

Hochschule für Angewandte Wissenschaften Hamburg

Fakultät Design, Medien und Information

Department Medientechnik

IT-Systeme

Prof. Dr. Torsten Edeler

Projektbericht  
Raspberry Car — Racingteam II

Jörn Kogerup, 2248604

Darius Weiberg, 2144123

Mirco Hülsemann, 2248464

20. Februar 2018

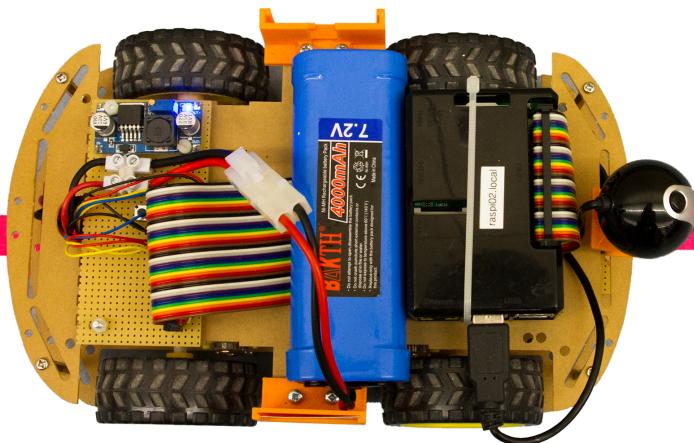


Abbildung 1: Auto von oben

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>3</b>
<b>2 Konzept</b>	<b>3</b>
2.1 Projektvorstellung . . . . .	3
2.2 Projektziel . . . . .	4
2.3 Die Rennstrecke . . . . .	4
2.4 Das Auto . . . . .	5
2.5 Konstruktion des Autos . . . . .	5
2.6 Programmierung . . . . .	6
2.7 Schaltplan . . . . .	6
<b>3 Code</b>	<b>7</b>
3.1 main.py . . . . .	7
3.1.1 import . . . . .	7
3.1.2 main . . . . .	8
3.1.3 line . . . . .	9
3.1.4 linienfahren . . . . .	11
3.1.5 lenken . . . . .	12
3.1.6 checkgreen . . . . .	14
3.1.7 checkblue . . . . .	15
3.1.8 makevideo . . . . .	16
3.2 setup.py . . . . .	17
3.3 aufraeumen.py . . . . .	18
<b>4 3D-Druck-Halterungen</b>	<b>18</b>
4.1 Kamerahalterung . . . . .	18
4.2 Ultraschallsensorhalterung . . . . .	19
<b>5 Probleme</b>	<b>20</b>
<b>6 Ergebnis</b>	<b>20</b>
<b>7 Geplante Weiterentwicklungen</b>	<b>21</b>
<b>Tabellenverzeichnis</b>	<b>22</b>
<b>Abbildungsverzeichnis</b>	<b>22</b>
<b>Literaturverzeichnis</b>	<b>22</b>

# **1 Einleitung**

Dieser Projektbericht dient der Dokumentation des Projekts „Entwicklung eines selbstfahrenden Autos“ im Modul IT-Systeme des Studiengangs Medientechnik an der HAW Hamburg im Wintersemester 2017/18.

Autonomes Fahren ist derzeit ein großes und hochaktuelles Forschungsthema vieler Universitäten, IT-Unternehmen und Automobilhersteller. Mit Hochdruck arbeiten Unternehmen wie Tesla, General Motors oder Waymo zusammen mit IT-Firmen wie Intel und Google daran, selbstfahrende Autos für die breite Masse verfügbar zu machen.

Aufgrund der aktuellen Relevanz und der Tatsache, dass ein selbstfahrendes Auto die ideale Verkörperung eines informationstechnischen Systems ist, wurde dieses Thema als Projekt für die Veranstaltung „IT-Systeme“ im Wintersemester 2017/2018 auserwählt.

Der vorliegende Abschlussbericht enthält Erläuterungen zum Projektziel, Projektumfeld sowie zu den gegebenen technischen Rahmenbedingungen. Es werden die einzelnen Schritte des Entwicklungsprozesses dokumentiert und Probleme genannt, die während der Entwicklung auftraten. Der Bericht endet mit einer Erörterung über die möglichen Weiterentwicklungen des Projekts.

## **2 Konzept**

### **2.1 Projektvorstellung**

Im Rahmen der Vorlesung IT-Systeme (5. Semester des Studiengangs Medientechnik) sollen die Studierenden in kleinen Gruppen ein autonomes Modellbau-Auto entwickeln, welches in der Lage sein soll, mithilfe von Sensoren und Kamera selbstständig eine vorher definierte Rennstrecke abzufahren. Die Aufgaben der Studierenden umfassen dabei:

- eine konzeptionelle Planung zur Umsetzung des Projekts,
- die Konstruktion des Modellbau-Autos (dabei wird ein Bausatz verwendet) sowie der Rennstrecke,
- die Auseinandersetzung mit der Funktionsweise der elektrischen Komponenten,
- die Verschaltung der elektrischen Komponenten,
- Programmierung und Implementierung des Steuer-Codes auf einem Raspberry Pi.

Als zeitlicher Rahmen ist ein Semester vorgesehen. Pro Woche gibt es eine dreistündige Einheit, in der die Studierenden unter Betreuung am Projekt arbeiten können. Es ist jedoch auch möglich, außerhalb dieser Einheit mit dem Projekt fortzuführen.

Es gibt zudem zwei Challenge-Termine im Semester, zu denen bestimmte Zwischenziele erreicht werden müssen. Auf diese Art wird das Vorankommen des Projekts überprüft und gewährleistet. Am Ende des Semesters erfolgt eine Generalprobe und schließlich die Projektausstellung.

## 2.2 Projektziel

Ziel des Projekts ist es, ein autonomes Auto zu entwickeln, welches mithilfe von Mikrocomputer, Kamera und Abstandssensoren zwei Runden auf einer definierten Rennstrecke selbstständig und erfolgreich absolviert. Dabei sollen die Autos einzeln starten, es befindet sich also nie mehr als ein Auto auf der Strecke. Die Gruppe, deren Auto den Kurs am schnellsten durchfährt, wird als Gewinner geehrt.

Auf dem Weg zu diesem Ziel gibt es im Laufe des Semesters zudem zwei Zwischenziele, so genannte Challenges, die erfolgreich absolviert werden müssen. Das erste Zwischenziel ist eine Geradeausfahrt von 10 Metern mit Geschwindigkeitswechseln, das zweite Zwischenziel ist das Fahren entlang einer Wand mithilfe von Abstandssensoren.

## 2.3 Die Rennstrecke

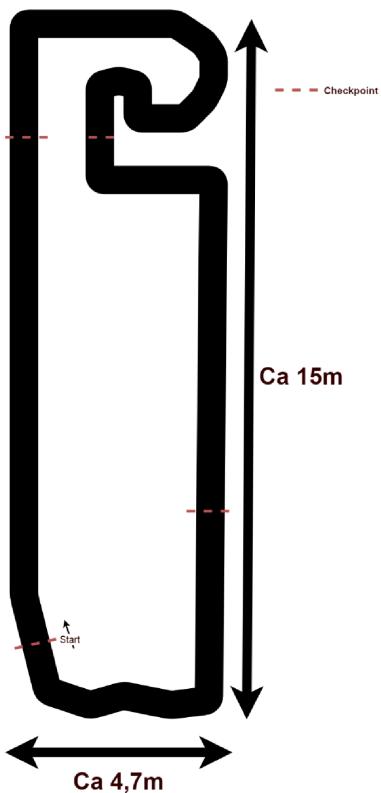


Abbildung 2: Rennstrecke

Die Rennstrecke, die es zu meistern gilt, wird am Anfang des Semesters grob definiert: Sie soll als geschlossene Rundstrecke (Startlinie = Ziellinie) in einem Laborraum der Hochschule aufgebaut werden. Im Wesentlichen soll sie aus einer Mittellinie sowie zwei Begrenzungslinien bestehen. Der Mittellinie gilt es zu folgen, die Begrenzungslinien dürfen nicht überschritten werden. Die Strecke soll außerdem mehrere Kurven mit verschiedenem Radius enthalten sowie einen geraden Abschnitt ohne Mittellinie entlang einer Wand. In diesem Abschnitt ohne Mittellinie sollen Abstandssensoren genutzt werden, um auf der Spur zu bleiben. Die finale, endgültige Version der Rennstrecke wird im Laufe des Semesters anhand der oben genannten Richtlinien von einer Gruppe im Kurs entworfen und ist in nebenstehender Abbildung zu sehen.

