Implementation of robotic manipulator for use in weed management in the agricultural industry

Thomas Kjær Nowak and Christian Tranholm Novrup

Applied Industrial Electronics
Aalborg University Esbjerg AIE6-2-F22
June 10, 2022

Bachelor project (6th semester)

LaTeX is used for formatting this report and all images which are not cited are made by the authors.

AALBORG UNIVERSITY

STUDENT REPORT

Title:
Implementation of robotic manipulator for use in weed management in the agricultural industry

Theme:
Modeling and control of robotic manipulator

Project Period:
Spring Semester 2022

Project Group:
AIE6-2-F22

Participants:
Christian Tranholm Novrup
Thomas Kjær Nowak

Supervisors:
Petar Durdevic Løhndorf
Daniel Ortiz Arroyo

Page Numbers: 85

Date of Completion: June 10, 2022

Abstract:

This project is based on an concept, proposed by Mostec ApS, Klovborg I/S and AAU, where a small robot will move around on a field used for growing crops, with a mounted robotic manipulator on the robot, which can remove weeds from the field. An Intel L515 LiDAR camera was mounted to the end effector of the manipulator, and the images from the camera was passed through a neural network (YOLOv5n), that was trained to recognize a specific plant through transfer learning, which used automatic image collection, labelling and augmentation. The network returns a position of the plant on the image, which is used together with a distance reading to compute an error between the gripper and the plant. The error is passed to a P-controller, that will send velocities in the x, y and z direction to the end effector, making it move towards the plant. In the case of detection loss, a state space model based estimator was implemented, that will estimate the error, when the neural network cannot detect a plant, and pass the estimated error to the P-controller instead. During the testing of the complete system it was found that most of the requirements could be met. Even though all of the requirements could not be met, the overall performance of the system setup proved to be good, although with room for improvements.The findings shown in this report can thereby be a basis for further development of this technology.

List of Figures

2.1 Weed combatting robot [23] . . . 7

3.1 Block diagram of the system on an overall level . . . 13
3.2 Setup of the manipulator . . . 14
3.3 Code structure diagram . . . 14

4.1 Frames for the Kinova robotic manipulator [32] . . . 15
4.2 Forward and inverse kinematics . . . 16
4.3 3D plot which shows a point in space 'P' . . . 18

5.1 RGB image (left) and depth colormap (right). Images are set to the same resolution 25
5.2 Central projection of a pinhole camera model . . . 26
5.3 Projection of Fig. 5.2 on the zy-plane . . . 26
5.4 Representation of the world point with respect to another reference frame . . . 28
5.5 3D rendering of the camera mount and the offset wrt. the gripper origin . . . 31

6.1 Figures shows how the kernel, in the Convolutional layer, can reduce data size . . . 35
6.2 The pooling used is the 'max pooling' which takes the highest value and transfers to the output . . . 35
6.3 Overview of a neural network . . . 36
6.4 Weed1 . . . 39
6.5 Weed2 . . . 39
6.6 Different images of the weed . . . 39
6.7 Plant bounding box in 3D . . . 39
6.8 Projection of plant bounding box on image . . . 40
6.9 Bounding boxes on the images shown in Fig. 6.6 . . . 41
6.10 Post processing which shows the data augmentation of the weeds shown in Fig. 6.9 42
6.11 The yaml file content is shown in listing 6.3, and the folder structure can be seen in Fig. 6.11a . . . 43
6.12 Images examples from Weed7 network . . . 45
6.13 mAP comparison graph for the different training runs . . . 47

7.1 Results of experiment . . . 50
7.2 Model and experiment comparison . . . 53
7.3 Block diagram of the estimator . . . 55

8.1 General control system block diagram . . . 57
8.2 Flowchart of the final system . . . 65

9.1 One of the testing setups . . . 66
9.2 Plot of gripper position and plant during general test. Dashed line is when the estimator is working . . . 67

9.3 Plot of error vector between gripper and plant during general test. Dashed line is when the estimator is working. True error is given by the end effector position (measured) subtracted from the plant position, relative to the base 68
9.4 Plant far away from camera (left) and plant close to the camera (right) 69
9.5 Plot of gripper position and plant during estimator test. Dashed line is when the estimator is working 70
9.6 Plot of error vector between gripper and plant during estimator test. Dashed line is when the estimator is working. True error is given by the end effector position (measured) subtracted from the plant position, relative to the base 71
9.7 Confidence vs distance test for the plant and 2 other items: ethernet cable and a yellow cable 72

List of Tables

4.1 DH parameters for the Kinova robotic manipulator [32] 20

6.1 Comparison table of different transfer learning runs 47

7.1 Model parameters 52

Table of contents

List of Figures **iv**

List of Tables **vi**

Preface **x**

1 Introduction **1**
1.1 Initiating problem . . . 1

2 Problem analysis **2**
2.1 Weeds . . . 2
2.1.1 Relevant types of weed . . . 3
2.1.2 Other threats to crops . . . 3
2.2 Weed control . . . 3
2.2.1 Conventional weed control . . . 3
2.2.2 Organic based weed control . . . 4
2.2.3 Weed combatting in Klovborg I/S . . . 5
2.2.4 Weed control using agricultural robots . . . 5
2.3 Setup . . . 6
2.3.1 Robotic manipulator (Kinova Gen3 Lite) . . . 7
2.3.2 Intel RealSense LiDAR Camera L515 . . . 8
2.3.3 Nvidia Jetson Nano Developer Kit . . . 8
2.4 Objectives and goals . . . 8
2.4.1 Moving the arm . . . 8
2.4.2 Visual servoing . . . 9
2.4.3 Weeds detection . . . 9
2.4.4 Goals of this project . . . 10
2.4.5 Delimitation . . . 10
2.4.6 Requirements . . . 11

3 System overview **13**
3.1 System block diagram . . . 13
3.2 Code structure . . . 14

4 Robot manipulator **15**
4.1 Controlling of the manipulator . . . 15
4.1.1 Manipulator . . . 15
4.1.2 Forward and inverse kinematics . . . 16
4.1.3 Reference frame . . . 16
4.1.4 Rotation matrices . . . 17
4.1.5 Composite rotation matrix . . . 17
4.1.6 Homogeneous transformation . . . 17
4.1.7 Denavit-Hartenberg Parameters . . . 19

4.2 Kinova Kortex API 20

5 Object localization **22**
5.1 Camera 22
5.1.1 Setup and initialization 22
5.1.2 Retrieval of data 23
5.2 3D coordinate from image 25
5.2.1 Projective geometry and camera model 25
5.2.2 Image plane projection in different reference frames 28
5.2.3 3D position from 2D image projection 30
5.2.4 Obtaining translation vector and rotation matrix 30
5.3 Implementation of projections 32

6 Object detection with deep neural network **34**
6.1 Convolutional neural network 34
6.1.1 Training and transfer learning of a network 36
6.1.2 YOLOv5 and PyTorch 37
6.2 Transfer learning with YOLOv5 37
6.2.1 Gathering images 38
6.2.2 Automatic drawing of bounding box 39
6.2.3 Post processing 41
6.2.4 Practical transfer learning 42
6.2.5 Comparison of different transfer learning runs 45
6.3 Implementation of the neural network 47

7 State estimation **49**
7.1 Model selection and identification 50
7.1.1 Benefits and limitations of chosen model 53
7.2 Estimator design 54
7.3 Estimator implementation 54

8 Control **57**
8.1 Measuring the error 58
8.2 Controller design 60
8.2.1 Controller gains 61
8.3 Controller implementation 62
8.4 Full implementation 63

9 Testing **66**
9.1 General function and performance 66
9.2 Performance during detection loss 69
9.3 Neural network performance 71

10 Discussion / Perspectives **73**
10.1 Neural network . . . 73
10.2 Controller . . . 73
10.2.1 Error measurement . . . 74
10.2.2 Controlling the manipulator . . . 75
10.3 Estimator . . . 76
10.3.1 Model selection . . . 77
10.3.2 Kalman filter . . . 77
10.3.3 Parallel processes . . . 77

11 Conclusion **79**

Bibliography **80**

A Transformation matrices for the Kinova robotic arm **85**

Preface

This project was made as a final bachelor thesis of the education B.Sc. in Applied Industrial Electronics offered by Aalborg University in Esbjerg. This project is created within the theme of "Modelling and Control of Robotic Systems" the purpose of this report is to get a general and fundamental understanding within the topic of robot manipulators, aswell as general project mangement skills.
Every mention of "we", "us" or "the group" refers to the two co-authors listed below.

Aalborg University, June 10, 2022

Thomas Kjær Nowak
<tnowak18@student.aau.dk>

Christian Tranholm Novrup
<cnovru19@student.aau.dk>

Chapter 1

Introduction

In the world today, there is an estimated population of roughly 7.7 billion people, compared to around 1 billion living back in the 1800 century, and the world population tends to only increase as time passes, although not necessarily with the same rate [1]. In the future, bigger demands will automatically be set to the agricultural sector in order to continuously provide food to the population, and livestock on farms. Conventional farming, with the usage of herb- and pesticides, has already been under heavy regulation, since it has shown that excess use of herb- and pesticides can contaminate the ground water. Contamination of the groundwater is of course something the society should keep to a minimum, since it is heavily dependant on clean and safe groundwater to be used for eg. drinking water [2]. During recent years, organic farming, which is opposed to conventional farming, has become increasingly more widespread, with 310.210 ha in 2020, equal to 11.7% of the total agricultural area in Denmark, and has been increasing each year since 2015 [3]. Organic farming does not allow use of herb- and pesticides, promotes animal welfare, and is generally aimed to be more environmentally friendly compared to conventional farming [4] [5]. A downside of organic farming, is the decreased yield of product, and thereby increasing the price of a product, compared to traditionally farmed products, since other working principles is required to produce the same amount of food [6]. This can, among other things, be due to unwanted plants, also known as weeds, growing in the same soil as the crops, that is to be harvested, as the weeds "steal" nutrients in the soil, from the actual crops [7]. Since organic farming does not allow use of herb- and pesticides, other measures must be taken, eg. weed removing machinery on tractors, or manual removal of weeds [8]. Common for these methods, is that they are expensive and somewhat inefficient, especially compared to using herb- and pesticides.
Based on the increasing use and research related to the use of robots in agriculture [9], a concept has been proposed by Aalborg University, Mostec ApS and Klovborg I/S, which is to develop a robot with a manipulator (robotic arm), that can drive on a field with crops, and detect, locate, and remove weeds, with the manipulator. This way, weeds can be removed from a field without the usage of manual labor, and generally with as little human interaction as possible. This report will therefore focus on the topic of weed removal using a robot manipulator. This focus area has resulted in the initiating problem:

1.1 Initiating problem

"How can a robot manipulator be used to remove weeds?"

Chapter 2
Problem analysis

2.1 Weeds

Weeds in agriculture can be defined as a plant, that grows where it is not supposed to, and requires active management to keep it from interfering with crops or livestock production [10]. These weeds are a problem for farmers, as they often use the same nutrients and resources, as the crops that grows on the fields. By nutrients and resources, is meant minerals in the soil, sunlight, water and space. In fact, the more similar the weed types and crops are, the more the two types of plants will compete for the same nutrients and resources in the soil. Generally, the weeds that compete aggressively with the crops, can result in decreased crop yield, and the weeds that have an advantage over the crop, are usually the most damaging ones [7]. This essentially means, that the crops and weeds will have to share the same resources, resulting in a lower yield of crops. Only the crops are harvested and sold by the farmer, meaning a lower yield of crops gives a lower income for the farmer, which is of course not desirable. Furthermore, if the farmer uses some kind of fertilizer and actively waters the crops, then some of that water and fertilizer will also be used by the weeds, if the weeds are not kept under control. Where the conditions are right, something will grow whether it is intended or not. The farmer is naturally not interested in spending fertilizer and water, and thereby money, on the weeds which brings no income, and make conditions worse for the crops. Weeds can also actively inhibit the growth of crops on fields, either by releasing chemicals in the soil that are harmful to the crops (allelopathy), the weed itself being poisonous and contaminating the crops, being home to harmful insects and diseases, and cause problems during harvest, especially if the weeds have strong stems and reach above the crops [7]. If weeds are not kept under control, they might reproduce significantly, only increasing the problem, and especially invasive weeds that are not native to the area they grow in, meaning they have been imported at some point, can grow and spread faster, and can be more difficult to keep down [10].
Even though weeds are undesired by the farmer, as it brings down crop yield, there are also benefits to weed. Weed growth can prevent erosion from wind and rain of the topsoil, as the roots bind the soil, and the plant itself covers and protects the topsoil. Growth of weeds is nature's own way of protecting and restoring the soil it grows on. Weeds can also be used to restore degraded soils and provide good living conditions for organisms which are beneficial for the ecosystem, which in some cases can be a natural way of controlling some insect pests. Furthermore, weeds also perform photosynthesis, and contribute to converting carbon dioxide into oxygen, which is of great importance to the ecosystem. Some weeds can also be nutritious and be used for food and/or fodder, but this might also mean that it will steal the nutrients from the actual crops [10]. Overall it is important to notice that a plant can be seen as a 'weed' hereby meaning it is undesired in one place, for different reasons, but seen as a crop in other places where it is actively been farmed for food or fodder.

2.1.1 Relevant types of weed

Weeds are in general a broad category of plants, and what categorizes a plant as a weed, depends on several factors such as the crop and region, so it is a difficult thing to accurately define. In relation to this, Klovborg I/S was asked about, which weeds are most common in their fields, where they expect to use the robot. Klovborg responded, that the most widespread weeds in their fields are *knotweed* and *poaceae*. Ideally, these are the plants that the robot should be able to recognize as weeds, however, it was not possible to obtain these specific plants during the project, so a plastic plant, resembling these types of weed was chosen as the plant that the robot must be able to recognize and pick up.

2.1.2 Other threats to crops

Weeds are not the only factor that can reduce crop yield. There are also insects and other pests, that can eat the crops, the crops can get a disease or fungus, and in the recent years, effects of climate change has also been stated as a threat to farming and crop yield [11] [12]. Although this project will only be concerned with weed control, it is also relevant to keep these other threats to the crop yield in mind, as the way of controlling weeds, can impact how the other threats affect the crops or the environment in general for that matter. As an example, as also described above, minimizing weeds can also minimize the habitat for diseases and insects that may feed on the crops, meaning that controlling the weeds will have a side effect of also minimizing pests such as disease and insects.

2.2 Weed control

Weeds control is, as the name suggest, different methods to keep weed under control. When it comes to weed control, there are several techniques to use, depending on whether it is a conventional or organic based farm, which will be described in the upcoming sections. However, common for both kinds of farm, the timing is crucial in weed control. There is a critical period, which is a period where the weed significantly reduces crop yield. When the weeds are removed before this period, the loss in crop yield is negligible. The critical period depends on the crop, the weed species, and conditions such as location, weather, time, etc. Generally speaking though, the first plant to emerge out of the ground, wins[7].

2.2.1 Conventional weed control

In conventional farming, the objective is often to maximize the yield of crops, which is achieved through using synthetic chemicals as fertilizer, herb- and pesticides, and genetically modified organisms, that are genetically optimized to withstand external threats to the crops. Conventional farming is good at maximizing the crop yield, but typically at the cost of environmental health. Farmers will plant one type of crop per field, and plant the same crop on the same field year after year, also called 'monocropping', which reduces labor and makes harvesting easier, but at the cost of biodiversity, which can make the crops more vulnerable to e.g. pathogens. Using monocropping also relies heavily upon spraying the fields with herb- and pesticides [13] [14]. Using synthetic chemicals for herbicides, to combat weeds, can be efficient, as it can relatively simple be sprayed over a large field using either tractors or airplanes. Herbicides can also be

designed selectively, so they only attack the weeds, and leave the crops. Some species of crops have also been designed to resist certain herbicides, which further increases the efficiency of using herbicides, but compromises environmental health, as it encourages use of herbicide [15]. Equipment also exist, that can plant the seed, spray with insecticide, herbicide and fertilizer [16]. Even though herb- and pesticides help with maximizing crop yield, it is not good for the environment. The synthetic chemicals can be absorbed into the soil, making it into ground water reservoirs, which are used as drinking water, it can runoff into a stream or the ocean, impacting fish and other animals, or when sprayed, the chemicals can get spread to neighbouring fields by the wind, and thereby exposing other plants to the chemicals, which may be harmful [17]. It should also be mentioned, that using chemicals is not the only way weeds are controlled in conventional agriculture, but in combination with some of the methods described in the next section. Chemical weed control is however defining for conventional farming.

2.2.2 Organic based weed control

In organic farming, the objective is, as in the whole agricultural sector, to harvest as many crops as possible, but with an emphasis on sustainability and environmental health, and avoiding synthetic pesticides and fertilizers, compared to conventional farming. In general, organic farming use fewer pesticides, reduces erosion and nitrate in the ground water. In return, the yield of crops is typically lower and more expensive (around 80% yield of that of conventional farming [18]). Since pesticides are not allowed, other methods needs to be used to combat weeds [19].

Mechanical weed removal

Mechanical removal and prevention of weeds includes manually pulling weeds, using machinery to pull weeds out of the soil, cultivation, tillage in general, burning and grazing. All of these methods are relatively inefficient and time consuming for large scale agriculture. Manually pulling weeds out of the soil requires a lot of labor, and is exhausting for the workers doing it, and expensive in terms of the salary for the workers. Using machinery is the most widely used organic method of weed control, and has been improved through the years, but the tillage tools used for weed removal can still damage the crops. Tillage tools does furthermore not remove the weeds, from the field, it just pulls it out of the ground, meaning there is a risk of the weeds reappearing later, and the tillage tools can help spread seeds from the weeds. The fuel and carbon emissions for the machinery are also a downside of this approach. Burning weeds is also very fuel consuming, and has the potential for damaging crops and the nearby environment [20] [21].

Chemical weed removal

There exist natural and organic herbicides, often consisting of organic acids like vinegar, but these are most often not selective to only prevent weeds, and will also attack the crop itself. The use of organic herbicides thus only plays a minor role in organic farming [21].

Biological weed removal

Animals can be used for weed control, as they can feed on the weeds, and thereby keep the weeds down. However, even though being more environmentally friendly than using synthetic

chemicals, etc. for weed control, in some cases the use of animals for weed control is more destructive and useful, and is not considered to be feasible [20].

Other general methods

Using crops with a solid stand prevents weeds itself, as the crop will be stronger and decreases light and space for the weeds to grow. When planting the seeds for the crops, it is also important that the seeds are clean, to not introduce new plants to the soil. By utilizing crop rotation, the crop of one specific field will change after each harvest, and thereby increasing soil health, breaking disease and pest cycles, and will provide more healthy and strong crops. When a field is bare, it is also an advantage to plant cover crops, which are crops, not necessarily to be harvested, but to protect the soil, when it would otherwise lay bare. This method will improve soil health and smother weeds, among other benefits. Mulching is when a layer of e.g. hay, straw or wood chips are spread out over a field, which can prevent weeds from growing [8] [21].

2.2.3 Weed combatting in Klovborg I/S

In the case of this report a specific farming company namely Klovborg I/S is the user who are going to use this robot manipulator to remove weeds. When asked, Klovborg is stating that their ways of combatting weeds involves: chemical and manual removal using tractors.

2.2.4 Weed control using agricultural robots

In the previous sections, different methods and approaches has been looked at which can combat weeds in different ways, depending on a conventional or organic farming approach. The typical conventional approach with spraying synthetic chemicals all over a field, is not sustainable, as there is a high risk of contamination/pollution. While the organic approach is better for the environment, the weed control techniques used here, along with pest control in general, is challenging, and not necessary as efficient as the conventionally used methods resulting in a loss of crop yield. There is clearly possibility for improvement in both ways of farming, and this project will focus on improving weed control using robot technology.
It is commonly known, in today's farming industry, that different machines such as tractors, harvesters, etc. are already used to ease different tasks for the farmers. Generally in today's field, the farmers will actively apply weed combating to the whole field, either spraying the whole field, or plowing the whole field, but if this large scale approach was reduced to only focus on more local areas of the field, there would be potential for increased efficiency and lower environmental impact. When looking at a more local approach it can be expected that the deployment of pesticides will be reduced to a minimum since it can actually be sprayed on the specific weeds. Also if a more local approach is used for weed combating in the ecological part then this can also seem beneficial, since avoiding excess plowing of the field can have some benefits [22].
By using robots for weed control, or agriculture in general, the workload will also be minimized, and thereby likely also labor costs for the farmer, and the workers on the farm can perform other tasks than the tedious and sometimes strenuous work of removing weeds. By running the robot via electricity, it will also save fuel costs for the farmer, and help reduce carbon emissions, provided that the electricity comes from renewable and sustainable sources. By using robots for weed removal, Klovborg expects greater yield and better utilization of the fields. However, the

robot may have limited capabilities and might be slower than existing methods, but since it runs autonomously, it should be able to run continuously for a long time. Since this is a relatively new and custom made technology, maintenance costs might be high, and it might run into errors and faults often, which are concerns Klovborg also has, but ideally the maintenance and errors for the robot should reduce as the technology gets further developed.

In order to use robots to combat weeds, some different aspects needs to be considered. Firstly it is obvious that the robot has to be able to move around within the field since the location of the weeds is not necessarily the same throughout the season however, some parts of the field could be more affected than other parts. The robot also needs some sort of tool such it can become capable of actually removing the weeds, then the robot should be able to detect objects, meaning it should be capable of actually detecting the weeds. Lastly, but most importantly, the robot needs to be able to distinguish between weeds (unwanted plants) and crops (wanted plants), and perform these tasks autonomously. These different aspects can be formulated shortly in the following list:

1. The robot is able to move around
2. The robot can combat weeds, through some tool
3. The robot can detect weeds
4. The robot can distinguish weeds from crops

These different aspects, which will lay the ground foundation to the weed combating robot, will be looked at individually in the upcoming sections in order to identify any possible strengths or weaknesses, which may be relevant for the development of a weed combatting robot.

2.3 Setup

As stated in the introduction, the goal of this project is to construct a robot which can combat weeds in fields, and in relation to this some outer perimeters of the final solution has already been decided upon. A picture of the robot can be seen in Fig. 2.1, and shows the base platform with continuous tracks, and on top of the base a robot manipulator is placed. These different parameters which has already been decided on, will make the further investigation of the four different topics, stated in the previous section, easier to continue with. Firstly, the base platform and the robotic arms will be analyzed. Then the weed combatting tools, and finally the weed detection and differentiating will be looked at.

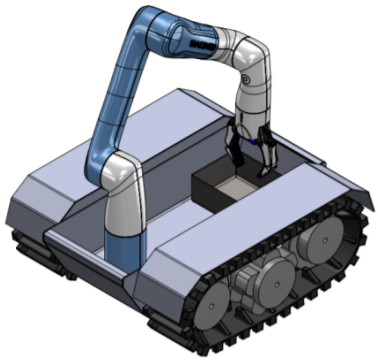


Figure 2.1: Weed combatting robot [23]

2.3.1 Robotic manipulator (Kinova Gen3 Lite)

The robotic manipulator fitted on the base, is the Kinova Gen3 Lite, and this will be the robot's tool for removing/controlling the weeds. It has 6 degrees of freedom (DOF), a maximum reach of 760 mm, and a continuous payload weight of 0.5 kg. The end effector of this manipulator is a two finger gripper, but a specialized tool for picking up weeds must be attached to this gripper. The manipulator can be controlled in both high and low level from a programmatic point of view. This will be elaborated later in this report.

Selection of tools

With regards to the tool used to remove the weeds, there are a few options. The robot can be fitted with a tool that can spray pesticide on the weeds it detects, to kill the weeds, and possibly also other threats, such as insects. This will though require that the robot carries a tank of pesticide, which might will set inconvenient requirements for the size and power of the robot. Although this will decrease use of synthetic pesticide, it will still be used, and thereby the robot will be limited to conventional farms. One could fit it with a water jet instead of spraying chemicals, but this will still require a water tank, and the water jet might not completely remove the weed, and possibly cause unintended destruction of the soil, which again not is good. Fitting the robot with a burning device, that can burn the weeds, will require a lot of fuel, and might pose danger to the environment and nearby crops, and it is likely that the safety approval of such a device would be difficult. Lastly, the robot can be fitted with a tool, that can grab the weed out of the ground, and place it onto a tray on the robot, like a human would do manually. This removes the weeds without the need of any resources, and does not leave the weed on the soil, where it might set roots again, if the conditions are right. This can be performed either by the robot gripping the stem and pulling the weed out, which is the least invasive, but the stem might snap and leave the roots, or the weed might not have a stem at all. Alternatively, the robot can scoop up the weed plant, collecting both plant and roots, but also a lot of excess soil. The robot must then remove the soil from the weed, to not remove soil from the field, and not add unnecessary mass to the tray on the robot. With regards to pulling the weed, the types of knotweed should be strong enough to be pulled, and it is the recommended way of removal. For some of the grasses, the stem might also be strong enough to not break when pulled. It also depends on where in the lifecycle the weed is, it is likely that the plant is weaker when younger. Furthermore, it was found that the

arm can only lift 0.5 kg continuously, meaning that the force needed to pull the weed must not exceed that. Depending on how developed the weed is, the root system might bind a lot of soil, making the plant and roots heavy and difficult to pull out. It might be the better option, to make a tool that digs the plant up, to be sure the stem will not snap, and to collect a more or less known amount of soil, at the expense of potentially cutting some roots and leaving them in the ground.

2.3.2 Intel RealSense LiDAR Camera L515

To capture the images of the different weeds, which will be used for detecting, the Intel RealSense L515 will be used. This camera has both a built in camera and LiDAR, which makes the camera excellent in getting images of an object and knowing how far away the object is. The Ideal range is from 0.25m to 9m, and it can work with a RGB frame resolution of 1920x1080 with a RGB frame rate of 30 fps. The Camera connects with a plain USB-C 3.0 Gen1 cable [24] [25].

2.3.3 Nvidia Jetson Nano Developer Kit

The Nvidia Jetson Nano will be the main computing unit in this setup, where it will collect the data coming from the camera and the Lidar, and control the robot arm. The Jetson is a small compact development board with 4 USB 3.0 ports, 1 USB micro port, 1 HDMI port, 1 Display port, 1 Ethernet port, SD card, also it has a 40 pin header which contains GPIOs, SPI, UART, I2C. Lastly, the whole board is capable of running on a 5 volts DC supply. The Jetson has 472 GFLOPS[1] and it can process many high resolution sensors in parallel, as well as supporting multiple neural networks on each sensor stream. Meaning that even thought the board only is 100x80mm, it is contains lots of processing power [26].

2.4 Objectives and goals

Different analyses has been carried out on In the previous sections and in order to arrive at some requirements, which the final solution can be built from, the different points in the previous sections must be tied together and narrowed down, which is the content of the next upcoming sections.

2.4.1 Moving the arm

When moving the arm, it must be ensured that the arm does not crash into other objects, or itself for that matter. For this, certain protection zones can be defined, that defines spaces in the workspace of the manipulator, it must not move to or within. These protection zones must include the base. Furthermore, a rather large camera will be mounted on the end effector, and the robot must also be configured to not crash this camera into both the base or itself.
Controlling the arm can be done using the Kortex API. It is an API for either C++ or Python, where the C++ version allows for high- and low level control, whereas the Python version only allows high level control. High level control is limited to only sending either position and velocities to the joints of the robot, where low level control gives access to all controllable parameters,

[1] floating point operations per second

but is also correspondingly more difficult. When operating in high level control mode, the manipulator can be thought of as a closed loop system with an in-built controller, meaning e.g. a desired position of the manipulator is the reference set point for the controller, and the controller in the manipulator will handle moving the arm. This project will mainly be concerned with high level control of the manipulator. It is although still necessary to develop a model of the arm, not for controller design, but as an estimator, that can, given the same input as the manipulator, estimate the movement/position/state of the manipulator.

2.4.2 Visual servoing

Visual servoing is a term used in robotics, where a robotic system is controlled based on visual input from a camera. In modern visual servoing systems, the camera will take a picture of what it sees, and some feature extraction algorithm, determines what in the image is of interest. This data is then used for controlling the robot, by generating input signals to the robot such that it will eg. track a point of interest in the image. This is done continuously in a closed loop, such that the robot will keep tracking the object of interest in the image, and keep updating features in the image [27]. In this setup the robotic manipulator (Kinova Gen3 Lite) is to be controlled. On the end effector, there is a camera mounted (Intel RealSense L515), which is to be the main input for the system. This type of setup is called an *eye in hand* configuration, because the camera is rigidly mounted on the robot's end effector (hand). The central problem in visual servoing, can be described by minimizing the error $\boldsymbol{\epsilon}$ between e.g. the current pose of the robot $\boldsymbol{r}$ end effector, and a desired pose of the end effector $\boldsymbol{r}^*$ [28]:

$$\hat{\boldsymbol{r}} = \arg\min_{r} \boldsymbol{\epsilon}\left(\boldsymbol{r}^* - \boldsymbol{r}\right) \tag{2.1}$$

Where $\hat{\boldsymbol{r}}$ is the pose of the manipulator's end effector after reaching the optimized pose. The error can be defined as the difference of a set of features $\boldsymbol{s}(\boldsymbol{r})$, when the end effector has the pose $\boldsymbol{r}$, and a set of features $\boldsymbol{s}^*$, when the end effector is at the pose $\boldsymbol{r}^*$. The features can e.g. be image coordinates or vectors in 3D, that describe the end effector pose, and the desired pose

$$\boldsymbol{\epsilon}(\boldsymbol{r}) = \boldsymbol{s}(\boldsymbol{r}) - \boldsymbol{s}^* \tag{2.2}$$

In other words, the objective of visual servoing is to move the manipulator from a given pose $\boldsymbol{r}$, to a desired pose $\boldsymbol{r}^*$, by the help of the error function, that tells the algorithm in which direction the manipulator must move, based on the input from the camera. In this project, $\boldsymbol{s}^*$ will be assumed to be stationary. Visual servoing is a very comprehensive and big topic which will not be covered extensively in this report, but a more thorough description and analysis of the topic can be found in [27] and [28].

