RUHR-UNIVERSITÄT BOCHUM

Autonomous Vehicles and Artificial Intelligence

Final Report

Group 35

Denis Meral	108019212469
Julian Claudius Willingmann	108022109291
Ayman Soultana	108019213176
Athithan Konoswaran	108019212326
Abinash Selvarajah	108019238460

Bochum, 1. March 2023

Contents

Chapter 1

Introduction

This is the final report for the course: Autonomous Vehicles and Artificial Intelligence in the WS22/23 by group ROS-ID-35. The goal of the course is to develop a software system for autonomously driving a ROS2 Turtlebot while utilizing multiple types of sensors and artificial intelligence. This report summarizes all assignments completed over the course of the project and documents the problems and challenges we faced, as well as the design decisions and implementation decisions we made. The full code and a demonstration video of our project can be found on our GitHub repository.
GitHub repository: `https://github.com/RUB-AVAI-22/allassignmens-35`

Chapter 2

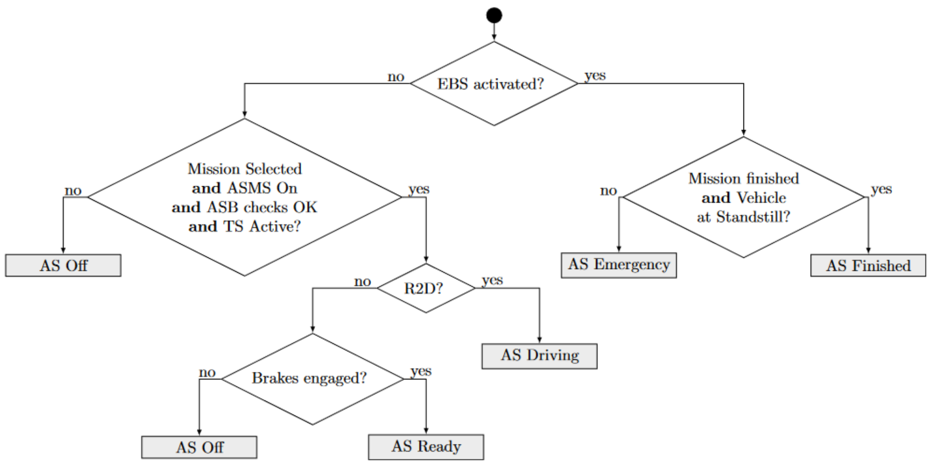
Requirements for an Autonomous Race Car

2.1 Requirements with WISE Drive Requirements Analysis Framework

An important aspect of the project was to extract requirements needed to ensure a successful execution of the autonomous driving with the turtle-bot. For this, we used the WISE Drive Requirements Analysis Framework[10]. We extracted the following functional and quality requirements and constraints we had to consider:

Functional requirements	• The ADS should ... ◦ recognize lighting level ◦ correctly identify yellow, blue, small orange and big orange cones[4, p. 12] ◦ correctly identify colored start and finish lines[4, p. 12] ◦ identify and localize threats[10, ADS-HLQR-Safety, Section 3a] ◦ assure minimum separation between the subject vehicle static objects ◦ consider multiple measures characterize separation (distance gaps, time gaps, time to collision, and lateral clearance) ◦ be able to monitor brake pressure[3, p. 63] ◦ be able to speed control, especially in avoidance of possible crashes ◦ be able to increase its vehicle speed ■ Parameters are the speed to be archived and the acceleration profile ■ At the beginning, the vehicle is set up for forward movement and at the end the vehicle should be close to the target speed ■ The task can be aborted at any time by switching to another longitudinal task ■ The duration depends on speed difference and acceleration level ◦ be able to decrease its vehicle speed ■ Parameters are the archive target speed and the declaration profile ■ At the beginning the vehicle is setup-up for forward movement. At the end the speed should be close to the target speed ■ the task can be aborted anytime by switching to another longitudinal task. ■ The duration depends on initial and target speed and deceleration level. ◦ be able to maintain the current speed ■ At the beginning the vehicle is set up for forward movement ■ At the end the speed should have been maintained throughout ■ The task can be aborted at any time ◦ be set up for forward driving, which includes starting the propulsion system and shifting into drive. ■ At the beginning, the vehicle stopped or was already moving forward ■ After the task, the vehicle set-up for forward movement ■ The duration is fixed to a few seconds, depending on the propulsion system • The ADS should include ... ◦ Primary and secondary maneuvers since it enables the completion of tactical driving tasks ◦ Vehicle stability: Skid and roll is required of the ADS for a vehicle to be controllable ◦ Stability recovery maneuvers (recover vehicle stability) ◦ Emergency braking maneuvers (avoid near crash by braking) ◦ Emergency steering and braking maneuvers (combination of the previous ones where we steer and brake) ◦ Car status data [4, p. 16, table 4] needs to be send to a Data-Logger and a remote emergency system every 100 ms[4, p. 13] ◦ Behavior modifiers that are related to the driving environment, mission, or the subject vehicle state

	• The ADS must at least implement the following missions[3, p. 66] ○ Acceleration ○ Skidpad ○ Autocross ○ Track drive ○ EBS test ○ Inspection ○ Manual driving • The ADS must have access to the car's shutdown circuit[3, p. 63] ○ only close it if in manual mission, and it has no access to the autonomous brakes ○ OR if mission is selected and sufficient brake pressure is built up • The autonomous brake system and the shutdown circuit must be continuously monitored[3, p. 68]
Quality requirements	• Paths outside of the blue cones on the left and the yellow cones on the right should not be considered by the ADS • Every mission should start in a start area and end in a finish area marked by orange lines and big orange cones • The ADS should consider limited lighting or limited camera vision for mission planning[10, ADS-HLQR-Safety, Section 3a] ○ Fog/mist/haze affects lighting conditions. It would have an impact on the reliability of the object detection system ○ Clouds affect background illumination from the sun or the moon and may cast shadows ■ It would have an impact on the reliability of the object detection system ■ Extra lights need to be activated, if lighting conditions are low • the ADS must be in one of the five following states at all time ○ ADS OFF ○ ADS Ready ○ ADS Driving ○ ADS Finished ○ ADS Emergency • The ADS is only allowed to access steering when in ADS Driving state[3, p. 64]

	• The state has to be determined similar to[3, p. 65]:  EBS activated? no yes Mission Selected and ASMS On and ASB checks OK and TS Active? no yes Mission finished and Vehicle at Standstill? no yes AS Off AS Emergency AS Finished R2D? no yes AS Driving Brakes engaged? no yes AS Off AS Ready • Minimum separation has to include sufficient safety margin to accommodate perception, prediction, and control uncertainties • Manual braking must always be possible[3, p. 67] • The ADS needs to keep the distance between the road structure and the subject vehicle[10, ADS-HLQR-Safety, Section 3a]
Constraints	• Rules from the Formula Student Germany • stopping distance (normal and in danger) • Assured Clear Distance Ahead (ACDA) is the path distance ahead of the vehicle that ensures that the ADS is able to drive and halt and is a minimum standard of care in driving[10, ADS-HLQR-Safety, Section 5, LOT98 law]

2.2 Task reflection

Writing the requirements helped us define the goal we want to reach at the end. It also gave us some ideas, which function we needed to implemented. Some requirements were not fulfillable, like testing humidity or wind speed. In the context of our project, Quality Requirements such as weather conditions (Rain, Fog) affecting the object detection were not of importance, while the indoor lightning conditions still played a role for us. Overall, it was still a bit difficult to extract the relevant requirements for us, since the sources were very focused on actual vehicles with ADS systems in larger racing settings.

