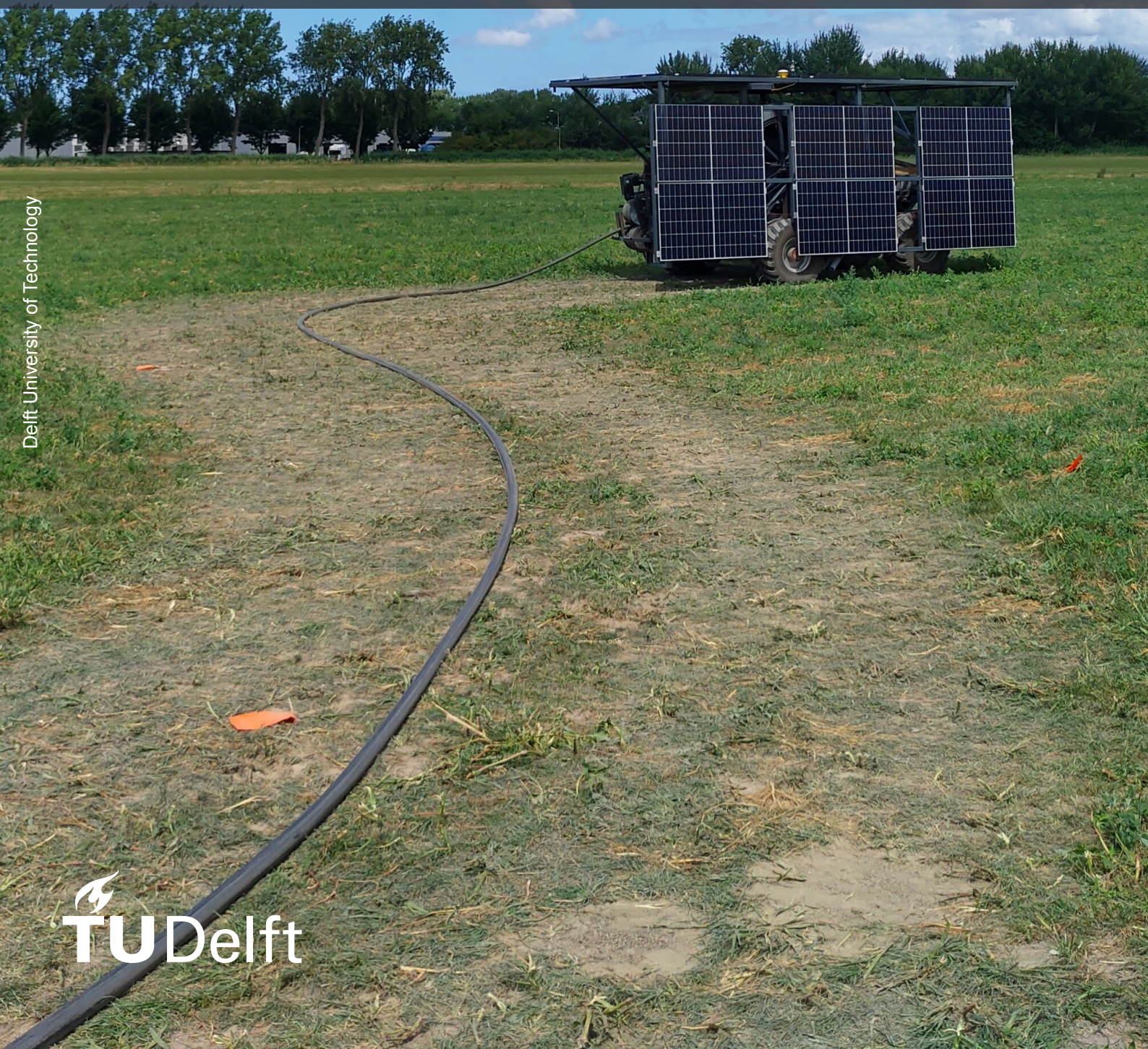
Implementation of Vision Control in an Agricultural Robot

RO57035: Thesis Report

Pieter van Driel



Delft University of Technology

TU Delft

Implementation of Vision Control in an Agricultural Robot

by

Pieter van Driel

Student Name	Student Number
Pieter van Driel	4570758

The work in this thesis was made at:

Cognitive Robotics
Mechanical Engineering
Delft University of Technology

In collaboration with:

Lely Technologies

Committee: dr. Y.B. Eisma
dr. J. F. P. Kooij
K. van den Berg

Project Duration: 01/2022 - 10/2022

Preface

This master thesis has been written to fulfil the graduation requirements of the Robotics master at the Technical University of Delft. Prior to the robotics program, I successfully followed the bachelor of Mechanical Engineering at the Technical University of Delft.

The main objective of my research is to propose a new method to follow a trajectory with an agricultural robot. The challenge lies in the fact that the hose is physically attached to the vehicle. The method that is proposed, uses a visual control system to perform this task. During the master a lot of work is being conducted using simulations. However, this thesis applied the proposed method to a real robot. Therefore, I gained a lot of experience by working with robots outside a simulation. Besides, making a thesis in collaboration with a company gives its own challenges, which are not taught at the university. Therefore, this thesis has been a great opportunity to gain those additional skills.

I would like to thank everyone who contributed to this project, in particular my supervisors Yke Bauke Eisma and Karel van den Berg for their critical thoughts and ideas. Also, I would like to thank Robin Oosterbaan for setting up the general software of the robot. Lastly, I would like to thank Howard Sie for making the hardware for the robot itself.

Pieter van Driel
Delft, November 2022

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
AI	Artificial Intelligence
AV	Autonomous Vehicle
DBSCAN	Density Based Spatial Clustering non-parametric Algorithm with Noise
FoV	Field of View
IBVS	Image-Based Visual Servoing
IPM	Inverse Perspective Mapping
LKAS	Lane Keeping Assist system
MIoU	Mean Intersection of Union
PBVS	Position-Based Visual Servoing
PID	Proportional Integral Derivative
RANSAC	Random Sample Consensus
RIVM	Rijksinstituut voor Volksgezondheid en Milieu
RoI	Region of Interest
ROS 2	Robot Operating System 2
ROV	Remotely Operated Vehicle
VO	Visual Odometry
VS	Visual Servoing

List of Figures

List of Tables

1

Summary

This thesis proposes a new method for a visually controlled agricultural robot. The robot has to follow and simultaneously reel in a hose which lies on a grass field. It is possible that the already laid down hose shifts while the robot is driving forwards or backwards. Due to the changing position of the hose, the robot will lose information about the exact location of the hose. The trajectory of the hose is required to let the robot follow the hose. Therefore, the goal of this thesis is to research the possibility to control the robot, based on a vision system. This problem is further explained in the chapter 2 of this thesis. By making a comparison with similar robots, the conclusion can be drawn that this problem is unique because the 'trajectory' is physically attached to the robot. It means that when this robot follows the trajectory, the trajectory can move with the robot. This differs with other robots, because they follow a trajectory without changing their trajectory.

For this thesis, a vehicle controller has been developed that uses the input from a vision system. The performance of this controller is tested by conducting two experiments. The method is developed using the Sense-Think-Act robot model. The main part of this thesis is devoted to explain the workings of these methods used to control the robot. Starting with discussing the design choices made concerning the sensor, and how the images will be transformed such that the pixel coordinates are mapped to the real X and Y coordinates. Thereafter, it is explained how the data from the sensor is used to extract the information concerning the hose. This is done by using a vision pipeline, meaning that there are multiple consecutive steps involved. The deep neural network of Deeplabv3+ [6] is used to segment the hose such that the hose and the background are classified individually on pixel level. The next step is to cluster the output of the segmentation network using the DBSCAN algorithm [20]. The clustering algorithm is used to filter out noise and to cluster elements that are close to each other. The clustered output will undergo a polynomial fit using polynomial regression [34]. The polynomial acts as the 2D representation of the hose. From this polynomial, 5 points are extracted. The vehicle controller is based on those 5 points as an input for the Proportional (P) controller to steer the front axle. The input is also used to control the linear velocity of the vehicle using a Proportional Integral Derivative (PID) controller.

The prosed method to visually control the vehicle is tested and validated. Two tests are conducted to test different driving behaviours and to see how the vision pipeline performs compared to examples found in the literature. The first test was conducted to identify differences in driving behaviour when the robot drives along different curves. The first test is conducted with a linear velocity of 0.2 m/s and with a camera setup that observes closely in the vicinity of the vehicle. It turns out that the vehicle shows different behaviour, while driving along a left or right curve. The second test is conducted to see whether there are differences between directional transitions. Furthermore, the overall length of the second track was larger. This test is repeated four times using different parameters for each subtest. The effect of changing the linear velocity is tested and the effect of changing the camera angle is tested. It is found that the robot steers too early for when the camera has a larger Field of View. When the vehicle drives faster, it tends to steer earlier. Besides, a higher velocity causes oscillations. The conclusion can be drawn that the oscillations are caused by the lag of the camera. It has been found that the robot is able to correct itself when it steers too early. During the tests, images were recorded

to validate the performance of the semantic segmentation. The neural network used in this research reached a Mean Intersection over Union (MIoU) score of 58.6%. This score implies the degree of similarity of the output produced by the semantic segmentation with the ground truth. It is observed that this model can perform better when there is more training data available. Furthermore, the data augmentation used to create extra images can be extended.

The final conclusion of this report is that the proposed method of visual control performs well in a situation with short grass. The segmentation model troubles with tall grass, due to the partial occlusion of the hose caused by the grass. This method shows a proof of concept, meaning that such a method of visual control can work properly for situations when the trajectory is physically attached to the robot itself.

2

Introduction

This thesis will research the possibilities of implementing a new method for navigating and controlling an agricultural robot. A hose is attached to the robot, which it has to follow. This method to navigate is based on visual control, using semantic segmentation in combination with clustering and polynomial regression to create a path, which will be used as input for the vehicle controller in order to follow the trajectory of a hose. This thesis is carried out in collaboration with the University of Technology Delft, and Lely Technologies.

2.1. Motivation

Lely Technologies develops new robots for the dairy farm sector. Their goal is to automate the farming sector to make farming more environmental and animal friendly. Furthermore, they want to lighten the job of the farmer. The environment is especially important for the Netherlands, because of the ongoing nitrogen crisis. The contribution of nitrogen emissions from the agricultural sector is 45 % according to the RIVM [36]. Lely Technologies develops different robotic solutions to reduce emissions from dairy cows. This thesis investigates a new method for navigating and controlling one of their robots, that aims to further accomplish the goals of Lely. This robot will reduce emissions by automatically fertilizing grasslands with liquid manure coming from milking cows. The process of spreading manure, according to W. Koopman [19], is normally done by a farmer using either a trailing foot, trench coulter or slurry injector. The mentioned methods are used in combination with a tractor coupled to a manure trailer. Another way to supply the liquid manure is by using a long hose between the tractor and the storage of the liquid manure. The liquid manure is diluted with water and then pumped towards the tractor that spreads the manure using either a trailing foot or trench coulter. A benefit of using a diluted solution is that it reduces the amount of ammonia [19] and according to A. van der Wal [41] it also reduces the track formation coming from the wheels of the tractor. The track formation is reduced, because the weight of the trailer does not need to be pulled while driving. With their robot, Lely tries to tackle two common problems that occur by using the current methods. First, spreading liquid manure is a time-consuming task for the farmer. Secondly, it is common that a farmer spreads a large amount of manure at one instance. But, eutrophication will occur when the liquid manure is spread all at once. Furthermore, it emits nitrogen and methane gasses in large amounts when spread at once, as described in the review by S. G. Sommer, and N.J. Hutchings [38]. The robot that is developed will perform this task autonomously, meaning that the limiting factor of labour is out of the picture. The robot stands alone, meaning that one robot will fertilize the grass field. Because the robot performs this task autonomously, the threshold to spread liquid manure at higher frequencies decreases. Fertilizing the ground at a higher frequency, means that the amount of liquid manure at one instance can be reduced. Nevertheless, the total amount of fertilization on a yearly basis stays equal. From Lely's own research, it has been found that fertilizing at higher frequencies reduces the amount of eutrophication. The ground can absorb a limited amount of minerals. When the fertilizing is done all at once there are too many minerals for the ground to absorb. So, when it rains after fertilizing the ground, eutrophication occurs. This happens less when the liquid manure is spread at a higher frequency with lower amounts.

Additionally, fertilizing at higher frequencies reduces the forming of nitrogen gasses.

In order to provide liquid manure to the robot, a long stiff hose is attached between the robot and the pump station. The pump station, together with a manure storage, is located close to the dairy farm. The goal of the robot is to spread the liquid manure. An additional task is to reel in or reel off the hose. The entire hose fits on a reel that is positioned on top of the robot. When the robot drives forward, it will spread the liquid manure. When the robot drives backwards, it will reel in the initially laid down hose. Laying the hose down is done based on a map that contains GPS locations of the path that should be fertilized. When unwinding the hose, the possibility exists that the position of the earlier laid down hose is changed. This behaviour occurs when the rate of unwinding does not match the rate of driving. With mismatching rates, the robot pulls or pushes too hard on the hose, thereby changing the position of the hose. Due to this behaviour, the position of the hose as it is remembered by the robot can differ from its real position. A problem arises when the robot drives based on its GPS locations, when the hose is laying at a different location than where the robot thinks it is. It could result in a situation where the robot will push too hard, thereby creating a buckle in the hose. This could potentially damage the hose. Therefore, the robot should know the exact position of the hose, while it is driving backwards in combination with reeling in the hose. To let the robot perform this task autonomously, there is need for a robust solution that uses real-time information to determine the path of the hose.

2.2. Problem statement

According to M.B. Alatise and G.P. Hancketo [1], every robot has four base challenges: localization, navigation, path planning and obstacle avoidance. The above-mentioned robot from Lely localizes itself with respect to a map using GPS. The localization of the hose with respect to the vehicle could be lost due to hose shifts. Lely wants to research the possibility to localize the position of the hose with respect to the vehicle. In addition, they want to research the possibility to control the vehicle based on the trajectory of the hose. Therefore, this thesis will develop a method that detects and follows the hose of the robot. Following the hose is only possible when the robot drives backwards. It means that driving forwards is out of the scope. However, an overview between driving forwards and backwards is given in table 2.1. The table shows the differences on how these challenges are implemented when the robot drives forward and how they should be implemented in the case when the robot drives backwards.

Table 2.1: The four challenges concerning the robot

Challenge	Driving forwards (laying the hose down)	Driving backwards (reeling in the hose)
Navigation	The robot has to navigate from point A to B while laying the hose on the field. This is done by using the coordinates from the GPS.	The robot has to navigate from point B to A while reeling in the hose. The robot should navigate based on the trajectory of the hose.
Localization	The robot uses GPS for the global localization. It means that the robot will localize itself with respect to a map.	The robot should localize its position with respect to the hose. This cannot be done using GPS and should be done using another sensor.
Path planning	A path planner is used to create a path from A to B	When the robot drives back from point B to point A, it should create a path based on the trajectory of the hose. So, it plans a path relative to the position of the robot.
Obstacle avoidance	The obstacle avoidance is offline. It means that only the path planning ensures that the path does not cross such that the hose cannot cross an earlier laid hose.	Online obstacle avoidance is out of the scope for this thesis

To recap table 2.1: the robot should navigate from B to A, based on the trajectory of the hose; the robot localizes itself with respect to the hose; and it plans a route such that the robot can follow the trajectory of the hose. Lely wants to use one additional sensor to sense its environment and detect

the trajectory of the hose such that is able to perform the above-mentioned challenges. In table 2.2 common sensors are shown to sense the environment of a mobile manipulator. Tactile sensors, wheel encoders, heading sensors and active ranging sensors are not suitable for detecting the hose, because they are not giving any information about the trajectory of the hose. Optical sensors and vision-based sensors are the only two that could be applicable. The review from E. Arnold et al. [2] describes pros and cons between using cameras or LiDAR for 2D and 3D object detecting. Their use case is for autonomous driving, which comes close to our case. They conclude that cameras are cheap in their usages. However, they are prone to adverse light and they do not provide any depth information. A LiDAR gives precise depth information, but they lack in showing texture and they expensive and large. In addition, they contain rotating parts which can get dirty over time. A camera sensor is easier to clean than mechanical parts. Due to the fact that cameras are cheap and that it only has to detect the hose, the decision is made to use a monocular camera for detecting the trajectory of the hose.

Table 2.2: The four challenges concerning the robot, table adapted from [1]

Tactile sensors	Contact switches, optical barriers, proximity sensors	Tactical sensors are designed to determine the exact position of an object at a short distance via physical contact. Tactical sensors are mostly used to calculate the amount of force applied by the robot's end effectors.
Wheel encoders	Optic, magnetic	They measure the distance or speed of the robot. The wheel encoders also count the revolutions of each wheel and orientation.
Optical sensors	Infrared, LiDAR	These are light-based sensors which produces range estimates based on the time needed for the light to reach the target and return.
Heading sensor	Gyroscope	Sensors that are measuring the angular velocities and orientations.
Vision-based sensors	CCD / CMOS cameras	Cameras are recording the environment via the light that is reflected by objects that are in the Field of View of the camera sensor. They enable intelligent interaction in dynamic environments.
Active ranging sensors	Ultrasonic, laser rangefinder, Optical triangulation	Active ranging sensors generate precise distance measurements between the sensor and the target

2.3. Background

In the robotics field there are different kinds of robots that have to follow trajectories. The robots from table 2.3 use visual control by using monocular cameras. Visual control is a method to use a camera as an input to control the motion of the vehicle. Every robot uses their own computer vision method to extract the path that has to be driven. The first two represent mobile robots that can turn without requiring a linear velocity. It means that they are holonomic and that they can manoeuvre easily to their path for when they make a mistake. Besides, their environment is indoor with controlled lighting conditions. In addition, the background on where the robot drives on is homogenous white. The third example is a Autonomous Vehicle that applies a Hough Transform [5] to get straight lines from road marks, other example are found in [11] [13] [4] [25] [14]. The Hough transform is a feature extraction technique that is able to extract straight lines from an image. These straight lines are representing the road markings. Using the lines, the vehicle is able to calculate its relative position with respect to the line such that the vehicle can keep its own lane. The fourth robot is somewhat similar by also using edge detection and the hough transform. However, this robot has to follow one 'line' instead of multiple lines. The last robot uses a different approach. It segments individual tea rows from the background by using semantic segmentation. Semantic segmentation is segmentation on pixel level. By using this method, they know which pixels are labelled as tea rows and which pixels are labelled as background. From the tea rows, they can create a path that is driven by the robot itself.

Table 2.3: Trajectory following robots

Robot type	Path type	Computer vision method
Mobile robot from M.A. Putra et al. [35]	Yellow taped path, straight or curved	The robot extracts the path using a filtering technique based on colour thresholds. It means that yellow pixels are labelled as path and other coloured pixels are labelled as background.
Mobile robot from A. H. Ismail [17]	White taped path, straight or curved	The robot extracts the path to use histogram equalization to even out the illumination in the image before detecting the line based on colour thresholds. They label with pixels as path and other coloured pixels as background.
Autonomous Vehicle from A. Mammeri, and G. Lu, and A. Boukerche [28]	Road marks, straight	First they blur the image; secondly, they select the Region of Interest (ROI); thirdly, they apply Inverse Perspective Mapping (IPM) to create a bird eye perspective. Finally, they extract the road marks by using edge detection and the Hough Transform [5].
Remotely operated Vehicle (ROV) from M. Narimani [29]	Pipes and electrical cables, straight	They extract the pipelines and electrical cables using edge detection and by using the Hough Transform. This is possible because of the high contrast and the straightness of the pipelines.
Tea Field machine from Y. K. Lin and S.F. Chen [23]	Tea bushes, straight	They use semantic segmentation to label each pixel individually. They segment between persons, background and tea bushes. The bushes are straight such that they can use a combination of edge detection and the Hough Transform to extract the paths from the tea bushes.

The robot from Lely is in some aspects different from what has been found in the reviewed literature. The three following aspects are different: the trajectory, the background and the lighting conditions. First, the trajectory differs the most. It is more random than the other cases and the trajectory is physically attached to the robot. The first two examples from table 2.3 do have random trajectories, but because of their ability to manoeuvre on their spot makes it easier to follow the path precisely. The other three examples have trajectories that are either straight or slightly curved. The hose from the Lely robot can lay straight or curved. In addition, the robot from Lely is non-holonomic. It means that the robot is unable to correct its rotation without a linear velocity. When the position of the vehicle is slightly off, it will result in a hose that shifts. When the hose shifts, the new trajectory is shifted as well. It means that the robot will follow a slightly new trajectory, but with an additional error in it. As mentioned above, this behaviour is not preferable. So, compared to the examples found in the reviewed literature: this robot can change its own path that should be followed, whereas the other robots are not able to do so. The other two differences are only affecting the detection of the trajectory. The hose lies in a grass field, meaning that the hose can be occluded by long grass. Our method to detect the hose robustly can not use colour thresholds, because of uncontrolled lighting conditions due to weather changes. The colours are also affected by shades of the robot itself. As mentioned in the table, the hose lies randomly. It means that the hose can be curved such that the Hough transform can not be used to find the trajectory of the hose. A method such as the one found in the last example from table 2.3 is suitable, because it can detect the hose on a pixel level. Meaning that the trajectory can be filtered from its background even though the background is not homogeneous.

2.4. Research question

To recap: the robot from Lely should detect and follow the hose. This thesis will research the possibility to use a form of visual control such that the robot can follow the trajectory of the hose. This leads to the following research question: *"With what level of accuracy can the implementation of visual control using a combination of segmentation, clustering and polynomial regression be used, as input for a PID based vehicle controller?"* The methods of segmentation, clustering and polynomial regression are used to give a representation of the hose, that can be used as an input for the vehicle controller. This research will oversee the integral implementation of the proposed method, meaning that it will look to the implementation of the entire vision pipeline. A common finding in the reviewed literature it that methods such as semantic segmentation are presented as standalone methods. But, the implementation and everything that has to happen after the segmentation before it can be used, is usually not presented. This research looks from a different perspective. It means that it looks further than only a performance score. It will research the possibility to implement such a network to use as an input to control the motion of the vehicle.

2.5. Outline

This thesis is split into two parts. The first part is more theoretical, discussing the entire vision pipeline and the framework around the vehicle controller. The second part will validate the performance of the entire method. The content per chapter will be as follows. Chapter 3 highlights how the image acquisition is done such that the robot gets an overview of its surroundings. In addition, the position and the calibration of the camera is described. Chapter 4 focuses on the vision pipeline. This chapter will explain how the raw images are used such that the information of the hose is extracted. This will be done on the basis of semantic segmentation to classify the hose in the image. A clustering technique will be explained in order to filter the noise from the output of the segmentation. Subsequently, a cluster with the highest likelihood is chosen to represent the hose. Then the filtered cluster is fitted using polynomial regression such that the trajectory is represented by one polynomial. From this fit, a set of points can be selected that will be the input for the vehicle controller. This is described in chapter 5. Before this input can be used to control the vehicle, the hardware modelling is described. The hardware modelling consists of a brief description of the vehicle dynamics and how the vehicle controller is deployed. In chapter 6, performance metrics are set up and a method is described to test the performance of the robot. This is divided into two parts. First, the performance of the segmentation model is compared with results from examples found in literature, using the Mean Intersection overFtlkjsdf Union (MIoU). Secondly, the performance of the hose following capabilities is tested.