2.4.3 Weeds detection

The data coming from the camera and the LIDAR will be used together with some sort of computer vision, with the goal of training the system such it can recognize and locate weeds. To train the algorithm to recognize these specific weeds, as described before, it will need training data, in the form of image data of the weeds, ideally as many images from many angles as possible of as many different plants, are needed to build a good model for the computer vision algorithm. It might not be necessary for the robot to differentiate between different types of weed, if it only

has one way of removing the plants anyways. It should rather focus on classifying, whether it is a weed or not, and then locating the plant in space, for the manipulator to remove it. It might also be necessary to train the algorithm to recognize the crops, so it does not accidentally remove those plants too.
For detecting and locating objects in an image, there are several different algorithms to use. In this project, the focus will be on the YOLO[2] version 5 (YOLOv5) algorithm. The YOLO family is well known for its speed and accuracy [29] [30], where YOLOv5 is the newest addition to the family. It is furthermore suitable for this project, due to its easy implementation and ability to re-train the algorithm in Python.

2.4.4 Goals of this project

The goal of this project is to create a robot which is capable of combatting weeds in the fields of Klovborg I/S. The analysis of this problem has previously resulted in the following four sub goals: "The robot is able to move around", "The robot can combat weeds, through some tool", "The robot can detect weeds" and "The robot can distinguish weeds from crops". These four points are highly relevant, but are still very broad meaning it is needed to narrow them down. The first point relates to the mobility of the robot, and it has already been described that the base platform has been decided upon, meaning this point is not within the scope of this project. The only focus regarding mobility of the robot will be on the manipulator, and how this is controlled in order to move around.
The second point relates to the usage of a tool in the robot manipulator. Earlier, different potential tools has been looked at, where it has been concluded that the tool should either be a small shovel or mechanism to grab the weed. The shape and length of this tool has to be known since this will set some demands for the control of the robot manipulator, since only the end of the tool should reach the weeds.
The third and fourth point are similar in the fashion that they both evolve around detection and recognition. It has previously been stated that a camera and a LIDAR will be used together to get data about the specific object. These two attachments will give some data which, by applying some signal processing, is used to detect both the object, and the distance to it.
To sum up, this project has two overall problems or objectives. One is to move the arm in a controlled manner, without it damaging itself or the crops on the field, and make it move such that it can collect the weeds. Secondly, the arm must, with the help of the camera and YOLOv5, detect, recognize and remove weeds growing from the ground. For this it must utilize a neural network (YOLOv5), that has been trained to classify objects in the camera image as weeds, and locate them in a way such that the manipulator arm can pick up the weed. When the arm is controlled based on visual input, the technique used is called visual servoing, as described earlier.

2.4.5 Delimitation

To make the problems related to this project solvable within the time frame of the project, a few limitations have been set, to define what the focus areas are, and which areas will not be considered:

[2]You Only Look Once

- It will be assumed that the weeds, which the robot manipulator has to remove, is always within the work space of the robot, meaning that the base of the robot manipulator will be stationary.
- It will be assumed that the field will consist mostly of dirt with only weeds growing periodically throughout the field, meaning that the whole field is not covered in weeds, and that it should be possible to make out individual weed plants.
- Since different tools can be used, in order to remove weeds, as discovered earlier, a decision has to be made in relation to what tool will be used during this project. Based on the time frame and extent of this project is has been decided that the standard gripper, which is already mounted on the robot manipulator, will be used.
  - Furthermore, the main focus of the project is identifying weeds and moving the gripper to a position where a tool can remove the weed, and actually removing the weed is secondary.
- Due to limitations with keeping real plants alive during this project a plastic plant, has been chosen to test and train the system on. This plastic plant is deemed to have similar characteristics to the actual weeds which Klovborg I/S is trying to combat.
- The velocity at which the robotic manipulator moves will not be a focus point in this project. However, it will be kept in mind in order to not make the manipulator unnecessary slow.

2.4.6 Requirements

1. The robot manipulator will be capable of moving in any direction needed.
   (a) Controlling the robot manipulator (Kinova Gen3 lite) is will be through the API using Python.
2. The system as a whole is based on a Nvidia Jetson Nano, and the camera will be an Intel RealSense L515.
3. The system will detect the plant with a transfer learned YOLOv5 network.
4. The system will detect the plant with a confidence level of ≥ 0.85, when the plant is within 1 meter from the camera.
5. The system will detect the plant, that is within 2 meters from the camera, with a confidence level of ≥ 0.50.
6. When the plant is within 2 meters from the base of the robotic manipulator, the system will be able to detect: the tilted plant, the plant viewed from the side and the plant seen from the top.
7. The system will not detect other items as weed (avoiding false positives).
8. The manipulator is capable of moving towards the plant even though no images or bounding boxes are received, after a weed has been detected at least once.

9. The system will be able to detect the plant, move the gripper onto the plant, and pick the plant up.

Chapter 3

System overview

3.1 System block diagram

A block diagram of the complete system, which is to be designed and implemented in this project, can be seen in Fig. 3.1. This complete system shows that, as stated earlier, the main computing unit for this system will be the Nvidia Jetson Nano, which will get data from the Intel camera and then control the robot manipulator accordingly, in order to reach a desired position and orientation of the end effector. To have this system working as intended the different blocks shown in the figure has to be looked into which will happen in the upcoming sections. The physical system of the manipulator and plant can be seen in Fig. 3.2. In the right of Fig. 3.1, the plant position relative to the end effector is represented. The camera will take a picture of the plant in the current pose of the end effector, and the image is sent to the neural network, that will detect the plant on the picture. The neural network outputs a pixel coordinate, representing the plant position, which is used with an inverse projection method, that reconstructs the 3D position of the plant, relative to the gripper, denoted as the error. The error will be used to update the estimator, and if no plant is detected, the estimator will use the previously measured error to estimate the error when detection loss occurs. The error is then passed to the controller that will generate control signals to the manipulator, which will then move the end effector, and change the relative position of the plant to the end effector. As indicated with the arrows, this happens continuously in a loop.

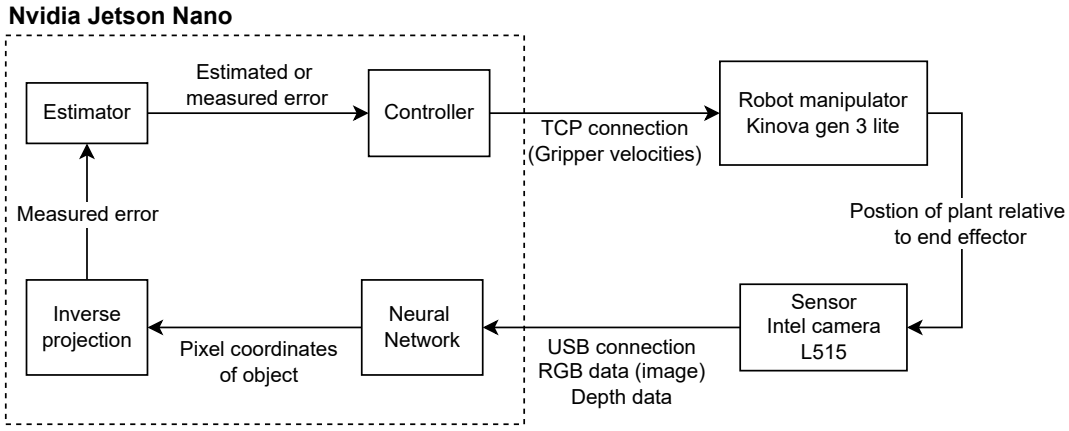


Figure 3.1: Block diagram of the system on an overall level

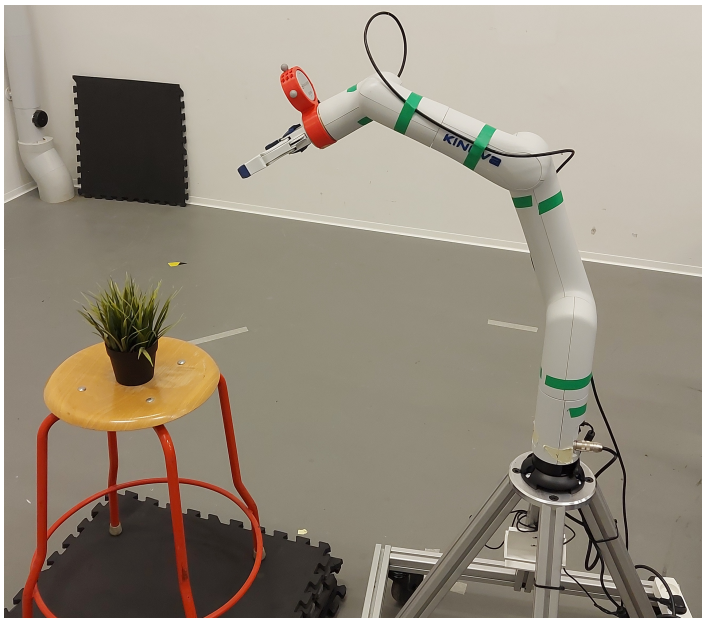


Figure 3.2: Setup of the manipulator

3.2 Code structure

The system in this project is going to include several different parts, where the three main parts are getting data from the camera, performing object detection and moving the arm accordingly. For this reason, it has been decided to make a modular structure of the code, to keep things separated and organized. On Fig. 3.3, there is a diagram that helps in visualizing how the structure of the code is going to be. In the middle is `main.py`, which is the main file that will contain the code that connects all the different parts of the system. It is shown in the figure, that `utilities.py`, `KinovaArm.py`, `Detection.py`, `Projection,py`, `Control.py` and `IntelCamera.py` are imported into the `main.py` file. Each file contains a class with the relevant functions, for each respective part of the system.

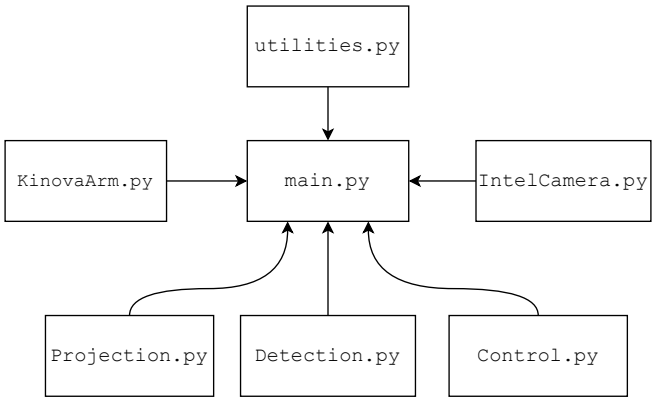


Figure 3.3: Code structure diagram

Chapter 4

Robot manipulator

Controlling a robot manipulator is usually done with the purpose of the manipulator reaching a certain object located within its vicinity. The upcoming section will give a brief introduction into how the robot manipulator can be controlled such that it can reach the object.

4.1 Controlling of the manipulator

4.1.1 Manipulator

On Fig. 4.1 the Kinova robotic manipulator is shown which will be used in this project. On the figure it can be seen that the manipulator is made up of six actuators and seven rigid bodies[1] connected between the actuators. The actuators are revolute joints meaning they can rotate. Since there are six actuators, and each joint yields one degree of freedom, then it can be determined that the robotic manipulator has six degrees of freedom. At the end of the manipulator the *end effector* is located which interacts which the real world, and this end effector is also called the 'gripper'. It is important to notice the different actuators, in the figure, and the fact that they can rotate, because this will be important in the following sections.

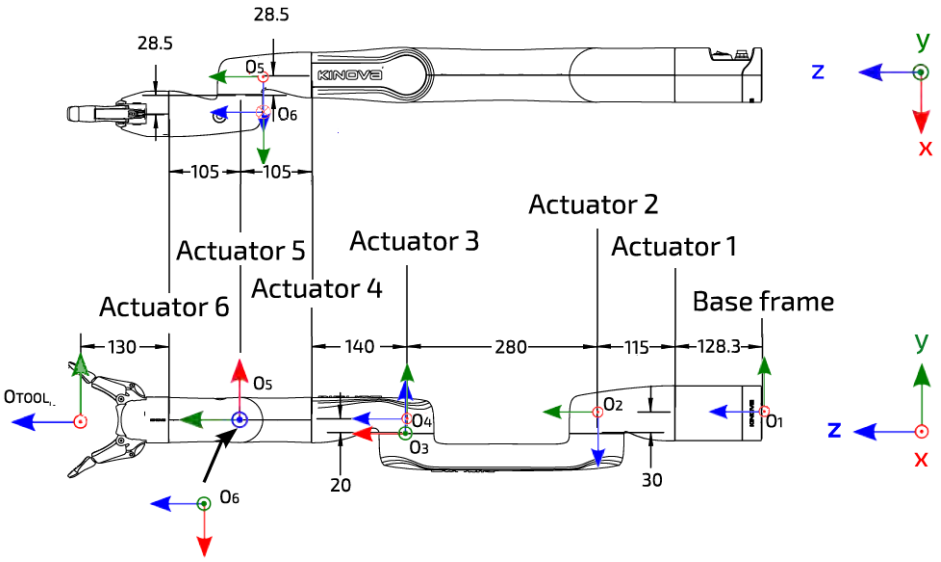


Figure 4.1: Frames for the Kinova robotic manipulator [32]

[1] Is a body which does not deform, meaning any distance between two points on the rigid body is always the same [31]

4.1.2 Forward and inverse kinematics

There are two concepts, related to movement, within the area of revolute robotic manipulators. The first is forward kinematics, and the other is inverse kinematics. Forward kinematics is used to calculate the position of the end effector, as a function of all the joint angles. The inverse kinematic will do the opposite, meaning it will, based on a desired position of the end effector in space, give us the different angles which the joints in the manipulator needs to be rotated with in order to reach the desired point. This relationship can be seen in Fig. 4.2. The notation for the joints angles can also be seen in Fig. 4.2 as $q_1, q_2, q_3 ... q_6$, since there are six actuators, where the lower case number relates the the specific actuator shown in Fig. 4.1. Furthermore, it is worth noticing in Fig. 4.2 that in order to use the kinematics, both inverse and forward, some 'link parameters' are also needed which relates to physical properties of the manipulator e.g. length of links, angles between links etc.

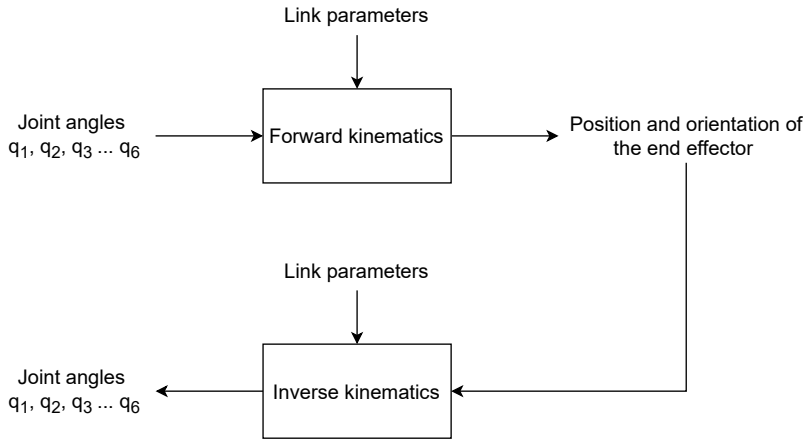


Figure 4.2: Forward and inverse kinematics

4.1.3 Reference frame

It has been established that the robot manipulator is made up of six rigid bodies and furthermore each of these bodies can rotate depending on the actuators connected to them. Since the goal is to find out the position and orientation of the end effector, of the manipulator, a relationship between the different rigid bodies has to be established. Before this relationship can be established a concept concerning the different *frames* of the manipulator has to be introduced. A frame is the span of a three dimensional orthogonal space denoted x,y,z, such a frame is connected to each rigid body of the robotic manipulator, and this frame reveals the orientation of each rigid body. The frames for the Kinova robot manipulator can be seen in Fig. 4.1. In the figure it can be seen that the manipulator is stretched out, meaning most of the frames are aligned in some of axes. However, if one imagines the manipulator in another position, these different frames will not be aligned. It can be seen that at the base of the manipulator the 'base frame' is shown it is assumed that this base frame is the origin of the space $(0, 0, 0)$, meaning the goal to find out which coordinate the O_{tool} frame is located at $(x_{tool}, y_{tool}, z_{tool})$ with respect to the base frame.
From Fig. 4.1 it can also be concluded that the robotic manipulator consist of seven rigid bodies, and therefore also seven frames $O_1 - O_6$ and O_{tool}.

4.1.4 Rotation matrices

The commonly known *elementary rotation matrices*, shown in eq. (4.1), eq. (4.2) and eq. (4.3) [33, p. 23-24], is used to rotate a vector with a given angle about a specific axis x,y or z. These elementary rotation matrices can also, on a more general basis, be interpreted as a transformation matrix, which is simply a matrix that transforms a vector into another vector, usually with the goal of changing some properties of the vector being either scaling or rotating.
Apart from rotating a vector inside the same coordinate frame, these elementary rotation matrices which has been introduced are also important in another sense since they can be used to described the orientation between two frames [33, p. 26]. However, since the elementary rotation matrices only rotates with respect to one axis each, then a composite of each elementary rotation matrix is needed in order to ensure that all three axis can be rotated about.

Elementary rotation matrices

$$\mathbf{R_z}(\alpha) = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.1}$$

$$\mathbf{R_y}(\beta) = \begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) \\ 0 & 1 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) \end{bmatrix} \tag{4.2}$$

$$\mathbf{R_x}(\gamma) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\gamma) & -\sin(\gamma) \\ 0 & \sin(\gamma) & \cos(\gamma) \end{bmatrix} \tag{4.3}$$

4.1.5 Composite rotation matrix

The elementary rotation matrices has been introduced above, and multiple of these rotation matrices can be combined into a *composite rotation matrix*. The elementary rotation matrices can only rotate the object about a single axis, where the composite rotation matrix can rotate the object about multiple axis. It is worth noticing that since the transformation is based on matrices, then the order of multiplication is important. This also makes sense since rotating about an axis, with a specific amount, and then rotating about a second axis, with a specific amount, will not necessarily result in the same vector, if the rotation order is switched. In eq. (4.4) a small example can be seen where $\mathbf{v_{xyz}}$ is the vector which is rotated about the x,y and z axis, $\mathbf{v_x}$ is the initial vector and $\mathbf{R_{xyz}}$ is the composite rotation matrix.

$$\mathbf{v_{xyz}} = \Big(\mathbf{R_z}\left(\mathbf{R_y}\left(\mathbf{R_x}\mathbf{v_x}\right)\right)\Big) = \left(\mathbf{R_z}\mathbf{R_y}\mathbf{R_x}\right)\mathbf{v_x} = \mathbf{R_{xyz}}\mathbf{v_x} \tag{4.4}$$

4.1.6 Homogeneous transformation

It has been stated that the rotation matrices can be used to describe the orientation between two frames, then looking at Fig. 4.1, the point O_1 is assumed (origin of the base frame), and then another point is assumed O_{tool} which is the origin of the frame of the tool. Then a point of

interest, called P, is also assumed, which is a point that wants to be known with respect to the base frame, which equals the vector $\mathbf{p^0}$. However, the point P is defined based on the tool's frame, meaning some sort of rotation needs to be carried out, since the tool frame, and the base frame, is not necessarily aligned. Furthermore, there is also an offset between the origin of the base frame and the origin of the tool frame which means that apart from the rotation, to align the two frames, a translation is also needed to account for this offset [33, p. 37]. These different relationships are visualized in Fig. 4.3.

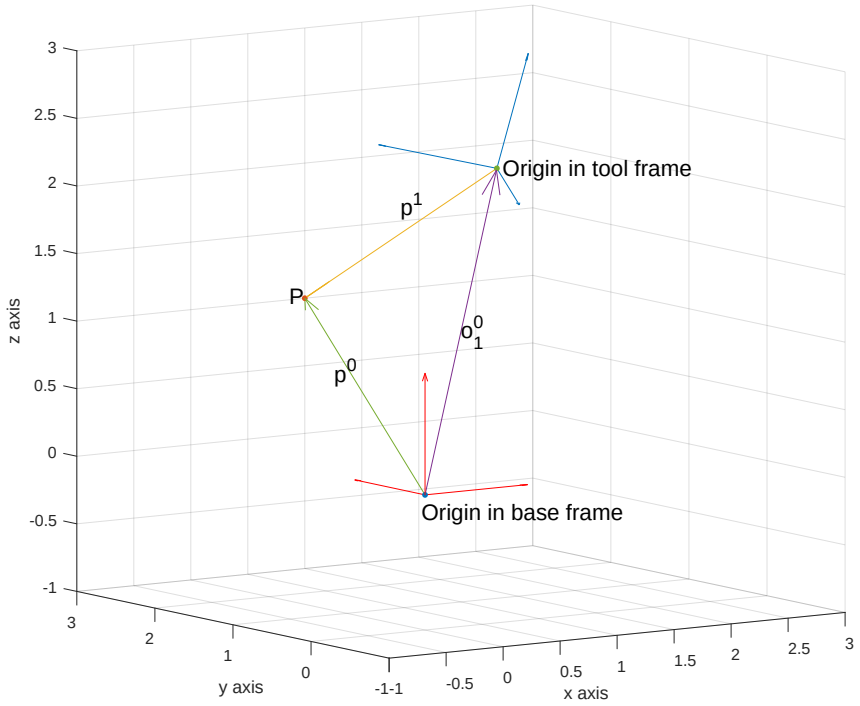


Figure 4.3: 3D plot which shows a point in space 'P'

In order to relate the wanted point P to the base frame, and hereby finding $\mathbf{p^0}$, a translation and rotation is needed, and this can be described by eq. (4.5), where $\mathbf{R_1^0}$ is the composite rotation matrix, $\mathbf{o_1^0}$ is the translation vector, $\mathbf{p^1}$ is the known description of the point P wrt. the tool frame and $\mathbf{p^0}$ is the position vector of the point P wrt. the base frame.

$$\mathbf{p^0} = \mathbf{o_1^0} + \mathbf{R_1^0 p^1} \tag{4.5}$$

To write eq. (4.5) in a more compact a 4x4 matrix is introduced, which includes both the rotation matrix and the translation vector, and this new matrix which is known as *Homogeneous transformation matrix* is shown in eq. (4.6). This equation yields the same answer as the equation shown in eq. (4.5), but now the resulting vector has four rows with the last row just being a one [33, p. 37-38]. Just to clarify, it is important to notice that $\mathbf{R_1^0}$ is a 3x3 matrix, $\mathbf{p^1}$ is a 3x1 vector and $\mathbf{p^0}$ is also a 3x1 vector.

$$\begin{bmatrix} \mathbf{p^0} \\ 1 \end{bmatrix} = \begin{bmatrix} & \mathbf{R_1^0} & & \mathbf{o_1^0} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{p^1} \\ 1 \end{bmatrix} \tag{4.6}$$

In eq. (4.7) the different notations related to the homogeneous matrix are shown. The important thing to notice is the fact that $\tilde{\mathbf{p}}^{\mathbf{0}}$ is the vector wrt. frame 0, $\tilde{\mathbf{p}}^{\mathbf{1}}$ is the same vector wrt. frame 1, and $\mathbf{A_0^1}$ is the homogeneous transformation matrix, which transforms the vector from frame 0 (subscript) to 1 (superscript). Additionally it can be seen that if a homogeneous transformation matrix is known, which transforms from frame 0 to 1, then a transformation matrix which does the opposite (transforms from frame 1 to 0) can be found by taking the inverse of original matrix [33, p. 38].

$$\tilde{\mathbf{p}} = \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} \qquad \mathbf{A_0^1} = \begin{bmatrix} & \mathbf{R_1^0} & & \mathbf{o_1^0} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.7}$$

$$\tilde{\mathbf{p}}^{\mathbf{1}} = \mathbf{A_0^1}\tilde{\mathbf{p}}^{\mathbf{0}} \Leftrightarrow \tilde{\mathbf{p}}^{\mathbf{0}} = \mathbf{A_1^0}\tilde{\mathbf{p}}^{\mathbf{1}}$$
$$(\mathbf{A_0^1})^{-1} = \mathbf{A_1^0}$$

In the previous sections the concept of rotation matrices, and transformation matrices has been introduced, which shows how a vector which is known wrt. a specific frame can be related to other frames, in the manipulator. In the examples discussed so far there has only, for simplicity, been shown two frames and their relation to each other, but the different concepts can of course be extended to included the amount of frames necessary. In relation to the robot manipulator it has already been stated that there are seven frames which needs to be accounted for, meaning that six homogeneous transformation matrices has to be found.

4.1.7 Denavit-Hartenberg Parameters

In the previous section it has been described that the six transformation matrices, which describes the transformation between the different frames of the robotic manipulator, has to be found. In order to do so the *Denavit-hartenberg convention* will be used which is a method that relates the physical dimensions of the manipulator to the transformation matrices. The deeper understanding of how the Denavit-Hartenberg convention is derived is not of interest in this report and is therefore omitted. The convention states that based on three constants α, a and d, which are related to the physical dimensions of the manipulator, and the rotation angle of each joint θ the transformation matrices for the robotic manipulator can be created as shown in eq. (4.8) [33, eqn. 2.47]. Furthermore, the four before mentioned parameters which describes the Kinova gen3 lite manipulator are shown in table 4.1. If these parameters are added to the standard Danevit Hartenberg equation shown in eq. (4.8), then the different transformation matrices can be created for the Kinova robotic manipulator and these are shown in appendix A.

$$\mathbf{A_i^{i-1}}(\theta_i) = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i)\cos(\alpha_i) & \sin(\theta_i)\sin(\alpha_i) & a_i\cos(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i)\cos(\alpha_i) & -\cos(\theta_i)\sin(\alpha_i) & a_i\sin(\theta_i) \\ 0 & \sin(\alpha_i) & \cos(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.8}$$

Table 4.1: DH parameters for the Kinova robotic manipulator [32]

i	$\boldsymbol{\alpha}$	**a**	**d**	$\boldsymbol{\theta}$
1	$\frac{\pi}{2}$	0.0	128.3+115.0	q_1
2	π	280.0	30.0	$q_2 + \frac{\pi}{2}$
3	$\frac{\pi}{2}$	0.0	20.0	$q_3 + \frac{\pi}{2}$
4	$\frac{\pi}{2}$	0.0	(140.0+105.0)	$q_4 + \frac{\pi}{2}$
5	$\frac{\pi}{2}$	0.0	(28.5+28.5)	$q_5 + \pi$
6	0	0.0	(105.0+130.0)	$q_6 + \frac{\pi}{2}$

Conclusion

The different transformation matrices for the Kinova robotic manipulator has been found, and can be seen in appendix A. These transformation matrices are also functions of the different joint angles. The transformation matrices relates the frames 0-6 together (7 frames in total), where the base frame, shown in Fig. 4.1, is the 0th frame, and the tool frame is the 7th frame. If one wants to use the Denavit Hartenberg transformation matrices to determine velocities of the gripper, one must be attentive to the fact, that the reference frame for the gripper of the DH matrices is not the same as the reference frame used to control velocities of the end effector [32]. The A_g^6 matrix in appendix A corrects for this, such that the reference frame of the gripper from the DH matrices, is the same for the one the velocity controller of the gripper uses.

4.2 Kinova Kortex API

For controlling the manipulator via code, the Kinova Kortex API can be used. The API supports both low-level control, where e.g. the voltages for the joint actuators can be individually controlled, and raw data from the sensors can be read, and high level, that treats the manipulator as a closed loop system, where a position or speed is given to the manipulator, and it will control its actuators, such that it meets this position and speed. The API can be used with both C++ (low- and high-level control) and Python (only high level control). For this project, the focus will be on the high level control as it matches the scope of the project, and that Python is easier to interface with the algorithms for detecting and locating weeds via. camera. Python generally runs slower than C++ but it is assessed to be sufficient for this project. The API and dependencies for Python for the Kinova Gen3 Lite manipulator can be downloaded from Kinovas website [34] as a Python Wheel package (`.whl` file extension) and can be installed via `pip3 install <wheel-file-name>` in a command prompt, in the directory where the `.whl` file is located. This section will not contain a detailed description of the code used for controlling the manipulator, as the implementation of the Kortex API is quite similar to the examples provided by Kinova [35].

However, a object oriented approach has been taken to implement the functionalities of the API. All the code for the arm is written in a file called `KinovaArm.py`. This file will contain a class, `Arm`, which when instantiated will connect to the arm, and handle all communication with the arm. There will be a main file, called `main.py` for the entire system, that ties all the different modules together, and contains the final implementation. The code implementation is based on the Kinova Gen3 Lite User Guide [32], the Kortex GitHub documentation and code examples [35] and Kinovas How-To video series about the Kortex API [36]. The implementation of the API can be seen and downloaded from the project GitHub page [37]. A small summary of the important functions will be stated here:
`KinovaArm.py` with class `Arm` - Handles communication with the manipulator

- `Arm` - makes a `Arm` object. Takes a router object as argument.
- `home()` - moves the arm to the predefined home position
- `get_data()` - returns a `Feedback` object that contains different data about the arm
- `gripper()` - opens and closes the gripper to the specified percentage in the argument. 0 is open, 1 is closed.
- `send_speeds()` - sends velocities to the end effector. Takes linear and angular velocities and duration of the command as arguments.