Chapter 3

Implementing basic functions

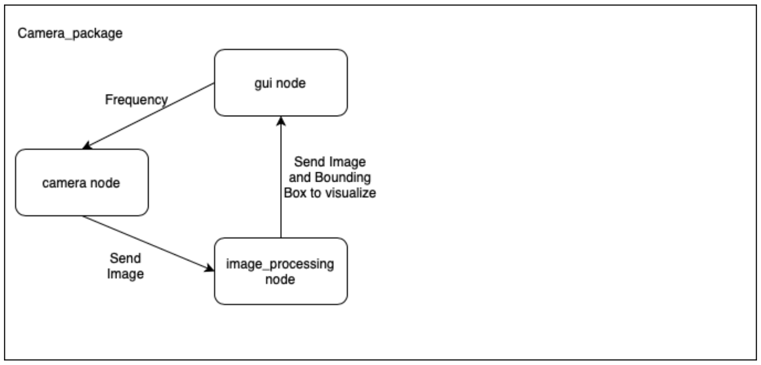


Figure 3.1: First node architecture

3.1 Camera node

The camera_node uses the OpenCV library[7] to capture video from a camera stream. The camera node has a publisher as an attribute to broadcast individual raw images of the video at an adjustable frequency with the topic "raw_image" of type sensor_msgs.msg.Image. To send the images, they must first be converted using the CvBridge library[2], as the OpenCV format is not directly compatible with ROS2. The frequency is implemented through a timer, which is destroyed and replaced with a new timer whenever the frequency is changed by the GUI. To receive requests for frequency changes from the GUI, the camera_node has a subscriber with the topic "set_frequency" of type std_msgs.msg.Float64 as an attribute.

Code snippet 3.1: set_timer function

```
def callback_set_frequency(self, msg):
    self.get_logger().info("Received new frequency")
    self.frequency_ = msg.data
    self.set_timer()

def set_timer(self):
    if self.publish_timer_ is not None:
        self.destroy_timer(self.publish_timer_)
        self.get_logger().info("Stopped timer")
    if self.frequency_ > 0:
        self.publish_timer_ = self.create_timer(1.0/self.frequency_,
            self.try_and_publish_image)
        self.get_logger().info("Set timer to frequency " + str(self.frequency_))
```

3.2 Image processing node

The image_processing node receives raw images from the camera_node via a subscriber for the topic "raw_image" and processes them. For this, the received data is first converted back into the OpenCV format using CvBridge. The "callback_raw_image" function converts the received images to OpenCV images, detects the cones using a pretrained YOLO model and saves the image together with x and y coordinates of the top-left corner, the width, and height of the bounding box, and the confidence score of the detection. After processing, the image is converted back into a deliverable format using CvBridge and then sent via a publisher attribute with the topic "processed_image" of type sensor_msgs.msg.Image.

Code snippet 3.2: Loading the YOLO model and detecting cones

```
def __init__(self):
    [...]
    self.interpreter = yolov5.models.common.DetectMultiBackend(MODEL_PATH,
        data=LABEL_PATH)
    self.interpreter = yolov5.models.common.AutoShape(self.interpreter)
def callback_raw_image(self, msg):
    [...]
    detect = self.interpreter(raw_image)
    # boxes contains information about the detected cones
    boxes = detect.pandas().xywhn[0].iloc[:,0:6].to_numpy().astype(np.float)
    # saving the information
    bounding_Boxes = []
    for i in range(0,len(boxes)):
        new_box = []
        for j in range(0,len(boxes[0])):
            new_box.append(boxes[i][j])
        bounding_Boxes.append(new_box)
```

3.3 GUI node

The GUI node is implemented together with the Qt-GUI[8] and is subscribed to the image_processing node to access the bounding box information and processed images. As the arrival of the bounding box information is always delayed, a TimeSynchronizer is used on both subscribers to synchronize them. If the synchronization succeeds, the drawBoundingBoxes function is called. The processed image is displayed on the GUI together with the corresponding bounding boxes.

Code snippet 3.3: Synchronized Messages

```
self.image_sub = message_filters.Subscriber(self,CompressedImage,"processed_image")
self.boundingbox_sub = message_filters.Subscriber(self,FloatArray,"bounding_box")
ts = message_filters.TimeSynchronizer([self.boundingbox_sub, self.image_sub], 10)
```

Each received image will be saved and numbered, which allows us to refer to each image at a specific time. The mapped cone points published by the occupancy map node are drawn on the map.

Code snippet 3.4: Drawing map on the GUI

```
def update_plot(self, msg):
    [...]
    #colors = ['blue', 'orange', 'yellow', 'black']
    [...]
    colors = []
    for classes in dfClasses:
        if classes == 0:
            colors.append("blue")
        if classes == 1:
            colors.append("orange")
        if classes == 2:
            colors.append("#FFD700")
        if classes == 3:
            colors.append("black")
    [...]
    ax.scatter(dfX, dfY, c=colors)
```

To initialize the Qt GUI and the corresponding ROS node, we start and execute it inside a separate thread using a MultiThreadedExecutor. Both objects can access each other using attributes which are set during the initialization.

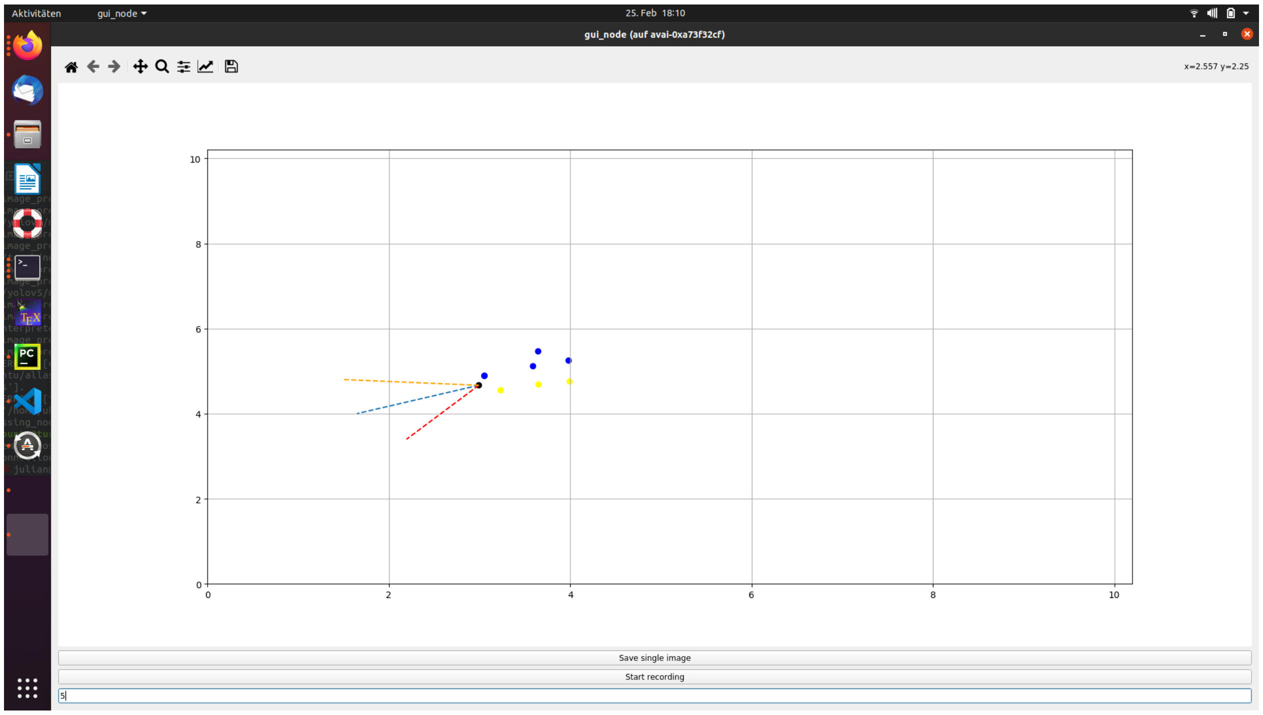


Figure 3.2: GUI Node with occupancy map

3.4 Remote Movement

3.4.1 Twist objects

To obtain data and commands for the movement actuators in the turtlebot, it is best to use the cmd_vel topic. The topic is part of the turtlebot software, and a subscriber for it is initialized automatically at bring_up. A publisher to cmd_vel needs to make sure to use the default quality of service setting and Twist objects as messages. Twist objects are part of the geometric messages. They have a linear and an angular field, which both require a three element vector to set. Alternatively, a single element can be set individually by referencing the x, y or z component of the vector directly. There are two important parts of the Twist objects for the movement actuator: the linear.x value, which is used to set linear velocity, and the angular.z value, which is used to change angular velocity.