Die Linien werden mit Klebeband realisiert, welches auf den Boden des Laborraums aufgeklebt wird. Als Mittellinie dient rotes Tape, als Begrenzungslinie schwarzes.

Aufgrund von Problemen mit den verfügbaren Abstandssensoren, die sich erst im Laufe des Semesters ergeben haben, ist der gerade Abschnitt entlang der Wand in der endgültigen Version auch mit einer Mittellinie versehen. Die Aufgabe, mithilfe von

Abstandssensoren entlang einer Wand zu fahren, fällt somit – zumindest in der Generalprobe und Projektausstellung – weg.

Eine weitere Änderung der anfänglichen Version der Rennstrecke ist die Einführung einer Ampel an der Start- und Ziellinie. So sollen die Autos nur bei grünem Licht starten und automatisch anhalten, wenn das Licht blau geschaltet ist. Die Ampel wird durch einen LED-Streifen auf dem Boden realisiert.

## 2.4 Das Auto

Die Komponenten, die für den Bau des Autos verwendet werden dürfen, sind vorgegeben und werden von der Hochschule bereitgestellt:

- Einfacher Modellbau-Auto-Bausatz, bestehend aus Rahmen, DC-Getriebemotoren und gummierten Rädern
- 7,2V NiMH-Akku mit einer Kapazität von 4000 mAh
- Motortreiber L298N für DC-Motoren mit Dual-H-Bridge
- Raspberry Pi 3 Model B (4-Kern-CPU, 1,2 GHz, 1 GB Ram, WLAN, Bluetooth)
- USB-Webcam (640 × 480 Pixel, 30 fps, manueller Fokus)
- Ultraschallsensor HC-SR04 (messbare Distanz: 2–300 cm, Auflösung: 3 mm, max. 50 Messungen pro Sekunde)

Alle Komponenten (mit Ausnahme des Raspberry Pis) wurden im Hinblick auf einen möglichst günstigen Einkaufspreis ausgewählt. Dementsprechend sind die Erwartungen an Haltbarkeit, Genauigkeit und Ausfallsicherheit der Produkte und letztendlich an die Performance des Autos nicht allzu hoch.

Neben den aufgeführten Komponenten steht den Studierenden eine Werkstatt mit 3D-Drucker, Lötgerät und einer Vielzahl an Elektronikbauteilen (Kabel, Widerstände, Platinen ...) zur Verfügung.

Als Programmiersprache wird Python verwendet. Diese universelle Sprache zeichnet sich durch einfache und leicht verständliche Syntax, große Nutzerbasis sowie mächtige Bibliotheken aus und ist quasi Standardsprache auf dem Raspberry Pi.

## 2.5 Konstruktion des Autos

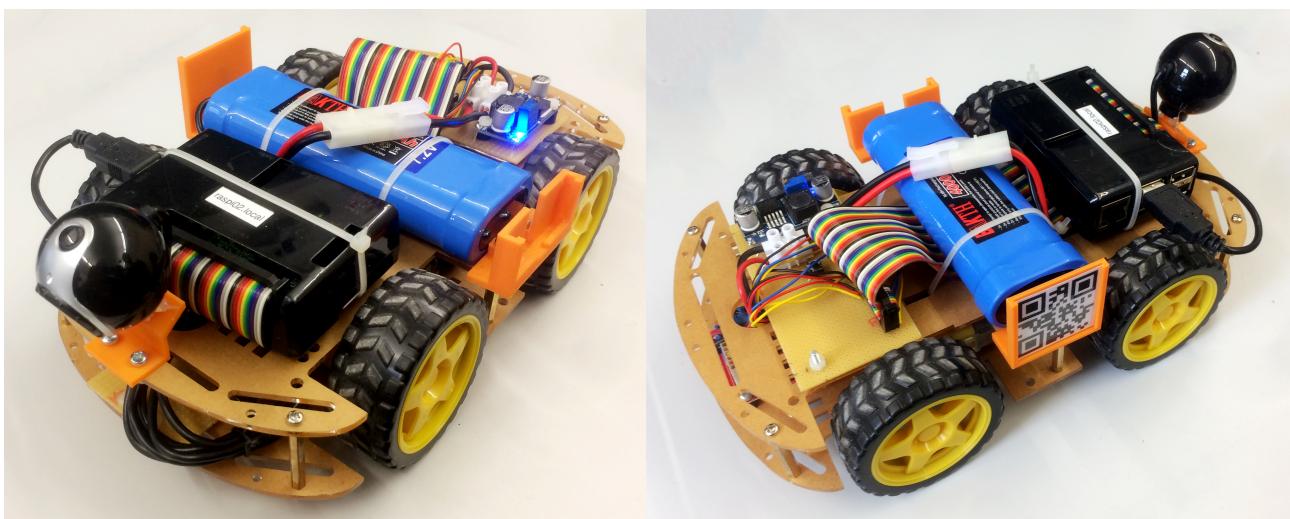


Abbildung 3: Das Auto im finalen Aufbau

Schon seit Beginn des Projekts sollten alle Bestandteile des Autos kompakt verbaut werden. Die Bauteile wurden fest angebracht und sind so angeordnet, dass sie das Gewicht des Autos gleichmäßig verteilen. Auf besondere äußerliche Ausstattungsmerkmale wurde deshalb bewusst verzichtet, um mehr Geschwindigkeit zuzulassen und die Motoren weniger zu belasten.

## 2.6 Programmierung

Da die Rennstrecke im Wesentlichen daraus besteht, einer roten Linie zu folgen, haben wir uns bereits früh mit der Bildverarbeitung beschäftigt. Dabei wurde in der Gruppe parallel an der Wandfahrt mit den Ultraschallsensoren und an der Linienfahrt mit der Kamera gearbeitet.

Das Auto soll sich in der Fahrweise der Strecke so anpassen, dass es bei geraden Abschnitten mit Höchstgeschwindigkeit fährt und je nach Kurvenradius etwas langsamer werden muss, ohne viel an Tempo zu verlieren. Die Lenkung ist ebenfalls, je nach Position der roten Linie, unterschiedlich stark und soll dennoch überall sanft und stufenlos geregelt werden. Hierfür setzt man die veränderliche Position der roten Linie ins Verhältnis zum theoretischen Idealwert des Mittelpunktes und berechnet daraus die Lenkung und Geschwindigkeit.

Nachdem das Grundgerüst für die Linienfahrt innerhalb weniger Wochen erarbeitet wurde, gab es genug Zeit für die Optimierung des Algorithmus. Funktionen wie die Erkennung von verschiedenfarbigen Ampeln oder die Videoaufnahme während der Fahrt wurden erst am Schluss hinzugefügt.