`utilities.py` - Contains utility functions for the manipulator (from the Kortex API)

- `createTcpConnection` - creates a router object and takes `KinovaArm.get_args()` as arguments

Chapter 5

Object localization

One of the central objectives in this project is detecting and locating the plant, relative to the gripper, so the arm knows where to move in order to get near the plant. This chapter will cover how the Intel L515 camera can be interfaced in Python and how image and depth data can be obtained from it. Following this, it will be explained how to use the data from the camera, to estimate a position/direction of the point of interest. The last part is of significant interest to the object detection chapter, following this chapter, where the techniques in this chapter can be used for recognizing and locating an object in space and automatic drawing of bounding boxes for training of the neural network.

5.1 Camera

The main input to the system, is input from the camera. As previously mentioned, the camera used is the Intel RealSense L515 LiDAR camera. It can both provide an RGB color image stream, along with a depth image from the LiDAR. This allows both object recognition from the color image, along with a distance measurement to the given object. The camera is mounted on the gripper and is fixed with regards to the gripper reference frame, making it an eye-in-hand configuration, and will therefore also be considered as a part of the arm configuration. To get data from the camera, there is a Python library `pyrealsense2` that can be used. This library can retrieve both depth and color images, along with other parameters of the camera. This section will show, how the camera can be interfaced in Python, and how to retrieve the depth and color images from the camera. The section will be based of the `pyrealsense2` documentation [38] and example code from the Github repository [39]. Note: to install the library on ARM based systems, as the Jetson Nano, it is needed to build the library from source. Instructions are supplied on the Github page [39]. The code implementation in the project is made modular, to have better organized code and documentation. This means that everything, that has to do with the camera, is organized in a seperate file called `IntelCamera.py`. When imported via `import IntelCamera`, it will load the necessary libraries. The file furthermore contains a class, `Camera`, that when instantiated will connect to the camera, and handle all the communication. A description of this file and class, will follow in the next subsections.

5.1.1 Setup and initialization

The code for setting up and configuring the camera, can be seen in listing 5.1. On the first three lines, the necessary libraries are imported. Then the `Camera` class and the constructor function `__init__` is defined. When making a `Camera` object, the arguments it takes are the width and height (resolution) in pixels of the image it needs to produce. These arguments are stored in the object itself. On lines 10 and 11, the `pipeline` and `config` objects are instantiated, which is used for data streaming and configuration of the camera. The actual data stream of color and depth

images are enabled on lines 13-14 with the desired resolution and framerate. On line 16, the actual stream is initiated, with the configurations specified on the previous lines. The next four lines are only for retrieving the camera intrinsics, that is to be used for estimating a direction of a point in space. `fx` and `fy` are the focal lengths of the camera in pixels, and `u0` and `v0` are the coordinates for the principal point of the camera. More on that in the next section. Lines 22-24 are configuring the depth sensor to measure as small distances as it can, and getting the conversion factor from depth units to meters. Whenever depth data is retrieved, it is in some arbitrary unit, that needs to be multiplied with `depth_scale` to get it in meters. Finally, on lines 26-27, the align object is defined, that will make sure that the depth image, is aligned with the color image.

Listing 5.1: Setup and initialization of the class

```
1  import pyrealsense2 as rs
2  import numpy as np
3  import cv2
4
5  class Camera:
6      def __init__(self, width, height):
7          self.width = width
8          self.height = height
9
10         self.pipeline = rs.pipeline()
11         self.config = rs.config()
12
13         self.config.enable_stream(rs.stream.depth, self.width, self.height, rs.
               format.z16, 30)
14         self.config.enable_stream(rs.stream.color, self.width, self.height, rs.
               format.bgr8, 30)
15
16         self.profile = self.pipeline.start(self.config)
17         self.rgb_profile = self.profile.get_stream(rs.stream.color)
18         self.intr = self.rgb_profile.as_video_stream_profile().get_intrinsics()
19         self.fx, self.fy = self.intr.fx, self.intr.fy
20         self.u0, self.v0 = self.intr.ppx, self.intr.ppy
21
22         self.depth_sensor = self.profile.get_device().first_depth_sensor()
23         self.depth_sensor.set_option( rs.option.min_distance, 0)
24         self.depth_scale = self.depth_sensor.get_depth_scale()
25
26         self.align_to = rs.stream.color
27         self.align = rs.align(self.align_to)
```

5.1.2 Retrieval of data

The code on listing 5.2 describes a few of the member functions of the `Camera` class, all related to retrieval of data from the camera. The first function is `get_frames` that does not take any arguments. On line 2, it retrieves the data from the camera, and on the following three lines, it aligns the depth frame to the color frame in `aligned_depth_frame` and `color_frame` respec-

tively. Then it checks if there is any content other than zero in the depth and color frame, and if there is it will mean that it has not retrieved the data sucessfully, and will return a zero. Otherwise, it will convert `color_frame` to an image as an RGB matrix. Finally the method returns both the depth frame and the color image. The color is ready to be shown with `cv2` but the depth frame just contains raw depth data. If this is passed to the `get_depth_color`, also in listing 5.2, it will return a color image, mapped to the jet color scheme, where blue is near and red is far from the camera. To get the distance to a specific point, described by a pixel coordinate on the image, `get_dist` can be called, and it will return the distance in meters, to a pixel coordinate where `x` is the x position of the pixel coordinate and `y` is the y position of the pixel coordinate.

Listing 5.2: Methods of the `Camera` class. This example is an extension of listing 5.1

```
def get_frames(self):
    frames = self.pipeline.wait_for_frames()
    aligned_frames = self.align.process(frames)
    aligned_depth_frame = aligned_frames.get_depth_frame()
    color_frame = aligned_frames.get_color_frame()

    if not aligned_depth_frame or not color_frame:
        return 0

    color_image = np.asanyarray(color_frame.get_data())

    return aligned_depth_frame, color_image

def get_depth_color(self, depth_frame):
    depth_image = np.asanyarray(depth_frame.get_data())
    return cv2.applyColorMap(cv2.convertScaleAbs(depth_image, alpha=0.03), cv2
        .COLORMAP_JET)

def get_dist(self, depth_frame, x, y):
    return depth_frame.get_distance(x,y)
```

On Fig. 5.1, a snapshot of the color and depth (colormap) output of the camera can be seen. There are placed some green circles at selected distinct points/objects in both of the images, and a red circle at the center of the image. It can be seen that the further away from the center the points/objects are, the larger the deviation between the color and depth map. The two red circles are placed approximately the same place in the color and depth image, relative to the object in the image, but with the top left green circles, are placed noticeably far from each other. Although it is a noticeable difference, it is rather small, and furthermore, there is not much to do about it. The images are as well aligned as they can be from the camera, and it is probably not worthwhile to try and improve such a small deviation.

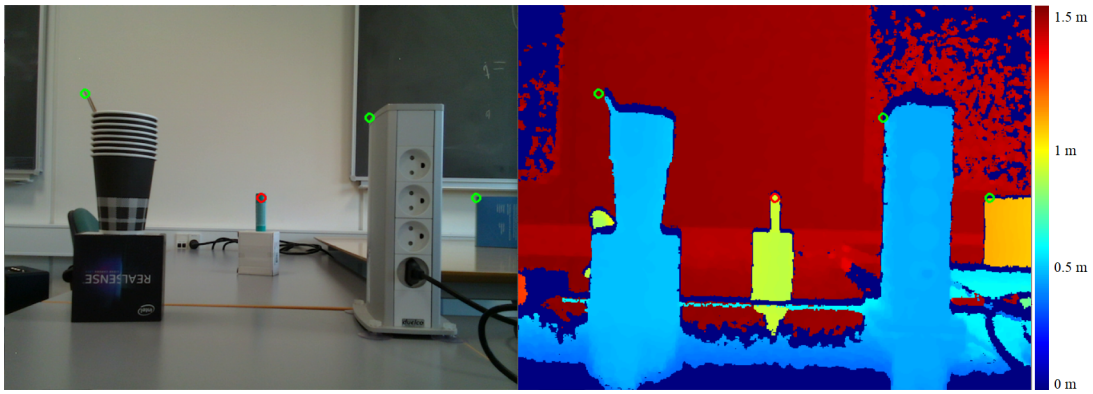


Figure 5.1: RGB image (left) and depth colormap (right). Images are set to the same resolution

5.2 3D coordinate from image

In visual servoing applications like this project, it is not always enough to just detect an object and its location in an image, but it must also relate this to a location and/or direction in space. This section will cover, how a 3D point in world units (eg. meters) can be mapped to a 2D point on the image plane, as a pixel coordinate, and how a 2D point can be related to a point or direction in 3D. This section is based on the knowledge in chapter 11.3 in [40].

5.2.1 Projective geometry and camera model

For many cameras with a thin lens, the pinhole camera model can be used as a good model for a camera. The pinhole camera model assumes that all rays of light from the source, will pass through one small hole in the camera, and be projected on the image plane. The pinhole camera model performs a central projection of a point in space, represented by the vector $\boldsymbol{\epsilon_c}$, as illustrated on Fig. 5.2. The c-subscript denotes that the point/vector is with respect to the camera basis vectors and origin, also shown on the figure. In this representation, O_c is the optical center of the pinhole model and origin for the camera reference frame/basis vectors, f is the camera focal length, (u_c, v_c) is the *principal point* of the camera, i.e. the center of the image plane. O_i is the image plane origin, and has the two basis vectors $\boldsymbol{u_i}$ and $\boldsymbol{v_i}$ originating from that point, also shown on the figure. All image coordinates/pixel coordinates, is with respect to these vectors. The z-axis of the camera reference frame, is the line that goes through the optical center and the principal point of the camera, and the image plane is parallel to the xy plane of the camera reference frame.
The pinhole camera model possesses a property, that allows for a linear transformation to map a point in 3D space, to a point on the 2D image plane. To derive the transformation, it is firstly needed to assume a point (vector) in 3D space, which will be denoted as $\boldsymbol{\epsilon_c}$, where X_c, Y_c and Z_c are the 3D coordinates in Cartesian space, with respect to the optical center of the camera, and the camera reference frame/basis vectors, x_c, y_c and z_c.

$$\boldsymbol{\epsilon_c} = \begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} \tag{5.1}$$

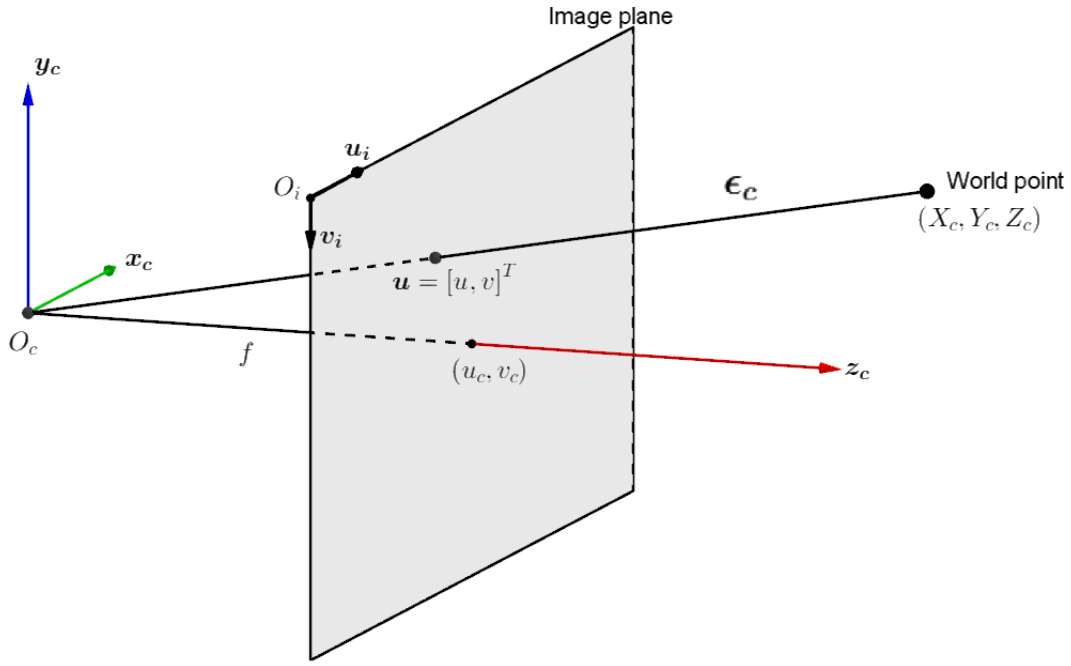


Figure 5.2: Central projection of a pinhole camera model

The objective is to obtain the $\boldsymbol{u}$ vector, which will represent the projection of the point, that is represented by $\boldsymbol{\epsilon_c}$ on the image plane in pixels, where u and v are the Cartesian pixel coordinates, as seen from the image plane origin, O_i and its assiociated basis vectors, u_i and v_i:

$$\boldsymbol{u} = \begin{bmatrix} u \\ v \end{bmatrix} \tag{5.2}$$

If Fig. 5.2, is projected onto the zy plane, one can see that the projection consists of a right triangle between the optical center, the principal point, and the image plane projection of the 3D coordinate $\boldsymbol{\epsilon_c}$ (the projection of the point represented by $\boldsymbol{\epsilon_c}$ on the zy plane is equal to the Z_c and Y_c components of the point), where the direction/angle of the hypotenuse is determined by the 3D coordinate (the Z_c and Y_c components of $\boldsymbol{\epsilon_c}$). The projection of Fig. 5.2 on the zy-plane can be seen on Fig. 5.3.

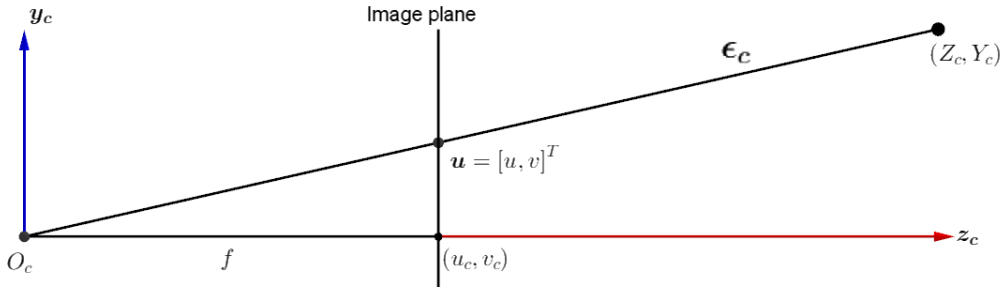


Figure 5.3: Projection of Fig. 5.2 on the zy-plane

From this visualization, it can be derived, from rules of right and similar triangles, that the ratio

between f and $v - v_c$ must be equal to the ratio between Z_c and Y_c:

$$\frac{v - v_c}{f} = \frac{Y_c}{Z_c} \tag{5.3}$$

This can then be solved for v:

$$v = \frac{fY_c}{Z_c} + v_c \tag{5.4}$$

Using a similar approach, just with the projection of Fig. 5.2 on the zx-plane, an expression for u can be derived:

$$u = \frac{fX_c}{Z_c} + u_c \tag{5.5}$$

It can be observed from eq. (5.4) and eq. (5.5), that these equations are nonlinear, due to the division of Z_c in both of the equations. To solve this, the $\boldsymbol{\epsilon_c}$ and $\boldsymbol{u}$ can be converted to homogenous coordinates. To convert cartesian coordinates to homogenous coordinates, an extra component is appended to the coordinate, which serves as a scaling factor for the other components. If this scaling factor is 1, then the homogenous coordinate is equal to the cartesian coordinate. $\boldsymbol{\epsilon_c}$ and $\boldsymbol{u}$ is converted to homogenous coordinates like so (the accented tilde indicates that the point/vector is stated in homogenous coordinates):

$$\boldsymbol{\epsilon_c} = \begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} \longrightarrow \tilde{\boldsymbol{\epsilon}}_{\boldsymbol{c}} = \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ W \end{bmatrix} \tag{5.6}$$

$$\boldsymbol{u} = \begin{bmatrix} u \\ v \end{bmatrix} \longrightarrow \tilde{\boldsymbol{u}} = \begin{bmatrix} u \\ v \\ Z_c \end{bmatrix} \tag{5.7}$$

where W is a scaling factor for the 3D coordinate, which should be set to 1 when projecting a known 3D coordinate to a point on the image plane. The scaling factor for $\boldsymbol{u}$ is equal to Z_c, the same as the Z_c-component in $\boldsymbol{\epsilon_c}$, which makes a linear transformation between $\boldsymbol{\epsilon}$ and $\boldsymbol{u}$ possible. By combining the homogeneous coordinates for $\boldsymbol{\epsilon_c}$ and $\boldsymbol{u}$ with eq. (5.5) and eq. (5.4), a linear transformation can be formulated:

$$\tilde{\boldsymbol{u}} = \begin{bmatrix} u \\ v \\ Z_c \end{bmatrix} = \begin{bmatrix} f_x & 0 & u_c \\ 0 & f_y & v_c \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} = \boldsymbol{K}\boldsymbol{\epsilon_c} \tag{5.8}$$

Equation (5.8) is a powerful and compact way to project a point in 3D space, to a image plane coordinate. It is important to note and emphasize however, that the components of $\boldsymbol{\epsilon_c}$ is with respect to the camera basis vectors and the optical center. The 3×3 matrix $\boldsymbol{K}$, also introduced in eq. (5.8) is the *intrinsic camera matrix*, and contains the internal parameters of the camera, in order to perform this $3D \longrightarrow 2D$ transformation. This can either be obtained by calibration or using

datasheet values. For the Intel LiDAR camera used in this project, the intrinsic parameters can be obtained directly through the Python API. Note that there are two different focal lengths, f_x and f_y. For high quality cameras, these are approximately the same, but can differ slightly. Intuitively, the unit of $\boldsymbol{\epsilon_c}$ must be world units (e.g. millimeters), and the unit of u, v, u_c and v_c must be pixels. Therefore it must also follow that the focal lenghts f_x and f_y must be a conversion between world units (mm) and pixels for this transformation to take place. When used in the camera matrix K in eq. (5.8), the focal lengths is defined as how many pixels per millimeter the camera has, i.e. the inverse of the pixel width and height in millimeters, respectively for f_x and f_y. Also, one must keep in mind that the obtained $\tilde{\boldsymbol{u}}$ will be in homogeneous coordinates, and must be normalized wrt. Z_c (when $Z_c = 1$, u and v are in cartesian coordinates, this can be obtained by dividing all components by Z_c).

5.2.2 Image plane projection in different reference frames

The method described in the previous section is a powerful tool to project a point in 3D to a image plane coordinate in 2D, but only when the 3D point is specified in coordinates with respect to the camera. In the case of this project, the camera is mounted on the end effector of the 6DOF Kinova manipulator, where it is most appropriate to express the 3D world coordinates with respect to the manipulator base frame vectors. All commands given to the manipulator, and all values retrieved from the manipulator are with respect to the manipulator itself, so if it should be able to use the data from the camera, it must be able to relate the world point, with respect to the manipulator base vectors, instead of the camera reference frame. This is illustrated in Fig. 5.4.

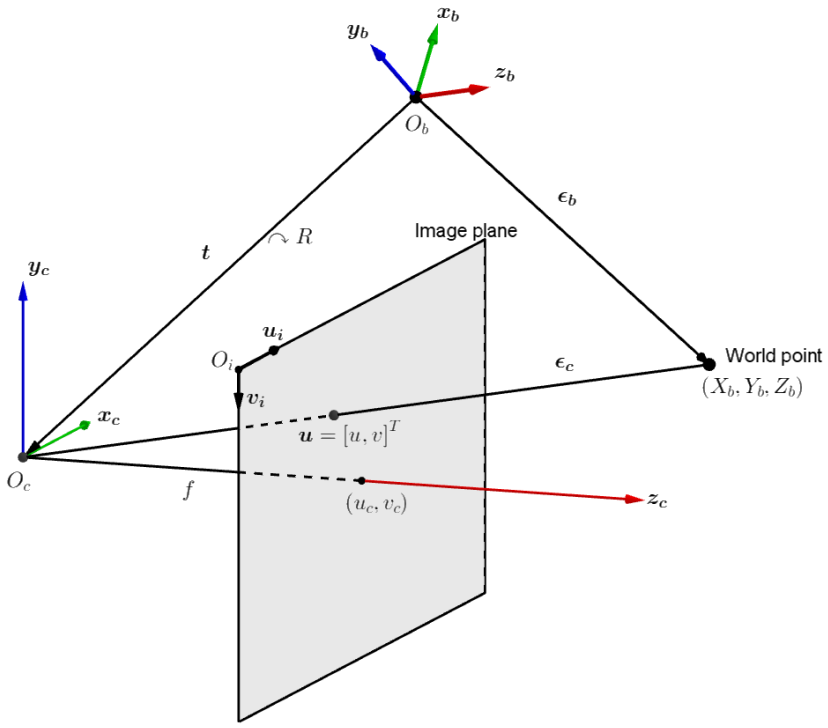


Figure 5.4: Representation of the world point with respect to another reference frame

Fig. 5.4 shows a figure similar to Fig. 5.2, but with the added coordinate system of the manipulator

base, originating from O_b and with the basis vectors $\boldsymbol{x_b}$, $\boldsymbol{y_b}$ and $\boldsymbol{z_b}$, the subscripted b denoting that it is the manipulator base reference frame. Similarly, the world point is now also defined with respect to the base reference frame, through the vector $\boldsymbol{\epsilon_b}$. $\boldsymbol{\epsilon_c}$ is still the same world point, just with respect to the camera reference frame. To relate the camera reference frame and the base reference frame, there is the translation vector $\boldsymbol{t}$ that is a vector between the origins of the manipulator base and camera reference frames. A rotation also takes place, which is facilitated through the rotation matrix $\boldsymbol{R}$. Recall from eq. (5.8) the projection of the world point (with respect to the camera reference frame) to the image:

$$\boldsymbol{u} = \boldsymbol{K}\boldsymbol{\epsilon_c} \tag{5.9}$$

Now the objective is to state the vector $\boldsymbol{\epsilon_c}$ using the vector $\boldsymbol{\epsilon_b}$ (the world point with respect to the manipulator base frame), the rotation matrix between the two reference frames $\boldsymbol{R}$, and the translation vector $\boldsymbol{t}$, between the camera origin and the manipulator base frame. The relation between $\boldsymbol{\epsilon_c}$ and $\boldsymbol{\epsilon_b}$ will first be derived, and later explained how it is related to the setup with the manipulator. Using the existing definitions for $\boldsymbol{\epsilon_b}$ (defined as the world point wrt. the manipulator base reference frame), $\boldsymbol{\epsilon_c}$ (world point wrt. the camera base frame), and defining $\boldsymbol{t}$ as the vector that translates the manipulator base reference frame, to the camera reference frame, and $\boldsymbol{R_c^b}$ as the rotation matrix, that is the rotation of the camera reference frame, with respect to the manipulator base reference frame, the following relation can be set up[33, p.37]:

$$\boldsymbol{\epsilon_b} = \boldsymbol{t} + \boldsymbol{R_c^b}\boldsymbol{\epsilon_c} \tag{5.10}$$

This can then be solved to obtain $\boldsymbol{\epsilon_c}$ (noting that for an orthogonal rotation matrix, $\boldsymbol{R^{-1}} = \boldsymbol{R^T}$ and $(\boldsymbol{R_c^b})^{-1} = \boldsymbol{R_b^c}$, [33, p.23, 38]):

$$\boldsymbol{\epsilon_c} = (\boldsymbol{R_c^b})^T\,\boldsymbol{\epsilon_b} - (\boldsymbol{R_c^b})^T\,\boldsymbol{t} = \boldsymbol{R_b^c}\,\boldsymbol{\epsilon_b} - \boldsymbol{R_b^c}\,\boldsymbol{t} \tag{5.11}$$

The above equation shows how $\boldsymbol{\epsilon_c}$ can be expressed, using the world coordinate with respect to the manipulator base, $\boldsymbol{\epsilon_b}$ and the rotation and translation between the camera and manipulator base reference frames. Note that $\boldsymbol{R_b^c}$ is now the rotation of the manipulator base reference frame, with respect to the camera reference frame i.e. the inverse or transpose of $\boldsymbol{R_c^b}$. If $\tilde{\boldsymbol{\epsilon_b}}$ is the 3D world coordinate in homogeneous coordinates, where the scaling is set to 1, $\tilde{\boldsymbol{\epsilon_b}} = \begin{bmatrix} X_b & Y_b & Z_b & 1 \end{bmatrix}^T$, the above equation can be stated in a more convenient way:

$$\boldsymbol{\epsilon_c} = \begin{bmatrix} \boldsymbol{R_b^c} & -\boldsymbol{R_b^c}\,\boldsymbol{t} \end{bmatrix} \boldsymbol{\epsilon_b} \tag{5.12}$$

Now, eq. (5.8) can be combined with eq. (5.12) to yield the final equation, to project a 3D world point onto a 2D image plane in pixels:

$$\boldsymbol{u} = \boldsymbol{K} \begin{bmatrix} \boldsymbol{R_b^c} & -\boldsymbol{R_b^c}\,\boldsymbol{t} \end{bmatrix} \boldsymbol{\epsilon_b} = \boldsymbol{M}\,\boldsymbol{\epsilon_b} \tag{5.13}$$

To summarize, $\tilde{\boldsymbol{u}}$ is the 2D image plane projection in homogeneous coordinates, $\boldsymbol{K}$ is the intrinsic camera matrix (defined in eq. (5.8)), $\boldsymbol{R_b^c}$ is the orthogonal rotation matrix, that describes the rotation of the manipulator base reference frame with respect to the camera reference frame, $\boldsymbol{t}$ is the translation vector between the manipulator base origin and the camera origin, and finally $\tilde{\boldsymbol{\epsilon_b}}$ is the 3D world coordinate in homogeneous coordinates, where the scaling is set to 1.

5.2.3 3D position from 2D image projection

In eq. (5.13) it was stated how to go from a point in 3D space, to a 2D image plane projection in pixels. However, this transformation shows a reduction in dimension and thereby loss of information, therefore it is not a trivial task to go the other way. The representation Fig. 5.4 also shows, that every point on the line in the direction of $\boldsymbol{\epsilon_c}$, originating from O_c, will map to the same point on the image plane. Thereby, if one performs the inverse operation of eq. (5.13), one would end up with a 1D solution space, i.e. a straight line. In this section, this inverse operation will be defined. In order to perform the inverse operation, eq. (5.13) will need to be solved for $\boldsymbol{\epsilon_b}$. Since $\boldsymbol{M}$ is a 3×4 matrix, it is clear that it is not invertible, and hence the pseudo-inverse of $\boldsymbol{M}$ must be used instead, where the supercripted $+$ denotes the pseudo-inverse:

$$\tilde{\boldsymbol{\epsilon_0}} = \boldsymbol{M}^{+}\tilde{\boldsymbol{u}} \tag{5.14}$$

From the pseudoinverse ($\boldsymbol{M}^{+} = \boldsymbol{M}^{T}(\boldsymbol{M}\boldsymbol{M}^{T})^{-1}$), $\boldsymbol{M}^{+}$ will be a 4×3 matrix. However, this equation does not necessarily provide us with $\boldsymbol{\epsilon_b}$ from eq. (5.13), it will just give a point on the line, where all the points on this line will project to $\boldsymbol{u}$ [40, p.601]. $\tilde{\boldsymbol{\epsilon_0}}$ from the above equation is converted to cartesian coordinates, as it is more convenient for the following derivation:

$$\tilde{\boldsymbol{\epsilon_0}} = \begin{bmatrix} X_0 \\ Y_0 \\ Z_0 \\ W_0 \end{bmatrix} \Rightarrow \boldsymbol{\epsilon_0} = \begin{bmatrix} X_0/W_0 \\ Y_0/W_0 \\ Z_0/W_0 \end{bmatrix} \tag{5.15}$$

$\boldsymbol{\epsilon_0}$ is then a point on the line, where every point maps to $\boldsymbol{u}$ in the image plane. Recalling that this line will also pass through O_c, which can be represented by $\boldsymbol{t}$, two points on the line is now known, and a line parameterization can be made:

$$\boldsymbol{\epsilon_b}(\lambda) = \boldsymbol{t} + \lambda(\boldsymbol{\epsilon_0} - \boldsymbol{t}) = \boldsymbol{t}(1-\lambda) + \lambda\boldsymbol{\epsilon_0} \tag{5.16}$$

The parameter λ can be obtained by eg. triangulation, or a distance reading, where the latter is most relevant for this project, as it can be measured directly using the LiDAR on the camera, or estimated using the neural network or other algorithms. If a distance reading from the camera is denoted by d, then the following relation can be set up:

$$d = \left\| \lambda(\boldsymbol{\epsilon_0} - \boldsymbol{t}) \right\| \tag{5.17}$$

$\lambda(\boldsymbol{\epsilon_0'} - \boldsymbol{t})$ denotes the vector going from the camera to the world point. This is then solved for λ, to obtain a coordinate relative to the manipulator base, $\boldsymbol{\epsilon_b}$ using eq. (5.16).

