POLITECNICO DI TORINO

POLITECNICO DI TORINO

Master Degree course in Mechatronic Engineering

Master Degree Thesis

Robust Autonomous Navigation in Vineyard for complete path coverage

Supervisors
Prof. Marcello Chiaberge

Candidate
Alessandro Di Stazio

Academic Year 2024-2025

Acknowledgements

First and foremost, I would like to thank Prof. Chiaberge for the possibility to work on such an interesting thesis project, that has thought me many things from a wide range of fields. I'm really grateful for the help and guidance of Marco Ambrosio and Mauro Martini throughout these months, without which I would have struggled. Thank you also to all my colleagues at Pic4SER, for the fun times and the help they provided during the few coffee breaks we had.

In ricordo di Mamma.
Grazie per tutto quello che hai fatto.

Abstract

Autonomous navigation in agricultural environments has emerged as a key innovation in modern agritech, providing significant benefits in terms of efficiency, cost reduction, and sustainability. In particular, autonomous systems in vineyards can alleviate labor shortages, optimize resource usage, and enable precision agriculture practices. However, the structured yet highly variable nature of vineyards presents unique challenges for autonomous navigation. Vineyards are characterized by narrow and uneven pathways, curved and sloped terrains, and dense vegetation that can obstruct visibility and interfere with traditional localization methods. Furthermore, environmental factors such as changing lighting conditions, seasonal variations, and occlusions from foliage add additional layers of complexity to perception and decision-making systems.

To achieve reliable autonomy, a combination of advanced perception, localization, and control strategies is necessary. Traditional approaches primarily rely on GNSS-based localization, often complemented by LiDAR, IMU, and wheel odometry to enhance accuracy and robustness. While these sensor fusion techniques have proven effective in open-field agricultural applications, they encounter significant limitations in vineyards and orchards. Tall and dense vegetation can obstruct GNSS signals, leading to localization errors and reduced reliability in environments with limited satellite visibility. Additionally, wheel odometry can suffer from drift and inaccuracies on uneven or slippery terrain, further complicating long-term navigation.

This thesis explores both localization-based and position-agnostic solutions to address these challenges. By leveraging behavior trees, an approach widely used in robotic control and decision-making, the proposed navigation system benefits from modularity, hierarchical structure, and real-time feedback mechanisms. These characteristics enable flexible and adaptable control pipelines capable of handling complex vineyard environments with varying terrain and occlusion conditions. The developed solutions are extensively evaluated through a combination of simulated environments and real-world experiments, providing a comprehensive assessment of their performance, robustness, and applicability in practical agricultural settings.

Contents

Chapter 1

Introduction

1.1 Objective of the thesis

Agricultural technology (agritech) has been fundamental to the development of human society since its inception. Every major historical era has been shaped by innovations in this field, from the first deliberate crop cultivation to the mechanization of agriculture during the Industrial Revolution. The pursuit of increased farming efficiency and productivity has consistently been a major driving force in society.

As demand for resources increased dramatically over the last century, agricultural infrastructure has evolved to keep pace. Precision agriculture, driven by data analytics and GNSS technology, emerged as a key solution to maximize efficiency. However, with a projected need to increase food production by up to 56% between 2010 and 2050 [13], the demand for highly autonomous and intensive farming solutions is at an all-time high. In response, agritech has expanded over the last decades to incorporate autonomous machinery, drones, and robotics to monitor and tend to crops.

The use of autonomous machinery in open field crops has become more widespread as the technology matures. These systems primarily rely on GNSS-based localization and satellite imagery for navigation planning and control. However, these solutions often face limitations in more complex environments, such as vineyards or orchards, which are characterized by narrow and uneven pathways, curved and sloped terrains, and dense vegetation that can obstruct visibility and disrupt GNSS localization. As a result, for these complex environments, more robust and adaptable control architectures are needed. These systems must handle navigation through a combination of traditional localization methods, when available, and environmental sensors such as cameras, LiDAR, IMUs, etc. to extrapolate navigation information.

This thesis focuses on the development of two control algorithms for autonomous navigation in vineyards. The first, a baseline algorithm, employs a traditional approach that uses GPS for robot localization along with GPS waypoints to mark the start and end of each vineyard row. In addition, it integrates LiDAR, IMU, and wheel odometry through sensor fusion to enhance obstacle avoidance and accurately determine the robot's position and orientation.

