Ruhr-Universität Bochum
Autonomous Vehicles and Artificial Intelligence
Summer Term 2022

Final Report

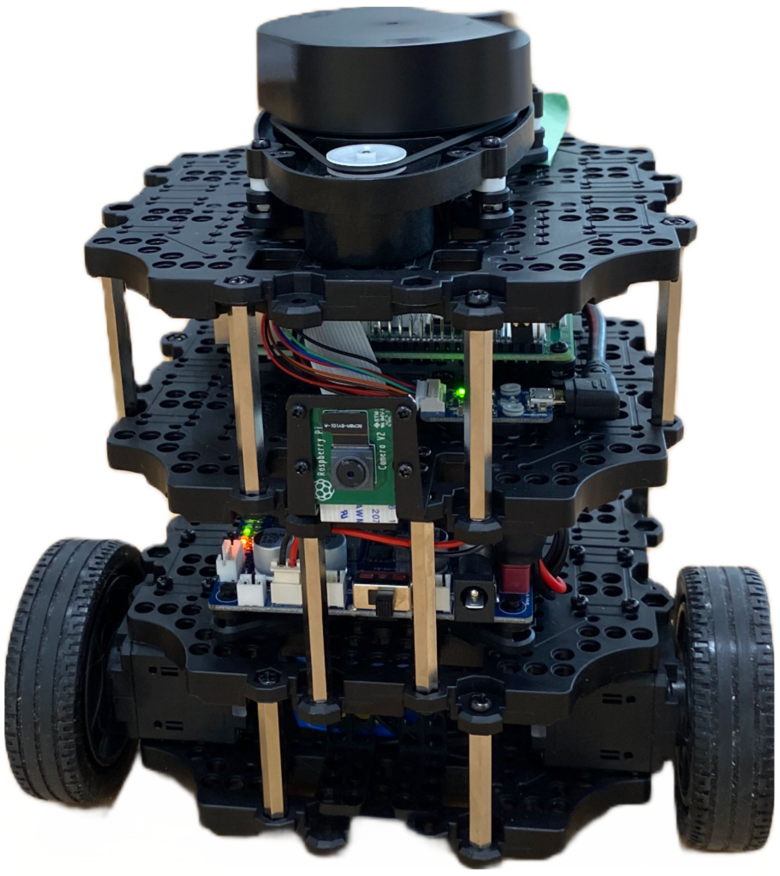


Table of Contents

Introduction

This report documents the design decisions, problems and challenges that we were facing during the practical sessions of the Autonomous Vehicles and Artificial Intelligence course. The chapters in the report follow the order of the assignments. In the final chapter (Assignment 5: Autonomous driving) the steps of making the turtlebot drive autonomously on a track are described. This chapter also includes an overall reflection and outlook on the project and its final state.
For a more detailed look at the code, we invite you to check out the GitHub repository of our project. There you'll find instructions on how to install and run the code as well as videos showing the turtlebot driving autonomously in simulation and on a real track.

GitHub repository: https://github.com/esatbayhan/avai

Assignment 1: Recording and storing images

The first assignment was about recording and storing images with the turtlebots camera. The requirements of the task can be summed up like this:

- Setup a basic node structure of three nodes:
    - One that takes the raw input from the camera and sends it to another processing node.
    - The processing node receives the images but only resends them further at a certain frequency to another node that is located on the computer.
    - The Node on the computer is responsible for storing the received images and allows setting the frequency with which new images are sent.
- Allowing to request a single image by sending a trigger from the node that is located on the computer.
- The picture-taking should work while remotely driving the turtlebot.
- Setting the frequency of the image processing should be implemented by making use of ROS Parameters.

Implementation

The node architecture of our implementation is displayed in figure 1 and represents the required node structure. The nodes do exactly the things that are specified in the requirements, however, there are a few design decisions that we had to make to derive our specific implementation of the task. The choices we made can be summed up like this:

- We decided to use a publisher/subscriber workflow to send and receive images instead of ROS services. The decision was made to have a consistent interface between the nodes.
- Once the image display node is run the user is asked via the command line to set a frequency at which images should be processed. That frequency is then published in a specific topic that the image processing node subscribes to. There, the frequency is declared as a parameter of the node and therefore is always modifiable by the user. ROS allows setting parameters of nodes via the command line.
- We also configured the image display node such that all received images are stored in a folder but also displayed in a window that updates itself every time new images arrive. The user can press s on the keyboard when the display window is active to manually request an image.

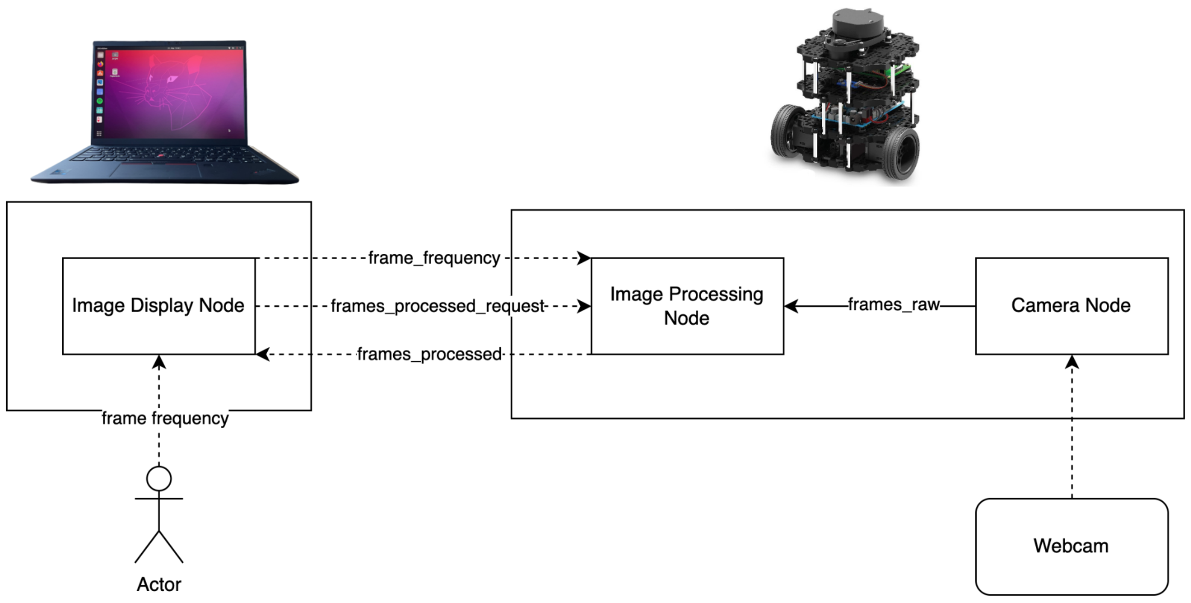


Figure 1: Implemented node architecture for recording and storing images

Task reflection

Although the task seems fairly straightforward we certainly had our struggles from which we learned quite a lot. The main problems and challenges are described below:

- Setting up the communication between the ROS nodes running on the turtlebot and the one that was running inside the virtual machines on our pcs. The solution turned out to be to change the network configurations of the VMs, especially choosing a bridge network adapter.
- The second challenge was in accessing the camera in a ROS-specific way so we eventually went with the standard python solution of working with the OpenCV library.
- After going with OpenCV we had the issue of not being able to send the images on the ROS message Type sensor_msg.Image. Eventually we figured out that we had to use a module called “cvBridge” to make the formats compatible.
- Although the images were now being published we did not receive them on the computer within the image display node. Only after some trial and error, we concluded that the network speed in the seminar room must be the problem. We then figured out that changing the ROS message type from sensor_msgs.Image to sensor_msgs.CompressedImage somewhat solved this issue so that even in a crowded seminar room we were able to receive some images.

Probably the most important learning from the first assignment was the general way of setting up nodes and communication between them. The core principles of ROS and how to build on them were the takeaway that helped us in the following assignments.

Assignment 2: Requirement analysis

Since the turtlebot project was built on the idea of it being an autonomous race car the second assignment was all about defining requirements for such a vehicle. These requirements can be understood as a starting point for the development of a race car but they exceed the scope of the project in this course.

Requirements

In the following chapter, all requirements regarding the functions of the car are defined. This includes a section regarding the quality at which the car needs to operate and the constraints that the car has to follow.