5.2.4 Obtaining translation vector and rotation matrix

In section 4.1.7, the Denavit-Hartenberg parameters were presented, and it was shown how these can be used to construct a transformation matrix using structured parameters, for the manipulator. These transformation matrices are used for linking eg. the manipulator base reference frame, to the end effector reference frame, and position in space. $\boldsymbol{A}$ transformation matrix for this purpose is denoted as $\boldsymbol{A_i^{i-1}}$, and this transformation matrix will specify the position and orientation of joint i with respect to the position and orientation of joint $i-1$. All the transformation matrices are structured in the following way:

$$A_i^{i-1} = \begin{bmatrix} R_i^{i-1} & t \\ 0 & 1 \end{bmatrix} \tag{5.18}$$

The notation follows the previous analysis, where R_i^{i-1} is a 3×3 orthogonal matrix, that describes the rotation of reference frame i with respect to reference frame $i-1$, and t is the translational vector from joint $i-1$ to i with respect to reference frame $i-1$. If the transformation matrices from the manipulator base to the camera are multiplied in sequence, it will then provide both a rotation and a translation between the camera and the manipulator base, which can be used in eq. (5.13):

$$T_c^b = A_1^0 A_2^1 A_3^2 A_4^3 A_5^4 A_6^5 A_g^6 A_c^g = \begin{bmatrix} R_c^b & t \\ 0^T & 1 \end{bmatrix} \tag{5.19}$$

This gives the rotation matrix R_c^b, but for eq. (5.13), the R_b^c rotation matrix is needed. Fortunately, as described previously, this can be obtained by the inverse or the transpose of the R_c^b matrix:

$$R_b^c = (R_c^b)^{-1} = (R_c^b)^T \tag{5.20}$$

All of the transformation matrices for the manipulator can be found in appendix A, but the transformation matrix between the gripper and the camera, A_c^6, is still needed. On Fig. 5.5, a 3D drawing of the camera mount and gripper can be seen. The lower green dot is the gripper origin, and the upper green dot is the camera origin, and the dY and dZ distances on the picture are the displacements with regards to the gripper reference frame.

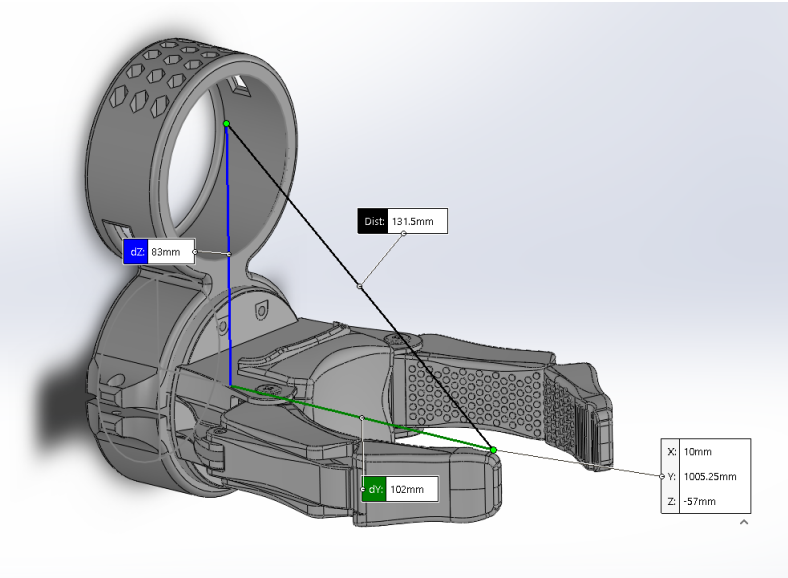


Figure 5.5: 3D rendering of the camera mount and the offset wrt. the gripper origin

The x-axis of the camera will be opposite the x-axis of the gripper, the y-axis of the camera will be opposite the y-axis of the gripper, but the two z-axes will be aligned. This was derived based on the camera reference frame, where it is defined that the image plane is parallel to the face of the camera, and the z-axis is perpendicular to this plane, and has its origin in the camera

origin (principal point). The y-axis is positive in the downwards direction (with respect to the orientation on Fig. 5.5) and the x-axis is set such the reference frame satisfies the right hand rule. This is compared to the gripper reference frame, seen in [32]. If a a tilted camera mount is used instead, a rotation with the given tilted angle should just be multiplied to this matrix. The translation vector is just the numerical values from Fig. 5.5.

$$A_c^g = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0.083 \\ 0 & 0 & 1 & -0.102 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.21}$$

5.3 Implementation of projections

This section will show how the above equations and methods are implemented in Python. The code implementation in the project is made modular, to have better organized code and documentation. This means that everything, that has to do with the projections and transformation with the camera, is organized in a seperate file called `Projection.py`. When imported via `import Projection`, it will load the necessary libraries. The file furthermore contains a class, `Projection`, that when instantiated will connect to the camera, and handle all the communication. A description of this file and class, will follow in the next subsections. The derivations in this chapter, was based on a transformation from the manipulator base to camera, but the implementation will consider the gripper reference frame as the "base" reference frame, and the transformation will therefore only be between the camera and gripper.

The code of the `Projection.py` and the `Projection` class can be seen in listing 5.3. In the `Projection` class constructor, it takes a camera object of type `Camera` (`cam_obj`), a displacement in y and z direction, that represents the displacement of the camera, relative to the gripper origin, and with respect to the gripper coordinate system (`cam_y` and `cam_z`). `width` and `height` is the image height and width. In the constructor, the rotation matrices from this section, between gripper and camera are defined as numpy matrices, along with the camera matrix $\boldsymbol{K}$ and the combined camera and transformation matrix $\boldsymbol{M}$. The inverse of $\boldsymbol{M}$ is also computed in the constructor as it stays constant all the time.

`get_3d_point(center, d)`, takes a coordinate on the image `center`, and applies eq. (5.14) and eq. (5.16), in order to obtain the vector from the gripper to the plant, and returns that vector.
`get_proj(X)` does the opposite of `get_3d_point`, as this function takes a 3D point and uses eq. (5.13) to compute the projection of a 3D point relative to the gripper, onto the image. This function is useful to see how the estimator works, but that will be covered in chapter 7 and chapter 8.

Listing 5.3: `Projection.py`

```
1 import math
2 import numpy as np
3 from custom_utils import saturation
```

```
4
5  class Projection:
6      def __init__(self, cam_obj, cam_y, cam_z, width,height):
7          self.R = np.matrix([[-1,0,0],[0,-1,0],[0,0,1]])
8          self.K = np.matrix([[cam_obj.fx,0,cam_obj.u0],[0, cam_obj.fy, cam_obj.
               v0],[0,0,1]])
9          self.t = np.matrix([[0],[cam_y],[cam_z]])
10         self.M = self.K * np.c_[np.transpose(self.R), -np.transpose(self.R)*
               self.t]
11         self.Minv = np.linalg.pinv(self.M)
12         self.width = width
13         self.height = height
14
15     def get3d_point(self, center, d):
16         epsilon = self.Minv * np.matrix([[center[0]], [center[1]], [1]])
17         epsilon = epsilon*(1/float(epsilon[3]))
18         epsilon = epsilon[0:3]
19         l = d/np.linalg.norm(epsilon-self.t)
20         epsilon = self.t+(epsilon*l)
21         return epsilon
22
23     def get_proj(self, X):
24         est_dot = self.M*np.matrix([[float(X[0])],[float(X[1])],[float(X[2])
               ],[1]])
25         est_dot = est_dot/float(est_dot[2])
26         est_dot_x = saturation(int(est_dot[0]), self.width-1, 0)
27         est_dot_y = saturation(int(est_dot[1]), self.height-1, 0)
28         return est_dot_x, est_dot_y
```

Chapter 6

Object detection with deep neural network

Since the final system will be able to detect weeds a look will be taken at how a neural network is used to detect objects. This chapter will begin with an overall look at how deep neural networks are used for object detection more specifically convolutional neural networks, and afterwards the focus will be shifted to YOLOv5, which will be the specific network architecture used in this project.

6.1 Convolutional neural network

A Convolutional Neural Network (CNN) is a type of neural networks which is primarily used for object detection in images [41]. An image is created from three color matrices (RGB, one matrix for for each color) with dimensions being equal to the resolution of the image. This means that a single image, with a good resolution, will contain a huge amount of data. The convolutional layer, in the network, can reduce the dimension of the image while still preserving the essential features in the image, that will be used for object detection in the image. The kernel is the specific tool used in the convolutional layer which gatherers data in order to simplify it. The kernel traverses through the three color matrices and computes the dot product between the kernel and sub part of the individual matrices and finally sums the three parts together in order to create a resulting matrix with a depth of one. This is opposed to the dimension of three it takes as an input. This progress can be visualized in Fig. 6.1, where the kernel is of size 2x2 with all the entries being 1, and the green and blue matrix all have the entries 0. The kernel traverses through the matrix starting in the upper left corner (black box), and then the dot product is applied and the result is transferred to the output. Then the kernel shifts 1 column to the left (dotted box), computes the dot product and transfers all the data to the output, and so on. Once the kernel reaches the last column, the kernel shifts one row down and goes all the way back to the first column, and then it continues shifting one column to the left computing the dot product and transferring the data as it goes along. The goal of doing these convolutional operations is to extract features from the three input matrices, the lowest layers, in the total neural network, captures low level features such as edges, colors and with more layers the abstraction levels increases which results in a network having having an understanding of what is going on in the image. The convolutional layers in the network, can be seen as a way of filtering the image, to highlight the important features in the image. The specific features that are highlighted, depends on the weights in the kernel. After the convolutions has been carried out, the extracted features are further reduced in dimension by the process of pooling, which can be max pooling or average pooling, max pooling is generally better than average pooling since it discards noise. Max pooling is shown in Fig. 6.2 where it can be seen that the maximum value, inside the sub matrix used in the pooling, is transferred to the output. This sub matrix also traverse through the matrix the same way as in the convolutional layer. The convolutional layer and pooling layer goes together as a pair and creates a single layer in a convolutional neural network. After the pairs of convolutional and pooling layers has

extracted features from the original image, and reduced the dimensions the data is flattened and then fed into fully connected layers part of the total neural network, which is used to classify these different features into different objects [42]. These fully connected layers will be elaborated on in the next section.

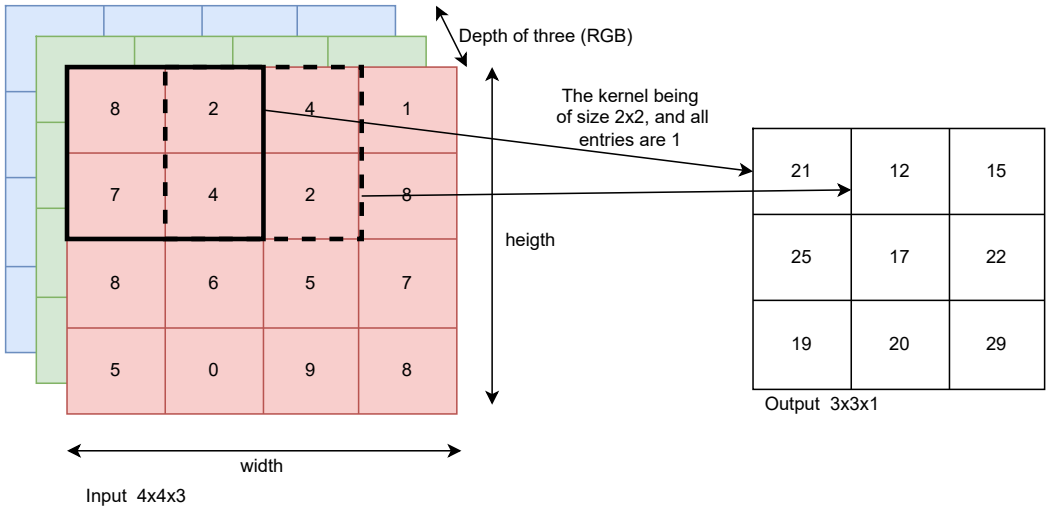


Figure 6.1: Figures shows how the kernel, in the Convolutional layer, can reduce data size

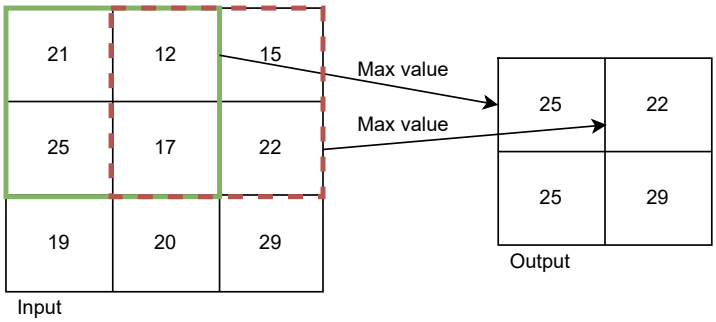


Figure 6.2: The pooling used is the 'max pooling' which takes the highest value and transfers to the output

Fully connected layers

The flattened data from the convolutional layers are passed into the fully connected layers as shown in Fig. 6.3. This network has, for simplicity, only three inputs shown but the amount of inputs should match the amount of outputs from the previous convolutional layer. These three inputs (input variable 1-3) are then fed forward into the first hidden layer (hidden layer 1) which then feeds it forward again to hidden layer 2, which yields the output. In the bottom of the figure a zoomed in view of a single neuron is shown. Inside these neurons is where the actual data managing is happening, and it can be seen that the arithmetic operations happening in the fully connected layers is fairly straightforward. The operation is a sum of all the inputs $x_1, x_2 ... x_n$ which are scaled individually with a factor $w_1, w_2 ... w_n$. This sum is then passed to the activation function. The activation functions used on YOLOv5, which will be used in this project,

are Sigmoid and Leaky ReLU [43]. A Sigmoid activation function is a continuous function which can take any input value and map it into a value between 0 and 1 [44]. The Leaky ReLU is based on a ReLU, which outputs zero, until the input reaches a threshold. When the input is larger than the threshold, it gives an output which is equal the input. The 'leaky' part added to the ReLU term means that instead of the output being 0 when the input is below the threshold (the slope being 0), then a small slope is introduced [45], which gives a scaled negative output, of the input. After the activation function the neuron gives an output y which can then be used by the next layers in the network.

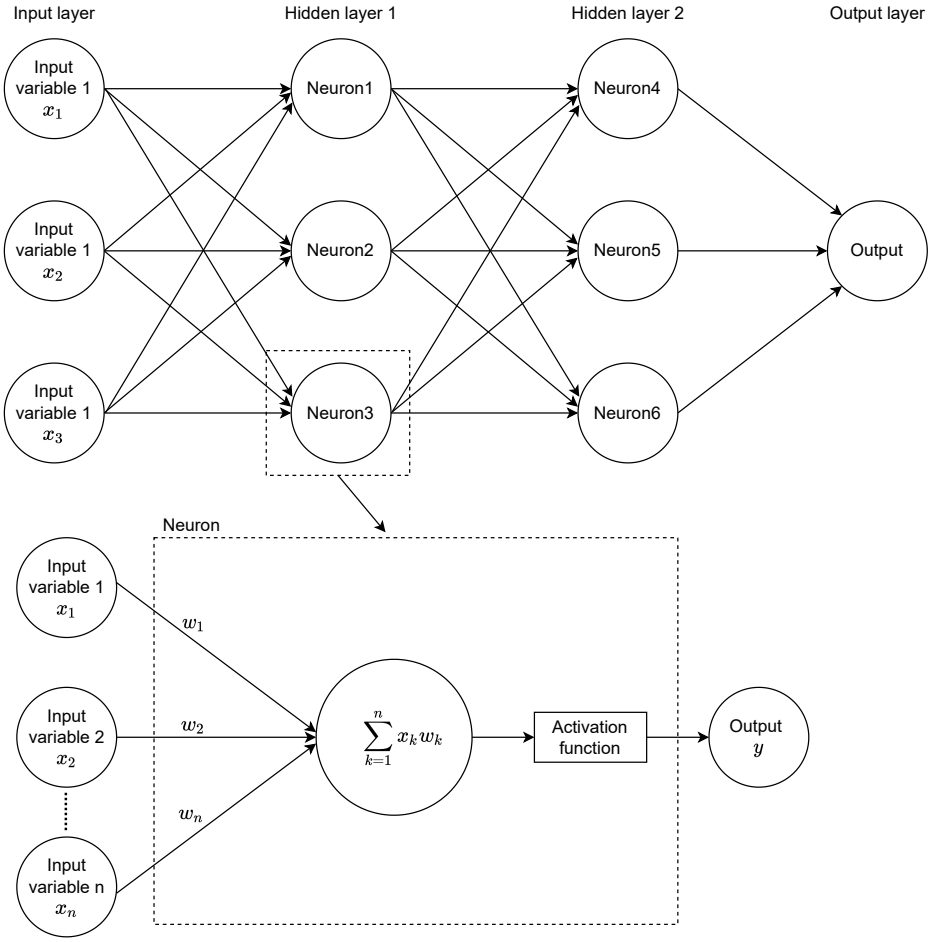


Figure 6.3: Overview of a neural network

6.1.1 Training and transfer learning of a network

When a neural network is trained the goal is that this new network will be capable of detecting a certain objects, or multiple objects. The way which the network becomes capable of detecting different items in images is by changing these different weights w_n in the network which connects the different neurons between the layers. The process of changing these weight is not found analytically, meaning there is not only one unique solution, the changing of these weights is an iterative process. The training process involves the network going through the gathered training data multiple times, and finding the best weights which results in the most optimal output [46].

Because of this iterative process the training of a neural network is a heavy task to execute, that requires a lot of training data, and a lot of computations, in order to obtain a good network. Therefore it is natural to think, if a good network that has been trained for several days with millions of samples, can be modified slightly to serve other purposes, than what it was designed for, to avoid the need of collecting additional enormous amounts of data, and training from scratch, which takes a lot of time. This is where *transfer learning* is particularly useful. Transfer learning is the method of re-training a network to match some specific data. Transfer learning relies on the fact that if a neural network is used to detect objects, like a cat, in images, then this network have all the functioning layers to do so. These layers starts by recognising simple patterns such as lines, which is then combined into figures, which is then combined into more abstract figures and so on. On the final layers the more abstract figures can then be classified into the object of interest in this case a cat. This means that the bottom layers which handles the lower abstraction levels are not needed to be changed since detecting a new object, like a dog, will be similar to detecting a cat, meaning lines, figures and so on still has to be detected as well. The only thing being changed, when doing transfer learning, will be the top layers or the classification layers which connects these different types of abstract figures into an object [47]. This can be done by *freezing* the bottom layers of the network, meaning the weights cannot be changed in the part of the network that is frozen. The network is then trained with new data material that matches the new desired output. Finally, when the new classification layers are done being trained, the bottom layers can be unfrozen and a final fine tuning training session can be carried out on the complete network [48].

6.1.2 YOLOv5 and PyTorch

Since this project will focus on object detection, and more specifically detection of weeds, then different neural networks can be used which each has their own pros and cons in relations to speed and accuracy. The supervisors suggested looking into the neural network *YOLO* which stands for "you only look once", and during the initial testing, of this network, it has been concluded that the newest version of YOLO, namely YOLOv5, is sufficient in relation to the scope of this project. YOLOv5 is introduced through *PyTorch* which is an optimized machine learning library which can be used through python [29] [49].

6.2 Transfer learning with YOLOv5

YOLOv5 already has been trained on a data set called COCO, which is a data set containing 300 thousand images and contains 80 categories of different objects which the network can classify [50]. These objects are everyday items such as chairs, dinning tables, cups and potted plants. This means that the network is already trained very thorough in order to detect these 80 different objects, but since the object detection list does not contain any plants, and only potted plants, transfer learning has to be applied to the network in order ensure that the new network can detect the weeds which are of interest to this project. In order to transfer learn on such a network some different steps has to be carried out, and these are as following:

1. Gather training images (training data)

- Apply bounding boxes to the gathered images
- Post processing (Apply data augmentation)

2. re-train the network with the above gathered images

6.2.1 Gathering images

In order to transfer learn on an already existing network a lot of new images is needed a rule thumb says that at least 1500 images per new class is needed in order to create a good network [51]. Generally, the more images the better the resulting network will be. Apart from the data set containing many images these images also needs to taken from different angles, in order to further increase the performance of the network [52]. Generally it can be said that if the network is trained with images from a specific angle the network can only detect the object from this angle, meaning that in order to create a good overall network a lot of images in various positions and angles is needed.

To automate the process of gathering images a script has been written which can move the robot manipulator, which has the camera mounted on its end effector, and take a photo of the plant and save it. The code used can be seen in listing 6.1. On line 1-6 the x, y, z coordinates for the hand are randomly generated. On line 7 the `send_pose` is used to send the three coordinates to the robot manipulator which then moves the end effector to these coordinates. Line 8-10 is where the color image of the weed is taken, and saved to a folder such that it can be used later in the transfer learning process. An example of the images taken with this code can be seen in Fig. 6.6

Listing 6.1: Image gathering

```
1  Height = 0.01889*random.randint(0,9)+0.05 #Random height
2  Deg = random.randint(0,180) #Random amount of degrees
3  r = random.randint(0,9)*0.0127778+0.01 #[m]
4  x = r*cos(deg2rad(-90-(Deg)))+float(Plant[0]) #Hand x position [m]
5  y = r*sin(deg2rad(-90-(Deg)))+float(Plant[1]) #Hand y postion [m]
6  z = Height                                    #hand z position [m]
7  send_pose(base, base_cyclic,x,y,z) #Move the robot manipulator
8  color_image, depth_image, depth_scale, intr = TakePhoto(n) #TakePhoto
9  folderStructure(color_image, depth_image, depth_scale, intr) #Save data
10 SavingImage(color_image, n) #Save image
```

Figure 6.4: Weed1

Figure 6.5: Weed2

Figure 6.6: Different images of the weed

6.2.2 Automatic drawing of bounding box

Once the images has been gathered for the transfer learning, the network needs to know where the object of interest is located in the image. This is done by a *bounding box*. The bounding box is, as the name implies, a box which bounds the object, which is to be detected. This bounding box has some coordinates on the images which is given to the network. Drawing bounding boxes, on images, can be a time consuming task if done manually, which is why an automatic approach has been looked at in the project.
When performing the automatic bounding box labelling, the position and orientation of the camera mounted on the gripper, can be very precisely known using direct kinematics, and using the methods in the previous section, a known position in 3D space, relative to the manipulator base, can be projected to the image, and the pixel coordinates can be calculated. This is illustrated on Fig. 6.7 and Fig. 6.8. On Fig. 6.7, a 3D box is drawn, and this box needs to enclose the plant as much as possible. By creating the box as shown on the figure, it will create 8 distinct points, one for each corner of the box, that make out the box. On the figure, it is also shown how these 8 points project towards the optical center, O_c.

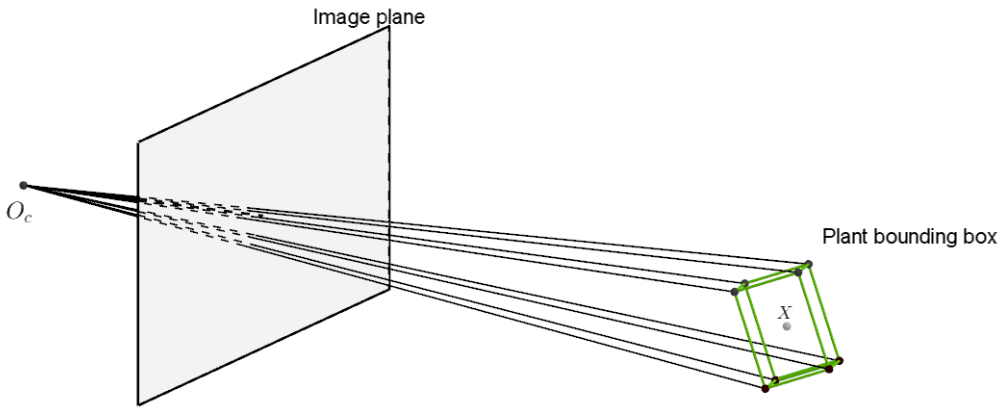


Figure 6.7: Plant bounding box in 3D

If the view is then changed, to view from behind the image plane, it can be seen how the 8 points and the sides connecting the corners, are projected onto the image plane, as seen on Fig. 6.8. The dashed line indicates the projection of the sides of the 3D bounding box. The idea is then, that

a 2D rectangle will enclose all the projected points as shown on Fig. 6.8. Then in theory the 2D bounding box should enclose all of the plant. Depending on the orientation of the 3D box relative to the camera, it might also take some of the background within the 2D bounding box, which can be a problem since the network will take all features inside the bounding box and regard them as the correct object. During the bounding box creation it has been observed that the bounding box is not unnecessary big. Furthermore, if many different backgrounds are used during the gathering of the training images, then the new trained network should be able to generalize only on the item which is the same in all of the images i.e. the plant while sorting out the background. The 2D bounding box also does not consider the orientation of the 3D bounding box, and will always have the sides parallel to the edges of the image, but this the kind of simple bounding box that the detection algorithm (YOLOv5) can handle, so for this purpose it should be sufficient.

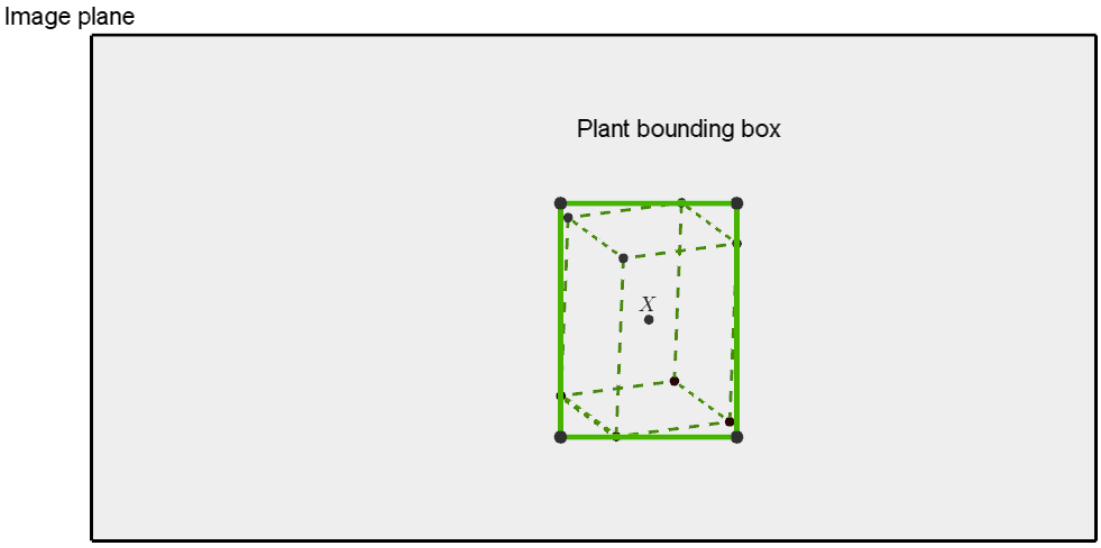


Figure 6.8: Projection of plant bounding box on image

Once the equations from the previous sections are known and understood, the implementation of the automatic labeling and bounding box drawing is relatively simple. The implementation described here, will assume that the sides of the 3D bounding box is parallel to the reference frame of the manipulator base, it will assume that all the data has been collected beforehand, and that every picture taken has a text file associated with only that picture, that will contain the joint angles of the manipulator. Then for every picture in the collected data, the procedure is:

1. Given a center coordinate and dimensions of the plant, calculate the coordinates of the 8 corners of the 3D box, $p_1, p_2, ...p_8$
2. Retrieve joint angles, $q_1, q_2, q_3, q_4, q_5, q_6$
3. Compute the transformation matrix from base to camera using eq. (5.19)
4. Isolate the R_b^c rotation matrix and the $\boldsymbol{t}$ translation vector using eq. (5.19) and eq. (5.20)
5. Use eq. (5.13) to compute the 2D projections on the image plane, $\tilde{\boldsymbol{u}}_1, \tilde{\boldsymbol{u}}_2 ... \tilde{\boldsymbol{u}}_8$

6. Convert $\tilde{\boldsymbol{u}}_1, \tilde{\boldsymbol{u}}_2 ... \tilde{\boldsymbol{u}}_8$ to cartesian coordinates and obtain the pixel coordinates of all points
7. In $\boldsymbol{u}_1, \boldsymbol{u}_2 ... \boldsymbol{u}_8$, find the largest and the smallest values for u and v (width and height coordinates in pixels), and pair them up such that there are now two coordinates: (u_{min}, v_{min}) and (u_{max}, v_{max}). These are the top left and bottom right coordinates of the 2D bounding box. One must also make sure that these coordinates are within the image, meaning none of the coordinates must be negative, or larger than the width or height of the image, respectively.
8. Calculate the center coordinate and the height and width of the bounding box, and normalize these values with respect to the width and height of the image, such all values lie between 0 and 1
9. Format a string as: `label center_u center_v width height` of the bounding box, all in normalized values
10. Save this string in a text file associated to the image

These above steps were implemented in Matlab and used on several images from several different angles, and produced a good bounding box around the plant. The implementation can be seen on the github page for the project [37]. Two samples of the output can be seen on Fig. 6.9. Most of the bounding boxes are of good quality, although some seem to be slightly misaligned. This might be improved by providing a more accurate coordinate and/or better measurements of the plant dimension.

(a) Weed1 with bounding box

(b) Weed2 with bounding box

Figure 6.9: Bounding boxes on the images shown in Fig. 6.6

6.2.3 Post processing

In section 6.2.1 it was stated the in order to train a class properly at least 1500 images is required in order to have a proper working neural network. However, when only a small data set is gathered *data augmentation* can be applied to the images in order to increase the size of the data set. Data augmentation is adding copies of the original images which are changed in different ways. An example of data augmentation can be seen in Fig. 6.10 which shows the data augmentation of the weeds in Fig. 6.9. The data augmentation which is used on the gathered training data can increase the data set size, in this project a factor of four is used, meaning each gathered image is augmented in four different ways. The code which were used to create the data augmentation can be seen in listing 6.2. On line 1 the albumentation library is imported, which is used for image augmentation, and on line 3-7 the different data augmentation parameters are defined. The data

augmentation can either flip the image (line 4), scale and crop the image (line 5) or rotate the image (line 6). These different parameters are all applied randomly with the parameter `p = 0.5` which is the probability (0.5 = 50%) of the specific data augmentation being applied to the image and bounding box. On line 7 color jitter is also applied randomly which changes the brightness, contrast, saturation and hue of the image. On line 9 the above stated augmentations are applied to the images and bounding boxes, and on line 10 and 11 the processed image and bounding box is extracted.

