**WRO Saison 2024 - Future Engineers**

**Team: Lego Sapiens**

**Team-Mitglieder: Richard Schütz, Malte Eikmeier, Emil Petini**

**Coach: Jenny Schütz**

**Heisenberg-Gymnasium Bruchsal**

**<Team-Foto>**

Inhalt

[Einleitung und Motorisierung 2](#_Toc165203357)

[Zielsetzung 2](#_Toc165203358)

[Fahrzeugkonzept 2](#_Toc165203359)

[Motorisierung 2](#_Toc165203360)

[Energie und Sensoren 2](#_Toc165203361)

[Verwendete Sensoren 3](#_Toc165203362)

[Einzelheiten zur Verarbeitung der LIDAR-Daten 4](#_Toc165203363)

[Einzelheiten zur Bildverarbeitung 4](#_Toc165203364)

[Trainingsdaten und Training der KI 5](#_Toc165203365)

[Hindernisse 5](#_Toc165203366)

[Programmierung 5](#_Toc165203367)

[Rennen mit KI-Steuerung 6](#_Toc165203368)

[Datenverarbeitung / Modellarchitektur des neuronalen Netzes 6](#_Toc165203369)

[Besonderheiten Parkmanöver 7](#_Toc165203375)

[Fotos 9](#_Toc165203376)

[Engineering / Design 12](#_Toc165203380)

[Entwicklungshistorie 12](#_Toc165203381)

[Maßnahmen zur Gewichtsreduktion 12](#_Toc165203385)

[Bordrechner 13](#_Toc165203386)

[Zusammenfassung, Ausblick und Diskussion 14](#_Toc165203387)

[Ergebnisse 14](#_Toc165203388)

[Einschränkungen bzw. Verbesserungsmöglichkeiten 14](#_Toc165203389)

[Anhang: Quellcode zur Bildverarbeitung 16](#_Toc165203390)

[Anhang: Informationen zu den KI-Trainingsläufen und Tests 17](#_Toc165203391)

[Literaturverzeichnis 18](#_Toc165203392)

# Einleitung und Motorisierung

## Zielsetzung

Das Team Lego Sapiens nimmt zum ersten Mal am Wettbewerb WRO Future Engineers teil. Zielsetzung ist, sich in diesem Projekt auch mit den praktischen Anwendungsmöglichkeiten künstlicher Intelligenz zu befassen.

Bei der Konzeption des Fahrzeugs flossen die Erfahrungen des Teams aus der Teilnahme am früheren FLL- und WRO-Wettbewerben ein.

## Fahrzeugkonzept

Es sollte keiner der angebotenen Fertigbausätze zur Teilnahme an der WRO verwendet werden. Da das Fahrzeug von Grund auf neu konzipiert wurde, waren mehrere Fahrzeuggenerationen zur Entwicklung notwendig.

Die Entwicklung und Tests erstreckten sich über einen Zeitraum von über einem Jahr. Zur Vorbereitung auf den Wettbewerb wurden umfangreiche Probefahrten zum Training des neuronalen Netzes durchgeführt. Während der Probefahrten wurden auch die bereitzustellenden Demovideos gedreht.

## Motorisierung

Zum Antrieb wird ein bürstenloser Motor vom Typ „Castle 10th Scale SCT 1410 Sensored Motor - 3800KV“ eingesetzt (Abbildung 1: Fahrmotor und Lenkservo). Grund der Auswahl des Motors sind das hohe Drehmoment bei niedriger Geschwindigkeit und die Vermeidbarkeit des „Cogging“ (Ruckeln) durch die Nutzung der Hall-Sensorschnittstelle des ESC. Dies erlaubt sanftes Anfahren und extreme Langsamfahrt beim Parkmanöver. Zusätzlich wurde die Getriebeübersetzung des Kits durch Wahl eines alternativen Ritzels auf 3.5:1 erhöht.

Der Antrieb erfolgt über vier Räder und einen Riemenantrieb. An der Hinterachse steht ein Differential zur Verfügung. Damit sind gute Traktion und Manövrierbarkeit gegeben.

Die Lenkung erfolgt an der Vorderachse mittels eines Lenkservos. Zum Einsatz kommt ein „Miuzei 20KG Servo Motor High Torque Digital Servo Metal Waterproof 1/10 Scale Servo for R/C Car Robot, DS3218 Control Angle 270°“. Dieser Servo ist ausreichend schnell und hat eine hohe Stellkraft (20kg). Wichtig ist, dass der Lenkeinschlag groß genug ist, um enge Kurven fahren zu können.

Ein Bild, das Zylinder enthält.

Automatisch generierte BeschreibungEin Bild, das Text, Werkzeug, Batterie enthält.

Automatisch generierte Beschreibung

Abbildung 1: Fahrmotor und Lenkservo

# Energie und Sensoren

Die Energieversorgung des Fahrzeugs erfolgt mit einem „Flitezone 1300 mAh 3S“ LiPo-Akku mit einer Spannung von 11.1 V. Die Kapazität des vollgeladenen Akkus reicht aus, um damit alle Rennen zu absolvieren. Ausschlaggebend für die Wahl des Akku-Typs war neben der erforderlichen Kapazität ein möglichst niedriges Gewicht (95g), um das Maximalgewicht von 1500 g für das gesamte Fahrzeug einhalten zu können. Zur Sicherheit werden zum Rennen zwei geladene Reserve-Akkus mitgebracht.

Mit dem Akku können sowohl die Motoren als auch die Steuerungselektronik (5V) betrieben werden. Hierzu wird als Spannungswandler ein „LM2596 DC-DC Abwärtswandler“ mit LED-Anzeige genutzt

Als ESC zur Motorsteuerung kommt ein „RC Auto HobbyWing QuicRun 10BL120 120A“ zum Einsatz. Der Lüfter wurde zur Gewichtsersparnis entfernt. Die kurze Wirkbetrieb beim Rennen kann nicht zu einer Überhitzung führen. Wichtig ist die Nutzung des Hallsensor-Anschlusses um Clocking bei Langsamfahrt zu vermeiden.

Die Ansteuerung von Servomotor und ESC erfolgt mit dem „SUNFOUNDER PCA9685 16 Channel 12 Bit PWM Servo Driver“. Dieser wird an die I2C-Busschnittstelle des Raspberry Pi angeschlossen.

Ein Bild, das Text, Elektronik, Schaltung, Elektrisches Bauelement enthält.

Automatisch generierte BeschreibungEin Bild, das Text, Elektronik, Kabel enthält.

Automatisch generierte Beschreibung Ein Bild, das Schaltung, Elektronik, Elektronisches Bauteil, Elektrisches Bauelement enthält.

Automatisch generierte Beschreibung

*Abbildung 2: Spannungswandler, ESC, I2C-Schnittstelle*

## Verwendete Sensoren

Zur Erfassung der Umgebungsparameter werden diverse Sensoren eingesetzt (Abbildung 3: LIDAR-Scanner, Open MV H7+ Kamera, Sense Hat, Geschwindigkeitsmesser, Touch Button):

* 1D Lidar Scanner 10 Hz mit 30 Meter Reichweite und 3240 Datenpunkten im Vollkreis mit Ethernet-Anschluss: „SLAMTEC RPLIDAR S2E 360° LASER SCANNER (30 M)“ zur Erkennung der Fahrbahnbegrenzungen und Hindernisse. Der LIDAR-Sensor ist über Kopf verbaut, um die Erfassungshöhe um ca. 2 cm zu erniedrigen. Zusätzlich ist die Hinterachse des Fahrzeugs um ca. 3mm erhöht: So zielen die Laserstrahlen leicht abwärts und treffen immer die 10 cm hohen Begrenzungswände des Spielfelds sowie die Hindernisse. (Hinweis: Zunächst wurde mit einem einfacheren Lidar vom Typ Lidar RPLIDAR A1 ROS S1 mit seriellem Anschluss experimentiert. Leider war dieser nicht genau genug.)
* Zwei Open MV H7+ Kamerasysteme mit Fisheye-Objektiv mit integrierter Datenauswertung im Kamerasystem (programmierbar in Micro Python) zur Farberkennung der Hindernisse. Die Kameras wurden mit Blickwinkel jeweils schräg nach vorne (im 45-Grad-Winkel) verbaut, um Hindernisse bei Kurvenfahrten möglichst früh erkennen zu können.
* Gyroskop (Komponente des Raspberry PI Sense Hat) zur Erkennung der Ausrichtung des Fahrzeugs und Ermittlung des Rennendes nach drei Runden: Das Rennen ist beendet, wenn das Fahrzeug um 12 Ecken von 90 Grad gefahren ist. um die eigene Achse gedreht hat.
* Geschwindigkeits- und Streckenmessung mit Lichtschranke / Schlitzscheibe am rechten Hinterrad zur PID-basierten Geschwindigkeitssteuerung auf Basis von Geschwindigkeitsmesssensor „LM393 Geschwindigkeitsmessmodul Tacho Sensor Slot Typ IR Optokoppler mit Encoder“.
* Tastsensor für Start und Stop des Rennens: Kapazitiver Touch Sensor Taster TTP223b. (Abb. 9).
* Bluetooth-Schnittstelle zur Anbindung eines Xbox-Controllers für Trainingsfahrten zum Datensammeln für die KI

Ein Bild, das Waffeleisen, Gerät enthält.