## 2.7 Schaltplan

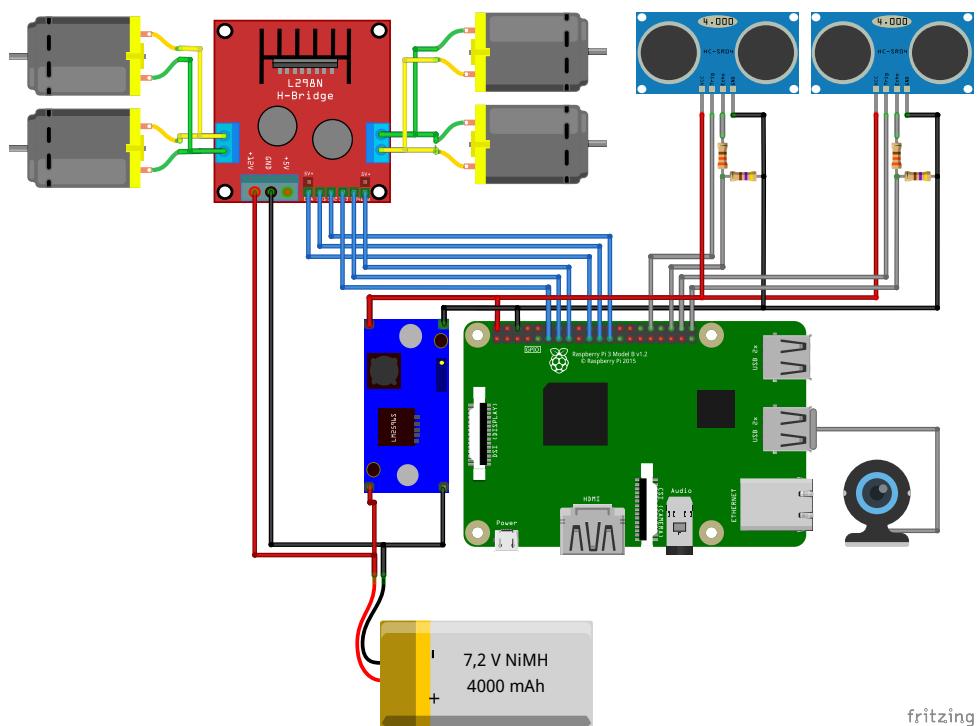


Abbildung 4: Schaltplan

In dem Auto sind insgesamt vier DC-Motoren verbaut, die in zwei Seiten links und rechts eingeteilt werden. Um die Motoren über den Raspberry Pi anzusteuern, werden sie jeweils als Gruppe an die Dual-H-Bridge angeschlossen, die als Motortreiber dient. Darüber kann mithilfe einer einfachen Logiktabelle die Stellung des Schalters und damit die Bewegung der Motoren eingestellt werden.

Die Ultraschallsensoren SC-HR04 arbeiten mit einem High-Pegel von 5 V. Für den Echo-Pin, der mit dem Raspberry Pi verbunden ist, muss zwingend ein Spannungsteiler verwendet werden, da die GPIOs vom Raspberry Pi nur maximal eine Spannung von 3,3 V erlauben. Die Webcam wird per USB angeschlossen.

Die gesamte Stromversorgung des Autos läuft über den 4000 mAh großen Akku. Der eingebaute Gleichspannungsregler sorgt dafür, dass nur 5 V am Raspberry Pi und den Sensoren anliegen.

## 3 Code

Das Programm wird über die Datei *main.py* ausgeführt. Zwei Module *setup.py* und *aufrauem.py* sind ausgelagert, dadurch ist die Hauptdatei übersichtlicher und die Module können einfacher von anderen Dateien mitbenutzt werden.

### 3.1 main.py

In *main.py* wird das Modul *Threading* verwendet, was gleich mehrere Vorteile hat. Die Bildaufnahme ist außerordentlich schneller, wenn sie auf einem anderen Thread als der restliche Algorithmus läuft und das Abspeichern der Bilder für eine Videoausgabe würde ohne das Threading die Rechendauer um ein Vielfaches erhöhen. Außerdem können so andere Funktionen unabhängig von dem restlichen Programm problemlos ausgeführt werden.

Es laufen drei Threads parallel: Der eigentliche Fahralgorithmus *linienfahren*, die Videoausgabe *makevideo* und die Ampelüberprüfung *checkblue*. *linienfahren* ruft zwei Funktionen auf, *line* zur Bildaufnahme und *lenken* zur Motorsteuerung.

Es werden mehrere globale Variablen zwischen den einzelnen Threads benutzt. Zum einen die Bildvariablen *ret* und *img*, damit nicht jede Funktion eigene Bilder aufnehmen muss und zum anderen der Linienmesswert *x* und die verstrichene Zeit *minutes*, *seconds*, die zur Videoausgabe benötigt wird.

#### 3.1.1 import

```
1 import threading      # Modul threads
2 import cv2            # Dies ist die Bildverarbeitungsbibliothek OpenCV
3 import numpy as np    # Rechnen mit vielen Zahlen in einem Array (z. B. Bilder)
4 import math           # Modul math
5 import time           # Modul time

7 from aufrauem import aufrauem, losfahren, bremsen # Funktion für start/stop importieren
8 from setup import *   # GPIO Setup importieren und ausführen

9
10 cap = cv2.VideoCapture(0)  # Input 0
11 # Codec und VideoWriter object für Video Output
```

```

1   fourcc = cv2.VideoWriter_fourcc(*'XVID')
2   13  out = cv2.VideoWriter('output.avi',fourcc, 15, (640,480))
3   ret, img = cap.read()

```

Am Anfang der Hauptdatei werden die erforderlichen Module importiert, gefolgt von den beiden ausgelagerten Dateien. Von *aufrauemen.py* werden die benötigten Funktionen und von *setup.py* wird alles importiert, da diese Datei keine Funktion enthält. In den Zeilen 10 bis 14 werden die notwendigen Einstellungen für OpenCV vorgenommen und die Kameraausgabe vorbereitet.

### 3.1.2 main

```

1   def main():
2       losfahren()
3       pr.start(0) # Motor A, speed Tastverhältnis
4       pl.start(0) # Motor B, speed Tastverhältnis
5
6       run_event = threading.Event()
7       run_event.set()
8
9       th1_delay = .01    # sleep dauer der Funktion
10      th2_delay = .01    # sleep dauer der Funktion
11      th3_delay = .001   # sleep dauer der Funktion
12      th1 = threading.Thread(target=linifahren, args=(th1_delay, run_event)) # Funktion in
13      einem neuen Thread zuordnen
14      th2 = threading.Thread(target=checkblue, args=(th2_delay, run_event))      # Funktion in
15      einem neuen Thread zuordnen
16      th3 = threading.Thread(target=makevideo, args=(th3_delay, run_event))      # Funktion in
17      einem neuen Thread zuordnen
18
19      th3.start() # Video Thread starten
20
21      no_green = 0
22      print("Warten auf grüne Ampel")
23
24      while no_green < 500:
25          no_green = checkgreen()
26
27      th1.start() # Linienfahren Thread starten
28      th2.start() # Ampel blau Test Thread starten
29
30      # Warten bis Strg+C gedrückt wird:
31      try:
32          while 1:
33              time.sleep(.01)
34
35      except KeyboardInterrupt:
36          print("attempting to close threads. Max wait =", max(th1_delay, th2_delay, th3_delay))
37          #
38          bremsen()
39          run_event.clear()
40          th1.join()
41          print("Thread 1 closed")
42          th2.join()
43          print("Thread 2 closed")
44          th3.join()
45          print("Thread 3 closed")
46          aufraeumen()
47          print("Threads successfully closed")
48
49          cap.release()
50          out.release()

```

```

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

```

Das Programm wird in *main* gestartet und beendet. Als erstes werden die Motoren mit der Funktion *losfahren* gestartet, allerdings noch mit einem Tastverhältnis von 0%. Anschließend wird das Threading aktiviert. Dafür wird *run\_event*, das als *while*-Bedingung für die Threads dient, gesetzt. Die Delay-Angaben und die drei verwendeten Threads *th1*, *th2* und *th3* werden definiert. Dabei ist *target* die Funktion, die in dem neuen Thread laufen soll und *args* sind die Variablen, die mit übergeben werden.

Der erste Thread (die Videoaufzeichnung) wird gestartet, gefolgt von einer *while*-Schleife, die überprüft, ob die Ampel grün ist. Wenn der zurückgegebene Wert der *checkgreen*-Funktion größer gleich 500 ist, also ausreichend Grünanteile erkannt wurden, endet die Schleife. Dadurch werden die anderen beiden Threads gestartet, wodurch das Auto losfährt.

Damit das Programm wieder beendet werden kann, folgt eine *try-while*- und *except*-Schleife, die weiterhin keine andere Funktion hat. Sobald ein *KeyboardInterrupt* durch das Drücken von Strg-C auftritt, bremst das Auto und die Threads werden geschlossen. Allerdings muss dabei gewartet werden, bis die *while*-Schleifen aller Threads durchgelaufen sind. Anschließend werden die GPIO-Einstellungen bereinigt, die Videoaufnahme beendet und das Video gespeichert.

Gestartet wird die *main*-Funktion durch die Abfrage *if \_\_name\_\_ == "\_\_main\_\_"*, um die Datei auch als Modul verwenden zu können.