The second algorithm offers a position-agnostic solution that does not rely on GPS.

Instead, it uses LiDAR data to detect row edges, plan navigation within the row, identify the row's end, and guide the robot to the next row. Like the baseline approach, this method also utilizes LiDAR, IMU, and wheel odometry for obstacle avoidance and for robust estimation of position and orientation via sensor fusion.

The common thread linking both algorithms is the implementation of an underlying control architecture based on Behavior Trees (BTs) [7]. BTs are a formalism for structuring decision-making processes using a tree-like hierarchy that organizes actions and conditions. This modular and hierarchical framework not only makes the system more responsive through internal feedback loops but also simplifies debugging and future expansion, which is particularly beneficial for complex agricultural control tasks.

In addition to BTs, this work leverages the Robot Operating System (ROS) [8], an open-source middleware widely used for robotic application development. ROS provides a rich ecosystem of packages for sensor simulation, data filtering, navigation, and control, significantly streamlining the integration of various sensor inputs and control strategies. This allowed the focus to be on refining the BT-based algorithms while relying on robust, community-supported tools for key functionalities.

The performance of both algorithms was rigorously evaluated through extensive testing in simulation. This comprehensive evaluation enabled a direct comparison of the two approaches, highlighting the benefits and limitations of each method in addressing the challenges of autonomous vineyard navigation.

1.2 Thesis structure

- Chapter 2, State of the Art.
- Chapter 3, Algorithms for Complete Path Coverage.
- Chapter 4, Simulation Evaluation of Algorithms.
- Chapter 5, Conclusion.

Chapter 2

State of the Art

Autonomous navigation in complex environments requires robust control systems. This chapter reviews current approaches for vineyard navigation and examines how BTs can enhance control architectures.

2.1 Algorithms for Autonomous navigation in vineyards

Recent research has focused on complete path coverage in vineyards employing different combinations of sensors, navigation data, and underlying algorithms for control.

For example, in *A Deep Learning Driven Algorithmic Pipeline for Autonomous Navigation in Row-Based Crops* [3], a georeferenced occupancy grid map of the field is used to generate a global path composed of geographic waypoints. Navigation is performed either with pure reliance on GNSS or augmented with a Semantic segmentation algorithm that leverages visual perception. Within a row the Semantic segmentation algorithm [6] maintains the Autonomous Ground Vehicle (AGV) centered, while the end of a row is identified by a threshold on the Euclidean distance between the next waypoint and the AGV's estimated position. Navigation to the next row follows the previously computed global path. While this approach is highly scalable thanks to the low cost sensors it employs, its reliance on GNSS signals and satellite imagery considerably affects its flexibility. Satellite imagery may not be accurate due to seasonality or changes in the layout of the field, moreover the reliance on geographical waypoints for the end row navigation is not compatible with areas where GNSS signals are weak or absent altogether.

An alternative is proposed in *A Map-Free LiDAR-Based System for Autonomous Navigation in Vineyards* [2], in this work the simple geometrical structure of row based crops is used to extrapolate navigation information from LiDAR data. While navigating inside a row, the algorithm tries to keep the AGV centered, while avoiding any obstacles. The end of a row is detected if the number of points obtained from the LiDAR projection in front of the robot falls below a certain threshold. Finally, the AGV navigates to the next row aligning itself with the row ends, which are detected through Euclidean Cluster Extraction. In this approach a robust map-free solution was implemented; however, the

algorithm would highly benefit from the control architecture provided by BTs, which allow for modular behavior integration without restructuring the entire navigation algorithm.

2.2 BTs in Control Systems

BTs were originally developed in the gaming industry to overcome the limitations of Finite State Machines (FSMs). While FSMs can provide hierarchical modularity [5], their reliance on one-way control transfers forces each state to be directly connected to every other state. This results in up to N^2 possible transitions for N states, which causes exponential growth in complexity as the system scales. The increase in complexity naturally leads to longer development times and higher maintenance costs.

In contrast, BTs employ a two-way control transfer mechanism that is more aligned with modern programming practices. This paradigm allows execution to return to the main control flow once a function completes, enhancing modularity and facilitating easier feedback integration. This structure is particularly advantageous when developing complex, reactive and modular control systems.