Automatisch generierte BeschreibungEin Bild, das Elektrisches Bauelement, passives Bauelement, Elektronik, Elektronisches Bauteil enthält.

Automatisch generierte BeschreibungEin Bild, das Elektronik, Computerkomponenten, Elektronisches Bauteil, Elektrisches Bauelement enthält.

Automatisch generierte BeschreibungEin Bild, das Text, Elektronik, Elektronisches Bauteil, Elektrisches Bauelement enthält.

Automatisch generierte BeschreibungEin Bild, das Text, Elektronik, Elektrisches Bauelement, Elektronisches Bauteil enthält.

Automatisch generierte Beschreibung

Abbildung 3: LIDAR-Scanner, Open MV H7+ Kamera, Sense Hat, Geschwindigkeitsmesser, Touch Button

## Einzelheiten zur Verarbeitung der LIDAR-Daten

In ROS2 ist ein Hardware-Treiber für den verwendeten LIDAR-Scanner verfügbar. Dieser fungiert als Publisher und versendet die mit 10 Hz gescannten 360 Grad-Umgebungsdaten als Liste von jeweils 3240 Entfernungsdaten (Vollkreis). Genutzt werden im Rennmodus die vorderen 180 Grad des Sichtfelds. Der zentrale Knoten zur Datenerfassung (s2e\_lidar\_reader\_node) bzw. Renndurchführung (full\_drive\_node) empfängt die Nachrichten mit den Entfernungsmessungen und verarbeitet diese weiter:

**Quellcode-Ausschnitt Subscription:**

self.subscription\_lidar = self.create\_subscription(

LaserScan,

'/scan',

self.lidar\_callback,

qos\_profile

)

Die vom LIDAR-Scanner empfangenen Rohdaten werden zunächst aufbereitet: Messfehler (Distanzen > 2.8m) werden unterdrückt. fehlende Werte werden interpoliert. Zur stärkeren Betonung von nahen Objekten werden die Entfernungsmessungen invertiert.

**Quellcode-Ausschnitt Weiterverarbeitung Scan-Daten:**

scan = np.array(msg.ranges[self.num\_scan+self.num\_scan2:]+msg.ranges[:self.num\_scan2])

scan[scan == np.inf] = np.nan

scan[scan > self.scan\_max\_dist] = np.nan

x = np.arange(len(scan))

finite\_vals = np.isfinite(scan)

scan\_interpolated = np.interp(x, x[finite\_vals], scan[finite\_vals])

scan\_interpolated = [1/value if value != 0 else 0 for value in scan\_interpolated]

scan\_interpolated = list(scan\_interpolated)

## Einzelheiten zur Bildverarbeitung

Auf den Kameras wird lokal Python Code ausgeführt. Dieser erkennt Hindernisse auf der Fahrbahn in Form von Blobs (farbige Blöcke) und überträgt deren Koordinaten mittels serieller Kommunikation via USB an die zentrale Steuerung. Dort werden die Daten von einem ROS2-Knoten empfangen und weiterverarbeitet. Der Quellcode zur Bildbearbeitung auf den Open MV H7+-Kameras und den zugehörigen ROS2-Knoten auf dem Raspberry PI findet sich im Anhang.

## Trainingsdaten und Training der KI

Bei Trainingsfahrten werden Daten zur Fahrbahn und potenzieller Platzierung von Hindernissen gesammelt: Zehnmal pro Sekunde wird ein Datensatz der Steuerungs- und Umgebungsparameter erzeugt (ergibt sich aus LIDAR-Scan-Frequenz). Der Datensatz enthält folgende Werte:

|  |  |
| --- | --- |
| **Feld #** | **Feldinhalt** |
| 0 | Lenkeinschlag -1 … 1 – 0 ist Neutralstellung (geradeaus) |
| 1 | Gas -1 …1 (rückwärts … vorwärts) |
| 2..1621 | LIDAR-Daten – Hindernisentfernung von 0..2.8m/Inf (unendlich) |
| 1622..1941 | Kamera 1 (links) GRÜN – Pixelwert 1 an allen Positionen, an denen Hindernis gesehen wird |
| 1942..2261 | Kamera 1 (links) ROT – dito |
| 2262..2581 | Kamera 2 (rechts) GRÜN – dito |
| 2582..2901 | Kamera 2 (rechts) ROT – dito |

Die Übertragung der Trainingsdaten während der Fahrt erfolgt über NFS und WLAN. Im Rennbetrieb wird das WLAN regelkonform nicht genutzt / benötigt; der Rennstart wird über einen Touch Sensor ausgelöst.

Mit den übertragenen Daten wird ein neuronales Netzwerk trainiert. Dazu wird das Programmpaket Tensorflow verwendet. Zur Beschleunigung der Berechnungen wird die CUDA-Bibliothek in Verbindung mit einer Nvidia RTX 4070 TI-Grafikkarte auf einem PC eingesetzt.

# Hindernisse

Als Besonderheit erfolgt die Steuerung des Fahrzeugs mit künstlicher Intelligenz. Das komplette Programmpaket ist in Github abgelegt unter dem Account **rrrschuetz** und dem Paketnamen **s2e\_lidar\_reader**. Damit sind verteilte Entwicklung und Konfigurationsmanagement möglich. Auch die Entwicklungshistorie lässt sich lückenlos nachvollziehen.

## Programmierung

Die komplette Software wurde in Python erstellt und besteht aus den folgenden Elementen:

|  |  |  |
| --- | --- | --- |
| Aufgabe | Dateinamen | Erläuterung |
| Sammeln von Trainingsdaten | s2e\_lidar\_reader\_node.py, s2e\_lidar\_reader\_parking\_node.py | Datensammlung Rennen  Datensammlung Parkmanöver |
| Training der KI | calc\_model\_new.py, calc\_parking\_model.py | Training KI Rennen  Training KI Parkmanöver |
| Steuerung Rennen | full\_drive\_node.py | Zentraler ROS2-Knoten (Launch) |
| ROS2-Knoten zur Steuerung von Sensoren und Aktoren | display\_node.py, openmv\_h7\_node1.py, openmv\_h7\_node2.py,  speed\_control\_node.py, xbox\_reader\_node.py, touch\_button\_node.py | Statusanzeige  Kamera links  Kamera rechts  Motorsteuerung  Fernsteuerung für Datensammlung  Startknopf für das Rennen |
| Kamera-Steuerung | main.py  h7\_cam\_exec.py | Start  Lokale Verarbeitung der Hinderniserkennung auf Kamerachip |
| Dateninspektion | cam2\_viewer.py,  data\_viewer.py | Visualisierung der Dateien mit Trainingsdaten |

## Rennen mit KI-Steuerung

Beim Rennen werden dieselben Daten erfasst wie beim Training. In Echtzeit werden mittels Interferencing mit dem vortrainierten neuronalen Netzwerk Steuerungsdaten (Lenkung) auf dem Raspberry PI 4B-Bordrechner ermittelt. Hierdurch entstehen Fahrkurven um die Hindernisse, wie sie ein menschlicher, vorausschauender Fahrer wählen würde.

Beim Start des Rennens muss ermittelt werden, ob das Fahrzeug für ein Rennen im Uhrzeigersinn oder im Gegenuhrzeigersinn aufgestellt ist. Dies entscheiden wir durch Messungen mit dem LIDAR-Sensor: Wenn in der Startaufstellung bei Sicht nach vorn rechts mehr freie Fläche gemessen wird, handelt es sich um ein Rennen im Uhrzeigersinn, bei mehr freier Sicht nach links ist es ein Rennen im Gegenuhrzeigersinn.

## Datenverarbeitung / Modellarchitektur des neuronalen Netzes

Die Architektur ist speziell darauf ausgelegt, die Daten von LIDAR- und Farbsensoren zu verarbeiten, um Entscheidungen zur Fahrzeugnavigation im Hindernisparcours in Echtzeit zu treffen: Während die LIDAR-Daten präzise Konturen der Fahrbahnbegrenzung und der Hindernisse liefern, wird die Information der Kameras benötigt, um den Hindernissen die korrekte Farbe (rot, grün, magenta) zuzuordnen. Nur so kann die KI entscheiden, ob ein Hindernis links oder rechts passiert werden muss oder ob es sich um ein Parkplatzhindernis handelt.