### 3.1.3 line

```

1 def line(zeileNr):
2     global img, ret
3     ret, img = cap.read()
4
5     img_r = img[zeileNr, :, 2] # Alles aus zeileNr und Breite und Farbkanal 2
6     img_g = img[zeileNr, :, 1]
7     img_b = img[zeileNr, :, 0]
8
9     zeile_bin = (img_r.astype('int16') - (img_g / 2 + img_b / 2)) > 60 # Rotanteil über
10    Threshold
11
12    # Mittelpunkt berechnen und return:
13    if zeile_bin.sum() != 0:
14        x = np.arange(zeile_bin.shape[0]) # x=0,1,2 ... N-1 (N=Anzahl von Werten in zeile_bin)
15    else:
16        return None

```

In *line* wird für eine Bildzeile der Mittelpunkt der roten Anteile ermittelt. Hierfür wird zunächst ein Bild von der Webcam mit der *cap.read()*-Funktion aufgenommen und als Variable definiert, wodurch das Bild nicht abgespeichert werden muss.

Der Funktion wird beim Aufrufen nur die Zeile 70 (*zeileNr*) übergeben, die sich für unsere Kameraposition besonders gut eignet, um bereits frühzeitig Kurven zu erkennen. Alle anderen Zeilen sind für die Bildverarbeitung nicht relevant und würden den Algorithmus verlangsamen. Außerdem wird ermöglicht, dieselbe Funktion zum Analysieren unterschiedlicher Zeilen zu ver-

wenden. Dies wird in der finalen Version aber nicht benutzt, da das Auslesen einer einzigen Zeile zu keinem dauerhaften Verlust der Linie führt.

Als erstes werden die drei Farbkanäle B, G und R aus dem Bild extrahiert (Zeile 5–7). Es lässt sich nicht nur anhand des Rotkanals feststellen, ob die Linie im aufgenommenen Bild wirklich rot ist, da zum Beispiel die Farben Weiß und Gelb ebenfalls einen Rotanteil von 100 % besitzen. Dennoch kann die „Rotheit“ des Bildes durch folgende Formel einfach berechnet werden:

$$\text{Rotheit} = R - \frac{G + B}{2} \quad (1)$$

Durch die Anwendung der Formel wird nur eine hohe Rotheit erzielt, wenn im Bild auch ein rotes Pixel vorliegt. Die Umrechnung zum Datentyp *int16* ist erforderlich, damit bei Überschreitung des Wertes 255 noch Platz nach oben ist (bis 16 Bit) und nicht fälschlicherweise wieder bei null weitergezählt wird.

Die Festlegung eines Schwellwertes, in diesem Fall 60, hat den Zweck, die niedrigen Rotwerte herauszufiltern, die nicht Teil der Linie sind. Die Variable *zeile\_bin* gibt dann als Ergebnis ein Array aus, in dem entweder *True*- oder *False*-Werte für die jeweiligen Pixel eingetragen sind (Zeile 9). Danach wird durch Multiplikation der Variablen *x* bestimmt, an welchen Positionen der Zeile jeweils ein *True* ausgegeben wurde, also in welchem Bereich des Bildes die Linie zu finden ist.

Für die Berechnung des Mittelpunktes wird genauer betrachtet die Formel des Schwerpunktes angewendet. Dafür wird die Summe der *x*-Positionen aller Pixel mit einer Rotheit von über 60 durch die Anzahl dieser vorkommenden Pixel geteilt (Zeile 14). Dies entspricht genau der Mitte zwischen dem ganz linken und rechten Pixel der roten Linie.

Falls das Auto an einer Stelle die Linie verliert, werden auch keine roten Pixel gezählt. Für solche Fälle wurde zusätzlich die *if*-Abfrage eingebaut, um eine Division durch Null auszuschließen. Als Ergebnis wird *None* ausgegeben und im Hauptprogramm auf den letzten gültigen berechneten Mittelpunkt zurückgeführt.

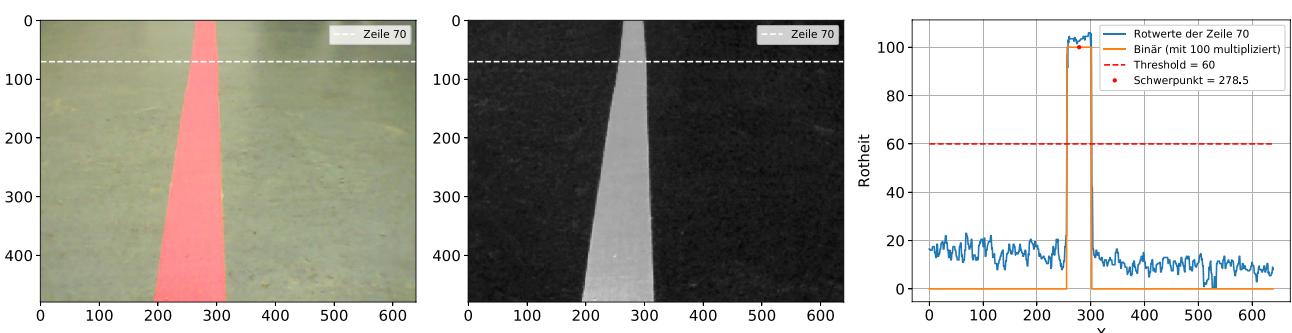


Abbildung 5: Berechnung des Mittelpunktes für die Zeile 70

Für das Testbild in Abbildung 5 wird folgender Mittelpunkt in der Zeile 70 berechnet:

$$\text{Mittelpunkt} = \frac{(\text{zeile\_bin} * x).\text{sum}()}{(\text{zeile\_bin}).\text{sum}()} = \frac{12811}{46} = 278,5 \quad (2)$$

### 3.1.4 linienfahren

```
1 def linienfahren(delay, run_event):
2     global cap
3     global x, minutes, seconds
4
5     ret, img = cap.read()
6     width = np.size(img, 1)
7     ideal = width/2
8     mitte = ideal
9     last_mitte = mitte
10    steer = 1
11    startzeit = time.time()
12
13    while run_event.is_set():
14        last_mitte = mitte
15        mitte = line(70)
16
17        if mitte is None:
18            if last_mitte > ideal:
19                mitte = 640
20            else:
21                mitte = 0
22        x = mitte
23
24        if mitte == ideal:
25            steer = 1
26        elif mitte < ideal:
27            steer = (mitte/ideal)
28            speed = steer*60+40
29            steer = steer*.9+.1
30        elif mitte > ideal:
31            steer = (width-mitte)/ideal
32            speed = steer*60+40
33            steer = 2-(steer*.9+.1)
34
35        lenken(steer, speed)
36
37        hours, rem = divmod(time.time()-startzeit, 3600)
38        minutes, seconds = divmod(rem, 60)
39
40        print(" " * int(mitte/10), "■", " " * int(64 - mitte/10), "x = %.1f" % mitte, ";steer = %.1f" % steer, ";speed = %.1f" % speed, ";time = {:0>2}:{:05.2f}".format(int(minutes),
41        seconds))
42        print(" " * int(ideal/10), "|")
43
44        time.sleep(delay)
```

*linienfahren* ruft die Funktion *line* auf und berechnet die Lenkung und Geschwindigkeit aus dem zurückgegebenen Wert.

Am Anfang der Funktion wird ein Testbild aufgenommen und die für die Berechnung benötigten Variablen erstellt. Danach folgt eine *while*-Schleife, die bis zum Beenden des Threadings durchläuft. Falls keine rote Linie im Bild war, oder in seltenen Fällen, wenn die Linie nicht erkannt wurde, gibt *line* *None* zurück. In diesem Fall wird in den Zeilen 17 bis 21 überprüft, ob die Linie zuletzt rechts oder links im Bild war. Der Messwert *mitte* wird dementsprechend auf das Minimum (0) oder das Maximum (640) gesetzt, um bei Verlust der Linie schnell zu reagieren. Anschließend wird der Wert für die Videoausgabe abgespeichert (Zeile 22).

In den Zeilen 24 bis 33 wird der Wert für die Lenkung und Geschwindigkeit berechnet. Unter-

schieden wird zwischen drei Fällen: Ist der Messwert in der Mitte des Bildes, wird geradeaus gelenkt, andernfalls wird die Lenkung und die Geschwindigkeit berechnet. Ist der Messwert größer als der Bildmittelpunkt, wird  $2 - \text{Lenkwert}$  berechnet. Es ergibt sich ein Lenkwert zwischen 0 und 1 für eine Linkskurve und 1 bis 2 für eine Rechtskurve. Das hat den Vorteil, dass der *lenken*-Funktion nicht zusätzlich eine Richtungsangabe übergeben werden muss. Anschließend wird die *lenken*-Funktion mit den Lenk- und Geschwindigkeitsvariablen aufgerufen.