3

Image acquisition

A robot can be modelled in many ways, the generalized model is called the **Sense**-Think-Act model [37]. An autonomous robot will loop through these three elements continuously. This method aims to run at 10 Hz, meaning that it will loop through this element every 100 ms. When the robot starts, it starts by sensing its environment. In order to get an overview of its surroundings, a robot usually uses multiple sensors to do so. With the raw data, the robot will be able to think and plan what to do. Then, it acts on its environment. This chapter will discuss the 'Sense' element. This includes the explanation of the sensor choice and an explanation on how the raw data from the sensor is used to let the robot Think and Act. An overview of the robot model is found in figure 3.1.

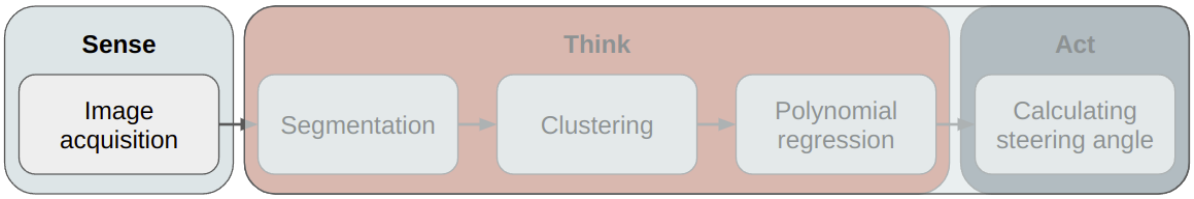


Figure 3.1: The flow scheme of the **Sense**-Think-Act robot model

3.1. Sensor requirements

While the robot drives backwards it has to continuously sense its environment. The raw data from the sensor should be sufficient such that the trajectory of the hose can be filtered from the raw data in the 'Think' stage of the robot. The information about the hose should be filtered from its background. As mentioned in the introduction 2, fertilizing happens when the grass is cut. It means that the background, in its final application, will mostly be short grass. However, the robot will sometimes encounter tall grass, for when it is not cut. In addition, the robot is tested at a testing facility that did not provide a grass field. The robot is, during development, tested by driving over concrete and asphalt. So, the robot should be able to drive over different backgrounds. That means that the method of detecting the hose should be robust against different backgrounds. It is also mentioned that the robot has to work continuously, so the robot should detect the hose at different times of the day. It means that the detection should be robust against changing weather conditions.

3.2. Sensor choice

The video from the camera should contain the hose in order to let the robot detect, track, and follow the hose. For this purpose a monocular camera is used, as mentioned in the introduction 2. The downside of using a monocular camera is that it does not give depth information about objects. Yet, the information of the X and Y coordinates are required for following the hose accurately. Figure 3.2

shows how the coordinates from object P(X, Y, Z) are mapped to pixel coordinates x and y (x and y are also written as u and v, respectively). Point P is in free space, and is mapped to point p on the image sensor (plane). Object P reflects light that will go through the lens of the camera at the centre of the projection. Point P is able to move over the line d without changing the pixel coordinates. It shows that depth information of point P is lost.

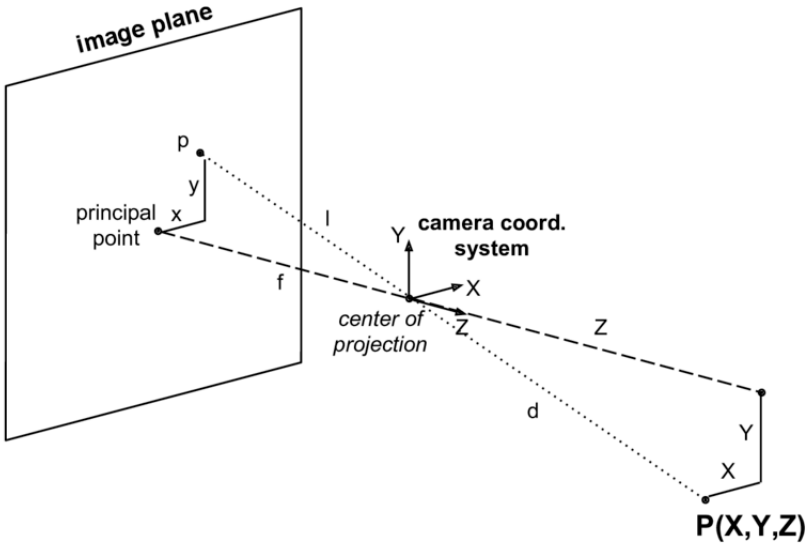


Figure 3.2: This figure shows how an object is mapped from real coordinates to pixel coordinates when the sensor is parallel to the object it is looking at. Image from [40]

With a few assumptions, it becomes possible to acquire the X and Y position of the hose using a setup that uses a single camera.

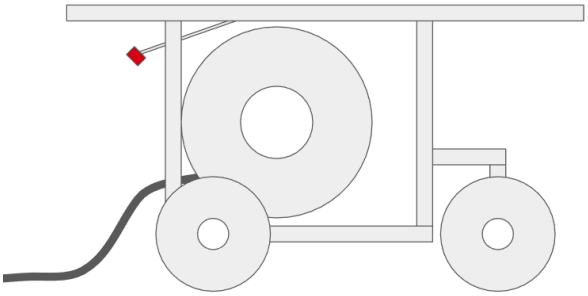
- The camera should stay at a stationary position and rotation with respect to the vehicle.
- The height and rotation of the Z-plane (the ground) should be constant with respect to the vehicle. When taking the first assumption into account, the Z-plane is also constant with respect to the camera.
- The height from the object and the Z-plane should be constant.

In our case, all the assumptions are met, with one exception. The hose leaves the vehicle above the ground, and drops straight towards the ground. From the point where the hose has contact with the ground, this method is viable. In order to explain how the X and Y coordinates are acquired from the image, a detailed explanation of the camera setup should be given, together with an explanation about how the image should be transformed before the X and Y coordinates can be extracted.

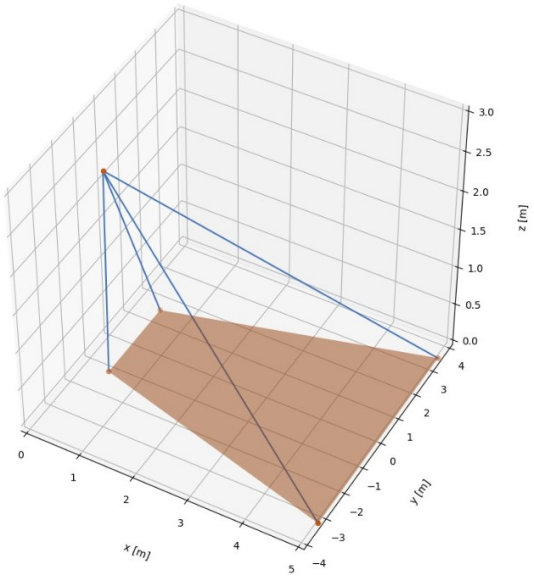
3.3. Camera setup

A visual representation of the robot is depicted in figure 3.3. Keep in mind that the scale and size of the image is not accurate. The hose leaves the vehicle at the rear side. The camera is represented by the red rectangle. The camera is positioned 2.28 meters above the ground, and has an 1.59 meter offset from the rear axle. The Field of View (FOV) shown in the right image. The FOV is depends on the following parameters: the Vertical Angle of View β and Horizontal Angle of View γ of the camera, the position $(x_{cam}, y_{cam}, z_{cam})$ of the camera, and the pitch angle α of the camera. The roll and yaw angle are both zero in our case. The values used are shown in table 3.1:

These specific camera parameters come from the OAK-D camera. This camera, from LUXONIS HOLDING CORPORATION [9], is used in this research. This camera is build specially for industrial AI



(a) A simple visualization of the vehicle. In red: the position and rotation of the camera



(b) The orange plane is the estimated FOV of the camera

Figure 3.3: Visualization of the camera's position and what the FOV includes

Table 3.1: Camera parameters of OAK-D camera

Parameter	value
α	40°
β	45.3°
γ	68.8°
x_cam	-1.59 [m]
y_cam	0 [m]
z_cam	2.28 [m]

applications. In retrospect, it has been discovered that the capabilities of the camera were not sufficient for the semantic neural network. In our case, it is only used to encode images to easily sent the images using an Ethernet connection.

Described by H. Kano et al. [18], there is a relation between the pixel coordinates u and v and the real coordinates X, Y and Z of objects occurring in the image. However, because the camera is tilted by angle α, this relation is not one to one. Due to the vanishing point that occurs when the camera is tilted, objects that are further from the camera appear smaller. This phenomenon is a self-explanatory, but it should be counteracted to create a mapping such that the pixel coordinates map the real coordinates with a simple linear relation. The following transformation is used: the FOV near the camera is squeezed, and the distant FOV is stretched. After the transformation, the transformed images do have a bird's eye perspective. An example is given in figure 3.4.

Figure 3.4: Visualization of the bird's eye perspective

The purpose of the transformation is to normalize the real distances such that x amount of pixels maps to x meters, independent of where the pixels are located in the image. Figure 3.4 shows an example of the transformation applied to real data. The tiles on the left images are squares, but they appear as trapezoids. The transformed image displays the tiles as squares. So, there is a mapping between the pixel coordinates (u, v) and the 3D world coordinates (X, Y, Z). This is done by using the projective transformation matrix P. The P matrix is composed out of the intrinsic camera matrix and the homogeneous transformation matrix. The vector of the pixel coordinate correlates to the dot product between the P matrix and the position vector of the object in the following way:

$$\begin{bmatrix} u & v & 1 \end{bmatrix}^t \approx P \cdot \begin{bmatrix} X & Y & Z & 1 \end{bmatrix}^t$$

The projective transformation is composed out of the intrinsic camera matrix and the homogeneous transformation matrix. The intrinsic camera matrix consists of the focal length in x and y direction. That is the length between the lens and the image sensor. When the camera does not focus, it means that the focal length is constant. The intrinsic camera matrix contains a scaling factor s, and the coordinates of the centre of the image (x_0, y_0). These values are fixed when the camera is calibrated. The homogeneous translation matrix is composed out of the rotation matrix and the translation matrix. The homogenous transformation matrix describes the position and rotation of the camera with respect to the vehicle. The P matrix can be written as:

$$P = \begin{bmatrix} f_x & s & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} cos\alpha & 0 & sin\alpha & x_{cam} \\ 0 & 1 & 0 & y_{cam} \\ -sin\alpha & 0 & cos\alpha & z_{cam} \end{bmatrix}$$

The hose of the robot lies on the ground, as mentioned in the assumptions from section 3.2. It means that every point from the hose lies on the Z plane. Such a point can be described as a 3D coordinate in free space. But also by a 2D coordinate (x, y) that lies on that Z plane. This is described by the dot product between the Q matrix and the 2D coordinate. The **i** and **j** vectors are both unit vectors that describe the direction of the Z plane. **d** Is the origin of the Z plane.

$$\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = Q \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} i_x & j_x & d_x \\ i_y & j_y & d_y \\ i_z & j_z & d_z \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Using the above formulas will create the Homography Matrix H. This final formula represents a linear correlation between real coordinates and pixel coordinates. It is used to calculate the bird's eye perspective seen in figure 3.4. So, with the following formula it is possible to calculate distances from the image.

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \approx P \cdot \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = P \cdot Q \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = H \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

3.3.1. Camera calibration

The relation between the pixel coordinates and the real world coordinates, which is described in section 3.3, holds only for when the camera is calibrated. It means that the distance information will be off for when the camera is not calibrated. Therefore, a calibration of the (intrinsic) parameters should be done before performing the perspective transformation to create a bird's eye view. There are different distortion types for when the camera is not calibrated. They are displayed in figure 3.5. Barrel distortion and pincushion distortion are both radial distortion types. A combination of the barrel and pincushion distortion is called the moustache distortion. There are also tangential distortion types. These distortions occur when the lens has a slight offset or when the lens is not parallel to the image sensor.

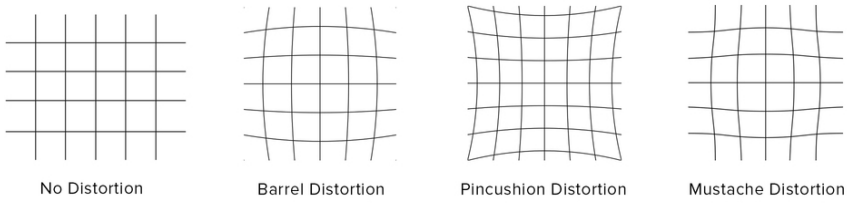


Figure 3.5: Three different types of image distortion. Image from: [16]

The radial and tangential distortions are described by the mathematical models from Bronw-Conrady [8]. They are displayed in table 3.2. u' And v' are the distorted pixel locations.

Table 3.2: Mathematical models for different distortion types

Distortion type	Mathematical model	Coefficients
Radial	$u' = u(1 + k_1r^2 + k_2r^4 + k_3r^6)$	k_1, k_2, k_3
	$v' = v(1 + k_1r^2 + k_2r^4 + k_3r^6)$	k_1, k_2, k_3
Tangential	$u' = [2p_1uv + p_2(r^2 + 2u^2)]$	p_1, p_2
	$v' = [2p_1uv + p_2(r^2 + 2v^2)]$	

By calibrating the intrinsic parameters of the camera, an estimation of the coefficients is made such that the pixel coordinates are correctly mapped. This is done by taking images from a reference object using the sensor that should be calibrated. In figure 3.6 the object is shown that is used to calibrate the OAK-D camera. The calibration method detects the centres of the black circles. A grid of lines can be made by connecting all the centres of the circles. When the sensor is not calibrated, this grid can show a distortion, as displayed in figure 3.5. The formulas from Bronw-Conrady are used in an iterative fitting process such that they can describe the lines of the found grid. When this method is applied to multiple images, a precise estimation of the coefficients is found. The correct coefficients are stored such that the calibration is performed once. The calibration is done by using the OpenCV

library [27]. This program calculates the coefficients and the adjusted camera matrix after it has seen multiple images from the checkerboard. OpenCV claims that an accurate calibration requires at least 10 images. S. Placht [33] validated the performance of the calibration by using 1 to 80 images and concluded that about 20 images is sufficient for a proper calibration.

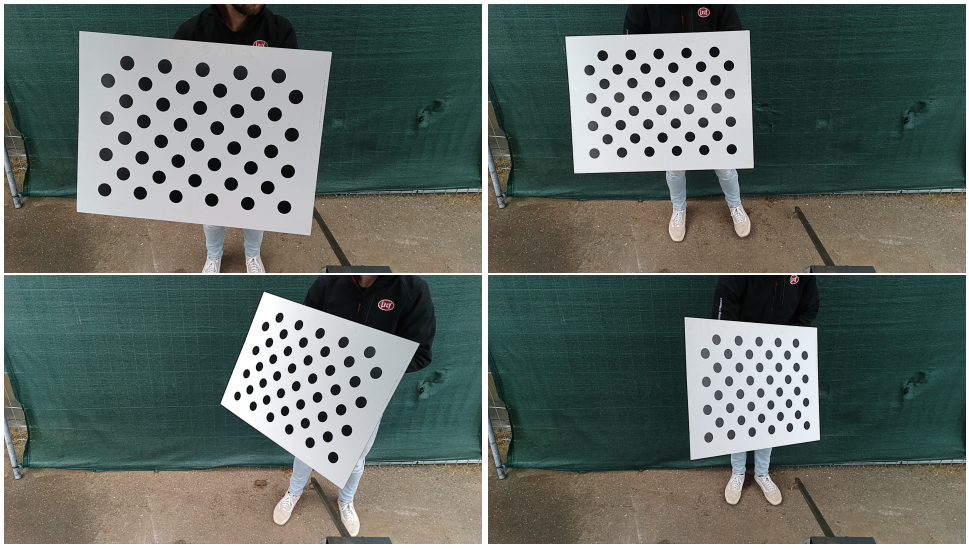


Figure 3.6: 4 Examples of calibration images used to calibrate the intrinsic parameters of the OAK-D camera

The camera setup can be used in the following way, once the calibration is performed. The recorded image is undistorted using the mathematical model with the correct coefficients. After which it goes through the inverse mapping transformation. When these steps are accomplished it becomes possible to extract the hose coordinates from the image.

4

Vision pipeline

After the robot has sensed its environment, it should interpret the information. This chapter will describe the next stage from the Sense-**THINK**-Act robot model. It describes the method that has been developed in order to gain logic from the images, such that it creates an input for the 'Act' stage. In other words: the image contains information on a high level and the vision pipeline will refine and filter the information, such that the quantity is downscaled and the quality is upscaled. The vision pipeline implies a process that consists of consecutive steps. The main three steps are shown in figure 4.1.

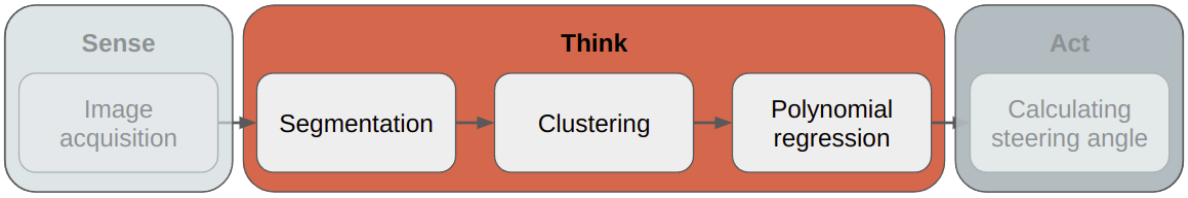


Figure 4.1: The flow scheme of the Sense-**Think**-Act robot model

4.1. Semantic segmentation

The vision pipeline starts with a method that detects the rough trajectory of the hose. The contrast between the hose and the background is rather high. For the human eye, it is not hard to distinguish the hose based on its shape and colour. The hose is black, and the background is either green from the grass or gray-ish from the concrete. The conducted preliminary research concluded that simple computer vision techniques like detecting the hose based on colour will work under normal conditions. However, these techniques will lose their accuracy with more demanding conditions. Object detecting by using bounding boxes is straightforward, but it is not sufficient because it will only tell where the hose is roughly positioned in the frame. The total trajectory of the hose is required to follow it accurately. The starting position and the end position should be known, but also the heading of the hose between these two points. So, the trajectory of the hose within one frame is the total position and rotation of each segment of the hose. Semantic segmentation is used, because it can label individual objects. That means that the entire shape of the hose can be extracted from the background. In our case, this implies that every pixel in the image will be classified either as background or as hose. Examples of semantic segmentation models are: Fully Convolution Network (FCN) [24], SegNet [3], and Deeplab [7]. The FCN is one of the earlier segmentation models. Therefore, the accuracy is not high according to the authors of A. Garcia et al. [15]. The architecture of the SegNet network is based on the VGG-16 backbone and decoder. The general architecture is also visible in figure 4.2. The layers seen in green on the left side are part of the backbone, which is also called the encoder. The decoder is visible on the right side of the figure. Deeplabv3+ can use different backbones, such as Xception, ResNet50 and MobileNet [10]. Deeplab and SegNet are both models that can reach high accuracies, an example can

be found in the paper which writes about cattle classification, written by D.A.B. Oliveira et al.[30]. Lely Technologies works with a version of Deeplab as well. They use Deeplabv3+ in combination with the Xception backbone with pre-trained weights. According to D.A.B. Oliveira et al., “The use of pre-trained weights can be an effective strategy to speed up the training process and improve network convergence, especially when given small datasets” [30]. According to the survey of F. Lateef and Y. Ruichek [22] Deeplabv3+ is one of the few semantic segmentation networks that are able to achieve MIoU score close to 90%. The Mean Intersection over Union is a score based on how many pixels are labelled correctly. The SegNet neural network comes close to the performance of Deeplabv3+. Both the models are compared with each other by using the Pascal VOC dataset [12]. This comparison includes lantern posts as well, because most of the images are from AVs. When the output from Deeplabv3+ [6] and SegNet [3] is compared with each other, it becomes visible that Deeplabv3+ labels the lantern posts slightly better. This conclusion is important, because lantern posts are shape wise comparable to the shape of the hose, because of their thin features. It means that Deeplabv3+ will be more suitable for detecting the hose due to its overall high accuracy and to its higher accuracy in detecting similar shapes.

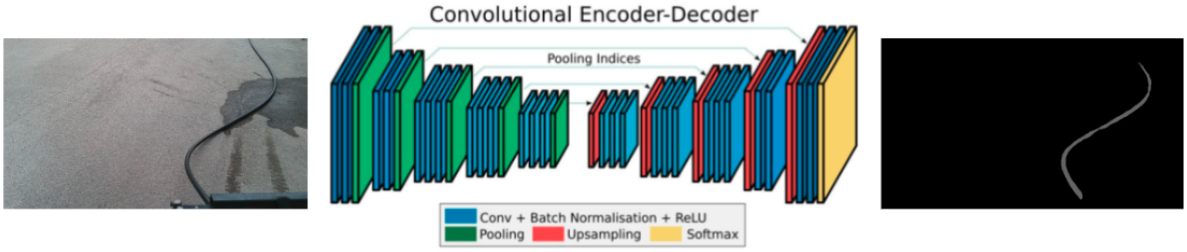


Figure 4.2: Example of a semantic segmentation network. The middle image is from [21]

4.1.1. Image resolution and inference time

The images that are shot by the OAK-D camera have an HD resolution, meaning that they have a width of 1920 pixels and a height of 1080 pixels. Although the network is capable of segmenting large images, the inference time will be low due to the 2 million pixels that have to be classified. The inference time is the time it takes the segmentation network to classify an entire image. Preliminary research showed better results with scaling the image equally. The image is downscaled with a factor of three, in order to decrease the inference time. The resolution now becomes 640 × 360, meaning the network has to classify only 230 thousand pixels, which is a factor nine less. Both the width and the height are scaled with the same factor. The width of the hose that appears in the image is around 10 pixels. The semantic segmentation is done on an industrial computer. With the smaller images, an average inference time of 35 ms is achieved. The industrial computer contains the following configuration: Intel Core i7-7700 CPU @ 3.60 GHz × 8 threads, 16 GB of RAM, Nvidia GeForce GTX 3060 GPU with 12 GB of RAM, 256 Gb mechanical hard disk, NVIDIA driver version 390, CUDA version 11.5.6, CUDNN 8.1 neural network acceleration library, Linux Ubuntu 20.04 LTS operating system, Python version 3.9, and TensorFlow version 2.1.0.