(a) Weed1 with applied data augmentation

(b) Weed2 with applied data augmentation

Figure 6.10: Post processing which shows the data augmentation of the weeds shown in Fig. 6.9

Listing 6.2: Post processing

```
1  import albumentations as A
2
3  transform = A.Compose([
4              A.HorizontalFlip(p=0.5),
5              A.RandomSizedBBoxSafeCrop(1080, 1920, p = 0.5),
6              A.Rotate(limit = [-30, 30],p = 0.5),
7              A.ColorJitter(brightness=(0.75, 1.65), contrast=(0.8, 1.2),
                   saturation=(0.25, 2), hue=(-0.2, 0.2))
8          ], bbox_params=A.BboxParams(format='yolo'))
9  transformed = transform(image=color_image, bboxes=bboxes)
10 transformed_image = transformed['image']
11 transformed_bboxes = transformed['bboxes']
```

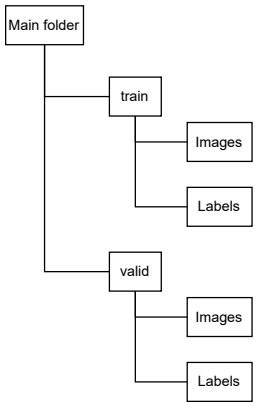
6.2.4 Practical transfer learning

When the images and bounding boxes has been gathered and data augmentation has been applied, transfer learning can be applied to the network. Since the network knows in which part of the images the desired object is located, by the help of the bounding box data, the network can learn to distinguish the desired object from the surrounding environment. Transfer learning on the YOLOv5 network has been made very user friendly, meaning the execution of the transfer learning is easy, once all the setup and image gathering/bounding box drawing has been carried out. Before transfer learning can be carried out, two things needs to be ensured. Firstly, the folder structure of the images and bounding box data has to be created in a specific way, then a .yaml file has to be created. This .yaml file specifies the location of training data for the transfer learning process. These two details are shown in Fig. 6.11.

Listing (6.3) .yaml file content

```
train: ../train/images
val: ../valid/images

nc: 1
names: ['Weed']
```



(a) Structure of data for transfer learning, labels = bounding box data

Figure 6.11: The yaml file content is shown in listing 6.3, and the folder structure can be seen in Fig. 6.11a

Once all of the above practical aspects has been taken care of, the transfer learning can be applied to the network. The transfer learning itself is carried out by running the command shown in listing 6.4.

Listing 6.4: Command to start training

```
python train.py --batch 48 --weights yolov5n.pt --data Weed3.yaml --epochs
    300 --cache --img 512 --freeze 10 --workers 4
```

The reason the transfer learning itself is so straight forward to use is is because a training script is included in the YOLOv5 repository from github, called `train.py` which carries out the training. This function just needs some different input parameters, in order to execute in a desired manner. After the `python train.py` command, different parameters are specified, and these are marked with `--` in the beginning, and the different inputs which are important for this specific project has been elaborated on below.

`--weights yolov5n.pt` means that the network will transfer learn based on the YOLOv5n network, which has been pre-trained on the COCO data set as described earlier. There are different types of the YOLOv5 networks which can be chosen from, namely n, s, m, l and x, for this project, the 'n' type, which is the smallest, was tested and assessed to be sufficient. The network runs on the Jetson, which is a powerful device compared to its size, but also has limitations. The smallest network provides the fastest inference time, so in order to get as fast recognition as possible, the smallest network was chosen. Another thing to keep in mind here is also that the bigger the network the longer it will take to train, but also the higher the accuracy can be obtained. If fine tuning is wanted to be carried out, the network loaded can be a custom made one.

`--freeze 10` will freeze the bottom 10 layers of the network, which is also called the backbone. The reason for freezing layers was explained in section 6.1.1. More or less layers, than the 10 chosen, could also be picked which will have an affect on the resulting neural network, and the time it takes to train. However, a small comparison of a YOLOv5m network was made which showed that freezing the backbone resulted in a transfer learning time reduction of about 20%,

compared to training from scratch, but still keeping a high mAP.
mAP stands for *mean average precision* and is a number between 0-1 which shows how certain the network is about detecting objects [53]. The formula for calculating the mAP can be seen in eq. (6.1), where N is equal the total number of images (or other dataset), which the mAP is to be found for. AP_i is the average precision, where precision is defined as: $P = \frac{TP}{TP+FP}$, where TP is the true positives[1], and FP is the false positives[2]. TP and FP are defined based on a parameter called *intersection over union* (IoU), which is defined as the ratio between the overlapped area, between the bounding box found by the neural network and the labelled bounding box given to the network, and the total area of the two. The value of the IoU defines if the detection is regarded as a TP or FP, and usually a threshold value of 0.5 is used. This means that if the IoU is < 0.5 then the detection is regarded as a FP and if IoU is > 0.5 then the detection is regarded as TP. Another threshold limit, which is also commonly used, is stepping it from 0.5 to 0.95 with a step size of 0.05, were it still holds that precision values below the threshold results in a FP, and above the threshold results in a TP. After the IoU has been calculated, the TPs and FPs can be found, then the precision can be found, which an average can be taken of, and finally the mean value of the different average precisions and be found, in order to calculate the mAP [54].

$$mAP = \frac{1}{N} \sum_{i=1}^{N} AP_i \tag{6.1}$$

`--epochs 300` epochs is the amount of times the collected data is iterated through, there is some relationship between the higher the epoch number the more precise the network. However, to many iterations can result in over fitting of the resulting network which one should be aware of [55].

`--img 512` the image resolution is scaled to match the resolution specific in this case the resolution will be 512x512. The higher the resolution the finer details the network can distinguish in the images, but also the longer time the network will take to train. A test was carried out which compared a resolution of 640x640 to 512x512, on the same data set, in order to see if any improvements are achieved. The conclusion is that the mAP increased with 1.12%, but the training time also increased from 3 hours and 48 minutes to 5 hours and 36 min. It was decided that the longer training time did not justify the small mAP improvement, which is why the size of 512 was used.

`--data Weed3.yaml` is the .yaml file explained earlier which specifies the specific training data which is to be used during the training process.

`--batch 48`, batch is the batch size used during training, and this batch size is the amount of small batches the total amount of training images is divided into. There is no exact answer on how big the batch size should be, and it also depends heavily on the used equipment, hyperparameters as well as the amount of images used, but a rule of thumb says it should be as high as possible [56].

[1]The network detects a correct object as being a correct object
[2]The network detects a wrong object as being a correct object

`--workers 4` The amount of workers specifies how many batches can be handled in parallel execution, this number can be increased to optimize the computational efficiency during training [57].

After the transfer learning has been executed the new network is ready to be applied in the real world. A small example of the network Weed7's performance can be seen in Fig. 6.12. This Weed7 network is fine tuned with an additional 861 images on Weed6 which were trained with 3180 images, meaning the total amount of training images is 4041 and this amount is definitely above the suggested 1500 images. The detection is considered acceptable, since it is detecting all of the weeds, with a mAP ranging from 0.9-1.0 (90%-100%). A comparison between the different trained networks, made during this project, will be made in the next section.

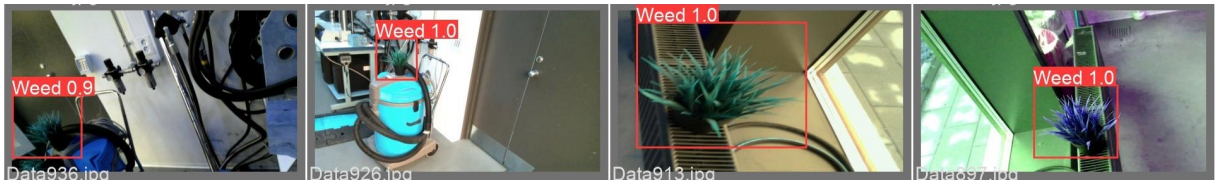


Figure 6.12: Images examples from Weed7 network

6.2.5 Comparison of different transfer learning runs

As stated previously, the goal with training a neural network is to be able to detect weeds. More specifically meaning that the network should be able to detect the weed with a high confidence level as well as having little to no false positives (FP) and false negatives (FN) [3].
When a neural network is being trained many parameters can be adjusted, in order to achieve the before mentioned goals. Firstly, the data which is given to the network can be changed for the better and this is done through parameters such as increasing the amount of images, having a high variety in how the images are taken, data augmentation on the images and so on. There are also internal parameters, which can be adjusted, related to the network and how training is carried out on the network, these being adjusting the learning rate, freezing layers, amount of epochs, etc.
To compare the different trained networks, which has been made during this project, a comparison has to be made. Doing such a comparison can be done with the help of different performance parameters which are gathered during the training of each network, as well as conducting individually testing of the networks. The parameters which this comparison will focus are mAP, and false positives. The different trained networks which will be looked at are represented in table 6.1. In the first column, named 'Session', the different trained networks' names are stated. The name 'WeedX' relates to the resulting network trained on the specified parameters, and 'X' is just a consecutive number, used for naming. The second column, in the table, 'images' is the amount of images taken of the plant, third column "Post processed images" is the total amount of images used to train the network, which is the same amount after post processing has been applied. For 'Weed3' the total number of training images were 130. The fourth column 'background images' is the percentage of the total amount of images, used for training, which were background images. In the fifth column 'Trained from' the network which the specific network were trained from is stated. In the last two columns, column six 'Avg confidence' and the seventh

[3]The network fails to detect the correct object

column 'Avg FP', two experiments has been carried out in order to compare the performance of the different networks. In the 'Avg confidence' test, a total of 146 images has been taken of the plant in various positions, angles and environments in order to fully test the networks. The Avg confidence value is calculated based on each network being fed all the 146 images, and then the confidence level for each image has been summed up, and then divided with the total amount of images in order to find the average, and this is done for each network. In the 'Avg FP' test, a similar test, as described before, is conducted, but this time 277 background images were taken. All these images were again fed to each network and the confidence level for each image were summed together, and then divided with the total amount of images to find the average, which again were done for each network.

Looking at table 6.1 and Fig. 6.13 which has the mAP value on the y-axis and the 'step' number on the x axis. The 'step' is the current step during the training were the specific mAP value were measured, and the total amount of steps which are gone through during the training is specified with the previously stated 'epoch' command. When comparing the table and the graph, in Fig. 6.13, it can generally be stated that increasing the amount of images used for training, as well as using data augmentation, increases the overall mAP value of the network. There are however two small notes which are worthy of notice. Firstly, looking at the graph, then it can be seen that Weed3 generally has a higher mAP compared to Weed4, which speaks against the statement just mentioned that more images result in better performance. The reason for this is believed to be the fact that the images used in Weed3 were only taken from a specific angle of the plant hereby resulting in Weed3 being extremely good at detecting plants from this specific angle, but would perform very poorly as soon as the plant were for instance tilted, or not represented in the same way as the images. The other thing to notice is Weed7. Weed7 has a lower mAP value compared to Weed6 which does not make sense since Weed7 is just further training on Weed6. It is however thought that since the training data for Weed7 contained such a high ratio of background images, then the network could maybe not be tested thoroughly during the training. Weed7 were in fact better at detecting objects compared to Weed6 during the testing in the real world, which makes the mAP value of Weed7, on the graph, a little misleading.

Conclusion

In relation to the previous comparison it were found that the network which had the best performance, of all the presented networks, is Weed7. This conclusion is based on testing which were carried out in the lab, and the performance evaluation written in table 6.1 the 'Avg confidence' and 'Avg FP'. In this table it can be seen that Weed7 is capable of detecting with a higher average confidence, on the images used during the test. Also Weed7 has a general lower 'Avg Fp' which indicates that this network is better at avoiding detecting false positives. It makes sense that Weed7 has an overall better performance since it is based on Weed6, which is trained with the highest amount of images, and the Weed7 training session only added even more images. Also, the Weed7 network had a good match of images of the plant from various angles, as well as having a higher number of background images.

Table 6.1: Comparison table of different transfer learning runs

Session	Images	Post processed images	background images	Trained from	Avg confidence	Avg FP
Weed3	130	Not used	Not used	YOLOv5n	0.0348	0.0086
Weed4	207	828	Not used	YOLOv5n	0.0977	0.1205
Weed5	617	2470	5%	YOLOv5n	0.1407	0.0248
Weed6	587	3180	23.27%	YOLOv5n	0.1345	0.0025
Weed7	102	861	52.50%	Weed6	0.3838	0.0015

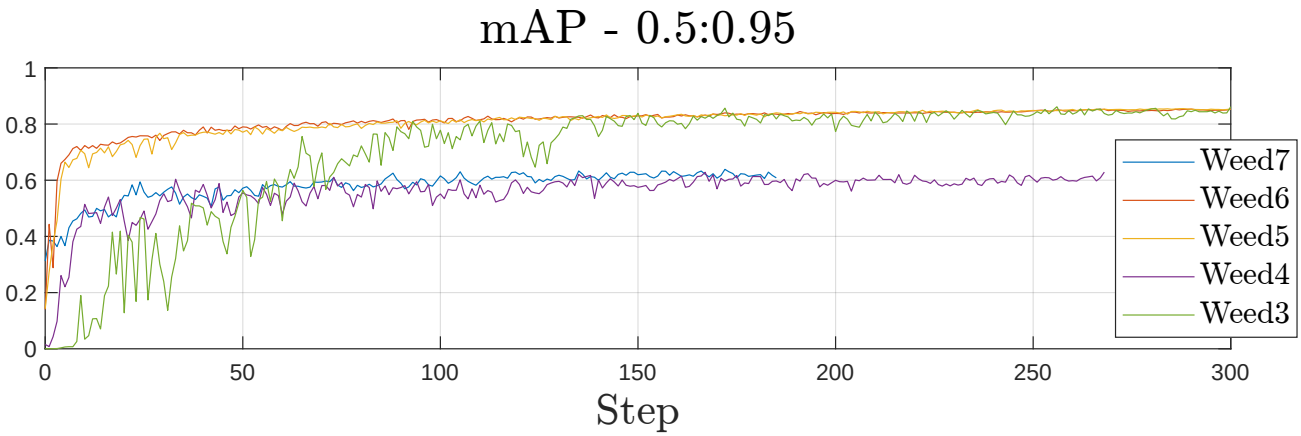


Figure 6.13: mAP comparison graph for the different training runs

6.3 Implementation of the neural network

The implementation of the final neural network, will be done in the file `Detection.py`. This module will contain everything related to the neural network, and will need to be imported in the `main.py`. On listing 6.5, the contents of `Detection.py` can be seen. The module will import `torch`, which is needed to use the model, and defines a class called `Network`. When instantiated, it takes the name as the model as argument, and assumes it is in the same directory as `Detection.py` itself. In the constructor the `model` object is defined, based on the file specified in `model_name`. When an image formatted as a matrix of RGB values from the camera, it can be passed to the `run_detection` function, which will perform the inference based on the model object, and returns a `Detections` object. This object is not useful for much as it is, which is where `get_weeds()` come into play. It takes one argument, `results` which needs to be a `Detections` object. By executing `result.pandas().xyxy[0]`, the `results` object will return a Pandas dataframe, containing all data about the detected objects in the image. This is stored in `objs`. On the next line, it extracts all the entries in `objs` that is labeled as weeds, and returns a dataframe, only containing the detected weeds, and stores this in `weeds`. The dataframe follows the format: `number xmin ymin xmax ymax confidence class_number name`. The network can only detect one type of weed so technically this step is not relevant, but will be needed in the case of a multi class network. `weeds` will contain all the weeds the network has detected, and as long as it has

detected at least one weed, it will return the bounding box of the first entry in the dataframe, along with its confidence. If no weed is detected, it returns -1 as bounding box coordinates, and 0 as confidence to indicate that nothing has been detected.
The final method, `get_middle` simply just returns the coordinate of the bounding box center, given the upper left and lower right corner coordinates. The center of the bounding box is where the system aims for, in order to move the manipulator towards the plant.

Listing 6.5: Content of `Detection.py`

```
import torch
class Network:
    def __init__(self, model_name):
        self.model = torch.hub.load('ultralytics/yolov5', 'custom', path=
            model_name, force_reload=True)
    def run_detection(self, image):
        return self.model(image)

    def get_weeds(self, result):
        objs = result.pandas().xyxy[0]
        weeds = objs.loc[objs['name'] == 'Weed']
        if(len(weeds) > 0):
            weed = weeds.iloc[0]
            return (int(weed.xmin), int(weed.ymin)), (int(weed.xmax), int(weed
                .ymax)), weed.confidence
        else:
            return (-1,-1), (-1, -1), 0

    def get_middle(self, p1, p2):
        x = (p2[0] - p1[0])/2 + p1[0]
        y = (p2[1] - p1[1])/2 + p1[1]
        return int(x),int(y)
```

Chapter 7

State estimation

As previously stated, one of the primary objectives of this project is to use the object detection algorithm (deep neural network/YOLOv5) to locate and identify a plant on an image, which is obtained from a camera, and use this information to control the manipulator. However, there are a few problems with this approach, that need to be addressed. When getting data from a sensor, which is a camera in this case, there is the risk of package/data loss, meaning the data/sensor reading is never received by the controller. This will have the effect on the system, that the controller will not receive updated data from the sensor, and it will have to wait until the next sample to get updated data. In the case that data has been received, it will be processed by the object detection algorithm, to identify and locate objects in the image. However, it will not always detect the object successfully, so the algorithm will receive an image, but it will not output a bounding box. With a well trained model, it will detect the object most of the time, but occasionally it will fail to detect the object. This again results in fact that the controller will not receive fresh data, and hence will not have any input to base the updated control signal on. Furthermore, the object detection algorithm runs quite slowly, which increases the sampling time of the system, and it will also introduce a delay.
To control the manipulator based on the camera and neural network input, it will be reasonable to assume that the manipulator can be controlled via. velocity commands to the end effector, in the x, y and z directions, along with rotation of the end effector. This is because, if it was position controlled, the function which moves the end effector to a position might be a blocking function meaning that the manipulator would not be able to execute any other action, before it has reached the desired position. Which makes tracking and visual servoing more difficult. By sending velocity commands to the manipulator, a command can set a speed and it will continue with this speed until it either receives an updated velocity command, or reaches the limit of how far the arm can reach. Furthermore, velocity control is also given as an example in visual servoing literature [27]. This means, that if at time t, the controller will have received an input from the object detection algorithm, it will calculate an error between the gripper and the plant, and based on this error, velocity commands will be sent to the manipulator, that will move the manipulator in a direction that will minimize the error. The manipulator will continue to move with this velocity until the next time the controller receives data, at $t + T$, where T is the period (assuming no loss, object detection is successful and no delay). If T is large, and the velocities sent to the end effector are also large, then the manipulator will have moved a big distance in the time between t and $t + T$, and it may occur that the error will then have increased at $t + T$, compared to the error at t. Increased error will give an increased control signal, which will make the arm move even faster and move an even longer distance. This shows that there is a risk of the system becoming unstable due to the large sampling period and fast speeds of the end effector. When the delays and data loss are introduced, the next sample will first arrive even later, or not at all, in some cases, and it will still be old data when it arrives, delayed by the time it takes to process the data. To identify the sampling time in the setup for this project, a small experiment was made. The

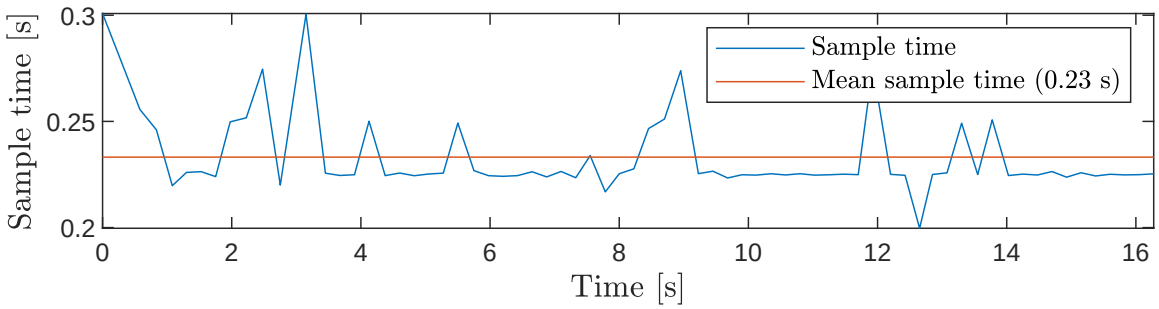


Figure 7.1: Results of experiment

system was configured to detect the plant using the camera and the YOLO network, calculate the error between the gripper and plant, and display the image, while moving around. These are the parts of the system that are assessed to take the longest time. The result can be seen on Fig. 7.1. The system was showed to have an average sampling time of 0.23 seconds. During this experiment, no detection loss from the algorithm was observed. Due to limitations of the manipulator it was difficult to get it to move much faster for a considerable time period, but if run at higher speeds, detection loss may be higher [58]. The delay in the system could also be determined from this experiment, but this parameter will not be used in the rest of the report so it is omitted in this section.

One solution to combat the before mentioned issue of moving the manipulator with random data losses and delays, is to make the controller and actuation very slow, i.e. low controller gains and saturation limits, causing low velocity commands to the manipulator, or adding a low pass filter, or apply a switching controller that will stop movement when something is not detected [58]. However, this will make the system slow, which is undesirable. Another approach is to implement an estimator, that will estimate the error between the manipulator gripper, and the plant it is moving towards. This way, in the case of data loss, the error will be estimated by a model, and a control signal will be calculated based on the estimated error. If it is possible to run the estimator and controller in a seperate thread than the object detection algorithm, the estimator should also be able to estimate the error in between the arrival of data from the camera and object detection algorithm, ideally increasing the speed of the actuation and control. Implementing an estimator like this, may also solve the problem of data loss, when the camera on the end effector gets too close to the plant, and fails to recognize the plant, so it will still be possible for the end effector to get sufficiently close to the plant, such that a tool can pick up the plant.
This outlines the motivation for implementing state estimation in this project, and this section will cover the modeling used for the estimator, along with the design and implementation of the actual estimator.

7.1 Model selection and identification

The objective of the modeling is to make a dynamic motion model of the manipulator, such that it can be used to design an estimator. The dynamic model of the manipulator is a very complex and nonlinear model, so since the scope of the project is mainly linear models, control and estimation,

the model to be developed will be a linear approximation of the manipulator. The model will be based on a state space model, where the states are the velocity and position vectors of the plant, relative to the manipulator, or in other words, the error vectors and velocity of the error vectors between the end effector and plant. The input to the model is the speed commands of the end effector in x, y and z directions respectively. By keeping it only to the translational velocities of the end effector, the model will stay relatively simple and linear, but it will also be at the expense of control performance, as this model does not take rotation of the end effector into account. The output/measurement of the model will simply be the error vector between the gripper and the plant. The model structure can be described as:

$$\begin{bmatrix} \ddot{\epsilon}_x \\ \ddot{\epsilon}_y \\ \ddot{\epsilon}_z \\ \dot{\epsilon}_x \\ \dot{\epsilon}_y \\ \dot{\epsilon}_z \end{bmatrix} = A \begin{bmatrix} \dot{\epsilon}_x \\ \dot{\epsilon}_y \\ \dot{\epsilon}_z \\ \epsilon_x \\ \epsilon_y \\ \epsilon_z \end{bmatrix} + B \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} \tag{7.1}$$

$$\begin{bmatrix} \epsilon_x \\ \epsilon_y \\ \epsilon_z \end{bmatrix} = C \begin{bmatrix} \dot{\epsilon}_x \\ \dot{\epsilon}_y \\ \dot{\epsilon}_z \\ \epsilon_x \\ \epsilon_y \\ \epsilon_z \end{bmatrix} \tag{7.2}$$

ϵ describes the vector components of the error between the manipulator and plant, and the accented dots describe the time derivatives of these states. v_x, v_y and v_z are the velocity commands sent to the end effector. The manipulator acts like a closed loop system, so the input speed commands are essentially reference values for an internal speed controller in the manipulator. The output vector is simply the components of the error vector between the gripper and the plant, ready to be used in the controller. By looking at the states, inputs and outputs, A will be a 6×6 matrix, B will be 6×3 and C will be 3×6.
It will be attempted to make a first order model of the system, with respect to the velocities, by estimating a first order transfer function with the velocity command in the x, y and z directions as inputs, and the error (ϵ) velocities as output (shown here for the x direction only but analogous to the other directions as well):

$$\frac{\dot{\epsilon}_x(s)}{V_x(s)} = \frac{K_x}{\tau_x s + 1} \tag{7.3}$$

To get the position vector instead of the velocity, an integrator is applied to eq. (7.3) (the sign of the integrator is negative, because when when a velocity is applied to the end effector in the direction of the plant, the error should decrease):

$$\frac{\epsilon_x(s)}{V_x(s)} = -\frac{K_x}{\tau_x s^2 + s} \tag{7.4}$$

The transfer function in eq. (7.4) can then be brought into time domain and stated as:

$$\ddot{\epsilon}_x = -\frac{1}{\tau_x}\dot{\epsilon}_x - \frac{K_x}{\tau_x}v_x \tag{7.5}$$

Using eq. (7.5) the A and B matrices can be defined as:

$$A = \begin{bmatrix} -\frac{1}{\tau_x} & 0 & 0 & 0 & 0 & 0 \\ 0 & -\frac{1}{\tau_x} & 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{\tau_x} & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \tag{7.6}$$

$$B = \begin{bmatrix} -\frac{K_x}{\tau_x} & 0 & 0 \\ 0 & -\frac{K_y}{\tau_y} & 0 \\ 0 & 0 & -\frac{K_z}{\tau_z} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{7.7}$$

$$C = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.8}$$

An experiment was done, where a step input was applied to the manipulator, to move the end effector in either the x, y or z direction, and the position and velocity was recorded using an OptiTrack system. The recorded data was filtered to get rid of high frequency noise. An input signal sequence was generated for the recorded data, and was then used to identify three first order models of the structure seen in eq. (7.3) using the System Identification Toolbox in Matlab the different parameters could be found, and are shown in table 7.1.

Table 7.1: Model parameters

	x	y	z
K	1.22	1.013	1.176
τ	0.3	0.16	0.16

The model comparison with the experiment data can be seen in Fig. 7.2. The figure shows that the model is a good approximation to the data.
In order for this model to be implemented on the Jetson, it needs to be discretized, such that it will be in the form of:

$$\begin{bmatrix} \epsilon_x[n+2] \\ \epsilon_y[n+2] \\ \epsilon_z[n+2] \\ \epsilon_x[n+1] \\ \epsilon_y[n+1] \\ \epsilon_z[n+1] \end{bmatrix} = A \begin{bmatrix} \epsilon_x[n+1] \\ \epsilon_y[n+1] \\ \epsilon_z[n+1] \\ \epsilon_x[n] \\ \epsilon_y[n] \\ \epsilon_z[n] \end{bmatrix} + B \begin{bmatrix} v_x[n] \\ v_y[n] \\ v_z[n] \end{bmatrix} \tag{7.9}$$

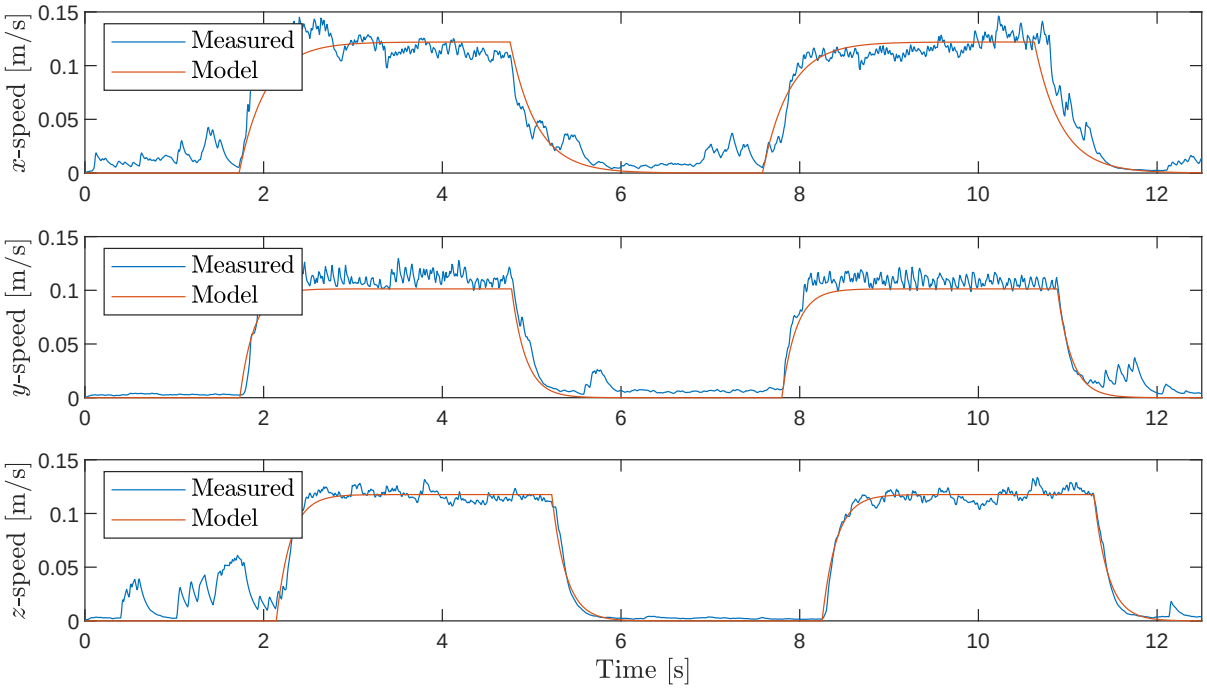


Figure 7.2: Model and experiment comparison

This can be done in Matlab using the `c2d` command, where the sampling time and method of discretization must be specified (the Tustin method has been chosen for this project). However, since the implementation of the system is done in Python, the `control` library[59], using the `control.c2d` command is used. More information will follow in the implementation section.