Der Wert für die Lenkung und die Geschwindigkeit wird aus dem Messwert und dem Bildmittelpunkt berechnet:

$$\text{Lenkung} = \frac{\text{Messwert}}{\text{Bildmittelpunkt}} \cdot 90\% + 10\% \quad (3)$$

$$\text{Geschwindigkeit} = \frac{\text{Messwert}}{\text{Bildmittelpunkt}} \cdot 60\% + 40\% \quad (4)$$

Damit die Lenkung nicht zu stark ist, wird der Wert kleiner gewichtet ( $\cdot 90\%$ ) und angehoben ( $+10\%$ ). So startet der Wert nicht bei 0%, sondern bei 10%. Die Geschwindigkeit wird gleich berechnet, allerdings mit ( $\cdot 60\%$ ) gewichtet und ( $+40\%$ ) angehoben. Es wird also immer mit 40% bis 100% Geschwindigkeit gefahren.

Für die Text- und Videoausgabe wird die verstrichene Zeit berechnet. Anschließend folgt in Zeile 40/41 eine Textausgabe über die Konsole. Dabei werden  $\frac{\text{Messwert}}{10}$  Leerzeichen, dann ein Blockzeichen als Position des Messpunktes im Bild und die verbleibenden  $64 - \frac{\text{Messwert}}{10}$  Leerzeichen geschrieben. Dann folgen noch Angaben zu Messpunkt, Lenkwert, Geschwindigkeit und Zeit. In der nächsten Zeile wird der Mittelpunkt mit einem Strich markiert. Es ergibt sich eine Textausgabe, die zum Beispiel so aussieht:

```

■          x = 245.0 ;steer = 0.8 ;speed = 85.9 ;time = 00:00.00
|
■          x = 265.0 ;steer = 0.8 ;speed = 89.7 ;time = 00:00.04
|
■          x = 284.0 ;steer = 0.9 ;speed = 93.2 ;time = 00:00.08
|
■          x = 292.0 ;steer = 0.9 ;speed = 94.8 ;time = 00:00.12
|
■          x = 304.0 ;steer = 1.0 ;speed = 97.0 ;time = 00:00.16
|
■          x = 312.0 ;steer = 1.0 ;speed = 98.5 ;time = 00:00.20
|
■          x = 325.0 ;steer = 1.0 ;speed = 99.1 ;time = 00:00.24
|

```

### 3.1.5 lenken

```

def lenken(steer, speed):
    if steer > 2:
        steer = 2
    elif steer < 0:
        steer = 0
    if speed > 100:
        speed = 100
    elif speed < 0:
        speed = 0

```

```

10    speedHead = (100 - speed)
12
13    if speedHead > speed:
14        speedHead = speed
15
16    if steer == 1:
17        pr.ChangeDutyCycle(speed) # rechte Motoren
18        pl.ChangeDutyCycle(speed) # linke Motoren
19    elif steer < 1:
20        pr.ChangeDutyCycle(((1 - steer) * speedHead + speed))
21        pl.ChangeDutyCycle(steer * speed)
22    elif steer > 1:
23        steer = 2 - steer # steer auf 0-1 normieren
24        pr.ChangeDutyCycle((steer * speed))
25        pl.ChangeDutyCycle(((1 - steer) * speedHead + speed))
26
27    return

```

Die *lenken*-Funktion steuert das Tastverhältnis der Motoren. Übergeben werden zwei Parameter für die Lenkgewichtung und die Geschwindigkeit. Am Anfang der Funktion werden einige Abfragen zur Fehlerverhütung vorgenommen, damit das Tastverhältnis nicht unter 0% oder über 100% liegt, dies würde sonst zu einem Absturz führen.

Damit das Auto in den Kurven nicht ungewollt langsamer fährt, wird ausgerechnet, wie viel schneller sich die äußeren Motoren drehen können.

$$\text{Kopfraum} = 100 - \text{Geschwindigkeit} \quad (5)$$

Ist der Kopfraum größer als der Geschwindigkeitswert, wird der Kopfraum gleich der Geschwindigkeit gesetzt.

Die Tastverhältnisse der Motoren werden gleich dem Geschwindigkeitswert gesetzt, wenn der Lenkwert  $steer = 1$  ist. Sonst wird unterschieden, ob der Lenkwert größer oder kleiner als 1 ist, um in die entsprechende Richtung zu lenken.

$$\text{Innere Motoren: } \text{Tastverhältnis} = \text{Lenkwert} \cdot \text{Geschwindigkeit} \quad (6)$$

$$\text{Äußere Motoren: } \text{Tastverhältnis} = (1 - \text{Lenkwert}) \cdot \text{Kopfraum} + \text{Geschwindigkeit} \quad (7)$$

Die Kombination aus dem Lenkwert, Geschwindigkeitswert und Kopfraum ermöglicht ein stufenloses Kurvenfahren, ohne dass an Geschwindigkeit verloren wird. Dabei ist es egal, worauf die Werte basieren, es wird immer gleich sanft gelenkt.

Lenken	Geschwindigkeit	$T_{\text{innen}} [\%]$	$T_{\text{außen}} [\%]$
0,2	20	2	36
	60	12	92
	80	16	96
0,8	20	16	24
	60	48	68
	80	64	84

Tabelle 1: Beispielwerte für die Motorensteuerung

### 3.1.6 checkgreen

```

1 def checkgreen():
2     global img, ret
3
4     # Take each frame
5     ret, img = cap.read()
6
7     # Convert BGR to HSV
8     hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
9
10    lower_green = np.array([45, 200, 200])
11    upper_green = np.array([80, 255, 255])
12
13    mask = cv2.inRange(hsv, lower_green, upper_green)
14    no_green = cv2.countNonZero(mask)
15
16    return no_green

```

Die *checkgreen*-Funktion ist eine einfache Methode zur Erkennung einer grünen Ampel. Sie analysiert ein ganzes Bild der Webcam auf ihren Grünanteil und gibt die Anzahl der Pixel in einem gewählten HSV-Bereich an. Dabei ist zu beachten, dass die Funktion nur auf die Farbe der Ampel reagiert und nicht einen Fehlstart des Autos durch andere Grüntöne im Bild auslöst. Um dies zu vermeiden, wird eine Umwandlung des Farbraums von BGR – was die Kamera nativ ausgibt – zu HSV vorgenommen (Zeile 8), um einen präzisen Farbbereich mit zwei Threshold-Werten für die untere und obere Grenze mithilfe von OpenCV festzulegen. Danach wird eine Maske mit der Funktion *cv2.inRange* erstellt, in der alle Pixel innerhalb dieses Farbbereichs liegen (Zeile 13). Um festzustellen, dass die Ampel im Bild grün ist, werden alle weißen Pixel der Maske mit *cv2.countNonZero* gezählt (Zeile 14). Erst bei einer genügend großen Anzahl an grünen Pixeln, hier mehr als 500, beginnt das Auto die Fahrt.