4.1.2. Data acquisition

As explained above, the used network is trained together with a backbone that already has pre-trained weights. That means that the model is already trained with a large set of images. The benefit of using a pre-set network, is that the dataset that is required to train the model can be smaller. This is beneficial because acquiring data is a time-consuming task. The process of gathering images contains a few steps: choosing the camera position, gathering images, labelling the images and augmenting the data. In the coming subsections, these steps will be explained and how they are executed in our method.

4.1.3. Camera position

Choosing the position and rotation of the camera is crucial to do before the images can be used for training the neural network. The position and rotation have a direct influence on the FOV. A large FOV has the benefit that it will give more information about the trajectory of the hose further away from the vehicle. The downside is that it gives a larger perspective, resulting in a smaller appearance of the hose in the image. Due to the small appearance, it becomes harder to detect the hose by the neural network. This problem does not occur when the image has a small FOV. The downside of having a small FOV is that the machine only knows the trajectory close to the vehicle. The speed of the vehicle becomes limited by using a smaller FOV. This limitation comes from the reaction speed of the vehicle. This theory is clarified with an example: when the hose lies on the field with a sharp s-curve, it is not preferable to perfectly follow that trajectory. The effort of the robot will be relatively high for when it follows this curve accurately. Besides, it is questionable if the robot reacts fast enough to make such a sharp corner. When the robot has a larger FOV, it could see a larger part of the s-curved section, meaning that it can calculate a route in advance. The calculated route can be 'soft', meaning that it can straighten the path of the hose. Lastly, with a large FOV, the angle of the camera becomes rather low, meaning that light incidence starts to play a role when the robot operates during dawn or dusk. (In)direct light can overexpose the sensor, resulting in a situation where the camera can not detect anything.

A balance is found between a large and small FOV, as seen in figure 3.3b. The camera has a FOV up to 5 meters, ranging from the position of the camera. It means that the robot has information about its environment up to 6.5 meters from the rear axle of the robot, when accounting for the camera offset of 1.59 meter. Figure 3.4 shows an example of what the FOV would look like. The robot has a maximum speed of 1 meter per second, meaning that the robot has a maximum of 5 seconds to make an emergency decision, which is plenty. Besides, the trajectory is better visible on the images with this smaller FOV.

4.1.4. Gathering images

The neural network requires data in the form of labelled images to train. Therefore, three datasets are created. The first one contains 194 images. These images are gathered in an early state. Therefore, the FOV differs from the other two datasets. The primary goal of this dataset was to see whether the hose could be detected by one of the segmentation networks. In addition, the grass is rather short, which is beneficial for the detection of the hose. During preliminary tests, the robot was stationed at the workshop of Lely. The pavement around the workshop is made from square concrete tiles and asphalt. In order to test the functionalities, it was necessary to create a trained neural network that would perform well around the workshop. Therefore, preliminary tests could be conducted such that the iteration process went rapid. To comply, this dataset has been created. This dataset contains 261 images. The last dataset was created during the final stage of this research, and is therefore the largest dataset, containing 720 images. The final tests are conducted in tall grass, hence the images from this last dataset contain tall grass. The other two datasets both have a sharp contrast between the background and the hose. This last dataset contains images where parts of the hose are occluded by the grass. Meaning that not every part of the hose is visible.

(a) Image from the first dataset **(b)** Image from the second dataset **(c)** Image from the third dataset

Figure 4.3: Example images from the three different datasets

4.1.5. Labelling images

The semantic segmentation used by Deeplabv3+ can be categorized as supervised learning. In its most basic form, this means that the model will be trained with an image together with a ground truth. The ground truth is a bitmap (also called mask), which is equally in size compared to the original image. The bitmap contains a class for each pixel. The process to create the mask is called annotating. The tool used to annotate is CVAT.ai [31]. To annotate the objects for segmentation, the spline tool has been used to draw a spline around the shape of the object. This job should be done with high accuracy, in order to get proper results during the semantic segmentation. The annotated images are used to 'learn' the weights from the neural network during the process of training. When it has seen enough 'known' images, it can give an estimation of the objects by itself.

4.1.6. Data augmentation

As mentioned before, the process of selecting and annotating images is time-consuming. Annotating a single image can take several minutes. Therefore, creating datasets would cost days. There are effective ways to get extra training data without the need of taking and annotating extra images. From the original images, it is possible to augment images. The performance of the neural network is increased by using augmented data, as described by R. Ma, P. Tao, and H. Tang [26]. They tested a total of seven different data augmentation techniques. The techniques are divided in global and local augmentation. An example of a global technique is to make a copy of the image and to flip or compress that copy. Examples of a local augmentation are to crop or to shift the image locally. This process is done both for the original images and for the ground truth. R. Ma, P. Tao, and H. Tang indicates that by using a combination of cropping, local shifting and JPEG compression the highest accuracy gain can be achieved. They went from an MIoU of 73.28 % to a MIoU score of 91.25 %. It does not mean that this combination is ideal for detecting the hose, because they applied the neural network to another problem. However, it indicates that a form of data augmentation helps to improve the performance of the model.

4.2. Clustering

The second step in the vision pipeline is clustering the output of the semantic segmentation. There are two reasons why this is necessary. First, it could happen that there are multiple objects labelled as hose within one image. Secondly, the mask from the segmentation is not perfect, meaning that some parts of the hose are not seen as hose. This will result in gaps in the segmentation of the hose. The gaps of the segmentation can occur because of wrong classification, but also through occlusion when the hose lies in tall grass. Filtering can be used to counter the first problem, and stitching different parts of the trajectory of the hose can be used to counter the second problem. However, in advance it is unknown which of the two techniques should be used. Using unsupervised learning by means of clustering is a way that can tackle both of the problems. There are different forms of clustering data points. D. Xu and Y. Tian [42] made an overview of the most common clustering techniques, which are visualized in figure 4.4. This image shows 10 different clustering algorithms that are able to cluster 2D data points. In the figure, the individual data points are not that close to each other. Looking at our case, every pixel which is labelled as hose can be seen as a data point, meaning that the pixels will be clustered.

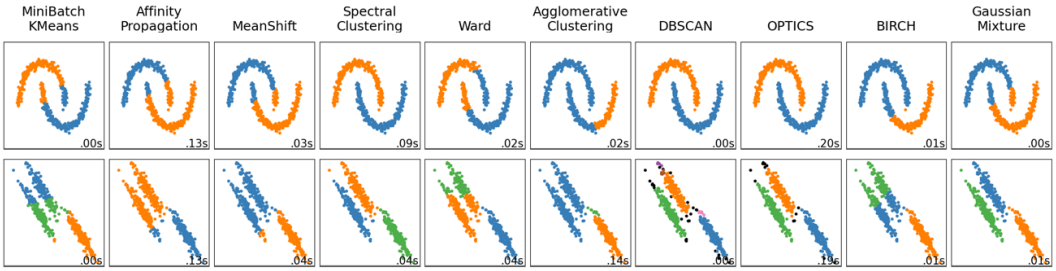


Figure 4.4: Different clustering algorithms within the Scikit-learn library. Image taken from [32]

A shape comparison is made between our case and the different cases seen in figure 4.4. The similarities are found in the stretched and thin clusters. In our case, the goal is to cluster adjacent pixels that form a long stretched cluster. By using DBSCAN or OPTICS, the best performance is achieved. However, the computational time of DBSCAN is 20 times less than with OPTICS, therefore this technique has been selected. The Density Based Spatial of Applications with Noise searches for the core of a high density in the spatial data. Thereafter, every density is expanded by neighbouring pixels, creating the individual clusters. Two examples are given to show how the clustering technique performs. They are shown in figure 4.5 and figure 4.6. In both examples, one can see the original image in the left upper corner. The right upper corner shows the segmentation mask received from Deeplabv3+. The left bottom corner shows the clustered output, every colour represents a different cluster. The right bottom corner shows a polynomial. The purpose of the polynomial is explained in the next section. The first figure is representative by means of a correct segmentation. The second example is taken from a poor performing neural network. Nevertheless, both examples show how powerful the clustering can be, by stitching and filtering the segmentation mask with high accuracy.

4.2.1. Cluster selection

When the robot finds more than one cluster (as in figure 4.6), it has to choose what the real representation is of the hose. This can be done by giving every cluster a weight: the cluster with the highest weight has the highest likelihood of being the hose and is therefore chosen. In advance, there is always one thing certain about the hose, which is that the hose always starts at the bottom of the images. This fact holds for every normal case: when the hose contains a sharp corner, and when the endpoint has been reached. Besides, when driving in a field without concrete tiles, the hose is likely to be the largest and longest object in the output of the segmentation. Therefore, all the objects are excluded that do not start at the bottom of the image. Then the longest cluster is chosen. In both examples, the hose is correctly found.

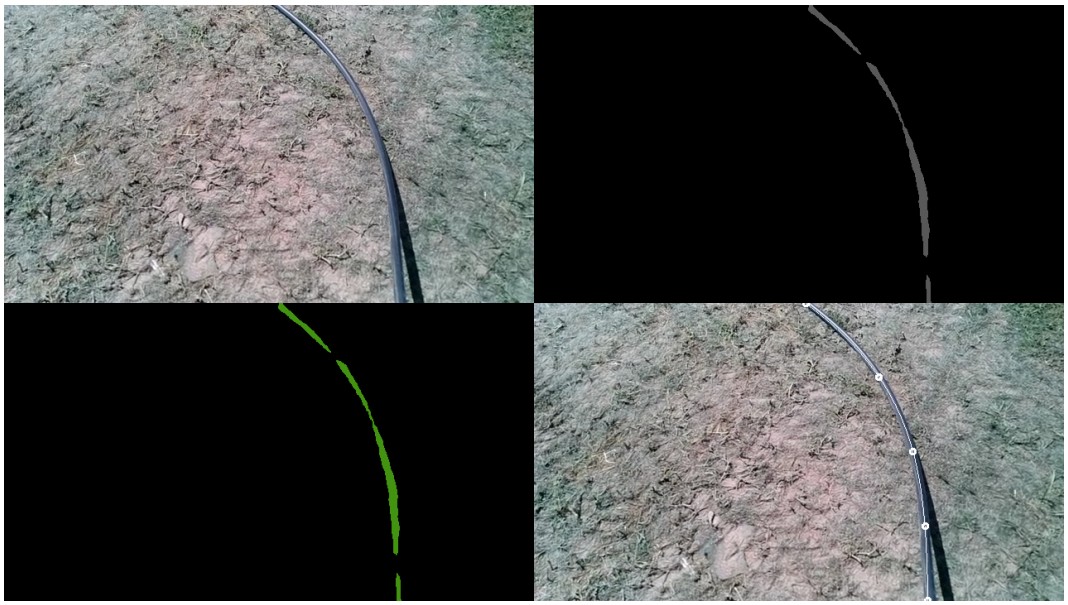


Figure 4.5: First example of the vision pipeline. From the left upper corner to the right bottom corner: 1. The original image; 2. The segmentation output; 3. The clustered output; 4. Overlay with a regression polynomial

Figure 4.6: Second example of the vision pipeline. From the left upper corner to the right bottom corner: 1. The original image; 2. The segmentation output; 3. The clustered output; 4. Overlay with a regression polynomial

4.3. Polynomial regression

Clustering does not mean that the gaps disappear between the individual parts of the cluster. The gaps are also visible in figures 4.5 and 4.6. In both examples, the gaps are relatively small, and the individual pieces of the hose are relative long. These gaps can occur with higher frequencies. It happens when there is occlusion due to tall grass, or when the segmentation fails to segment the hose. In order to compensate for the gaps, the cluster will be fitted using a polynomial.

The reason that the hose can be described by a polynomial lies in the behaviour of the hose itself. There are a few certainties that will always hold true: the hose starts at the bottom of the image; the hose has a stiffness to it, meaning that it cannot lay in a shape that contains a buckle; the hose cannot stop and start somewhere else in the image, it is always connected. Because of these conditions, there is a relation between the x and y locations of the pixels that are representing the hose. This relation can be described using a mathematical model. There are a lot of regression models, although most of them are linear. Except for the polynomial regression. Linear regression or polynomial regression by using a 1st order polynomial is not suitable, because it would oversimplify the trajectory of the hose. Polynomial regression as described by D. Polzer [34] will give a result that can be seen in the right bottom image, seen in figure 4.5 and 4.6. Multiple orders of polynomials are tested, it was found that lower order polynomials would result in an underfitted hose, because a first or second order polynomial would result in a linear line or parabola, respectively. Both of them would only able to describe one kind of trajectory. 3rd Or 4th order polynomials are suitable because they can describe the curvatures of the hose properly. It has found that higher order polynomials would result in an overfitted polynomial. It means that polynomial does not lay exactly on the hose, but it would oscillate over the hose instead.

The polynomial is the representation as if the hose did not contain gaps. It is possible to extract five points from this polynomial. The y value of the most upper and lower pixel from the cluster is used to get the corresponding x value from the polynomial. Between these two points, three other points are extracted from the polynomial. These are shown in figure 4.5 and 4.6. These 5 points are the output of the vision pipeline, and will be the input for the vehicle controller to control the motion of the robot.

5

Hardware modelling

The vision pipeline creates an output that contains five points, which represents the trajectory of the hose. This chapter will explain how these five points are linked to the vehicle controller such that it will control the motion of the vehicle. That is done by describing the vehicle dynamics and by describing the calculating of the steering angle. These two are linked together in order to implement the PID that will control the velocity and direction of the vehicle.

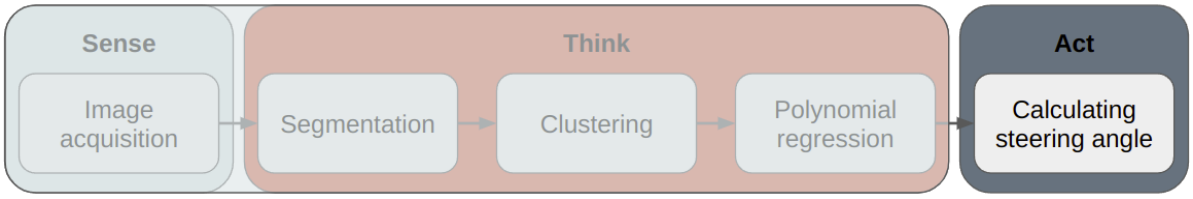
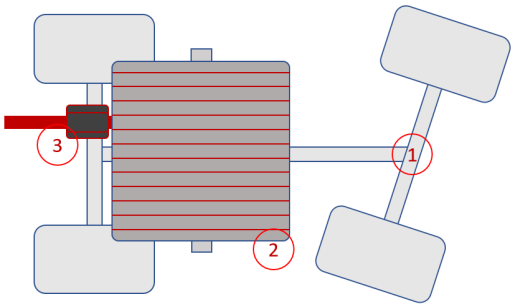


Figure 5.1: The flow scheme of the Sense-Think-**Act** robot model

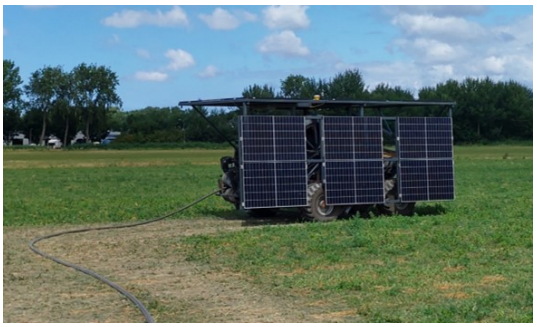
5.1. Vehicle dynamics

In figure 5.2 a top view of the robot is displayed. The right side of the figure shows the front of the vehicle. The rear side of the vehicle is on the left side of the figure. Number 1 represents the pivot point of the front axle. Steering of the front axle is done by rotating the wheels in the opposite direction. When the robot is positioned as shown in figure 5.2 and steers to its neutral position the following will happen: the left front wheel will rotate counterclockwise, and the right front wheel will turn clockwise. When the wheels turn in that specific rotation, the entire axle starts to rotate by angle α. The benefit of such a steering mechanism is that it can steer well without a linear velocity. In addition, the field underneath the wheel of the robot will not be damaged. Number 2 stands next to the spool of the robot. This spool contains up to 250 meters of hose. The hose is neatly coiled such that the total length of the hose fits the size of the reel. This is done by mechanism 3. Three rollers are holding the hose tight, and they put tension on the hose section between the spool and the rollers by adding a torque to the rollers. This tension is required, because otherwise the hose on the spool would unroll due to the stiffness of the hose itself. The rollers help to position the hose while reeling it in. This is made possible by a mechanical connection between the rollers and the hose such that the rollers are moving when the reel is moving. The rollers move from left to right (in the image: from top to bottom). With every revolution of the reel, the rollers are moved by one hose thickness. The rollers move by one hose thickness, such that the hose neatly joins the other hose windings. In chapter 4 it is described that the hose could start over almost the entire bottom of the image. The reason why it starts at this spot is due to this tension mechanism.

Furthermore, it is important to notice that there exists a relation between the velocity of the vehicle and the rotational velocity of the spool. This relation depends on different parameters. When the vehicle



(a) Schematic representation of the top view of the robot. The image is not to scale.



(b) Image of the robot in a field following the trajectory of the hose.

Figure 5.2: Schematic and real image from the robot

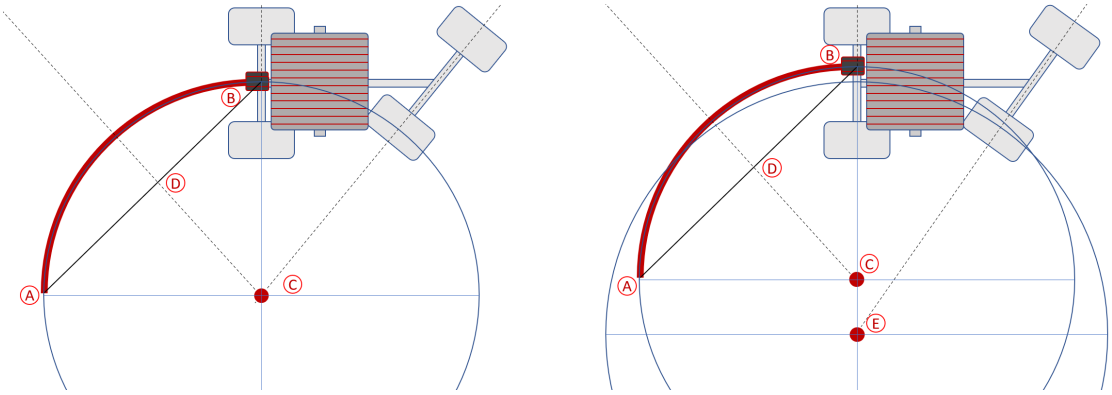
drives forward in a straight manner, only the linear velocity and the radius of the spool are factors that create this relation. However, the wheels of the vehicle can slip and the radius of the spool can differ due to the remaining length of the hose on the spool. When the robot drives along a curve, the length of the curve starts to play a role. The position of the rollers comes into play as well. Not all of these parameters are measured with high accuracy. When for example one of the wheels slip the correlation is off and therefore it does not make sense to come up with the exact correlation between the velocity of the vehicle and the rotational velocity of the reel. The reason why this correlation is required, is that the hose should lay in the position where it laid down in the first place. Just before the robot will shift the hose, the robot already exerts force on the hose. Therefore, a sensor is added to accurately measure this force. With this output, the rotational velocity is constantly adjusted such that the hose does not start to shift. This does not exclude that the hose cannot shift any more. When the robot drives into a wrong direction, the hose will still shift. This method deals with the hose shifts due to a different linear velocity of the vehicle and a rotational velocity of the reel.

5.2. Theoretical steering angle

The centre of ration is required to know how much the vehicle should steer. The centre of rotation depends on how the hose lies and how the steering wheel is positioned. These two factors should be connected with each other, such that the robot can follow a trajectory. Figure 5.3a shows a situation where the hose lies along a perfect circle. The robot can pivot the front axle by -45 degrees to 45 degrees. The distance between the front and rear axle is two meters and 55 centimetres. This means that the robot has a minimum turning radius of 2.55 meters. However, the hose is too stiff for such a small radius. The practical steering radius has a limit around five meters.

Three dotted lines are shown in figure 5.3a. The right dotted line is an extension of the front axle. The central dotted line is an extension of the rear axle. They intersect at point c, which is the centre of the turning circle or centre of rotation. When the wheels of the robot do not slip and if the front axle would keep its angle as displayed, it would drive along the circle that is shown in the figure. A similar centre of rotation is drawn for the trajectory of the hose. This is done by drawing the chord between point A and point B. The line between point D and point C is the perpendicular bisector of the chord and passes through the centre of the arc. The official theorem is written as: the perpendicular bisector of any chord of any given circle must pass through the centre of that circle, which is proofed by T.L. Heath [39]. When both the centres of rotation collide, the robot will accurately follow the hose. However, when they are not colliding such as in the example of figure 5.3b, the robot will not follow the trajectory of the hose accurately.

This method of steering works when both the turning centres collide. However, the reaction time of the robot to correct the steering angle must be instantaneous when driving with a certain velocity. Besides, the curvature of the hose must lay in a perfect circle. Both are not feasible in practical situations. In addition, most of the time it happens that the hose does not leave the vehicle at the centre of the vehicle. This difference is found at point B when comparing figure 5.3a and figure 5.3b. The hose can start almost over the entire width of the vehicle. Figure 5.3b shows that the coil system,



(a) The image shows two colliding centres of rotation. It means that the robot will follow the hose accurately. The image is not to scale.

(b) The image show two mismatching centres of rotation. It means that the robot will not follow the hose accurately. The image is not to scale.

Figure 5.3: Schematic representation to show the centre of rotation

that holds tension on the hose, is shifted upwards. It means that the circle of rotation from the hose is shifted upwards as well. However, the circle of rotation of the vehicle stays similar. Due to variable exit position of the hose, and the earlier told limitations, this method to calculate the steering angle of the front axle is not sufficient.

5.3. Calculating the hose angle

Another method to steer the robot is required in order to counter the problems from the theoretical steering method. This method should work in the following cases: when the hose does not leave the robot at its centre; when the trajectory of the hose is not a perfect arc of a circle. This method uses four segments that are visible in figure 5.4. A segment is created between every two consecutive points. The angle of each segment is calculated between the central axle and the line that represents the segment. A total of four angles is calculated. Thereafter, every angle will get a weight assigned to it. Using the angles and the weights, an average angle is calculated.

Figure 5.4: Curve segments visualized

The benefit of this method is that it enables a range of different control mechanisms that influence the steering behaviour differently. When the weights of the two upper segments (segment one and segment two) are greater than the weights of the lower two segments (segment three and segment four), the robot will steer earlier. This also works the other way around. When the lower two segments weigh more than the upper two segments, the robot will steer based on the track closer to the vehicle. Both of the two weight tactics will lead to a different driving behaviour, as will be explained using the trajectory of figure 4.6. The first weight tactic is expected to average out the s-curve, because the robot

does look less at segment one and two. Therefore, it is expected that the robot will drive in a more straight manner. With the other weight tactic, it is expected that the robot will follow the trajectory with more accuracy. Because it looks more to the lower segments. The first one has the benefit that the robot is 'lazy', meaning that the steering effort will be less. The downside is that the chance on shifting the hose becomes bigger. With the second tactic, these hypotheses are reversed. Both methods are validated during the testing phase of the thesis, in order to see which of the two tactics perform better.