Functional requirements

In this section the functions of the car are defined. The majority of the requirements concern different driving functions that the car needs to be able to perform.

Driving Task Specification

- Maneuvers
    - The race car has to follow the given track autonomously.
    - The race car has to observe the environment and make decisions regarding the motion control.
    - The race can handle different track points in the race (e.g. starting point, finish line).
    - The car has to detect the left and the right cones and always position itself between them.
    - The vehicle should not be driven in reverse.
    - The race car needs to be able to participate in different race modes (e.g. skitpad, acceleration).
    - The race car needs to process sensor data and plan ahead such that it is able to move at a competitive speed.
    - The race car needs to be able to detect humans and stop if they are on the track.
    - The car needs to react appropriately as soon as it is in danger of damaging itself or other things.

- Motion control
    - The actuators need to be controlled in such a way that the race car is following the calculated path.
    - The current speed of the race car needs to be detected and adjusted to fit the anticipated movement along the calculated path.
    - There is an emergency button which activates the emergency brake.
    - There needs be a limit on the velocity while turning to prevent toppling of the car
    - If the race car goes of track it immediately stops after handling corrections faulty
- State

- The car behavior is broadly defined by states:
    - The car is off: The car is standing still and all systems are turned off.
    - Booting state: The car is standing still and all required systems are booted up.
    - In AS Ready state the car is standing still but all systems are running in idl.
    - Driving State: The car is moving along the track while observing the environment and performing all processing to drive safe and competitively.
    - Finish-State: The car has arrived at the finish line and comes to a stop.
    - Emergency State: If the car is not already standing, it comes to a full stop and stays in emergency state until a human stops the program.
- The state of the race car is marked and can be identified by a light

Development / Debugging
- The system is providing at least temporarily comprehensive log data of all driving decisions, driving state, sensor detections and sensor data interpretation.
- The logging behavior of the car needs to be adaptable according to development and racing mode.

Quality requirements

This section is about requirements that concern the quality at which the car operates. Quality requirements tend to be observable by non technical end users.

Environment
- The autonomous system can handle appropriately changes of the environment e.g.
    - wrongly positioned cones
    - weather changes (rain, bright sunlight, lunar eclipse, …)
    - road structure (e.g. potholes)
    - demolished cones

System
- The car is able to drive itself and complete the track with a competitive speed and high reliability. The car must not fail to complete the track 99% of the time.
- During all driving tasks the computer should not exceed 80% of its processing and memory resources, to ensure availability.
- Any internal error (e.g. ROS Node crash) should be handled appropriately (e.g. emergency stop, restart, …)
- If the system or any component is not responding (e.g. by heartbeat) the race car immediately stops.

Safety
- The car needs to perform an emergency break when a human is detected on the track.
- Sensor failures need to be detected and the car needs to be stopped within two seconds (performing an emergency break).

Constraints

This section is about constraints by which the car is bound to operate.

Motion

- The period of pressing the emergency button and the car stops should not exceed 2 seconds
- In case of an emergency break scenario, the car must come to full stop within 2 seconds.
- After the run the vehicle must come to a full stop within 30 meter behind the finish line on the track and enter the finish-state described in T 14.10.

Performance

- The cone detection should have at least 20 FPS and a very high accuracy (>95%).
- The race car is not allowed to drive faster than the software is able to fully detect its surroundings.
- The speed of the race car should be reasonable by at least 0.5 meter per seconds on average (no slow motion)

Others

- The vehicle must comply with all design constraints defined in the “Formula Student Rules 2022, Version: 1.0”.

Testing Strategy

This section is about testing the final product and making sure it complies with all of the above defined requirements. When all tests are passed, the car can be considered a finished product.

Software Tests

- The Software must be developed with a testdriven framework.
- For all functions, there must be a unit test written and all the unit tests must pass before deploying the software onto the car's hardware.
- All software components must be tested with an integration test upon deploying the software onto the car’s hardware.

System Test

- Simulation of the environment: The test is passed if the race car can complete the track in the following weather conditions:
    - rain
    - bright sunlight
    - fog
    - cloudy conditions
- Simulation system stress test: The test is passed if the race car is able to complete the track under the maximum allowed resource consumption

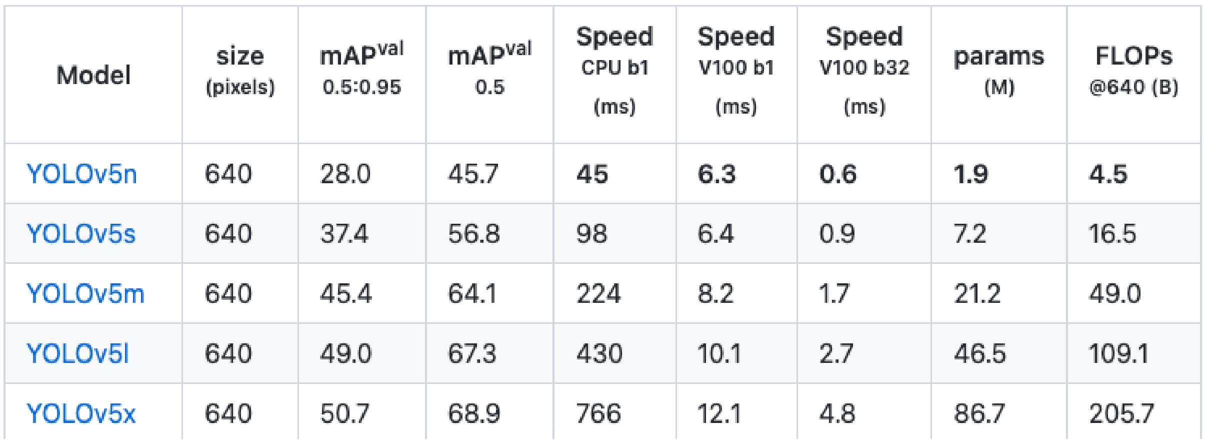
- Simulation track conditions: The test is passed if the race car is able to complete the track with the following conditions:
    - broken cones on the side
    - misplaced cones
    - extra cones in the background
    - tipped over cones
- Driving speed test: The test is passed when the car completes the track under the maximum allowed time specified by the “FS Competition Handbook”
- Sensor error test: The test is passed when the car stops as soon as the sensors operate at a lower rate then defined before.
- Emergency break test: The test is passed when the car performs an emergency break 99.9% of the time in all emergency scenarios like:
    - Emergency Button is pressed
    - Sensor error
    - Human jumps in front of the car

Assignment 3: Cone detection

For the third assignment, we made a bunch of blue, yellow and orange cones out of paper, because those will mark the boundaries of the race track the turtlebot should eventually drive on. Then we took images of them with the turtlebots camera, here the program from assignment 1 came in handy. We annotated the images so that we had the ground truth of where in the image a cone with a certain color was located and trained an object detection model that will be used in the perception module of the turtlebot. In this report, the training process and performance of the trained model are documented and evaluated.

Training process

For good accuracy and high detection speed, a pre-trained object detection model called YOLOv5s was chosen. The model is part of an open-source family of object detection architectures created by ultralytics. All models in that family are pre-trained on the COCO dataset and can easily be adapted to custom objects. The specific version of YOLOv5 was chosen based on the benchmark provided by ultralytics (Ultralytics, 2022). The S-version is the second fastest and second smallest model that processes 640 pixel-wide images (see figure 2). It has a significantly better Mean Average Precision (mAP) score than the fastest model in the family which made it seem to be a good compromise of speed and performance.



Model	size (pixels)	mAP^{val} 0.5:0.95	mAP^{val} 0.5	Speed CPU b1 (ms)	Speed V100 b1 (ms)	Speed V100 b32 (ms)	params (M)	FLOPs @640 (B)
YOLOv5n	640	28.0	45.7	**45**	**6.3**	**0.6**	**1.9**	**4.5**
YOLOv5s	640	37.4	56.8	98	6.4	0.9	7.2	16.5
YOLOv5m	640	45.4	64.1	224	8.2	1.7	21.2	49.0
YOLOv5l	640	49.0	67.3	430	10.1	2.7	46.5	109.1
YOLOv5x	640	50.7	68.9	766	12.1	4.8	86.7	205.7

Figure 2: Benchmarks of the YOLOv5 model family (Ultralytics, 2022)

The training with custom images was achieved by using the YOLOv5 tutorial notebook hosted on Google Colab. A total of 250 images were split into training, testing and validation sets. The split ratio was 70% for training, 20% for testing and 10% for the validation set.