### Eingabeschichten

**LIDAR Input:** Nimmt die LIDAR-Daten auf, welche die Distanzen zu nahegelegenen Objekten und Hindernissen messen. Diese Daten werden durch mehrere Conv1D Schichten verarbeitet. Diese Schichten helfen, die räumlichen Merkmale aus den LIDAR-Daten zu extrahieren, die für die Erkennung und Bewertung von Hindernissen wesentlich sind.

**Color Input:** Verarbeitet die Kameradaten. Diese Daten durchlaufen ebenfalls Conv1D Schichten. Die Verarbeitung der Farbdaten hilft dem Modell, visuelle Merkmale wie die Farbe und Textur der Hindernisse zu erkennen

### Merkmalsextraktion und Datenfusion

Nach den initialen Conv1D und MaxPooling1D Schritten werden die Features beider Sensortypen normalisiert durch Batch Normalization, um das Lernen zu beschleunigen und zu stabilisieren.

Die Merkmale aus beiden Eingabetypen werden mittels einer Concatenate Schicht zusammengeführt. Diese Fusion erlaubt dem Modell, eine umfassendere Sicht der unmittelbaren Umgebung des Fahrzeugs zu entwickeln, indem es sowohl räumliche als auch visuelle Informationen kombiniert.

### Dichte Schichten und Entscheidungsfindung

Mehrere Dense-Schichten folgen der Fusion, um die kombinierten Daten weiter zu verarbeiten. Diese Schichten dienen dazu, komplexe Muster und Beziehungen zwischen den extrahierten Features zu erkennen und zu lernen, die für die Steuerungsentscheidungen des Fahrzeugs kritisch sind.

Die Ausgabeschicht des Modells besteht aus Dense Neuronen, die direkt die Steuerungsbefehle für das Fahrzeug (wie Lenkwinkel und Geschwindigkeit) generieren, basierend auf der aktuellen Wahrnehmung der Umgebung.

### Training, Optimierung, Konversion

Das Modell wird mit einem Adam-Optimierer trainiert. Der Verlust wird als Mean Squared Error berechnet. Damit wird die Genauigkeit der Steuerungsbefehle bei der Trajektorienberechnung maximiert. EarlyStopping wird verwendet, um Overfitting zu verhindern und das Training zu überwachen.

Das Modell wird als TensorFlow Lite Modell gespeichert, um die ressourcenschonende Verwendung auf dem Raspberry Pi 4B zu gestatten.

### Quellcode zur Definition des neuronalen Netzes

def create\_cnn\_model(lidar\_input\_shape, color\_input\_shape):

# LIDAR data path

lidar\_input = Input(shape=lidar\_input\_shape)

lidar\_path = Conv1D(64, kernel\_size=5, activation='relu')(lidar\_input)

lidar\_path = MaxPooling1D(pool\_size=2)(lidar\_path)

lidar\_path = Conv1D(128, kernel\_size=5, activation='relu', kernel\_regularizer=l2(0.01))(lidar\_path)

lidar\_path = MaxPooling1D(pool\_size=2)(lidar\_path)

lidar\_path = Flatten()(lidar\_path)

color\_input = Input(shape=color\_input\_shape)

color\_path = Dense(64, activation='relu')(color\_input)

color\_path = Dropout(0.3)(color\_path) # Use dropout

color\_path = Dense(128, activation='relu', kernel\_regularizer=l2(0.01))(color\_path) # Regularization

color\_path = Flatten()(color\_path)

# Concatenation

#concatenated = Concatenate()([lidar\_path, color\_path])

concatenated = WeightedConcatenate(weight\_lidar=0.2, weight\_color=0.8)([lidar\_path, color\_path])

# Further processing

combined = Dense(64, activation='relu')(concatenated)

combined = Dense(64, activation='relu')(combined)

combined = Dense(64, activation='relu')(combined)

combined = Dense(64, activation='relu')(combined)

combined = Dense(32, activation='relu')(combined)

output = Dense(2)(combined)

model = Model(inputs=[lidar\_input, color\_input], outputs=output)

model.compile(optimizer='adam', loss='mean\_squared\_error', metrics=['accuracy'])

return model

## Besonderheiten Parkmanöver

Für das Parkmanöver ist ein separat trainiertes KI-Modell erforderlich: Es wird mit beiden Kameras gleichzeitig gearbeitet. Die Gewichtung des LIDAR-Kanals beträgt 90%, die des Farbkanals 10%. (Beim Hindernisrennen ist es umgekehrt.) erfolgt nicht vollständig mittels KI. Dies ist erforderlich, um die beiden magentafarbenen Parkplatzbeschränkungen im Entscheidungsprozess zu betonen.

Nach Beenden der drei Umläufe des Hindernisrennens schalten wir auf das Park-KI-Modell um und fahren in die Parklücke weiter. Der genaue Zeitpunkt ist nach Umfahren der Kurve nach dem Start-/Zielabschnitt. Dieser Zeitpunkt wir mit dem Gyroskop ermittelt. Zum Ausgleich der geringen Messgenauigkeit des Gyroskops werden die passierten Ecken (90 Grad Richtungsänderung) gezählt und zusätzlich mit dem LIDAR-Sensor der Abstand zur Wand nach Beenden der Kurve überprüft. Damit kann das Risiko einer zu späten Umschaltung auf die Parksteuerung wirksam gesenkt werden.

Während des Rennens zählen wir die Sichtung magentafarbener Hindernisse, um zwischen einer Fahrt mit oder ohne Parkmanöver entscheiden zu können: Am Ende des Hindernisrennens mit Parkmanöver steht dieser Zähler je nach Lichtverhältnissen ungefähr auf dem Wert 200, ansonsten auf < 10. Ab 50 gehen wir von einem Hindernisrennen aus. Damit senken wir das Risiko von Fehlentscheidungen bei Sichtung magentafarbener Objekte am Spielfeldrand.

Aufgrund des Fehlens rückwärtiger Sensoren und der niedrigen Präzision unseres Antriebs bei sehr kurzen Fahrstrecken parken wir nur vorwärts und teilweise ein (Regel 7.13).

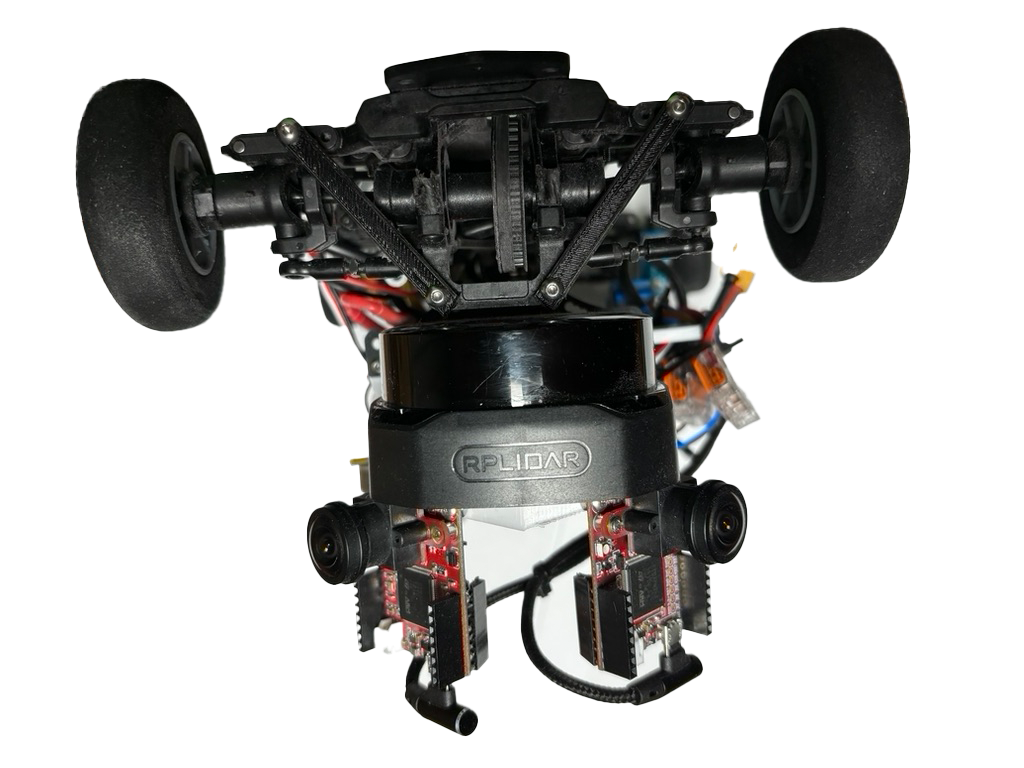
Beim Einfahren in die Parklücke wird mit den Daten des LIDAR-Sensors der Abstand nach vorn errechnet und rechtzeitig vor Berührung eins Parkplatzhindernisses bzw. der Wand gestoppt. Damit ist das Hindernisrennen mit Parkmanöver beendet.