This method of calculating the desired angle requires testing. However, it will solve the problems that are highlighted in section 5.3. Every trajectory of the hose can be described by the four segments, such that non-perfect circular trajectories are possible to follow. Besides, when the robot responds too late due to the delay, it is possible to adjust the weights such that the robot will steer earlier to compensate for the delay.

5.4. Implementing the vehicle controller

This last section combines the parts described above such that the motion of the vehicle is controlled by the vehicle controller. With the known trajectory of the hose, it becomes possible to calculate the reference angle of the front axle. Initializing the controllers is done by using the vehicle dynamics. In addition, the vehicle controllers are limited such that the control signals will stay under the limits that the robot is cable of.

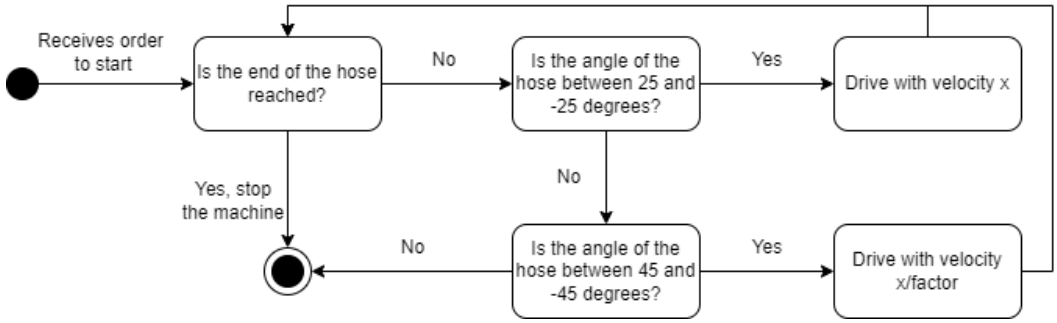


Figure 5.5: State machine of the vehicle controller visualized

A state machine is used to control the desired behaviour of the robot. The state machine is visualized in figure 5.5. The state machine is dictated by the trajectory of the hose. When the hose lies straight, the vehicle can use a maximum velocity threshold, as input for the PID controller. There are two individual cases where the velocity threshold is lowered. The first one is when the average angle of the hose with respect to the robot becomes sharper than 25 degrees. The lowered threshold for the linear velocity results is an added safety measure. It prevents an aggressive stop for when the angle of the hose exceeds 45 degrees. The robot comes to a hold when the angle of the front axle becomes sharper than 45 degrees. To prevent this, the threshold for the linear velocity is linearly lowered such that the robot can stop safely when it exceeds the 45 degrees. It means that threshold is maximum when the angle is between 0 and 25 degrees and the velocity decreases linearly to 0 when the angle approaches 45 degrees. When the robot has found the end of the trajectory, the linear velocity is also lowered slowly such that it comes to a hold without an aggressive stop. The reason why the robot should stop slowly at the end of the hose is because of the docking point that is normally attached to the hose of the robot. When the velocity is reduced, it will also reduce the forces that are exerted on this docking point.

Two different controllers are used to control the motion of the vehicle. The controls for the linear motion are different from the controls for the rotational motion. Both the control schemes are shown in figure 5.6. A Proportional Integral Derivative (PID) is used for the linear velocity of the vehicle, and a single Proportional (P) controller is used for controlling the steering angle. The reference signal that comes from the vision pipeline is expressed as $r(t)$. The error $e(t)$ is calculated using the measured state of the vehicle $y(t)$ by using the following formula:

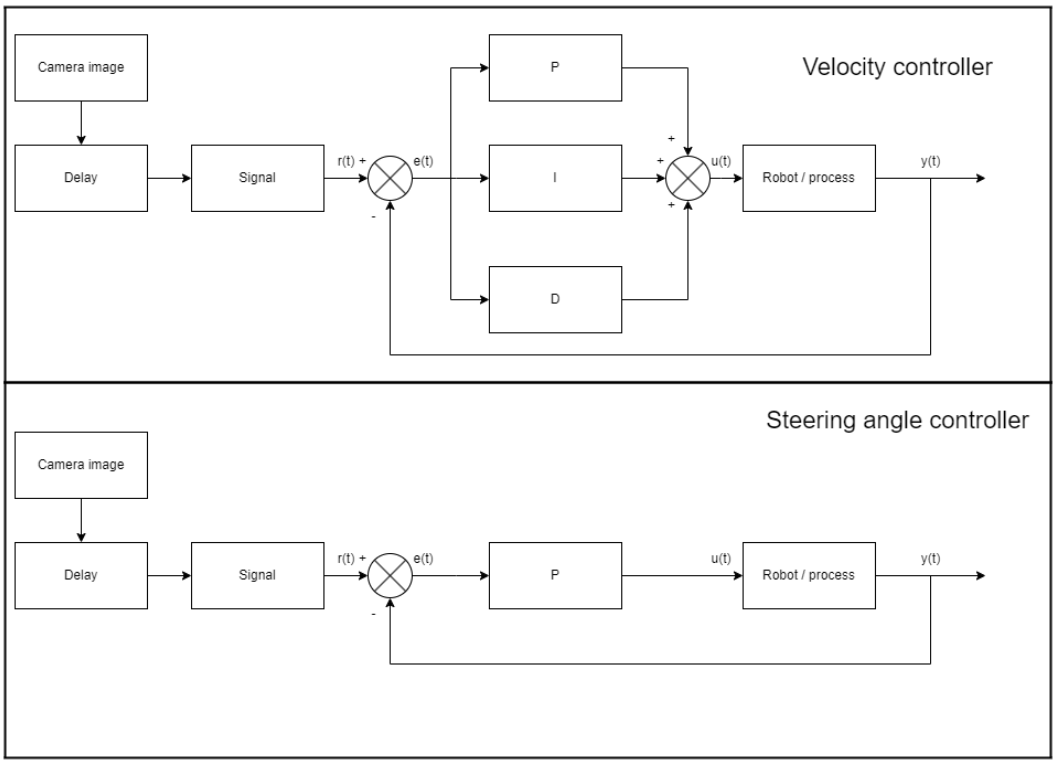


Figure 5.6: The upper control scheme shows the velocity controller of the vehicle controller. The bottom control scheme shows the steering controller.

$$e(t) = r(t) - y(t)$$

The error represents a difference in velocity for the velocity controller and a difference in position in degrees for the steering angle controller. The error is used by the PID or P controller to calculate the desired velocity or position, which is expressed as $u(t)$. The PID controller is calculated in the following way:

$$u(t) = K_p e(t) + K_i \int_t^0 e(\tau)\, d\tau + K_d \frac{de(t)}{dt}$$

The position controller is shorter because it only contains the proportional part:

$$u(t) = K_p e(t)$$

The robot will use the input $u(t)$ to update its state. Thereafter, the update is measured by the various sensors and is looped back in the system. The refresh rate is limited to 10 Hz, which is the refresh rate of the vision pipeline. The controller is chosen for the initial acceleration and de-acceleration, when the vehicle starts or stops. To make sure that it accelerates and de-accelerates smoothly, a PID has been selected to counter the effects of the mass inertia of the vehicle. The difference of the controller originates from what it has to control. The PID controllers control the velocity of a vehicle, whereas P controllers controls the position. The velocity should be controlled smoothly, because the vehicle has a large mass inertia. Besides, the robot together with the reel can act like a mass-damper-spring system, meaning that it can be modelled as a higher order system. P controllers are not sufficient for such a system. However, it is sufficient for a first order system. Controlling only the position of the front axle is simpler, because every state of the steering wheel is seen as steady state. Therefore, it can be modelled as a first order system, making a P controller sufficient. That means that the average angle, calculated based on the trajectory of the hose, is directly coupled to the angle of the front axle by a linear scaling.

6

Experimental methodology

The focus so far was to explain the setup of the robot itself. Chapter 3 focused on gathering images. Chapter 4 focused on the design choices considering the semantic segmentation and the vision pipeline. Chapter 5 focused on the hardware modelling to give an overview on how the robot is build and configured. This chapter will explain how the robot is tested such that it can validate the proof of concept. In order to draw conclusions about the performance of the proposed method, an experimental methodology is set up to test the robot. The experimental methodology is split into two parts. The first part will focus on how the validation will be done concerning the driving behaviour of the robot and the second part focuses on how the validation will be done concerning the semantic segmentation.

6.1. Approach

To validate the driving behaviour of the robot, the proposed method uses a 'track' along which the robot can drive repeatedly. In this case, the track is located where the hose was laid down on the ground. The path that the robot should follow is according to the two driving schemes found in figure 6.1. In order to perform one run, the track was prepared in the following way: the end of the hose was laid down at the finish line of the driving scheme. With an Xbox controller, the vehicle is manually driven back to the starting point. Before the first test, small orange flags were put in the ground to mark the path. To compensate for potential driving errors, the position of the hose was corrected at the points where it differed from the driving scheme. This was done by manually moving the hose towards the flags on places where there was a difference in lateral position. When the hose is laid down and the vehicle is positioned at the starting point, a run can start. The robot is switched from manual modus to hose following modus. While the vehicle follows the hose, multiple sensors are monitoring different conditions of the robot: the global position in X and Y coordinates of the robot by using the onboard GPS; the linear velocity and angular velocity by using the onboard GPS; the steering angle of the front axle by using an encoder. Finally, images are saved during each run, together with the output of the semantic segmentation model. When the robot has found the end of the hose, the robot and the process of monitoring stops.

Depending on which test is conducted, different parameters were changed beforehand, in order to map the different driving behaviours of the motion of the robot. When a change was made, a set of ten repetitive runs was conducted, to get an overview on how the changed system was performing. The different tests and parameters will be explained in the subsections below. A small hypothesis for each test is given as well.

6.2. Test 1 - left and right corner

The first test is about testing the consistency of the vehicle, while driving through a corner. The left track from figure 6.1 displays two curves, which is the reference on how the path laid on the ground. The two curves have the following in common: the starting position, the radius and the path length of 8.85

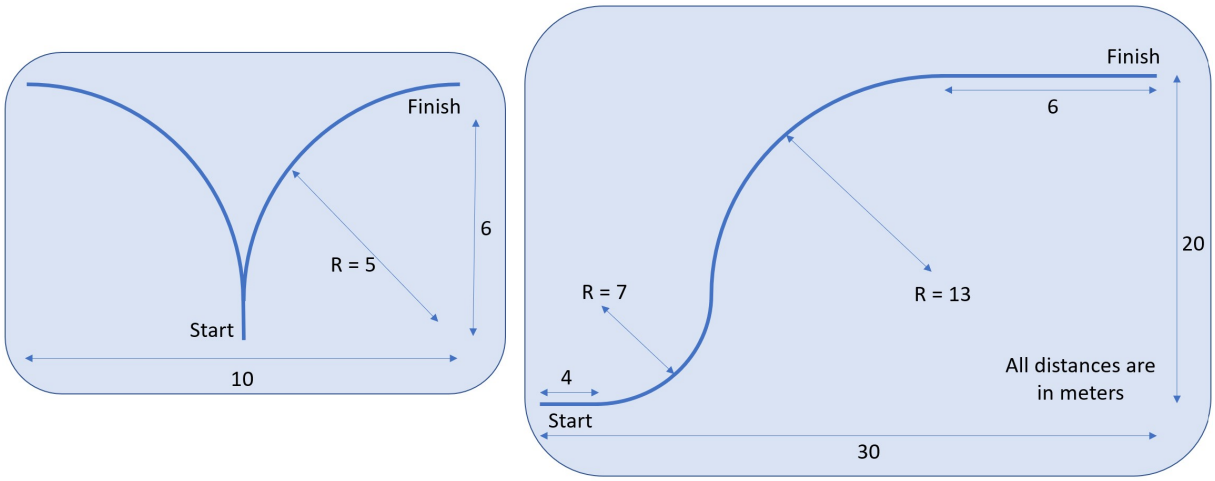


Figure 6.1: The two different driving schemes that the robot should follow during test 1 and 2.

meter. The consistency is tested by comparing the results of the driving behaviour between driving along the left and right curve. In both cases, the deviation of the individual runs are compared to each other. This test is performed using a linear velocity threshold of 0.2 m/s, using a close horizon. This means that segment 3 and 4 are weighing more than segment 1 and 2 in the average angle calculation of the hose compared to the vehicle, as explained in chapter 5. The deviation of the lateral position of the initially laid path is $\pm$ 0.25 m.

6.3. Test 2 - two consecutive corners with a straight section

The second test has the purpose to see how the vehicle will manage transitions between sections of the path. The path is shown on the right side of figure 6.1. This path has four transitions build in: a transition from a straight section to a sharp left corner; from a sharp left corner to a dull right corner; and from a dull right corner back to a straight section. The first corner has a radius of seven meters, which is also close to the limit of what the vehicle and the hose are capable of. This test is performed under four different circumstances. The different circumstances are shown in table 6.1. The total length of this path is 41.4 meters, with a lateral deviation of $\pm$ 0.25 m.

Table 6.1: The four different scenarios for test 2

Scenario	Velocity $[m/s]$	Type of horizon $[-]$
1	0.20	close
2	0.20	far
3	0.40	close
4	0.40	far

Just as in test 1, the lowest chosen velocity is 0.20 m/s. The reason for choosing this velocity is that it is certain that the vision pipeline and reaction time of the robot are under the limitations of what they are capable of. This was concluded during preliminary tests during the development of this system. From this baseline, it becomes possible to see the limitations of the robot, by changing the velocity and by changing the horizon of the vision pipeline. So, with a velocity of 0.20 m/s it is expected that the robot is able to follow the line better than when using a higher velocity. By using a higher velocity, the reaction time of the system becomes more important. Therefore, it was expected that the robot would be more likely to steer later. When the robot steers later, it is prone to steer more aggressively to stay on track. The horizon, as explained in chapter 5, can be close or far with respect to the vehicle. The hypothesis is when a close horizon is used, that the robot will follow the hose more accurately. The downside is that it would put more effort in steering into the right direction. But with using a far horizon,

both effects are expected to flip. So, it is expected that the robot will follow the hose less accurately, however it is expected that the steering effort will be less.

6.4. Test 3 - validation of the vision pipeline

The Deeplabv3+ network is the core of the vision pipeline. It is used in this method to detect the hose such that the motion of the vehicle can be controlled after the implementation of the vision pipeline. As mentioned in the sections above, test 1 and 2 will validate the driving performance of the entire robot. This test only includes the validation of the neural network to measure its performance and to compare it to the performance found in the reviewed literature. Deeplabv3+ performs well when trained on the Pascal VOC dataset. However, the case of detecting a long thin feature is different, as mentioned in section 4.1. The goal is to find the best performing network. However, finding possible trends on why a model outperforms another model is an important goal for the further development of the neural network.

In section 4.1.4, an overview is given of the three datasets that were created during this research. To recap: the first dataset contains images with short grass with a large FOV; the second one with concrete as background and with small FOV; and the last one with tall grass as background and a small FOV as well. Different dataset combinations were created to train the neural network, using the three individual datasets that were created initially. In addition, data augmentation is used to extend the size of the dataset by creating 'new' images from the old images, as described in chapter 4. The use of the augmented images are validated by comparing the MIoU, to conclude if it has a positive effect on the performance of the neural network.

Table 6.2: Dataset combinations with individual datasets: Dataset 1: short grass; Dataset 2: concrete; Dataset 3: tall grass. The augmented images are created from the original images.

Dataset combination	Augmented images (yes/no)	amount of images
3	No	720
3	Yes	2880
1, 2	No	455
1, 2	Yes	1820
1, 2, 3	No	1175
1, 2, 3	Yes	4700

The different datasets are found in table 6.2 and are validated in the following way: a random selection was made from all the images, existing out of 134 images. This dataset is used as to validate all the six different trained Deeplabv3+ models.

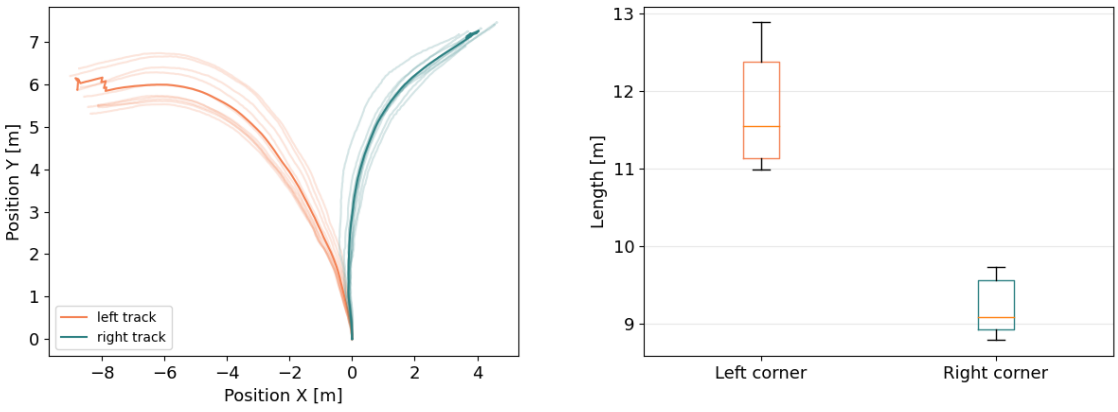
7

Results

The outline for this chapter is similar to the one for the experimental methodology. First, the results of test 1 are displayed in section 7.1. Secondly, the results of test 2 are displayed in section 7.2. Lastly, the results of test 3 are displayed in section 7.3.

7.1. Results from test 1

The goal of the first test is to measure the accuracy difference between driving along a right or left curve. Both tests have been conducted ten times. The GPS coordinates of the vehicle are shown in figure 7.1a. In light blue, the ten results of driving along the right corner are shown. The darker line represents the average over the ten runs. The same is done for the left corner, the results are shown in orange. At the ends of the lines, a jitter occurs for both the average paths. This is due to how the average is calculated. Every run has a different end, due to the different stopping time of the robot. The average is calculated in the following way: at the start there are ten data points because of the ten runs. However, when a run stops, the average is then calculated over the remaining nine data points, and so on. Meaning that it could result in a positional shift, because the average of the nine other results can differ from the other average. It is visible that the left curve is shorter than the right curve. While the initial laid path is equally. This result is found back in figure 7.1b. The length of the path initially laid was 8.85 meters. The mean of the length of the path differs from the initial length. The average driven length of the left curve is 11.54 meters. The average driven length of the right curve is 9.08 meters, meaning that it differs with 30% and 2% respectively.



(a) x and y position of the robot according to the robot GPS

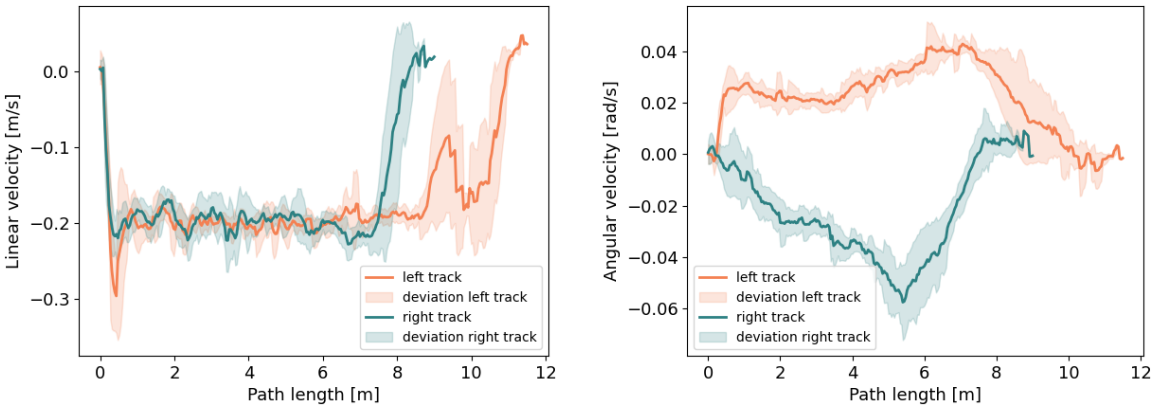
(b) The standard deviation of the length of the right and left path

Figure 7.1: Main results from test 1: the left image shows position and the right image shows the standard deviation of the track length.

Figures 7.2a and 7.2b show the average linear velocity and the average angular velocity, respectively. The peaks that are visible at the starting point and at the end of figure 7.2a are the result of the PID of the vehicle controller. The vehicle starts from a hold and accelerates to a velocity that has been set at 0.2 m/s. The peak at the end of the figure is spread out, again due to the fact that not every end has the exact timing. The blue and orange area around the averages are showing the standard deviation. The standard deviation is calculated for every time instance by using the following formula:

$$SD = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2}$$

$\overline{x}$ Represents the mean of the measurement, x_i is the linear or angular velocity of one single run and N is the total amount of runs at that time instance. The deviation at the end of the runs seems to be higher. This is again because of the different ending times of the single runs. It means that the SD at the end is calculated with fewer runs, resulting in larger deviations. The reason why the velocity is negative is due to the backwards driving direction. The blue line (linear velocity from the left curve) is shorter than the orange line, due to the shorter path. The angular velocity along the two curves is different as well, apart from the opposite direction of the angular velocity. The blue line from the right corner shows a sharper change in angular velocity than the orange curve. The images are displayed in a larger format in appendix A.



(a) The mean and deviation of the linear velocity along the curves

(b) The mean and deviation of the angular velocity along the curves

Figure 7.2: Main results from test 1: the left image shows the linear velocity and the right image shows the angular velocity: x and y position, linear velocity, path length, angular velocity.

7.2. Results from test 2

Figure 7.3 shows the travelled path from the 40 individual runs that were performed in test 2. The darker lines are representing the averages of the 4 different tests that were conducted. Each of the averages is calculated from ten runs and is calculated in the same way, as in 7.1. In figure 7.3 a jump is seen in the dark blue line. Some of the runs did not finish at the end of the path. The reason for this abrupt stop was that the robot had to steer too sharp, making it to stop. Therefore, some runs are shorter than others, which results in jumps in the averages. The black line represents the position of where the hose laid at the beginning of each run. The light blue line shows the case using a velocity of 0.2 m/s and a far horizon. The dark blue line shows the case using a velocity of 0.4 m/s and a far horizon. The red line shows the case using a velocity of 0.4 m/s and a close horizon. The orange line shows the case using a velocity of 0.2 m/s and a close horizon. These conditions are summarized in table 7.1.

Table 7.1: The conditions for every run summarized

Scenario	Velocity condition	Horizon condition	Colour in the graphs
1	0.2 m/s	close	orange
2	0.2 m/s	far	dark blue
3	0.4 m/s	close	red
4	0.4 m/s	far	light blue

It stands out that the deviation is larger for both cases with the far horizon. Both blue lines are showing behaviour of early steering. The red line lies close to the ground truth and the orange line shows a late steering behaviour. Both cases using a velocity of 0.4 m/s are steering earlier than the slower version. The maximum amount of deviation from the ground truth, the average path length, and the path length deviation from the ground truth can be found in table 7.2.