Der HSV-Raum eignet sich im Gegensatz zu BGR gut für die Selektion eines Farbbereiches, weil er sich aus drei Komponenten Hue (Farbton), Saturation (Farbsättigung) und Value (Hellwert) zusammensetzt. Als Testbild wurde mit der Webcam ein Bild des LED-Streifens aufgenommen, wobei zusätzlich noch ein ähnlicher Farbton in Form eines grünen Klebebandes dazukommt, um die Grenzen besser festzulegen. Für die Umrechnung von BGR- nach HSV-Werten gelten je nach Formelsatz andere Intervalle. Die Tabelle 2 zeigt einen Vergleich zwischen gängigen Bildbearbeitungsprogrammen und OpenCV:

Komponente	Bildbearbeitungsprogramm	OpenCV
Hue (Farbton)	$H \in [0^\circ, 360^\circ]$	$H \in [0, 179]$
Saturation (Farbsättigung)	$S \in [0, 1]$	$S \in [0, 255]$
Value (Hellwert)	$V \in [0, 1]$	$V \in [0, 255]$

Tabelle 2: Intervalle des HSV-Farbraums

Eine schnelle Umrechnung von BGR (Werte zwischen 0 und 255) nach HSV lässt sich durch folgende Befehle umsetzen:

```

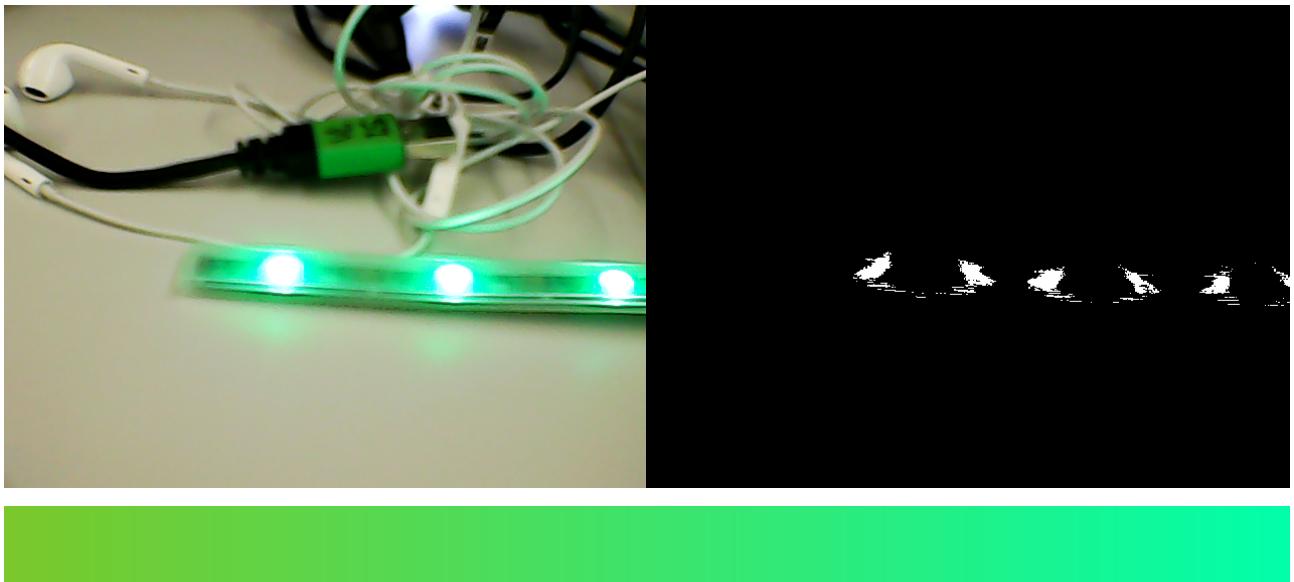
import cv2
2 import numpy as np
bgr_color = np.uint8([[B,G,R]])

```

```

4 hsv_color = cv2.cvtColor(bgr_color, cv2.COLOR_BGR2HSV)
print(hsv_color)

```



Untere Grenze:

$$R = 122, G = 200, B = 43$$

$$H = 45, S = 200, V = 200$$

Obere Grenze:

$$R = 0, G = 255, B = 170$$

$$H = 80, S = 255, V = 255$$

Abbildung 6: Erkennung der grünen Ampel

Nach der Funktion werden 2616 grüne Pixel für den gewählten Bereich gezählt, die nur das Licht von den LEDs einschließen und nicht das grüne Klebeband. Es war auch darauf zu achten, dass unser Auto nur bei einer grünen und nicht auch bei einer blauen Ampel losfährt, da die LEDs auf dem Kamerabild teilweise ins Türkis und Hellgrün hineinragen.

### 3.1.7 checkblue

```

1 def checkblue(delay, run_event):
2     global img, ret
3     time.sleep(1)
4
5     while run_event.is_set():
6         # Convert BGR to HSV
7         hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
8
9         lower_blue = np.array([90, 100, 255])
10        upper_blue = np.array([100, 255, 255])
11
12        mask = cv2.inRange(hsv, lower_blue, upper_blue)
13        no_blue = cv2.countNonZero(mask)
14
15        if no_blue > 500:
16            print("Blaue Ampel, warte 1,5 sekunden...")
17            time.sleep(1.5)
18            bremsen()
19            print("STOP!!! Rennen fertig")
20            run_event.clear()
21
22        time.sleep(delay)

```

*checkblue* funktioniert ähnlich wie *checkgreen*, allerdings läuft die Funktion in Dauerschleife und stoppt das Auto nach 1,5 Sekunden, nachdem die blaue Ampel erkannt wurde. Dies hat den einfachen Grund, dass das Auto nicht sofort bei Erkennung der blauen Ampel vor der Ziellinie stehen bleibt, sondern diese sicher überschreiten kann. Die Funktion nimmt dabei kein eigenes Bild auf, sondern verwendet das Bild, das von der *linienfahren*-Funktion aufgenommen wurde. Beim Starten des Threads wartet die Funktion eine Sekunde, damit die grüne Ampel nicht fälschlicherweise als Stoppsignal erkannt wird.

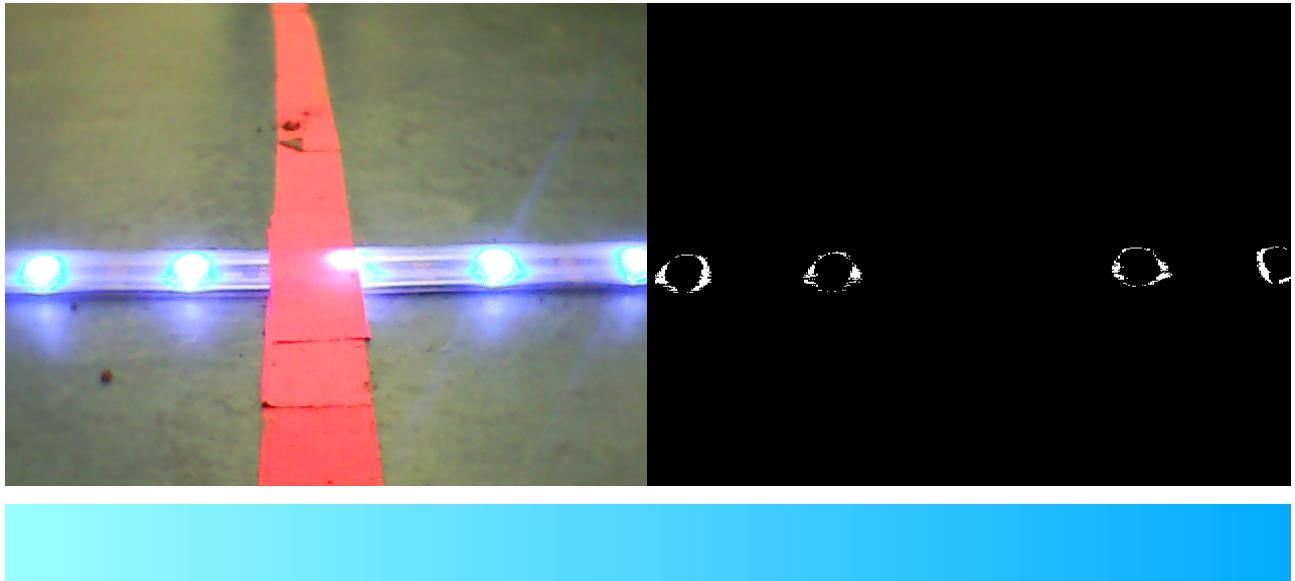


Abbildung 7: Erkennung der blauen Ampel

### 3.1.8 makevideo

```

1 def makevideo(delay, run_event):
2     global img, ret, out
3     global x, minutes, seconds
4
5     minutes = 0
6     seconds = 0
7     font = cv2.FONT_HERSHEY_SIMPLEX
8     x = 320
9     capture_video = True
10
11    while run_event.is_set():
12        if ret==True and capture_video == True:
13            cv2.line(img,(int(x)-1,70),(int(x)+1,70),(255,0,0),5)
14            cv2.putText(img,'{:0>2}':{:05.2f}'.format(int(minutes),seconds),(10,470), font,
15            2,(255,255,255),2,cv2.LINE_AA)
16            cv2.putText(img,"% .0f" % x,(int(x)-30,100), font, 1,(255,255,255),2,cv2.LINE_AA)
17            out.write(img)
18            time.sleep(delay)

```

Die Aufgabe der *makevideo*-Funktion ist es, die von der Kamera aufgenommenen Bilder als ein Video zu exportieren. Da das Video nur zur Fehleranalyse und Veranschaulichung dient, wird

es vernachlässigt, ob das Video in Echtzeit läuft. Dafür wird die aktuelle Fahrzeit – ab dem Erkennen der grünen Ampel – unten links im Bild eingeblendet. Außerdem wird der zu dem Bild gehörende Messpunkt und dessen Wert eingezeichnet.