Mean Average Precision Theory

A common way to evaluate object detection models is by calculating a mean Average Precision (mAP) score. This is done by analyzing a new image set and its ground truth labels. The model performs inference on the images and the detected bounding boxes are stored. Then the Intersection over Union (IoU) is calculated for each detected bounding box.

If the IoU is above a certain threshold, the bounding box is considered to be a True Positive (TP) otherwise it's a False Positive (FP).

Calculating IoU:
The calculation of the intersection over union is visualized in figure 3 and is done in three major steps:

1. All detected bounding boxes are matched with one ground truth bounding box that shares the same label and has the highest overlapping area.
2. The area of the intersection is divided by the area of the union of the two bounding boxes.
3. The ground truth bounding box which was matched with a detected bounding box is ignored in future matchings to prevent multiple detections of the same object.

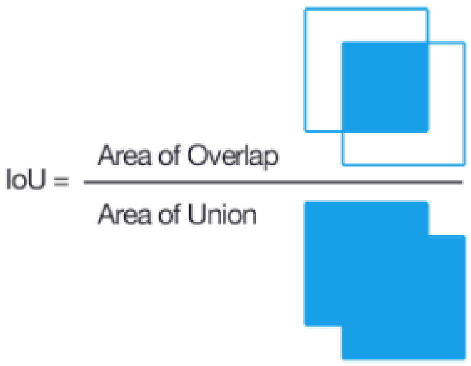


Figure 3: IoU calculation visualized (Cartucho, 2019)

For the IoU calculation done in this report the detected bounding box is considered to be TP if the IoU ≥ 0.5.

Calculating Precision and Recall:
By dividing the number of TPs by the total number of detected bounding boxes (TP + FP) the precision for a class can be calculated. By dividing the number of TPs by the total number of ground truth bounding boxes (TP + FN) the recall value can be calculated. The formulas for these calculations are displayed in figure 4.

$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}}$$

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

Figure 4: Formulas for calculating precision and recall (Khandelwal 2020)

Calculation mAP
The calculation of precision and recall in this report was done on a per image and per class basis. All the Precision and Recall values are plotted on a precision-over-Recall curve where the values are sorted such that the precision values are monotonically decreasing. Then the area under the curve is calculated by numerical integration. The derived value is considered to be the mAP score, which can be calculated on a per class level or model level. The mAP score is between 0 and 1. A higher score indicates a better performing model.

Results of the model evaluation

For the evaluation in the report, an open-source python implementation of the mAP score calculation was used (Cartucho, 2019). The implementation uses the mAP criteria defined in the PASCAL VOC 2012 competition and outputs some visualizations. For the test set, all the annotated images from group E were used, since they were not included in the training process. The output of the mAP calculation included the test images with the ground truth and detected bounding boxes on top.

Detection Results:
As can be seen in figure 5, the overall mAP of our model is 97,58% which can be considered to be good enough for this project. In figure 6 a little more insight is given into the detection results by displaying the number of TP vs FP. The somewhat high number of FP can be explained by considering that we have a high number of instances per image. There are about 500 instances in 173 images, it's obvious that some of those instances are located in the background of the images. Instances in the background are not relevant for the task of autonomous driving, so we concluded that the performance of the cone detection model is fine, meaning that it will probably not be the reason for bad driving behavior.

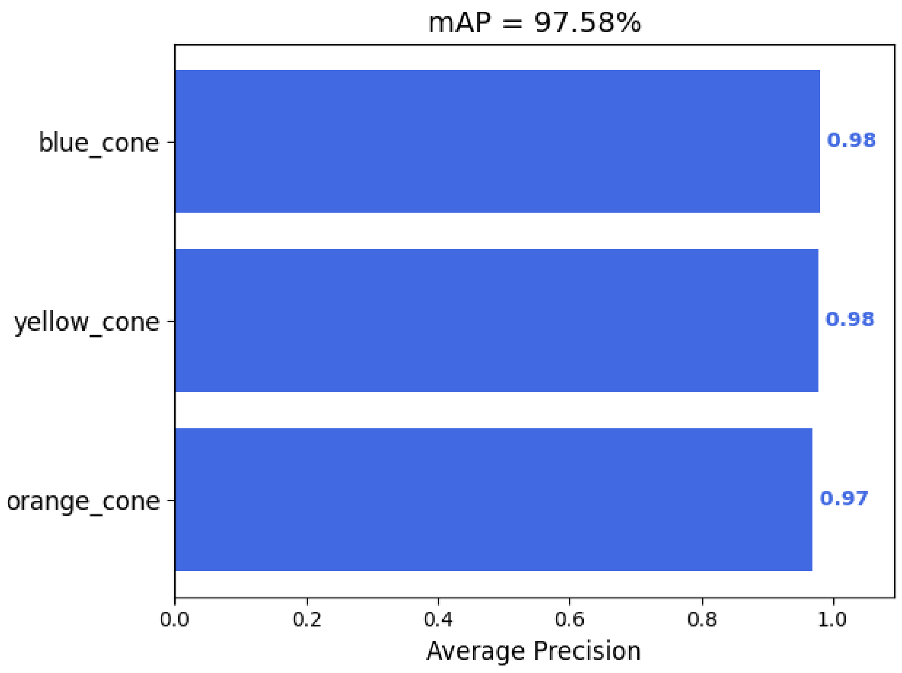


Figure 5: Average Precision scores for all classes (higher is better)

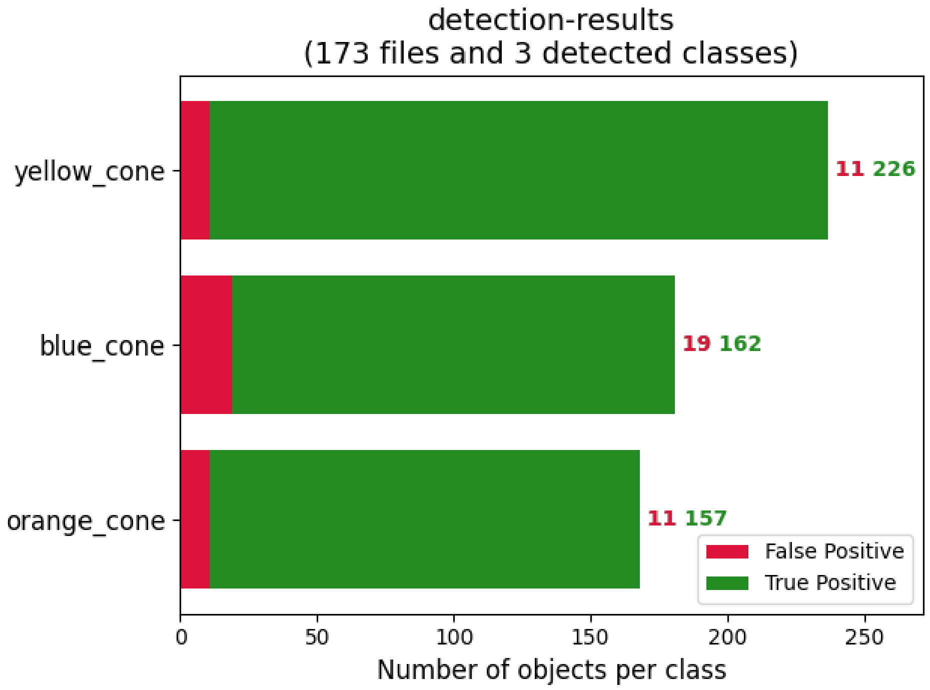
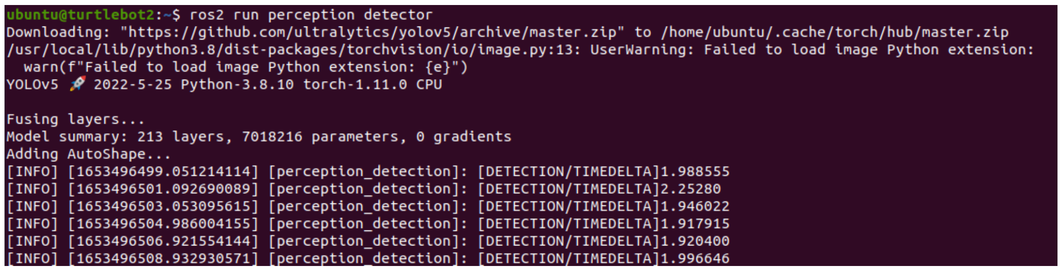


