UNIVERSITÄT ZU LÜBECK

Follow-Me Algorithmus für den Jackal-Roboter

Bachelorprojekt

WS 2021/2022

Erarbeitet von:

Lydia Engelhardt (lydia.engelhardt@student.uni-luebeck.de)

Michaela Stein (michaela.stein@student.uni-luebeck.de)

Betreut durch:

Georg Schildbach (georg.schildbach@uni-luebeck.de)

Inhaltsverzeichnis

1. Einleitung

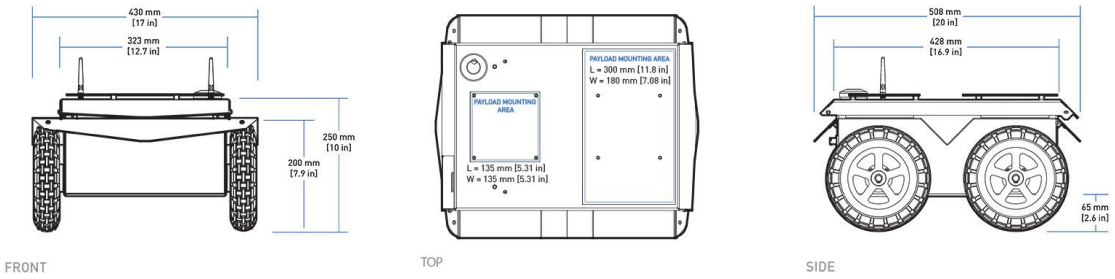
“Bei Fuß!” Mehr braucht es nicht, dass ein gut trainierter Hund dem Herrchen dicht hinterher folgt. Der Outdoor-Roboter Jackal hört zwar nicht auf Hundebefehle, aber mit Hilfe des von uns implementierten “Follow Me”- Algorithmus folgt der Roboter autonom einem Menschen in Simulation und im Gelände. Dies ist einfacher und weniger aufwändig als den Jackal selbstständig durch die Natur zu steuern. Außerdem kann der Roboter gegebenenfalls Gegenstände transportieren, um den Menschen zu entlasten. Die Person, der der Roboter folgen soll, trägt ein Smartphone bei sich, dessen GPS-Daten mit Hilfe der ROS-Mobile App an den Jackal gesendet werden. Dieser berechnet zusammen mit seiner eigenen GPS-Position und Orientierung die Richtung, in die er fahren muss. Weitere Verhalten wie Avoid und Stop dienen zur Kollisionsvermeidung und Sicherheit und verbessern den Algorithmus.

2. Vorstellung Jackal

Jackal ist der Name eines kleinen, schwarz-gelben Roboters, der für den Einsatz im Gelände entwickelt wurde. Er zählt zu den UGV (englisch für unmanned ground vehicle - unbemanntes Landfahrzeug), die sich ohne Fahrer ferngesteuert oder autonom fortbewegen (“Unbemanntes Landfahrzeug – Wikipedia”).

Grundausstattung des Jackal
(The Construct)

Das Ziel der in Kanada ansässige Firma Clearpath Robotics ist es, die Robotikforschung für Universitäten und private Unternehmen zu modernisieren (“Clearpath Robotics”).
Die verschiedenen Roboter dienen dabei als Plattform für den einfachen Einstieg und können durch verschiedene Zubehörteile individuell und modular an die Bedürfnisse der Kunden angepasst werden. Die Zusatzausrüstung kann dabei im Inneren oder auf der Montageplattform oben auf dem Roboter angebracht werden. Die Grundausstattung des Jackal beinhaltet einen fahrzeugeigenen Computer, ein GPS Modul, W-LAN sowie eine IMU (“Jackal UGV - Small Weatherproof Robot - Clearpath”). Die inertiale Messeinheit (englisch inertial measurement unit, IMU) kombiniert Daten des Beschleunigungssensors, des Gyroskops und des Magnetometers um die Orientierung, die Winkelgeschwindigkeit und die lineare Beschleunigung zu schätzen (Bovbel) (“Inertiale Messeinheit – Wikipedia”).



Technische Zeichnung des Jackal
von vorne, oben und der Seite
(“Jackal UGV - Small Weatherproof Robot - Clearpath”)

Der 50x43x25cm große Roboter wiegt ohne Zusatzausstattung 17 kg und kann bis zu 20 kg Nutzlast tragen. Das Gehäuse ist passend für die Nutzung in der Natur wasserfest und kann Temperaturen von -20°C bis +45°C aushalten. Der Jackal ist mit vier Luftreifen sowie einen Allradantrieb ausgestattet, erreicht eine Höchstgeschwindigkeit von 2 m/s und hat eine Akkulaufzeit von je nach Nutzung zwei bis acht Stunden.
Es gibt drei mögliche Steuermodi: durch Angabe der linearen Geschwindigkeit und Winkelgeschwindigkeit, mittels Angabe der Spannung für die Motoren oder direkt durch Befehle mit den Geschwindigkeiten für die Räder. Für die einfache drahtlose Bedienung ist ein Gamecontroller beigefügt (“Jackal UGV - Small Weatherproof Robot - Clearpath”).

