Eötvös Loránd University

Faculty of Informatics

Erasmus Mundus Joint Master in

Intelligent Field Robotic Systems

Vision Based Reactive Navigation for Agricultural Robotics Operations

Author:

Muhammad Umar

Intelligent Field Robotic Systems MSc

Internal supervisor:

Dr. Zoltán Istenes

Associate Professor

External supervisor:

Dr. Carlos Rizzo

Principal Investigator

Budapest, 2023

STATEMENT

OF THESIS SUBMISSION AND ORIGINALITY

I hereby confirm the submission of the Master Thesis Work on the IFRoS MSc course with author and title:

Name of Student: Muhammad Umar

Code of Student: TOJK2M

Title of Thesis: Vision Based Reactive Navigation for Agriculture Robotics Operations

Supervisor: Dr. Zoltán Istenes

.....

at Eötvös Loránd University, Faculty of Informatics.

In consciousness of my full legal and disciplinary responsibility I hereby claim that the submitted thesis work is my own original intellectual product, the use of referenced literature is done according to the general rulesof copyright.

I understand that in the case of thesis works the following acts are considered plagiarism:

• literal quotation without quotation marks and reference;

• citation of content without reference;

• presenting others' published thoughts as own thoughts.

Budapest, 2023.

Muhammad Umar

.....

student

EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

MASTER THESIS TOPIC DECLARATION FORM

Name of Student: **Muhammad Umar**

Neptun code: TOJK2M

Training: Full-time

Technical major: Intelligent Field Robotic Systems (IFRoS)

Email Address: muhammadumar559@gmail.com

Phone Number: +34 632208598

Name of ELTE supervisor: ZOLTÁN ISTENES

Department: Faculty of Informatics

Contact (email, phone): istenes@inf.elte.hu

Name of Industrial supervisor: Carlos Rizzo

Department: Robotics And Automation

Contact (email, phone): carlos.rizzo@eurecat.org | (+34) 93 594 47 00

Information about the internship

Name of the company: FUNDACIÓ EURECAT

Starting date of internship: 27/02/2023

Closing date of internship: 29/05/2023

Weekly schedule: 30 hours per week, from Monday to Friday, with flexibility of schedule (from 8 AM to 19 PM) and teleworking

The purpose of the admission declaration is to certify that the student of the MSc in Intelligent Field Robotics System at ELTE Faculty of Informatics may complete the mandatory internship in the selected institution within the framework detailed hereby and in accordance with the learning outcomes required by the program.

Title of the thesis: Vision based reactive navigation for agricultural robotics operations

Given the lack of labour and the increasing demand of agriculture requiring a lot of manpower, agriculture is a sector that would greatly benefit from automation and robotics solutions. The technologies and algorithms developed in the robotics sector for localization and autonomous navigation have been intensively addressed in the last two decades, mainly in indoor environments (for example, logistics warehouses). In these cases, the scenario is usually quite structured, controlled, predictable and limited in size, and the terrain is usually quite flat and regular. Lidar based SLAM, for instance, has been intensively implemented and deployed with great results.

These solutions, however, do not perform well in agricultural scenarios. The objective of this Thesis is to develop:

- A reactive navigation solution, based on vision and lidar sensors, in order to obtain a robust navigation solution.
- Overcome the problems associated with navigation derived from an erratic localization system, given the difficulty of the scenario.

Proposed, preliminary research questions:

- Using the perception module, how can the robot estimate the center of the vineyard row based on the information from sensors such as lidars or cameras (including research, state of the art review)
- How should the navigation module, given information of the center of the row, output velocity commands (using PID controllers, pure pursuit, etc).
- How well the robot performs in a real environment (precision, accuracy of both perception and navigation module in a vineyard)

Methodology: how, what methods, technology, tools, could be used?

The solution will be built with the ground robotics team at Eurecat. The perception can be done using OpenCV and ML models. We will also look into how LiDAR can help us in estimating the rows of vineyards. It needs research and state of the art review. The Navigation module will be using the information from the perception module to develop a motion controller for the robot. It could use PID controllers or pure pursuit technologies etc. All the modules will be integrated in ROS with the robot. C++ will be mainly used for programming.

5 keywords:

ROS, Perception, Navigation, PID, Image Processing.

Encryption of the topic is necessary: **YES/NO**

I ask for the acceptance of my thesis topic.

Budapest, 06/02/2023

.....

Student

I approve of the suggested topic of the Master's Thesis:

Budapest, 16/2/2023

.....

ELTE supervisor

I approve of the suggested topic of the internship, and in the name of the institution (organization, company) above I agree, that the named student will carry out his/her internship along the conditions detailed above:

Budapest, 07/02/2023

.....

Dr. Carlos Rizzo
Industrial supervisor

The topic of the thesis and internship is approved by the Dean of Faculty of ELTE Informatics

Budapest, …./…./202

Kozsik Tamás
Digitally signed by Dr. Kozsik Tamás
Date: 2023.02.23 19:45:32 +01'00'

.....

Dr. Tamás Kozsik
Dean of Faculty of Informatics

Contents

Acknowledgements

I am thankful to the consortium of Erasmus Mundus Joint Master in Intelligent Field Robotic Systems for selecting me for this scholarship and giving me an opportunity to achieve my career goals. I am extremely grateful to my supervisors Dr. Zoltán Istenes and Dr. Carlos Rizzo, for their guidance and to my colleagues at Eurecat, Mateus, Pau, and Xavier, for their support. A special mention to all my class fellows from the first IFRoS batch who made this program memorable.

Dedicated to my Wife and my Parents for their endless support

List of Figures

List of Tables

List of Algorithms

List of Codes

List of Abbreviations

The next list describes several abbreviations that will be later used within the body of the document

AI	Artificial Intelligence
EKF	Extended Kalman Filter
FPS	Frames Per Second
$GNSS$	Global Navigation Satellite System
GPS	Global Positioning System
IMU	Inertial Measurement Unit
IR	Infrared
$LiDAR$	Light Detection and Ranging
$ONNX$	Open Neural Network Exchange
PID	Proportional–Integral–Derivative
ROS	Robot Operating System
$SLAM$	Simultaneous Localization And Mapping
WSL	Windows Subsystem for Linux

Abstract

Food production and agriculture have been of utmost importance for the development of humanity. Given the lack of labour and the increasing demand for agriculture requiring a lot of manpower, this sector would greatly benefit from automation and robotics solutions. The technologies and algorithms developed in the robotics sector for localization and autonomous navigation have been intensively addressed in the last two decades, mainly in indoor environments (for example, logistics warehouses). In these cases, the scenario is usually structured, controlled, predictable, and limited in size, and the terrain is usually flat and regular. LiDAR-based SLAM, for instance, has been intensively implemented and deployed with excellent results. These solutions, however, do not perform well in agricultural scenarios. We aim to use vision-based sensors such as RGB-D cameras to estimate traversable lanes in agriculture fields. We are using the Detectron2-based Keypoint Detection model to detect trees. The detection is performed on the color image, and the position is estimated using the depth image. Vision-based systems are cheaper than LiDAR but require much more computation and storage space. We will convert our model to OpenVino and TensorRT engine to optimize the deep learning model for embedded boards. So far, the Keypoint Detection model has not been optimized with such tools. Our implementation converts this model to the TensorRT engine, increasing the inference speed by almost 2.5x. Reactive navigation based on the Pure Pursuit algorithm is used to traverse the fields while maintaining a safe distance from plants. The navigation module is tested in simulation with lane traversal and lane changing, and the perception module is tested in both simulation and real-world environments.

Chapter 1

Introduction

1.1 Background and context

The world is currently facing an increasing demand for food production. And given the lack of labour and human resources required in the agriculture sector, it can benefit significantly from automation and robotic solutions. The current methods of planting, weeding, etc., require a lot of resources. Most algorithms developed in the last two decades for localization and autonomous navigation have been mainly for the indoor environment. These solutions are deployed in warehouses, factories, and other indoor industries.

In indoor environments, the scenario is usually quite structured, controlled, predictable, and limited in size. The terrain is also relatively flat and regular. For instance, LiDAR-based SLAM has been intensively implemented and deployed with good results and precision. These solutions, however, do not perform well in agricultural scenarios with uneven terrains and uncertain environments.

1.2 Problem statement

The growing population demands automated agricultural practices. Due to the technological advancements in Computer Vision and Robotics in the past few years, it is possible to develop precise and safe robotic solutions for farming. While there are many state-of-the-art indoor navigation solutions, they do not perform well in uncontrolled outdoor environments.

1.3 Research questions

We will focus on presenting an answer to the following questions:

- Using a perception module, how can the robot estimate a feasible path to traverse a row (e.g., in a vineyard) based on the information from sensors such as RGB-D cameras?
- How can the perception module be optimized to achieve suitable FPS (at least 5 FPS) on embedded boards?
- How should the navigation module, given information of the center of the row, output velocity commands (using PID controllers, pure pursuit, etc.)?

1.4 Objectives and scope

The objectives of this thesis are to:

- Create a tool for adapting to different agricultural environments by generating simulation environments to test/tune the developed algorithms.
- Build a perception module capable of estimating a feasible path for an agricultural robot based on information from "cheap" sensors such as cameras.
- Develop a reactive navigation solution, based on outputs from the perception module, to safely follow the estimated path.
- Test the developed modules in simulations and real environments.
- Overcome the problems associated with navigation derived from an erratic localization system, given the difficulty of the scenario.

1.5 Methodology and approach

The perception module uses a deep learning model to detect trees in an apple plantation orchard. Using an RGB-D camera, We can estimate the position of these detected trees through the depth image. The tree detection using the deep learning model can be done using RGB or depth images. We are, however, using RGB images for tree detection and depth images for position estimation of the detected trees.

The thesis also expands on optimizing the deep learning model using tools such as OpenVino and TensorRT.

The navigation module uses the perception module's information about the tree location to develop a motion controller for the robot. The Pure Pursuit algorithm is used to reactively navigate the agriculture field by following the center of the vineyard rows. All the modules will be integrated into the robot using ROS2 Foxy. The modules are written using C++ and Python language. Gazebo is used for simulating the environment, and RViz is used for visualizations.

1.6 Contribution of the thesis

The thesis develops an industrial robotics solution for reactive navigation using vision-based tree detection and position estimation. The solution is developed using ROS2 and state-of-the-art Detectron2[1] model. We will verify the results of [2] and check if it can work with an industrial robot to detect trees as landmarks for navigation. The deep learning model is further optimized using OpenVino and TensorRT frameworks to run on embedded boards installed on the robots. There is no available solution to convert Keypoint Detection in Detectron2 to TensorRT, which this thesis successfully does.

1.7 Overview of the thesis structure

The thesis begins with a literature review of the vision-based systems currently or previously deployed in agriculture fields. We will dive deep into state-of-the-art systems for reactive navigation in outdoor fields. A solution is proposed describing the algorithms, approach, and tools used for precise detection and navigation. After that, the results are presented and discussed. The main part of the thesis ends with a conclusion of the findings and answers the research questions presented at the start. The thesis also includes an appendix at the end for further reading.

Chapter 2

Literature Review

2.1 Overview of relevant literature

Localization based on the surroundings and navigation in complex environments is a very broad topic in robotics, and much research is being done in these domains. The indoor navigation and localization are usually very precise due to the nature of the surroundings, as it is more controlled and has better landmark features for position estimation of the robot. However, in outdoor environments, specifically in agricultural fields, the uneven terrain combined with the uncontrolled environment gives very erratic localization and navigation results.

Research is done to use algorithms with a combination of different sensors to fix this erratic behaviour. We will be focusing on finding ways to combine vision-based systems such as cameras with reactive navigation approaches.

2.2 Related work and previous research

Agricultural robotics has been a hot topic for at least a decade, and different solutions have been put forward for navigation in the farming fields. We will be discussing the use of cameras and depth sensors for navigation.

2.2.1 Vision based navigation

Initially, plant detection was mainly done using color contrast of plants and soil. We can use different types of cameras for these purposes. The authors in [3] are using an IR camera and applying Hough Transform for row detection in a field. The

high contrast between the soil (dark) and plants (light) in the infrared image helps with the estimation of plant rows. The same can also be done by RGB cameras using color segmentation [4]. The robot looked for green color using the three color channel values. After that, they grouped contiguous pixels and eliminated low count. The path was determined based on the center of the left and right segmentation boundary, and a map was generated at the end.

Fast forward a few years, and newer techniques, such as 3D structures, were used. Depth cameras were used by [5] for clustering and detection of humans in a depth image. The detection was used for human tracking purposes, but it can be extrapolated to different objects, such as plants and trees. Methods using 3D structures mainly used stereo vision, depth cameras, or laser scanning but [6] used Monocular vision for robot navigation in outdoor fields. They tracked the direction of the rows of crops. It was assumed that most crops are in a straight row and on flat ground. IMU sensor was used for estimating camera attitude and for tracking the visual horizon. The method used no image segmentation of plants and soil and did not need any assumptions about the appearance of row spacing, color, and lighting. The algorithm was later improved in [7], which utilized different sensors such as stereo cameras, GPS, and INS for obstacle detection and developed kinematically feasible navigation.