Figure 6: Detection Results (higher is better)

Running the model on the Turtlebot:
The evaluation of the model was done on a laptop but it is assumed that the performance is the same on the turtlebot, except for the speed. In the test of running the model on the turtlebots hardware and analyzing a live video stream from its camera, an average inference time of about 2 seconds was achieved. The conclusion was derived by logging the inference time to the console (see figure 7).



```
ubuntu@turtlebot2:~$ ros2 run perception detector
Downloading: "https://github.com/ultralytics/yolov5/archive/master.zip" to /home/ubuntu/.cache/torch/hub/master.zip
/usr/local/lib/python3.8/dist-packages/torchvision/io/image.py:13: UserWarning: Failed to load image Python extension:
  warn(f"Failed to load image Python extension: {e}")
YOLOv5 🚀 2022-5-25 Python-3.8.10 torch-1.11.0 CPU

Fusing layers...
Model summary: 213 layers, 7018216 parameters, 0 gradients
Adding AutoShape...
[INFO] [1653496499.051214114] [perception_detection]: [DETECTION/TIMEDELTA]1.988555
[INFO] [1653496501.092690089] [perception_detection]: [DETECTION/TIMEDELTA]2.25280
[INFO] [1653496503.053095615] [perception_detection]: [DETECTION/TIMEDELTA]1.946022
[INFO] [1653496504.986004155] [perception_detection]: [DETECTION/TIMEDELTA]1.917915
[INFO] [1653496506.921554144] [perception_detection]: [DETECTION/TIMEDELTA]1.920400
[INFO] [1653496508.932930571] [perception_detection]: [DETECTION/TIMEDELTA]1.996646
```

Figure 7: Screenshot of console output running the trained model on the turtlebot.

Task reflection

Due to good documentation of the open source code we used and prior experience in the team, the task was fairly straightforward. The only two challenges we encountered were training the model without Google Colab closing the session because of inactivity in the tab and getting the color space right. The prediction result had to be transformed from the RGB to BGR color space to be displayed correctly.

Assignment 4: SLAM

In the fourth assignment, the task was to write a program that could localize and create a map of the racetrack. For that, the turtlebot should be remotely operated, while this program detects all the cones, locates them relative to the turtlebots own position and stores their absolute position on a two-dimensional map.

To achieve this task, the perception node we started to develop in the third assignment somehow needed to be advanced such that the bounding boxes of the cones get enriched with distance data relative to the bot. During the lecture we were introduced to several possible ways to achieve this, like depth estimation from consecutive frames, working with a dual camera setup, or fusing the lidar data with the output of the object detection we already implemented.

Since the turtlebot we had access to had a lidar and a single camera we thought the best way for us would be to do a sensor fusion of the bounding boxes and the lidar data. But to figure out how to do this, we thought, it would require us to have more hands-on time with turtlebot than we usually had during the sessions and we needed a quicker way to iterate different solutions. Therefore we decided to invest some time to figure out how to simulate a racetrack and test our algorithms in the simulation.

Building the simulation

With the turtlebot installation, the simulation software “gazebo” was also installed, and we discovered that it includes a “model editor” which allows the placement of different 3D objects in a world. We used a website called “Tinkercard” to create the mesh files for yellow, blue and orange cones and imported them into the model editor. Then we placed the cones such that we had a very basic racetrack. Gazebo allows us to save the entire world in a model file which we then had to launch simultaneously with the turtlebot. With the turtlebot installation came a launch file to launch the turtlebot in an empty simulated world. We took that file and edited it such that it would launch our custom world. After adding a new simulated camera node we got our perception node running and finally had everything the way we wanted to start with the actual assignment.

Although the actual process of creating this simulation is fairly straightforward looking back, it was not when we first figured it out. It took a long time to figure out each step and delayed the start of the actual assignment quite a bit. These were the main issues we encountered:

- Gazebo allows importing mesh files in different formats. However, after some trial and error, we learned that importing only works when they are in the .stl format.
- We had to learn about the “simulation description format” (.sdf) to adjust the size of the cones.
- In the real world, we use the burger variant of the turtlebot which does not include a camera by default. That's because the preconfigured .sdf file of the robot simulation does not provide a camera as well. At first, we tried to edit the .sdf file to include a camera but eventually ended up just using the waffle variant, since it supports the same sensor setup we have on the burger model we had access to. It does have the disadvantage that its lidar sensor is lower and the wheels are further apart with

stronger motors. This leads to more lidar points per cone in simulation than in real life and different driving behavior.

- In the simulation, the camera images are published under a specific topic and in an uncompressed format, so the camera node we used for the real turtlebot didn't work. We wrote a new simulation camera node to bridge this gap to our perception node.
- Using all the default packages of the turtlebot resulted in a hot mess of dependencies which we tried to resolve for quite some time. Trying to extract just the files we needed always seemed to break the simulation or resulted in some weird behavior. So eventually we made use of GitHub submodules to link forks of all the packages to our code.

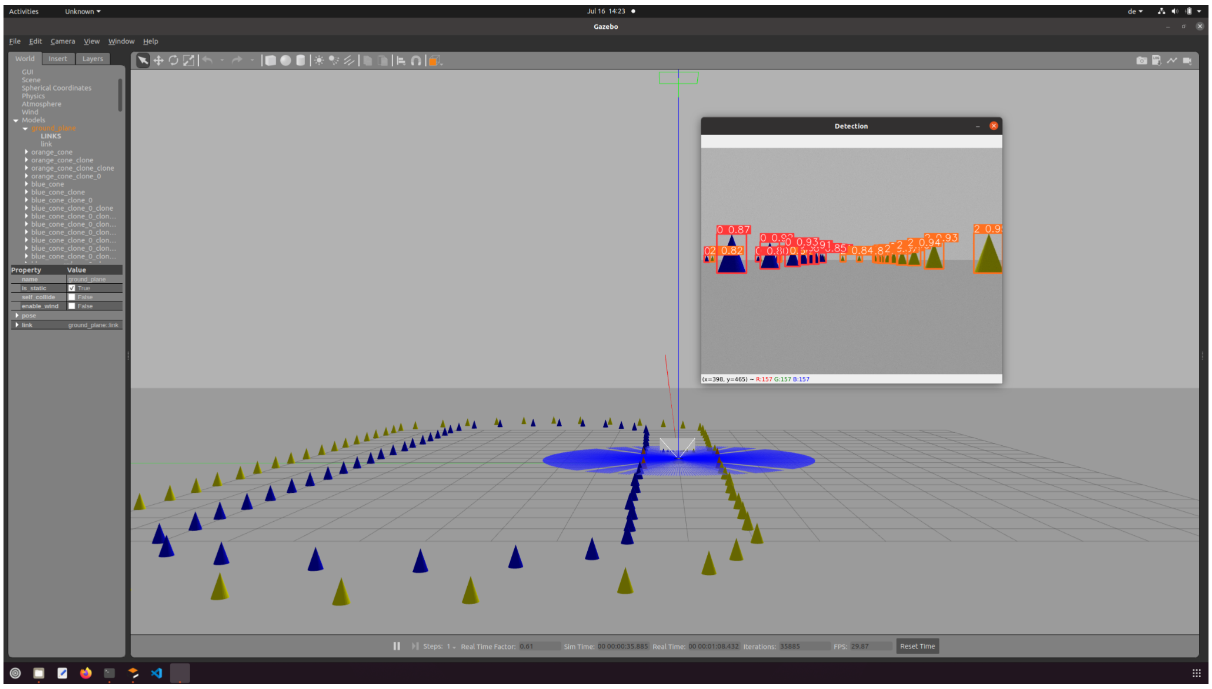


Figure 8: Screenshot of the simulation running with the object detection node.

Solving SLAM

Since the simulation took much longer than we originally thought, we wanted to solve the problem of simultaneous localization and mapping (SLAM) fast. We thought it should be possible to use the SLAM node already provided with the turtlebot installation to achieve the task of fusing the sensors and tracking the cones. The belief was supported by the fact that we saw a new topic called /scanMarchedPoints popping up every time we launched the slam node. We spent quite some time reading about the Nav2 package and the included slam_toolbox only to eventually realize that the only chance of finishing the task on time would be to just do it by hand.