Jackal mit der von uns verwendeten Ausstattung

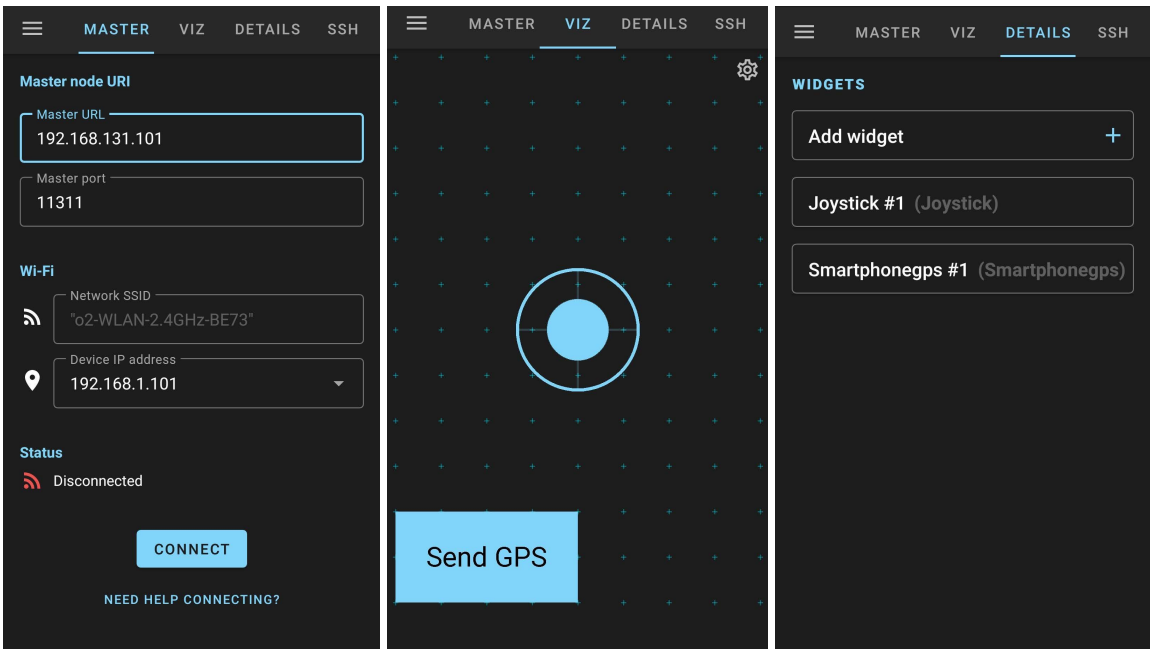
In unserem Projekt wurde zusätzlich ein Bluetooth-Modul, ein Jetson Computer als Grafikprozessor sowie ein LiDAR, der 2D- und 3D-Laserscans anfertigen kann, eingesetzt. Des Weiteren wurde Ubuntu 16 und das Robot Operating System ROS verwendet.

3. Teilkomponenten des Algorithmus

Im Folgenden werden die einzelnen Softwarekomponenten beschrieben, die wir für die Realisierung des Gesamtverhaltens entwickelt haben.

3.1 Erweiterung der ROS-Mobile App

ROS-Mobile ist eine von Nils Rottmann und Nico Studt an der Universität zu Lübeck entwickelte Android-App für einfache und dynamische Visualisierung und Steuerung von mobile Robotern, die mit ROS laufen. Die Kommunikation mit dem Roboter läuft über Subscriber- und Publisher-Nodes und standardisierten ROS-Nachrichten (Rottmann et al. #) (Rottmann and Studt).
Die Programmierung der App erfolgte in Java und baut auf dem Model View ViewModel (MVVM) Architekturmuster auf. Dies dient zur Trennung von Logik und Darstellung der Benutzerschnittstelle und erlaubt einfache Änderungen und Erweiterungen (“Model View ViewModel – Wikipedia”).
Die hier wichtigsten Hauptbestandteile von ROS-Mobile sind der Master, Viz und Details. Im Master-Fenster kann mittels der IP-Adresse eine Verbindung über das WLAN des Roboters aufgebaut werden. Im Details-Reiter können Komponenten, Widgets genannt, mit bestimmten Funktionalitäten hinzugefügt werden, die dann unter Viz dargestellt werden. Verfügbare Widgets sind beispielsweise ein Joystick für die Steuerung des Roboters, eine Batterieanzeige und die Übertragung des Kamerabildes des Roboters.



Screenshots der App ROS Mobile

In unserem Projekt nutzen wir die Anwendung, um die GPS-Daten des Smartphones über Wi-Fi an den Jackal-Roboter zu senden. Um dies zu realisieren, haben wir ein eigenes Widget namens “Smartphonegps” mit entsprechender Funktionalität programmiert und in die App eingebunden.