3.4.2 Key Events

To properly control the movement of the turtlebot it was necessary to use Key Events, so that the turtlebot only moves whenever a key is held. For that reason, we used the Pynput library to implement two functions that will help us solve this problem. The first function is called "on_press" and it will be called when the user presses a button. We can use If-statements to ask for certain button presses such as 'W' or 'S' and then create Twist objects accordingly.

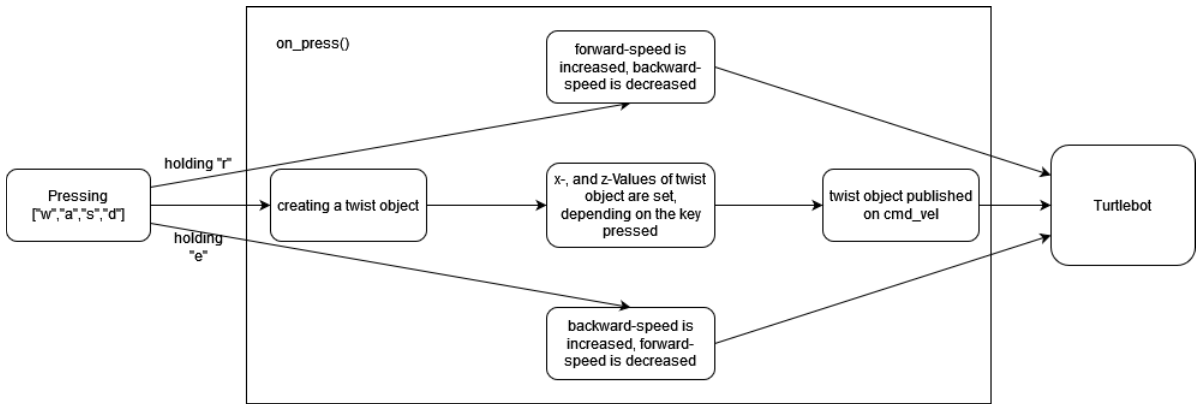

on_press()
Pressing ["w","a","s","d"]
holding "r"
holding "e"
creating a twist object
forward-speed is increased, backward-speed is decreased
x-, and z-Values of twist object are set, depending on the key pressed
backward-speed is increased, forward-speed is decreased
twist object published on cmd_vel
Turtlebot

Figure 3.3: on_press function

We also need to implement the function “on_release”. It will be used to reset the linear and angular field in the cmd_vel topic, so that the turtlebot stops when no buttons are pressed. We can do that by publishing a Twist object with a value of zero for both linear.x and angular.z.

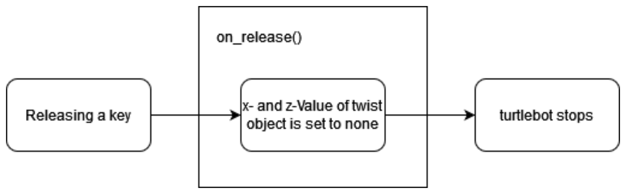


Figure 3.4: on_release function

3.5 Task reflection

Programming with ROS2 was a new experience for us. Despite the short introduction to ROS2, we learned a lot from the many examples showed to us. We also consulted YouTube and Udemy anytime we encountered some issues. But there were many problems left. For example, we could not access the camera feed because of connectivity errors. Because of that, we used a simple .jpg file instead. Besides implementation problems, there were some technical issues, such as low performance with VirtualBox, which was improved by downgrading the resolution. One of our team members couldn’t run the virtual machine at all, since he used an Apple M1 Laptop, which was not supported by VirtualBox.

Chapter 4

Object Detection

Object detection is used in the image_processing node, as it enables the turtlebot to understand and interact with its environment. We used the popular object detection algorithm YOLO[11] (You Only Look Once), which makes the real-time detection of cones possible.

4.1 Training

The training was hosted on Google Colab[5], a GPU cloud, because it is free and enables the training on laptops. The dataset was imported to roboflow[9], which made it easy to split and upload them directly to the Google Colab notebook. The dataset contained 1300 images from which 70 percent was used for training and the rest for testing. Furthermore, the images were divided into batches of 16 images. After training, the model tensorboard was used to visualize the results of our model.

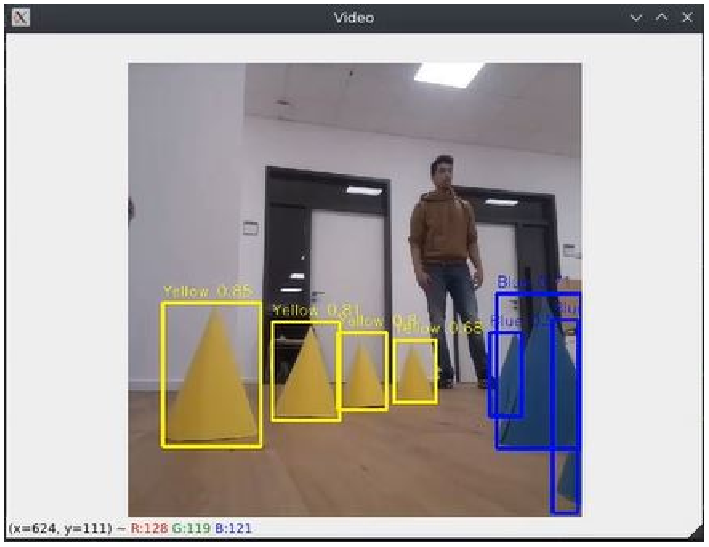


Figure 4.1: Cone Detection

4.2 Metrics

The term with utmost importance is the Mean Average Precision or for short mAP. The mAP compares the ground-truth bounding boxes to the detected bounding boxes and returns a score. This is achieved by calculating the IoU (Intersection over Union). It calculates the percentage of the bounding box overlap. Thus, it is possible to use it to calculate how similar the ground-truth bounding boxes are compared to the detected ones.

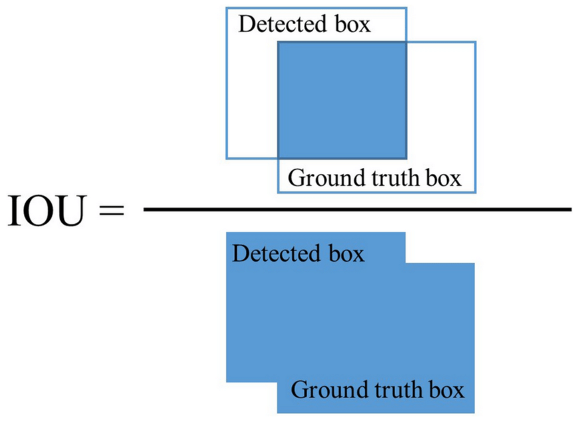


Figure 4.2: Calculation of IoU[6]

Another term that is important for our metric is recall. Recall describes how often the object was actually recognized. The recall is calculated by dividing the true positives with the overall sum of recognitions. A true positive is the amount of times the object was correctly recognized. The overall sum of recognitions is just the sum of the true positives and the false negatives. False negatives are the amount of times the object was not recognized when it was in the image.

$$\text{Recall} = \frac{\textit{True Positive}}{\textit{True Positive} + \textit{False Negative}}$$

4.3 Results

As mentioned, tensorboard is used to visualize the results of the model. Tensorboard is a framework which provides measurements and visualizations during the machine learning workflow. It is therefore possible to see the process of our training in real-time and the main metrics that are used in this project such as mAP and recall are also available to be reviewed.