With the advancement of Machine Learning and Neural Networks, AI-based vision robotics is quickly taking over the former methods of row approximations. [8] uses a Convolution Neural Network (CNN) for semantic segmentation of a color image into binary classification. It divides the terrain into crops or no crops, i.e., not drivable and drivable terrain, respectively. These techniques to find rows are also applied for road line detection in [9]. And after estimating the row or road, they used the vanishing point, i.e., the intersection of the road line, and estimated the angle from the vanishing point. Instead of detecting rows, we can also detect plants. [10] performed vegetation segmentation and used connected components operations to find individual plants. Secondly, they detected the number of crop rows by estimating the moving variance. Finally, the rows were tracked by centering a parallelogram on each row. [11] used supervised learning for under-canopy navigation. They estimated the heading and placement in a row using the machine learning model and fused this data with inertial measurements using EKF filter. The robot stayed in the center of the row using a robust nonlinear controller such as Model Predictive Control (MPC).

For our use case in Apple plantation orchards or strawberry fields, we can detect tree trunks or poles as landmarks and estimate the position of these landmarks using depth images. Using Detectron2 Module provided by the Facebook AI Research team, [12] used two annotated datasets of synthetic and real RGB-D images in natural forest environments to train a tree-trunk detection model and then modified the object detectors to estimate the felling cut, diameter, and inclination for each tree. The training parameters and inference time of trained models on the two datasets are provided in [2]. They have been able to achieve around 18 FPS using the ResNet-50-FPN backbone in the Detectron2 model, which has 25.6 million parameters. This FPS is achieved using a very powerful 24GB NVIDIA RTX-3090 GPU. Given that the model is already trained with good results, we will use the same pre-trained model and optimize it further using TensorRT.

2.2.2 Reactive navigation

We will be focusing on a reactive navigation approach for traversing the fields. Reactive Navigation, in simple terms, is to plan and control the motion of the robot to reach a desired goal position, but the knowledge of features in the environment becomes available only after the robot starts moving [13]. The robot reacts to the information in real-time and plans its path while avoiding obstacles.

In a corn field, [14] presents a reactive navigation solution using a LiDAR sensor and H∞ controller. The proposed navigation system is an alternative to GNSS-based systems. It uses a histogram filter for row detection and an H∞ controller to keep the robot in the lane center. The controller was also compared with PID controller in a controlled environment. [15] uses a hybrid reactive and GPS-based navigation where LiDAR and RGB-D device is used to perform row following and obstacle avoidance. A reactive approach or GPS waypoints are used for changing from row to row or field to field.

As we detect the trees using a deep learning model, we can estimate the row of trees and take the center of the fields as our navigation path. We will then use the pure pursuit algorithm developed by [16]. The pure pursuit was later tested against different algorithms, such as the Quintic Polynomial approach and a "Control Theory" approach by [17]. The implementation details of the pure pursuit algorithm are explained in detail by [18], and we will be following it to implement and integrate

pure pursuit in our navigation solution.

2.3 Gaps in the literature and research questions

While research is being done on developing robot navigation solutions using different vision algorithms, there is a need to deploy newer deep learning modules to test the feasibility of real-time detection and navigation using Machine Learning on robot's embedded boards. These vision algorithms should be optimized using different tools to provide good inference time and FPS on low-power embedded boards.

Chapter 3

Methodology

3.1 Research design and approach

For the perception module, we have decided to use state-of-the-art tree trunk detection with a pre-trained Detectron2 module [12]. The model outputs keypoint detections on a color image, and we can map those keypoints to the depth image and extract distance information. From this information, we can estimate the row center, which will be the input for the navigation stack. The model, however, is computationally very heavy and requires a good GPU for faster inference. As the authors of [2] described that around 10-18 FPS was achieved using the models on one of the best available GPUs, it will decrease significantly on Jetson Devices and other embedded boards. For this, we will be optimizing the model using OpenVino and TensorRT. OpenVino is provided by Intel for optimizing models, mainly for their processors. TensorRT is a tool by NVIDIA for optimizing the inference time of deep learning models for NVIDIA GPUs.

We will use Gazebo as our simulation environment to effectively test our algorithms. The navigation module will require the implementation of the pure pursuit algorithm as mentioned in Algorithm 1. The algorithm will take the path as input and output velocity commands.

3.2 Tools and technologies used

For communication with the robot, we are using ROS2-Foxy. We will start with building a docker environment. Docker ensures the code can run without installing

specific versions of Ubuntu, and other dependencies can be resolved easily. For the simulation, we have measurements from an apple field in Girona, Spain, where we will perform real-world tests. The same field will be regenerated in the Gazebo environment so our tests are as close to reality as possible.

3.2.1 Docker environment for ROS2 Foxy

As ROS2 needs newer Ubuntu versions making it difficult to run and test our code on different machines, we will take Docker environments' help. Docker helps to virtualize software so it can run independently of the platform. The ROS2 Foxy is set up inside Docker with all the required dependencies, such as OpenCV, PyTorch, Gazebo, etc. A volume with ROS2 workspace is mounted to efficiently work with the Docker environment. To run the Docker environment, we will be using WSL2 enabled Windows platform.

3.2.2 Gazebo environment

To test our code before going to the real robot, we can generate a simulated environment of an agriculture field inside the Gazebo. A tractor model is used, closely resembling the real tractor used in an apple field in Girona. The tractor robot is equipped with an RGB-D camera and LiDAR.

We need an apple tree model for Gazebo to create the apple field. We have information about the number of rows and distances between each apple tree from the real field in Girona. We will try to place trees at the same distances to make the simulation as close to the real field as possible. For this, a Python script is written that generates world files used by Gazebo. The script takes different parameters such as the model name, the total number of rows, the number of trees in a row, the distance between each row, the distance between each tree, and the initial distance of trees from the robot. The script then places the trees according to the given dimensions and generates a world file ready for use with Gazebo. We can tweak the parameters to match the real-world measurements as close as possible.

3.2.3 Apple tree model

A 3D model of apple trees, as shown in Figure 3.1, is used, which is 110-120 meters high and 10-14 meters wide. It is then converted to Collada dae file format

so it can be used inside Gazebo. We will apply a scaling factor of 0.04 to change the height and width to match our real-world requirements. The scaling factor can be changed depending on the given measurements of the real trees. The model also has different nodes to display the leaves of the trees. But as our model only uses tree trunks for detection, we will turn off the other layers in Gazebo to save resources.

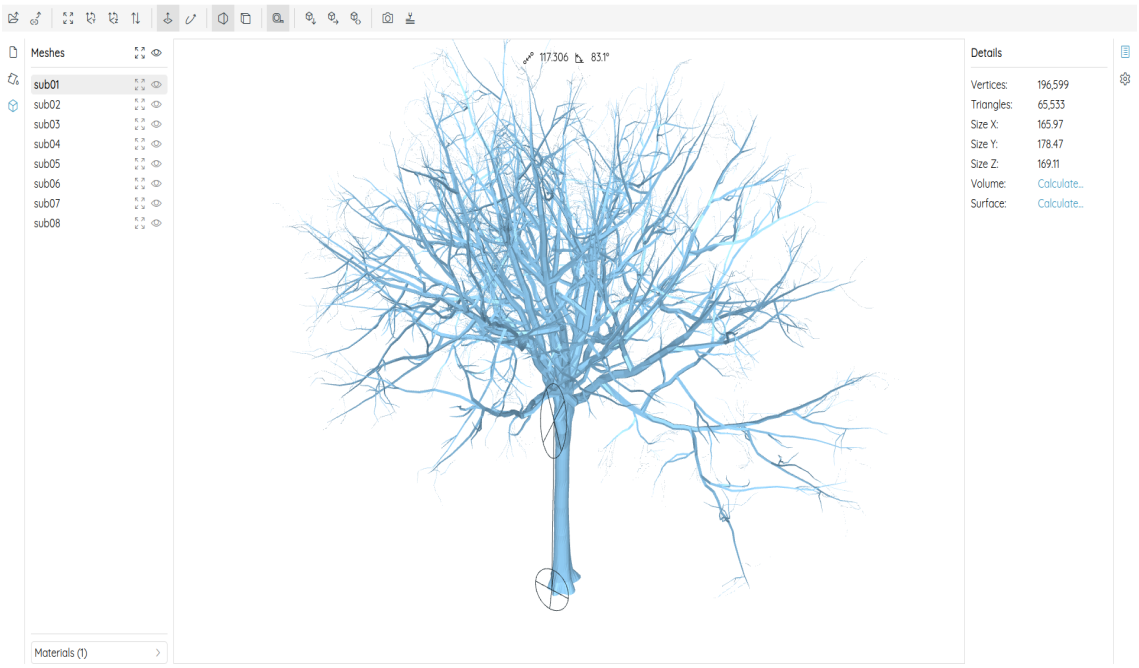
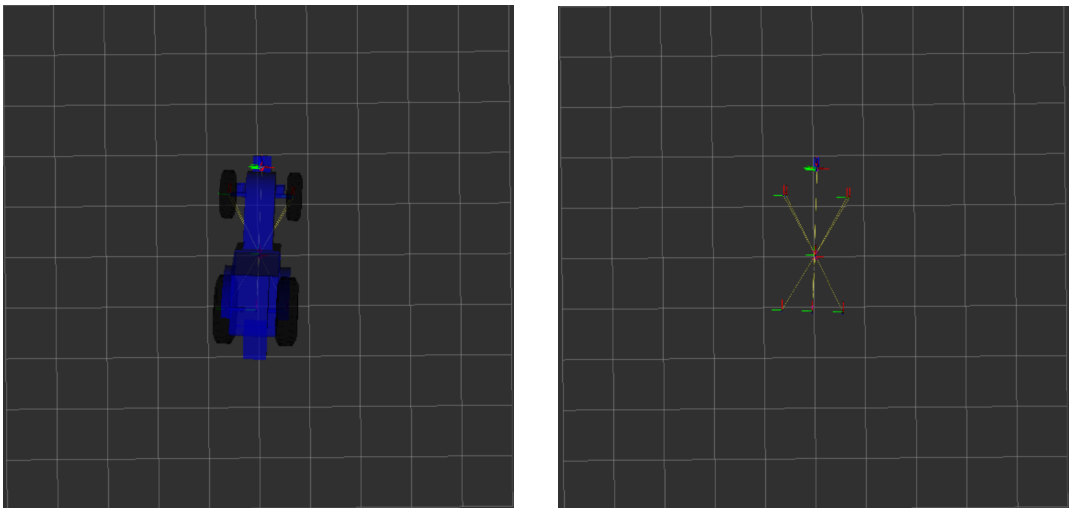


Figure 3.1: Apple tree model

3.2.4 Robot model in Gazebo

A tractor model is used in Gazebo, which has a LIDAR and camera mounted. The camera is an RGB-D camera that has both color and depth images. The RViz window in Figure 3.2 shows the robot model and the frames.



(a) Robot description RViz (b) Robot frames RViz

Figure 3.2: Robot model and frame visualization

3.2.5 Tree detection using Detectron2

The pre-trained weights provided by [2] will be used for tree detections. We will not train the model as the results described in [2] are pretty good for our use case. The weights are compatible with Detectron2, and the pre-trained weights are provided for different backbone networks. We will use the backbone network that provides the most FPS, i.e., ResNet-50-FPN. So we will use the ResNet-50-FPN pre-trained weights and try to optimize the model further using OpenVino and TensorRT.

The model performs keypoint detection, which we can use to estimate the location of the trees through the depth image. The keypoint detection can be performed on color or depth images. As the model performs better on colored images, we will be doing the inference on colored images and then mapping those points to the depth images.

3.2.6 Color image to depth image correspondence

Based on the keypoints detected in the RGB image using the pre-trained weights, we can estimate the location of the trees by mapping the same keypoints from RGB to the depth image. The article [19] provides a solution using the intrinsic camera matrices. As we have the camera info in the camera topics using ROS, we can easily do color-to-depth or depth-to-color correspondence. Mainly, if K_c and K_d are

intrinsic matrices for color and depth cameras respectively and T_{dc} represents the transformation between them, a point in color image $[u_c\, v_c\, Z_c]$ can be mapped to its corresponding depth pixel $[u_d\, v_d\, Z_d]$ using Equation 3.1:

$$Z_d[u_d\, v_d\, 1] = K_d * T_{dc} * K_c^{-1} * Z_c[u_c\, v_c\, 1] \tag{3.1}$$

Our case has no rotation or translation, as one camera provides color and depth images. Hence the transformation matrix will be an Identity matrix, and the equation simplifies to

$$Z_d[u_d\, v_d\, 1] = K_d * K_c^{-1} * Z_c[u_c\, v_c\, 1] \tag{3.2}$$

3.2.7 Estimating tree position

As we now have the depth pixels, which give us the distance of the point from the camera frame in millimeters, we can easily get the distance in meters. The model outputs five keypoints for each tree trunk. Three keypoints are horizontal along the base of the trunk. The rest of the points are vertical along the length of the trunk. We can use the middle keypoint at the base and the two keypoints along the trunk to better estimate tree location. One solution is to take the average of the three keypoints, but we also want to remove outliers and fuzzy detections.