# Fotos

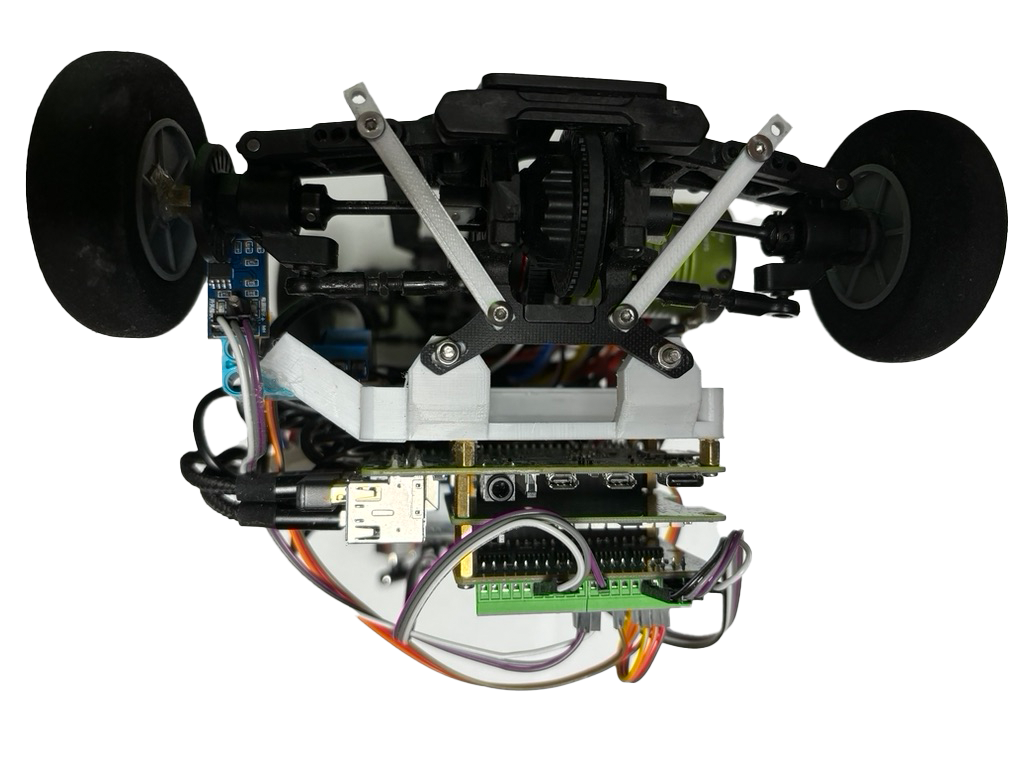
Auf den nachfolgenden Abbildungen sind die wesentlichen Komponenten des Fahrzeugs ersichtlich. Hier die Beschreibungen, die sich anhand der Nummern in den gelben Markierungen zuordnen lassen.

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

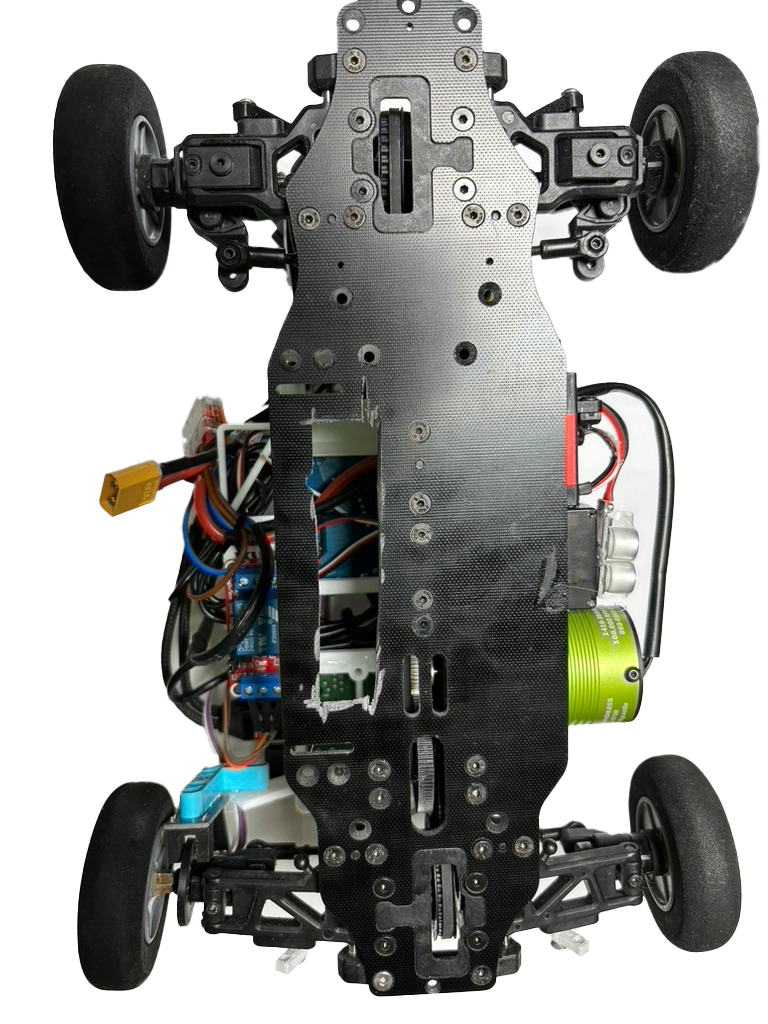
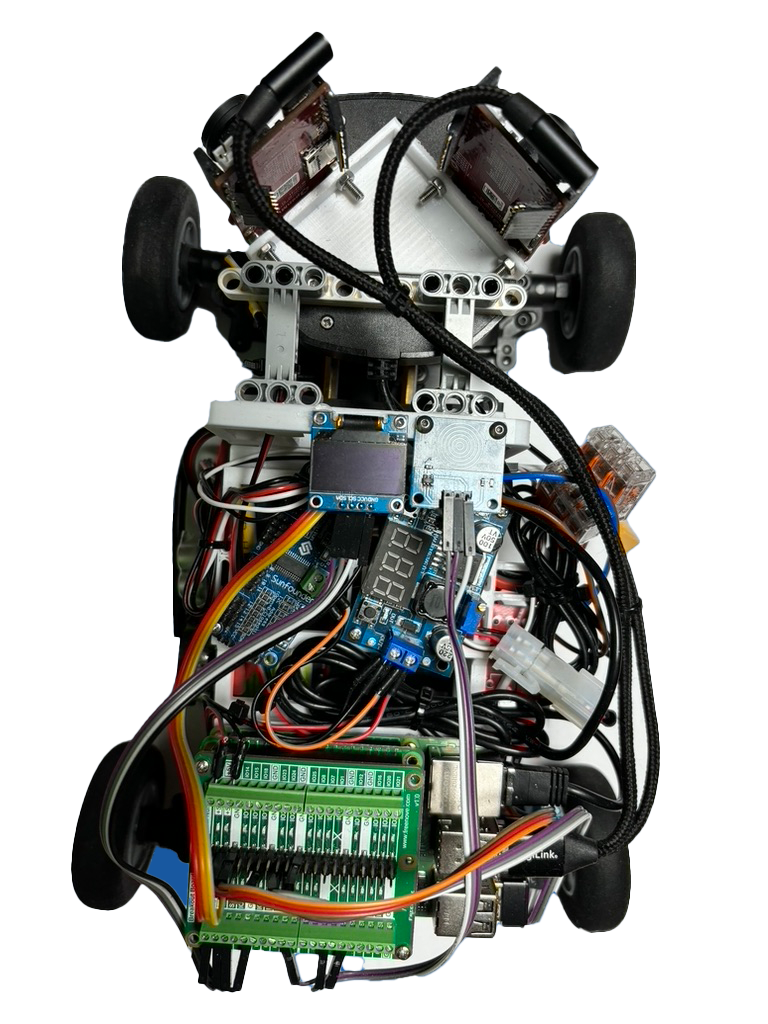
Automatisch generierte Beschreibung