7.1.1 Benefits and limitations of chosen model

As mentioned, the model is a linear model so it will only be an approximation of the motion of the robot in its linear region. This means some types of motion, such as rotational velocities/motion of the end effector is not supported. When controlling the manipulator based on actual (not estimated) data, it should still be possible to control the end effector via rotational velocities of the end effector, it is just not something the model and thereby also estimator can support. The main benefits of having the model this way is that it is simple, and a good starting point for developing more complex nonlinear estimators etc.
When controlling the manipulator via velocities of the end effector, the manipulator will occasionally bring its joints into a configuration, where it cannot execute a given velocity command, and the manipulator will not move due to this. Thereby, when the model is given an input, but the manipulator does not move, the model/estimator will accumulate a lot of error and will not work. This is as much a control problem as an estimator problem, as the manipulator should bring itself into a configuration where it can move, if this situation occurs, but this is not considered in this project. A simple workaround could be to use the measured velocity of the end effector, to correct the velocities in the state vector.
If the plant is placed outside the workspace of the manipulator, it will start moving towards the

plant with the commanded velocities, but when it is stretched as far as it can, it physically cannot move any further, but with the velocity commands to the manipulator, the model will accumulate error as previously explained. It is however stated in the delimitation for the project that the plant will always be within the workspace of the robot, hereby avoiding this problem.
Based on the above discussion, it can be summed up that the assumption required for the estimator to work, is that it must operate in its *linear region*, meaning that it is assumed that the two above cases will not occur.

7.2 Estimator design

In conventional estimator designs, the model based estimator, and the actual system is given the same input, and the output from the model is subtracted from the output of the actual system, and fed back through a gain matrix, to prevent error due to different initial conditions in the model and actual system[60, p.535]. This method is most useful when either all states cannot be measured, or if an accurate measurement of the initial conditions of the system cannot be obtained, such that the output of a system is used to estimate the states. However, for this system, the velocity of the end effector $\dot{\boldsymbol{\epsilon}}$ can be easily measured, and as soon as a measurement is obtained, the error vector $\boldsymbol{\epsilon}$ can also be measured. Therefore, a slightly different approach is taken. When a measurement can be obtained (i.e. a point on the image, that corresponds to where the plant is in the image, along with a distance to the plant), the error vector can be calculated, and the end effector velocity can be measured by the sensors in the manipulator. These measurements/calculations are then used to update the states in the model continuously. If then suddenly a measurement is lost due to the neural network failing to detect the plant, or similar, the system will detect the data loss, and it will use the model based on the latest update of the states, based on the last time a measurement was taken, to estimate the error between the end effector and plant, and use this error to generate a control signal (velocity commands to the end effector). On Fig. 7.3, a simplified view of the estimator can be seen. For each iteration in the control loop, there are two cases. Either an object is detected/data is recieved, or it is not. In the case where data is recieved, the error between the end effector and the plant will be calculated using the methods described in chapter 5, and the controller will use the error to generate a control signal (velocity commands) to the end effector. When the error is calculated based on the received data, it will also update the states in the estimator, along with measurements of the end effector velocities. In the other case, where no data is received, or an object is not detected, the estimator will continue to estimate the error between end effector and the plant. This is done based on the states that were updated right before the data loss occurred and the velocity commands sent to the end effector.

7.3 Estimator implementation

The code implementation in this project is made modular, so each part of the system that needs to be implemented in code, has its own file. This also includes the code for the estimator, which will be written in a file called `ModelEstimator.py`. This file will contain a class, `ModelEstimator` , which when instantiated will define the model, discretize the model and initialize the states. There will be a main file, called `main.py` for the entire system, that ties all the different modules

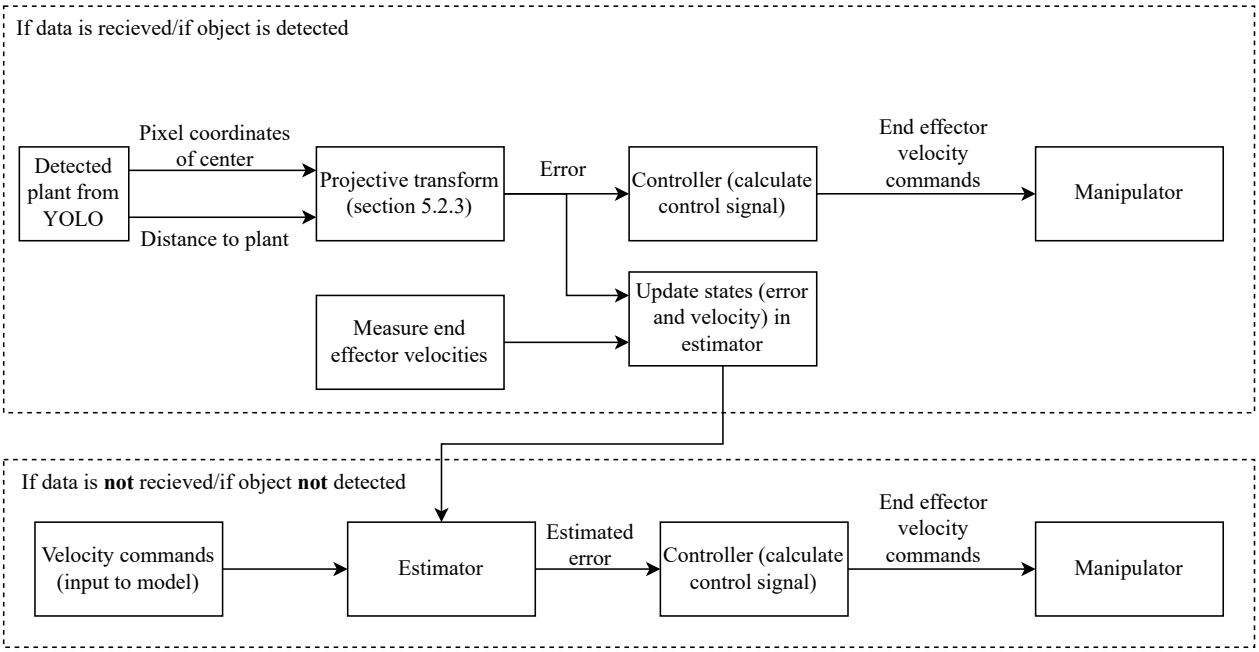


Figure 7.3: Block diagram of the estimator

together, and contains the final implementation. This section will describe the estimator class. The implementation of the estimator in the full control loop will follow in the next section. The `ModelEstimator` class can be seen in listing 7.1. The class takes one argument, which is the sampling time, used to discretize the model. In the constructor of the class, the states and next states of the model is defined. The model parameters and model matrices are defined (the matrices are not explicitly shown to save space, but they are numpy matrices of the ones seen in eq. (7.6), eq. (7.7), and eq. (7.8)). A state space model object is defined using `control.ss`, which is then discretized using `control.c2d`, with the specified sampling time. This will define the `A`, `B` and `C` matrices of the discrete system, which will be used for the system. Following this, the `update()` function is defined. It takes three arguments, `epsilon` and `epsilondot` which is the measured error vector and the error velocity respectively and `u` is the input velocity commands. It calculates the next error vector, based on the current error and the velocity times the sampling time. This will calculate the $\epsilon[n+1]$ value in the discrete state space model, and stores it in `next_epsilon`. It concatenates the `next_epsilon` and `epsilon` arguments into one column vector, that corresponds to the current states of the system/estimator. It also updates the next states in the state space model, using the model, the state vector that was just created, and the `u` argument. The second function, `estimate`, takes one argument, `u`, which are the velocity commands sent to the end effector. This function uses the model and the input `u`, to estimate the error, which can then be used to generate a control signal to the manipulator.

Listing 7.1: The `ModelEstimator` class

```
1 import numpy as np
2 import control
3 class ModelEstimator:
```

```
4       def __init__(self, Ts):
5           self.states = np.array([[0],[0],[0],[0],[0],[0]])
6           self.next_states = np.array([[0],[0],[0],[0],[0],[0]])
7           self.Ts = Ts
8
9           Kx, Ky, Kz = 1.22, 1.013,1.176
10          tau_x, tau_y, tau_z= 0.3, 0.16, 0.16
11
12          A = np.matrix( ... )
13          B = np.matrix( ... )
14          C = np.matrix( ... )
15          D = np.matrix( ... )
16
17          H = control.ss(A,B,C,D)
18          Hd = control.c2d(H, self.Ts, 'tustin')
19          self.A, self.B, self.C = Hd.A, Hd.B, Hd.C
20
21      def update(self, epsilon, epsilondot, u):
22          next_epsilon = epsilondot * self.Ts + epsilon
23          self.states = np.concatenate([next_epsilon, epsilon])
24          self.next_states = (self.A @ self.states) + (self.B @ u)
25
26      def estimate(self, u, Ts):
27          self.states = self.next_states
28          self.next_states = (self.A @ self.states) + (self.B @ u)
29          return = self.C @ self.states
```

Chapter 8

Control

The purpose of the controller and the control system, is to move the arm in such a way, that the distance between the gripper and the plant is minimized, and eventually becoming zero meaning the end effector is positioned on the plant. The control system will also tie all the previous topics of the project together, in order to complete the objective of this project which is to move the end effector on to the plant. An intuitive way to configure the control system, is that the system takes an input from the camera (chapter 5), feed it through the neural network to get an image position of the plant, chapter 6, and then it will use an inverse projection together with a distance (chapter 5), so it can calculate an error vector between the end effector and the plant that was detected. It can then use this error vector, to calculate velocity commands to send to the end effector (chapter 4), and it will start to move towards the plant, minimizing the error vector between the end effector and plant. As the arm moves, the error vector between end effector and the plant will change, and the system will update the error and control signals based on the new error.

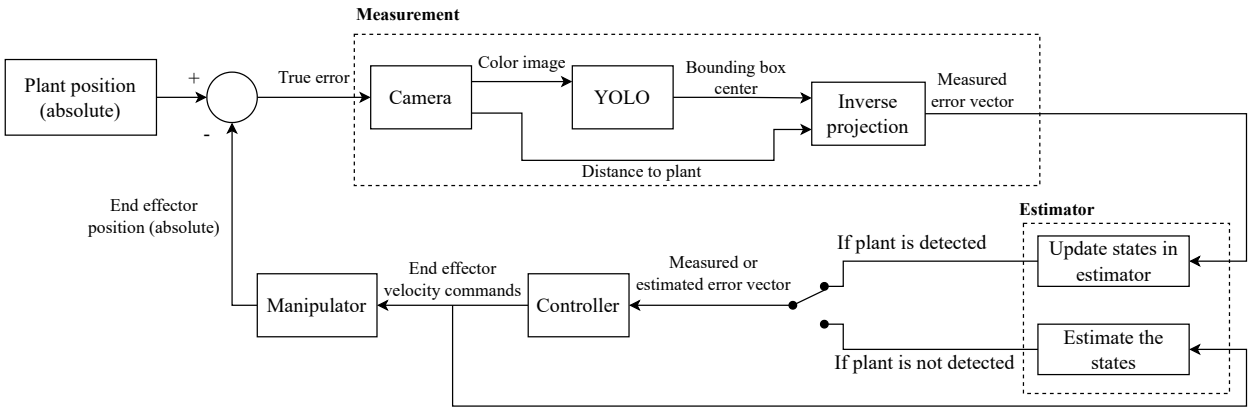


Figure 8.1: General control system block diagram

Often in control systems, there is a reference value, and a measured value, which get compared to produce an error. This error is the input signal to a controller, and the controller controls an actuator, that changes the measured value, in order to minimize the error signal, and make the measured value the same as the reference value. However, for this system, it is most convenient to directly measure the error, instead of a value to compare with a reference. The general control system can be seen on Fig. 8.1. The general problem at hand for the system, is to minimize the error between the plant and the end effector. One way to *calculate* the error is to subtract the end effector position from the plant position (where the plant position can be seen as the reference), with respect to some constant point in space, e.g. the manipulator base. However, in this case it will be more convenient to instead *measure* the error between the end effector and the plant, and use this for the controller input. On Fig. 8.1, the absolute position of the plant and the end effector

are never explicitly known, so they should be seen as abstract values on the block diagram. It will still hold, that if they are subtracted, they will give the error, which is marked as *true error* on the block diagram. The true error is converted to a measured error, by taking a picture with the camera, feeding the image through the neural network which will give a location of the plant on the image. The location on the image is combined with the distance to the plant (obtained via. the LiDAR in the camera) and used in the inverse projection, to get a vector between the gripper and plant, as described in chapter 5. This is the *measured error vector*, which is then passed to the estimator. If the neural network has been successful in detecting a plant on the image, it will simply just update the estimator, and if the neural network fails to detect the plant, the estimator will estimate the error between end effector and plant, as explained in chapter 7, until a successful measurement/detection is obtained. The measured (or estimated) error, will then be passed to a controller, that will, based on the error vector between gripper and plant, generate control signals which are velocity commands to the end effector. The end effector will then move to a new position in space, which will result in a new error to be measured. This control loop will then continue, until the error is small enough, that the end effector is in a position to pick up the plant.

The next sections in this chapter will cover, how the error vector is calculated both in theory and in practice, how the controller is designed and how it converts the error vector into velocity commands that can move the end effector, and it will also cover the implementation of the control system, including the estimator from chapter 7.

8.1 Measuring the error

This section will cover, how the error is measured, in order to produce some data that can be used to create the control signals. The first step is to obtain the image from the camera. Provided an object of the `Camera` class, as defined in section 5.1, listing 5.1 and listing 5.2, the member function `get_frames()` on the instantiated `Camera` object, will return the a depth image and a color image, of what the camera sees. When a color image has been obtained, it can be passed to the neural network, using the `run_detection(color_image)`, while passing the color image to the method, provided that an object of the `Network` class has been instantiated. `run_detection (color_image)` will return a pandas dataframe, which must be passed to `get_weeds(result)` (also a member function of the `Network` class). `get_weeds(result)` will return two coordinates on the color image, corresponding to the bounding box corners, along with a confidence of the detected object. The confidence can be used to filter false positives of low confidence. Finally, to get the center of the bounding box, which is the piece of data that will be used to aim the gripper at, is obtained by `get_middle(p1, p2)` (also a member function of the `Network` class), which takes the corner coordinates of the bounding box and returns the center coordinate. Now an image coordinate has been obtained, so eq. (5.14) can be used:

$$\tilde{\boldsymbol{\epsilon}}_{\mathbf{0}} = M^{+}\tilde{\boldsymbol{u}} \tag{8.1}$$

where $\tilde{\boldsymbol{\epsilon}}_{\mathbf{0}}$ is the 3D vector in meters between the gripper and the plant (the error vector) in homogenous coordinates, $\tilde{\boldsymbol{u}}$ is the image coordinate in pixels corresponding to the center of the bounding box, also in homogenous coordinates, M is defined as:

$$\boldsymbol{M} = \boldsymbol{K} \begin{bmatrix} \boldsymbol{R}_g^c & -\boldsymbol{R}_g^c \boldsymbol{t} \end{bmatrix} \tag{8.2}$$

R_g^c is the rotation matrix between the camera and gripper coordinate frame and $\boldsymbol{t}$ is the translation between the camera center and the gripper center defined as:

$$R_g^c = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{8.3}$$

$$\boldsymbol{t} = \begin{bmatrix} 0 \\ 0.083 \\ -0.102 \end{bmatrix} \tag{8.4}$$

The above equations and definitions mean, that the homogenous image coordinate is transformed to a vector in 3D, that points in the direction of the center of the plant, and originating from the camera. The $\boldsymbol{R}_g^c$ matrix and the $\boldsymbol{t}$ vector will then transform the 3D vector, to originate from the gripper, and be in the gripper reference frame. Also recall from chapter 5, that the $\tilde{\epsilon}_0$ obtained from the above equation, will only be a directional vector, and the distance is also needed in order to calculate the correct error vector. The implementation of the projection and transformations stated above, is described in listing 5.3 and the corresponding section.
An example of how the error is measured practically, is shown in listing 8.1, which is a combination of the previously explained modules and classes. The final error between the gripper and plant is given by `epsilon` on the final line, which is a numpy vector.

Listing 8.1: Short example of error measurement

```
# importing modules
import Detection
import Projection
import IntelCamera

width, height = 640, 480 # image dimensions

cam = IntelCamera.Camera(width, height) #making camera object
net = Detection.Network() #making YOLO net object
proj = Projection.Projection(cam, 0.083, -0.0102, 0, width,height) #
    projectionobject

d_frame, c_img = cam.get_frames() #get color image
result = net.run_detection(c_img) #run detection
p1, p2, conf = net.get_weeds(result) #weed bounding box
center = net.get_middle(p1,p2) # get middle of bounding box
d = cam.get_dist(d_frame, center[0], center[1]) #distance to plant
epsilon = proj.get3d_point(center, d) #error vector
```

8.2 Controller design

Now that a method is used to calculate the error vector between the gripper and plant, a controller must be developed to minimize the error, and bring the gripper close to the plant so the plant can be grabbed. As explained previously, the gripper will be controlled via end effector velocities in the x, y and z directions of the end effector reference frame, based on the error. A simple P-controller is proposed as the control method, which is due to several reasons. It is simple but powerful, controlling the end effector via velocities resembles an integrator, so ideally it should not have steady state error, and it is also proposed in literature as a reasonable control law [27, p.35]. This means, that the controller block in Fig. 8.1 is essentially just a gain matrix.

When controlling end effector velocities, there are 6 different speeds that can be controlled, linear velocities in the x, y and z-directions, along with angular velocities around the three axes. The most crucial to control are the three linear velocities, as they move the end effector in space. If an angular velocity is applied around the x-axis of the end effector, it will pitch up and down, if an angular velocity is applied at the end effector y-axis, it will start to yaw, and with the z-axis it will roll. The angular control of the end effector mostly comes into play, when it is important that the gripper has a certain pose to approach the plant with. In this project, the plant is assumed to stand upright (perpendicular to the ground), and the robot base is also assumed to be perpendicular to the ground, the gripper should always approach the plant in the same way. This also means that the control of the z-axis can be discarded, as it will not be necessary to roll the gripper to grab the plant. Applying an angular velocity to the x-axis of the gripper, can be used to control the vertical angle of attack to the plant. This can also be discarded, if the gripper is configured to have the same vertical angle at all times, but e.g. if it is tilted at the point where it detects the plant, and it needs to be level wrt. the ground to grab the plant, it will need to apply an angular velocity to level off the end effector. It also needs to do this at a velocity, such that the plant is still kept in the view of the camera. Controlling the angular velocity of the y-axis of the gripper is not crucial, as it does not matter where the plant is grabbed in the horizontal plane, as the plant can be approximated to be symmetric about its vertical axis/the stem. However, applying a velocity around the y-axis of the gripper, will make sure that the approach vector is quickly aligned with the direction of motion, and makes sure that the gripper will approach the plant, such that the plant will enter between the fingers of the gripper, and not bump into the fingers of the gripper, which could happen with only translational control. By also enabling control of the angular velocity around the y-axis, it should also be possible to reach plants that are closer to the edge of the reachable workspace, as it is not restricted by keeping the orientation of the end effector in a special configuration. This will although come at an expense of slightly more complex control, as the angular velocity should ideally be controlled by an angle between the approach vector of the end effector and the z-component of the error vector, which will need to be calculated. By including the angular velocities, the model will become nonlinear, and thereby also more complex (also the reason why the estimator will not consider the angular velocities based on the plant position).
In addition to the gains, the controller can also utilize a saturation of the velocities that are sent to the manipulator. This is so that the end effector will not move too fast, causing violent movement and rapid acceleration of the manipulator, which is not desired. Furthermore, the sampling rate of the system is quite low, so if the end effector moves fast, it will move a long distance in between

samples, which it may need to correct for when it gets the next sample, because it has overshot its reference. The saturation boundaries, and the proportional gain for that matter, must be low enough so that the system will not become unstable and that unnecessary jitter in the manipulator and mechanical construction is avoided, but also high enough that good performance is still ensured [27, p.35].

8.2.1 Controller gains

This subsection will concern, how the controller gains are determined. As mentioned, the controller will be a gain matrix, in the place of the controller block on Fig. 8.1.

Linear velocity control

Even though it is not considered in this project, in future iterations of the systems, the base of the manipulator will be mounted on a small rover, that will move while picking up the weeds, i.e. the base will not be stationary relative to the weeds, when weeds are being collected. In order for the robot to be able to pick the weeds up, before the rover has passed the weeds, a maximum time requirement must be set for the weed to be collected, before the rover has driven too far for the manipulator to pick up the weeds, and still allow enough time to collect other weeds that are nearby. If it is assumed that the manipulator base is 30 cm above the ground, and the fact that the maximum reach of the manipulator has a radius of 760 mm [32], then a weed at one of the furthest points away from the manipulator base, that is still within the reachable workspace, will be 300 mm below the base origin, and 698.3 mm in front of it. Assuming that the model that was developed in chapter 7 is a decent enough approximation in this case, simulations show that a proportional gain of 1 for the linear speeds, the gripper will be within 1 cm of the plant within 2 seconds. This is assessed to be a reasonable performance, so the proportional gain for the linear velocity control will be 1. However, 2 seconds is a quite fast response, so for parts of the testing, a saturation will be implemented, that limits the speed in the three directions to 5 cm per second, so it can be more clearly observed how the system performs, especially wrt. the estimator. It should also be noted that the numerical values in this analysis are quite arbitrary, but merely serves as a reasoning for the particular proportional gain, and a description how it can be related to some requirements that might be set to the system in the future.

Angular velocity control

For controlling the angular velocities of the gripper, only the angular velocity around the y-axis (yaw) will be considered. There is no point in applying a velocity around the z-axis (roll) if the gripper is in the right orientation to begin with, and it is assessed to be too complicated at this point in the project, to implement a velocity around the x-axis (pitch). As mentioned before, it will need to rotate with a velocity that keeps the plant in view at all times, while leveling off at the same time, which sets further requirements to the control. However, it is expected to not have any large implications on performance, as the gripper should be able to grab the plant regardless of the pitch of the gripper. The controller must compute the angle between the error vector and the z-axis, to get something that the controller can act on. The z-axis direction of the gripper is

given by:

$$z_g = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{8.5}$$

The angle that is needed, is the angle between the gripper z-axis and the projection of the error vector to the xz-plane, with respect to the gripper, and this angle can be determined to be:

$$\theta = \text{sgn}(\epsilon_x) \cos^{-1}\left(\frac{z_g \cdot \epsilon}{|z_g||\epsilon|}\right) = \text{sgn}(\epsilon_x) \cos^{-1}\left(\frac{\epsilon_z}{\sqrt{\epsilon_x^2 + \epsilon_z^2}}\right) \tag{8.6}$$

An angle between two vectors are given by the dot product of the two vectors, divided by the product of their lengths, which is also what eq. (8.6) is based on, but it does not provide information about the direction of the angle, whether it is to the right or left of the gripper center. Therefore, it is multiplied by the sign of ϵ_x, so that when the weed is to the left of the center, θ is negative, and if it is to the right, θ is positive.
A proportional gain must be applied to θ, to generate the control signal for the angular velocity around the y-axis. There are no specific requirements to this other than of course it should not make the performance of the system worse, so the controller gain for the angular velocity. A dynamic model for the angular velocity will not be set up, but when giving an angular velocity command, the dynamics are quite fast as with the linear velocities, so it will be assumed that the commanded velocities will be reached fast, after the command is given. The horizontal field of view of the camera is 43 degrees [25], meaning that the magnitude of the largest error is 21.5 degrees. It is assumed that the angular movement can be modeled as an integrator, where the input is angular velocity, and the output is angular position. It can be proven by simulation, that if feedback is applied with a gain of 1, and the integrator has an initial condition set to 21.5 degrees, the error angle will be less than three degrees within 2 seconds, which is assessed to be sufficient.

8.3 Controller implementation

The code implementation in this project is made modular, so each part of the system that needs to be implemented in code, has its own file. This also includes the code for the controller, which will be written in a file called `Controller.py`. This file will contain a class, `Controller`, which when instantiated will take the proportional gain values, and store them in the object, and contain a member function to compute the control signals to the manipulator. There will be a main file, called `main.py` for the entire system, that ties all the different modules together, and contains the final implementation. On listing 8.2, the code for the `Controller.py` file can be seen.

Listing 8.2: `controller.py` module

```
1 from custom_utils import saturation
2 import numpy as np
3 import math
4
5 class Controller:
```

```
6     def __init__(self, Kp_lin, Kp_ang, max_ang_vel, max_lin_vel):
7         self.Kp_lin = Kp_lin
8         self.Kp_ang = Kp_ang
9         self.max_ang_vel = max_ang_vel
10        self.max_lin_vel = max_lin_vel
11
12    def get_control(self, epsilon):
13        lin_speed_x = saturation(self.Kp_lin*float(epsilon[0]), self.
              max_lin_vel, -self.max_lin_vel)
14        lin_speed_y = saturation(self.Kp_lin*float(epsilon[1]), self.
              max_lin_vel, -self.max_lin_vel)
15        lin_speed_z = saturation(self.Kp_lin*float(epsilon[2]), self.
              max_lin_vel, -self.max_lin_vel)
16        y_angle = np.arccos(float(epsilon[2])/(math.sqrt(float(epsilon[0])**2
              + float(epsilon[2])**2)))
17        y_angle = math.degrees(math.copysign(y_angle, float(epsilon[0])))
18        ang_speed_y = saturation(self.Kp_ang*y_angle), self.max_ang_vel, -self
              .max_ang_vel)
19
20        return lin_speed_x, lin_speed_y, lin_speed_z, ang_speed_y
```

The class `Controller` takes 4 arguments, the proportional gain for the linear and angular velocities respectively, along with maximum linear and angular velocities, to limit the maximally outputted value. All of these arguments are stored in the object. There is only one member function in the class, `get_control()`. It takes the error vector `epsilon` as argument. The three linear speeds in respectively in the x, y and z direction, are computed by multiplying the component of `epsilon` in the respective directions with the proportional gain, and put into the `saturation()` function, which essentially ensures that the control value is between `-max_lin_vel` and `max_lin_vel`. The `y_angle` variable is the computed angle between the z-axis and the error projected to the xz plane, given by eq. (8.6). Finally, all five control signals are returned to be used in the main code.

8.4 Full implementation

This section will be describing the final implementation of the system, and will conclude the development part of the project. Throughout the report, several different parts have been explained, along with the respective implementations of the project. As mentioned several times in the report, the implementation is made modular, so all the different parts of the system has their own module and class, to keep the code in the main code file organized. The structure of the code can be seen in Fig. 3.3. The full implementation of the code can be seen and downloaded from the project GitHub page [37].
On Fig. 8.2, the flow diagram of the final implementation of the system can be seen. The first step is to import the necessary modules, so all the above mentioned modules are imported, along with the standard python libraries, `cv2`, `numpy` and `time`. All the variables are defined and initialized, and among these are e.g. the image dimensions, controller gains, confidence threshold and the sampling time of the system, which are used different places in the code or to instantiate objects. Two variables are worthy of a more thorough explanation, `started` and `detected`. Both of these

variables are flags, and initialized as `False`. `started` will be set to true when the system is ready to start the control sequence, and `detected` will be set to `True` when an object has been detected. Their meaning in practice will become clear shortly.
The code will then instantiate the different objects for the camera, neural network, estimator, controller and projection, as described above. The program then connects to the arm, and instantiates the arm object, moves the arm to the home position and opens the gripper completely. A depth and color image is then obtained from the camera. The system then checks if `started` is set to `True`. The reason for having `started` is that the first image takes a long time to be displayed by the software, and this is not good for control. Therefore, `started` will be set to `True` as soon as the first image has been displayed, so that the system is ready to operate. If `started` is `False` it will keep waiting until the system is ready. Once it is set to `True`, it will run the object detection on the newly received image. It then checks that a plant is detected, and that the confidence is above the previously defined confidence threshold, and if it is, it will calculate the center of the bounding box, measure the distance to the plant, get the velocities to update the estimator, and set `detected` to `True`. It will then compute the control signals, send the control signals and display the image with the bounding box. If the button "q" is pressed on the keyboard, the program will stop and exit. In the case that a plant is not detected, it will check if `detected` is set to `True`, and if it is, it means that a plant has been discovered recently, meaning that the reason for a plant not being detected is detection loss, so the plant must still be there but just not detected. This calls for the estimator to be used, so it will estimate the error vector, and as long as the length of the error vector is more than 1 millimeter, it will display the estimated position of the plant on the image, and compute and send control signals based on the estimated error. If the error is less than 1 millimeter, it means that the gripper is in the end position, and it should grab the plant. At this point, the control loop stops, and the gripper grabs the plant, pulls it up vertically, and goes back to the home position, and the program stops and exits after this.