4.4 Implementation

For Interference, we aimed to use YOLOv5s algorithms while still utilizing the performance boost provided by the Edge TPU. To achieve this, we used YOLOv5s DetectMultiBackend function. It

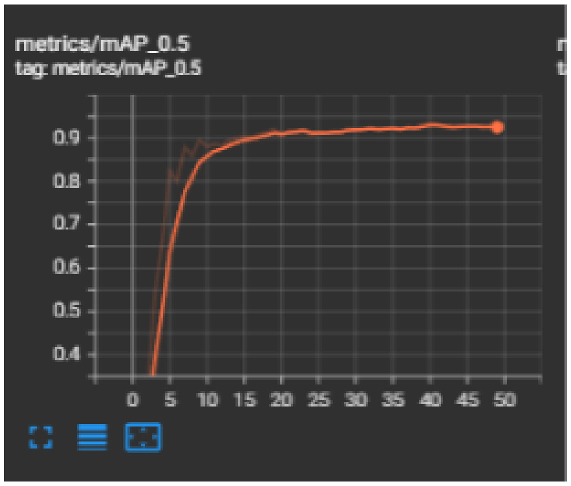


Figure 4.3: mAP results

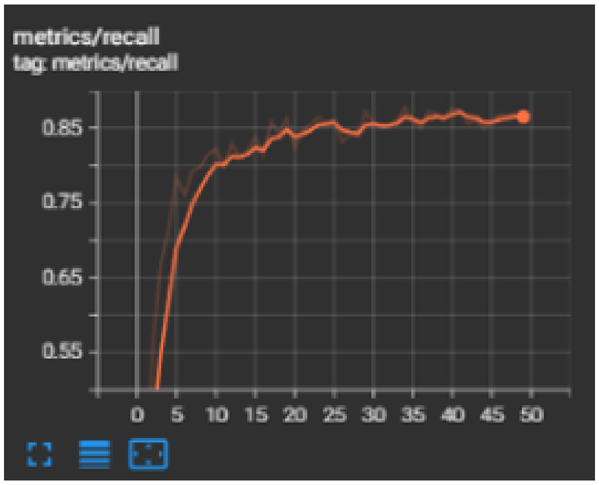


Figure 4.4: recall results

is able to create YOLO models from different weight input formats, including the edgeTPU.tflite format needed for TPU usage. Next we use Autoshape to configure the model (e.g., setting detection threshold) and as a wrapper, so we can use the cv2 NumPy arrays as input into our model. For the actual interference, we use the YOLOv5 build in detection by calling the model with an image inside our callback function.

4.5 Task reflection

Training with the free tier of the Google Colab cloud did come with some disadvantage. There was an anti-afk-check every 30 minutes, that will restart the session and delete all data and progress, if we fail to click a popup in time. Uploading images to Google Colab was also problematic, since you can not load all the images at the same time and sometimes the upload even fails. Here we used roboflow as a solution to this problem.
There was also an under-/overfitting issue, which is caused by an imbalance between training and validation dataset. This results in an altered perception in rare cases for the turtlebot, as he recognized some chairs as yellow cones. We needed to create a more diverse/robust dataset to solve this issue, but this would consume a lot of time.
Conflicting PyTorch and Torchvision versions also caused some issues, but were fixed after some trial-and-error.
We could have done some testing at home with the turtlebot to compare our different TPU models, but as the Coral simulation did not work properly, we could not do that.

Chapter 5

Lidar – Sensor fusion

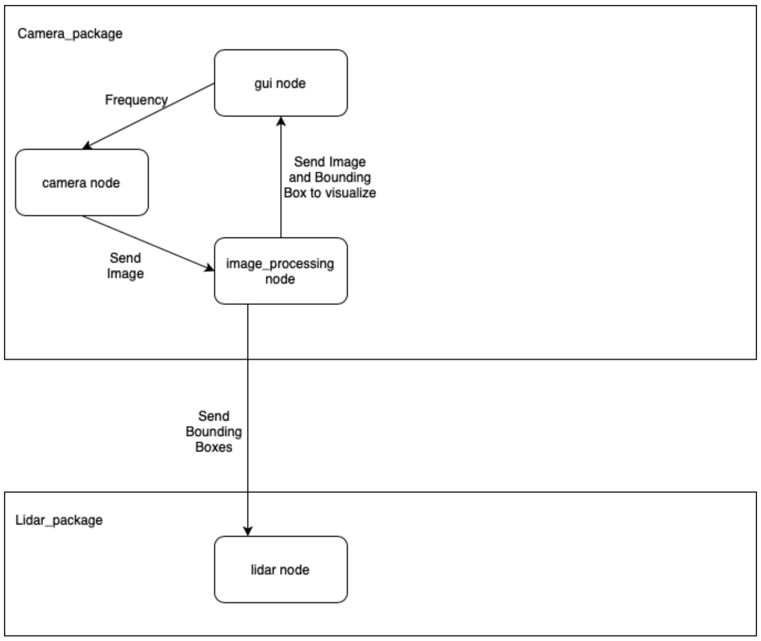


Figure 5.1: Lidar added to node architecture

5.1 Lidar

The Lidar node is our approach at a sensor-fusion between the light detection and ranging sensor (Lidar) and the detected cones from the camera. The node is subscribed to both the image_processing node, to receive bounding box data, and the "scan" Topic, that provides Lidar data. Both subscribers are synchronized to a 0.1 second delay, which is needed to account for turtlebot movement and the frequency difference between both publishers. Our Implementation can be split in two separate parts. The first part prepares the Lidar data and the second part performs the actual fusion.

5.1.1 Lidar-Data preparation

The main task of this step is to find actual objects in the Lidar range data that gets sent via the scan topic. This is done by a simple clustering algorithm. The main idea is to find sequences in the range array that differ only by a small error of 10 percent of the distance, and group them together in a cluster. Whenever the range difference is bigger than the error, a new cluster starts. The error is needed, because of measuring errors and the cone-shape of the objects we want to detect. In a final step, a mean distance is calculated for every cluster.
During filtering, single range points in groups of zeros get removed, as they happen mainly due to sensor errors. We also remove points that are detected too far away to increase the fusion accuracy.

5.1.2 Sensor fusion

During the actual fusion, we try to map the angles of the clusters onto the x-Values of the bounding boxes. In a first attempt we tried to achieve this with a simple linear mapping where we map our 0 to 55 degree angle space into the 0 to 1 Space of x-Values. This approach worked fine for cones in the very center of the pictures, but had problems on both edges. But due to a higher need of correctly detected cones on both edges to detecting both track borders at the same time, we changed the mapping to two different linear mappings for each side that are optimized for objects on the picture edge.
The actual algorithm is pretty simple as we just run through all bounding boxes and all clusters, calculate the linear map of the cluster angle and if it is within a small error margin of the bounding box's x value, we append the cluster and the bounding box class to an array of all fused objects, which is then send to the occupancy map.