## Ansicht von vorn/hinten

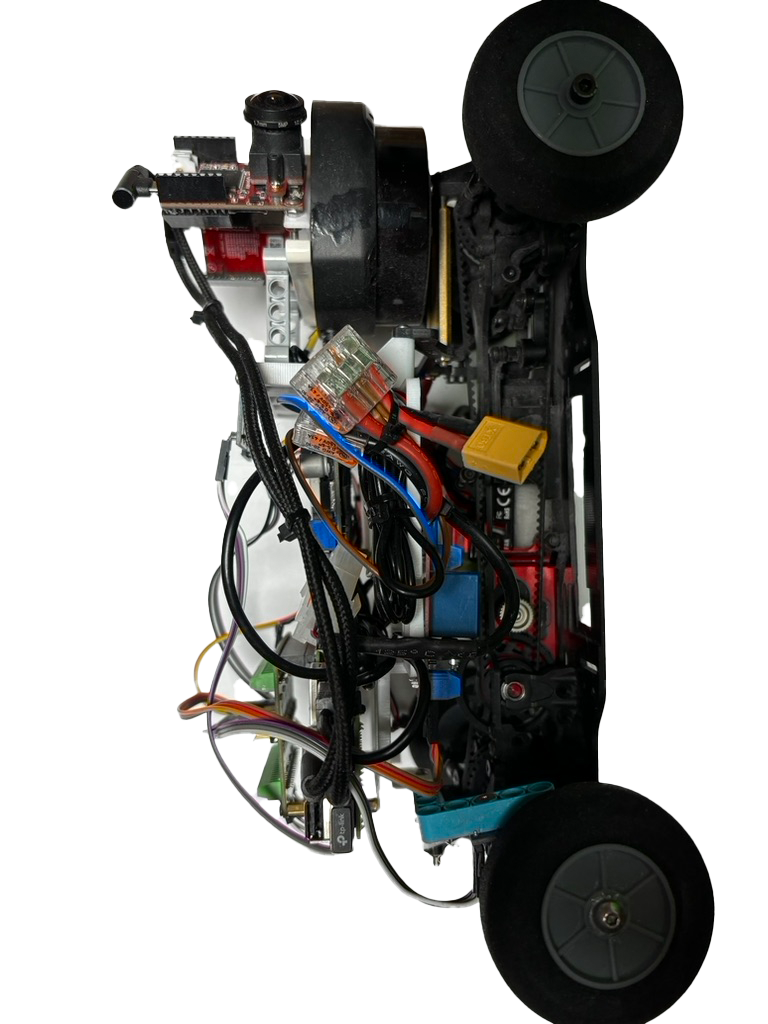
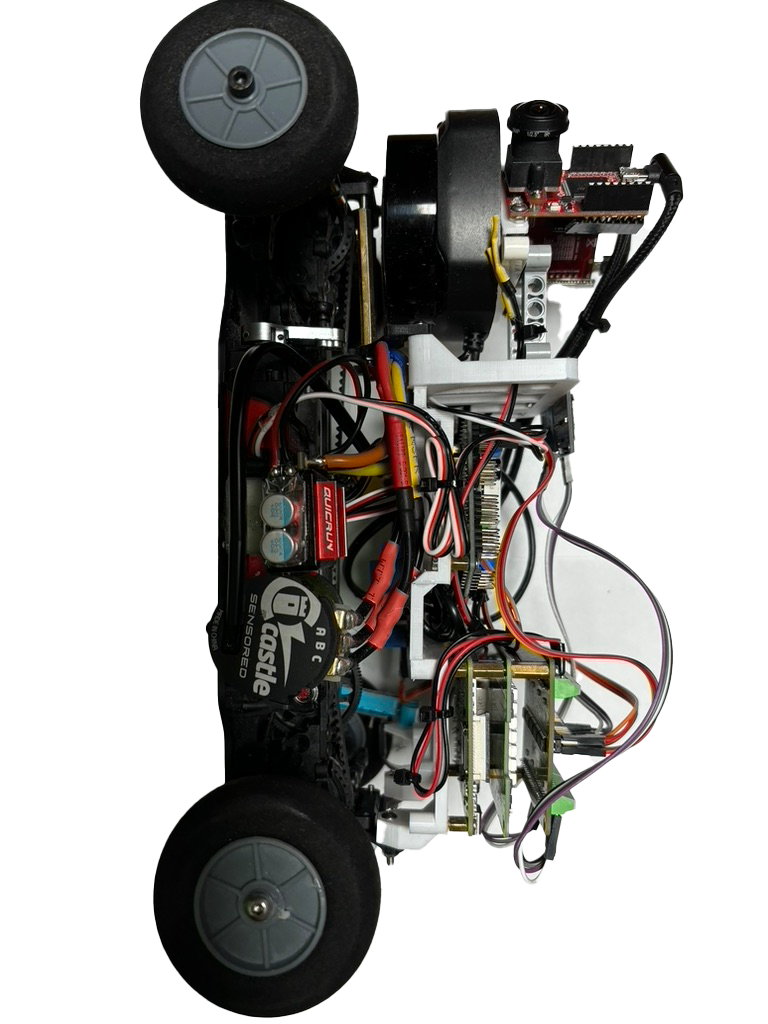


## Ansicht von oben/unten



## 

## Ansicht von links/rechts



# Engineering / Design

## Entwicklungshistorie

Es wurde keines der empfohlenen Standard-Chassis eingesetzt, um das Konzept der Steuerung mit LIDAR und KI umsetzen zu können. Das Fahrzeug ist eine Eigenentwicklung unter Verwendung von Standardbauteilen. Insgesamt wurden drei Evolutionsstufen des Fahrzeugs mit unterschiedlichen Chassis-Komponenten durchlaufen:

### Erster Prototyp zum Machbarkeitsnachweis der KI-Steuerung

Basis: Freenove 4WD Smart Car Kit for Raspberry Pi 4 B

Ein Bild, das Autoteile, Reifen, Rad, Spielzeug enthält.

Automatisch generierte Beschreibung

### Zweiter Prototyp mit regelkonformer Fahrzeugsteuerung aber Übermaßen und Übergewicht

Basis: Reely TC-04 Onroad-Chassis 1:10 RC (4WD) ARR (überwiegend aus Metallteilen)

Ein Bild, das Maschine, Autoteile, Rad enthält.

Automatisch generierte Beschreibung

### Rennfahrzeug in Leichtbauweise mit Einhaltung aller Regeln

Basis: Abgespeckter Bausatz XPRESS 90005 - EXECUTE XM1S - 1:10 4WD M-Chassis (überwiegend aus Kunststoffteilen). Niedrige Bauweise, um die Montage des LIDAR-Sensors in einer Weise zu ermöglichen, dass die Fahrbahnbegrenzungen und Hindernisse auf einer Höhe < 10 cm erfasst und erkannt werden.

Ein Bild, das Spielzeug, Maßstabsmodell enthält.

Automatisch generierte Beschreibung

## Maßnahmen zur Gewichtsreduktion

* Weglassen nicht unbedingt notwendiger Teile: Karosserieträger, Stoßdämpfer, sämtliche nicht für den Antrieb erforderlichen Teile.
* Einfügen von Aussparungen im Chassis (mit Dremel): Chassis ist im Rennbetrieb aufgrund des geringen Tempos und der ebenen Fahrbahn keinen starken Kräften ausgesetzt.
* Um Elektronik und Sensoren auf dem Chassis montieren zu können, wurde eine Tragkonstruktion mittels 3d-Druck mit minimalem Infill erstellt. Das Design ist aus Abbildung 5: Elektronikträger (3D-Druck) ersichtlich.
* Es wurden schmale Räder aus dem Modellflugzeugbau in Leichtbauweise mit Moosgummi-Bereifung ausgewählt. Um den notwendigen Grip und Lenkpräzision zu erreichen, wird ein Chassis mit Allrad-Steuerung und Riemenantrieb verwendet. Nur an der Hinterachse ist ein Differential verbaut. Die Vorderachse wird über ein Lenkservo angelenkt.
* Verkürzung der schweren Kupferkabel zwischen Akku, ESC und Motor.
* Kürzestmögliche Verbindungskabel zwischen den Komponenten (z.B. USB-Kabel für die Kameras)
* Leichter LiPo-Akku mit gerade so viel Kapazität wie für die Stromversorgung während der Rennen erforderlich.
* Speicherung von Betriebssystem, Programm und Daten auf einer microSD-Karte statt des zuvor genutzten SSD.2-Drives.

**Ein Bild, das Mond, Schwarzweiß, monochrom enthält.

Automatisch generierte Beschreibung**

Abbildung 4: Elektronikträger (3D-Druck)

## Bordrechner

Die Steuerung erfolgt via Raspberry PI 4B unter Ubuntu Linux 22.03 und ROS2 (Humble). Bei ROS2 handelt es sich um eine modulare Open Source Robotik-Middleware zur asynchronen Integration von Aktoren und Sensoren und dynamischer Steuerung. Die einzelnen Komponenten werden in Knoten abgebildet, die mittels Nachrichtenversand in Echtzeit miteinander kommunizieren.