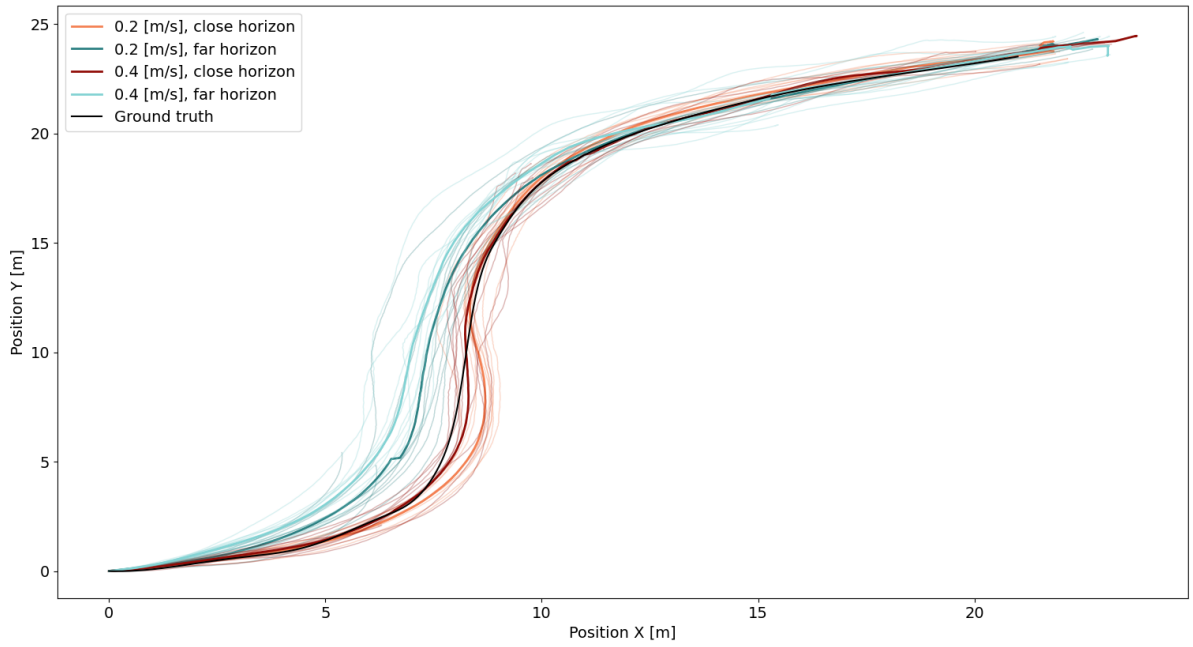


Figure 7.3: The measured GPS position of the vehicle in X and Y coordinates.

Table 7.2: The table shows: the maximum deviation in lateral position, the average path length and the deviation of the path length in percentage.

Scenario	Max deviation [m]	Median of the path length [m]	Average deviation from gt [%]
1	0.74	37.4	90.3
2	1.11	36.3	74.8
3	0.33	30.5	70.0
4	1.53	35.8	86.5

Figure 7.4 shows the standard deviation of the path length that has been driven. As expected, the first scenario performed best. It has a deviation of 90.3% from the original path length. Besides, the maximum lateral deviation is within 1 meter and the deviation over the path length is the smallest. Scenario 2 performed almost equally well. Unfortunately, it had two outliers caused by a hard stop, due to a sharp angle. Yet the median of the path length lies close to the one of scenario 1 and 4. Scenario 3 performed poorly, the standard deviation is large and it had a total of five outliers. Scenario 4 shows that a higher velocity does not necessarily mean that it will perform poorly. Unfortunately, it had an outlier as well, due to the oscillations of the robot.

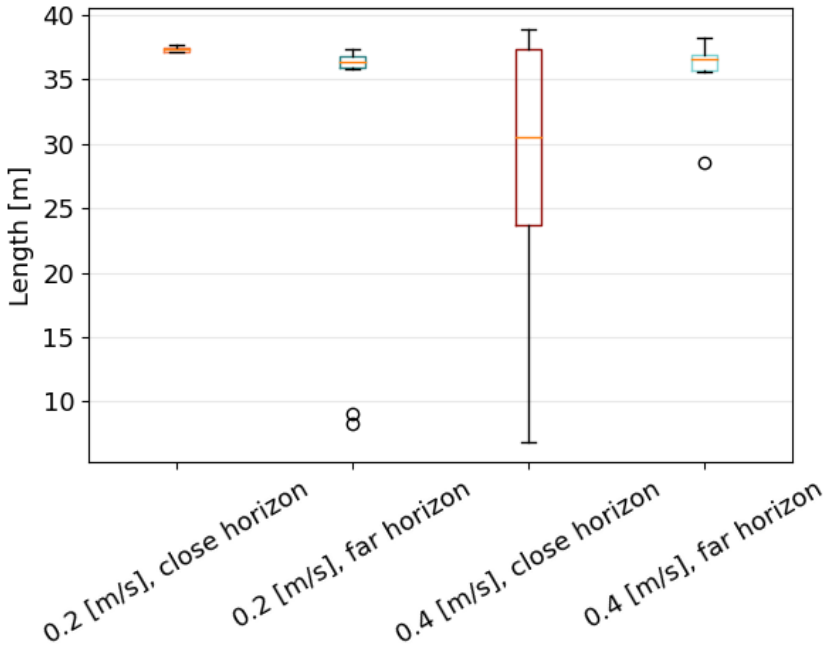


Figure 7.4: Standard deviation over the path length from test 2

The average linear velocity of the entire path is shown in figure 7.5. The averages are similarly calculated as before, and the peak at the beginning and end of the figure are similar as seen in earlier results. As told in chapter 5, the set point velocity of the PID controller is lowered as the steering angle of the vehicle becomes too sharp. This behaviour is clearly visible in the scenarios where the robot drives faster. It is visible that the velocity did not change in the scenarios where the robot drives slower. When the robot changes a lot in linear velocity, it means that it has to correct itself more. This is visible in the velocity curve from the moment that it leaves the first corner. In this graph, it is from the moment it passes the 21 meter. These oscillations are better shown in the bottom left figure of figure 7.6. The average angular velocity and the deviation of the angular velocity are shown here. In the upper left figure, scenario 1 is shown. This scenario has the least amount of jittering, and it shows the least amount of deviation. This scenario is closely followed by scenario 2, except for two minor difference. The deviation and average are both showing slightly more jittering. Another minor difference is that the transition between the curved and straight section are somewhat sharper as seen in the other scenarios. The bottom right scenario performed the worst. The transitions are barely visible, and the robot is continuously oscillating.

Figure 7.7 shows the standard deviation of the linear velocity and figure 7.8 shows the standard deviation of the angular velocity. The tenor of the results of both figures is similar, compared to figure 7.6 and figure 7.5. However, it stands out that the standard deviation for the fast scenarios is much wider. These two figures are showing clear differences between the slow and fast scenarios, but it lacks to show a difference between the slow or fast runs themselves.

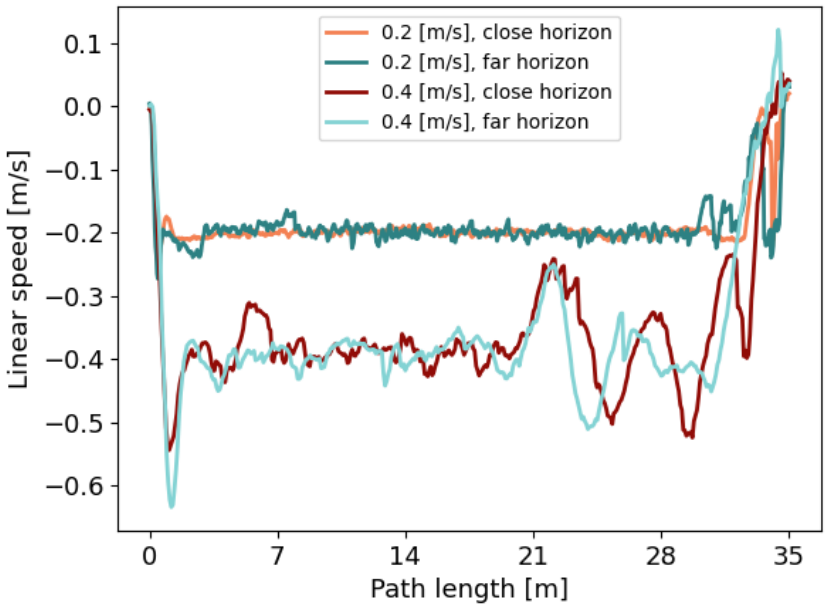


Figure 7.5: The measured linear velocity along the entire path.

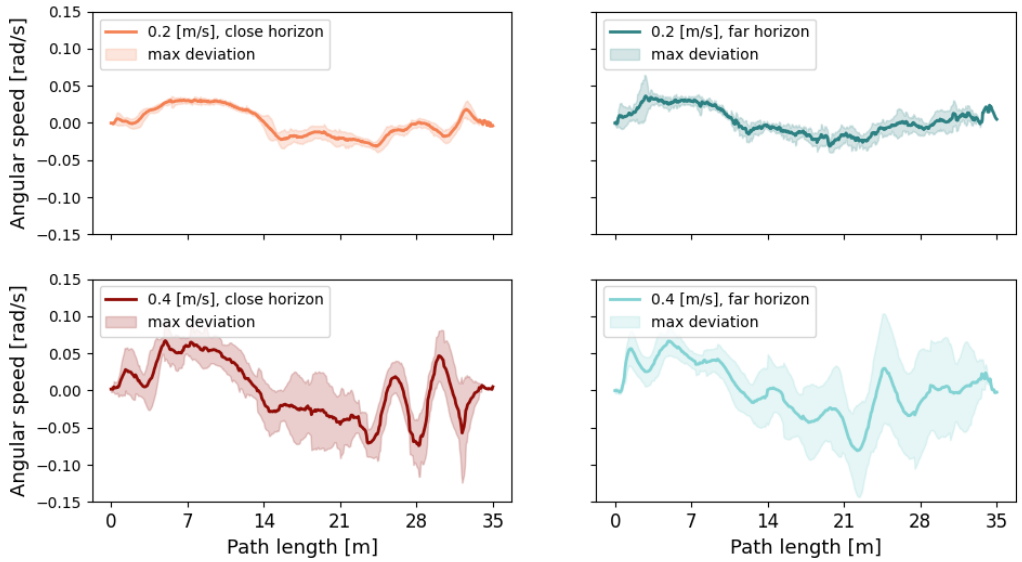


Figure 7.6: The average and maximum angular velocity. From the left upper image to the right bottom images:0.2 m/s, close horizon; 0.2 m/s far horizon; 0.4 m/s, close horizon; 0.4 m/s, far horizon.

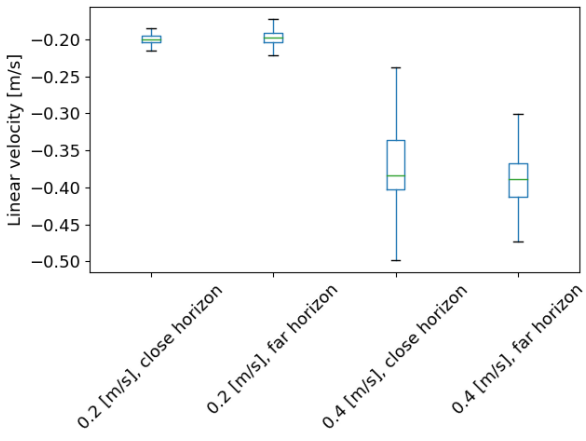


Figure 7.7: Standard deviation of the linear velocity of all scenarios.

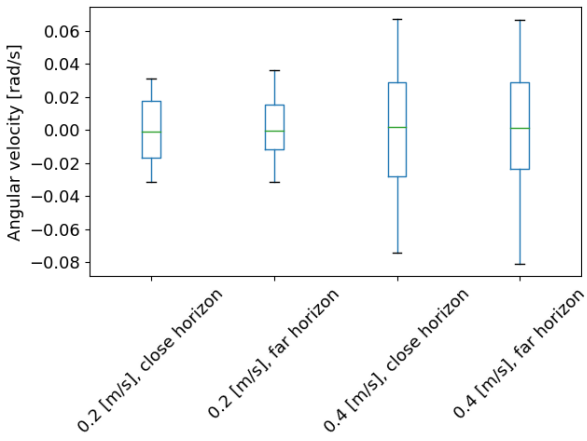


Figure 7.8: Standard deviation of the angular velocity of all scenarios.

7.3. Results from test 3

As told in chapter 6, dataset 1 contains images with a large horizon. The grass on these images is short and equal in length. The hose in these images is rather thin, due to the larger FOV. Dataset 2 contains images with a small FOV. The background contains concrete instead of grass, and the hose appears to be larger. Lastly, dataset 3 contains images with the same FOV as in dataset 2, the background contains tall grass. The hose in these images is partially occluded because of the tall grass.

Table 7.3 shows the performance of the DeeplabV3+ model trained with different combinations of datasets. The validation of all three sets is performed on the same dataset. Table 7.3 shows that the combination of dataset 1, 2, and 3 without the augmented images performed the worst compared to the others. It also shows that the combination of dataset 1 and 2 with augmented images performed the best. This table shows the trend that a dataset with augmented images performs better.

Table 7.3: MIoU score for the Deeplabv3+ model with 6 different datasets.

Dataset combination	Augmented images	Images count	MIoU score in [%]
3	No	720	30.1
3	Yes	2880	51.9
1, 2	No	455	28.8
1, 2	Yes	1820	58.6
1, 2, 3	No	1175	22.0
1, 2, 3	Yes	4700	54.6

Table 7.4 shows the average true and false positive rates for the same datasets from the Deeplabv3+ model. A true positive is a hose pixel that is correctly labelled as hose pixel. A background pixel that is labelled as hose pixel is called a false positive. The averages are coming from averaging the true and false positive rates from the 134 images that are used for the validation. Table 7.4 shows that a dataset with little images will contain more false positives. This is especially shown in the difference between only using dataset 3 without augmented images and by using all the datasets with data augmentation.

Table 7.4: Average true and false positive rate for the deeplabv3+ model with 6 different datasets.

Dataset combination	Augmented images	True positives rate in [%]	False positives rate in [%]
3	No	33.1	21.3
3	Yes	61.8	16.7
1, 2	No	31.0	10.4
1, 2	Yes	65.4	14.2
1, 2, 3	No	24.0	16.0
1, 2, 3	Yes	57.6	4.2

Figures 7.9, 7.10, 7.11, 7.12 are showing a total of four example images. Every example has an original image with the annotated ground truth next to it. The bright pixels that are displayed are representing the correct position of the hose. Beneath the original image and the ground truth, there are six results coming from the six differently trained neural networks. Figure 7.9 shows an image coming from dataset 2. It is visible that the trained neural network using dataset 1 and 2 with augmented images performs the best. One can see a failure in all the other models at the point where the hose intersects with the steel cover. The example from figure 7.10 does not have a model that performed significantly better than the other models. The models trained on dataset 3, dataset 1, 2 and 3 (with or without augmented images) performed equally well. The model trained on dataset 1, 2 and 3 with augmented images performed best in figure 7.11. It stands out that the segmentation from the models trained on dataset 1 and 2 stops working when the sun shines partly on the image. Figure 7.12 shows an example that comes from test 2. It stands out that some of the models are not detecting the hose at all, whereas the other models are working properly.

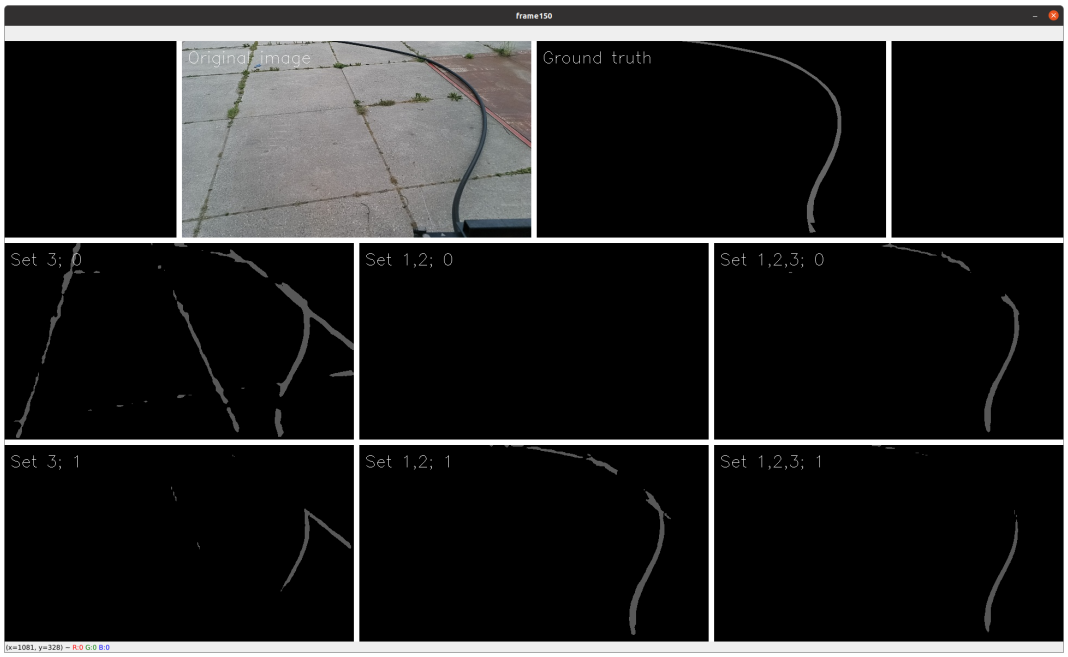


Figure 7.9: Image from dataset 2, the 0 after the dataset means that it does not have augmented images.

Figure 7.10: Image from dataset 3, the 0 after the dataset means that it does not have augmented images.

Figure 7.11: Image from dataset 3, the 0 after the dataset means that it does not have augmented images.

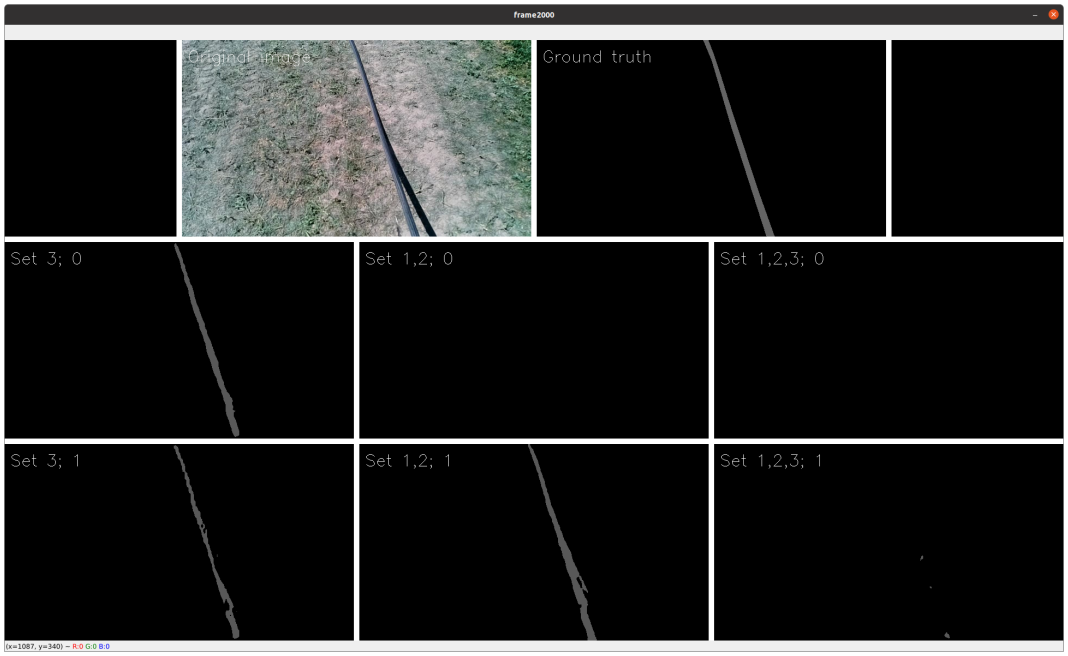


Figure 7.12: Image from test 2, the 0 after the dataset means that it does not have augmented images.

8

Conclusion

The conclusion has the same structure as chapter 7. First, an overall conclusion will be drawn concerning the performance of the proposed method. Thereafter, individual conclusions are drawn concerning the three tests that are conducted. The main research question is represented again, as stated in chapter 2: *"With what accuracy can the implementation of visual control using a combination of segmentation, clustering and polynomial regression be used, as input for a PID based vehicle controller?"*

8.1. Main findings

The method that has been proposed, was successfully implemented such that the agricultural robot was able to follow a hose, which was laid down in a grass field. During the tests, the robot drove about 1800 meters in total, while performing all the steps done described by our method. The robot successfully found the end of the hose 52 times out of 60 times. Meaning that the robot stopped 8 times too early because of a failure. The single longest run was about 40 meters, achieving a maximum velocity of 0.4 meters per second. The best accuracy is achieved by driving with a linear velocity of 0.2 m/s using a close horizon. The maximum lateral deviation is 0.74 meter and the average longitudinal deviation from the ground truth is 9.7% (less is better). This accuracy has been achieved by using a non-ideally performing neural network. The deeplabv3+ model used during this test had an MUoI of 58.6%. The proposed model can perform better, as mentioned later. The accuracy is further evaluated in the coming sections, by drawing conclusions regarding each individual test. Besides, it will show how the proposed method should be changed to achieve a higher accuracy.

8.2. Conclusions based on test 1

Test 1 is conducted to spot differences between a path that is curved similarly to the right as to the left. It is clear that there is a noticeable difference. This can be deducted from: the deviation in the length of the path; the deviation in x and y coordinates of the robot at the end; and because of the different steering behaviour of the robot. A reason for this difference is based on how the calculation for the front axle is conducted. The direction of the hose is taken into account, but the point where the hose leaves the vehicle was left out of the equation. When the hose leaves the vehicle on the right side and the path is to the right side as well, the robot will drive along the inside of the curve. Whereas the robot would drive along the outside curve when the path lies in the other direction. When looking at figure 7.1a, it turns out that the calculation of the steering angle is generalized too much. The point where the hose leaves the vehicle should be involved as well. It means that the calculation of the steering angle will become more accurate.