5.2 Task Reflection

This was the first task where a node relied on two different sensor inputs, so a big difficulty was to find a good way of synchronizing both inputs. Another, even the bigger difficulty, was fine-tuning all parameters, that were used in the fusion (e.g., finding a field of view that represents the part of the camera's field of view that is correctly detecting cones). Our findings in this task, especially time-synchronisation and the correct usage of sensor data and messages, helped a lot in the later tasks.

Chapter 6

Localization and Mapping on the Racetrack

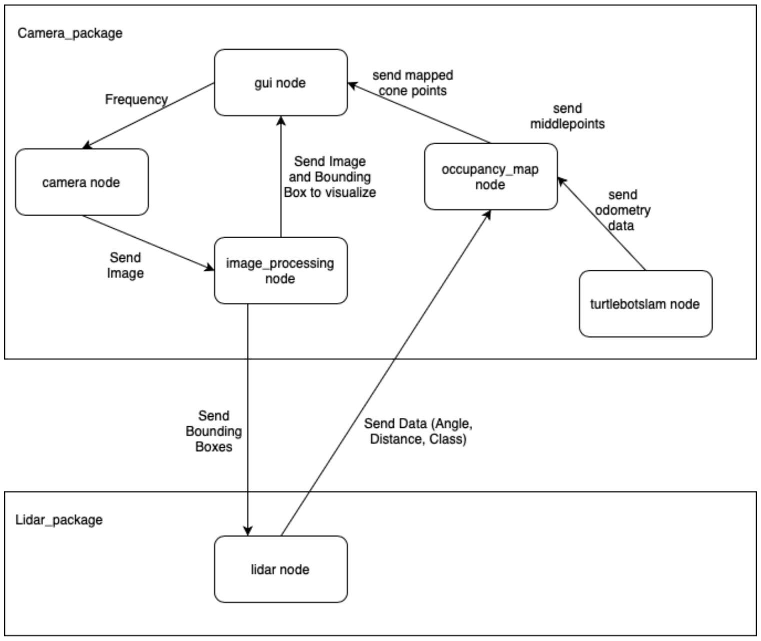


Figure 6.1: Occupancy map and turtlebotslam added to node architecture

6.1 Occupancy Map

An occupancy map is a representation of the environment. It indicates areas occupied by objects and areas that are free of obstacles. In case of this project, the map is represented as a list of tuples which contains the positions and class IDs of the detected cones. The positions of the cones are calculated using data from the Lidar sensor. To determine the x and y position of cones, detected by the Lidar sensor-fusion, some trigonometric calculations are required.

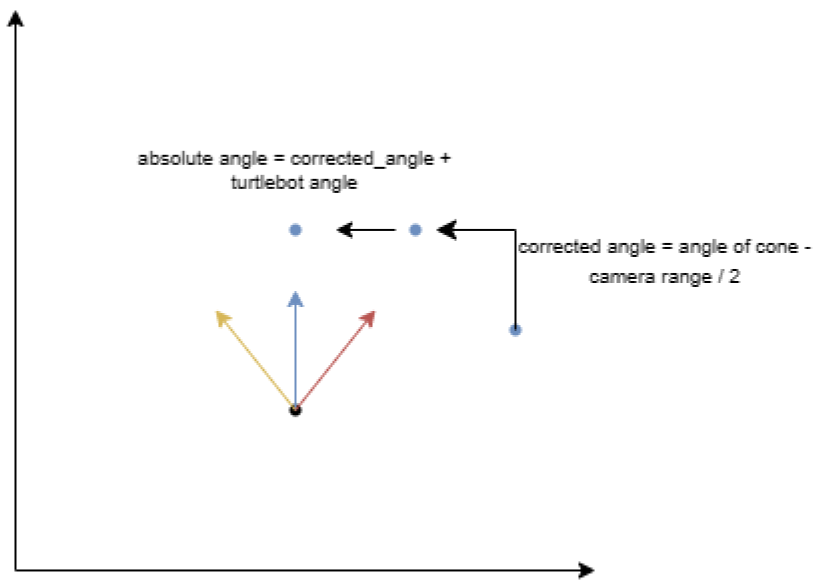

absolute angle = corrected_angle + turtlebot angle
corrected angle = angle of cone - camera range / 2

Figure 6.2: Calculation of the absolute angle for a cone

To determine the absolute angle of an object, we first need to subtract the angle of the cone by half of the camera range. This is necessary since the Lidar angles are mapped between 0 and 55, so if for example a cone has an angle of 28 degrees, it should be in the middle of the turtlebots view. Afterward, we calculate the absolute angle by adding the turtlebot's angle to the relative angle. Next, we calculate the x-coordinate using the "math.cos" function, which takes an angle in radians as input and returns the cosine of that angle. In this case, the "absolute_angle" variable (Angle of the cone on the map) is used as input for the "math.cos" function. By multiplying the cosine with the distance, we get the x-coordinate of the Lidar point relative to the turtlebot. Finally, we add the x-coordinate of the turtlebot to get the absolute x-coordinate of the Lidar point. We do the same calculation to get the y-coordinate of the Lidar point, except that instead of "math.cos" we use "math.sin".

6.2 Clustering

Clustering is a crucial step in generating an accurate occupancy map, since detecting cones multiple times and placing them in almost the same position on the map can lead to issues, particularly when using the occupancy map to calculate the turtlebot's track. To address this problem, we utilize the DBScan clustering algorithm to group cones that are in close proximity to each other. DBScan is a density-based clustering algorithm that uses the distance between the nearest points to form clusters. By leveraging this approach, we can more effectively group cones that are located near each other, which in turn helps to improve the accuracy and reliability of the occupancy map.