Hinweis: Mittlerweile ist der ca. doppelt so schnelle Raspberry Pi 5 lieferbar. Leider lag zum Zeitpunkt der Programmentwicklung für das dafür notwendige Ubuntu Linux 23.04 noch keine Anpassung der ROS2-Bibliotheken vor. Damit konnte die schnellere Hardware (noch) nicht eingesetzt werden.

Zur Erleichterung des Anschlusses der Sensoren an den Raspberry Pi wird das „FREENOVE Breakout Board for Raspberry Pi 5 / 4B / 3B+ / 3B / 3A+ / 2B / 1B+ / 1A+ / Zero W/Zero, Terminal Block HAT, GPIO Status LED“ genutzt (Abbildung 4: Breakout Board).

Ein Bild, das Elektrisches Bauelement, Elektronisches Bauteil, Elektronik, passives Bauelement enthält.

Automatisch generierte Beschreibung

Abbildung 5: Breakout Board

Zur Echtzeitfähigkeit der Ermittlung der Steuerungsinformationen wurden folgende Maßnahmen ergriffen:

* Die Bildauswertung erfolgt zur Senkung des Rechenaufwands auf dem Raspberry Pi direkt auf den Open MV H7+-Kameras, s.u.
* Während des Rennens werden nur die Daten der kurveninneren Kamera ausgewertet. Es wird eine horizontale Bildauflösung der Kameras von nur 320 Pixeln verwendet.
* Das KI-Interferencing erfolgt mit Tensorflow Lite. Gegenüber Tensorflow kann der Rechenaufwand des Interferencing deutlich gesenkt werden.
* Es wird nur der vorwärtsgerichtete Erfassungsbereich des Scanners ausgewertet. Dadurch kann die LIDAR-Datenmenge halbiert werden (1620 statt 3240 Datenpunkte).

Die von der KI errechnete Fahrmotor-Einstellung wird bei den Rennfahrten nicht genutzt: Die hohe Leistung des Motors führte sonst zu einer zu starken Beschleunigung und Kontrollverlust. Stattdessen wird die Geschwindigkeit mit einem PID-Controller im ROS2-Motorsteuerungsknoten konstant gehalten. Die Verwendung des PID-Controllers führt bei seiner Einregulierung beim Start zu einer Anfahrverzögerung von einigen Sekunden. Dies wird in Kauf genommen: Ein ruckartiger Start führte zu Oszillationen der Geschwindigkeit mit Auswirkungen auf die Orientierungsfähigkeit.

Die Maximalgeschwindigkeit des Fahrzeugs ist im Wesentlichen durch die Abtastfrequenz des LIDAR-Scanners (10 Hz) bestimmt. Damit kann das Hindernisrennen nahezu fehlerfrei in ca. 90-100 Sekunden gefahren werden. Der Ortsabstand zwischen zwei Umgebungsscans liegt somit bei weniger als 5 cm. Dies ermöglicht eine sehr präzise Steuerung: Wird die Fahrgeschwindigkeit deutlich erhöht, steigt das Risiko für Kollisionen.

Vor dem Start des Rennens wird der vollgeladene Akku angeschlossen, das Fahrzeug bleibt aber stromlos. Die eigentliche Aktivierung erfolgt gem. Regelwerk mit einem einfachen Kippschalter. Nach dem Booten des Bordrechners wird per Autostart das ROS2-Netz mit den Steuerungsprogrammen gestartet und verharrt im Bereitschaftsmodus. Dies wird auf dem kleinen OLED-Display auf der Oberseite des Fahrzeugs angezeigt. Durch Berühren des daneben liegenden Touch Sensors wird der Rennmodus gestartet. Erst ab diesem Zeitpunkt werden Daten erfasst, Berechnungen gemacht und Steuerungskommandos ausgeführt.

Während der Test- und Trainingsfahrten erfolgt ein umfangreiches Logging, welches mit einem ssh-Login via WLAN an der Konsole zur Verlaufskontrolle und zum Debugging eingesehen werden kann.

# Zusammenfassung, Ausblick und Diskussion

## Ergebnisse

Mit dem vorgestellten Fahrzeug wurde der Nachweis erbracht, dass die Aufgabenstellung des autonomen Fahrens im Hindernisparcours mit einem für Edge Computing geeigneten, einfachen neuronalen Netzwerk (1D CNN) lösbar ist. Dem Team ist klar, dass die Aufgabenstellung grundsätzlich auch mit niedrigerem Aufwand (ohne LIDAR und teures Training des neuronalen Netzes) lösbar ist.

Der Vorteil der KI liegt in den sehr ausgeglichenen Fahrkurven und des quasi weitsichtigen Fahrverhaltens wie bei einem menschlichen Fahrer.

## Einschränkungen bzw. Verbesserungsmöglichkeiten

**Separates Training je eines KI-Modells für Fahrt im Uhrzeiger- und Gegenuhrzeigersinn erforderlich:** Es wird jeweils ein Netz für die Fahrt im Uhrzeiger- und im Gegenuhrzeigersinn trainiert. Theoretisch wäre dies nicht notwendig, wenn man bei der Steuerung die Symmetrieeigenschaften der Rennstrecke ausnutzte. Dies haben wir getestet: Allerdings stellte sich heraus, dass aufgrund der leicht asymmetrischen Bauweise des Fahrzeugs - insbesondere wegen des ausgestellten rechten Hinterrads zur Unterbringung des Geschwindigkeitsmessers - die Fahreigenschaften bei Links- und Rechtskurven zu unterschiedlich sind.

**Begrenzte Merkfähigkeit der KI:** Kämen weitere komplexe Fahraufgaben hinzu, müsste das KI-Modell in seiner Merkfähigkeit gesteigert werden. Enthält das Modell zu viele anpassbare Parameter, steigt allerdings der Rechenaufwand bei Interferencing an. Dies könnte dazu führen, dass ein Bordrechner mit höherer Rechenleistung erforderlich wird.

**Begrenzung der Fahrgeschwindigkeit durch die Abtastfrequenz des LIDAR-Sensors:** Die Kosten schnellerer LIDAR-Sensoren sind unverhältnismäßig höher. Außerdem sind diese Sensoren zu schwer, um die Gewichtshöchstgrenze einhalten zu können.

**Minimaler Kurvenradius:** Aufgrund der Konstruktion des Chassis und dem Maximalausschlag des Lenkservos ergibt sich ein minimaler Kurvenradius, der die Bewältigung einer Hinderniskonstellation ohne Rangiermanöver unmöglich macht. Das Rangieren mit abwechselnden Vor- und Rückwärtsfahrabschnitten ist konzeptionell nicht vorgesehen. Tritt diese Konstellation im Rennen auf, führt dies zu einem entsprechenden Fehler bzw. Punktabzug.

Ein Bild, das Reihe, Diagramm, parallel, Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 6: Nicht fahrbare Hinderniskonstellation

**Langsamfahrten:** Es wäre vorteilhaft, wenn die Getriebeübersetzung weiter erhöht werden könnte. Damit könnte das Fahrzeug langsamer fahren und präziser sehr kurze Fahrstrecken zurücklegen, wie dies beim Parken erforderlich ist. Leider war für das verwendete Chassis keine Zahnradkombination zu finden, die dies erlaubt hätte. Auf einen Austausch des kompletten Antriebsstrangs zu diesem Zweck wurde aus Platz- und Gewichtsgründen verzichtet.

**Einschränkung beim Parkmanöver durch Gewichtslimit (1500g):** Wir mussten zur Einhaltung des Maximalgewichts auf den Einbau weiterer Sensoren und Features zur Sicht nach hinten verzichten. Damit lässt sich beim Parken nicht die notwendige Präzision erreichen, wie sie die geringen Platzverhältnissen in der Parklücke erforderten. Verschärft wird dieses Problem durch das ruckartige Anfahrverhalten. Damit ist das Risiko einer Berührung der magentafarbenen Parkplatzhindernisse unattraktiv hoch. Wir beschränkten uns folglich auf eine Teilrealisierung gem. Regel 7.13:

Ein Bild, das Screenshot, Rechteck, Kinderkunst, Farbigkeit enthält.