Für die Erstellung eines neuen Widgets werden grundlegend vier neue Klassen benötigt: Entity, View, Data und Detail Viewholder (“How to contribute?”).

In der Entitätsklasse werden die Objekteigenschaften und Standardwerte definiert. In unserem Fall legen wir die Größe und den Text für den Button sowie den Topic-Namen und die ROS-Nachrichtenart fest. Da wir GPS-Daten verschicken, wählen wir die NavSatFix-Nachricht.

```
public SmartphonegpsEntity() {
   this.width = 4;
   this.height = 2;
   this.text = "Send GPS";
   this.topic = new Topic("jackal_gps", NavSatFix._TYPE);
}
```

Codeausschnitt aus `SmartphonegpsEntity.java`

Darüber hinaus werden in der Daten-Klasse die ROS-Messages erstellt und die entsprechenden Werte für Längengrad (Longitude), Breitengrad (Latitude) und Höhe über dem Meeresspiegel (Altitude) zugewiesen.

```
public Message toRosMessage(Publisher<Message> publisher, BaseEntity
widget) {
   sensor_msgs.NavSatFix message = (NavSatFix) publisher.newMessage();

   message.setLongitude(longitude);
   message.setLatitude(latitude);
   message.setAltitude(altitude);

   return message;
}
```

Codeausschnitt aus SmartphonegpsData.java, Methode toRosMessage

Die Darstellung und Interaktion des Widgets wird in der View-Klasse definiert. Unser Button wird als einfaches Rechteck mit den zuvor spezifizierten Eigenschaften gezeichnet. Bei Knopfdruck wird ein Handler aktiviert, der jede Sekunde eine neue GPS-Nachricht sendet. Handler sind gut dazu geeignet, Nachrichten asynchron im Hintergrund zu versenden. Die Methode sendGps() erstellt dann ein vorerst leeres SmartphonegpsData-Objekt.

```
Handler handler = new Handler();
int delay = 1000; //milliseconds

handler.postDelayed(new Runnable(){
   public void run(){
       if(sendingGPS){    //sendingGPS wird bei Knopfdruck auf True gesetzt
           sendGps();}
       handler.postDelayed(this, delay);
   }
}, delay);
```

Codeausschnitt aus SmartphonegpsView.java aus der Methode init

Zudem wird in der View-Klasse überprüft, ob die ROS-Mobile App die notwendigen Berechtigungen hat, um auf das GPS des Smartphones zugreifen zu können.
Die Detail Viewholder-Klasse initialisiert die View und aktualisiert die Entity falls nötig.

Um die tatsächlichen GPS-Koordinaten des Smartphones abzufragen, wurde der LocationUpdateService verwendet. In Android ist ein Service ein Dienst, der lang andauernde Operationen im Hintergrund ausführen kann (“Services overview”).

Der LocationUpdateService fragt Daten des FusedLocationProviderClient an, ein Dienst von Google, der mithilfe der Kombination von GPS und WIFI möglichst akkurate Standortinformationen bereitstellt (“Fused Location Provider API”).
Für die Speicherung der GPS-Daten gibt es die Klasse LocationDTO. DTO steht hierbei für data transfer object, zu deutsch Datentransferobjekt. Es wird zuerst ein Standortupdate des FusedLocationClients angefragt und dann in der Methode LocationCallback ein neues Objekt der Klasse LocationDTO erstellt und ihm die Daten zugewiesen.
Anschließend wird eine neue Nachricht auf den Eventbus verschickt. Der Eventbus ermöglicht die Kommunikation zwischen verschiedenen Klassen per Publisher-Subscriber-Pattern über sogenannte Events (“EventBus: Events for Android - Open Source by greenrobot”).
Dafür wird das LocationDTO-Objekt noch in ein Objekt der Klasse LocationUpdateEvent verpackt, bevor es auf dem Eventbus versendet wird.

```
private void startLocationUpdates() {

   mFusedLocationClient.requestLocationUpdates(this.locationRequest,
           this.locationCallback, Looper.myLooper());
}
```

Codeausschnitt aus LocationUpdateService.java, Methode startLocationUpdate

```
private LocationCallback locationCallback = new LocationCallback() {
   @Override
   public void onLocationResult(LocationResult locationResult) {
       super.onLocationResult(locationResult);
       Location currentLocation = locationResult.getLastLocation();
       LocationDTO location = new LocationDTO();
       location.latitude = currentLocation.getLatitude();
       location.longitude = currentLocation.getLongitude();
       location.speed = currentLocation.getSpeed();
       location.altitude = currentLocation.getAltitude();

       EventBus.getDefault().post(new LocationUpdateEvent(location));
   }
};
```

