierhake**

3.2.3 Cascade Classifer erstellen | Kurzanleitung | **Wierhake**

3.2.4 Distanzbestimmung der klassifizierten Verkehrsschilder | **Wierhake**

3.2.5 Erste Überarbeitung der Klassifizierung von Verkehrsschildern | **Wierhake**

3.2.6 Parallelisierung der Linienverfolgung und Klassifizierung von Verkehrsschildern | **Wierhake**

3.3 Navigation über maschinenlesbare Codes | **Büttner**

3.3.1 Ausgangspunkt | **Büttner**

3.3.2 Überarbeitung der QR-Codes | **Büttner**

3.3.3 Änderung der Code Art | **Büttner**

3.4 Erkennung der Barcodes | **Büttner**

3.5 Fahren durch den Parcours | **Büttner**

3.5.1 Abbiegen, Wenden und Parken | **Büttner**

3.5.2 Allgemeine Empfehlungen für den Aufbau | **Teamleistung**

3.5.3 Testaufbau für maschinenlesbare Codes und Parkplätze | **Büttner**

3.5.4 Testaufbau für Linienverfolgung und Verkehrsschilderkennung | **Wierhake**

3.6 Verwendung der Programme

3.6.1 Aufbau der Ordnerstruktur | **Teamleistung**

3.6.2 Linienverfolgung | **Teamleistung**

3.6.3 Kreuzungserkennung | **Büttner**

3.6.4 Barcodeerkennung | **Büttner**

3.6.5 Parkplatzerkennung | **Büttner**

3.6.6 Linienverfolgung und Verkehrsschilderkennung | **Wierhake**

3.6.7 Klassifizierung von Verkehrsschildern | Anzeige | **Wierhake**

3.6.8 Richtungsänderungen | **Teamleistung**

3.6.9 Navigation | **Büttner**

3.6.10 Der Swarmrobot | **Teamleistung**

3.6.11 Zusammenstellung eines Programms | **Teamleistung**

3.7 Programmverwendung | Kurzanleitung | **Teamleistung**

4 Kritische Reflexion und Ausblick

4.1 Kritische Reflexion und Verbesserungspotenzial | **Teamleistung**

4.2 Ausblick | **Teamleistung**