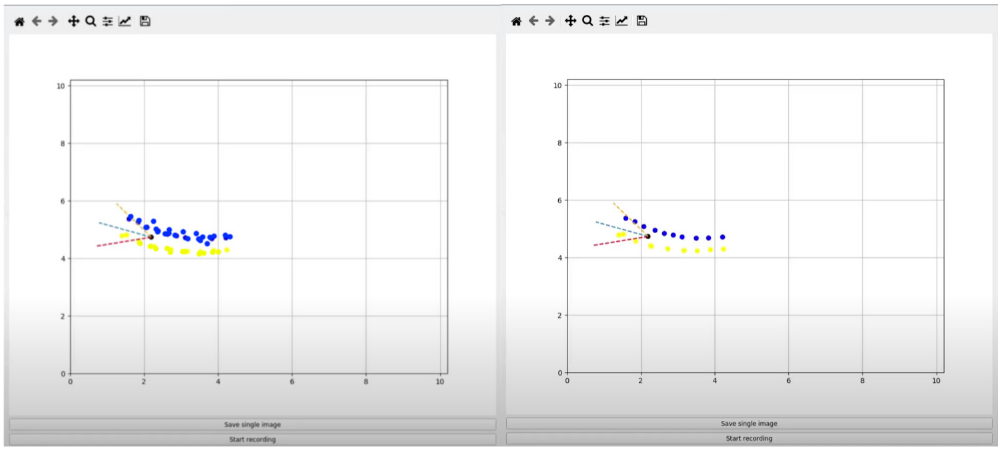


Figure 6.3: Without clustering and with clustering

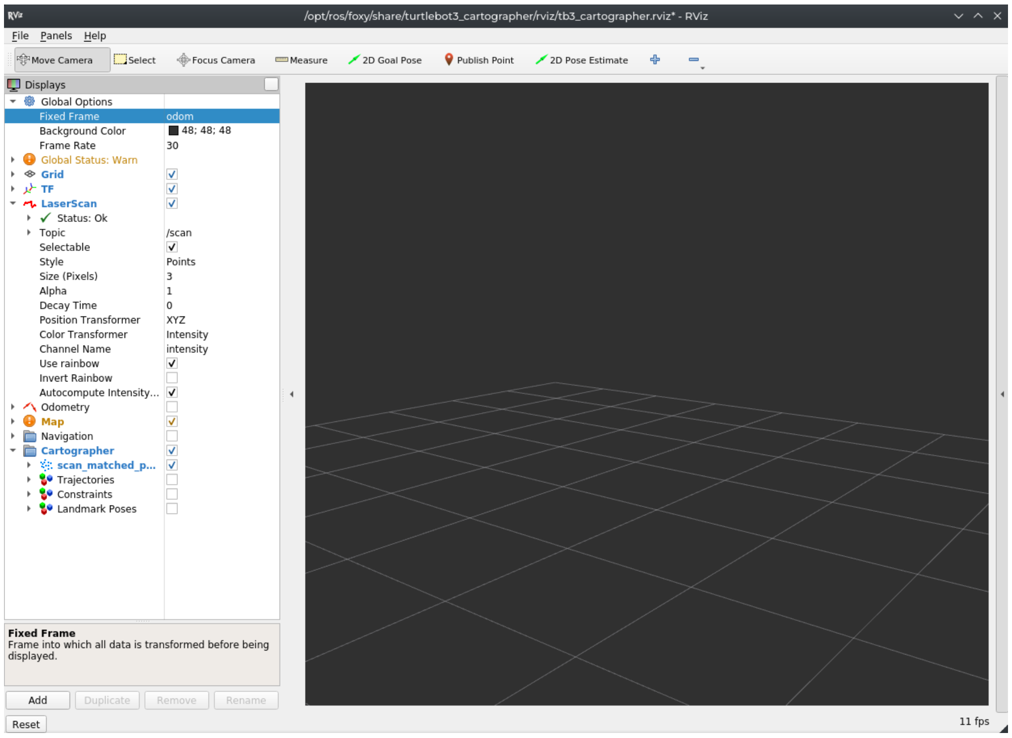
Our clustering approach is designed to group cones of the same color class (e.g., blue, orange, yellow). While this would typically suffice for our needs, we encountered the problem that our model occasionally detects a blue cone as a yellow cone, or vice versa. As a result, a yellow cone may be placed on top of a blue cone or vice versa, which can lead to inaccurate occupancy maps. To overcome this issue, we developed a filtering algorithm that removes cones that are in close proximity to each other but belong to different color classes. This ensures that wrongly detected cones are filtered out, which helps to maintain the integrity of the occupancy map and ensures accurate tracking of the turtlebot. By utilizing this filtering algorithm in our clustering process, we can more effectively generate reliable and precise occupancy maps.

Code snippet 6.1: Filtering wrongly detected blue and yellow cones

```
# remove wrongly detected yellow and blue cones
points_to_filter = []
new_positions = [self.lidar_to_xy(new_lidar) for new_lidar in positions]
for new_pos in new_positions:
    for point in self.map:
        if (point[2] == 0 and new_pos[2] == 2) or (point[2] == 2 and new_pos[2] == 0):
        # check only different colors
            distance = math.sqrt((new_pos[0]-point[0])**2+(new_pos[1]-point[1])**2)
            if distance < 0.2:
            points_to_filter.append(new_pos)
            points_to_filter.append(point)
                break
points = self.map
points += new_positions
for point in points_to_filter:
    if point in points:
        points.remove(point)
```

6.3 SLAM

For localizing the turtlebot and simultaneously drawing the map, we used the SLAM method. In the case of our turtlebot, SLAM algorithms are used to navigate the robot through unknown environments while creating a map of the environment as the robot moves. This is done using the Lidar sensor and the camera. The robot's location and the map of the environment are updated in real-time as the robot moves and gathers new data. To achieve this, the odometry of the turtlebot is used. By subscribing to the "odom" topic, it is possible to obtain the pose of the robot, which can be used to update the turtlebot's position on the map. Furthermore, the angle of the turtlebot can be updated in the same way by using the odometry. Unfortunately, the odometry is quite unstable so that the positions of the turtlebot and the cones are severely influenced to the point that the resulting occupancy map is not a correct representation of the environment anymore. In order to acquire a more stable odometry, we use the transformations that are published by the cartographer node. The cartographer node is a built-in SLAM node that also publishes transformation of incoming sensor data.



Displays

Global Options	
Fixed Frame	odom
Background Color	■ 48; 48; 48
Frame Rate	30
Global Status: Warn	
Grid	✓
TF	✓
LaserScan	✓
Status: Ok	
Topic	/scan
Selectable	✓
Style	Points
Size (Pixels)	3
Alpha	1
Decay Time	0
Position Transformer	XYZ
Color Transformer	Intensity
Channel Name	intensity
Use rainbow	✓
Invert Rainbow	☐
Autocompute Intensity...	✓
Odometry	☐
Map	✓
Navigation	☐
Cartographer	✓
scan_matched_p...	✓
Trajectories	☐
Constraints	☐
Landmark Poses	☐

Fixed Frame
Frame into which all data is transformed before being displayed.

Figure 6.4: Cartographer

In order to obtain accurate odometry for our turtlebot, we rely on transformations from the cartographer node. Specifically, we use the transformation from "odom" to "base_footprint". To make these transformations available to the occupancy map node, we have created a new node that leverages the tf2 library. To achieve this, we create a tf buffer that contains the transformations published by the cartographer node. This allows us to easily search for and obtain specific transformations, such as odom $\rightarrow$ base_footprint.

We can then use these transformations for our odometry calculations. This approach provides a more reliable and stable means of obtaining accurate odometry data for the turtlebot.

Code snippet 6.2: transformation

```
trans = self.tf_buffer.lookup_transform('odom', 'base_footprint',
    rclpy.time.Time(),timeout=rclpy.duration.Duration(seconds=1.0))
pose = TransformStamped()
pose.header.frame_id = 'map'
pose.header.stamp = self.current_header.stamp
pose.child_frame_id = 'base_link'

pose.transform.translation.x = trans.transform.translation.x
pose.transform.translation.y = trans.transform.translation.y
pose.transform.translation.z = trans.transform.translation.z

pose.transform.rotation.x = trans.transform.rotation.x
pose.transform.rotation.y = trans.transform.rotation.y
pose.transform.rotation.z = trans.transform.rotation.z
pose.transform.rotation.w = trans.transform.rotation.w
self.publisher.publish(pose)
```

6.4 Task Reflection

Creating the occupancy map was by far the most challenging task in our project. Our main challenge was accurately determining the position of each cone relative to the turtlebot's rotation and camera range, which required extensive trial and error. The odometry being unstable also made debugging even more complicated. We had to rely on the cartographer to provide us with stable odometry data, as without it, we would not have been able to generate an accurate occupancy map.