Codeausschnitt aus LocationUpdateService.java, Methode locationCallback

Da die Kommunikation mit dem Eventbus nur mit UI-Klassen wie Activities und Fragments sowie Hintergrund-Threads funktioniert, konnten die Events nicht direkt in der SmartphonegpsData- oder SmartphonegpsView-Klasse gelesen und die Daten in die rosMessage eingefügt werden (“EventBus: Events for Android - Open Source by greenrobot”).
Zur Lösung dieses Problems haben wir weiteren Code in die bereits bestehende VizFragment-Klasse eingefügt. Ein Fragment ist ein Teil der Benutzeroberfläche oder des Verhaltens einer Anwendung und stellt eine bestimmte Operation oder Schnittstelle dar (“Fragment”).
Das VizFragment ist in der ROS-Mobile App für die Visualisierung der Widgets zuständig. Die neu geschriebene Methode onMessageEvent wartet nun auf die Veröffentlichung von Nachrichten des Typs LocationUpdateEvent auf dem Eventbus und speichert dann die in den Events enthaltenen Informationen in Klassenvariablen zwischen.

```
@Subscribe(threadMode = ThreadMode.MAIN)
public void onMessageEvent(LocationUpdateEvent event) {

   latitude = event.getLocation().latitude;
   longitude = event.getLocation().longitude;
   altitude = event.getLocation().altitude;

}
```

Codeausschnitt aus VizFragment.java, Methode onMessageEvent

Des weiteren überprüft die Methode onNewWidgetData nun auch, ob die Daten vom Typ SmartphonegpsData sind. Ist dies der Fall, so wird das bis dahin leere Datenobjekt mit den Standortinformationen gefüllt, bevor die Nachricht veröffentlicht und an den Roboter gesendet wird.

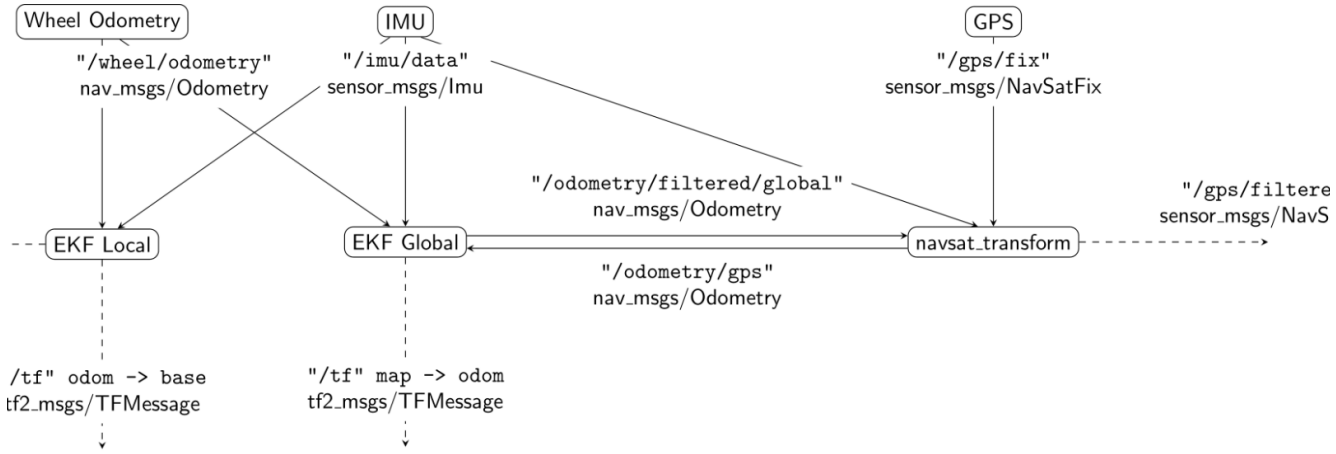
```
public void onNewWidgetData(BaseData data) {

       if (data instanceof SmartphonegpsData) {
           ((SmartphonegpsData) data).setGPS(longitude, latitude,
altitude);
       }
       mViewModel.publishData(data);
   }
```

Codeausschnitt aus VizFragment.java, Methode onNewWidgetData

3.2 Robot Localization Package

Damit wir genauere GPS-Positionen des Jackals für die Verfolgung des Smartphones erhalten, haben wir die GPS-Position des Jackals mit Hilfe des robot_localisation package geglättet. Dazu haben wir mit den ekf_localization_node die IMU Daten mit den Odiometrie Daten verschmolzen und diese wiederum an den navsat_transform_node weiter gegeben. Der navsat_transform_node hat dann diese Daten, sowie die originalen IMU Daten und die GPS Position des Jackals zu einer neuen GPS Position verschmolzen, die wir dann für das FollowMe Verhalten benutzt haben.