Das Video in Zusammenhang mit den eingeblendeten Werten ermöglicht eine deutlich bessere Fehleranalyse als eine Textausgabe über die Konsole (Abb. 8).



Abbildung 8: Kamerabild mit Zeitanzeige und Messpunkt, abgelenkt von einem roten Schuh

### 3.2 setup.py

```
1 import RPi.GPIO as GPIO # GPIO-Bibliothek importieren
2
3 GPIO.setmode(GPIO.BCM) # Verwende BCM-Pinnummern
4
5 # GPIO für Motoren
6 # Motor A
7 ENA = 10 # Enable Motor A
8 IN1 = 9 # In 1
9 IN2 = 11 # In 2
10 # Motor B
11 ENB = 22 # Enable Motor B
12 IN3 = 17 # In 3
13 IN4 = 27 # In 4
14
15 # GPIOs als Ausgang setzen
16 GPIO.setup(ENA, GPIO.OUT)
17 GPIO.setup(IN1, GPIO.OUT)
18 GPIO.setup(IN2, GPIO.OUT)
19 GPIO.setup(ENB, GPIO.OUT)
20 GPIO.setup(IN3, GPIO.OUT)
21 GPIO.setup(IN4, GPIO.OUT)
22
23 # PWM für Motor A und B
24 pr = GPIO.PWM(ENA, 73) # Motor A, Frequenz = 73 Hz
25 pl = GPIO.PWM(ENB, 73) # Motor B, Frequenz = 73 Hz
26
27 GPIO.output(IN1, 0) # Bremsen
28 GPIO.output(IN2, 0) # Bremsen
29 GPIO.output(IN3, 0) # Bremsen
30 GPIO.output(IN4, 0) # Bremsen
31
32 print("GPIO-Setup erfolgreich")
```

In *setup.py* werden die GPIOs definiert, das Skript wird am Anfang von *main.py* aufgerufen. Verwendet wird der BCM-Modus. Für die PWM wurde eine Frequenz von 73 Hz gewählt, was mit den Ultraschallsensoren zusammenhängt.

Dadurch, dass das GPIO-Setup ausgelagert war, konnten alle Python-Skripte während der ganzen Entwicklungsphase problemlos verwendet werden. Wenn die GPIOs sich ändern, muss dies nur in dieser einen Datei eingetragen werden.

### 3.3 aufraeumen.py

```
import RPi.GPIO as GPIO # GPIO-Bibliothek importieren
2 import time           # Modul time
from setup import *
4
5 def aufraeumen():
6     # Erst bremsen dann cleanup
7     GPIO.output(IN1, 0)  # Bremsen
8     GPIO.output(IN2, 0)  # Bremsen
9     GPIO.output(IN3, 0)  # Bremsen
10    GPIO.output(IN4, 0)  # Bremsen
11    time.sleep(.1)
12    GPIO.cleanup()       # Aufräumen
13    print("GPIOs aufgeräumt")
14
15 def bremsen():
16     GPIO.output(IN1, 0)  # Bremsen
17     GPIO.output(IN2, 0)  # Bremsen
18     GPIO.output(IN3, 0)  # Bremsen
19     GPIO.output(IN4, 0)  # Bremsen
20
21 def losfahren():
22     GPIO.output(IN1, 1)      # Motor A Rechtslauf
23     GPIO.output(IN2, 0)      # Motor A Rechtslauf
24     GPIO.output(IN3, 1)      # Motor B Rechtslauf
25     GPIO.output(IN4, 0)      # Motor B Rechtslauf
```

In *aufraeumen.py* sind drei Funktionen ausgelagert. *losfahren* setzt die vier GPIOs der Motoren auf die entsprechenden Werte zum Vorwärtsfahren, *bremsen* setzt die GPIOs zum Stoppen des Autos auf null und *aufraeumen* stoppt das Auto und bereinigt die GPIO-Einstellungen.

## 4 3D-Druck-Halterungen

### 4.1 Kamerahalterung

Um die Kamera besser in das Auto integrieren zu können und um ein ständiges Nachjustieren zu umgehen – da die Kamera bei Anbringung vor dem Auto immer verstellt wird, wenn ein Unfall gebaut wird – wurde eine spezielle Halterung entwickelt. Diese erlaubt die Positionierung der Kamera oberhalb unseres Autos und bietet somit mehr Sicherheit vor Hindernissen.

Erstellt wurde die Halterung mit dem Online-3D-Design-Tool Tinkercad. Da die Kamera auch mit dieser Halterung immer noch von Hindernissen erreicht werden konnte (zum Beispiel hervorstehenden Schrankwänden), wurde die Verbindung von Kamera und Halterung zusätzlich mit Sekundenkleber verstärkt.

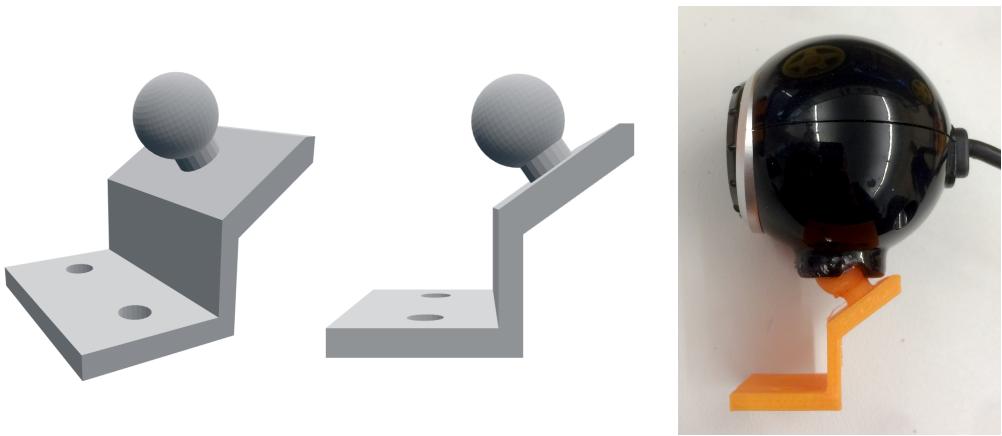


Abbildung 9: Kamerahalterung 3D-Modell und fertiger 3D-Druck

## 4.2 Ultraschallsensorhalterung

Für den Ultraschallsensor HC-SR04 wurde eine Halterung entworfen, die es ermöglicht, den Sensor sowohl aufrecht als auch kopfüber an dem Auto zu befestigen. Außerdem sollte der Sensor, wie alle Teile des Autos, einfach abnehmbar sein, falls Änderungen vorgenommen werden müssen. Entworfen wurde das Modell in Blender 2.79 mit zwei Löchern für Schrauben, Platz für weitere Bohrungen und einen Schlitz für die Steckverbindung.

Der 3D-Druck ging problemlos und die Halterung erfüllte die Anforderungen, es stellte sich aber heraus, dass die Maße der Sensoren teilweise zu stark voneinander abweichen, deswegen wurde das Modell noch einmal mit mehr Spielraum überarbeitet.