So we worked out an approach that involves receiving the lidar scan, odometry and bounding box data fusing it all to create a map. The actual fusion of the bounding boxes and lidar ranges was easier than we thought because it only involved matching indexes of the range array to the x coordinate of the image. Unfortunately, we didn't finish the implementation of the entire approach before the deadline of the assignment because we struggled with the math involved in mapping the cone's location on an absolute cartesian coordinate system. The chart below in figure 9 shows the steps involved in our approach and what we got working and what not.

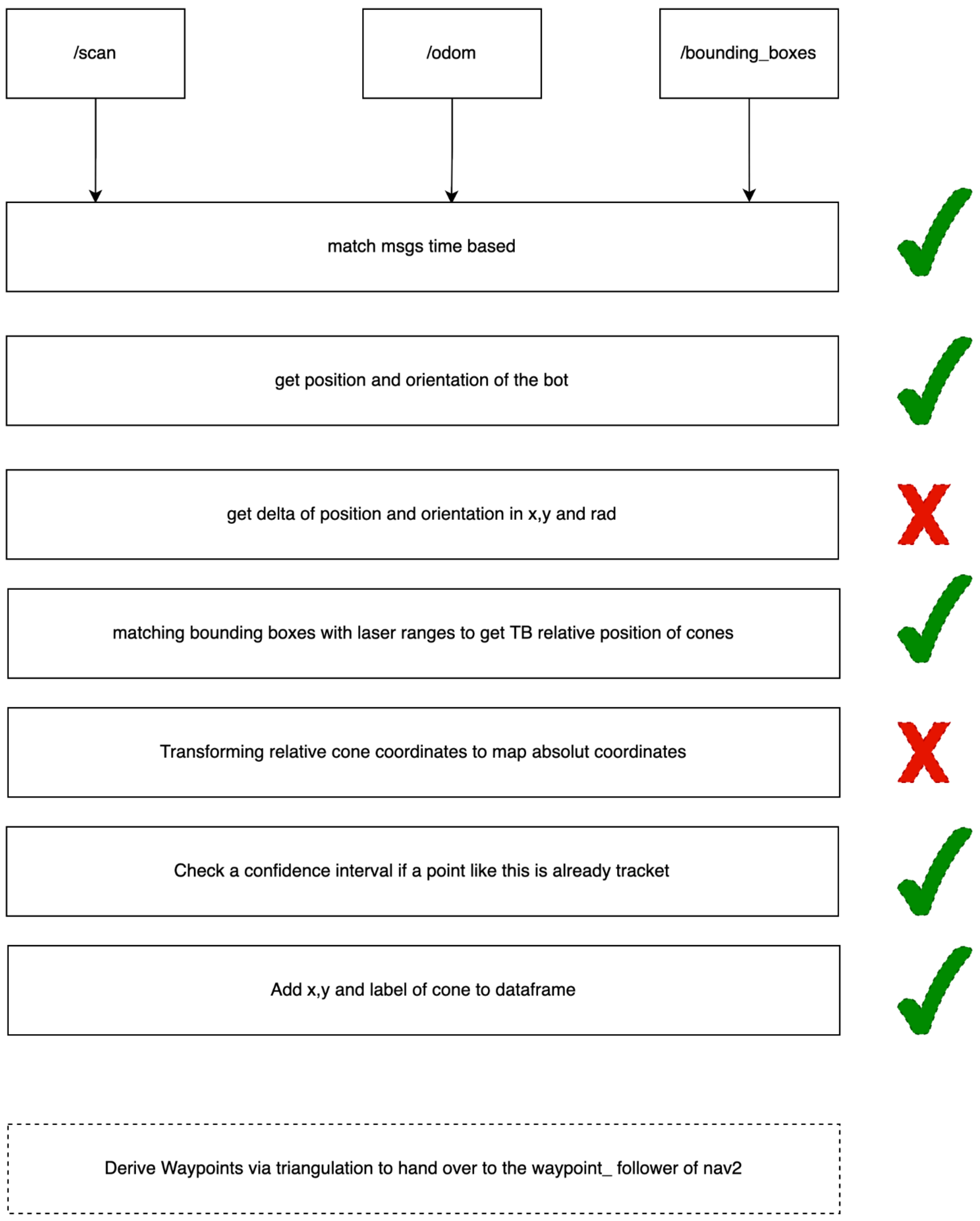
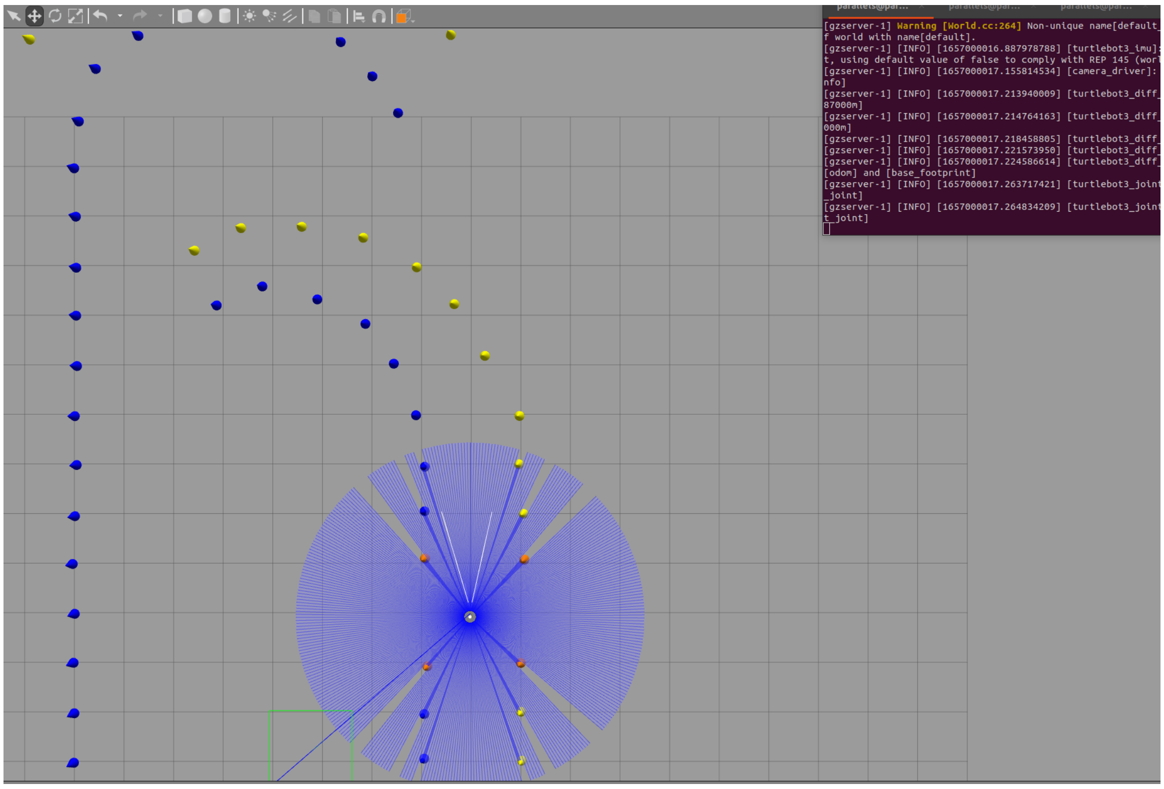

/scan
/odom
/bounding_boxes
match msgs time based
✓
get position and orientation of the bot
✓
get delta of position and orientation in x,y and rad
✗
matching bounding boxes with laser ranges to get TB relative position of cones
✓
Transforming relative cone coordinates to map absolut coordinates
✗
Check a confidence interval if a point like this is already tracket
✓
Add x,y and label of cone to dataframe
✓
Derive Waypoints via triangulation to hand over to the waypoint_ follower of nav2

Figure 8: Screenshot of the simulation running with the object detection node.