Einstellung für die Benutzung des navsat_transform_node (Moore)

Damit die Abschätzung der GPS Position möglichst genau ist haben wir in den Konfigurationseinstellungen der einzelnen Sensoren die Begrenzungen des Roboters beachtet und so zum Beispiel die Werte der Beschleunigung der IMU in die Z-Richtung nicht mit verschmolzen, da der Jackal nicht hoch und runter fahren kann und sonst nur Fehler mit in die Berechnungen der geschätzten GPS Position mit einbezogen würden.

Aufbau der Konfiguration eines Sensors:

$$(X, Y, Z, roll, pitch, yaw, \dot{X}, \dot{Y}, \dot{Z}, \dot{roll}, \dot{pitch}, \dot{yaw}, \ddot{X}, \ddot{Y}, \ddot{Z})$$

Einstellung für die Benutzung des navsat_transform_node (Moore and Stouch)

Unsere Konfiguration für die Daten der IMU:
<rosparam param="/imu/data_config">
[false, false, false,
false, false, false,
true, true, false,
false, false, true,
false, false, true]</rosparam>

3.3 Verhalten

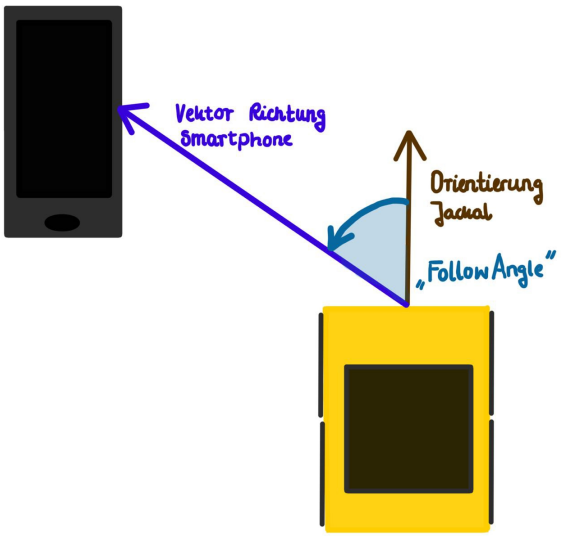
Damit der Jackal der GPS-Position des Smartphones folgen kann ohne dabei gegen Hindernisse zu fahren haben wir eine Reihe von Verhalten implementiert, die jeweils eine Twist-Message publishen. Im Anschluss werden diese Verhalten durch den Arbiter kombiniert, der die kombinierte Twist-Message schlussendlich auch zu den Motoren des Jackals weiterleitet. Dadurch wird das gewünschte Gesamtverhalten erreicht.

Im Anschluss werden die Verhalten im Detail erklärt und die genaue Funktionsweise des Arbiters.

3.3.1 Follow-Me

Das Follow-Me Verhalten berechnet anhand der GPS-Position des Smartphones und des Jackals und der Orientierung des Jackals in der x-y-Ebene (-Die Orientierung des Jackals erhalten wir durch das Magnetometer-), wie stark und in welche Richtung sich der Jackal drehen muss damit er zum Smartphone fährt und wie schnell er Fahren soll.

Dazu werden die geglätteten GPS-Positionen des Smartphones und des Jackals, sowie die Orientierung des Jackals, kontinuierlich (alle 10 Hz) eingelesen.
Daraufhin wird der Winkel zwischen der Orientierung des Jackals und dem Smartphones berechnet. Aus dem Winkel wird die Rotationsgeschwindigkeit berechnet, die proportional zum Winkel größer oder kleiner wird.
Um die Lineargeschwindigkeit des Jackals zu bestimmen, wird der Abstand zwischen dem Smartphone und dem Jackal berechnet. Je kleiner der Abstand ist, desto langsamer fährt der Jackal. Unterschreitet der Abstand einen bestimmten Schwellwert (1,5 m), so bleibt der Jackal stehen.
Wenn die Rotations- und die Lineargeschwindigkeit berechnet wurden, werden diese Daten mit Hilfe einer Twist Message auf das “FollowMe”-Topic gepublisht.
Zu dem wird noch der Winkel zwischen dem Smartphone und der Orientierung des Jackals auf ein Topic Namens “FollowAngle” gepublisht, damit der Arbiter das Avoid Verhalten mit dem Follow-Me Verhalten kombinieren kann.