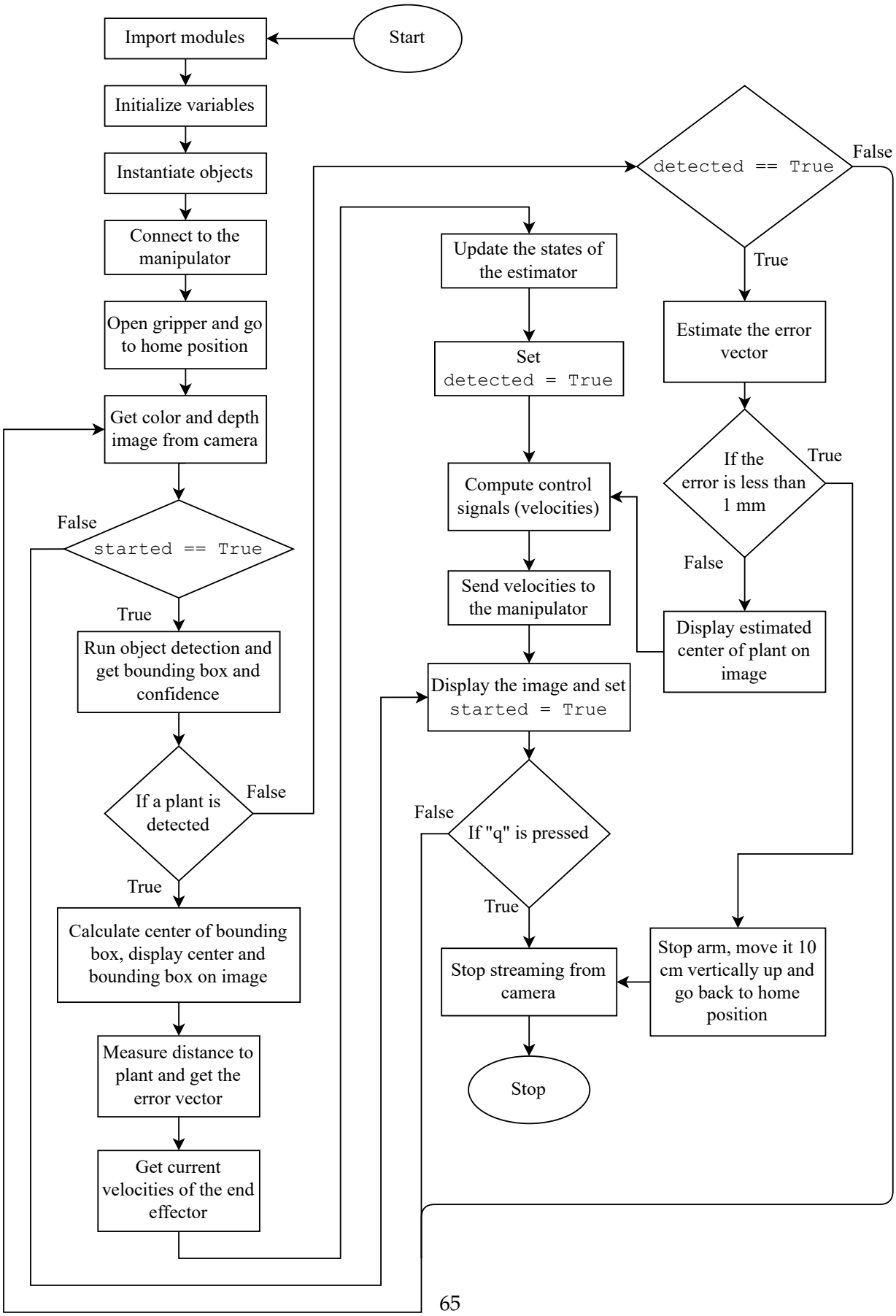


Figure 8.2: Flowchart of the final system

Chapter 9

Testing

9.1 General function and performance

This test serves as a general test of the system, with regards to verifying if it works as intended, and how well it performs under normal conditions. The plastic plant was placed within the workspace of the robot, and slightly below the base, to simulate the arm being on the rover, and the plant being in the ground. It is also made sure that the camera can spot the plant when it starts up. The system was then powered on, and the gripper started moving towards the plant, while the time, gripper position and orientation, error between gripper and plant, and confidence level of the plant was logged. The test setup can be seen in Fig. 9.1.

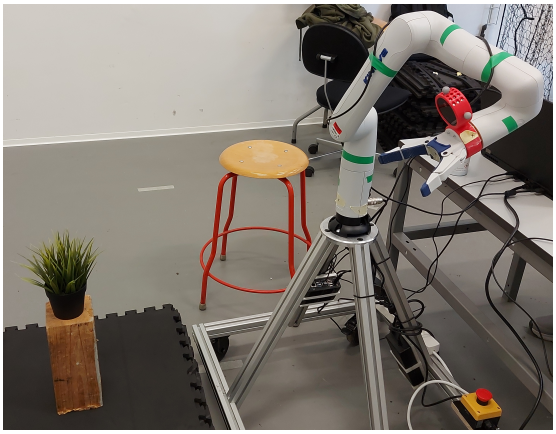


Figure 9.1: One of the testing setups

From this test it can be shown that the robotic manipulator is capable of moving towards the plant as well as detecting the plant. The detection is done through the Weed7 neural network which was made in chapter 6, also the complete system is running on the Nvidia Jetson Nano, and the camera used for taking the images is the L515 Intel camera. Also the manipulator is controlled through the python API.
It can be concluded based on this first test that requirement 1, 2 and 3 is met since the system is capable of doing basic tasks as well as running on the previous specified hardware.

To get a better view of how the system performs, parameters such as the gripper position and the error was logged during the test, to see how close the gripper actually gets to the plant, and how fast it does it. On Fig. 9.2, the gripper position over time can be seen, in the x, y and z direction (relative to the base). The position of the plant is also displayed on the plot, as it is at a stationary and known position relative to the manipulator. On the colored lines, the dashed lines indicates that the estimator is working. On the figure, it can be seen that the motion of the gripper reaches steady state after around 3 seconds, which is similar to what was obtained in the simulation in

section 8.2. In both the x and y directions, the gripper position is within 2 cm of the plant center, but the z-position is 6 cm off. Overall, it can be observed, that the gripper is able to move itself close to the plant, so it is capable of grabbing it, but there is a small steady state error, between the steady state position of the gripper, and the plant position. To investigate this, the measured error and the true error are plotted on Fig. 9.3.

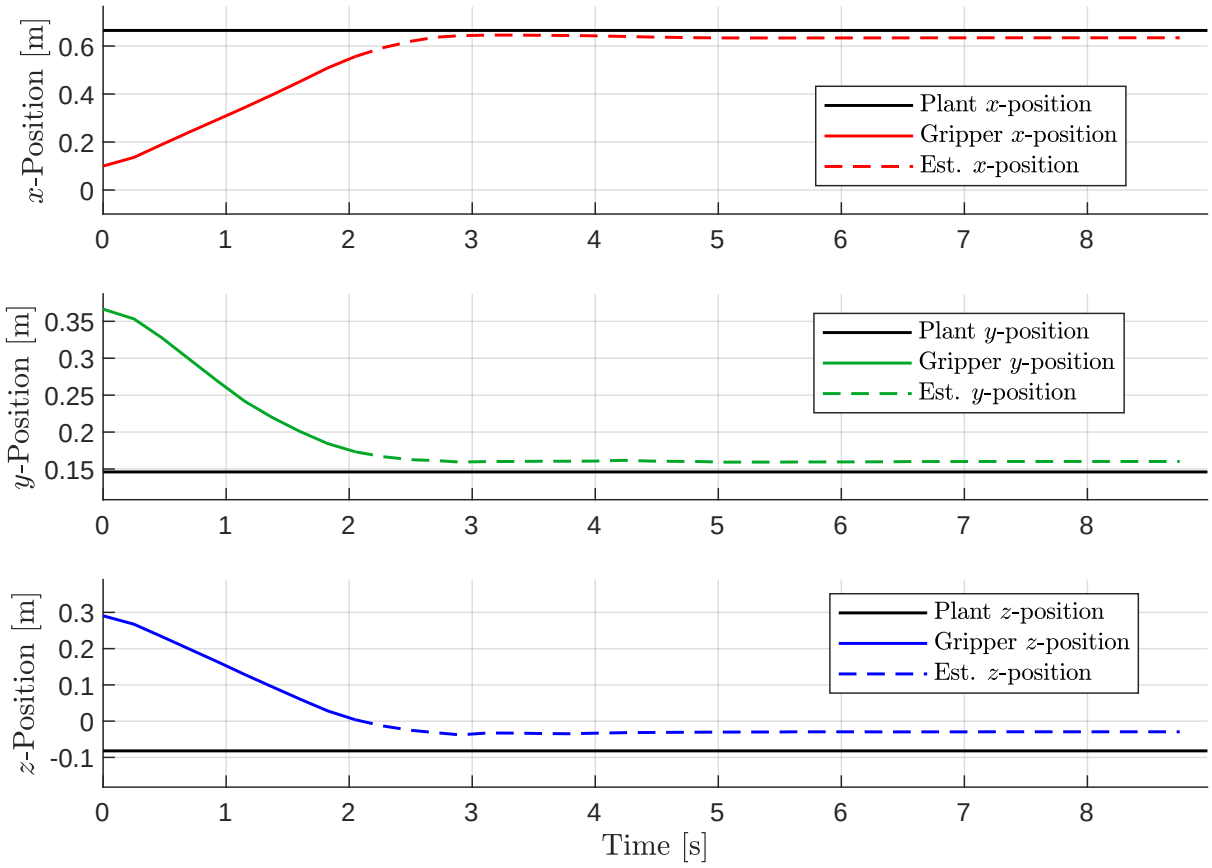


Figure 9.2: Plot of gripper position and plant during general test. Dashed line is when the estimator is working

On Fig. 9.3, the measured error and the true error can be seen. Both of these errors, are errors between the gripper and the plant, where the control signals are based on the measured error, and ideally the measured error should be equal to the true error. To describe the difference between true error and measured error, see Fig. 8.1. In short, the measured error uses the camera input to detect where the plant is, along with a distance reading to the plant and the projective transformation. The true error is calculated based on the collected data, namely the gripper position subtracted from plant position (the plant position was measured before the experiment was carried out). Ideally, the two errors should be the same or at least very similar. It can be observed, that all the measured errors go to zero, which means the controller is working as it should. However, the true errors do not, and it is clear to see that there is some deviance between the true and measured values. It indicates, that the error is not measured accurately, but if the error measurement could be done more accurately, the system would perform better, and the steady state errors on Fig. 9.2 would be smaller.

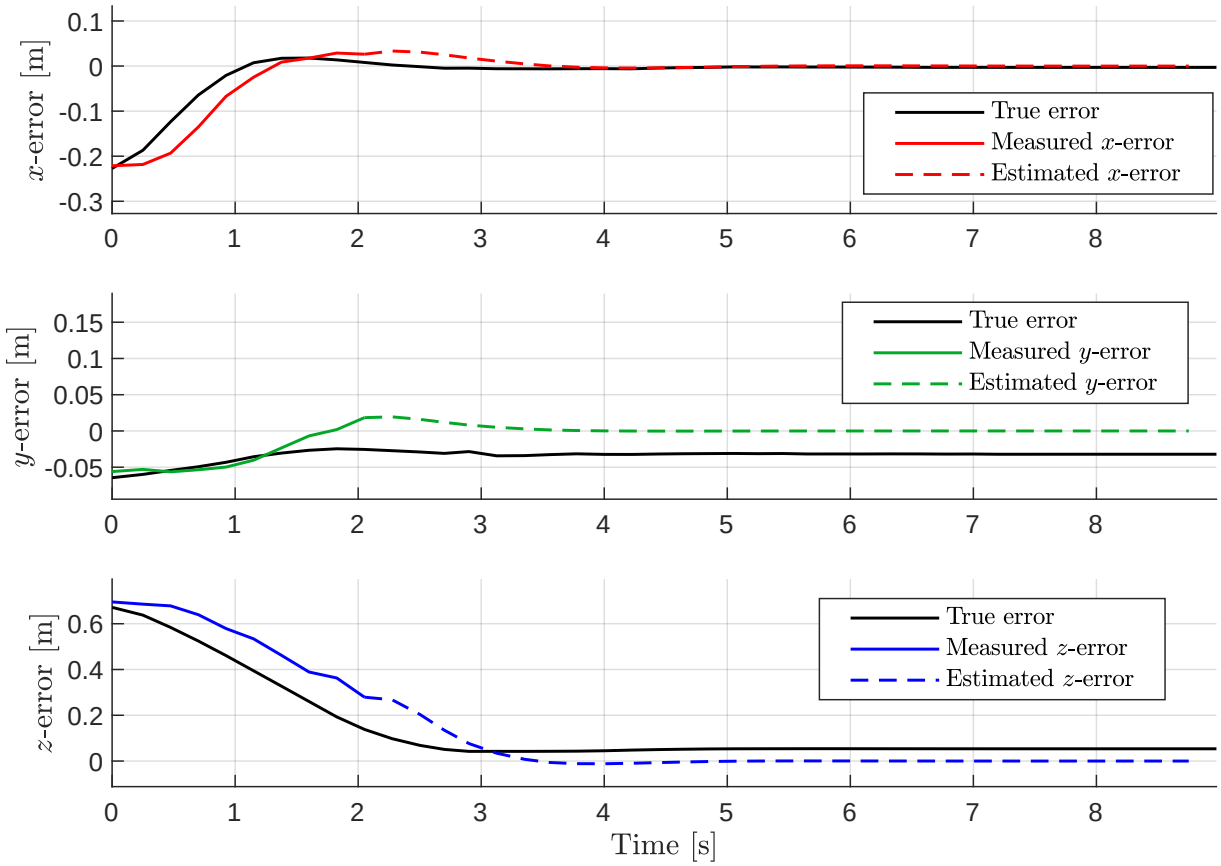


Figure 9.3: Plot of error vector between gripper and plant during general test. Dashed line is when the estimator is working. True error is given by the end effector position (measured) subtracted from the plant position, relative to the base

When the gripper/camera gets too close to the plant, the plant cannot be detected, and the system treats this as detection loss, and will let the estimator work based on the last known error. This occurs at around 2 seconds, and from there on and to the end of the test, it can be seen that the estimator brings the gripper the final way to the plant. Since the estimator is responsible for the final part of the movement of the gripper, the steady state problem could also be related to the estimator, if it thinks the arm moves faster than it does, and thereby decrease the error faster than it actually does. During the testing, it was also observed that the center of the plant, that was detected using the YOLO network (Weed7), moved as the camera got closer, because as it gets closer it can still see and detect the plant, but only partially. It draws a bounding box around what it can see of the plant, but this causes the center of the bounding box to move. The error vector is measured based on the center of the bounding box, so if the center of the bounding box is not correct, the error vector will not be as well. This phenomenon is illustrated on Fig. 9.4, where on the left, the plant is further away from the camera, and the center is fairly true to the center of the plant, but as soon as the plant gets close, it is still detected by the network, but the center is not true to the center of the plant, so the gripper will move in the wrong direction. In this case the LiDAR may also measure a distance to the wall behind the plant, because it sees

through the "leaves", making the distance measurement wrong as well, and making the gripper move in the wrong direction.

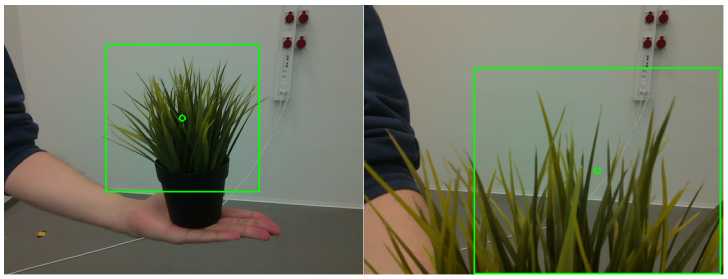


Figure 9.4: Plant far away from camera (left) and plant close to the camera (right)

This test was repeated with the gripper in different intial positions, and with the plant in different positions around in the workspace of the arm. The results shown in Fig. 9.2 and Fig. 9.3 shows an average performance, but during other tests the steady state error between the gripper and plant was larger, and sometimes the distance measurement was faulty due to measurement error, which then made the measured error vector all wrong, and the arm moved in a wrong way relative to the plant. It is difficult to give a fair number of how often it succeeds and how often it does not, as the system performs better in some conditions (wrt. plant position and initial position of the gripper) than others. But as mentioned, the data shown is a quite average performance of the system, which shows that the system works well, but with some imperfections, that need to be analyzed and improved.

9.2 Performance during detection loss

A similar test as in the previous section was made again, but here the camera was covered with a hand, as the arm moved closer to the plant, to simulate detection loss or frame dropouts. The idea is that when data is lost, the estimator will continue to generate a control signal to the robot, based on the last known error vector between the gripper and plant. The controller for the gripper was configured to move with maximally 5 cm per second, in order to have time enough to block the camera a few times, while it approached the plant. The same data as the previous test was collected, and a plot for the position and error over time was made. On both plots, the sections with a dashed line indicates that detection loss has occurred, and the estimator is generating the error vector between gripper and plant. Firstly, on Fig. 9.5, the position of the gripper over time can be seen, compared to the position of the plant. The plot shows that detection loss occurs 5 times, including when the camera is too close to the plant to detect it. It also shows that the gripper gets close to the plant, with some steady state error, as shown in the previous test. However, it can be seen that when detection loss occurs, the gripper moves with roughly the same speed/slope as it did before the detection loss occurred, until it can detect the plant again. This means, that the arm can move closer to the plant even though detection loss occurs. It was only tested with detection losses of short duration, as it is assumed that when they occur, it will either only be momentary, or because the camera is too close to the plant, so it will only move a small distance in either case. It is likely, that if it did not detect the plant again after it was lost the first time, it would not get so close to the plant in the end.

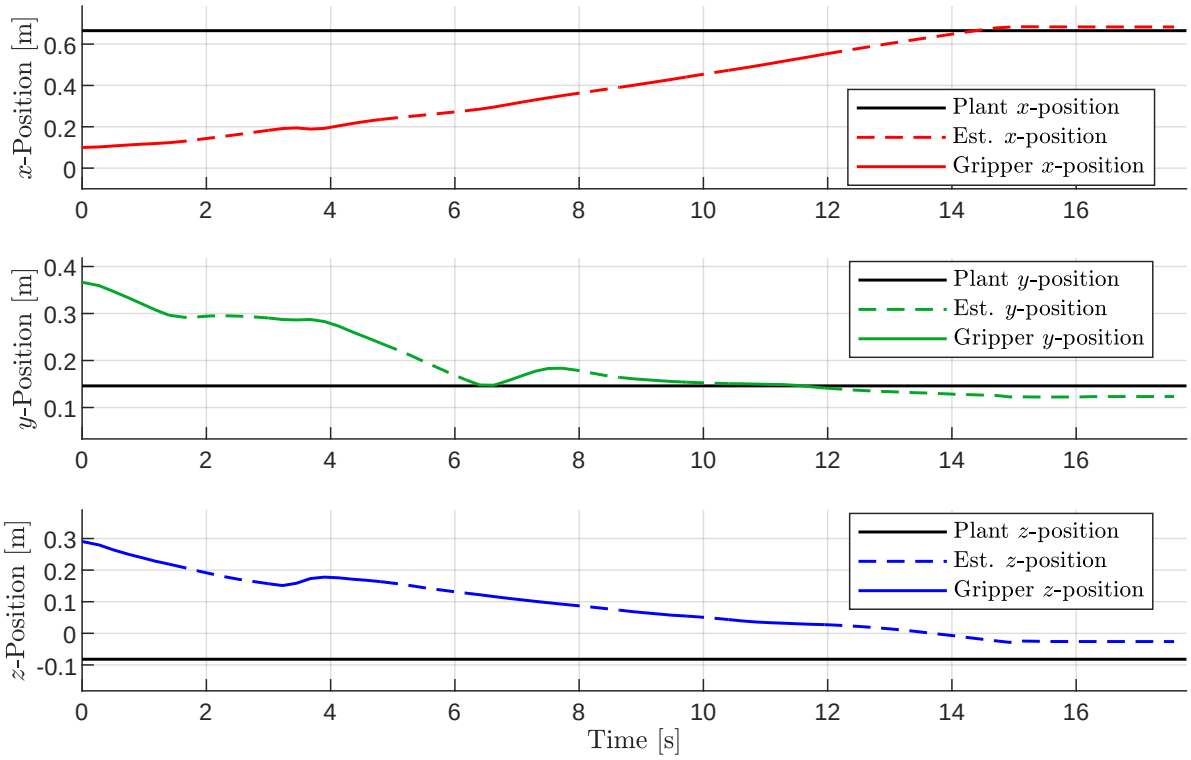


Figure 9.5: Plot of gripper position and plant during estimator test. Dashed line is when the estimator is working

To look at the test in more detail, the error between the gripper and the plant over time, can be seen in Fig. 9.6. It can be seen, that the measured error follows the true error relatively well, although with some offset/shift. In the top plot, for the error in the x-direction, the measured error follows the true error quite well, except for the drop at around 5-6 seconds. It is unknown why it does this, but maybe because of model inaccuracy, or inaccuracies in the estimator itself. In the y-error, it also follows the true error well, except for a spike around 3 seconds, and after 8-10 seconds, but this is likely some measurement error, as the measured error goes to zero, but the true error continues to decrease and becomes negative. However, looking at the y-axis scale, the numerical difference is small. The spike at 3 seconds is most likely a distance measurement that went wrong, and calculated a wrong error vector. It is most likely not related to the estimator. The same spike occurs in the z-error, although in the opposite direction, but it is likely the same reason. In the z-error, it can be seen that the error generally follows the true error, but when the detection loss occurs, the estimated error increases almost instantly, with a considerable amount. It is not clear why this happens, but it may be a mistake in the implementation of the estimator. Despite this, the measured error still goes towards zero, and meets the true error quite well after 17 seconds.
Generally when detection loss occurs or input data is missing, the estimator is able to relatively well estimate the error vector, and generate control signals to the controller, and move the arm

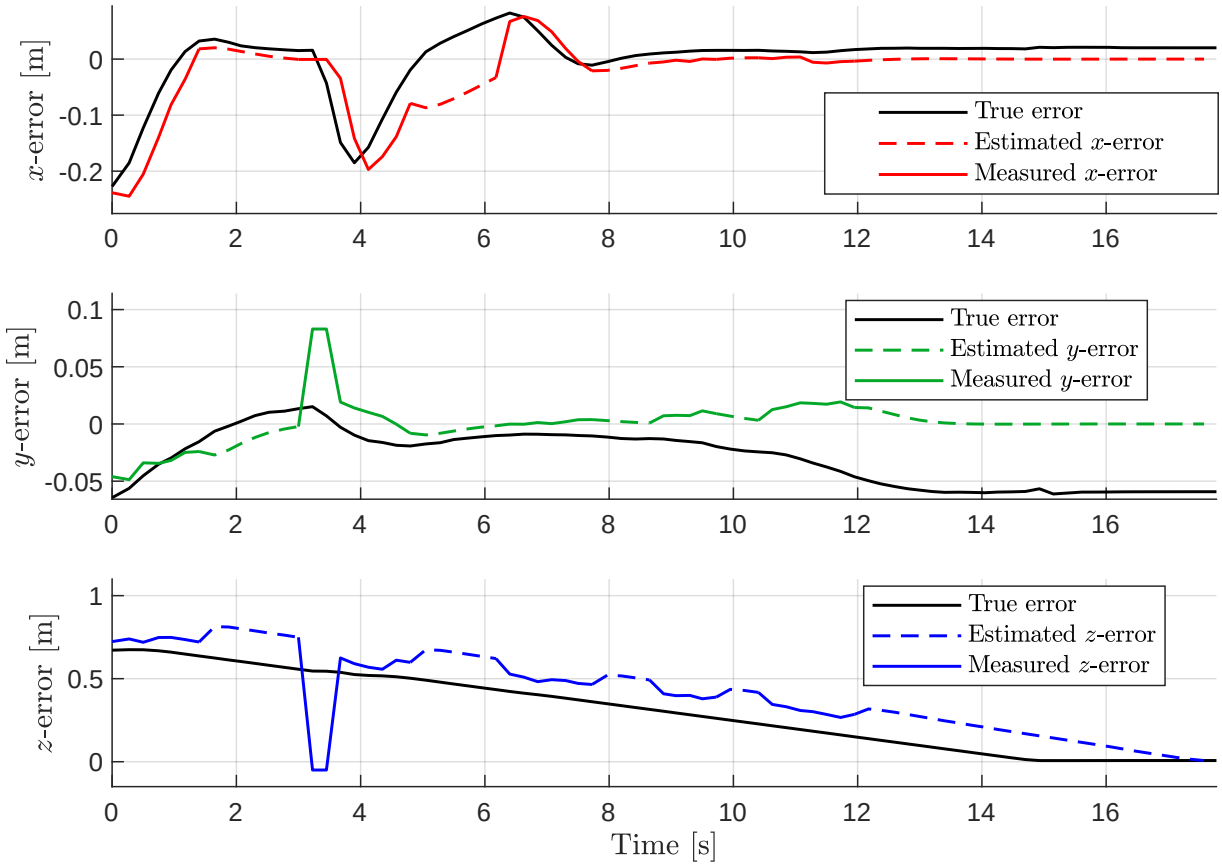


Figure 9.6: Plot of error vector between gripper and plant during estimator test. Dashed line is when the estimator is working. True error is given by the end effector position (measured) subtracted from the plant position, relative to the base

towards the plant. Also it can relatively well move itself the final distance to the plant, when the camera is too close to the plant for it to detect the plant. However, it is still clear that it is not perfect, based on the irregular deviations to the true error as seen on Fig. 9.6, and can be improved.

9.3 Neural network performance

To test the system as a whole wrt. the neural network (weed7) a test was made were the manipulator, with the camera, was stationary, and then the plant was moved in front of the camera. The plant was both tilted, moved closer to the camera and further away, to the sides, and up and down. The plant was not moved further away from the camera than 2.5 meters, since the LiDAR in the camera could not detect distances greater than this. The results of the test can be seen in Fig. 9.7. Looking at the figure it can be observed that when the distance between the camera and the plant is between 0.5 and 2.25 meters then the confidence level for the system is rather high. The confidence level is laying around 0.92. These high confidence levels shows that the performance of the neural network is good, and generally the system is good at locating the plant, even

though the distance is changed, or the plant is tilted. However, once the plant gets closer to the camera, below 0.5 meters, then the confidence levels decreases and ends at values below 0.3. The reason for these low confidence levels could be the fact that the images used for training only had a small percentage which were close up images. The figure also shows how the neural network detects two other items, and gives them a confidence level meaning they are false positives, these items are an ethernet cable and a yellow cable roll. Even though the system gets these false positives it can clearly be seen on the figure that the network is generally more confident looking at the plant, compared to when it looks at the other items. Looking at requirements 4, 5, 6 and 7 it can be concluded that these four requirements are not all achieved. In requirement 4 it is stated that the confidence level should be >=0.85, when the plant is within 1 meter of the camera, and it can be seen on Fig. 9.7 that both the confidence level drops when the plant gets to close to the camera, also single points are laying below the threshold value of 0.85. Generally it can be stated that this requirement is only met in the interval between 0.5 and 1 meter. In requirement 5 it is stated that the confidence level should be above 0.5, when the plant is within 2 meters of the camera. This requirement is also only partially met, since there are single points which are below the 0.5 confidence level threshold, as well as the in the distance interval between 0.3 and 0 meters where the confidence level also drops below 0.5. Requirement 6 states that tilted plants should be detected when it is within 2 meters of the camera. In these data points on the graph the tilted plant is also included, meaning the system is capable of detecting the tilted plant hereby resulting in this requirement being met. In requirement item 7 it is stated that the system should not detect false positives. It can be seen on Fig. 9.7 that other items, these being an ethernet cable and the yellow cable, are being detected as weeds, and is given a confidence level. This mean that this requirement is not met. However, on the figure it can be seen that even though the confidence level varies, for the plant, with the distance between the plant and the camera, then the wrong objects are generally detected with a lower confidence level, meaning these can be sorted out to some extent, by setting a minimum confidence level threshold.

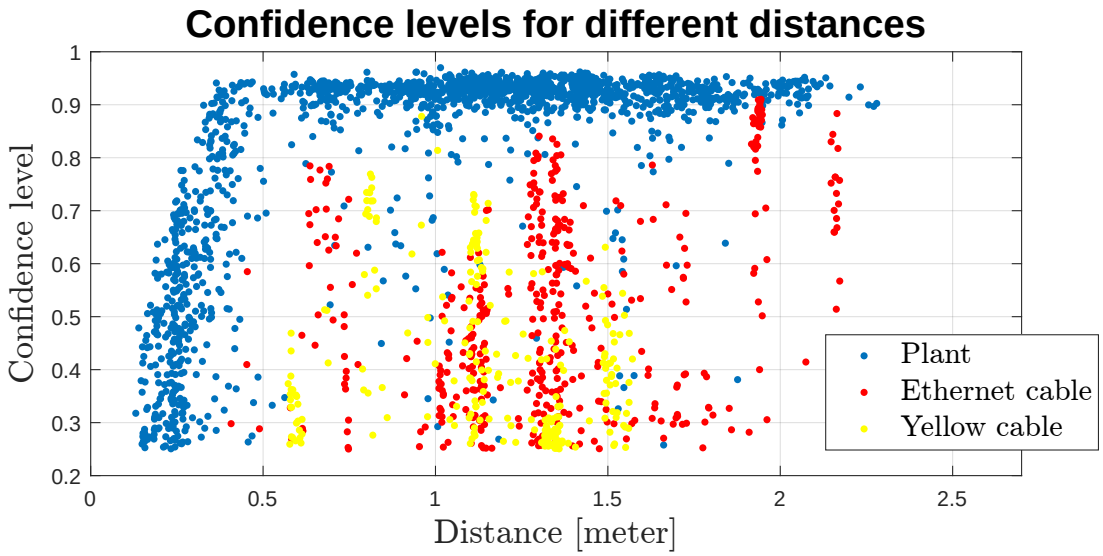


Figure 9.7: Confidence vs distance test for the plant and 2 other items: ethernet cable and a yellow cable

Chapter 10

Discussion / Perspectives

10.1 Neural network

The neural network which were used throughout this project, and described in depth in chapter 6, which as stated earlier, is based on YOLOv5 has shown to be very effective in relation to this project, as shown in section 9.3. In the testing section it was found that the used network 'Weed7' has a general confidence level above 0.9, in the distance interval of 0.5-2.25 meters, for the plant which it were trained on. This confidence level is considered to be high, taking into consideration that the network is smallest network type of the YOLOv5 familiy (the 'n' type). Also, the training made during this project has, as it can be seen in the previous sections, been based on supervised learning, meaning the data has been labelled before it was given to the network. The process of labelling these has been made automatically meaning a script was made which could produce the needed labelling data. Based on this automated labelling process, and the fact that the original images, taken of the plant, were taken by the manipulator itself. Resulted in the whole process of gathering images, and labelling them being greatly simplified, compared to a manual process. This resulted in the fact that collecting the about 1000 images, which were used during this project, could be done with ease. This gathering and labelling process which resulted in the network is assumed to have a great potential, in relation to the scope of this project. Meaning that this project only trained the network on a single type of plant, as a proof of concept, but taking into consideration how small the workload is when training and gathering the images/labelling, and also how generally good the performance of the network is then it is believed that this network could easily be expanded to contain all the classes needed. For future development of this project, it would be interesting to look into training on real plants, instead of a plastic plant as in this project, and to see how it will perform in knowing the difference between weeds and crops.
The network still has some weaknesses; these being the false positives as well as the lack of high confidence levels when the plant is close to the camera. It was observed in chapter 6 that these false positives can be minimized when more background images are added, which is something that could be expanded on if this project is to be further iterated upon. Another important aspect, which could be interesting looking into, could be the usage of a YOLOv5 network of a large type, because a larger network will result in more layers, meaning the network could be better at generalizing.