The result of this unfinished approach allows us to map the first cones that are visible to the camera and lidar. In the situation displayed in figure 9, we can create a map of the cones that are visible to the camera and are in the range of the lidar scanner. So we get the map displayed in figure 10 where the turtlebots location is at the point (0,0). But as soon as we start driving the curve visible in figure 9, the cone placements on the map are incorrect as viewable in figure 11.



```
[gzserver-1] Warning [World.cc:264] Non-unique name[default_
f world with name[default].
[gzserver-1] [INFO] [1657000016.887978788] [turtlebot3_imu]:
t, using default value of false to comply with REP 145 (wor
[gzserver-1] [INFO] [1657000017.155814534] [camera_driver]:
nfo]
[gzserver-1] [INFO] [1657000017.213940009] [turtlebot3_diff_
87000m]
[gzserver-1] [INFO] [1657000017.214764163] [turtlebot3_diff_
000m]
[gzserver-1] [INFO] [1657000017.218458805] [turtlebot3_diff_
[gzserver-1] [INFO] [1657000017.221573950] [turtlebot3_diff_
[gzserver-1] [INFO] [1657000017.224586614] [turtlebot3_diff_
[odom] and [base_footprint]
[gzserver-1] [INFO] [1657000017.263717421] [turtlebot3_joint
_joint]
[gzserver-1] [INFO] [1657000017.264834209] [turtlebot3_joint
t_joint]
```

Figure 9: Starting situation and modified track for SLAM development in simulation.

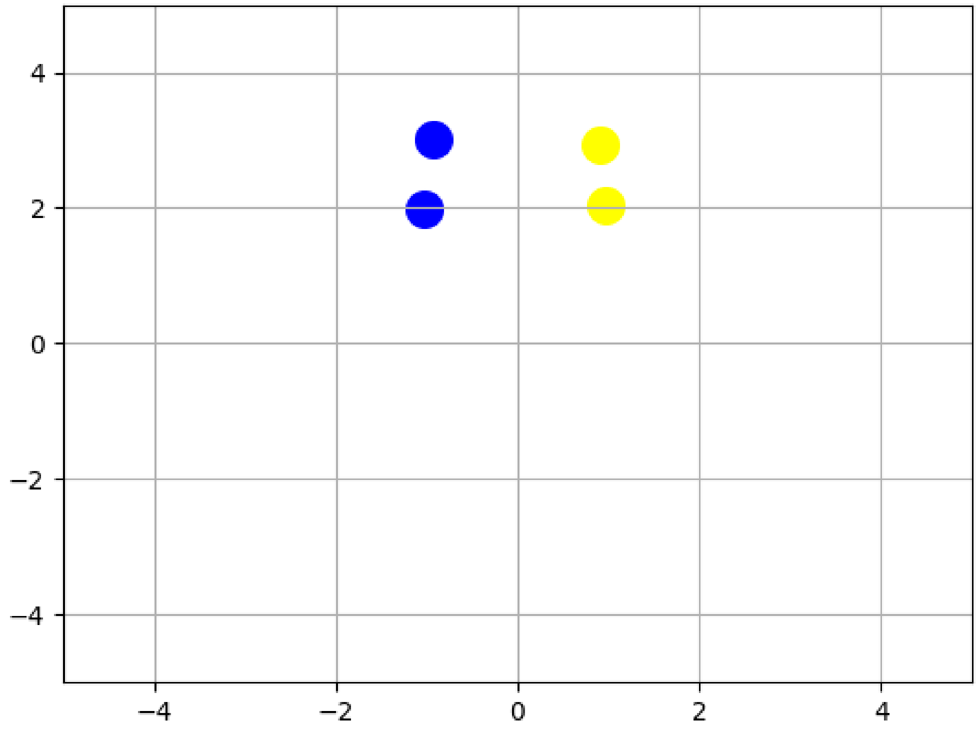


Figure 10: Initial racetrack map before moving the turtlebot.

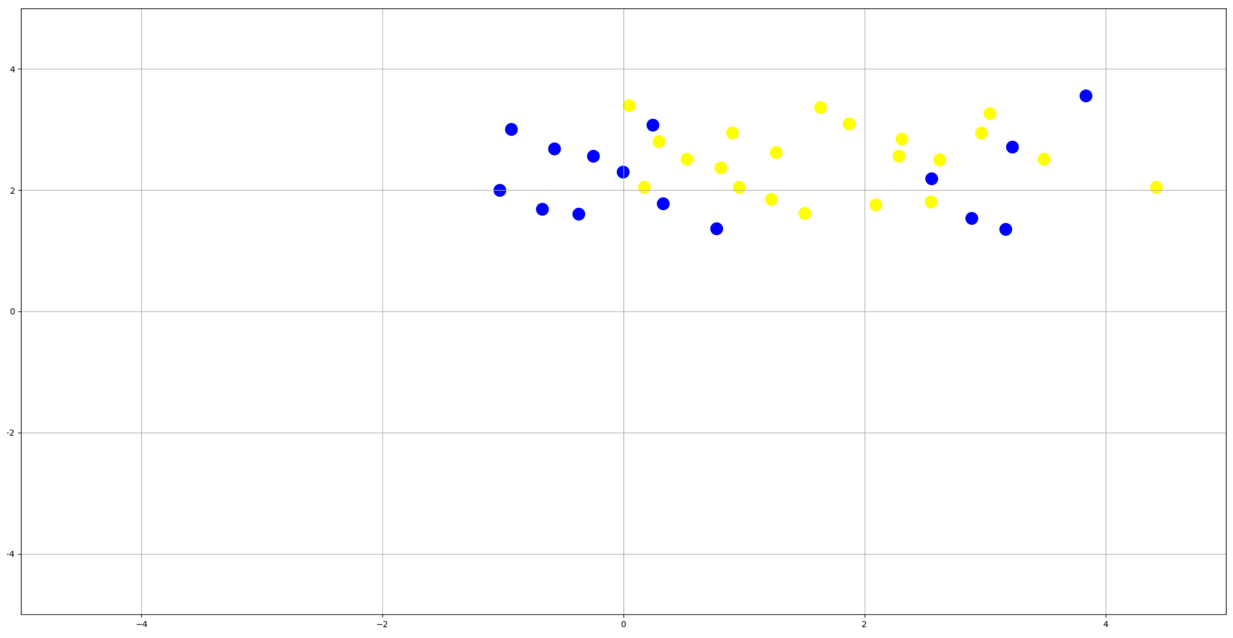


Figure 11: Racetrack map after driving the curve visible in figure 10.

Task reflection

Although we didn't succeed in finishing the SLAM task, we learned quite a lot and produced results we could use in the final task of making the turtlebot drive autonomously. Building the simulation was the right decision because it allowed us as a team to work more independently and intensely on the development of our algorithms. The initial education efforts on the default ROS navigation packages made us realize the potential of using it for autonomous driving. Taking the steps towards building the map allowed us to understand the available data structure of the laser scan message and get a first intuition on how to work with the orientation of the turtlebot. It also taught us about fusing the bounding boxes with the laser data. These things translated to faster progress in the upcoming task of making the turtlebot drive autonomously. However, we would have liked to finish the mapping task instead of spending so much time reading about the nav2 package. We might have learned even more.

Assignment 5: Autonomous driving

Right at the beginning of the autonomous driving assignment, we agreed on implementing a stateless approach. Although there is a good chance that a stateful approach would work better than a stateless, considering the time we had left to finish the project we thought it was more realistic to get something working by going stateless.

So far we only had a camera processing node that sends images to the perception node which generates bounding boxes for all cones. Now we needed to find a way to merge the lidar data with the bounding boxes, to get the position of each cone relative to the turtlebot. With the position of each cone in the camera's field of view, we needed to define an algorithm that outputs a distance and an angle at which we need to drive. We also wanted to implement some sort of emergency break in case the connection to the turtlebot is lost.

Implementation

All the code that is responsible for sensor fusing, locating the cones and making decisions on the driving parameters can be found in the “controls” package. The package contains two nodes, one is called controller and the other heartbeat.

Heartbeat

Heartbeat is there for safety and is run on the turtlebot. It receives all the driving commands from the controller as well as a message called heartbeat. When the heartbeat message comes in for the first time, a timer is set which regularly checks the time of the successive heartbeat messages arriving. If the most recent heartbeat message has arrived within a specified period, the heartbeat node keeps forwarding all driving commands received. Else an emergency brake is executed and no more driving commands are forwarded.

Controller

The controller node does all the sensor interpretations and calculations necessary to generate a distance and an angle to drive. For the sake of this report, let us call the angle and distance value combination waypoint. The entry point of the waypoint calculation is the “subscribe_bounding_boxes” method, which is the callback function for the bounding boxes published by the perception node. The controller node also receives the lidar data but those values are only stored in a class variable for further processing. The bounding boxes are published less frequently than the laser scan so that they determine the rate at which new waypoints are calculated.