3.3.2 Stop

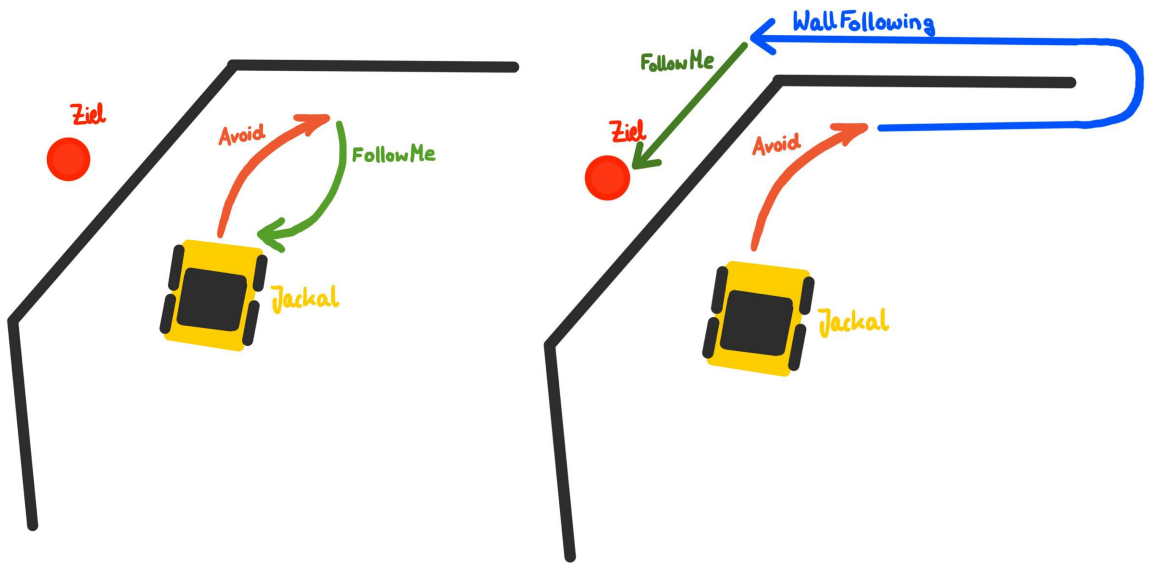
Das Stop Verhalten dient der Sicherheit des Jackals und publisht lediglich eine Twist-Message mit einer Linear- und Rotationsgeschwindigkeit von 0 m/s auf das “Stop”-Topic wenn der LiDAR des Jackals in irgendeine Richtung einen geringeren Abstand als 0,4 m misst.

3.3.3 Avoid

Das Avoid Verhalten dient dazu, dass der Jackal Hindernissen ausweicht. Hierzu wird mit Hilfe der LiDAR Daten ein abstoßendes Potentialfeld erzeugt.

3.3.4 Wall-Following

Das Wall-Following Verhalten ermöglicht es dem Jackal einer Wand folgen zu können.
Dies wird zusätzlich zu dem Avoid und Follow-Me Verhalten benötigt, da es Situationen gibt in denen der Jackal einem Hindernis durch das abstoßende Potentialfeld ausweicht bis dieses keine abstoßende Kraft mehr erzeugt. Daraufhin steuert er wegen der gewünschten Fahrtrichtung des Follow-Me Verhaltens wieder auf das Hindernis zu und verfängt sich somit in einer sich unendlich wiederholenden Schleife. Durch das Wall-Following ist es möglich aus dieser Schleife auszubrechen. Denn dann wird das Hindernis umrundet bis wieder eine direkte Verbindung zum Ziel gefahren werden kann auf der keine Hindernisse detektiert werden (Bug2-Verhalten).



ohne Wall-Following mit Wall-Following

Da es in der Natur keine richtigen Wände gibt, sondern eher Baum Alleen, Wälle oder Büsche in einer Reihe, haben wir drei Bereiche (links, vorne, rechts) durch den LiDAR abgetastet und daraus die jeweilige mittlere Distanz zu Hindernissen berechnet, um zu vermeiden das z.B. vermeintlich auf der Linken seite kein Hinderniss erkannt wird, obwohl dort nur eine Lücke zwischne zwei Bäumen ist.

```
if left_mean_dist >= right_mean_dist:
            # Follow right wall
            self.follow_right = True
            msg.angular.z = (-1.0) * self.correction_strength *
(self.desired_dist - right_mean_dist)
      else:
            # Follow left wall
            self.follow_right = False
            msg.angular.z = self.correction_strength * (self.desired_dist -
left_mean_dist)
      if front_mean_dist < self.desired_dist + 0.4:
            if self.follow_right:
                  # turn hard left:
                  msg.angular.z = -2
            else:
                  # turn hard right:
                  msg.angular.z = 2
```

Codeausschnitt aus WallFollowing.py aus der Methode get_vector

In diesem Codeausschnitt ist zu sehen wie wir entschieden haben, ob der Jackal entlang der linken oder der rechten Wand fahren soll. Der Abstand nach vorne wird zudem bedacht, da es sonst möglich wäre das der Jackal bei starken rechts oder links Kurven mit der Wand kollidiert.
Zudem sieht man, dass wir die Rotationsgeschwindigkeit in Abhängigkeit des Abstands zu dem Hindernis bestimmt haben. Die Lineargeschwindigkeit haben wir konstant auf 1 m/s gesetzt.
Wie auch bei den anderen Verhalten wird eine Twist-Message mit der Linear- und Rotationsgeschwindigkeit auf das Topic des Verhaltens (hier: “WallFollowing”) gepublisht.