2D to 3D point conversion

The 2D detected points in the depth image are first converted to 3D points using the intrinsic parameters of the camera. To convert the points, the following equations are used

$$Z = z = depth\_image(x, y) \tag{3.3}$$

$$X = \frac{(x - c_x) * z}{f_x} \tag{3.4}$$

$$Y = \frac{(y - c_y) * z}{f_y} \tag{3.5}$$

X, Y, and Z are points in 3D, x and y are the 2D image pixels, and z is the depth value in millimeters.

3D keypoints to distance and angle

The 3D Keypoints mapped onto the depth image can extract the distance and angle of the measured landmark from the camera. To make our detections more

robust, we will measure the distance from the camera to the line generated by the three points in 3D space instead of taking the average of the three points. The line can be used to trim out the outliers.

First, we need to fit a line to the 3D points, which we can do by centering our points around the origin, i.e., (0, 0, 0) by subtracting the mean of the points from each point. Now we need its direction and a point that goes through the line. We can take Singular Value Decomposition and use the V^* matrix to get the eigenvectors. The first eigenvector is the best representation of the given data, which is also the line that will fit the 3D points. Let x be the 3D points and $\bar{x}$ the mean of all the points.

$$U\Sigma V^* = SVD(x - \bar{x}) \tag{3.6}$$

$$p(t) = p_0 + V^*[0] * t \tag{3.7}$$

Where t can be any real number. To project the robot's origin on the line and find the distance and angle from the robot to the landmark, consider Figure 3.3

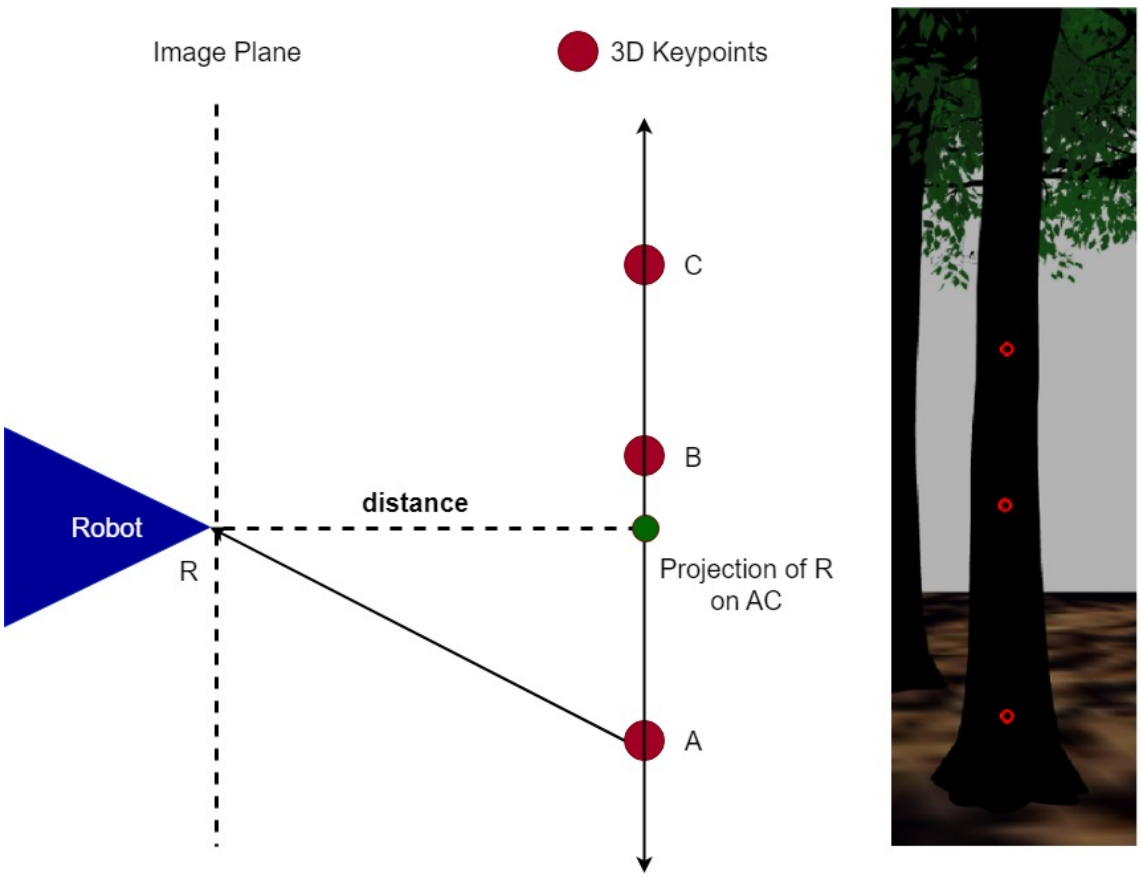


Figure 3.3: Projection of robot's origin on the line connecting 3D keypoints

We need to calculate the projection point, i.e., the green circle from Figure 3.3,

and then take the Euclidean distance from the projection to the robot's origin. Consider $\vec{AR}$ as $\vec{a}$ and $\vec{AC}$ as $\vec{b}$ then

$$\vec{a}.\vec{b} = |\vec{a}||\vec{b}|cos(\theta) \tag{3.8}$$

Then the projection of $\vec{a}$ on $\vec{b}$ is just [20]

$$|\vec{a}|cos(\theta) = \frac{\vec{a}.\vec{b}}{|\vec{b}|} = \frac{\vec{b}}{|\vec{b}|}.\vec{a} \tag{3.9}$$

The resultant is just a scalar which gives us the distance along $\vec{AC}$. For the full projection vector, we need to multiply by the unit vector as

$$|\vec{a}|cos(\theta) = \frac{\vec{b}}{|\vec{b}|}.\vec{a}.(\frac{\vec{b}}{|\vec{b}|}) = \vec{a}.\vec{b}\frac{\vec{b}}{|\vec{b}|^2} \tag{3.10}$$

This is the projected vector. To get the projected point, we need to add the resultant vector to point A. Let P be the projected point in 3D, then

$$P = A + \vec{a}.\vec{b}\frac{\vec{b}}{|\vec{b}|^2} \tag{3.11}$$

Now that we have the projected point, the distance and angle are easy to calculate, Let $R = (x_1, y_1, z_1)$ and $P = (x_2, y_2, z_2)$

$$distance = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2} \tag{3.12}$$

To calculate the angle, consider the camera frame in Figure 3.4

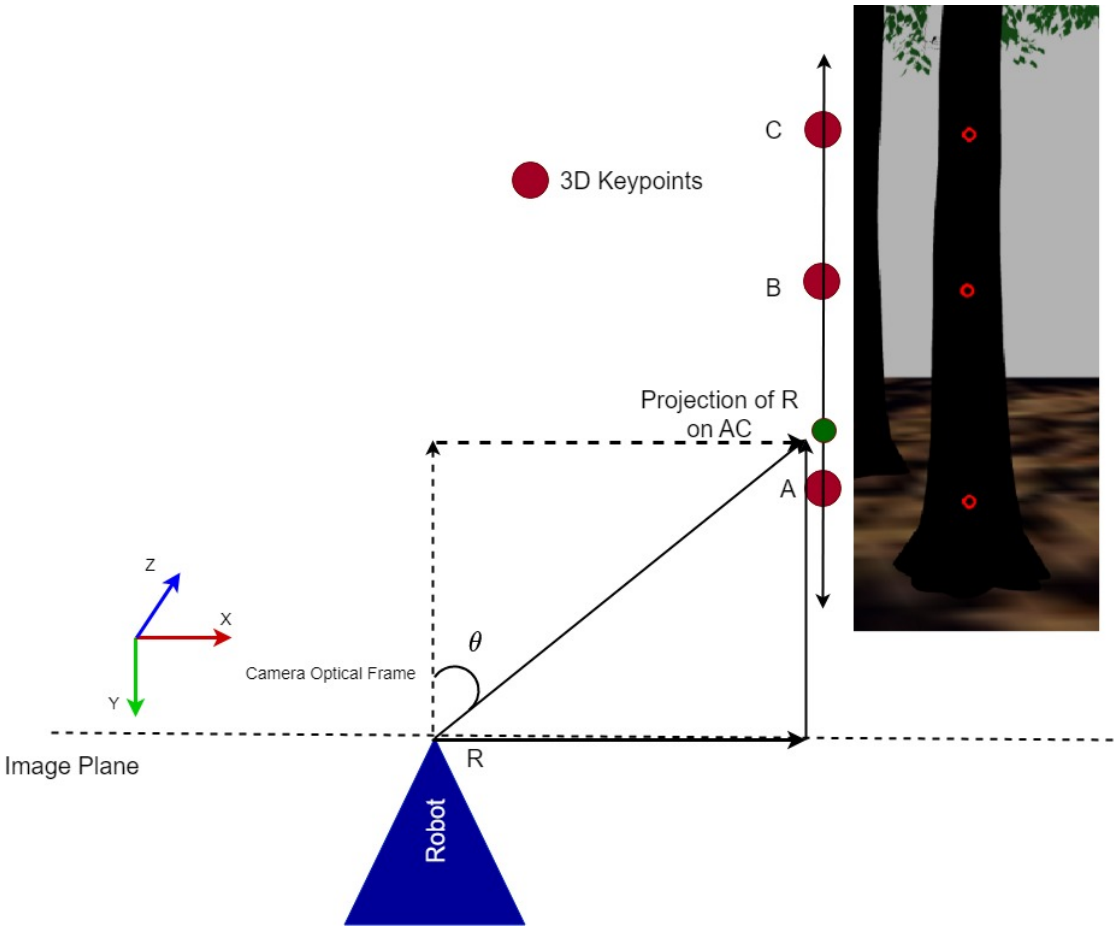


Figure 3.4: Camera optical frame for angle calculation

If projected point $P = (x_2, y_2, z_2)$ then the angle can then be calculated as

$$\theta = atan2(\frac{-x_2}{z_2}) \tag{3.13}$$

Convert landmarks from camera to world frame

Now that we have the distance from the camera plane and the angle from the robot to the landmark, we need to convert the landmark position from the camera frame to the world frame. For this, we will use the transformations package provided by ROS2 to apply transformations from the camera and robot frame to the world (Odom) frame. From 3.4, we can see that we need to find $\vec{RA}$, which gives us the actual distance of the landmark from the robot. We can compute $\vec{RA}$ easily using 3.14 and 3.15

$$camera_x = distance * \cos\theta \tag{3.14}$$

$$camera_y = distance * \sin\theta \tag{3.15}$$

This is still the distance from the camera to the tree. Using the tf2_ros package, we can use the lookup_transform() function to get the transformation from the camera optical frame to Odom (world) frame. Consider t_{cw} to be the translation from camera to world frame. Also, we need to consider the robot's orientation with respect to the world frame. This can be done by calculating the rotation matrix from the robot to the world frame. Let R_{rw} be this rotation matrix. The yaw angle can be found from the rotation matrix as:

$$\alpha = tan^{-1}(\frac{r_{21}}{r_{11}}) \tag{3.16}$$

The final equation to find the world coordinate is given in 3.17 and 3.18

$$world_x = t_{cw}.x + distance * \cos(\theta + \alpha) \tag{3.17}$$

$$world_y = t_{cw}.y + distance * \sin(\theta + \alpha) \tag{3.18}$$

3.2.8 Optimize for CPU - OpenVino

The Detectron2 Keypoint Detection works pretty well with the pre-trained weights of [2], but as we need the solution to work on embedded boards, we will try to optimize the model as much as possible. The embedded board currently present on the robot does not have a GPU. So, our first approach is to optimize it for CPU processing and check if the inference time is good enough for the navigation to work. We aim to achieve at least 5 FPS on inference to run the navigation stack at 5 Hz.

To achieve better inference on Intel-based CPUs, we will be using the OpenVino platform. OpenVino is a toolkit provided by Intel to optimize deep learning models from almost any framework and perform faster inference on Intel-based hardware. It does so by using load balancing, memory reuse, graph fusion, and parallelism of inferencing across Intel CPU, GPU, and VPU, etc.

OpenVino Model Optimizer

To convert to OpenVINO, we first need to convert our model to Open Neural Network Exchange (ONNX) format. Facebook provides this ONNX conversion for any Detectron2 module, including Keypoint Detection. The out-of-the-box support for ONNX conversion makes Detectron2 an ideal choice for our use case. Facebook has provided a script, **export_model.py**, inside the official Detectron2 repository on GitHub. We can run the script with our pre-trained weights and convert it

to ONNX format. Check Code A.1 to convert the Detectron2 Keypoint model to ONNX. We must also provide a sample image of the data where we will run the inference. The dimensions of the image matter for optimizing the model. The script will generate an ONNX format output which can easily be converted to OpenVINO IR (Intermediate Representation) format.

Once the ONNX file is ready, we can use OpenVino's built-in Model Optimizer tool to convert ONNX to OpenVINO IR format. We can also change the precision of the output to fp16, which corresponds to 16-bit floating point precision. This will further optimize our model. The Model Optimizer will output two files which can then be used by OpenVino Runtime.

OpenVino Runtime

OpenVino Runtime is a library built on C++, and it provides support for both C++ and Python to run inference on any platform. With the conversion to OpenVino IR format, we can remove the model's dependency on the Detectron2 module, as it can run without using the Detectron2 engine for inference. We will load our network in OpenVino Runtime and preprocess our image before feeding it to the compiled IR model. We will do a performance comparison of the Detectron2 engine vs. the OpenVino Runtime in the Results section.