Selecting nearest cones for Waypoint calculation
In the callback function for the bounding boxes, the first step is to get the location of the nearest blue and yellow cone. To identify those cones the x coordinates of all bounding boxes are matched with their corresponding lidar range. All the ranges and angles for blue and yellow cones are stored in separate lists so that after iterating over all bounding boxes the minimum distance to a blue cone and minimum distance of a yellow cone can be selected. The selection is then considered to be the nearest cones.

Rotation of the turtlebot incase of missing nearest cone

In the case of an empty value for a blue or yellow cone, the turtlebots forward speed is set to zero and his turning speed is increased. Because, if the value for one of each cone is empty, this means there is no cone of that color in the camera's field of view that is also in the range of the lidar. Therefore, the turtlebot will turn in the direction of where such a cone is expected until a new cone is detected. If the blue cone is missing, the turtlebot turns left, if a yellow cone is missing, the turtlebot turns right. Once the turtlebot senses both nearest cones again, the waypoint calculation can start.

Waypoint Calculation

The waypoint calculation part in the controller node is the most mathematical complex. It generates a distance and an angle at which the turtlebot will drive by adjusting its angular and linear speed accordingly. For the stateless approach to work, this is the most crucial part to get right. We needed to draw some visualizations of the calculations for us to understand them so to explain them, we think it is best to share them here.

1. Let's recall the situation from above where the turtlebot senses two nearest cones: A blue one on the left and a yellow one on the right, with the turtlebot in front of them (see figure 12, turtlebot is displayed as the circle). What we know is the distance to each cone (d1 and d2) and their angles relative to the turtlebots orientation (α and β).

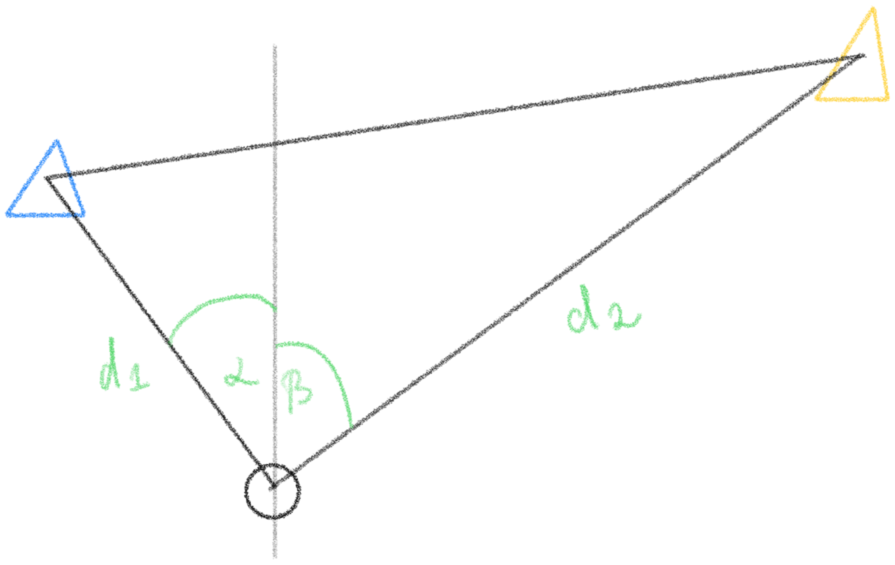


Figure 12: The starting situation of the waypoint calculator.

2. Next we want to find the angle and the distance to the midpoint of the two cones. The angle has to be relative to the turtlebots orientation (see figure 13).

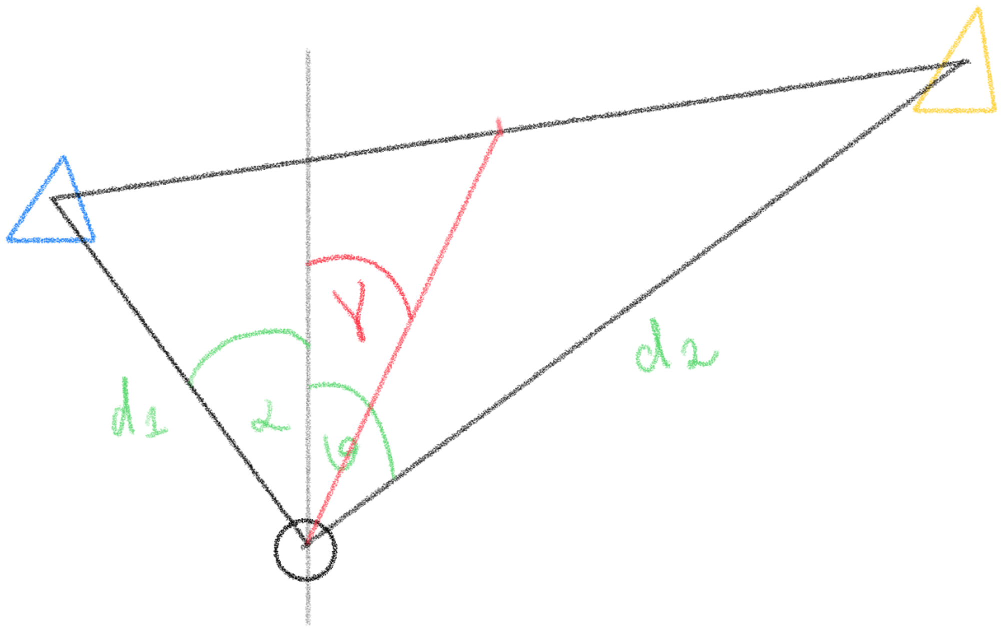
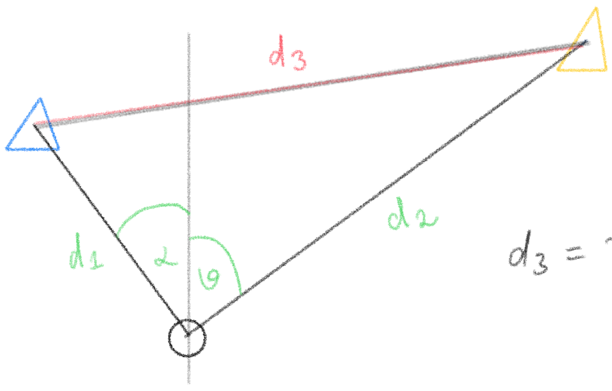


Figure 13: In the starting situation we are looking for gamma.

3. To get this angle, we first have to calculate the distance from one cone to another like it is shown in figure 14.



$$d_3 = \sqrt{d_1^2 + d_2^2 - 2 \cdot d_1 \, d_2 \cdot \cos(\alpha + \beta)}$$

Figure 14: Calculation of the distance between the two nearest cones.

4. Now the triangle marked by the cones and the turtlebot is known. Because of that, we can calculate the median triangle which is the distance from the turtlebot to the midpoint of the two cones (see figure 15).

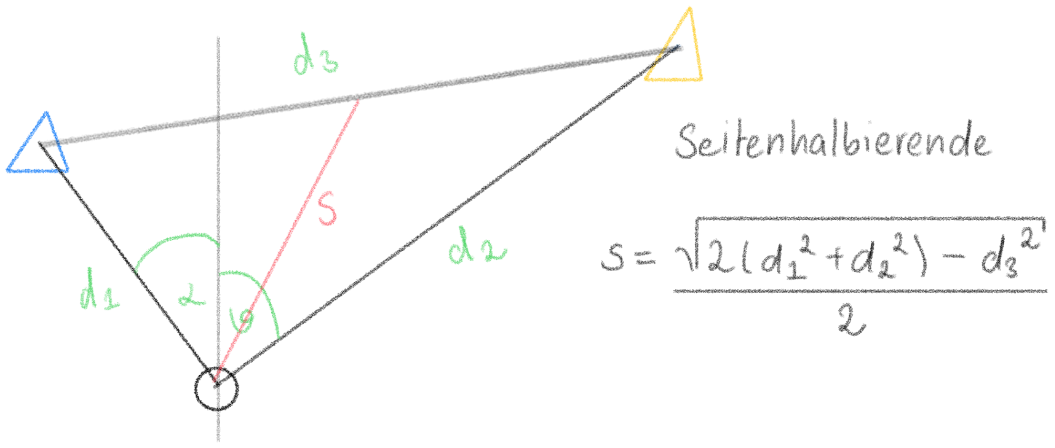


Figure 15: Calculation of the median triangle.