10.2 Controller

The overall performance of the system was proved by the tests to be good. The manipulator could bring the gripper close to the plant, in a position where the plant could be grabbed, and it could do that within a few seconds, more or less as intended. The test that was described

in the testing section, was performed several times, to see how the system would react to the manipulator in different initial positions, and the plant in different positions, although always within the workspace. In most cases it could bring the gripper close to the plant, so that the plant could be picked up, but in a few cases it could not. The main problems that occurred, were that the manipulator brought itself in a configuration, where it could not react to the velocity commands that were sent to it, it made a wrongful measurement of the error (which also affected the estimator), which both includes when the LiDAR could not measure a distance, and the error vector was therefore very inaccurate, and the cases where the system detects the top of the plant, as described in Fig. 9.4. These are some of the main points that will be analyzed and discussed in this section.

10.2.1 Error measurement

The testing showed, that the measured error was in most cases a good approximation of the true error, but the steady state error in the position, and the fact that the measured error converged to zero, indicates that there is some inaccuracies present in the error measurement. Based on the testing of the system in this project, there are three parts related to this problem, one is faulty measurement of the distance from the LiDAR, and the second is when only the top of the plant is detected, and the system aims for the middle of the top of the plant. The third, is the fact that when the plant is detected, it draws a 2D bounding box around the plant, and calculates a center of this bounding box. The center of the bounding box is used to calculate the direction to the center of the plant, and it is combined with the distance measurement, to get the error vector from the gripper to the plant. However, this vector describes the distance and direction to the center of the surface/outside of the plant, where it is most convenient to have the error vector point to the 3D center of the plant instead, so the plant will be fully inside the gripper, before it is grabbed.

With regards to the LiDAR distance measurement, it showed generally that it was reliable to measure distances in most cases, but sometimes when detecting the plant, the system had difficulties measuring the distance to it. In some cases it would measure the distance to the wall behind the plant, because of the gap between the leaves of the plant, and sometimes it could not even measure the distance to the central region of the plant, despite it not having any gaps. This may be due to some reflective properties of the material the plastic plant is made of, that the LiDAR simply cannot determine a distance due to the lack of reflected light. In any case, it may be so that future iterations of the product will not use a LiDAR, in order to reduce the price and complexity, so in any case it is needed to figure out a way to measure the distance to the plant, without a LiDAR. One way is to assume a known size of the plant, and use the bounding box shape to calculate the distance. This might not work well, as plants are known to have many different shapes and sizes, despite them being the same species. Furthermore, the irregular leaves of a plant may make the bounding box smaller or larger than it ideally should be, which will also cause an inaccurate measurement. A second way is to implement a second neural network, that can approximate the distance to the plant, based on what it has been trained on. It would likely require a lot of training to get useful results, if the results will even be useful, and it will add extra computations to the Jetson, which already is heavily loaded just from YOLO, but the idea would be interesting to investigate in the future. A third idea is to not calculate or measure a distance to begin with, when going into the control loop, but only a direction. The gripper can still use

the direction to get it aligned with the plant, which means it will start to shift its view between samples. Since it now views the plant from different angles, and it calculates a direction to the plant, it should be able to use the directions to the plant from different views, to triangulate a position, and thereby be able to measure a distance. This would assume having a good detection of the plant, as it is crucial that the direction vectors from the different view point to the same 3D position.

The second issue, with regards to detecting the top of the plant, so the gripper will aim for a point on the top of the plant, instead of lower on the plant, where it is actually best to grab it. Some measures were taken against this, and one of them was to apply the confidence filter, so any detections with a confidence below 0.8 would be treated as detection loss. This would both filter most false positives, and the idea was also that the confidence would decrease, when it can only see the top of the plant. However, the network was actually too good at detecting the plant, even up close, so the confidence filter did not do much of a difference. Maybe it could be a solution to only train the network to whole plants, instead of also including partial plants, so that when the camera cannot see an entire plant, it will treat it as detection loss, and let the estimator do the rest. It is then questionable whether the network would still have as good as a performance as it was found to be in this project, when it is detecting several different plants. It was also attempted to use a tilted camera mount, so the camera was tilted 25 degrees forward (looking more downwards). This actually helped with this problem, but created a new one. When the camera was tilted, the IR camera (used for depth image) could see the fingers of the gripper, and since they were so close, they lighted up the IR/depth image so much, that distance measurements anywhere else but the center region were useless, which is why this idea got discarded. However, if it was not for the issue with the gripper finger, this would be a promising fix to the problem. A final idea to this problem could be to use the first few images of the plant, and apply pose estimation of the plant, so a 3D box is created around the plant. Then it should be possible to calculate, when the entire plant is not in view, and the system should then be able to know, that what is sees is not the entire plant, and still keep the entire box around the plant in mind, when moving the gripper close to the plant. Pose estimation of the plant could also help solve the last problem (the gripper aiming for the edge of the plant, instead o of the 3D center), as when a 3D box is calculated, the center of the box can be calculated, which should be a good approximation to the true center of the plant, which the gripper can aim for. In this project, it was assumed that the plant stem is perpendicular to the ground, but that may not always be the case in real scenarios, which is also a motivation to implement pose estimation

10.2.2 Controlling the manipulator

The control method used in this project is a proportional controller which controlled the velocities of the end effector, in order to bring it close to the plant. This proved to be a simple yet effective method, but the control aspect is probably also something that should be looked into in further iterations of this product. One of the motivations for improving the control method, is that the controller implemented in this project, assumes a linear system, or at least that it operates in the linear region. This may be a good enough approximation in some cases, e.g. when the arm can move easily in a linear direction. When the arm reached the edge of the workspace, and thereby was not capable of moving any further, it was not too big of a problem control-wise, as only a

P-controller was used. However, if one were to implement integral action as well, the error in the integral term would start to accumulate as soon as it cannot move any further, giving rise to integral windup.
As mentioned, the control method in this project gave a good performance, but one of the inherent issues with this method, is that while moving based on the linear and rotational velocities, the arm will sometimes bring itself into a configuration wrt. its joint angles, where it cannot continue to execute a velocity command, and as a result just stays still. It is difficult to give a good reason as to why it cannot bring itself into another configuration, where it can follow the velocity commands sent to it, it has probably not been programmed to do so. If it should be able to bring itself into a configuration, where it can follow the velocity commands, it will need to disobey the velocity commands for a short period, until it has brought itself into a configuration where it can follow the commanded velocities. This is probably why it does not do it out of the box, because when you send velocity commands to the end effector, it should be expected that it moves with these velocities at all times, and if it is not physically able to, it will stop. While this may be the generally safe way to approach the problem, it is at the expense of performance. To solve this problem, the robot must be in an initial position relative to the plant, that guarantees that it will not encounter this case, where it cannot follow the velocity commands, or a new control method needs to be developed. It is likely very difficult to guarantee the first case, so the best option would be to implement a new control method. If a more advanced control method is implemented, that controls the individual joint angles based on the error between the end effector and the plant, it has the potential to avoid the manipulator getting stuck, as the control method can be tailored to a certain use case, such as picking up weeds, when applying more advanced control. In return, controlling the individual joint angles to get the end effector to move a certain trajectory, with certain performance criteria is a very complex problem, that involves methods such inverse kinematics, nonlinear control, model predictive control and adaptive control, which is beyond the scope of this project.

10.3 Estimator

The estimator that was implemented in this project was a deterministic state space model based estimator. The model takes the commanded velocities from the controller as input, and the states contain the velocities and the position of the error vector, between the gripper and the plant. The model was a simple linear first order model, that should be able to describe the translational movement of the error vector, as the gripper got closer to the plant. When the system could measure the error between the plant and the gripper, it would continuously update the states in the model, and as soon as no measurements could be obtained, the estimator would take over, and estimate the error between plant and gripper. The purpose of the estimator was to make the arm keep moving, despite detection loss, as the alternative would be to let the arm stop moving upon detection loss. The estimator proved to have a good performance. It was overall able to keep the gripper moving in the right direction, and in many cases, when a measurement was obtained after a short period of estimation, the measured error would be very close to the estimated error. However in some cases, it gave a slightly wrong estimate of the error. This may be due to a wrong error measurement, that was used to update the states in the model, and when the estimator has a wrong initial state when it takes over, then naturally the future estimated

errors are wrong. Issues and solutions to wrong error measurements was covered previously, so only issues directly related to the estimator is covered here. Due to the fact that a simple linear model is used, as an estimator, for the rather complex nonlinear system. As an example, when a velocity is sent to the end effector, it might not move with these exact speeds, and it might change direction slightly, in order to keep it in a configuration where the end effector can move with the commanded velocities, and the model does not take this into consideration.

10.3.1 Model selection

Improving the model in the estimator might have potential to also improve the performance of the estimator. The model used in this project was a simple linear model, that only considered the translational motion of the end effector, based on the velocity commands. One of the main issues with this model is that the commanded velocities are not always the true velocities of the end effector, as there might be a slight deviation sometimes which will cause error to accumulate in the estimator model. If some of the aforementioned nonlinear control methods are implemented, where the control inputs are joint angle velocities instead of end effector velocities, it might be possible to obtain a more accurate model. However, this model will be inherently nonlinear, so this is also something that must be taken into consideration.

10.3.2 Kalman filter

The Kalman filter is an optimal state estimator, that estimates the states of a system, and minimizes the variance of the estimation noise, and can give quite good estimates of the system states, despite it being influenced by noise/disturbances [61]. Currently, no type of filter is implemented in this system, so it might improve the performance, if some kind of filtering is added to the system. The random spikes on Fig. 9.6 might also be attenuated by a filter, as they have a short duration and should thereby be detectable by the system. Also, on the z-error plot on Fig. 9.6, there are some noticeable fluctuations of the measured error, which is not present in the true error, so it is reasonable to assume that these may be due to noise and/or inaccuracies in the estimator, which a Kalman filter may be able to improve. The standard Kalman filter needs a linear model to estimate the states, so it should be possible to implement the current model from chapter 7 with a Kalman filter, and investigate if it will give an improvement. An issue with the current estimator is also, that if the system experiences detection loss, it will start to estimate the error for as long as the detection loss occurs. Then, when it gets a new measurement from the camera/YOLO, and the estimated error differs a lot from the newly measured error (might happen due to modeling inaccuracies), it will have a large discontinuity at the point where it gets fresh data, which is not desired. A Kalman filter might also help alleviate this problem. The Kalman filter has not been attempted to be implemented during this project, as it is out of the scope of the project, and also due to time and resource constraints.

10.3.3 Parallel processes

The current control system only runs on a single threaded process, meaning that it runs sequentially, and it needs to wait for the previous code to execute, before the next code can be executed. In this system, the object detection algorithm is one of the main reasons the sample time is quite high (0.23 seconds, see chapter 7), and for each iteration of the main loop, the system must wait

for this to finish, before it can calculate the error and the control signals. If the object detection and the control system along with estimator, could be split into two parallel processes, that run simultaneously, the control system would not have to wait for the object detection to finish, before sending control signals to the manipulator, and it could just estimate the error between the gripper and the plant, while the object detection is working on detecting where the plant is in the image. As soon as the object detection has finished, and sucessfully detected a plant, it will update the states of the estimator, and the control signals, and the estimator will handle the control until the next measurement is obtained. This was looked a bit into during the project, but it was assessed that it would take too much time to implement well, so it was not looked into further.

Chapter 11
Conclusion

Due to the increasing development in robot technology usage for robots within agriculture has opened up, and an idea has been proposed by Mostec ApS, Klovborg I/S and Aalborg University, about creating a robot, with a robotic manipulator, which can autonomously drive around on the fields and remove weeds. This idea was formulated into the initiating problem for this project: "How can a robot manipulator be used to remove weeds?". Based on the initial problem, it was decided that the main focus of the project was detecting weeds and moving the manipulator onto the weeds, meaning that the moving robot platform, as well as a potential special tool used for removing the weeds, is not a part of the project. Controlling the manipulator, and moving it towards the weeds, is based on image inputs from an Intel RealSense L515 LiDAR camera which was mounted on the end effector of the manipulator. The image from the camera is passed to a neural network based on YOLOv5n, that was trained to recognize a specific plant, using transfer learning. The network will then detect the plant, and return the position of the center of the plant, on the image. The image coordinate of the plant center, was converted into an error between the gripper position and the position of the weeds using inverse projection and a distance measurement from the LiDAR. A P-controller converts the error between gripper and plant to velocity inputs for the end effector that was used to control the manipulator, such that it would move towards the plant. The resulting YOLOv5n network, called Weed7, was trained with a total of 4041 images of the plant, out of which 853 images are background images. It was found, that adding background images will lower the amount of false positives, which the network generated. Also it has been found that increasing the amount of images used during training, as well as applying data augmentation, will increase the overall performance of the network. The training was made into a semi self supervised process where the manipulator would gather the needed images, and various scripts could apply labelling data as well as data augmentation.
In the event that a plant could not be detected by the neural network, an estimator was implemented. This was based on a linear model of the gripper movement. When the neural network is able to detect the plant, the states in the model are updated, and as soon as it fails to detect a plant, the model will estimate the error between the gripper and the plant, such that the gripper still moves towards the plant, despite not getting a measurement from the neural network. During testing of the final system it was found that both the neural network, the estimator and control system worked quite well. The system was able to detect the plant reliably, and move the gripper close to the plant, and it could estimate the error in the event of detection loss. However, some of the requirements related to confidence level, could not be fulfilled and in some cases, the error measurement and/or estimation were inaccurate, causing wrong movements of the manipulator. The system fulfils most of the requirements and was shown to generally work well. Improvements to the system could include better trained neural network to avoid false positives, implementing a better estimator and model, implementing 3D pose estimation of the plant, and improving the error measurement.

Bibliography

[1] Max Roser; Hannah Ritchie and Esteban Ortiz-Ospina. *World Population Growth*. https://ourworldindata.org/world-population-growth. (Accessed on 4/2/2022). 2019.

[2] Lanbrug og fødevarer. *Grundvand og pesticider*. https://lf.dk/viden-om/miljoe-og-klima/grundvand. (Accessed on 4/2/2022).

[3] Landbrugsstyrelsen. *Statistik over økologiske jordbrugsbedrifter 2020*. https://lbst.dk/fileadmin/user_upload/NaturErhverv/Filer/Tvaergaaende/Oekologi/Statistik/Statistik_over_oekologiske_jordbrugsbedrifter_2020.pdf. (Accessed on 06/02/2022). 2021.

[4] Landbrug og Fødevarer. *Om økologi*. https://lf.dk/viden-om/oekologi/om-okologi. (Accessed on 06/02/2022).

[5] Rådet for Grøn Omstilling. *Økologisk versus konventionelt - Rådet for Grøn Omstilling*. https://rgo.dk/vi-arbejder-for/okologisk-versus-konventionelt/. (Accessed on 06/02/2022).

[6] *Økologiske varer er dyrere, men er det dyrere at være en økologisk forbruger? – Københavns Universitet*. https://ifro.ku.dk/debatindlaeg/debatindlaeg-2017/oekologiske-varer-er-dyrere-men-er-det-dyrere-at-vaere-en-oekologisk-forbruger/. (Accessed on 06/02/2022). July 26, 2017.

[7] B. Frick; E. Johnson. *Weeds - when are they a problem? - Organic Agriculture Centre of Canada - Dalhousie University*. https://www.dal.ca/faculty/agriculture/oacc/en-home/resources/pest-management/weed-management/organic-weed-mgmt-resources/weeds-problem.html. (Accessed on 06/02/2022).

[8] Dan Rossman. *Organic weed control in field crops - MSU Extension*. https://www.canr.msu.edu/news/organic_weed_control_in_field_crops. (Accessed on 06/02/2022). 2012.

[9] Automate.org. *Different Types of Agricultural Robots*. https://www.automate.org/blogs/robotics-in-agriculture-types-and-applications. (Accessed on 06/02/2022).

[10] Mark Schonbeck. *An Ecological Understanding of Weeds | eOrganic*. https://eorganic.org/node/2314. (Accessed on 07/02/2022).

[11] K. Walker and R. Frederick. "Entomological Risks of Genetically Engineered Crops". In: *Encyclopedia of Environmental Health (Second Edition)*. Ed. by Jerome Nriagu. Second Edition. Oxford: Elsevier, 2019, pp. 339–346. ISBN: 978-0-444-63952-3. DOI: https://doi.org/10.1016/B978-0-12-409548-9.11677-X. URL: https://www.sciencedirect.com/science/article/pii/B978012409548911677X.

[12] FAO. *Damage and loss*. https://www.fao.org/resources/digital-reports/disasters-in-agriculture/en/. (Accessed on 07/02/2022).

[13] Stony Brook University. *Sustainable Vs. Conventional Agriculture | Environmental Topics and Essays*. https://you.stonybrook.edu/environment/sustainable-vs-conventional-agriculture/. (Accessed on 08/02/2022).

[14] Rune Gottlieb Skovgaard; cand.scient.pol.; Flemming Schiøtt Hansen; restauratør. *Konventionelt landbrug er ikke vejen frem*. https://www.berlingske.dk/kommentatorer/konventionelt-landbrug-er-ikke-vejen-frem. (Accessed on 08/02/2022). 2014.

[15] Encyclopaedia Britannica. *weed - Chemical control | Britannica*. https://www.britannica.com/plant/weed/Chemical-control. (Accessed on 08/02/2022).

[16] Encyclopaedia Britannica. *herbicide | History, Types, Application, & Facts | Britannica*. https://www.britannica.com/science/herbicide. (Accessed on 08/02/2022).

[17] Samantha Jakuboski. *The Dangers of Pesticides | Green Science | Learn Science at Scitable*. https://www.nature.com/scitable/blog/green-science/the_dangers_of_pesticides/. (Accessed on 08/02/2022). 2011.

[18] Tomek de Ponti, Bert Rijk, and Martin K. van Ittersum. "The crop yield gap between organic and conventional agriculture". In: *Agricultural Systems* 108 (2012), pp. 1–9. ISSN: 0308-521X. DOI: https://doi.org/10.1016/j.agsy.2011.12.004. URL: https://www.sciencedirect.com/science/article/pii/S0308521X1100182X.

[19] Encyclopaedia Britannica. *organic farming | Definition, History, Methods, Practices, & Benefits | Britannica*. https://www.britannica.com/topic/organic-farming. (Accessed on 08/02/2022).

[20] *weed | Definition, Examples, & Control | Britannica*. https://www.britannica.com/plant/weed. (Accessed on 08/02/2022).

[21] Mark Schonbeck. *An Organic Weed Control Toolbox | eOrganic*. https://eorganic.org/node/2782. (Accessed on 08/02/2022).

[22] Henrik Terp;Søren Ilsøe. *Hvis vi landmænd stopper med at pløje jorden, kan vi lagre CO2 i markerne | Information*. https://www.information.dk/debat/2019/02/landmaend-stopper-ploeje-jorden-kan-lagre-co2-markerne. (Accessed on 08/02/2022). 2019.

[23] Petar Durdevic Løhndorf and Daniel Ortiz Arroyo. *ROBOT PLATFORM WITH A ROBOTIC ARM FOR AGRICULTURE APPLICATIONS USING VISUAL SERVOING*. https://www.moodle.aau.dk/pluginfile.php/2452479/mod_resource/content/3/AIE6Project_Agriculture-2022.pdf. (Accessed on 27/05/2022). 2022.

[24] Intel. *Buy Intel® RealSense™ LiDAR Camera L515*. https://store.intelrealsense.com/buy-intel-realsense-lidar-camera-l515.html. (Accessed on 15/02/2022).

[25] Intel. *Intel® RealSense™ LiDAR Camera L515 Datasheet*. https://docs.rs-online.com/f31c/A700000006942953.pdf. (Accessed on 15/02/2022).

[26] Nvidia. *NVIDIA Announces Jetson Nano: $99 Tiny, Yet Mighty NVIDIA CUDA-X AI Computer That Runs All AI Models | NVIDIA Newsroom*. https://nvidianews.nvidia.com/news/nvidia-announces-jetson-nano-99-tiny-yet-mighty-nvidia-cuda-x-ai-computer-that-runs-all-ai-models. (Accessed on 14/02/2022). 2019.

[27] Danica Kragic, Henrik Christensen, and Fiskartorpsv A. "Survey on Visual Servoing for Manipulation". In: *Comput. Vis. Act. Percept. Lab. Fiskartorpsv* 15 (Feb. 2002).

[28] Zakariae Machkour, Daniel Ortiz-Arroyo, and Petar Durdevic. "Classical and Deep Learning based Visual Servoing Systems: a Survey on State of the Art". In: *Journal of Intelligent & Robotic Systems* 104.1 (2021), p. 11. ISSN: 1573-0409. DOI: 10.1007/s10846-021-01540-w. URL: https://doi.org/10.1007/s10846-021-01540-w.

[29] ultralytics. *YOLOv5 Documentation*. https://docs.ultralytics.com/. (Accessed on 18/04/2022).

[30] *YOLOv5 is Here: State-of-the-Art Object Detection at 140 FPS*. https://blog.roboflow.com/yolov5-is-here/. (Accessed on 04/27/2022).

[31] *Rigid body definition and meaning | Collins English Dictionary*. https://www.collinsdictionary.com/dictionary/english/rigid-body. (Accessed on 23/02/2022).

[32] Kinova. *Kinova® Gen3 lite robot User guide*. NA.

[33] L. Sciavicco and B. Siciliano. *Modelling and control of robot manipulators*. 2. ed. Springer, 1996.

[34] Kinova. *Index of generic-public/kortex/API/2.3.0*. https://artifactory.kinovaapps.com/artifactory/generic-public/kortex/API/2.3.0/. (Accessed on 22/02/2022).

[35] Kinova. *GitHub - Kinovarobotics/kortex: Code examples and API documentation for KINOVA® KORTEX™ robotic arms*. https://github.com/Kinovarobotics/kortex. (Accessed on 22/02/2022).

[36] Kinova. *Learning Hub - How-to series: Kortex™ API - YouTube*. https://www.youtube.com/playlist?list=PLz1XwEYRuku5rZjJWBr6SDi93jgWZ4FHL. (Accessed on 22/02/2022).

[37] Thomas Kjær Nowak and Christian Tranholm Novrup. *cnovrup/p6_project*. https://github.com/cnovrup/p6_project. (Accessed on 27/05/2022).

[38] Intel Realsense Team. *pyrealsense2 — pyrealsense2 2.33.1 documentation*. https://intelrealsense.github.io/librealsense/python_docs/_generated/pyrealsense2.html#module-pyrealsense2. (Accessed on 22/03/2022). 2017.

[39] Intel Realsense Team. *librealsense/wrappers/python at master · IntelRealSense/librealsense*. https://github.com/IntelRealSense/librealsense/tree/master/wrappers/python. (Accessed on 22/03/2022).

[40] Milan. Sonka. *Image processing, analysis, and machine vision*. eng. 3. ed. Stamford CT: Cengage Learning, 2008. ISBN: 9780495244387.

[41] *CS231n Convolutional Neural Networks for Visual Recognition*. https://cs231n.github.io/convolutional-networks/. (Accessed on 03/05/2022).

[42] *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way | by Sumit Saha | Towards Data Science*. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53. (Accessed on 25/04/2022).

[43] *YOLO V5 - Explained and Demystified – Towards AI*. https://towardsai.net/p/computer-vision/yolo-v5%E2%80%8A-%E2%80%8Aexplained-and-demystified. (Accessed on 23/05/2022).

[44] *A Gentle Introduction To Sigmoid Function*. https://machinelearningmastery.com/a-gentle-introduction-to-sigmoid-function/. (Accessed on 23/05/2022).

[45] *Leaky ReLU Explained | Papers With Code*. https://paperswithcode.com/method/leaky-relu. (Accessed on 23/05/2022).

[46] *Why Training a Neural Network Is Hard*. https://machinelearningmastery.com/why-training-a-neural-network-is-hard/. (Accessed on 03/05/2022).

[47] *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way | by Sumit Saha | Towards Data Science*. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53. (Accessed on 31/03/2022).

[48] *Transfer Learning Guide: A Practical Tutorial With Examples for Images and Text in Keras - neptune.ai*. https://neptune.ai/blog/transfer-learning-guide-examples-for-images-and-text-in-keras. (Accessed on 31/03/2022).

[49] *PyTorch documentation — PyTorch 1.11.0 documentation*. https://pytorch.org/docs/stable/index.html. (Accessed on 18/04/2022).

[50] *COCO - Common Objects in Context*. https://cocodataset.org/#home. (Accessed on 18/04/2022).

[51] *The practical guide for Object Detection with YOLOv5 algorithm | by Lihi Gur Arie, PhD | Mar, 2022 | Towards Data Science*. https://towardsdatascience.com/the-practical-guide-for-object-detection-with-yolov5-algorithm-74c04aac4843. (Accessed on 18/04/2022).

[52] *Transfer Learning Guide: A Practical Tutorial With Examples for Images and Text in Keras - neptune.ai*. https://neptune.ai/blog/transfer-learning-guide-examples-for-images-and-text-in-keras. (Accessed on 18/04/2022).

[53] *Transfer Learning with Frozen Layers - YOLOv5 Documentation*. https://docs.ultralytics.com/tutorials/transfer-learning-froze-layers/. (Accessed on 18/04/2022).

[54] *Breaking Down Mean Average Precision (mAP) | by Ren Jie Tan | Towards Data Science*. https://towardsdatascience.com/breaking-down-mean-average-precision-map-ae462f623a52. (Accessed on 02/06/2022).

[55] *Tips for Best Training Results- YOLOv5 Documentation*. https://docs.ultralytics.com/tutorials/training-tips-best-results/. (Accessed on 10/05/2022).

[56] *Epochs, Batch Size, & Iterations - AI Wiki*. https://machine-learning.paperspace.com/wiki/epoch?fbclid=IwAR090iyCVhsmrNutAfZKit0HyGgSzgVwvQ69HS-4qF5NEvgjkYaSR (Accessed on 12/05/2022).

[57] *A detailed example of data loaders with PyTorch*. https://stanford.edu/~shervine/blog/pytorch-how-to-generate-data-parallel. (Accessed on 12/05/2022).

[58] Petar Durdevic Løhndorf and Daniel Ortiz Arroyo. "Dynamic Analysis and Modeling of DNN-based Visual Servoing Systems". English. In: (July 2022). SAI Computing Conference 2022 ; Conference date: 14-07-2022 Through 15-07-2022.

[59] *Python Control Systems Library — Python Control Systems Library dev documentation*. https://python-control.readthedocs.io/en/0.9.1/. (Accessed on 16/05/2022).

[60] *Feedback control of dynamic systems*. eng. 8. ed. New York: Pearson, 2018. ISBN: 9780134685717.

[61] Dan Simon. *Kalman Filtering*. http://aug-roma.wdfiles.com/local--files/progetti:arpinpero/Kalman_filtering.pdf. (Accessed on 23/05/2022). 2001.

Appendix A

Transformation matrices for the Kinova robotic arm

$$\mathbf{A_1^0}(q_1) = \begin{bmatrix} \cos(q_1) & 0 & \sin(q_1) & 0 \\ \sin(q_1) & 0 & -\cos(q_1) & 0 \\ 0 & 1 & 0 & 243.3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{A.1}$$

$$\mathbf{A_2^1}(q_2) = \begin{bmatrix} -\sin(q_2) & \cos(q_2) & 0 & -280\sin(q_2) \\ \cos(q_2) & \sin(q_2) & 0 & 280\cos(q_2) \\ 0 & 0 & -1 & 30 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{A.2}$$

$$\mathbf{A_3^2}(q_3) = \begin{bmatrix} -\sin(q_3) & 0 & \cos(q_3) & 0 \\ \cos(q_3) & 0 & \sin(q_3) & 0 \\ 0 & 1 & 0 & 20 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{A.3}$$

$$\mathbf{A_4^3}(q_4) = \begin{bmatrix} -\sin(q_4) & 0 & \cos(q_4) & 0 \\ \cos(q_4) & 0 & \sin(q_4) & 0 \\ 0 & 1 & 0 & 245 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{A.4}$$

$$\mathbf{A_5^4}(q_5) = \begin{bmatrix} -\cos(q_5) & 0 & -\sin(q_5) & 0 \\ -\sin(q_5) & 0 & \cos(q_5) & 0 \\ 0 & 1 & 0 & 57 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{A.5}$$

$$\mathbf{A_6^5}(q_6) = \begin{bmatrix} -\sin(q_6) & -\cos(q_6) & 0 & 0 \\ \cos(q_6) & -\sin(q_6) & 0 & 0 \\ 0 & 0 & 1 & 235.0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{A.6}$$

$$\mathbf{A_g^6} = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{A.7}$$