Automatisch generierte Beschreibung

Abbildung 7: Fahrzeug teilweise eingeparkt gem. Regel 7.13

**Nur Vorwärtsfahrt mit KI:** Aktuell kann mit der KI nur vorwärts gefahren werden: Kreuzten sich beim Training Trajektorien, auf denen sowohl vorwärts als auch rückwärts gefahren wird, kann das Netzwerk keine eindeutige Steuerungsantwort auf die eingegebenen Umgebungsparameter ermitteln. Das Ergebnis wären zufällig aufeinanderfolgende Kommandos für Vor- und Rückwärtsfahrt. Dies hätte ein vollautomatisches Parkmanöver mit KI erschwert. Allerdings musste ja auf das vollständige Einparken aus anderen Gründen sowieso verzichtet werden. Um der KI die Unterscheidung der Fahrtrichtung zu erlauben, hätte ein aufwändigeres LSTM-Modell verwendet werden müssen, mit dem auch Zeitreihen trainiert werden können. Dies hätte den Trainingsaufwand wesentlich erhöht.

# Anhang: Quellcode zur Bildverarbeitung

|  |  |
| --- | --- |
| **Quellcode auf Open MV H7+:**  import sensor, image, time, math, pyb, os  import machine  # Get the unique machine ID  unique\_id = machine.unique\_id()  unique\_id\_hex = ''.join(['{:02x}'.format(byte) for byte in unique\_id])  usb = pyb.USB\_VCP()  red\_led = pyb.LED(1)  green\_led = pyb.LED(2)  blue\_led = pyb.LED(3)  red\_led.on()  green\_led.off()  blue\_led.off()  sensor.reset()  sensor.set\_pixformat(sensor.RGB565)  sensor.set\_framesize(sensor.QVGA)  sensor.set\_auto\_gain(True)  sensor.set\_auto\_whitebal(True)  sensor.skip\_frames(time = 2000)  green = (30, 100, -64, -8, -32, 32) # generic green  red = (30, 100, 15, 127, 15, 127) # generic red  magenta = (0, 100, 32, 127, 127, -94)  thresholds=[green, red, magenta]  roi = [0,0,320,140]  while True:  while usb.any():  data = usb.recv(4096) # Receive 64 bytes at a time  try:  img = sensor.snapshot()  img.lens\_corr(strength=2.6, zoom=1.0)  img.gamma\_corr(gamma=1.0,contrast = 1.0,brightnes =0.2)  blob\_entries = []  blobs = img.find\_blobs(thresholds,0,roi,pixels\_threshold=160, merge=False)  for blob in blobs:  (b\_x,b\_y,b\_w,b\_h) = blob.rect()  b\_c = blob.code()  if (b\_c == 4 and b\_h/b\_w < 1) or (b\_c in [1,2] and b\_h/b\_w > 1):  blob\_entries.append("{},{},{}".format(b\_c, b\_x, b\_x+b\_w))  bloblist = ','.join(blob\_entries)  if bloblist:  usb.write("{},".format(unique\_id\_hex))  usb.write(bloblist)  usb.write("\n")  usb.flush()  except Exception as e:  pass | **Quellcode des ROS2-Knotens zur Weiterverarbeitung der Kamerainformationen:**  def openmv\_h7\_callback(self, msg):  try:  data = msg.data.split(',')  if data[0] == '240024001951333039373338': cam =  elif data[0] == '2d0024001951333039373338': cam =  else: return  if cam == 1:  self.\_color1\_g = np.zeros(self.HPIX, dtype=int)  self.\_color1\_r = np.zeros(self.HPIX, dtype=int)  self.\_color1\_m = np.zeros(self.HPIX, dtype=int)  elif cam == 2:  self.\_color2\_g = np.zeros(self.HPIX, dtype=int)  self.\_color2\_r = np.zeros(self.HPIX, dtype=int)  self.\_color2\_m = np.zeros(self.HPIX, dtype=int)  blobs = ((data[i],data[i+1],data[i+2]) for i in range (1,len(data),3))  for blob in blobs:  color, x1, x2 = blob  color = int(color)  x1 = int(x1)  x2 = int(x2)  if color == 1:  if cam == 1 and not self.\_clockwise:  self.\_color1\_g[x1:x2] = self.WEIGHT  if cam == 2 and self.\_clockwise:  self.\_color2\_g[x1:x2] = self.WEIGHT  if color == 2:  if cam == 1 and not self.\_clockwise:  self.\_color1\_r[x1:x2] = self.WEIGHT  if cam == 2 and self.\_clockwise:  self.\_color2\_r[x1:x2] = self.WEIGHT  if color == 4:  if cam == 1: self.\_color1\_m[x1:x2] = self.WEIGHT  if cam == 2: self.\_color2\_m[x1:x2] = self.WEIGHT  self.\_parking\_lot += 1  except:  self.get\_logger().error('Faulty cam msg received: "%s"' % msg)  Hinweis: Zur korrekten Zuordnung der Kameras zur jeweiligen Blickrichtung nach links und rechts werden die Hardware-Adressen abgeglichen. |

# Anhang: Informationen zu den KI-Trainingsläufen und Tests

Bei den Trainingsläufen ist es wichtig, alle relevanten Konstellationen von Hindernissen zu trainieren. Zur Reduktion der Anzahl der Szenarien haben wir von den Symmetrieeigenschaften des Spielfelds Gebrauch gemacht.

Bei Tests haben wir ermittelt, dass eine Differenzierung des Trainings und der Daten für das Eröffnungs- und das Hindernisrennen nicht erforderlich ist. Das Eröffnungsrennen kann wie das Hindernisrennen ohne Hindernisse behandelt werden. Die unterschiedliche Positionierung und der unterschiedliche Grundriss des großen Hindernisses in der Mitte beim Eröffnungsrennen stellt für die KI keine sonderliche Herausforderung dar.

Für die Planung der Trainingsläufe haben wir alle denkbaren Hinderniskonstellationen aufgezeichnet und im Training dann sowohl im Uhrzeigersinn als auch im Gegenuhrzeigersinn abgefahren.

Ein Bild, das Diagramm, Symmetrie, Reihe, Rechteck enthält.

Automatisch generierte Beschreibung

Abbildung 8: Leere Seite aus dem Dokument zur Trainingsplanung

Für jede Umlaufrichtung waren ca. eine Stunde Fahrten zur Aufnahme der Trainingsdaten erforderlich. Weitere Trainingsfahrten waren für das Parkmanöver erforderlich. In Summe mussten knapp drei Stunden Daten aufgezeichnet werden. Aufgrund der Aufzeichnungsfrequenz von 10 Hz sammelten wir so insgesamt etwa 100.000 Datensätze in lesbaren .txt-Dateien. Diese ließen sich sehr flexibel kopieren, zerschneiden und kombinieren: Die Datensammlung haben wir in vielen einzelnen Trainingsläufen vorgenommen und die Daten erst für das KI-Training in einer Datei zusammengefasst.

Das Trainieren der KI-Modelle benötigt aufgrund der eingesetzten Hardware (High End-Grafikkarte mit CUDA-Bibliothek) nur wenige Minuten. Ohne die Grafikkarte hätte die Berechnung über eine Woche benötigt. Dies hätte den Test- und Entwicklungsprozess unmöglich gemacht.

Die Vorhersagegenauigkeit unserer KI-Modelle für das jeweils nächste Fahrmanöver liegt bei mehr als 80%. Dies ist mehr als ausreichend: Haupteinfluss auf die Präzision der Steuerung hat ja die Fahrgeschwindigkeit bei konstanter LIDAR-Scanfrequenz (10Hz). Eine durch die KI verursachte Ungenauigkeit kann somit bereits nach 0.1 s beim nächsten Steuerkommando kompensiert werden.

Die fertig trainierten drei KI-Modelle (je eines für das Hindernisrennen im Uhrzeigersinn und im Gegenuhrzeigersinn) haben wir auf die SD-Karte des Bordrechners kopiert. Sie werden beim Bootvorgang und dem Autostart des ROS2-Netzwerks in den Speicher geladen und stehen dann fürs Interferencing zur Verfügung. Je nach Fahrsituation wird zwischen ihnen umgeschaltet.

Die für die Planung der Trainingsläufe verwendete Unterlage haben wir auch zur Testfalldokumentation verwendet: Wir haben systematisch alle Konstellationen von Hindernissen ausprobiert. Bei den Testfahrten identifizierte Fehler wie z.B. die Berührung von Hindernissen konnten wir durch Nachtrainieren mit zusätzlichen Daten für die spezielle Fehlersituation ausmerzen. Um dabei nicht die Übersicht zu verlieren, haben wir zusätzlich die Fehler mittels Videoaufnahmen mit dem Handy dokumentiert.

# Glossar

**1D CNN** (eindimensionales Convolutional Neural Network, im Programm auch **Conv1D** genannt): Neuronales Netzwerk zur Verarbeitung sequentieller, eindimensionaler Daten

**Adam-Optimierer:** Algorithmus zur Optimierung von neuronalen Netzwerken, der auf stochastischem Gradientenabstieg basiert und adaptive Schätzungen von niedrigeren Ordnungen verwendet.