Chapter 7

Autonomous Movement

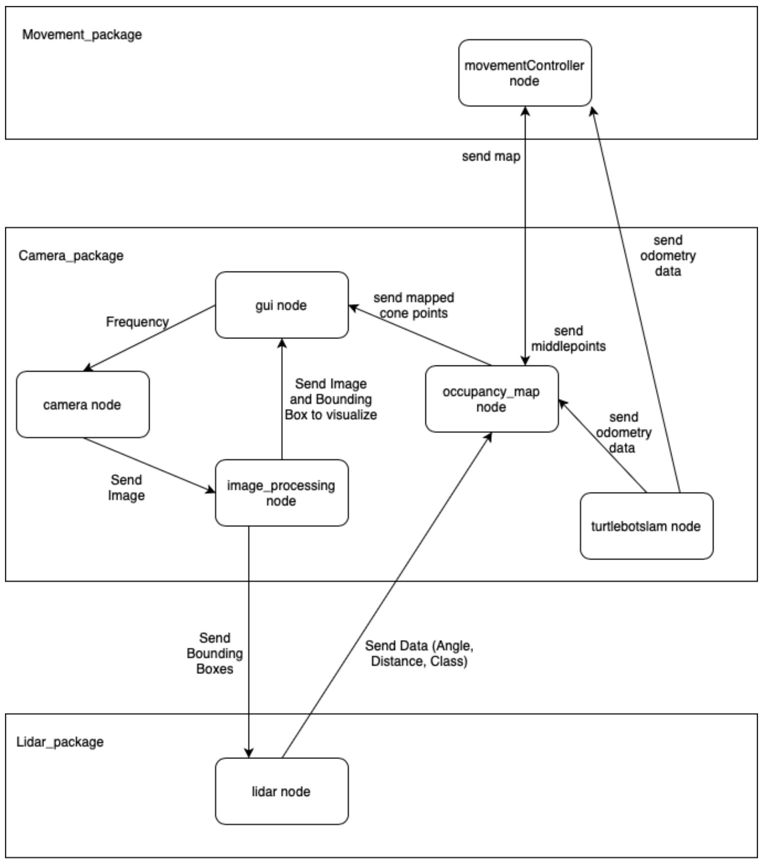


Figure 7.1: MovementController added to node architecture

7.1 Movement Controller

The movement controller is used to calculate the correct motor input depending on the current robot position and position of the next target point. This is done on every odometry message. The most important part is the calculation of the angular velocity. For an accurate calculation, we need two angles. First the angle between the turtlebot position and the target (marked as greenθ) can be calculated using the arctan of the vector between both points, and secondly the relative orientation marked as the red θ of the turtlebot, which is encoded in the pose quaternion the odometry message provides. The lager the difference between both angles, the faster the angular velocity needs to be to reach the target. If it is zero, we do not need any angular velocity. For the lateral part of the movement, we are using a constant speed. We tested with a linear speed, relative to the distance to the next target. But it just caused the turtlebot to slow down significantly when it was close to the current target point.

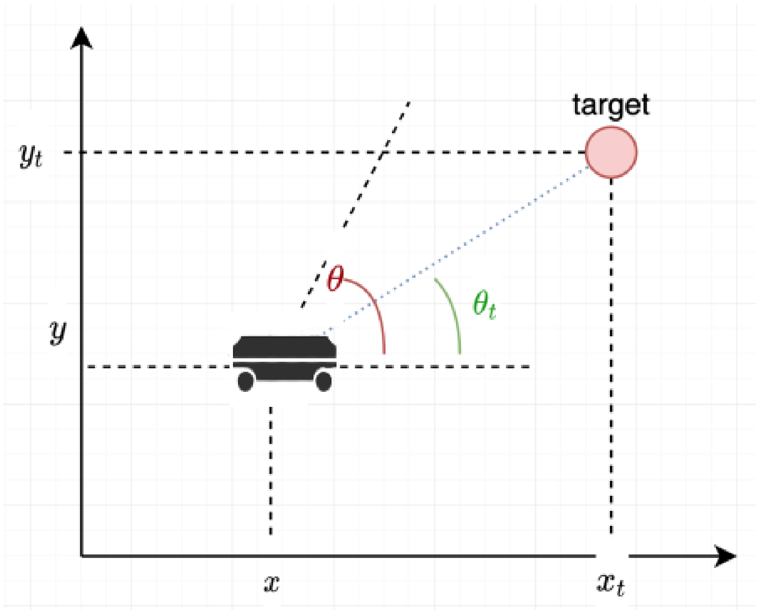


Figure 7.2: Angular velocity calculation[1]

7.2 Calculation of next target point

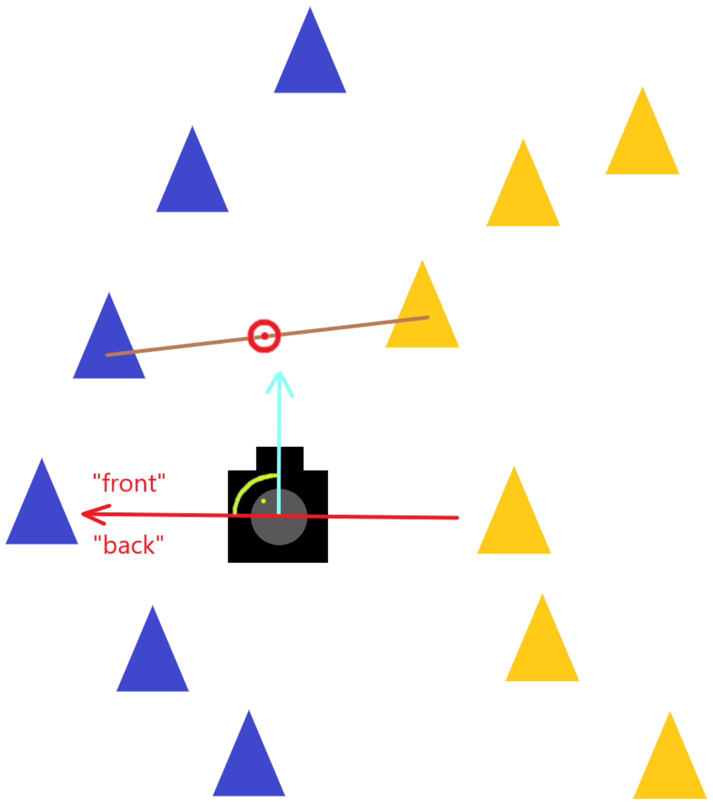


Figure 7.3: Sketch of the calculation of the next target point

The next target point is calculated based on the points, that are currently saved on the occupancy map. First, we need to only get the points, which are in front of the turtlebot. For that, we take the current angle of the turtlebot (light blue line) and add 90 degrees to it (green angle). Then we can calculate a vector in this direction (red line) and decide if a point is in front of the turtlebot by calculating on which side of the vector the point is. We then save only the points which are in front of the turtlebot. Next we take out the blue and yellow cone which have the smallest distance to the turtlebot and use linear interpolation to calculate the middle point between these two cones. This new point is the next target point towards which the turtlebot will drive.

7.3 Task reflection

Similar to the Lidar task, we had to manually adjust numerous parameters to achieve smoother movement of the robot. We had to test different parameters in a trial-and-error manner, until we found the right ones. Adjusting the calculation of the middle points accordingly was also necessary, to avoid any inaccuracies in the movement.

Chapter 8

Testing

8.1 Testing Strategy

Testing is a crucial part of the project. We conceived the following testing strategy beforehand: At the first testing level unit-tests should be used to assure correct behavior of methods and single nodes without considering inter-node communication, but with regard to edge cases and error exceptions. On the next unit-testing level, inter-node communication should be tested. The focus of this test it to assure that all nodes are correctly connected, receive all expected messages, and behave accordingly when receiving erroneous messages.

At the first integration testing level, the tests should assure the correctness of more complex tasks such as the detection of objects (e.g., cones). This shall be achieved by a coordinated series of individual tests. At the second integration testing level, the testing will observe even bigger tasks, such as driving a horizontal curve. Here, we will check how the different components of the ADS work together to process tasks. In general, we will test the components of the ADS to ensure that no defects are available. The third integration testing level includes the safety protocols, such as emergency breaking or other crash avoidance maneuvers. In this level, we will observe and check if the ADS is reacting correctly to crash situations.

After the integration tests, we continue with the acceptance tests. Here we will test whole missions like skip-pad with regard to all quality requirements and safety features tested before.