OpenVino GPU

We can also configure OpenVino Runtime to use Intel GPUs to run inference. But, as our Detectron2 model contains some unsupported layers for GPU, such as Non-Maximum Suppression Layers, we cannot run the model for Intel-based GPU.

3.2.9 Optimize for GPU - TensorRT

We will also be optimizing the Detectron2 Keypoint Detection model for NVIDIA-based GPUs. The embedded board on the robot currently does not have a GPU, but we can attach a Jetson device if needed. The make our model work on the Jetson device with optimal inference time, we will be converting the model to TensorRT format. In the same way that Intel published OpenVino to provide optimal inference for deep learning models on their hardware, NVIDIA has published

the TensorRT framework to run inference on their hardware, especially the CUDA cores. TensorRT is highly optimized and one of the fastest ways to do inference.

The TensorRT compiler compresses a model to run faster and use less memory. The compressed model has almost the same accuracy as before, with faster and more efficient memory allocation. This is possible as the compiler evaluates and fixes the computation graph and runs faster inference. It also combines some of the operations together, e.g., convolution and activation layers.

Conversion to ONNX format

Like OpenVino, TensorRT requires the Detectron2 model to be converted to ONNX format. For this, we can use the same script as before, provided by the Facebook team itself, to convert to ONNX. But, if we remember from last time, some layers are not supported on GPUs, such as the Non-Maximum Suppression Layer. So conversion to ONNX works fine, but converting it to TensorRT engine format will fail. To handle these cases, NVIDIA has provided an official example to support the conversion of the Detectron2 Mask RCNN ResNet-50-FPN model to TensorRT.

We are also using the ResNet-50-FPN backbone for our case but we are not using Mask RCNN. We are doing Keypoint Detection, and NVIDIA does not yet support the Detectron2 Keypoint Detection conversion. We will implement this conversion and test against the original model for inference time.

Create ONNX graph

To get the ONNX format suitable for TensorRT conversion, we will start the same way as we did for OpenVino. After converting the Detectron2 model to ONNX format using the official script provided by the Detectron2 team, we will generate an ONNX graph format suitable for TensorRT conversion. The sample image we are providing for simulation has dimensions of 1280x800. Anything having a width of less than 800 is not supported by Detectron2 ONNX conversion. We will remove this limit from the script as, for our real use case, we receive images of 640x480.

Optimizations in the ONNX file

After we have the ONNX format, we will convert the ONNX to Graph ONNX, which will replace and remove some layers from the original ONNX file that TensorRT does not support. The official example from NVIDIA does not work for our case as we are using Keypoint Detection, so we will be taking help from onnx-graphsurgeon. It is a convenient tool for modifying ONNX graphs. We will extend the official script for Mask RCNN to work with Keypoint Detection. We are removing any unconnected nodes and doing constant folding inside the script. We also remove all the pre-processing nodes other than image normalization in the ONNX graph. As Non-Maximum Suppression is not supported, GenerateProposals and BoxWithNMSLimit layers are replaced with Efficient NMS TensorRT. The Efficient NMS is a plugin by NVIDIA for faster non-maximum suppression. But to replace it with efficient NMS, we require anchors from feature maps of the Detectron2 model. The exported ONNX file from Detectron2 does not have these anchors, so we will also extract them offline. The Region of Interest Align layer is replaced with the TensorRT plugin called PyramidROIAlign_TRT. This, along with some other optimizations, removes a lot of unnecessary layers and replaces some of them with their efficient TensorRT version.

MaskRCNN to Keypoint Detection in ONNX graph

To change the output layers, we will remove the output layers for MaskRCNN and replace them with Keypoint Detection layers. We will extract the last Convolution layer which is actually a ConvTranspose layer, and output it as it is from the ONNX graph. We will later apply some processing during the TensorRT inference engine on this Convolution layer to extract keypoints. The final ConvTranspose, as seen in Figure 3.5 outputs a torch of size 100x5x28x28 where 100 is the maximum number of detections the model can perform, 5 is the number of keypoints to be detected, and 28x28 is the size of output mask. We will process this mask along with bounding boxes to get the keypoints.

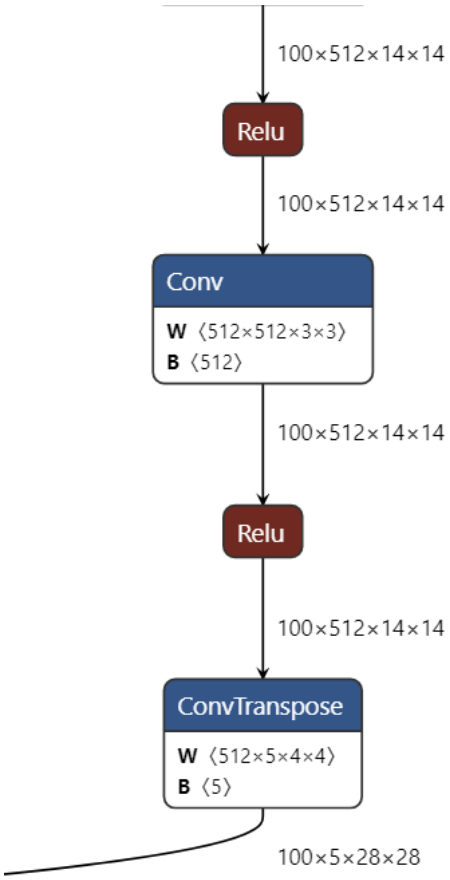


Figure 3.5: Converted graph ONNX final ConvTranspose layer

There are certain limitations, such as fixed input image size and the number of detections which will be discussed in the Limitations section.

Build TensorRT Engine

Now that we have the converted ONNX file suitable for TensorRT, we can easily convert the model to TensorRT. We can do it using the trtexec tool provided by TensorRT and give all the required parameters, or the NVIDIA team has provided a script, build_engine.py, in their official GitHub repository for building the converted ONNX to TensorRT with optimal parameters. We can change the precision to 16-bit floating point (fp16) or 8-bit integer(int8).

The int8 precision is supposed to work much faster than the fp16 or the original 32-bit floating point precision, but we will check for the trade-off between precision and speed and decide which one is better to use. For int8 precision, we also have to provide calibration files, which are basically sample images. It can be training or validation images as well. The more images for calibration, the better the precision.

After conversion to the TensorRT engine, we can run inference using the inference plugins provided by TensorRT. But as mentioned before, we need to do some processing during the inference for our model to work with Keypoint Detection.

TensorRT inference

TensorRT provides plugins to run inference in Python. We will use those plugins but extract different output layers, such as bounding boxes, masks, and scores. From the scores, we will test if the confidence level is higher than a certain level. From the bounding boxes and masks, we will extract keypoints. We will take the heatmap, i.e., mask values inside the bounding box, interpolate the values using the PyTorch interpolate() function and take keypoints as mid values of the heatmap. Also, the conversion from discrete pixel to continuous coordinate is done using [21]

$$c = d + 0.5 \tag{3.19}$$

Where c is continuous, and d is a discrete coordinate. We are also clipping the values of bounding boxes based on the image height and width.

3.2.10 Reactive navigation

After generating optimized models for tree detection and having keypoint location in the world frame, we can use it for navigation. We are doing reactive navigation, which means that we will perform navigation based on the current or most recent detected state from the sensors. We will not store the previous state anywhere and navigate the agriculture field reactively. For this, we will be using Pure Pursuit Algorithm.

Generating path

The pure pursuit algorithm takes a path as input and generates velocity commands. We will first look into how we can generate a path so robots can easily go through the fields. For this consider Figure 3.6

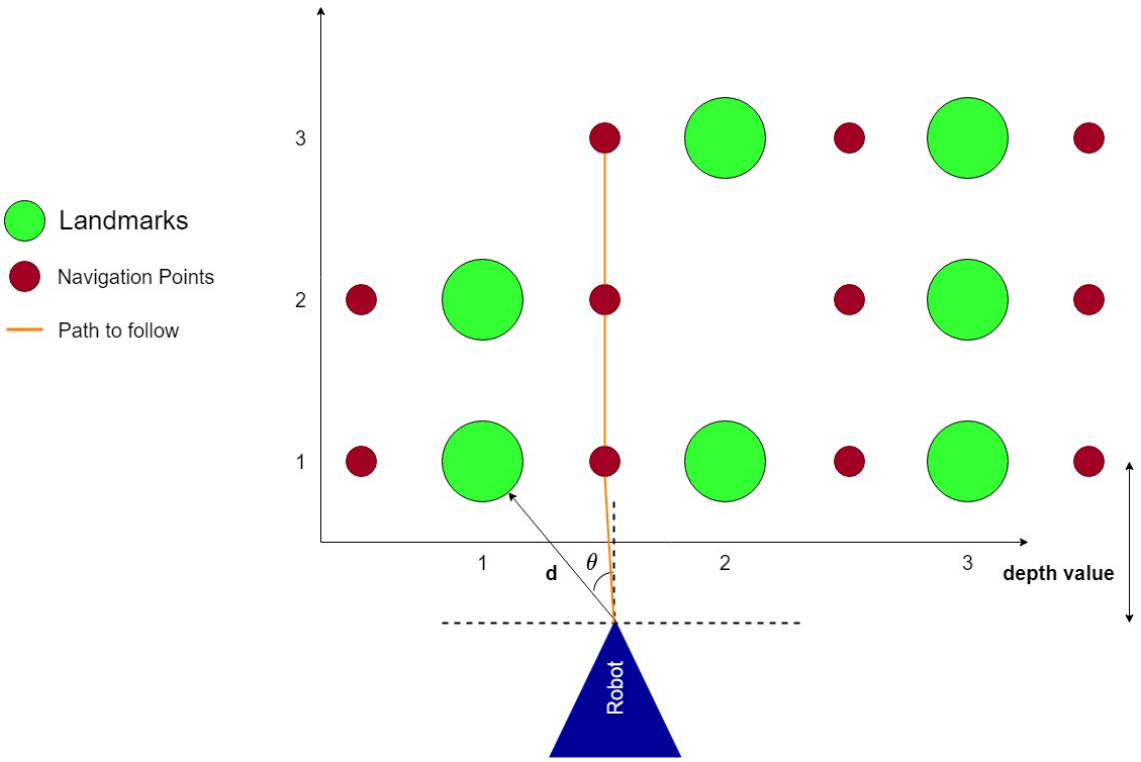


Figure 3.6: Generate path points based on detected landmarks

From the figure, we can see that the robot observes some landmarks (colored in green), and we know the world coordinates of these landmarks through Equations 3.17 and 3.18. We are assuming that we know the width of lanes. So, we will generate two navigation points for each landmark, colored in red, at half of the width distance on the left and right sides of the landmark. If we detect two landmarks at the same distance from the robot and their relative distance is close to the width of the lane, we will update the navigation point between them using the average of both landmark points. This will update the location of the navigation point in real time based on the position of landmarks.

Once we have the navigation points, we will check the angle of each navigation point relative to the robot. As we already know the current heading of the robot, we can check which navigation points are in direct sight of the robot. If the angle of the point with respect to the robot is below some threshold, e.g., 10°, then it is added to the generated path. This is shown in Figure 3.7

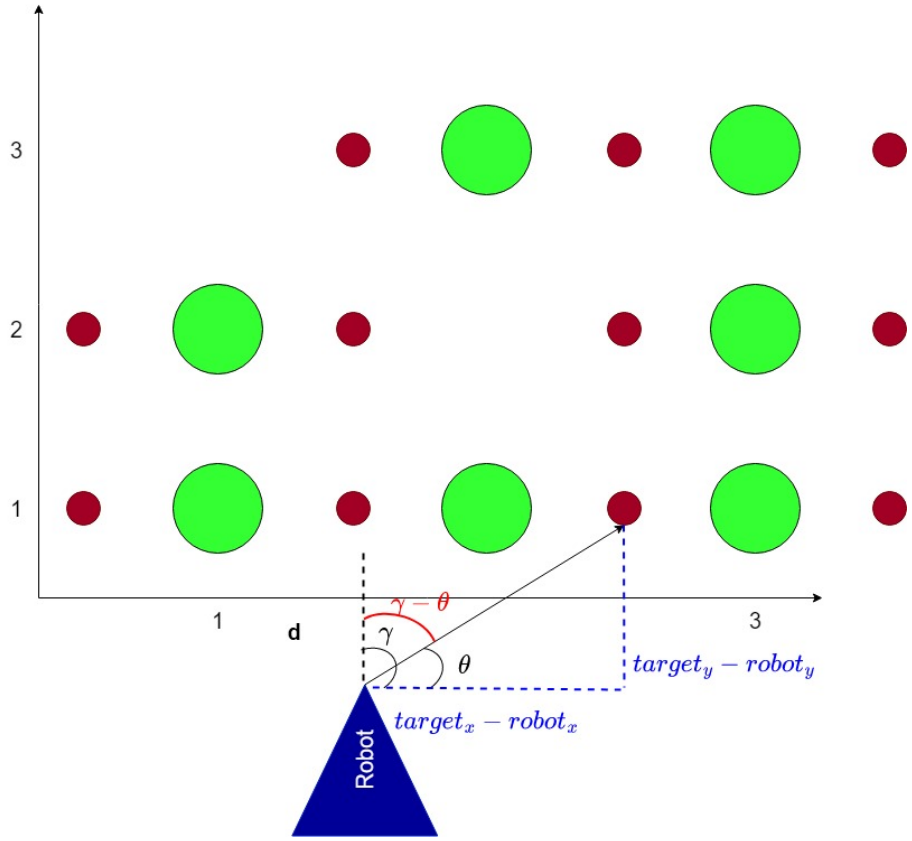


Figure 3.7: Detect navigation points in the same direction

The angle γ is already known as it is the current heading of the robot. We also know the world coordinates of both robot and the navigation point and can easily calculate θ. Hence we can apply some threshold value and only take points that are almost in the same direction as the robot. We will then sort these points based on their distance from the robot. This generated path, shown in orange color in Figure 3.6, is then passed to the pure pursuit algorithm that we will go over in the next section.