2.2.1 An Informal Description of BTs

A more in-depth and formal definition of BTs is provided in *Behavior Trees in Robotics and AI* [4]:

> "A BT is a directed rooted tree where the internal nodes are called *control flow nodes* and leaf nodes are called *execution nodes*"

In a BT, every node except the root has exactly one parent, with the root node having none. Execution begins at the root, which issues a tick signal at a fixed frequency. This tick propagates from parent to child nodes, ensuring that each node executes only when it is ticked. When a node executes, it returns one of three statuses: *Running*, *Success*, or *Failure*. As shown in Table 2.1, control flow nodes such as Sequence, Fallback, Pipeline, and Decorator determine their return status based on the statuses of their child nodes. While execution nodes, which include Actions and Conditions, operate based on their internal algorithms.

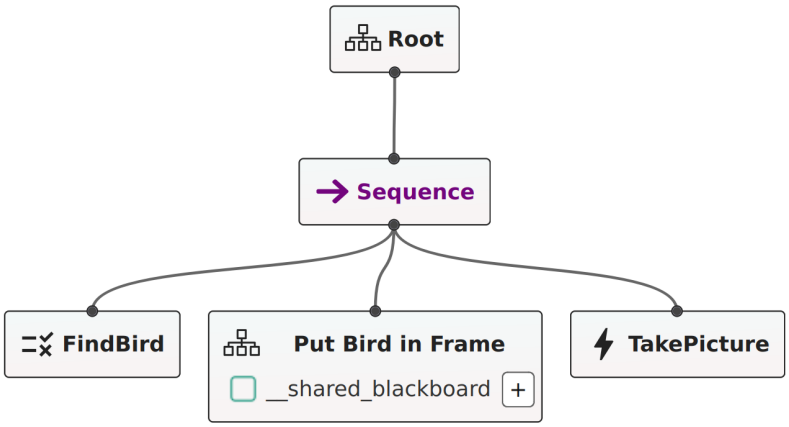
Node type	Symbol	Succeeds	Fails	Running
Action	text	Upon completion	If impossible to complete	During completion
Condition	text	If true	If false	Never
Decorator	◊	Custom	Custom	Custom
Fallback	?	If one child succeeds	If all children fail	If one child returns Running
Sequence	→	If all children succeed	If one child fails	If one child returns Running
Parallel	⇉	If $\geq M$ children succeed	If $> N - M$ children fail	else

Table 2.1: Adapted from *Behavior Trees in Robotics and AI* [4], p. 9. Overview of classical BT node types.

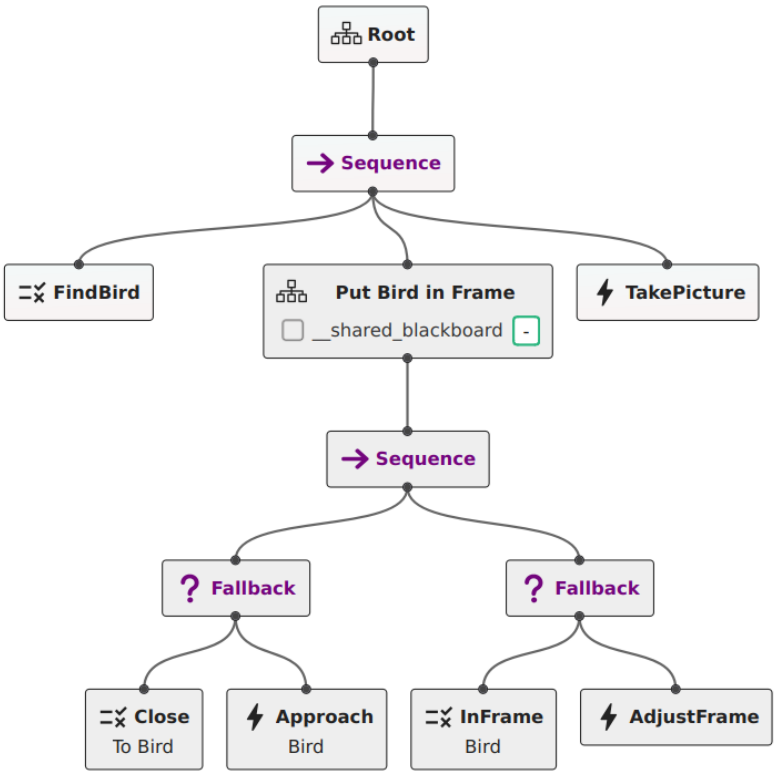
The hierarchical structure of BTs not only facilitates modular design but also allows each node to act as the root of a sub-BT. This means that complex behaviors can be encapsulated within subtrees and later reused or modified as needed for higher-level tasks. This modularity is exemplified in Fig. **??**, where a high-level behavior is decomposed into finer, lower-level activities.