5. The last step before calculating gamma is to find the x coordinate of the midpoint of the two cones relative to the turtlebot. This can be done by adding the relative x coordinates of the cones and dividing them by 2 (see figure 16).

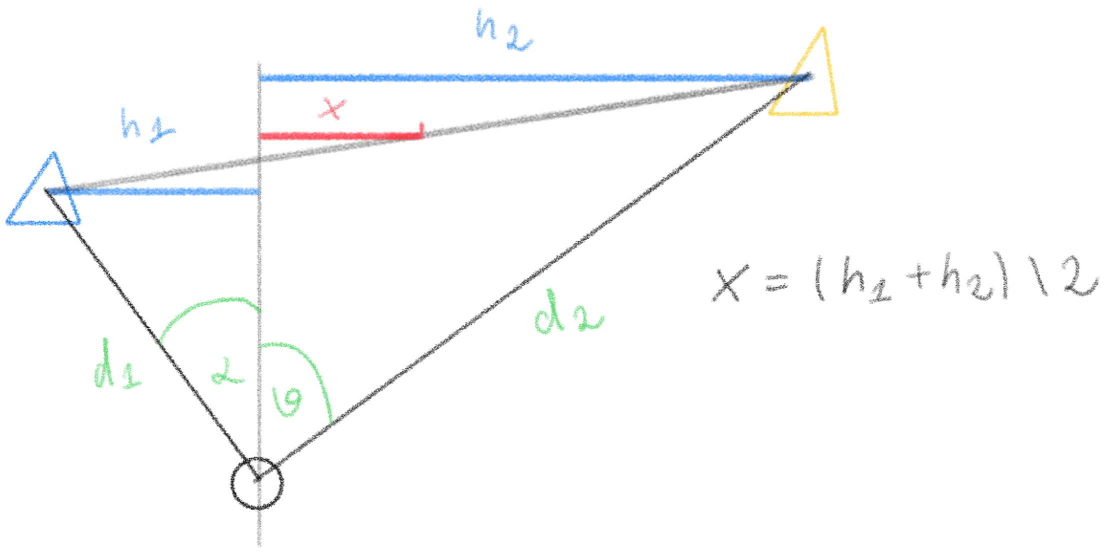


Figure 16: Calculation of the distance x.

6. In the final step, We calculate gamma so we know the angle at which we want to drive the previously derived distance “s” (see figure 17).

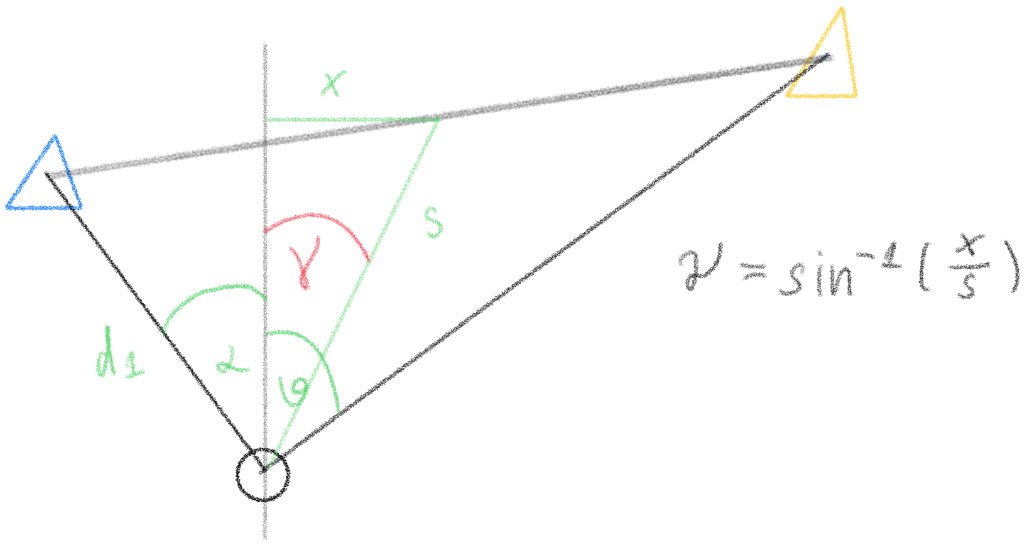


Figure 17: Calculation of gamma.

The derived angle gamma and the distance “S” are then sent as driving commandos to the turtlebot. The turtlebot will accelerate towards the calculated midpoint and will only change its course if the arrival of new bounding boxes leads to a new waypoint calculation.

Final node architecture

To get an overview of the entire node structure that is needed to achieve the autonomous driving task one can look at the generated ROS graph. Since the graph is displayed horizontally, which does not work well in this document, the graph is split into two figures: 18 and 19. The part in figure 19 displays the relevant nodes and topics for the recently implemented controls package.

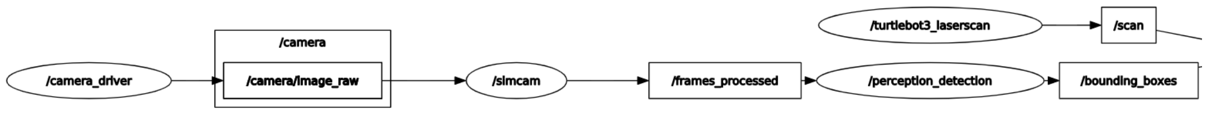


Figure 18: Final ROS Graph part 1.

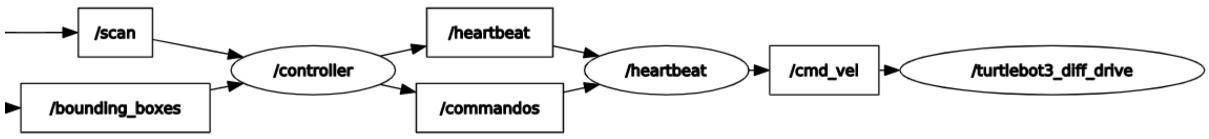


Figure 19: Final ROS Graph part 2.

Task reflection

To get an understanding of the driving performance of the turtlebot as well as in the real-world and simulation, please find the videos in our GitHub repository.

The driving performance is not perfect. However, we consider it to be a decent outcome for the time we had and for others to further develop the project. The challenges involved in making the controller work are many, so let's just name a few.

- Driving in simulation does not translate directly to driving in the real world: The turtlebot in the simulation can drive faster than the max speed of the model in the real world. In our case that leads to a better driving behavior in the simulation than in the real world.
- Also, the computations of calculating waypoints and sending images from one node to the other are faster in the simulation than in a real setting. This is mostly due to the network communications involved in the real setting.
- The narrow field of view of the camera makes the turtlebot cut corners pretty aggressively. So he does not do well in tight corners but is doing fine in wide ones.
- The turtlebots camera is not facing precisely forward, small angles at which the camera is tilted result in an increase in cutting corners because close cones are not in the field of view anymore.
- Uneven terrain can result in loss of orientation for the turtlebot, because its lidar ranges are tilted upwards and don't hit the tip of the cones anymore.
- There are many parameters like different thresholds and rates to determine when implementing such a controller as we did. The combination of all these parameters results in a certain performance. Further tuning those parameters could potentially lead to better driving performance.

If we think about further advancing our implementation a few things come to mind:

- We could try to make use of more sophisticated controller implementations like the ROS waypoint follower.
- We could advance the stateless implementation to keep track of close cones such that the corner-cutting issue can be solved.
- Maybe combining a stateless and stateful approach at the same time could make a huge difference in driving performance because more advanced path planning can be done.

In the end, doing this project was fun and taught us a lot. We feel like the learnings we got from this project translate into more domains of software development than just working with ROS.

References

Cartucho, J. (2019). *mean Average Precision - This code evaluates the performance of your neural net for object recognition.* https://github.com/Cartucho/mAP

Khandelwal, R. (2020). *Evaluating performance of an object detection model | by Renu Khandelwal | Towards Data Science*. https://towardsdatascience.com/evaluating-performance-of-an-object-detection-model-137a349c517b

Ultralytics. (2022). *ultralytics/yolov5: YOLOv5 ? in PyTorch > ONNX > CoreML > TFLite*. https://github.com/ultralytics/yolov5