This explains the different radius and the different angular velocity, but it does not explain why the path length deviated differently within the two cases. This is due to the relation between the linear velocity of the robot and the angular velocity of the reel. This correlation is crucial for the robustness of the entire robot. When this relation is off, it results in either a pushing or pulling behaviour, causing

the hose to shift on the field. This pushing behaviour occurred more when the robot was driving the left track, resulting in a path length that was longer by 30%. It could be solved by a faster angular velocity of the reel. The fact that this behaviour occurred, does not mean that the earlier drawn conclusions are incorrect. The pushing behaviour is partially caused by the different steering behaviour, but it is mainly caused by using the wrong correlation between the driving velocity and the turning rate of the spool. During this phase of testing, the sensor that measures the force which is exerted on the hose, was wrongly calibrated. With a correctly calibrated sensor it is expected that the (length) differences between the two paths found in figure 7.1a becomes less.

8.3. Conclusions based on test 2

The second test was conducted to see differences between the transitions in track sections. In addition, the test was repeated four times to change the linear velocity of the vehicle and to change the weight segments of the steering angle calculation, as explained in chapter 5. The conclusions that can be drawn from this test are: the robot should have a close horizon at lower speeds; the robot is able to correct itself when deviating from the path; a speed of 0.4 m/s is too high due to camera lag; and the robot is able to follow the hose accurately at lower speeds without a high steering effort.

8.3.1. Calculation of the hose angle

Starting with the horizon choice for the robot. This conclusion can be drawn based on figure 7.3. The black line represents the ground truth of where the hose initially laid. The two blue lines both have far horizons. Meaning that segment 3 and 4 from the angle calculation are weighing more than segment 1 and 2. Therefore, it was expected that the robot would steer earlier, as can be seen in the figure. It can be concluded that the red and orange line lie closer to the ground truth than the two blue lines, meaning that the robot should mainly base its steering angle on segment 1 and 2. Another argument is found in figure 7.6. When comparing both the slow scenarios, it stands out that the maximum deviation in angular velocity is smoother when the robot has a close horizon. Therefore, the conclusion can be drawn that the steering effort is slightly less with a close horizon.

Another behaviour concerning early or late steering stands out. In both cases, it becomes evident that when the robot drives faster, it steers earlier. This behaviour is found when both of the close horizon cases are compared to each other. The reason why this behaviour occurs cannot be concluded from the data. However, the reason could be an existing correlation between driving and steering. It could be that the vehicle steers smoother when it has a higher velocity.

Figure 7.3 makes it clear that the robot roughly ends at the same spot in all the cases. Even though the hose is sometimes shifted a lot in the first corner. It makes a difference that the next corner is in the opposite direction, yet it is possible to conclude that the robot has a certain ability to correct its own mistakes. But when the shifting happens, the robot should still reel in the hose without shifting the section of the hose that is positioned on the ground. When it happens, the robot is able to correct itself up to a certain height. The question remains if this behaviour is still visible when the robot only drives through one type of corner. This should be tested in a future research.

8.3.2. Camera lag

When the robot is driving based on a GPS signal, it is capable of driving speeds up to 1 meter per second. Most of the tests are conducted with a base speed of 0.2 meter per second. The reason for this relatively low velocity is to exclude problems concerning the velocity. However, it is interesting to see where the limits of this proposed method are: the limitation has been found to be 0.4 meters per second. This conclusion can be drawn based on figure 7.6. The oscillation behaviour is mainly found when the robot looks close to its base in combination with a velocity of 0.4 meter per second. This behaviour exists with a far horizon as well, but that appears to be more random. The oscillation is particularly found at the end of the track, where the robot should follow a straight line. The oscillation is caused by the lag of the camera. The inference time of the neural network is around 30 - 50 milliseconds, the rest

of the vision pipeline is around 30 milliseconds. But the time it takes the camera to record, encode and send an image can take up to 400 milliseconds. The last cause is also referred as camera lag. So, from recording a frame to receiving a velocity command can take up to half second, which is very long for a robot that steers based on visual commands. On top of that, the robot requires time to correctly steer its wheels. So, when the robot executes a command, it overshoots. When the robot tries to counter the overshoot, it will steer too aggressively, resulting in oscillation. Camera lag is something that can be decreased by using better equipment or by using a more efficient code. In a future research, this should be taken into consideration, such that the limit of the vehicle can be extended. All in all, in most cases the robot is still able to reach the desired goal, even with early steering or with oscillations.

8.4. Conclusions based on test 3

This last section will draw conclusions about the vision pipeline. These conclusions are drawn separately from test 1 and 2 because they do not conclude something about the driving behaviour. What stands out is that the segmentation performed worse than the examples found in the literature. Besides, the clustering is mainly included to counter the poorly performing network. Although, it is a good addition to the model because with or without a proper functional network, it makes the method more robust.

8.4.1. Validation of the neural network

According to the research group from Deeplabv3+, they achieve an average MIoU above 70% [6]. That implies that some of their classes perform better than the 70%, whereas other classes perform less than the 70%. Our method has to classify only between two classes, either by classifying between the background or hose class. Therefore, it was expected that the network would perform better. There are a few elements about our training method and data gathering technique that could be the reason why this model did perform less.

First, our dataset is rather small, containing just over 1000 images. Normally, these neural networks are trained on image sets containing 5000+ images, such as the PASCAL Visual Object Classes dataset. In that regard, the comparison is skewed. Adding more images to our dataset will probably result in an increase of the performance of the model. However, adding just more images is not going to solve every problem. Just as in the research of R. Ma, P. Tao, and H. Tang, [26] data augmentation increased the performance of the network. However, optimization of our data augmentation can be achieved. Flipping the image, together with darkening or brightening the image, is used to train our method. The researchers from [26] found that there is an optimum when the techniques of cropping, shifting the image and compression performs the best with using the Deeplabv3+ neural network. None of these methods have been used in our work, so it is believed that there is room for improvement when these techniques are included to train our model.

During the process of gathering the training data, the camera position is changed multiple times. Dataset 1 has a different camera angle than the other two datasets. Besides, this camera position is different from its final position. The conclusion is that the data that has been used, could be more representative. Extra data does help, as long as it is representative. Therefore, the performance could increase when this data will be replaced with more representative data. Besides, the final version of the robot will not operate on a concrete background. Therefore, new images should only contain grass as background, preferably with a combination of tall and short grass.

8.4.2. Clustering

The clustering algorithm performed well in most cases. However, an improvement could be made by choosing the cluster that represents the hose. Currently, this method does not store the location of the hose of where it was in previous frames. When the hose starts at position x in the bottom of the frame, it is not possible for the next frame to have a completely new starting position. So, when the robot has seen a few consecutive frames, it is possible to create a heat map of where the robot expects the hose to start. After each frame, this heat map will be updated such that the clustering algorithm has more information about the starting position of the hose.

8.5. Limitations

This method is build for a robot that follows a hose. When the hose moves, the trajectory moves with it. As told in the introduction 2, the trajectory is not allowed to move because the end goal is fixed and the way there and back must be similar. The results show that the robot is capable of reaching the end goal, even for when the trajectory is moved slightly by the robot itself and for when the end is not fixed. But, the limitation of the method is that the real displacement of the hose is not measured during the tests. The conclusions of the driving performance are made using the position of the robot. It gives a meaningful insight, but the real displacement of the hose is an important measurement to effectively measure the performance and the repeatability of this method. Besides, it would be a good metric to compare the work of others with the performance of this robot.

Another limitation is that the robot does not remember the start position of the hose. The vision pipeline works without this feature, yet it will make the system more robust when it would be added. The robot did not follow other 'tracks' than the hose, during the tests that were conducted. The limitation is that the robot is still able to switch between tracks. When the robot remembers the starting point from the hose based on its last few frames, the likelihood of switching between tracks will be less.

8.6. Final remark

This conclusion mentioned the performance of the robot in combination with the proposed method. This setup is different from other cases found in the reviewed literature. Track following systems are not using semantic segmentation on a regular basis, because simpler methods can cope with other problems found by the ROVs or AVs. Our method has to cope with either curved tracks or straight tracks. The proposed vision pipeline and the steering method are able to let the robot drive autonomously along those tracks. In addition, the other methods are navigating along stationary tracks. Although the track did change a bit while driving in our case, this method shows to be a proper method from navigation from point A to point B.

Lely wants to further research the possibilities to apply this method in their robot, because the concept of following the trajectory with the mentioned setup is proved to function properly. However, it should be mentioned that this method requires to perform in a robust manner before it can be applied in real world scenarios. Therefore, future research will focus on making the method of following the trajectory more robust. That means that the following elements should be improved: the lag of the camera should be decreased such that the velocity of the vehicle can be increased; the neural network should be trained with more representative (augmented) data such that the accuracy of the detection of the hose increases; the decision on how the hose is chosen after clustering the mask should be improved by adding the heat map; lastly, the research will be expanded towards conditions which include night operation. When these improvements are made, it becomes possible for Lely to apply this method in their robots.

References

[1] Mary B Alatise and Gerhard P Hancke. “A review on challenges of autonomous mobile robot and sensor fusion methods”. In: *IEEE Access* 8 (2020), pp. 39830–39846.

[2] Eduardo Arnold et al. “A survey on 3d object detection methods for autonomous driving applications”. In: *IEEE Transactions on Intelligent Transportation Systems* 20.10 (2019), pp. 3782–3795.

[3] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. “Segnet: A deep convolutional encoder-decoder architecture for image segmentation”. In: *IEEE transactions on pattern analysis and machine intelligence* 39.12 (2017), pp. 2481–2495.

[4] Amol Borkar et al. “A layered approach to robust lane detection at night”. In: *2009 IEEE Workshop on Computational Intelligence in Vehicles and Vehicular Systems, CIVVS 2009 - Proceedings* (May 2009), pp. 51–57. DOI: 10.1109/CIVVS.2009.4938723.

[5] Hough P. V. C. “Method and Means for Recognizing Complex Patterns”. US patent 3069654. Dec. 1962. URL: https://patents.google.com/patent/US3069654A/en.

[6] Liang Chieh Chen et al. “DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 40 (4 Apr. 2018), pp. 834–848. DOI: 10.1109/TPAMI.2017.2699184.

[7] Liang-Chieh Chen et al. *Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation*. 2018, pp. 801–818.

[8] A. E. Conrady. “Decentred Lens-Systems”. In: *Monthly Notices of the Royal Astronomical Society* 79.5 (Mar. 1919), pp. 384–390. URL: https://doi.org/10.1093/mnras/79.5.384.

[9] LUXONIS HOLDING CORPORATION. *OpenCV AI Kit: OAK—D*. 2022.

[10] David. *tf-keras-deeplabv3p-model-set*. 2021. URL: https://github.com/david8862/tf-keras-deeplabv3p-model-set.

[11] Richard O. Duda and Peter E. Hart. “Use of the Hough Transformation to Detect Lines and Curves in Pictures”. In: *Commun. ACM* 15.1 (1972), pp. 11–15. DOI: 10.1145/361237.361242.

[12] Mark Everingham et al. “The pascal visual object classes challenge: A retrospective”. In: *International journal of computer vision* 111.1 (2015), pp. 98–136.

[13] A A Fallah, A Soleimani, and H Khosravi. “Real-time Lane Detection Based on Image Edge Feature and Hough Transform”. In: *J. Electr. Comput. Eng. Innovations* 9 (2 2021), pp. 193–202. DOI: 10.22061/JECEI.2021.7659.418.

[14] A A Fallah, A Soleimani, and H Khosravi. “Real-time Lane Detection Based on Image Edge Feature and Hough Transform”. In: *J. Electr. Comput. Eng. Innovations* 9 (2 2021), pp. 193–202. DOI: 10.22061/JECEI.2021.7659.418.

[15] Alberto Garcia-Garcia et al. “A review on deep learning techniques applied to semantic segmentation”. In: *arXiv preprint arXiv:1704.06857* (2017).

[16] Drew Gray. *Distortion 101 - Lens vs. Perspective*. 2014. URL: http://www.drewgrayphoto.com/learn/distortion101.

[17] A. H. Ismail et al. “Vision-based system for line following mobile robot”. In: 2 (2009), pp. 642–645. DOI: 10.1109/ISIEA.2009.5356366.

[18] Hiroshi Kano et al. “Precise top view image generation without global metric information”. In: *IEICE TRANSACTIONS on Information and Systems* 91.7 (2008), pp. 1893–1898.

[19] Wichtert Koopman. “Minder ammoniak verliezen is meer dan streepjes trekken”. In: *Veeteelt* (2016). URL: https://edepot.wur.nl/370751.

[20] K Mahesh Kumar and A Rama Mohan Reddy. “A fast DBSCAN clustering algorithm by accelerating neighbor searching using Groups method”. In: *Pattern Recognition* 58 (2016), pp. 39–48.

[21] Siddhant Kumar. *Summary of — SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation*. 2021. URL: https://towardsdatascience.com/summary-of-segnet-a-deep-convolutional-encoder-decoder-architecture-for-image-segmentation-75b2805d86f5.

[22] Fahad Lateef and Yassine Ruichek. “Survey on semantic segmentation using deep learning techniques”. In: *Neurocomputing* 338 (2019), pp. 321–348.

[23] Yu-Kai Lin and Shih-Fang Chen. “Development of navigation system for tea field machine using semantic segmentation”. In: *IFAC-PapersOnLine* 52.30 (2019), pp. 108–113.

[24] Jonathan Long, Evan Shelhamer, and Trevor Darrell. “Fully convolutional networks for semantic segmentation”. In: (2015), pp. 3431–3440.

[25] Chan Yee Low, Hairi Zamzuri, and Saiful Amri Mazlan. “Simple robust road lane detection algorithm”. In: *2014 5th International Conference on Intelligent and Advanced Systems: Technological Convergence for Sustainable Future, ICIAS 2014 - Proceedings* (2014). DOI: 10.1109/ICIAS.2014.6869550.

[26] Rui Ma, Pin Tao, and Huiyun Tang. “Optimizing data augmentation for semantic segmentation on small-scale dataset”. In: (2019), pp. 77–81.

[27] Satya Mallick and Kaustubh Sadekar. *Camera Calibration using Opencv*. 2022. URL: https://learnopencv.com/camera-calibration-using-opencv/ (visited on 06/26/2022).

[28] Abdelhamid Mammeri, Guangqian Lu, and Azzedine Boukerche. “Design of lane keeping assist system for autonomous vehicles”. In: (2015), pp. 1–5. DOI: 10.1109/NTMS.2015.7266483.

[29] Mehdi Narimani, Soroosh Nazem, and Mehdi Loueipour. “Robotics vision-based system for an underwater pipeline and cable tracker”. In: (2009), pp. 1–6. DOI: 10.1109/OCEANSE.2009.5278327.

[30] Dario Augusto Borges Oliveira et al. “A review of deep learning algorithms for computer vision systems in livestock”. In: *Livestock Science* 253 (2021), p. 104700.

[31] Intel Corporation OpenVINO. *Computer Vision Annotation Tool*. 2022.

[32] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[33] Simon Placht et al. “Rochade: Robust checkerboard advanced detection for camera calibration”. In: *European conference on computer vision*. Springer. 2014, pp. 766–779.

[34] Dominik Polzer. *7 of the Most Used Regression Algorithms and How to Choose the Right one*. 2021. URL: https://towardsdatascience.com/7-of-the-most-commonly-used-regression-algorithms-and-how-to-choose-the-right-one-fc3c8890f9e3 (visited on 09/29/2022).

[35] Mukhamad Aji Putra, Endra Pitowarno, and Anhar Risnumawan. “Visual servoing line following robot: Camera-based line detecting and interpreting”. In: (2017), pp. 123–128. DOI: 10.1109/ELECSYM.2017.8240390.

[36] RIVM. 2021. URL: https://www.rivm.nl/stikstof.

[37] Mel Siegel. “The sense-think-act paradigm revisited”. In: (2003), 5–pp.

[38] Sven Gjedde Sommer and NJ Hutchings. “Ammonia emission from field applied manure and its reduction”. In: *European journal of agronomy* 15.1 (2001), pp. 1–15.

[39] Heath Thomas Little. *Euclid: The Thirteen Books of Elements, Volume 2*. Dover, 1926.

[40] Nick Van Oosterwyck. “Real Time Human Robot Interactions and Speed Control of a Robotic Arm for Collaborative Operations”. PhD thesis. May 2018. DOI: 10.13140/RG.2.2.28723.53286.

[41] A. van der Wal, E. de Lijster, and W. Dijkman. *Ontwerp Label Duurzaam Bodembeheer*. July 2016. URL: https://library.wur.nl/WebQuery/wurpubs/fulltext/401825.

[42] Dongkuan Xu and Yingjie Tian. “A comprehensive survey of clustering algorithms”. In: *Annals of Data Science* 2.2 (2015), pp. 165–193.

A

Appendix: Supportive images

This appendix shows similar images as see in the research, but are displayed larger.

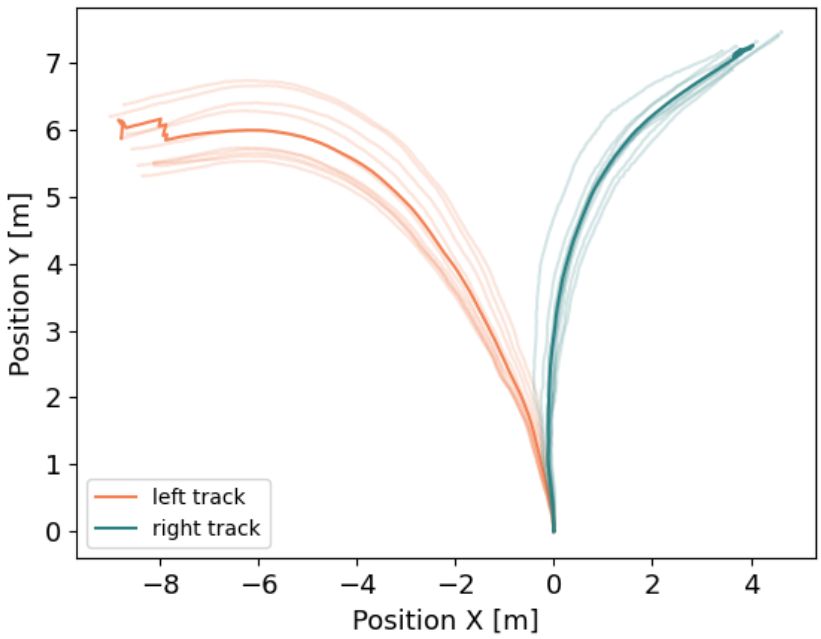


Figure A.1: x and y position of the robot according to the robot GPS

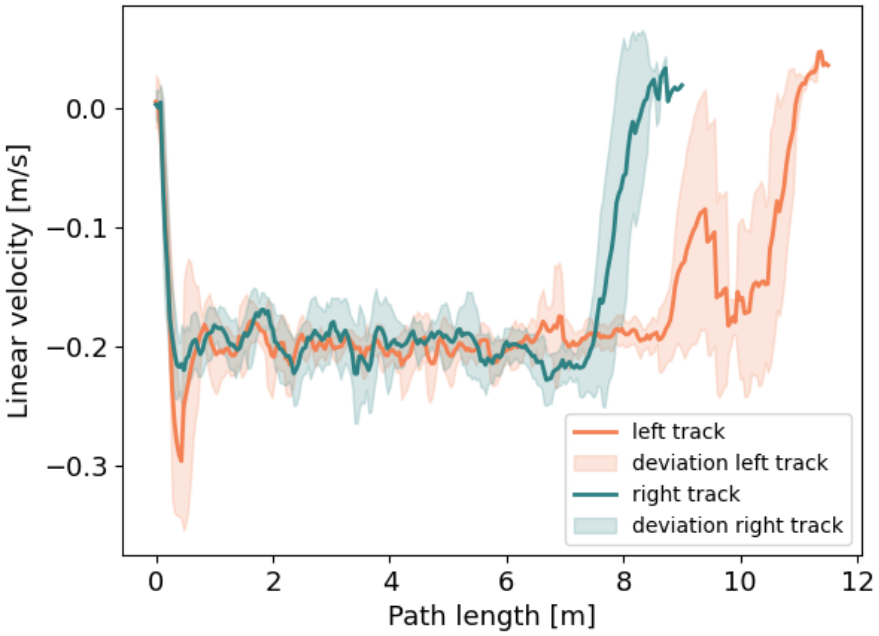


Figure A.2: The mean and deviation of the linear velocity along the curves

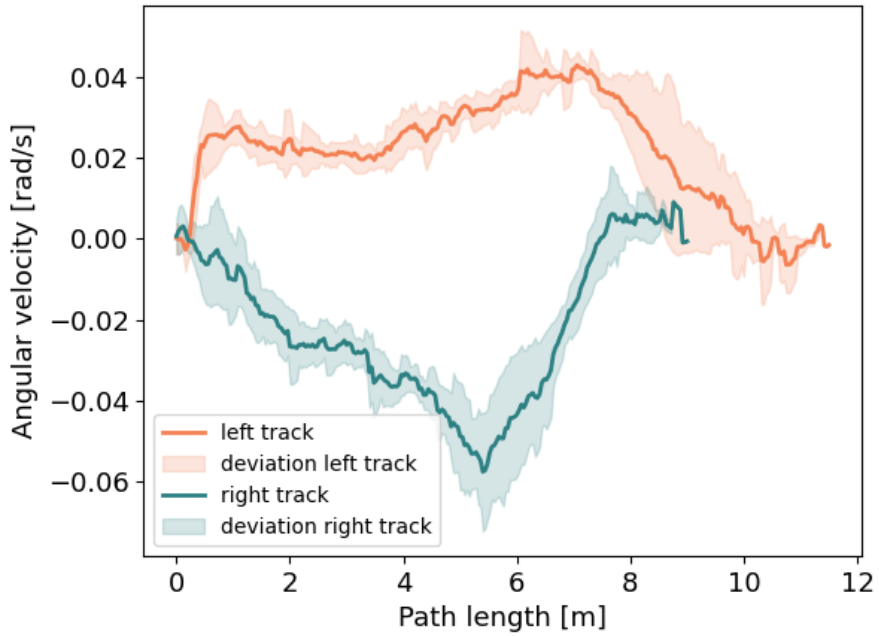


Figure A.3: The mean and deviation of the angular velocity along the curves

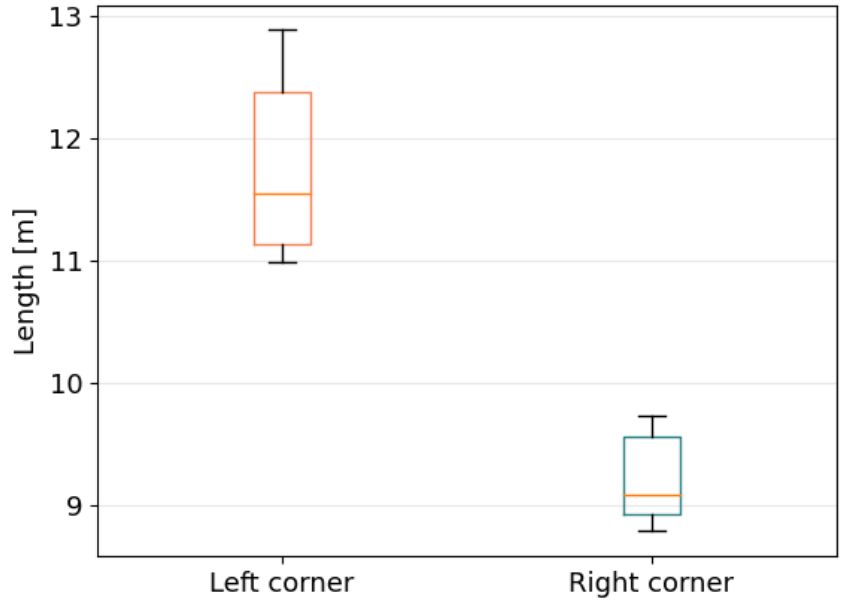


Figure A.4: The standard deviation of the length of the right and left path

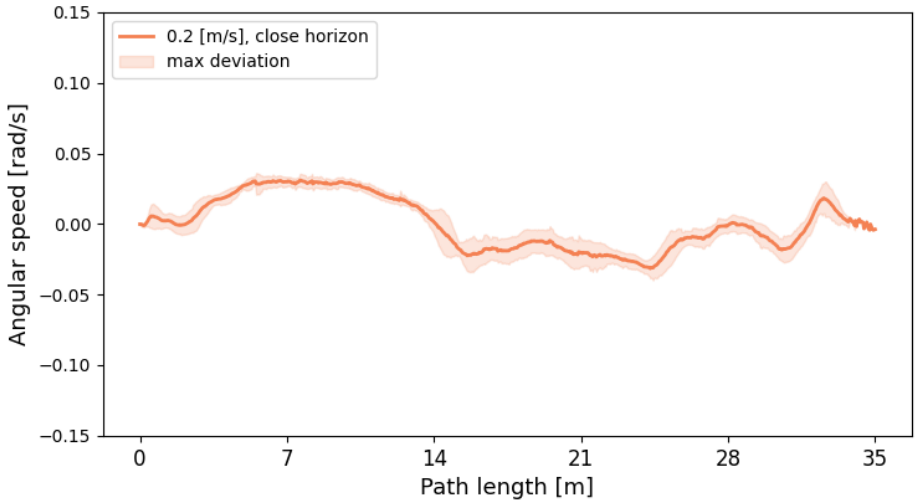


Figure A.5: The average and maximum deviation of the angular velocity of scenario 1