**Antriebsritzel:** Kleines Zahnrad auf der Antriebsachse des Elektromotors. Mit der Größe des Ritzels lässt sich das Übersetzungsverhältnis anpassen.

**Dense Layer:** Ein vollständig verbundener Layer in einem neuronalen Netzwerk, in dem alle Eingaben mit allen Ausgaben verbunden sind. Die Anzahl der Dense Layer entscheidet über die Gedächtnisleistung des neuronalen Netzwerks.

**CUDA:** Eine von Nvidia entwickelte Programmierschnittstelle, die es ermöglicht, allgemeine Berechnungen auf Grafikprozessoren (GPUs) durchzuführen. Durch CUDA und eine geeignete Grafikkarte (hier Nvidia RTX 4070 TI) lassen sich die Berechnungen mit Tensorflow drastisch beschleunigen.

**Dremel:** Ein Handwerkzeug, das oft für Präzisionsarbeiten wie Schleifen, Schneiden oder Gravieren verwendet wird.

**ESC** (Electronic Speed Controller): Steuermodul für Geschwindigkeit und Richtung von Elektromotoren, häufig in Drohnen und RC-Fahrzeugen verwendet.

**Ethernet:** Technologie für lokale Netzwerke (LANs), die die Kommunikation zwischen Geräten über Kabel ermöglicht. Mit Ethernet werden der Bordrechner (Raspberry PI) und der LIDAR Scanner verbunden.

**GPIO** (General Purpose Input/Output): Pins an Mikrocontrollern und Computern, die zur Interaktion mit anderen elektronischen Komponenten verwendet werden. Über diese Schnittstelle werden die Sensoren und die Steuerkomponenten an den Raspberry PI angebunden.

**Gyroskop:** Sensor auf dem Raspberry Pi Sense Hat zur Messung der Orientierung und Winkelgeschwindigkeit

**Hall-Sensor:** Umdrehungszähler am Elektromotor basierend auf kleinen Magneten, deren elektrische Felder beim Rotieren elektrische Signale erzeugen. Der Hall-Sensor wird benötigt, um beim Langsamfahren das Cogging (Stottern) des Motors mit dem ESC zu unterdrücken.

**I2C-Bus:** Kommunikationsbus des Raspberry Pi zur Anbindung von externen Sensoren oder Aktoren

**Interferencing:** Prozess des Anwendens eines trainierten KI-Modells auf neue Daten zur Vorhersage oder Analyse, wird genutzt um die Lenkbefehle des Servos zu errechnen.

**KI-Modell** (Künstliche Intelligenz-Modell): Ein computergestütztes Modell, das darauf trainiert ist, menschenähnliche Aufgaben durchzuführen oder Entscheidungen zu treffen.

**LIDAR** (Light Detection and Ranging): Eine Technik zur Entfernungsmessung, die Licht in Form eines gepulsten Lasers verwendet. Beim verwendete 1D-LIDAR rotiert der Laser mit einer Frequenz von 10 Hz und einem Puls von 32 kHz. Damit werden für einen Vollkreis (360 Grad) 3240 Entfernungsdatenpunkte ermittelt und via Ethernet übertragen.

**LIPO-Akku** (Lithium-Polymer-Akku): Wiederaufladbarer Akku, bekannt für seine hohe Energiedichte und häufig verwendet in mobilen Geräten und RC-Fahrzeugen.

**LSTM** (Long Short-Term Memory): Typ von rekurrenten neuronalen Netzwerken, die speziell dafür entwickelt wurden, Langzeitabhängigkeiten zu lernen.

**microSD-Karte**: Kleiner, tragbarer Speicherkartentyp, der häufig in Mobilgeräten verwendet wird. Der Raspberry PI verfügt über einen microSD-Kartenslot als Alternative zu einem externen Speichersystem.

**OLED-Display** (Organic Light Emitting Diode): Display-Typ, der organische Materialien verwendet, die Licht emittieren, wenn elektrischer Strom fließt. Sehr kleine und leichte Bauweise möglich.

**OPEN MV H7:** Kleines, leistungsstarkes Computer-Vision-Modul, das für maschinelles Sehen und ML-Anwendungen optimiert ist. Die Kamera (mit Fisheye-Objektiv) ist direkt auf der Platine mit dem Prozessor-Chip aufgebracht, dadurch sehr kompakte, leichte Bauweise. USB-Schnittstelle verfügbar.

**Raspberry PI 4B:** Kleiner, preiswerter Computer, der häufig in Bildung, Hobbyelektronik und DIY-Projekten verwendet wird.

**ROS2** (Robot Operating System 2): Open Source Set von Softwarebibliotheken und Tools, das hilft, Robotikanwendungen zu bauen. Unterstützt werden Programm-Module in den Programmiersprachen Python und C/C++.

**SSD.2 Drive:** Solid-State-Drive (SSD) für schnellen Datenzugriff und verbesserte Leistung mit USB-Schnittstelle. Kann alternativ auch mit dem Raspberry PI verwendet werden.

**Tensorflow:** Open-Source-Softwarebibliothek für maschinelles Lernen, entwickelt von Google, um neuronale Netzwerke zu trainieren und einzusetzen.

**Ubuntu Linux:** Populäres, benutzerfreundliches Betriebssystem auf Basis von Linux, bekannt für seine Stabilität und Community, auch für Raspberry PI verfügbar.

# Literaturverzeichnis

1. Mastering ROS for Robotics Programming - Third Edition: Best practices and troubleshooting solutions when working with ROS Taschenbuch – 15. Oktober 2021, Englisch Ausgabe von Lentin Joseph (Autor), Jonathan Cacace (Autor)
2. Learn Robotics Programming - Second Edition: Build and control AI-enabled autonomous robots using the Raspberry Pi and Python Taschenbuch – 12. Februar 2021, Englisch Ausgabe von Danny Staple (Autor)
3. Algorithmen in Python: Das Buch zum Programmieren trainieren. 32 Klassiker der Informatik, von Damenproblem bis Neuronale Netze Broschiert – 28. Juni 2020 von David Kopec (Autor)
4. Think Python: Systematisch programmieren lernen mit Python (Animals) Taschenbuch – 1. Juli 2021 von Allen B. Downey (Autor), Peter Klicman (Übersetzer)
5. Let’s code Python: Programmieren lernen mit Python ohne Vorkenntnisse. Ideal für Kinder und Jugendliche Taschenbuch – 28. Dezember 2018 von Hauke Fehr (Autor)
6. Parallel and High Performance Programming with Python: Unlock parallel and concurrent programming in Python using multithreading, CUDA, Pytorch and Dask. (English Edition) Taschenbuch – 14. April 2023, Englisch Ausgabe von Fabio Nelli (Autor)
7. Hands–On Machine Learning with Scikit–Learn and TensorFlow Taschenbuch – 24. März 2017, Englisch Ausgabe von Aurelien Geron (Autor)
8. Hands-on Convolutional Neural Networks with Tensorflow Taschenbuch – 29. August 2018, Englisch Ausgabe von Iffat Zafar (Autor), Giounona Tzanidou (Autor)
9. Neural Networks and Deep Learning: A Textbook Gebundene Ausgabe – 13. September 2018, Englisch Ausgabe von Charu C. Aggarwal (Autor)
10. Statistik: Der Weg zur Datenanalyse (Springer-Lehrbuch) Taschenbuch – 15. September 2016, von Ludwig Fahrmeir (Autor), Christian Heumann Rita Künstler
11. CUDA by Example: An Introduction to General-Purpose GPU Programming Taschenbuch – Illustriert, 19. Juli 2010, Englisch Ausgabe von Jason Sanders / Kandrot (Autor)
12. Raspberry Pi: Kompendium: Linux, Programmierung und Projekte Taschenbuch – 29. Juni 2020 von Sebastian Pohl (Autor)
13. Das offizielle Raspberry Pi Handbuch: Von den Machern von MagPi, dem offiziellen Raspberry Pi Magazin Taschenbuch – 12. Februar 2024 von Elektor International Media (Herausgeber)
14. Mastering Ubuntu Server - Fourth Edition: Explore the versatile, powerful Linux Server distribution Ubuntu 22.04 with this comprehensive guide Taschenbuch – 22. September 2022 Englisch Ausgabe von Jay LaCroix (Autor)
15. Linux Pocket Guide Taschenbuch – 23. Juni 2016, Englisch Ausgabe von Daniel J. Barrett (Autor)
16. OpenAI Chat GPT 4.0 (diverse Recherchen)