Lane change

Before going over the pure pursuit algorithm, we also need to define a maneuver path so the robot is able to change the lane and traverse through the next lane. For this, we again use the pure pursuit algorithm and we define some custom points in the path that the robot follows and is able to align itself with the next row. Consider Figure 3.8

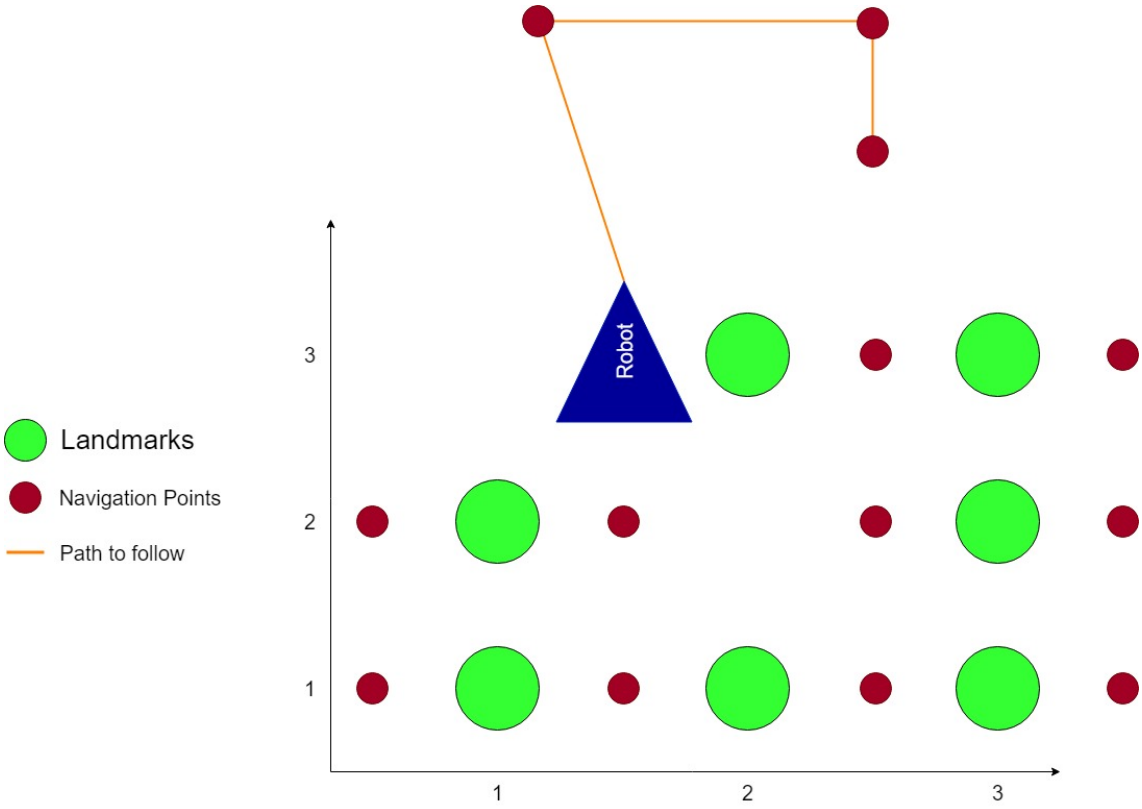


Figure 3.8: Lane change maneuver

We are generating three points in front of the robot that the robot follows using pure pursuit and changes the lane.

Pure pursuit algorithm

The pure pursuit algorithm is used to get the robot to follow a path by calculating the distance to the goal point and navigating to it by following an arc path. It makes the vehicle move in a curved path to reach the goal position. It also uses a look-ahead distance that measures the next goal of the robot. The point in the path that is closest to the look-ahead distance is taken as the next goal and the robot follows a curvature to reach it. It is called pure pursuit as the robot is in pursuit of the look-ahead distance. Consider Figure 3.9

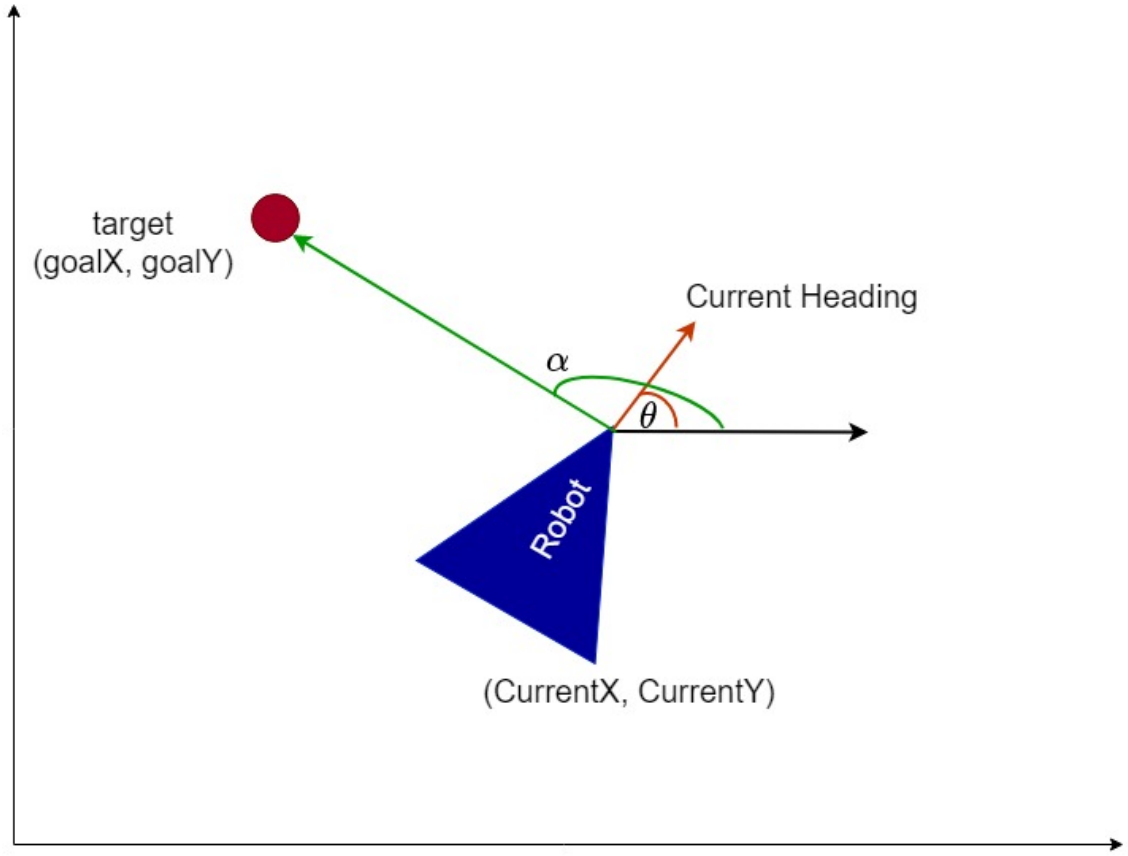


Figure 3.9: Robot pose and target position

From the figure, we can find the steering angle using Equation 3.20

$$\gamma = atan2(\frac{targetY - currentY}{targetX - currentX}) - \theta \tag{3.20}$$

We can find the distance between the points using the simple Euclidean distance and based on the look ahead, we will decide if it is the closest point. If the point is the closest, we will generate a pre-defined constant velocity along with the desired steering angle and pass it to the command velocity of the robot.

3.3 Limitations and assumptions

As we mentioned before, some of the limitations and assumptions are:

- The camera should be placed at a small height from the robot so the robot is able to see the tree trunks or strawberry poles easily.

- The width of the lane should be known beforehand so we can generate better navigation points. Also, the robot should be in line with the lane before we run the perception and navigation modules.
- The TensorRT engine model should be built on the same device where inference is to be done.
- The model optimized by TensorRT has a fixed input size. If the image size changes by using a different camera, we need to re-optimize our model by passing the sample image from the new camera.

3.4 Conceptual diagram

A visual representation of the conceptual diagram is depicted in Figure 3.10

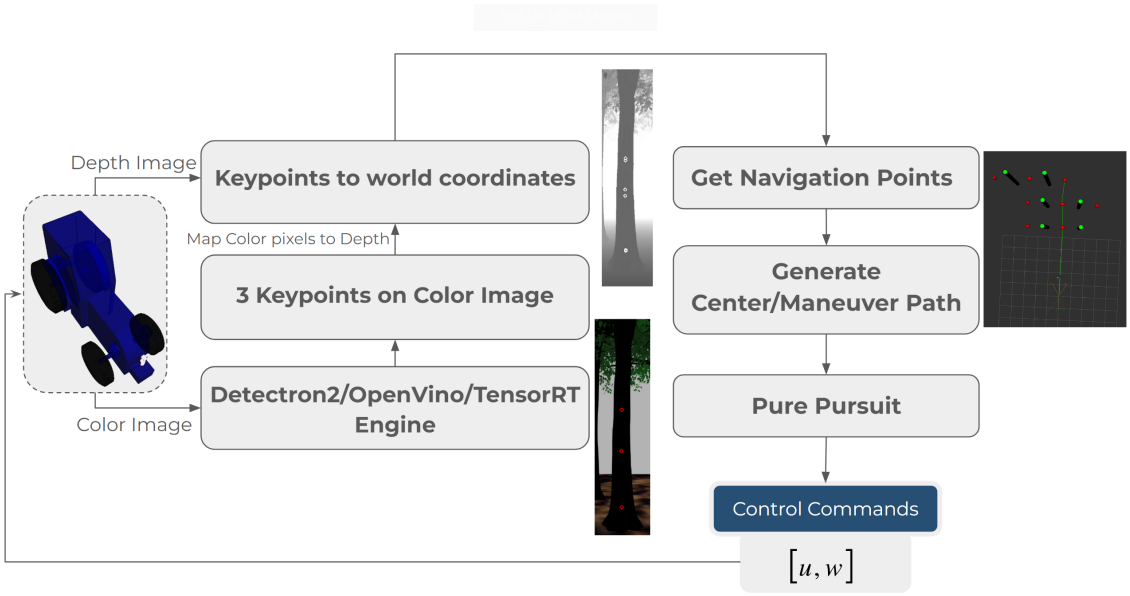


Figure 3.10: Conceptual diagram of the methodology

Chapter 4

Results

4.1 Simulation environment

4.1.1 Generating Gazebo world

We need to generate our simulation environment precisely so we can perform tests before going to the real fields. The Python script generates a custom number of rows and trees in each row at specific distances. It is shown along with the robot model in Figure 4.1

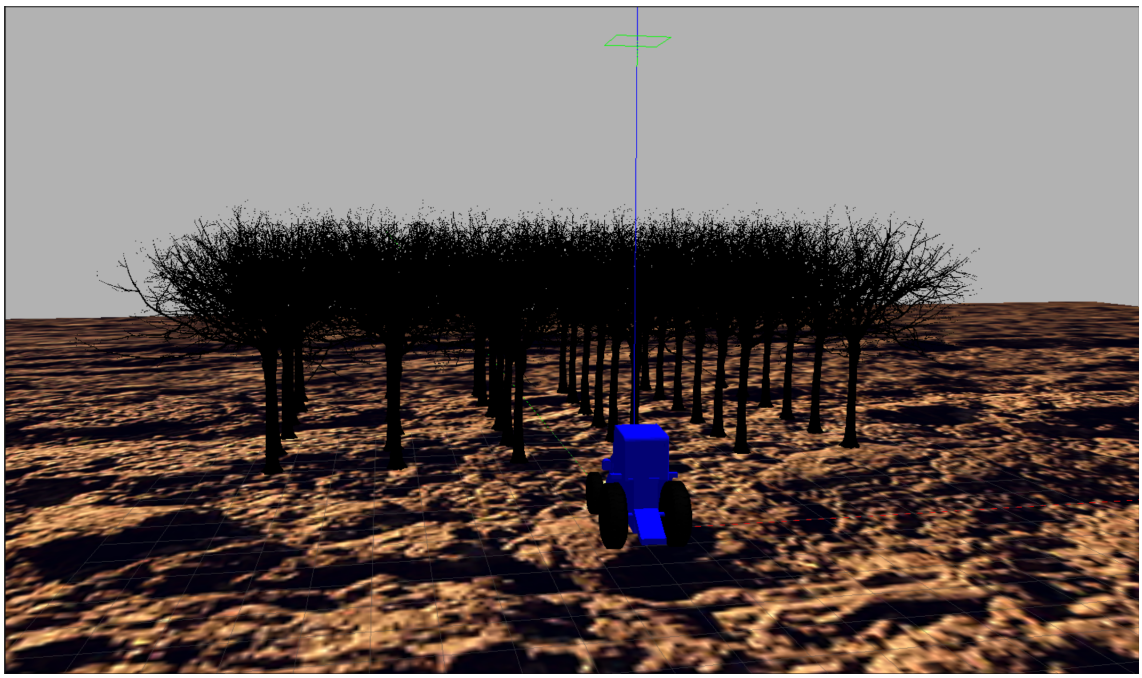


Figure 4.1: Gazebo environment with the robot and trees

The robot model is correctly generated with the correct transformation tree. The

robot is also equipped with a camera and LiDAR. Figure 4.2 shows the output of the camera and LiDAR when placed in the tree environment.