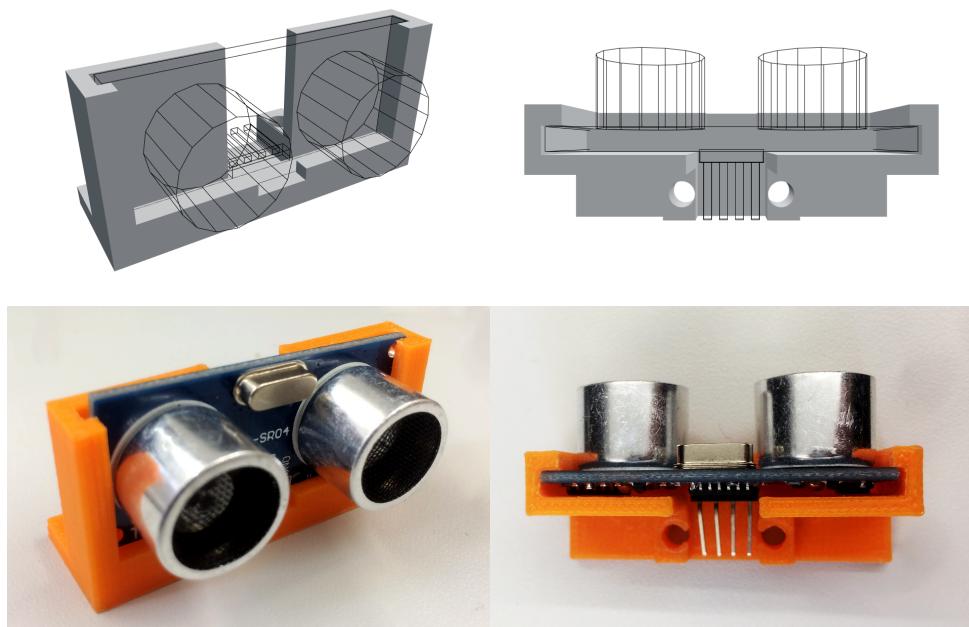


Abbildung 10: Ultraschallsensorhalterung 3D-Modell und fertiger 3D-Druck

## 5 Probleme

Der günstige Anschaffungspreis des Autobausatzes führte zu einem schnellen Verschleiß der linken Motoren, die wir vor der Prüfung beide austauschen mussten, weil sie durch die fast ausschließlich vorkommenden Rechtskurven stärker beansprucht werden, als die rechten Motoren. Wie in allen anderen Gruppen machte der Ultraschallsensor nach einiger Zeit Probleme, was allerdings nicht nur auf die minderwertige Qualität zurückzuführen ist. Meistens waren die gemessenen Abstände zur Wand bei Stillstand des Autos bis auf wenige Abweichungen im Toleranzbereich. Erst bei der Fahrt war die konstante Einhaltung des Abstandes zunächst in vielen Fällen nicht zuverlässig, da die Messwerte aus unbekannten Gründen völlig in die Höhe gestiegen sind, obwohl die Distanz nahezu identisch geblieben ist. Dies wurde aber zum Teil durch die Pulsweitenmodulation der Motoren beeinträchtigt, dessen Frequenzen der Rechtecksignale mit dem Senden des Triggers und Empfangen des Echos am Sensor zu Kollisionen führte. Nach einer Anpassung der PWM-Frequenz von 70 Hz auf 73 Hz funktionierte die Wandfahrt schon deutlich besser und wurde um einiges zuverlässiger. Für zukünftige Projekte wäre es ratsam, auf hochwertigere Ultraschallsensoren umzusteigen und einen Aufpreis in Kauf zu nehmen, aber dafür ein stabileres Fahrverhalten zu gewährleisten.

## 6 Ergebnis

Am Tag der Präsentation konnten wir unsere persönliche Bestzeit vom Vortag nochmals um 15 Sekunden verbessern. Pro Runde benötigte unser Auto durchschnittlich 52 Sekunden, also insgesamt 1:44 Minuten für zwei Runden. Die gesamte Strecke wurde völlig autonom bewältigt, denn unser Auto fuhr problemlos bei der grünen Ampel los und beendete seine Fahrt kurz hinter der Ziellinie nach zwei Runden durch die blaue Ampel. Während der Fahrt befand sich das Auto zu jeder Zeit innerhalb der schwarzen Begrenzungslinien und musste an keiner Stelle neu auf die Strecke gesetzt werden.

Racing Infos here										
	Teamname	Teammitglieder	Gesamtzeit	Rundenzeit	Checkpoint1	Checkpoint2	Checkpoint3	Checkpoint4	Info	Datum
2	Team 2	Joern; Darius; Mirco	104.0	37.0	10.0	14.0	13.0	0.0	Das ist doch egal! Das ist eh nur vorlaeufig...	30.01.2018 13:49:12
5	Optimus Pi	Amdi; Basti; Patrick	152.0	67.0	13.0	18.0	17.0	19.0	Erschaffen in den Flammen Mordors; gekommen u...	30.01.2018 14:11:40
3	X	Goekhan; Alex; Julia	159.0	77.0	14.0	22.0	20.0	21.0	empty	30.01.2018 13:57:49
4	Pink Danger	Rafia; Tobias; Luisa; Michael	255.0	108.0	23.0	28.0	10.0	29.0	The Underdogs will rise!	30.01.2018 14:04:49
1	Racing Team 1	Jannika; Hauke; Simon	283.0	120.0	49.0	34.0	29.0	8.0	empty	30.01.2018 13:43:14
6	The Highspeedcoder	Flo; Michel; Fabian	363.0	172.0	42.0	39.0	37.0	53.0		30.01.2018 14:21:19

Abbildung 11: Gesamtzeiten aller ITS-Gruppen

Der Abstand von 48 Sekunden zum zweitschnellsten Team ist dadurch zu erklären, dass die Motoren an keiner Stelle mit weniger als 40 % der Höchstgeschwindigkeit betrieben wurden und

das Auto selbst in den engsten Kurven immer noch sehr zügig der Linie gefolgt ist, ohne sie dauerhaft zu verlieren und vom Kurs abzukommen. Das Verhältnis zwischen sicherem Fahren und hoher Geschwindigkeit wurde auf dem Raspberry Pi durch unseren Code ideal ausgenutzt.

## 7 Geplante Weiterentwicklungen

Es sind diverse Weiterentwicklungen an dem Auto geplant, die bisher nicht verwirklicht wurden. Ein Plan ist es, die Videoausgabe deutlich auszubauen. So können mehr Messwerte angezeigt werden, wie zum Beispiel die Lenk- und Geschwindigkeitswerte oder die Drehzahl der Motoren und damit die zurückgelegte Strecke. Optimal wäre die Ausgabe eines Livestreams, was allerdings nur möglich ist, wenn dadurch der Fahralgorithmus nicht verlangsamt wird. Zudem war eine Anzeige der aktuellen Messwerte über LEDs in Entwicklung.

Der Aufbau der Bauteile auf dem Auto soll noch kompakter werden, indem platzsparender gelötet wird. Dadurch wären alle Platinen auf der unteren Platte platziert, sodass nur noch der Raspberry Pi, der Akku und die Kamera zu sehen sind.

Es ist auch eine andere Herangehensweise an die Programmierung mithilfe von neuronalen Netzen denkbar. So würde das Auto mit der Zeit selbstständig lernen, wie es die Rennstrecke zu absolvieren hat und wie Schwierigkeiten, zum Beispiel enge Kurven, am besten gehandhabt werden können. Danach wäre es auch möglich, wie im Straßenverkehr vorausschauend zu fahren und mögliche Gefahren und Hindernisse frühzeitig zu entdecken und dagegen vorzugehen. Nichtsdestotrotz werden damit Kenntnisse beim Programmieren vorausgesetzt, die wahrscheinlich nicht vollständig innerhalb eines Semesters erlernt werden können.

## **Tabellenverzeichnis**

1	Beispielwerte für die Motorensteuerung . . . . .	13
2	Intervalle des HSV-Farbraums . . . . .	14

## **Abbildungsverzeichnis**

1	Auto von oben . . . . .	1
2	Rennstrecke . . . . .	4
3	Das Auto im finalen Aufbau . . . . .	5
4	Schaltplan . . . . .	6
5	Berechnung des Mittelpunktes für die Zeile 70 . . . . .	10
6	Erkennung der grünen Ampel . . . . .	15
7	Erkennung der blauen Ampel . . . . .	16
8	Kamerabild mit Zeitanzeige und Messpunkt, abgelenkt von einem roten Schuh .	17
9	Kamerahalterung 3D-Modell und fertiger 3D-Druck . . . . .	19
10	Ultraschallsensorhalterung 3D-Modell und fertiger 3D-Druck . . . . .	19
11	Gesamtzeiten aller ITS-Gruppen . . . . .	20

## **Literaturverzeichnis**

- [1] Edeler, Torsten: *IT-Systeme*, HAW Hamburg, 13. Oktober 2017