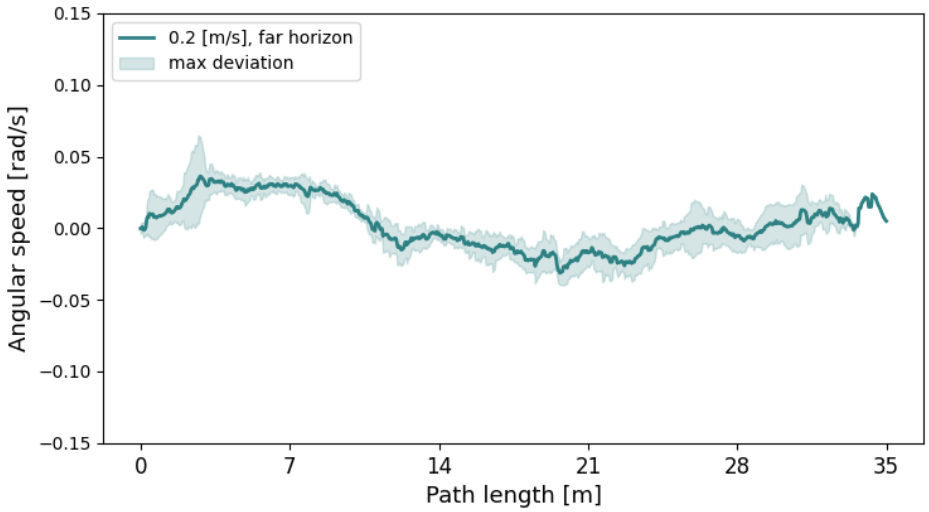


Figure A.6: The average and maximum deviation of the angular velocity of scenario 2

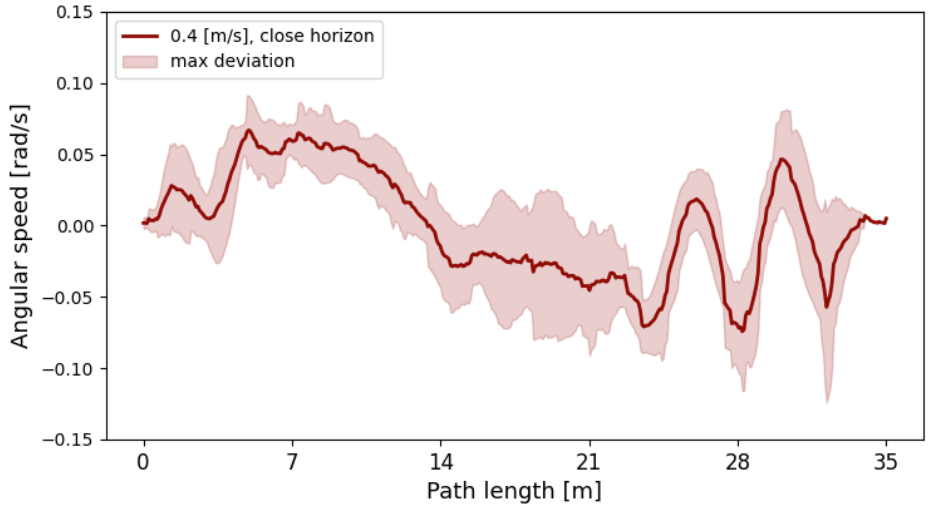


Figure A.7: The average and maximum deviation of the angular velocity of scenario 3

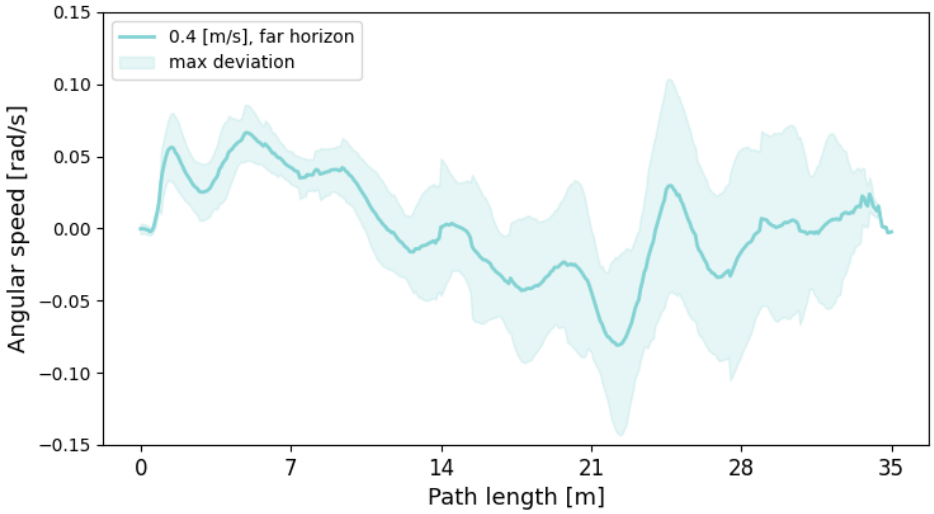


Figure A.8: The average and maximum deviation of the angular velocity of scenario 4

B

Appendix: Code overview for the vehicle

This appendix shows an overview of the code used to control the vehicle. The code is visualized in figure B.1. It uses the markup from ROS2 using nodes and topics. The nodes are displayed in the ovals, and the topics are displayed as the squares. Calculations happen in the nodes. A topic is published by a node, and can be subscribed by another node. A topic can have multiple subscriptions. The first node sends instruction to the camera, such as the required focus point, white balance etc. It also receives images from the camera. After which, it publishes the images at topic /RGB_image. The code that does this is found back in the python code B.1. The image is subscribed by the segmentation node. The segmentation node initializes the Deeplab neural network, after which it performs the segmentation. This node publishes the output of the segmentation, which is called the mask. As one can see, this node uses the Deeplabv3+ library. The python code is found back in B.2 and in B.3. The vision pipeline node subscribes to the output of the segmentation. It extracts the hose pixels. The extracted pixels are the input for the DBSCAN clustering algorithm. Thereafter, the cluster is chosen based on position and size. Polynomial regression is applied to the cluster that represents the hose. Lastly, the points are extracted that will go to the vehicle controller. They are published with a path message type. The code is seen in B.4. Lastly, the vehicle controller node subscribes on that path, and on the odometry of the vehicle. The path points will be converted to the four segments, with the corresponding weights. The information includes the GPS (odometry) and the current velocity and acceleration from the vehicle. Together with that information, a state machine is initialized, such that in calculates the linear and angular velocity. This code is found in B.5.

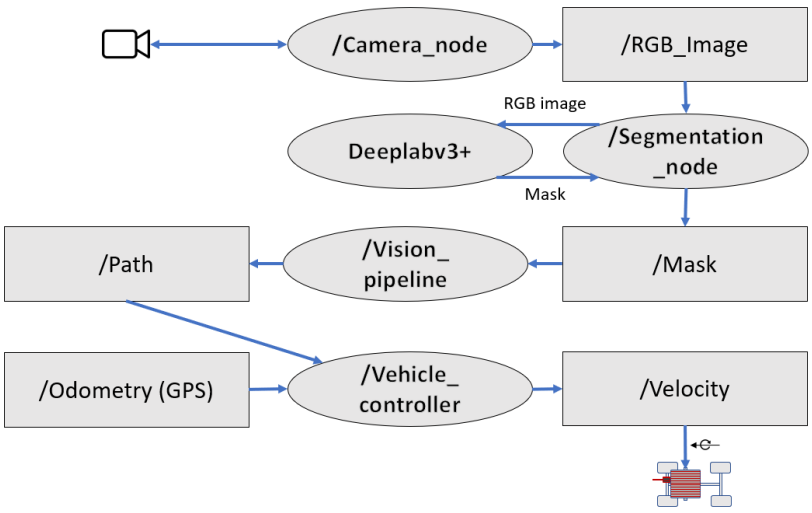


Figure B.1: A schematic overview of the code using nodes and topics as in ROS2

```
'''
@editor: Pieter van Driel
@source code: DepthAI SDK

Global explanation of the code:

Ros works with individual node that can publish or that can subscribe. This node lies connection with the cam
'''

#!/usr/bin/env python3
import rclpy                          # Python library for ROS 2
from rclpy.node import Node           # Handles the creation of nodes
from sensor_msgs.msg import Image     # Image is the message type
from cv_bridge import CvBridge        # Package to convert between ROS and OpenCV Images
import cv2                            # OpenCV library
import depthai as dai                 # Python library to control OAK-D camera
import numpy as np                    # Mathematical library

# Python class that creates the image publisher object
class ImageSubscriber(Node):
    def __init__(self):
        super().__init__('image_publisher')
        self.publisher_ = self.create_publisher(Image, '/image', 10)
        self.br = CvBridge()

# Creating a pipeline to connect to the camera
pipeline = dai.Pipeline()
rclpy.init(args=None)
image_subscriber = ImageSubscriber()

# Define sources and outputs
camRgb = pipeline.create(dai.node.ColorCamera)
videoEncoder = pipeline.create(dai.node.VideoEncoder)
controlIn = pipeline.create(dai.node.XLinkIn)
configIn = pipeline.create(dai.node.XLinkIn)
videoOutput = pipeline.create(dai.node.XLinkOut)
controlIn.setStreamName('control')
configIn.setStreamName('config')
videoOutput.setStreamName('video')

# Properties of the camera and setting up the camera with those properties
W=1920
H=1080
camRgb.setVideoSize(W, H)
videoEncoder.setDefaultProfilePreset(camRgb.getFps(), dai.VideoEncoderProperties.Profile.MJPEG)

# Linking the camera
camRgb.video.link(videoEncoder.input)
controlIn.out.link(camRgb.inputControl)
configIn.out.link(camRgb.inputConfig)
videoEncoder.bitstream.link(videoOutput.input)

# Because the camera uses Ethernet, addition information is required about the camera
device_info = dai.DeviceInfo()
device_info.state = dai.XLinkDeviceState.X_LINK_BOOTLOADER
device_info.desc.protocol = dai.XLinkProtocol.X_LINK_TCP_IP
device_info.desc.name = "192.168.1.3"

# Setting a manual focus, such that the hose is always sharp.
camRgb.initialControl.setManualFocus(50)
count = 0

# Connecting to device and starting the pipeline
with dai.Device(pipeline, device_info) as device:
    timestamp_frame_initial = dai.ImgFrame().getTimestamp()

    # Get data queues
    controlQueue = device.getInputQueue('control')
    configQueue = device.getInputQueue('config')
    videoQueue = device.getOutputQueue('video')

```

```
72      while True:
73          videoFrames = videoQueue.tryGetAll()
74          for videoFrame in videoFrames:
75              # Decoding the image in JPEG format
76              frame = cv2.imdecode(videoFrame.getData(), cv2.IMREAD_UNCHANGED)
77
78              # Transforming the image
79              image_message = image_subscriber.br.cv2_to_imgmsg(frame)
80
81              # Publishing the image
82              image_subscriber.publisher_.publish(image_message)
```

Listing B.1: Camera node

```
'''
@editor: Pieter van Driel
Global explanation of the code:
This node subscribes to the image topic. It initializes the Deeplab neural network, after which it performs t
'''
#!/usr/bin/env python3
import rclpy                            # Python library for ROS 2
from rclpy.node import Node             # Handles the creation of nodes
from sensor_msgs.msg import Image       # Image is the message type
from cv_bridge import CvBridge          # Package to convert between ROS and OpenCV Images
import cv2                              # OpenCV library
# Python code for semantic segmentation
from vision_pipeline.deeplab_tiny import DeepLab, segment_video
import numpy as np                      # Mathematical library

#initializes the neural network as an object to be used
deeplab = DeepLab()

# Python class that creates the image subscriber and mask publisher object
class ImageSubscriber(Node):
    def __init__(self):
        super().__init__('image_subscriber')
        self.subscription = self.create_subscription(Image, '/image', self.listener_callback, 10)
        self.subscription # prevent unused variable warning
        self.publisher_ = self.create_publisher(Image, '/mask', 10)
        self.br = CvBridge()

    # Class function for listening to the incoming images
    def listener_callback(self, data):
        # Convert ROS Image message to OpenCV image
        current_frame = self.br.imgmsg_to_cv2(data)

        # The frame is downscaled such that the images has a size of 360 by 640 pixels
        current_frame = conversion(current_frame)

        # Using the current fram as input for the segmentation mask
        mask = segment_video(deeplab, current_frame)

        # The mask is in bits (black and white). The mask is converted to an RGB image
        mask = np.expand_dims(mask, axis=-1)
        mask = np.concatenate([mask, mask, mask],axis =-1)

        # The Input and the output are stitched together before the image is published
        image = np.concatenate([mask, current_frame], axis=1)
        image_message = self.br.cv2_to_imgmsg(image)
        self.publisher_.publish(image_message)

# Function to downscale the frame, for when the format is wrong
def conversion(frame):
    if frame.size == 691200:
        frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
        frame = cv2.resize(frame, dsize = (360, 640), fx=1, fy=1)
    else:
        pass
    return frame

# The main function to control this node
def main(args=None):
    rclpy.init(args=args)
    image_subscriber = ImageSubscriber()
    rclpy.spin(image_subscriber)
    image_subscriber.destroy_node()
    rclpy.shutdown()

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

Listing B.2: Segmentation Node

```
'''
@editor: Pieter van Driel
```

```
@source code: David8862 -> https://github.com/david8862/tf-keras-deeplabv3p-model-set

Global explanation of the code:
This is the code that takes the RGB images, and calculates the mask using the segmentation network. It loads
'''

#!/usr/bin/env python3
import numpy as np                          # Mathematical library
from PIL import Image                       # Library to converted images
import tensorflow as tf                     # Machine learning platform
from tensorflow.keras import backend as K   # Machine learning platform

# The next form libraries are from David8862. They import functions, in order to let the neural network can o
from tensorflow.keras.models import Model, load_model
from vision_pipeline.deeplabv3p.model import get_deeplabv3p_model
from vision_pipeline.deeplabv3p.postprocess_np import crf_postprocess
from vision_pipeline.common.data_utils import preprocess_image, denormalize_image, mask_resize

# Checks if there is a GPU available
print("Num GPUs Available: ", len(tf.config.list_physical_devices('GPU')))
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
os.environ['CUDA_VISIBLE_DEVICES'] = '0'
optimize_tf_gpu(tf, K)

# Default configs for loading the parameters for the segmenation model
default_config = {
        "model_type": 'xception',
        "classes_path": os.path.join('configs', 'hose_background_classes.txt'),
        "model_input_shape" : (360, 640),
        "output_stride": 16,
        "weights_path": os.path.join('weights', 'segmentation_model.h5'),
        "do_crf": False,
        "pruning_model": False,
        }

# Using the configs, the neural network class is created
class DeepLab(object):
    _defaults = default_config

    @classmethod
    def get_defaults(cls, n):
        if n in cls._defaults:
            return cls._defaults[n]
        else:
            return "Unrecognized attribute name '" + n + "'"

    # Setting up the model
    def __init__(self, **kwargs):
        super(DeepLab, self).__init__()
        self.__dict__.update(self._defaults) # set up default values
        self.__dict__.update(kwargs) # and update with user overrides
        self.class_names = get_classes(self.classes_path)
        K.set_learning_phase(0)
        self.deeplab_model = self._generate_model()

    # Generating the model
    def _generate_model(self):
        # to generate the bounding boxes
        weights_path = os.path.expanduser(self.weights_path)
        assert weights_path.endswith('.h5'), 'Keras model or weights must be a .h5 file.'
        num_classes = len(self.class_names)
        assert len(self.class_names) < 254, 'PNG image label only support less than 254 classes.'

        # Load model, or construct model and load weights.
        try:
            deeplab_model = get_deeplabv3p_model(self.model_type, num_classes, model_input_shape=self.model_i

        except Exception as e:
            print(repr(e))
        return deeplab_model

```

```
    # Using the created network to segment an image
    def segment_image(self, image):
        image_data = preprocess_image(image, self.model_input_shape)
        mage_shape = tuple(reversed(image.size))

        # Making a predicting of the image using the predict funtion
        out_mask = self.predict(image_data, image_shape)
        out_mask_uint8 = np.uint8(out_mask)

        return out_mask_uint8

    # Making a prediction based on the image using the neural network
    def predict(self, image_data, image_shape):
        prediction = self.deeplab_model.predict([image_data])

        # Reshape prediction to mask array
        mask = np.argmax(prediction, -1)[0].reshape(self.model_input_shape)

        # Resize mask back to origin image size
        mask = mask_resize(mask, image_shape[::-1])

        return mask

# When an image is received, it will undergo the following functions once, this function is invoked by the se
def segment_video(deeplab, video_path):
    frame = video_path
    image = Image.fromarray(frame)
    image = deeplab.segment_image(image)

    return image
```

Listing B.3: Deeplab

```
'''
@editor: Pieter van Driel
@source code: sklearn -> https://scikit-learn.org/stable/
Global explanation of the code:

This code represents the rest of the vision pipeline. It subscribes to the output of the segmentation. It ext
'''

#!/usr/bin/env python3
import rclpy                            # Python library for ROS 2
from rclpy.node import Node             # Handles the creation of nodes
from sensor_msgs.msg import Image       # Image is the message type
from cv_bridge import CvBridge          # Package to convert between ROS and OpenCV Images
import cv2                              # OpenCV library
import numpy as np                      # Mathematical library
from nav_msgs.msg import Path           # Path is a message type
from geometry_msgs.msg import PoseStamped   # PoseStamped is message type
from sklearn.cluster import DBSCAN      # Clustering library

# The following three libraries are for the polynomial regression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import RANSACRegressor

# Image subscriber that subscribes to the original image and the mask from the segmentation.
class ImageSubscriber(Node):
    def __init__(self):
        super().__init__('image_subscriber')
        self.subscription = self.create_subscription(Image, '/mask', self.listener_callback, 10)
        self.publisher_ = self.create_publisher(Path, '/mask_points', 1)
        self.br = CvBridge()

    # callback function that receives the data and it calculates every step (sort of main function).
    def listener_callback(self, data):
        # The stitched frame is unstitched.
        frame = self.br.imgmsg_to_cv2(data)
        image = frame[:, 640:1280, :]
        mask = frame[:, 0:640, 0] * 100

        # The birdseye_view function applies the perspective transformation
        bird_mask, bird_img = self.birdseye_view(mask, image)

        # The path_extractor function extracts the hose pixels from the background
        path = self.path_extractor(bird_mask, bird_img)

        # The path_xy function transforms the pixel coordination to real world coordinations
        path_xy = self.pixel_to_xy(path)

        # The coordinates are published
        self.publish_wp(path_xy)









    # Class function to publish the segment points in a 'path' message type. The message is created and fille
    def publish_wp(self, way_points):
        wp = way_points
        time = self.get_clock().now()
        msg = Path()
        msg.header.frame_id = "map"
```

```
        msg.header.stamp = time.to_msg()

        for i in range(len(wp)):
            pose = PoseStamped()
            pose.pose.position.x = float(wp[i][0])
            pose.pose.position.y = float(wp[i][1])
            msg.poses.append(pose)

        self.publisher_.publish(msg)

    # Create an birdseye view from the original image
    def birdseye_view(self, mask, image):

        # Initializing the transformation
        H, W, D = image.shape
        target_size = (W, H)
        ratio = (W*3)/5.9

        # Calculating the transformation matrix
        A = (W-ratio)/2
        B = W-A
        pts1 = np.float32([[0,0], [W,0], [0,H], [W,H]])
        pts2 = np.float32([[0,0], [W,0], [A,H], [W-A,H]])
        M = cv2.getPerspectiveTransform(pts1, pts2)

        # Transforming the image and the mask
        bird_mask = cv2.warpPerspective(mask, M, target_size)
        bird_img  = cv2.warpPerspective(image, M, target_size)

        return bird_mask, bird_img

    def path_extractor(self, mask, image):
        if mask.size != None:
            # Initiating parameters
            V, W = mask.shape
            v = range(V)
            w = range(W)
            dist = 0.0
            hose_label = 0
            max_hose = 0
            min_hose = 0
            hose_path = []

            # Creating a copy of the mask in RGB
            mask_color = np.expand_dims(mask, axis=-1)
            mask_color = np.concatenate([mask_color, mask_color, mask_color],axis =-1)

            # Substracting the labelled pixels from the background
            points_v = []
            points_w = []
            for i in v:
                points_tmp = np.where(mask_tmp[i,:] == 100)
                for j in points_tmp[0]:
                    points_v.append(int(i))
                    points_w.append(int(j))









        # Making clusters within the mask with DBSCAN
        zip_data = np.stack((points_w, points_v), axis = 1)
        if zip_data == []:
            zip_data = None

        if zip_data.any() != None:
            # Creating the clusters
```

```
            clustering = DBSCAN(eps=60, min_samples=100).fit(zip_data)
            labels = clustering.labels_
            hose_dict = {}