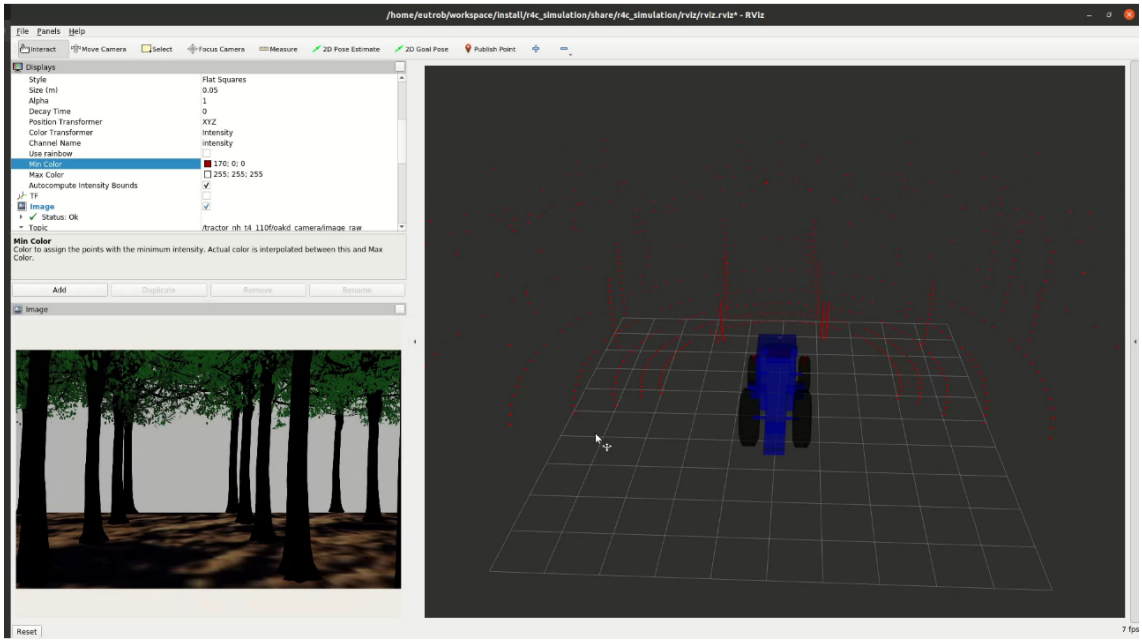


Figure 4.2: Camera and LiDAR output in RViz

4.2 Tree detection using deep learning

Tree detection is working pretty well on color images in the simulation environment. The confidence score for trunk detection is above 90% as we are not taking anything below 90% confidence into account.

4.2.1 Color image to depth image mapping

The keypoints are detected on the trunks of the trees as shown in Figure 4.3

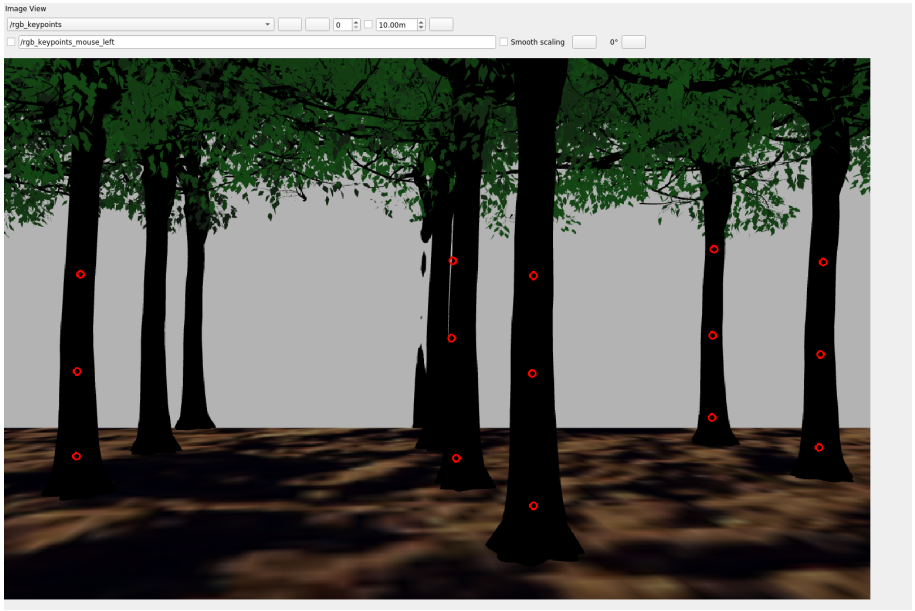


Figure 4.3: Keypoint detection on RGB image

These keypoints are later mapped on the depth image and as evident from Figure 4.4, they are in sync with each other.

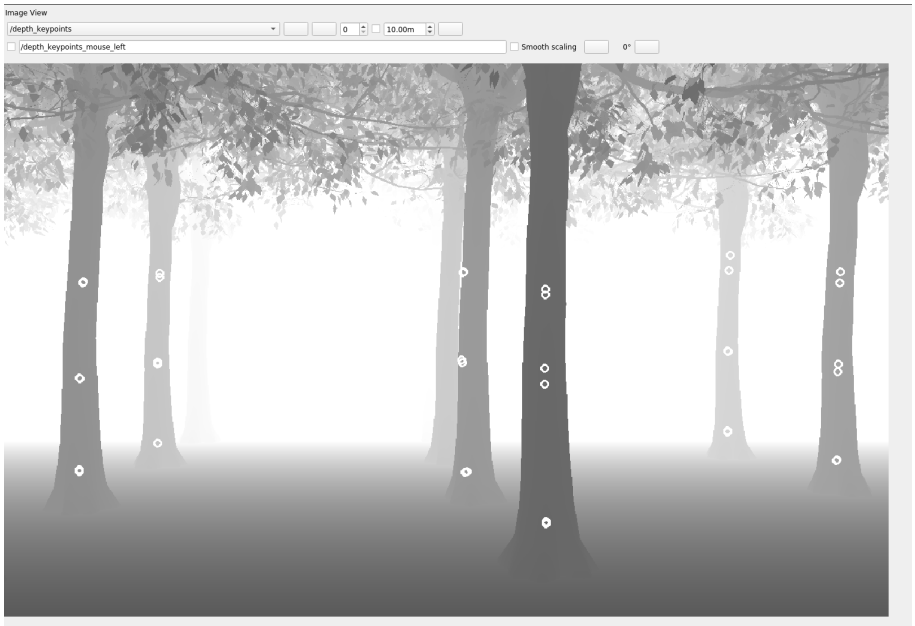


Figure 4.4: Keypoint detection on depth image

4.2.2 Precision of tree detection

The location estimation of trees is quite accurate in Gazebo. The first row is placed at a distance of six meters from the center of the robot. Figure 4.5 shows the

RViz display of tree detection. Each grid corresponds to one meter. The measured distance is almost six meters from the robot.

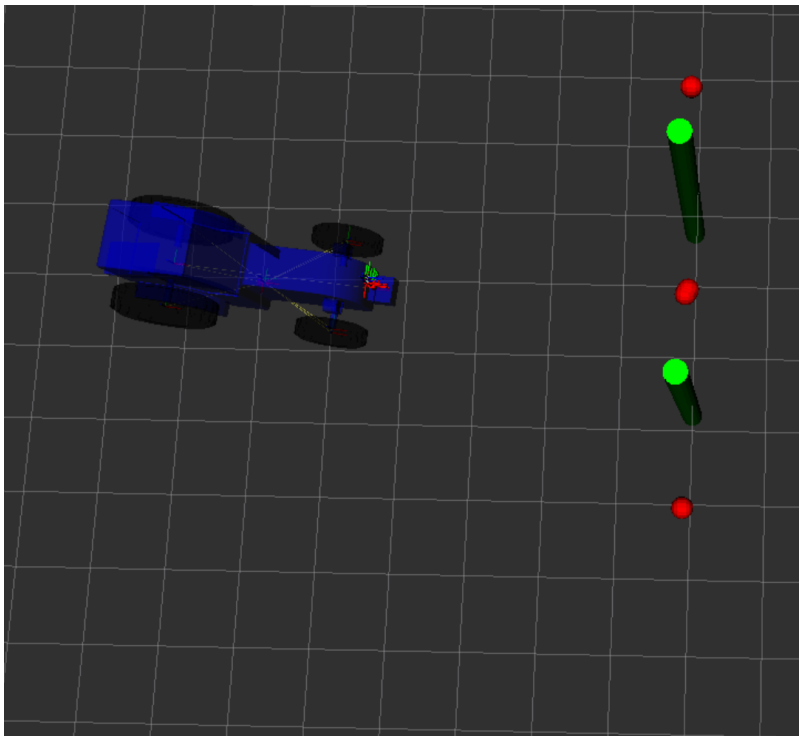


Figure 4.5: Estimated distance of six meters from the robot to the tree

4.2.3 Speed of tree detection

CPU inference time

As the embedded board on the robot does not have a GPU, we started with CPU inference of the Detectron2 model. The keypoints are detected fine using the CPU, but they are extremely slow. The inference for one frame took around 20 seconds with Gazebo and RViz.

We ran the gazebo in headless mode, and the inference time improved, and one frame now takes around 6 seconds. The need for OpenVino is evident from these findings. After converting the model to OpenVino, we ran it using the OpenVino Runtime and observed that the inference time had improved by a factor of 2.5x. The time taken by one frame dropped to 2.4 seconds. Although we are seeing much improvement, this is extremely slow for any reasonable results for robotic navigation. The 2.4-second inference time corresponds to 0.41 FPS, and our minimum is to obtain 5 FPS.

GPU inference time

We have to use GPU to obtain meaningful results. We will use a laptop fitted with NVIDIA RTX 3060 Laptop GPU with 6GB memory. Running the inference model without optimization takes around 0.12 seconds for one frame. This corresponds to 8 FPS. The FPS is good enough for navigation but the GPU is very powerful and we ideally want 5 FPS on a Jetson device. So we will try to optimize this time as much as possible. The FPS is obtained in a simulation environment where the camera gives us an image of 1280x800 size.

Using TensorRT to optimize the inference model and runtime, we ran our code with 16-bit floating point and 8-bit integer precision. The inference improved by a factor of 2x. With fp16 precision, the inference time is 0.055 seconds, giving us 18 FPS. We also traversed through the whole field and kept every detection as a marker in RViz to visualize the detection as shown in Figure 4.6

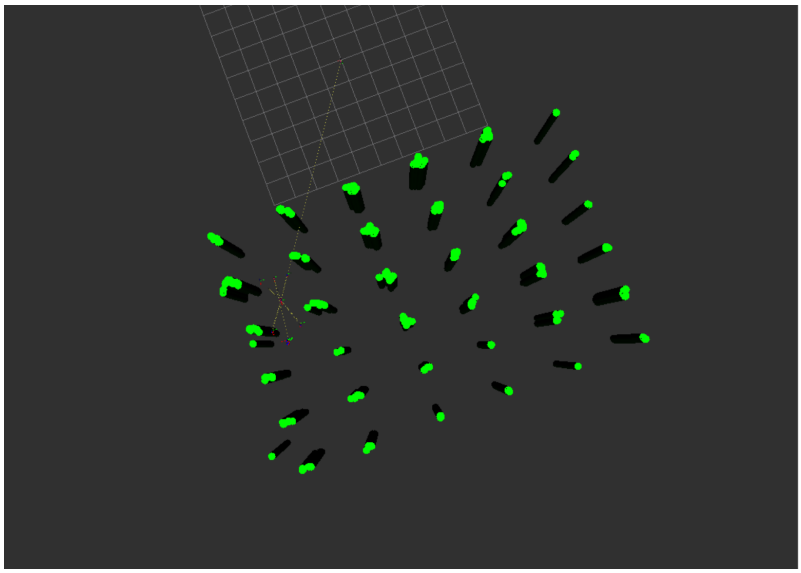


Figure 4.6: Tree detection using 16-bit floating point precision

Using int8 precision, it improves to 20 FPS. The precision is almost the same as with 16-bit precision, so it is better to use int8 precision. The results obtained with int8 precision are depicted in Figure 4.7