3.3.5 Festgefahren

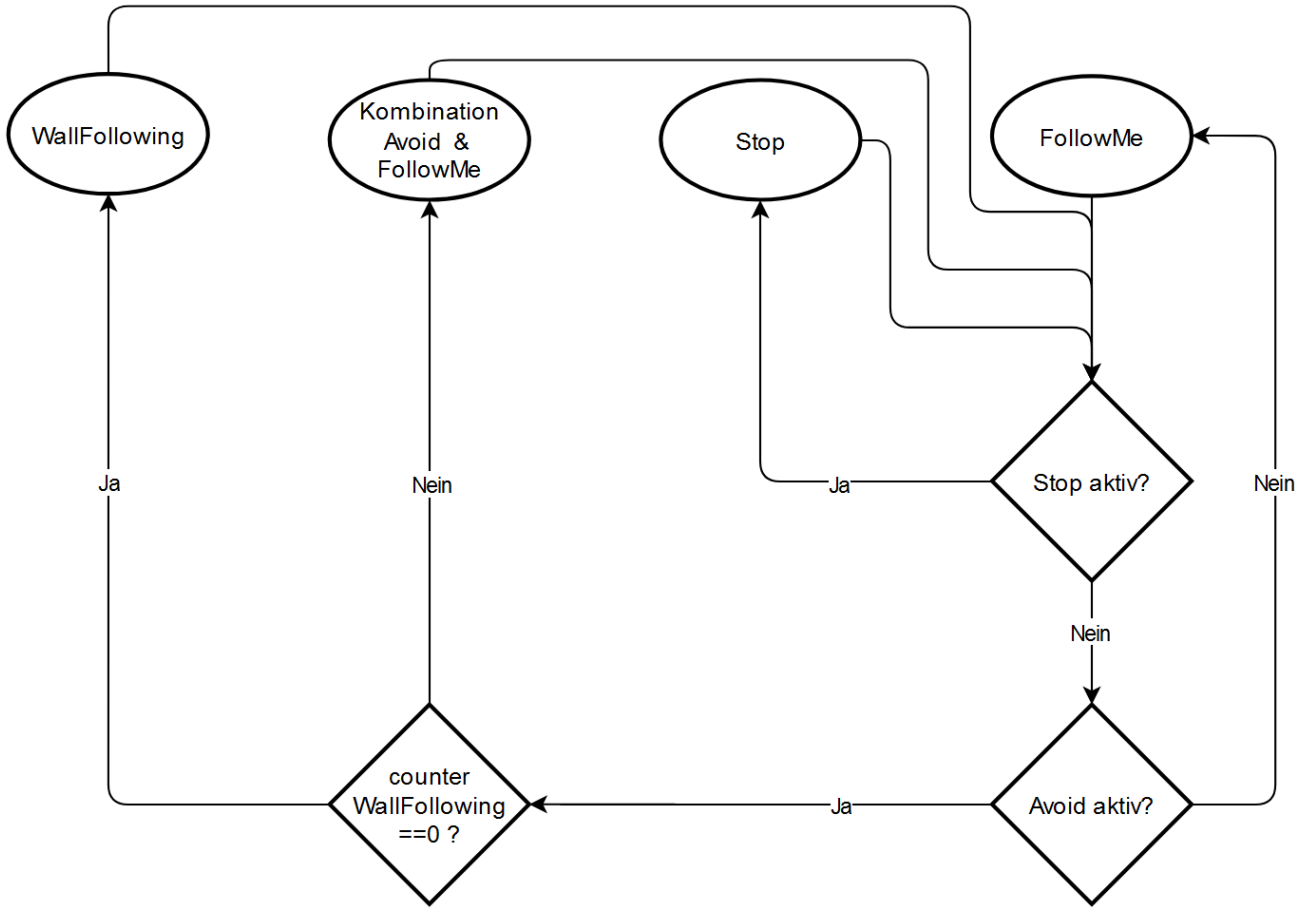
Dieses Verhalten erkennt, ob der Roboter sich festgefahren hat und die Räder sich weiter drehen, ohne dass der Jackal sich von der Stelle bewegt. Dafür wird die Twist Nachricht abgehört. Ist die Geschwindigkeit, mit der die Motoren sich bewegen sollen, größer null, obwohl sich die GPS-Position des Jackals über einen gewissen Zeitraum nicht wesentlich verändert hat, so wird eine Stop-Nachricht gesendet.

3.3.6 Arbiter

Der Arbiter kombiniert alle Verhalten, um das gewünschte Verhalten, der Smartphone GPS-Position zu folgen ohne dabei mit einem Hindernis zu kollidieren, zu erhalten.