8.2 Testing

After we had conceived a viable testing strategy, we continued with the actual testing. For the implementation of our test suite, we used the python unit-test package. It provides a test case class and methods to correctly setup and teardown the test environment, as well as methods to decide whether a test was successful or not.

8.2.1 Unit-Test

In our unit-tests we did white box testing of all methods inside our nodes. That means we implemented the tests with knowledge of the source code and in a way to achieve high code coverage. E.g. in the unit-tests for the remote control all the different IF-ELIF branches get tested. For the camera node multiple functions were tested, such as the "try_and_publish_image" function. It was possible to access certain functions and variables in the node by importing the node to the script and then creating an instance of it.

By using the launch_test command in the terminal, it was possible to call all the tests in the test script for a certain node. To make this possible, the "generate_test_description" method was implemented. With it, ROS2 parameters could also be set, which was helpful when the parameters were needed. The functions that are meant to test certain functionalities were then implemented in a test class that took the unit-test case as input and so was the testing of the functions in a node executed.

Code snippet 8.1: Example of a unit-test

```
def test_frequency_callback(self,camera,proc_output):
    msg = Float64()
    msg.data = 2.0
    self.camera_node.callback_set_frequency(msg)
    self.assertEqual(msg.data, self.camera_node.frequency_)
```

8.2.2 Integration

The integration tests are done as black box tests, without knowledge of the actual implementation of the Node, which in most cases means that we concentrate only on getting the correct output for certain inputs into the node, without really caring about what gets called or what code gets executed. At the moment our unit- and integration tests are quite similar as our nodes only execute a special task and a lot of methods do not return a value but rather directly publish their outcome. A refactor of our current nodes into more separated methods could lead to more unit-tests and high separation. An example for one of our integration tests is the Image_processing node test. An example image gets sent over the correct topic ('compressed image') to the node and the test checks whether an image with the correct bounding boxes gets published by the node.

Code snippet 8.2: Example of a Integration test: Image_processing

```
def test_image_processing_transmits(self, image_processing, proc_output):
    msgs_rx = []
    pub = self.node.create_publisher(Image, "raw_image", 10)
    sub = self.node.create_subscription(
        CompressedImage,
        "processed_image",
        lambda msg: msgs_rx.append(msg),
        10
    )
    try:
        msg = Image()
        raw_image =
            cv2.imread("/home/ubuntu/allassignmens-35/src/camera_pkg/test/ManualImage25.png")
        cv_bridge_ = CvBridge()
        msg = cv_bridge_.cv2_to_imgmsg(raw_image)
        time.sleep(10)
        pub.publish(msg)
        #Wait until the talker transmits two messages over the ROS topic
        end_time = time.time() + 10
        while time.time() < end_time:
            rclpy.spin_once(self.node, timeout_sec=0.1)
            if len(msgs_rx) > 2:
                break
        self.assertEqual(len(msgs_rx), 1)
    finally:
        self.node.destroy_subscription(sub)
        self.node.destroy_publisher(pub)
```

8.2.3 Acceptance

Acceptance tests evaluate the compliance of the ADS with the safety and business related quality requirements. The ADS will pass the acceptance tests, if it can automatically react to and avoid hazards (functional). The ADS also needs to monitor and log its own state and behavior, so that safety can be better ensured (functional). Besides that the ADS must perform well under varying environmental conditions (quality) and it also must actively adjust its own behavior. For example, the ADS has to slow down in certain conditions, so that crash situations can be avoided.

Functional: The ADS can correctly identify yellow, blue and orange cones. Even if driven manually, it does not yet detect or avoid crashes. Furthermore, the ADS does not recognize environmental conditions like lighting level or wind speed, which, as previously mentioned, are not needed for our project setting. It also does not monitor or log data concerning its own state, like its acceleration, speed, current action, etc. The brakes of the ADS are manual, that means the turtlebot has to be manually stopped after it drove autonomously through a racetrack.

Quality: Fog, mist and haze do affect the object detection system's reliability, as the training data did not feature such conditions. The training data was captured in a controlled indoor environment. Because of that, the ADS does perform well if placed in such an environment. The ADS does not feature special maneuvers to avoid crashes yet, nor does it have a maneuver to ensure vehicle stability.

8.3 Task Reflection

At first, we had problems writing tests for the GUI node, because we did not know how to access the current node from the tests. But as the GUI is tested in a similar fashion, as the other nodes, implementing more tests was easy as soon as we figured out how to access the current node.

The CARLA simulator is also a possible tool to run tests. In CARLA, it is possible to load custom maps and spawn cars. Furthermore, it is possible to decorate these cars with sensors or actors as needed and then use the corresponding topics as input into our ROS2 nodes to simulate the actual hardware.

For us, it was not possible to perform tests on the CARLA simulator, without a less complicated map or better hardware. Crashes caused timeouts and performance problems.

Chapter 9

Problems and Challenges

After our time with this project and the illustration of our final Results, we also wanted to reflect on some ongoing challenges and problems in our ADS.

Apple Silicon	Virtual Box don't support Apple Silicon (as of February 2023), so that people who use newer Apple MacBooks, can't run the VM at all.
CARLA simulation	CARLA simulation doesn't work in the Virtual Box due to restricted VRAM, but works on computers, which use Linux natively. The simulation would be helpful in testing the code at home.
Leftover Requirements	Some requirements we had still been not fulfilled after our time on the project, for example autonomous braking after finishing a racetrack or implementing reliable speed control. Considering this challenge, it would've helped to have more time to have a closer look at these requirements. But an Important part is also to have better hardware to deal with these requirements.
Odometry	The Odometry drifted a lot, which influenced the quality of the occupancy. Cartographer node was needed to solve this problem, which took a lot of time. Furthermore, sometimes the odometry drifts so hard that even the transformations are influenced by it.
Setuptools update	After updating setuptools, pip didn't work properly, which was resolved after a hard reset of the turtlebot. This consumed a lot of time, as we needed to install the libraries and setup the turtlebot again.

Turtlebot Moments	The turtlebot has a lot of issues. Sometimes it just does not connect, or it just crashes randomly out of nowhere. This makes the developing process harder, since a lot of time is wasted on rebooting the turtlebot.
Turtlebot computational power	Even with the use of hardware accelerators like Edge TPUs, there is still a considerable delay between an image capture and a send boundingbox message. This delay caused Problems, especially with slam and movement tasks

Bibliography

[1] *angular velocity picture*. URL: https://robotics.stackexchange.com/questions/21687/determining-heading-yaw-to-rotate-the-turtle-robot-to-desired-point-x-y-so-i.

[2] *CvBridge*. URL: http://wiki.ros.org/cv_bridge.

[3] *FS-Rules*. URL: https://www.formulastudent.de/fileadmin/user_upload/all/2023/rules/FS-Rules_2023_v1.0.pdf.

[4] *FSG22 Competition Handbook*. URL: https://www.formulastudent.de/pr/news/details/article/fsg2022-competition-handbook-v10-is-online/.

[5] *Google Colab*. URL: https://colab.research.google.com/.

[6] *IoC-Picture*. URL: https://www.researchgate.net/figure/Illustration-of-intersection-over-union-IOU_fig5_346512249.

[7] *OpenCV*. URL: https://opencv.org/releases/.

[8] *Qt-GUI*. URL: https://www.qt.io/product/framework.

[9] *Roboflow*. URL: https://roboflow.com.

[10] *WISE Drive - Requirements Analysis Framework for Automated Driving Systems*. URL: https://uwaterloo.ca/waterloo-intelligent-systems-engineering-lab/projects/wise-drive-requirements-analysis-framework-automated-driving.

[11] *YOLO*. URL: https://github.com/AlexeyAB/darknet.