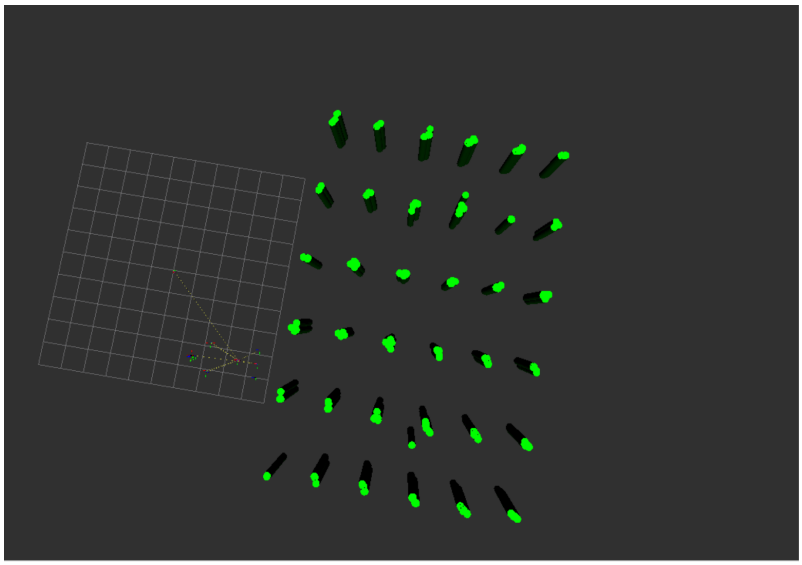


Figure 4.7: Tree detection using 8-bit integer precision

4.2.4 Lane center and change maneuver

We are generating the path as described in the methodology section above. We take the robot's current heading and only take the navigation points that exist in the line of sight of the robot. We then sort the points based on their distance from the robot and generate a path. The generated lane center is shown in Figure 4.8

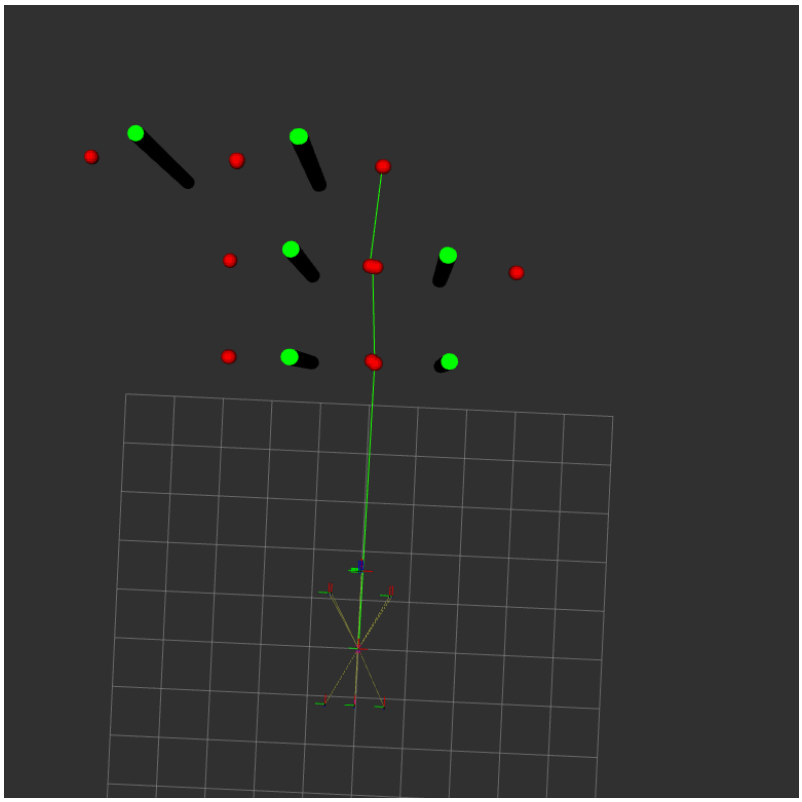


Figure 4.8: Lane center path

After the robot traverses through a lane, we perform the lane change maneuver and generate a new path that the robot follows using the pure pursuit algorithm and changes the lane. The lane change path is shown in Figure 4.9

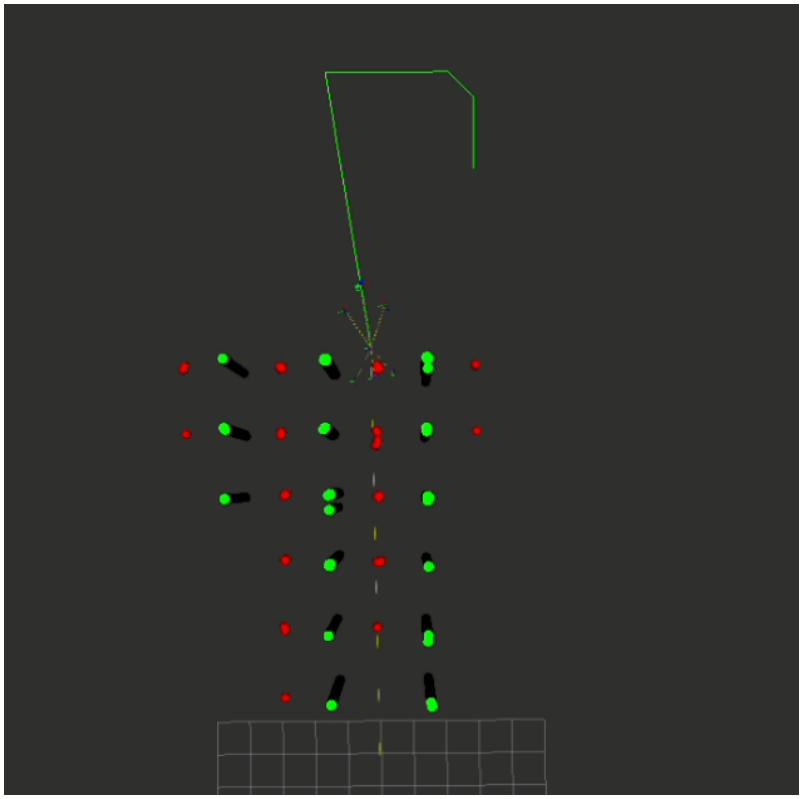


Figure 4.9: Lane change path

After changing the lane, the robot keeps going through lanes and keeps traversing until no more lanes are left. Figure 4.10 shows the reactive navigation in action where the robot is reactively navigating through lanes and changing lanes.

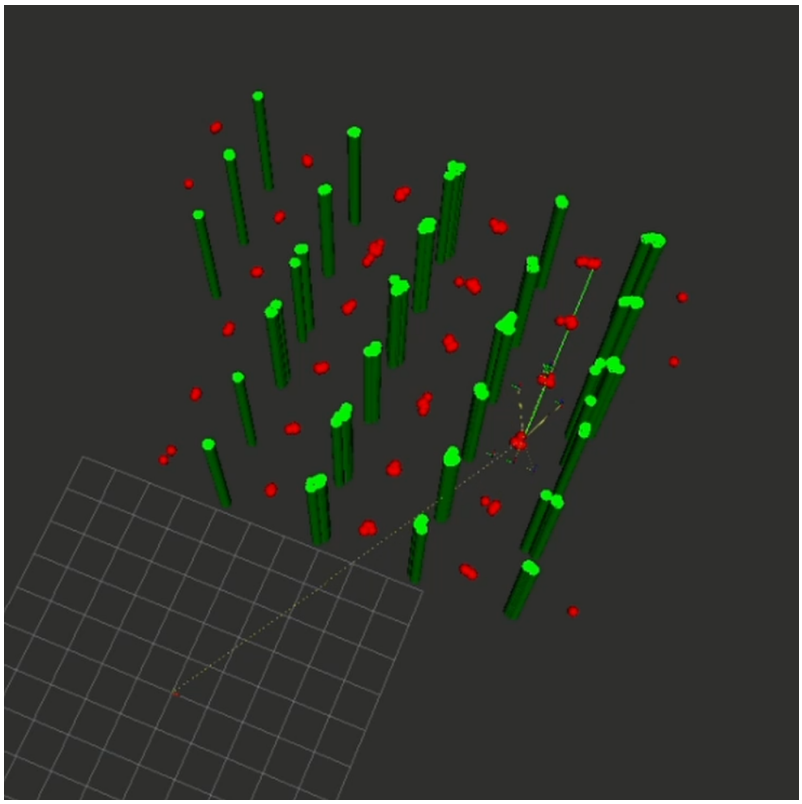


Figure 4.10: Reactive navigation using pure pursuit

4.2.5 Tilted lanes

Our algorithm will still work if the lanes are tilted at some angle. The navigation points that are in direct sight of the robot are taken into account. We calculate the angle using navigation points and robot position. The navigation module will use the generated path to follow it safely in a tilted direction, as shown in Figure 4.11

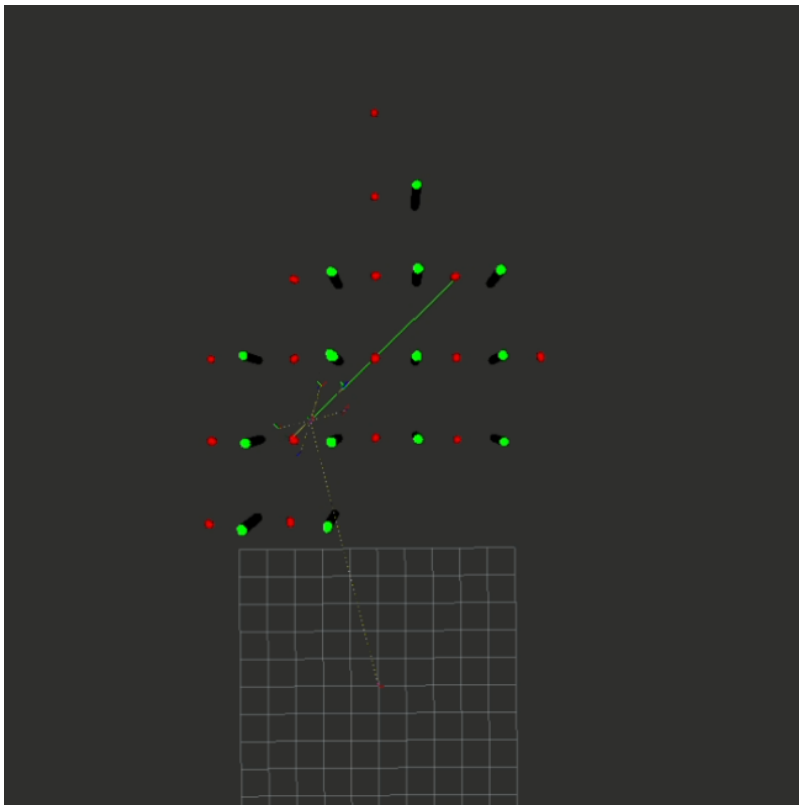


Figure 4.11: Reactive navigation on tilted lanes using pure pursuit

The lane change maneuver will also adapt to the tilted lanes and generate a path that the robot can follow to change lanes. The generated path is shown in 4.12

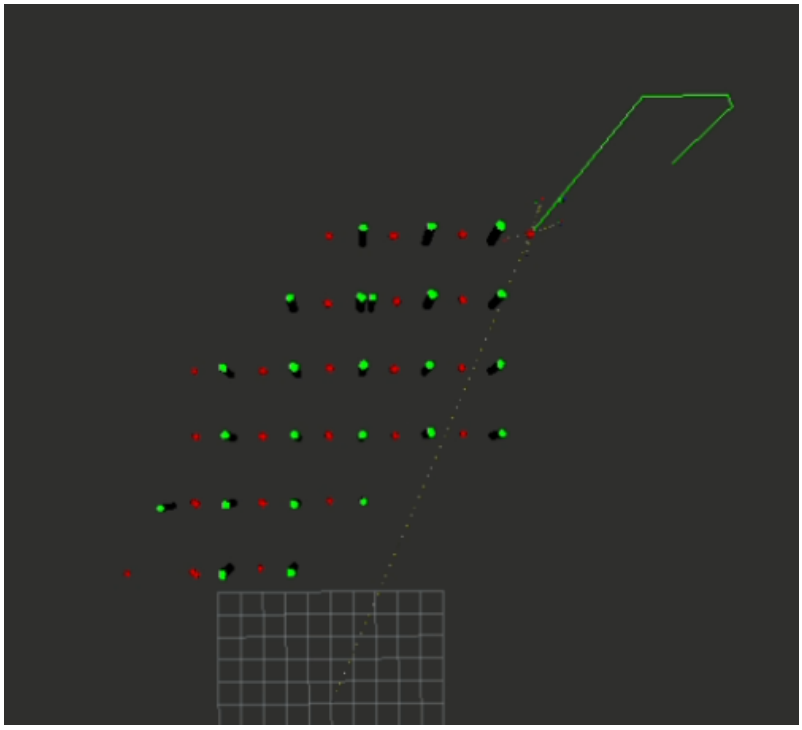


Figure 4.12: Lane changing on tilted lanes

4.3 Real environment

For the real environment, we have collected a rosbag file with the scout robot in Budapest. The robot goes through a path with some trees, and the detection on real trees is shown in Figure 4.13

Figure 4.13: Real tree detection using TensorRT engine

The robot uses a camera that outputs an image of 640x480 size. As the size has changed, we need to rebuild our ONNX file and TensorRT engine and then perform inference. The model is trained for trees, but we can also detect other cylindrical objects, such as in strawberry fields where the strawberries are planted on top of white poles. The model is able to detect some of these poles as shown in Figure 4.14. If the camera is fitted slightly lower, more poles can be detected.