Dazu werden alle Topics der Verhalten abonniert, sowie das des LiDARs.
Das Stop-Verhalten hat die höchste Priorität, d.h. wenn es aktiv ist überschreibt es alle anderen Verhalten. Wenn das Stop Verhalten nicht aktiv ist, aber die Twist-Message des Avoid Verhaltens Werte ungleich 0 m/s enthält, wird das Avoid und das FollowMe Verhalten kombiniert. Zudem gibt es einen Counter, der das WallFollowing Verhalten aktiviert, wenn dieser abgelaufen ist. Der Counter wird immer initialisiert, wenn das vorherige Verhalten ein anderes als Aovid oder

WallFollowing war und das aktuelle Verhalten Avoid ist. Der Counter wird reduziert, wenn das Vorherige Verhalten Avoid war und das aktuelle Verhalten Avoid ist.
Das WallFollowing Verhalten wird deaktiviert sobald in Richtung des FollowAngle keine Hindernisse durch den LiDAR erkannt werden oder wenn weder rechts noch links eine Wand erkannt wird.
Das FollowMe Verhalten alleine ist nur aktiv wenn alle anderen Verhalten nicht aktiviert sind.



Ablaufplan der Verhalten

4. Fazit

Zusammenfassend haben wir einen Algorithmus entwickelt, der in Simulation einem bewegten GPS-Signal folgen und sinnvoll auf Hindernisse reagieren kann. Leider konnten wir das Follow-Me-Verhalten aufgrund von Technikproblemen nicht in echt auf dem Roboter testen und somit die Verhaltensweise des Jackals in der Natur nicht untersuchen. Im Gelände herrschen komplexere Anforderungen wie Wurzeln und Bodenunebenheiten, die wir auf dem Computer nicht ausreichend simulieren konnten. Das Verhalten des Algorithmus in der Natur könnte in einem weiterführenden Projekt genauer beleuchtet werden. Außerdem besteht die Möglichkeit, das Programm mit intelligenteren Verhalten und lokalem Mapping mit Pfadplanung zu ergänzen.

5. Literaturverzeichnis

Bovbel, Paul. “Conventions for IMU Sensor Drivers.” *REP 145 -- Conventions for IMU Sensor Drivers (ROS.org)*, 2 February 2015, https://www.ros.org/reps/rep-0145.html. Accessed 28 February 2022.

“Clearpath Robotics.” *Wikipedia*, https://en.wikipedia.org/wiki/Clearpath_Robotics#Clearpath_Robotics. Accessed 28 February 2022.

The Construct. “Jackal Vehicle.” *The Construct*, https://www.theconstructsim.com/wp-content/uploads/2020/05/Jackal-Vehicle.png. Accessed 2 3 2022.

“EventBus: Events for Android - Open Source by greenrobot.” *Greenrobot*, https://greenrobot.org/eventbus/. Accessed 2 March 2022.

“Fragment.” *Android Developers*, https://developer.android.com/reference/android/app/Fragment. Accessed 2 March 2022.

“Fused Location Provider API.” *Google Developers*, https://developers.google.com/location-context/fused-location-provider/. Accessed 2 March 2022.

“How to contribute?” *ROS Mobile Android Wiki*, https://github.com/ROS-Mobile/ROS-Mobile-Android/wiki/How-to-contribute%3F#addOwnNodes.

“Inertiale Messeinheit – Wikipedia.” *Wikipedia*, https://de.wikipedia.org/wiki/Inertiale_Messeinheit. Accessed 28 February 2022.

“Jackal UGV - Small Weatherproof Robot - Clearpath.” *Clearpath Robotics*, https://clearpathrobotics.com/jackal-small-unmanned-ground-vehicle/. Accessed 2 March 2022.

“Model View ViewModel – Wikipedia.” *Wikipedia*, https://de.wikipedia.org/wiki/Model_View_ViewModel. Accessed 2 March 2022.

Moore, Tom. “Integrating GPS Data — robot_localization 2.7.3 documentation.” *ROS Documentation*, http://docs.ros.org/en/noetic/api/robot_localization/html/integrating_gps.html. Accessed 1 May 2022.

Moore, Tom, and D. Stouch. “robot_localization wiki — robot_localization 2.7.3 documentation.” *ROS Documentation*, http://docs.ros.org/en/noetic/api/robot_localization/html/index.html. Accessed 1 May 2022.

Rottmann, Nils, et al. “ROS-Mobile: An Android application for the Robot Operating System.” *arXiv:2011.02781*, 2020. *arxiv*, https://arxiv.org/abs/2011.02781.

Rottmann, Nils, and Nico Studt. “ROS-Mobile/ROS-Mobile-Android: Visualization and controlling application for Android.” *GitHub*, https://github.com/ROS-Mobile/ROS-Mobile-Android. Accessed 2 March 2022.

“Services overview.” *Android Developers*, 27 October 2021, https://developer.android.com/guide/components/services. Accessed 2 March 2022.

“Unbemanntes Landfahrzeug – Wikipedia.” *Wikipedia*, https://de.wikipedia.org/wiki/Unbemanntes_Landfahrzeug. Accessed 2 March 2022.