            # Coloring the individual clusters
            for i in range(0, len(zip_data)):
                mask_tmp[zip_data[i][1], zip_data[i][0]] = (labels[i]+1) * 100
                mask_color[zip_data[i][1], zip_data[i][0]] = (0,(labels[i]+1) * 100, 0)

                if labels[i] not in hose_dict:
                    hose_dict[labels[i]] = []
                hose_dict[labels[i]].append(zip_data[i])

            # Selecting the largest hose
            for key, value in hose_dict.items():
                # x_points and y_points are hose coordinates from different classes
                x_points, y_points = zip(*value)

                # Extracting the maximum and minimum from every cluster
                max_point = (x_points[np.argmax(y_points)], y_points[np.argmax(y_points)])
                min_point = (x_points[np.argmin(y_points)], y_points[np.argmin(y_points)])

                # Calculating the maximum distance for each cluster
                eucl_dist = np.sqrt((max_point[0]-min_point[0])**2 +
                            (max_point[1]-min_point[1])**2)

                # Selecting the largest cluster
                if eucl_dist > dist:
                    dist = eucl_dist
                    hose_label = key
                    max_hose = max_point
                    min_hose = min_point

            # Checking if the cluster starts at the bottom of the frame
            if max_point[1] > 300:
                hose_x, hose_y = zip(*hose_dict[hose_label])
                label_range = np.linspace(max_hose[1], min_hose[1], num=5, dtype=np.uint16)

                # Making the polynomial prediction based on the chosen cluster
                X = np.array(hose_y).reshape(-1,1)
                y = hose_x
                model = make_pipeline(PolynomialFeatures(4), RANSACRegressor())
                model.fit(X, y)

                # Making the prediction
                predictions = model.predict(np.arange(start=0, stop=V, step=1).reshape(-1,1))
                predict_small = model.predict(label_range.reshape(-1,1))

                # Visualizing the polynomial
                for i in v:
                    j = int(predictions[i])
                    if j in range(0, W):
                        mask_tmp[int(i), j] = 250

                # Visualizing the different segment points
                for i in range(len(label_range)):
                    cv2.circle(mask_color, (int(predict_small[i]), label_range[i]), 3,
                    (255,255,255), 2)
                    cv2.circle(image, (int(predict_small[i]), label_range[i]), 3,
                    (255,255,255), 2)
                    hose_path_it = [int(predict_small[i]), label_range[i]]
                    hose_path.append(hose_path_it)

        return hose_path

    def pixel_to_xy(self, path):
        scaling_factor_y = 4.59/360
        scaling_factor_x = 6.71/640
        offset_x = 6.71/2
        offset_y = 5.08
```

```
214         for i in range(len(path)):
215             point_tmp_x = path[i][0]*scaling_factor_x
216             point_tmp_y = path[i][1]*scaling_factor_y
217             path[i][0] = np.round(((point_tmp_x*-1) + offset_x)*-1, 2)
218             path[i][1] = np.round(((point_tmp_y*-1) + offset_y), 2)
219 
220         return path
221 
222 # Main function to start the node
223 def main(args=None):
224     rclpy.init(args=args)
225     image_subscriber = ImageSubscriber()
226     rclpy.spin(image_subscriber)
227     image_subscriber.destroy_node()
228     rclpy.shutdown()
229 
230 if __name__ == '__main__':
231     main()
```

Listing B.4: Vision pipeline

```
'''
@editor: Pieter van Driel
Global explanation of the code:
This action node combines all the information required to give commands to robot, such that it is going to mo

'''

#!/usr/bin/env python3
import rclpy                            # Python library for ROS 2
from rclpy.node import Node             # Handles the creation of nodes
from jojo_msgs.action import Hosefollow # Action message for the vehicle controller
from geometry_msgs.msg import Twist     # Twist is a message type
from nav_msgs.msg import Path, Odometry # Path and Odometry are messages types
import math                             # Mathematical library
import numpy as np                      # Mathematical library
# Libraries that handle the action server
from rclpy.action import ActionServer, CancelResponse, GoalResponse
from rclpy.executors import MultiThreadedExecutor

# Creating the hose following action server class
class HoseFollowActionServer(Node):
    def __init__(self):
        self.timer_period = 0.1
        super().__init__('hose_follow_action_server')
        self._action_server = ActionServer(self, Hosefollow, 'hosefollow',
            execute_callback    = self.execute_callback,
            goal_callback       = self.goal_callback,
            cancel_callback     = self.cancel_callback,)

        self.subscription = self.create_subscription(Path, '/mask_points',
            self.mask_points_callback, 10)
        self.subscription = self.create_subscription(Odometry, '/odom',
            self.odom_callback, 10)
        self.publisher_ = self.create_publisher(Twist, '/cmd_vel', 1)
        self.timer = self.create_timer(self.timer_period, self.timer_callback)

        self.odometry       = None      # Odometry parameter
        self.max_speed      = None      # Maximum speed
        self.twist_lin_odom = None      # Linear odometry
        self.twist_ang_odom = None      # Angular odometry
        self.delta = 0.1                # Step size of the PID
        self.max_angle = 150            # Maximum hose angle
        self.min_angle = 30             # Minimum hose angle
        self.error = 0                  # PID error
        self.prev_error = 0             # previous PID error
        self.prev_prev_error = 0        # Previous eror of the error of the PID
        self.output = 0                 # Output of the linear velocity of the PID
        self.prev_output = 0            # Previous output
        self.angle = 0                  # Steering angle
        self.x = 0                      # First x segment coordinate
        self.y = 0                      # First y segment coordinate
        self.x_look_ahead = 2.55        # Camera offset in x direction
        self.y_look_ahead = 0           # Camera offset in y direction
        self.state = 0                  # State of the state machine

    # Getting the odometry data
    def odom_callback(self, data):
        self.odometry = data
        self.twist_lin_odom = self.odometry.twist.twist.linear
        self.twist_ang_odom = self.odometry.twist.twist.angular

        return




    # Getting the path from the vision pipeline
    def mask_points_callback(self, data):
        hose_point = data
        x = []
        y = []
```

```
        header_hose  = hose_point.header
        path_hose    = hose_point.poses
        for i in range(len(path_hose)):
            x_i = path_hose[i].pose.position.x
            y_i = path_hose[i].pose.position.y
            x.append(x_i)
            y.append(y_i)

        # Calculating the average hose angle
        self.angle = self.calculate_angle(x, y)

        return

    # Goal callback
    def goal_callback(self, goal_request):
        # Accept or reject a client request to begin an action
        self.get_logger().info('Received goal request')

        return GoalResponse.ACCEPT

    # Cancel callback
    def cancel_callback(self, goal_handle):
        # Accept or reject a client request to cancel an action
        self.get_logger().info('Received cancel request')

        return CancelResponse.ACCEPT

    # Execution callback is executed once when it receives a goal
    def execute_callback(self, goal_handle):
        # Initializng the maximum speed, velocity is negative because the vehicle drives backwards
        self.max_speed = -1*(goal_handle.request.maxspeed)
        self.state = 1
        condition = 1
        feedback_msg = Hosefollow.Feedback()
        result_msg = Hosefollow.Result()
        goal_handle.goal_id

        # Condition 1 means that the end of the hose has not been found
        # Condition 2 means that the end of the hose is found
        while condition == 1:
            feedback_msg.feedback.data = "Driving on hose ..."
            goal_handle.publish_feedback(feedback_msg)
            if self.state == 3:
                condition = 2
                print('The end of the hose has been found')
                goal_handle.succeed()
            elif goal_handle.is_cancel_requested == True or self.state == 2:
                print('Goal is canceled')
                goal_handle.canceled()
                self.state = 2
                condition = 2

        # When condition 2 holds, it should break until the vehicle stops entirely
        while condition == 2:
            if self.twist_lin_odom.x < -0.05:
                self.state = 2
            else:
                self.state = 0
                condition = 0

        return result_msg



    # The timer callback sends a command to the vehicle controller every x miliseconds
    def timer_callback(self):
        cmd_vel = Twist()
        if self.odometry != None and self.max_speed != None:
            # Clipping the angle
            self.y_look_ahead = math.tan((self.angle - 90)*np.pi/180) * self.x_look_ahead
```

```
            self.delta_z = -(2.0 * self.y_look_ahead) /
                            (self.x_look_ahead**2 + self.y_look_ahead**2)

            # When the last point is closer than 3.5 meters away from the vehicle,
            # it goes to state 3
            if self.y < 3.5 and self.state != 0:
                self.state = 3

            # State 0: standing still
            if self.state == 0:
                pass

            # State 1: driving normally
            elif self.state == 1:
                # Hose is relative straight: the PID has an maximum velocity as input
                if self.angle > 65 and self.angle < 115:
                    self.calculate_speed_pid(self.twist_lin_odom.x,
                        k_p = 0.5, k_i = 0.6, k_d= 0.0, set_point = self.max_speed)
                    cmd_vel.linear.x = np.clip(self.output, (self.max_speed-0.1), 0.05)
                    cmd_vel.angular.z = cmd_vel.linear.x * self.delta_z
                    self.publisher_.publish(cmd_vel)

                # The hose angle is too sharp for a maximum velocity:
                # the PID gets a velocity input based on the sharpness of the hose
                elif self.angle > 30 and self.angle < 65:
                    adaptive_speed = ((self.angle-30)/(65-30)) * self.max_speed
                    self.calculate_speed_pid(self.twist_lin_odom.x,
                        k_p = 0.5, k_i = 0.6, k_d= 0.0, set_point = adaptive_speed)
                    cmd_vel.linear.x = np.clip(self.output, (self.max_speed-0.1), 0.05)
                    cmd_vel.angular.z = cmd_vel.linear.x * self.delta_z
                    self.publisher_.publish(cmd_vel)

                # The hose angle is too sharp for a maximum velocity:
                # the PID gets a velocity input based on the sharpness of the hose
                elif self.angle > 115 and self.angle < 150:
                    adaptive_speed = (1-((self.angle-115)/(150-115))) * self.max_speed
                    self.calculate_speed_pid(self.twist_lin_odom.x,
                        k_p = 0.5, k_i = 0.6, k_d= 0.0, set_point = adaptive_speed)
                    cmd_vel.linear.x = np.clip(self.output, (self.max_speed-0.1), 0.05)
                    cmd_vel.angular.z = cmd_vel.linear.x * self.delta_z
                    self.publisher_.publish(cmd_vel)

                # The hose angle is too sharp to continue:
                # entering the braking state
                else:
                    self.state == 2

            # State 2: braking state
            elif self.state == 2:
                self.calculate_speed_pid(self.twist_lin_odom.x,
                    k_p = 0.5, k_i = 0.6, k_d= 0.0, set_point = 0)
                cmd_vel.linear.x = np.clip(self.output, (self.max_speed-0.1), 0.05)
                cmd_vel.angular.z = 0.0
                self.publisher_.publish(cmd_vel)

            # State 3: The hose is to short: goal reached
            elif self.state == 3:
                self.calculate_speed_pid(self.twist_lin_odom.x,
                    k_p = 0.5, k_i = 0.6, k_d= 0.0, set_point = 0)
                cmd_vel.linear.x = np.clip(self.output, (self.max_speed-0.1), 0.05)
                cmd_vel.angular.z = 0.0
                self.publisher_.publish(cmd_vel)

        return

    # Calculating the speed according to input. A PID is used
    def calculate_speed_pid(self, v_in, k_p, k_i, k_d, set_point):
        self.error = set_point - v_in
        p = k_p * (self.error - self.prev_error)
        i = k_i * (self.delta * self.error)
        d = k_d / (self.delta) * (self.error - 2*self.prev_error+self.prev_prev_error)
```

```
214
215         self.output = self.output + p + i + d
216         self.prev_prev_error = self.prev_error
217         self.prev_error  = self.error
218         self.prev_output = self.output
219
220     # Calculating the angle of hose with respect to the vehicle
221     def calculate_angle(self, x, y):
222         self.y = y[-1]
223         self.x = x[-1]
224         angle = []
225         angle_avg = None
226
227         # The calculated angle is in degrees
228         for i in range(len(x)-1):
229             angle_it = math.atan2((y[i+1]-y[i]), x[i+1]-x[i])*(180/math.pi)
230             angle.append(angle_it)
231             print(angle_it, y[i], x[i])
232
233         # The weights of the segments are added
234         if angle != []:
235             angle_avg = ((angle[0]*2) + (angle[1]*1.5) + (angle[2]*1.0) + (angle[3]*0.5)) / 5
236
237         return angle_avg
238
239 # Main fuction to start the node
240 def main(args=None):
241     rclpy.init(args=args)
242     executor = MultiThreadedExecutor(num_threads=4)
243
244     hose_follow_action_server = HoseFollowActionServer()
245     executor.add_node(hose_follow_action_server)
246     executor.spin()
247
248
249 if __name__ == '__main__':
250     main()
```

Listing B.5: Vehicle controller

C

Appendix: Python code for training the neural network

C.1. Camera Calibration

The camera calibration of the intrinsic parameters is done using the tutorial from the makers of the camera [9]. The tutorial is found at `https://docs.luxonis.com/en/latest/pages/calibration/`.

The calibration of the excentric parameters are done using the following code:

```
'''
@editor: Pieter van Driel

Global explanation of the code:
In the vision pipeline node from appendix-b, coordinates are required to calculate the matrix, in order to cr
'''

#!/usr/bin/env python3
import numpy as np                  # Mathematical library
import matplotlib.pyplot as plt     # Plotting library

# Camera variables
cam_pos = np.array([0,0,2.14])      # Position of the camera with respect to the rear axle
alpha = (np.pi/180)*40              # Angle of the camera
beta  = (np.pi/180)*54.3            # Vertical field of view
gamma = (np.pi/180)*68.8            # Horizontal field of view
# Rotational matrix for the y-axis
rot = np.array([[np.cos(alpha),  0, np.sin(alpha)],
                [0,              1, 0            ],
                [-np.sin(alpha), 0, np.cos(alpha)]])

# Function to calculate the four coordinates of the four corners of the image
def cam_FOV(cam_pos, beta, gamma):
    points = np.zeros((4,3), dtype=float)
    points[0] = [cam_pos[2]*np.tan(beta/2),  -cam_pos[2]*np.tan(gamma/2), 0]
    points[1] = [-cam_pos[2]*np.tan(beta/2), -cam_pos[2]*np.tan(gamma/2), 0]
    points[2] = [-cam_pos[2]*np.tan(beta/2), cam_pos[2]*np.tan(gamma/2),  0]
    points[3] = [cam_pos[2]*np.tan(beta/2),  cam_pos[2]*np.tan(gamma/2),  0]

    return points

# Function to rotate the the Field of View using the angle of the camera
def rotation(cam_pos, points):
    rot_points = np.zeros((4,3), dtype=float)
    for i in np.arange(len(points)):
        rot_points[i] = np.subtract(points[i], cam_pos) @ rot + cam_pos

    return rot_points

# Function to calculate the unity vecotors of the ribs of the FoV of the camera
```

```
def unit_vec(cam_pos, rot_points):
    u_vec = np.zeros((4,3), dtype=float)
    for i in range(len(rot_points)):
        u_vec[i] = (np.subtract(rot_points[i], cam_pos) / np.linalg.norm(np.subtract(rot_points[i], cam_pos))
        u_vec[i] = np.add(u_vec[i], cam_pos)

    return u_vec

# Function to calculate the intersection points between the Field of View and the z-plane
def actual_FOV(cam_pos, u_vec):
    plane_points = np.zeros((4,3), dtype=float)

    for i in range(len(u_vec)):
        temp = np.subtract(cam_pos, u_vec[i])
        t = cam_pos[2]/temp[2]*-1
        plane_points[i][0:2] = temp[0:2] * t
    print(plane_points)
    return plane_points

# Supportive function to use a 3D plot. Used from:
# https://stackoverflow.com/questions/13685386/matplotlib-
# equal-unit-length-with-equal-aspect-ratio-z-axis-is-not-equal-to
def set_axes_equal(ax: plt.Axes):
    """Set 3D plot axes to equal scale.

    Make axes of 3D plot have equal scale so that spheres appear as
    spheres and cubes as cubes.  Required since `ax.axis('equal')`
    and `ax.set_aspect('equal')` don't work on 3D.
    """
    limits = np.array([
        ax.get_xlim3d(),
        ax.get_ylim3d(),
        ax.get_zlim3d(),
    ])
    origin = np.mean(limits, axis=1)
    radius = 0.5 * np.max(np.abs(limits[:, 1] - limits[:, 0]))
    _set_axes_radius(ax, origin, radius)

def _set_axes_radius(ax, origin, radius):
    x, y, z = origin
    ax.set_xlim3d([x - radius, x + radius])
    ax.set_ylim3d([y - radius, y + radius])
    ax.set_zlim3d([z - radius, z + radius])

# Plotting the Field of View
def plot(cam_pos, pp):
    origin = np.array([0,0,0])
    normal = np.array([0,0,1])
    d = -origin.dot(normal)
    xx, yy = np.meshgrid(np.arange(-5,5,1),np.arange(-5,5,1))
    z = (-normal[0] * xx - normal[1] * yy -d)*1. /normal[2]
    surface = [xx, yy, z]

    # Plotting the FOV surface
    fig = plt.figure()
    ax = plt.axes(projection='3d')
    orange = (0.77,0.35,0.07)
    blue = (0.26, 0.44, 0.73)
    ax.scatter3D(cam_pos[0], cam_pos[1], cam_pos[2], color=orange)
    ax.scatter3D(pp[:,0], pp[:,1], pp[:,2], color = orange)
    plane_x = np.array([[pp[1][0],pp[0][0]], [pp[2][0],pp[3][0]]])
    plane_y = np.array([[pp[1][1],pp[0][1]], [pp[2][1],pp[3][1]]])
    plane_z = np.array([[pp[1][2],pp[0][2]], [pp[2][2],pp[3][2]]])

    ax.plot_surface(plane_x, plane_y, plane_z, color=orange, alpha = 0.5)



    # plotting the cam_vectors
    for i in range(4):
        ax.plot([pp[i,0],cam_pos[0]],
```

```
                [pp[i,1],cam_pos[1]],
                [pp[i,2],cam_pos[2]], color = blue)

    # Figure markup
    ax.set_xlabel('x [m]', labelpad=20)
    ax.set_ylabel('y [m]', labelpad=20)
    ax.set_zlabel('z [m]', labelpad=20)
    ax.set_box_aspect([1,1,1])
    set_axes_equal(ax)
    ax.set_xlim(-0,5)
    ax.set_ylim(-4,4)
    ax.set_zlim(0,3)
    plt.show()

def main():
        points = cam_FOV(cam_pos, beta, gamma)
        rot_points = rotation(cam_pos, points)
        unity_vec = unit_vec(cam_pos, rot_points)
        plane_points = actual_FOV(cam_pos, unity_vec)
        plot(cam_pos, plane_points)

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

Listing C.1: Camera calibration

C.2. Data augmentation

```
'''
@editor: Pieter van Driel

Global explanation of the code:
This code will augment the original images. First it loads the images, then it augments them, lastly, they ar
'''

#!/usr/bin/env python3

import os
import numpy as np
import cv2
from glob import glob
from tqdm import tqdm
from albumentations import HorizontalFlip, GridDistortion, OpticalDistortion, ChannelShuffle, CoarseDropout,

# Creating a directory
def create_dir(path):
    if not os.path.exists(path):
        os.makedirs(path)

# Loading the images and labels
def load_data(path_img, path_lab, split = 0.15):

    if printing == True:
        for x, y in zip(X, Y):
            print("Found X and Y:")
            print(x, y)
            print("length = ", len(X))
            break

    return X, Y



def augment_data(images, labels, save_path_img, save_path_lab, augment=True):
    H = 360
    W = 640
    src_H = 360
    src_W = 640
```

```
42
43      for x, y in tqdm(zip(images, labels), total=len(images)):
44          # Extract the name
45          name_x = x.split("/")[-1].split(".")[0]
46          name_y = y.split("/")[-1].split(".")[0]
47
48          # Reading the image and label
49          x = cv2.imread(x, cv2.IMREAD_COLOR)
50          y = cv2.imread(y, 0)
51          y = y//100
52
53          # Augmentation
54          if augment == True:
55              x1 = cv2.flip(x, 1)
56              y1 = cv2.flip(y, 1)
57
58              img = np.zeros((src_H, src_W , 3), np.uint8)
59              img[:] = (40, 40, 40)
60              x2 = cv2.add(x, img)
61              y2 = y
62
63              aug = CoarseDropout(p=1, min_holes=5, max_holes=15, max_height=10, max_width=10)
64              augmented = aug(image=x, label=y)
65              x3 = augmented['image']
66              y3 = augmented['label']
67
68              x4 = cv2.subtract(x, img)
69              y4 = y
70
71              X = [x, x1, x2, x3, x4]
72              Y = [y, y1, y2, y3, y4]
73
74          else:
75              X = [x]
76              Y = [y]
77
78          index = 0
79          for i, m in zip(X, Y):
80              i = cv2.resize(i, (W, H),fx=1, fy=1)
81              m = cv2.resize(m, (W, H),fx=1, fy=1)
82
83              tmp_image_name = f"{name_x}_{index}.jpg"
84              tmp_label_name = f"{name_y}_{index}.png"
85
86              image_path = os.path.join(save_path_img, tmp_image_name)
87              label_path = os.path.join(save_path_lab, tmp_label_name)
88
89              cv2.imwrite(image_path, i)
90              cv2.imwrite(label_path, m)
91
92              index += 1
93
94  if __name__ == "__main__":
95      printing = False
96      load_data_path_img = "dataset_1/images"
97      load_data_path_lab = "dataset_1/labels"
98      save_data_path_img = "augmented_data/images"
99      save_data_path_lab = "augmented_data/labels"
100
101     # Load the dataset
102     X, Y = load_data(load_data_path_img, load_data_path_lab)
103
104     # Data augmentation
105     augment_data(X, Y, save_data_path_img, save_data_path_lab, augment=True)
```

Listing C.2: Camera calibration

C.3. Training the Deeplabv3+ neural network

The training and evaluation is done using the Deeplabv3+ tutorial found at `https://github.com/david8862/tf-keras-deeplabv3p-model-set` [10]. The configuration to train the model is as follows:

```
python3 train.py --model_type=xception --weights_path=weights/deeplabv3_xception_tf_dim_ordering_tf_kernels_c
```

The backbone used is the Xception model. The pre-trained weights originate from the Xception model trained by Cityscapes dataset. The input shape is 360X640 pixel. The dataset differed, in this case the model is trained using dataset 1 and 2 without the augmented images. The val.txt and train.txt files contain frame numbers. The classes.txt contains the classes found in our model. The batch size is 2, the optimizer used in the neural network is SGD.

When the model is trained, it can be used to show the results, by using images or by using a video, the configuration to show the results is as follows:

```
python3 deeplab.py --model_type=xception --weights_path=logs/000/trained_final.h5 --classes_path=configs/clas
```