Figure 4.14: Real strawberry pole detection using Detectron2 engine

The inference time in the real environment is even lower as there is no Gazebo overhead and the image resolution is also lower. With Detectron2 and GPU, we are getting around 0.09s inference time. The inference time drops to 0.04s with TensorRT optimization using 16-bit floating point precision. We also tested using 8-bit integer precision on the real environment, and the inference time goes back and forth between 0.05 and 0.025 seconds. It gives us 20-40 FPS. As the 16-bit floating point gives us more stable results in the real environment, we will take 25 FPS as our final optimized result. A summary of all the different inference speeds is presented in Table 4.1

Value	**CPU**		**GPU**		**Real GPU**	
	Detectron	OpenVino	Detectron	TensorRT	Detectron	TensorRT
Inference	6s	2.4s	0.12s	**0.05s**	0.09s	**0.04s**
FPS	0.167	0.41	8.33	**20**	11	**25**

Table 4.1: Inference time and FPS in simulation and real environment

Chapter 5

Discussion

5.1 Interpretation of results

The perception module is working quite well. The confidence score for the trees is above 95%. The only issue was the slow inference time which was improved by converting the model to the TensorRT engine. However, The TensorRT engine is not generic and must be recompiled if the image size changes. For one type of camera, it usually remains the same. The engine should be built on the machine where the inference will be done. But these are not very tiring tasks as we have the script for everything.

The navigation module is also working flawlessly in the simulated environment. We did not have the resources to test it in a real-world field yet. The pure pursuit algorithm works well if the path is correctly generated from the output of the perception module. The camera has to be positioned so that most of the tree trunk is visible for the detection to work. The detection does not only work for trees but for most cylindrical objects. That is why we have added some additional checks, such as checking the inclination angle of three keypoints with the y-axis of the camera.

As we are doing reactive navigation, we do not need to store the previously detected trees in the state. We do not care much about the robot's localization as it reacts to the information in real time and stays in the center of the field. The somewhat controllability of the environment helps us in navigating the fields.

5.2 Comparison with previous research

The authors of [2] mentioned that they achieved 18 FPS with ResNet-50-FPN backbone on a very powerful GPU. Our GPU is not as powerful and has less memory. In our tests, the same model gave around 8 FPS. But by implementing the TensorRT engine for Keypoint Detection, we optimized our model to achieve around 20 FPS with 8-bit integer precision. This FPS goes up to 25 with 16-bit floating point precision in the real world, where the camera gives us a lower resolution image, and there is no Gazebo overhead that takes up processing power. There is still room for improvement as the TensorRT inference is done in Python language, which is not very optimized for running CUDA Graphs. In the future, we can use C++ for inference, and the FPS is expected to increase even further.

5.3 Limitations and future research directions

5.3.1 Limitations

After performing both real-world and simulation tests, we found some limitations of our solution

- The solution works perfectly for tree trunks, but it should ideally work for all situations, such as strawberry fields where the plants are attached on top of poles. The solution does detect these poles but not with a good confidence score.
- The TensorRT engine has to be rebuilt if the image size or device changes.
- The navigation stack assumes that we already have information about the width of the lane, and the robot is centered around the first lane.

5.3.2 Future directions

To overcome these limitations, we will outline some future directions to solve these problems

- The Detectron2 model can be trained using transfer learning for new images of strawberry field poles. The weights will be changed but the conversion to

TensorRT will remain the same. The newly trained model can be easily converted to the optimized TensorRT engine.

- The optimized model should be tested on embedded boards such as Jetson devices to test the inference time.
- The TensorRT engine currently uses Python but should be written in C++ to utilize CUDA Graphs to achieve even more FPS.

Chapter 6

Conclusion

The Detectron2 model, coupled with the pre-trained weights, effectively detected trees and estimated their location in the global frame. However, it was extremely slow and not feasible for robotics navigation. The optimizations performed on the model using the TensorRT engine made it 3 times faster, achieving 25 FPS on real-world tests. We have converted the Detectron2 Keypoint Detection to an optimized TensorRT engine format. This will help save computation costs and make it feasible to run perception modules on embedded computers such as NVIDIA Jetson devices. The navigation stack takes input from the perception module and reactively navigates through lanes. It traverses through lanes, changes lanes, and goes through the whole field until no more lanes are left.

6.1 Research answers

- The perception module, equipped with an RGB-D camera, could precisely estimate the location of trees. Based on the measurements, we estimated a feasible path by calculating the center of the field rows for the navigation module.
- The perception module was optimized with OpenVino and TensorRT, and it achieved up to 20 FPS in simulation and 25 FPS in real-world testing using NVIDIA 3060 Mobile GPU.
- The navigation module, after taking the input from the perception module about the center of rows, used the pure pursuit algorithm to safely navigate through the lanes and perform lane change maneuver.

Bibliography

[1] Yuxin Wu et al. *Detectron2*. https://github.com/facebookresearch/detectron2. 2019.

[2] Vincent Grondin, François Pomerleau, and Philippe Giguère. "Training Deep Learning Algorithms on Synthetic Forest Images for Tree Detection". In: *ICRA 2022 Workshop in Innovation in Forestry Robotics: Research and Industry Adoption*. 2022.

[3] Björn Åstrand and Albert-Jan Baerveldt. "A vision based row-following system for agricultural field machinery". In: *Mechatronics* 15.2 (2005), pp. 251–269. ISSN: 0957-4158. DOI: https://doi.org/10.1016/j.mechatronics.2004.05.005. URL: https://www.sciencedirect.com/science/article/pii/S0957415804000935.

[4] Jose Ortiz and Manuel Olivares. "A Vision Based Navigation System for an Agricultural Field Robot". In: Oct. 2006. DOI: 10.1109/LARS.2006.334338.

[5] Arnaud Ramey, Maria Malfaz, and Miguel Salichs. "Fast 3D cluster tracking for a mobile robot using 2D techniques on depth images". In: *Cybernetics and Systems* 44 (May 2013). DOI: 10.1080/01969722.2013.789645.

[6] Andrew English et al. "Vision based guidance for robot navigation in agriculture". In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. 2014, pp. 1693–1698. DOI: 10.1109/ICRA.2014.6907079.

[7] David Ball et al. "Vision-based Obstacle Detection and Navigation for an Agricultural Robot". In: *Journal of Field Robotics* 33.8 (2016), pp. 1107–1130. DOI: https://doi.org/10.1002/rob.21644. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/rob.21644. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.21644.

[8] Vignesh Ponnambalam et al. "Autonomous Crop Row Guidance Using Adaptive Multi-ROI in Strawberry Fields". In: *Sensors (Basel, Switzerland)* 20 (Sept. 2020). DOI: 10.3390/s20185249.

[9] Leonard Rusli, Brilly Nurhalim, and Rusman Rusyadi. "Vision-based vanishing point detection of autonomous navigation of mobile robot for outdoor applications". In: *Journal of Mechatronics, Electrical Power, and Vehicular Technology* 12 (Dec. 2021), pp. 117–125. DOI: 10.14203/j.mev.2021.v12.117-125.

[10] Alireza Ahmadi, Michael Halstead, and Chris McCool. *Towards Autonomous Visual Navigation in Arable Fields*. 2022. arXiv: 2109.11936 [cs.RO].

[11] Arun Narenthiran Sivakumar et al. *Learned Visual Navigation for Under-Canopy Agricultural Robots*. 2021. arXiv: 2107.02792 [cs.RO].

[12] Vincent Grondin et al. "Tree detection and diameter estimation based on deep learning". In: *Forestry: An International Journal of Forest Research* (Oct. 2022).

[13] Henry Hexmoor. "Reactive Navigation". In: *Essential Principles for Autonomous Robotics*. Cham: Springer International Publishing, 2013, pp. 81–91. ISBN: 978-3-031-01563-2. DOI: 10.1007/978-3-031-01563-2_9. URL: https://doi.org/10.1007/978-3-031-01563-2_9.

[14] A. E. B. Velasquez et al. "Reactive navigation system based on H∞ control system and LiDAR readings on corn crops". In: *Precision Agriculture* 21.2 (2020), pp. 349–368. ISSN: 1573-1618. DOI: 10.1007/s11119-019-09672-8. URL: https://doi.org/10.1007/s11119-019-09672-8.

[15] Roberto Guzman et al. "Autonomous hybrid gps/reactive navigation of an unmanned ground vehicle for precision viticulture -VINBOT". In: Nov. 2016.

[16] Richard Wallace et al. "First Results in Robot Road-Following." In: Jan. 1985, pp. 1089–1095.

[17] Omead Amidi and Charles E. Thorpe. "Integrated mobile robot control". In: 1991.

[18] R. Craig Coulter. "Implementation of the Pure Pursuit Path Tracking Algorithm". In: 1992.

[19] Changquan Ding, Hang Liu, and Hengyu Li. "Stitching of depth and color images from multiple RGB-D sensors for extended field of view". In: *International Journal of Advanced Robotic Systems* 16.3 (2019), p. 1729881419851665. DOI: 10.1177/1729881419851665. eprint: https://doi.org/10.1177/1729881419851665. URL: https://doi.org/10.1177/1729881419851665.

[20] *Vector Projections*. https://xaktly.com/VectorProjections.html. Accessed: 2023-05-21.

[21] Paul Heckbert. "What Are the Coordinates of a Pixel?" In: *Graphics Gems*. Ed. by Andrew Glassner. Boston: Academic Press, 1990, pp. 246 –248.

Appendix A

Reference Codes and Algorithms

A.1 Detectron2 to ONNX

```
git clone 'https://github.com/facebookresearch/detectron2'

python detectron2/tools/deploy/export_model.py \
    --sample-image image.png \
    --config-file detectron2/configs/COCO-Keypoints/
       keypoint_rcnn_R_50_FPN_3x.yaml \
    --export-method tracing \
    --format onnx \
    --output ./ \
    MODEL.WEIGHTS ./output/R-50_RGB_60k.pth \
    MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE 256  \
    MODEL.ROI_HEADS.NUM_CLASSES 1 \
    MODEL.SEM_SEG_HEAD.NUM_CLASSES 1 \
    MODEL.ROI_KEYPOINT_HEAD.NUM_KEYPOINTS 5 \
    MODEL.ROI_HEADS.SCORE_THRESH_TEST 0.90 \
    MODEL.MASK_ON True \
    INPUT.MASK_FORMAT bitmask \
    INPUT.MIN_SIZE_TEST 480 \
    INPUT.MAX_SIZE_TEST 640 \
    DATALOADER.NUM_WORKERS 8 \
    SOLVER.IMS_PER_BATCH 8 \
    MODEL.DEVICE cuda
```

Code A.1: Export Model to ONNX

A.2 ONNX to converted ONNX for TensorRT

We need to remove some layers and replace them with the TensorRT version in the ONNX file generated by the Detectron2 script above. We can do it using TensorRT as:

```
git clone 'https://github.com/NVIDIA/TensorRT.git'

python TensorRT/samples/python/detectron2/create_onnx.py \
    --exported_onnx ./model.onnx \
    --onnx ./converted.onnx \
    --det2_config detectron2/configs/COCO-Keypoints/
       keypoint_rcnn_R_50_FPN_3x.yaml \
    --det2_weights ./R-50_RGB_60k.pth \
    --sample_image image.png
```

Code A.2: Optimize ONNX Model by TensorRT

A.3 Converted ONNX to TensorRT

Finally, we need to build the engine on the same device where we will run our inference node.

```
git clone 'https://github.com/NVIDIA/TensorRT.git'

!python3 TensorRT/samples/python/detectron2/build_engine.py \
    --onnx ./converted.onnx \
    --engine ./engine.trt \
    --precision fp16 \
    --det2_config detectron2/configs/COCO-Keypoints/
       keypoint_rcnn_R_50_FPN_3x.yaml \
```

Code A.3: Converted ONNX to TensorRT

A.4 Algorithms

A general Pure Pursuit algorithm is shown in Algorithm 1.

Algorithm 1 A general pure pursuit algorithm

Funct pure_pursuit($current\_x, current\_y, \theta, path, index$)

Set the look ahead distance $lookahead\_distance := 0.0$
for $i = index, \ldots, len(path)$ **do**
    $x = path[i][0]$
    $y = path[i][1]$
    $distance = math.hypot(current\_x - x, current\_y - y)$
    if $lookahead\_distance < distance$ **then**
        $p_{close} = (x, y)$
        $index = i$
        $break$
    end if
end for
if $closest\_point$ is not $None$ **then**
    $\alpha = math.atan2(p_{close}[1] - current\_y, p_{close}[0] - current\_x)$
    $\gamma = \alpha - \theta$
else
    $\alpha = math.atan2(path[-1][1] - current\_y, path[-1][0] - current\_x)$
    $\gamma = \alpha - \theta$
    $index = len(path) - 1$
end if
if $\gamma > \pi$ **then**
    $\gamma = \gamma - 2\pi$
end if
if $\gamma < -\pi$ **then**
    $\gamma = \gamma + 2\pi$
end if
if $\gamma > \frac{\pi}{7}$ or $\gamma < \frac{-\pi}{7}$ **then**
    $sign = 1$ if $\gamma > 0$ else -1
    $\gamma = sign * \frac{\pi}{7}$
end if
return γ