2.2.2 The Navigation 2 ROS package and BTs

To handle navigation, the Navigation 2 package (NAV2) for ROS Humble was used. For two reasons. Firstly, this package is widely adopted by the ROS community, providing a well-tested pipeline for path computation and navigation. It supports different types of controllers, is highly configurable, and benefits from documentation and readily available example code. The second reason is NAV2's integration of BTs. NAV2 includes custom nodes specifically designed for BT-based navigation, allowing seamless integration with its underlying algorithms. Table 2.2 provides a brief overview of the custom nodes implemented by NAV2 used in this work.



(a) A high level BT carrying out a task consisting of first finding, then getting into frame and finally taking a picture of a bird.



(b) The Action Put Bird in Frame from the BT in Fig. 2.1a is expanded into a sub-BT. The Bird is approached until it is considered close, and then the Action AdjustFrame is executed until the Bird is within frame.

Key Behavior Tree Nodes Used in This Work (continued)

Node Type	Node Name	Description

Table 2.2: NAV2 Behavior Tree Nodes Used in This Work

Node Type	Node Name	Description
Action	BackUp	Commands the robot to reverse a specified distance at a given speed.
	ClearEntireCostmap	Calls a ROS service to clear the entire costmap (global or local).
	ComputePathThroughPoses	Computes a path through multiple waypoints using the selected planner.
	ControllerSelector	Selects the controller based on topic input and outputs the chosen controller.
	DriveOnHeading	Commands the robot to drive along a specified heading.
	FollowPath	Instructs the robot to follow a precomputed path.
	PlannerSelector	Selects the planner based on topic input and outputs the chosen planner.
	RemovePassedGoals	Filters out navigation goals that have already been passed.
	Spin	Commands a spin maneuver as a recovery action.
	Wait	Pauses execution for a specified duration.
Condition	GoalUpdated	Checks whether the navigation goal has been updated.
	IsStuck	Determines if the robot is not making progress.
Control	PipelineSequence	Executes child nodes sequentially until one fails.
	RecoveryNode	Retries a set of recovery actions a specified number of times.
	RoundRobin	Cycles through its children, ticking each in turn.
Decorator	RateController	Regulates the tick rate of its child node by limiting its frequency.

Note: For a complete list of available NAV2 BT nodes and their details, please refer to the NAV2 documentation.

2.2.3 The BehaviorTree.CPP Library and BTs

To implement behavior trees, the BehaviorTree.CPP (BTCpp) library was used. This library is a widely adopted C++ framework for creating, managing, and executing behavior trees efficiently.

One of the key advantages of BTCpp is its visual design tool, Groot, which allows for intuitive construction and debugging of behavior trees. This enables rapid prototyping and real-time monitoring of tree execution. Additionally, BTCpp provides flexibility in defining custom nodes, which can be integrated either statically at compile time or dynamically as plugins. In this work, the plugin-based approach was adopted, allowing custom nodes to be loaded at runtime without modifying the core application.

BTCpp also features a structured mechanism for data management within behavior trees. Nodes can communicate using input and output ports, enabling the controlled exchange of parameters. Additionally, a blackboard is available for storing and sharing global variables across the tree, allowing information persistence between nodes. This facilitates modular design while keeping nodes decoupled from each other, improving reusability and maintainability.

Table 2.3: BehaviorTree.CPP Nodes Used in This Work

Node Type	**Node Name**	**Description**
Control	PipelineSequence	Executes child nodes sequentially until one fails.
	ReactiveSequence	Ticks all children continuously in order, proceeding only if each returns Success, ensuring real-time reactivity.
	ReactiveFallback	Ticks all children continuously and selects the first one that succeeds, enabling real-time failure recovery.
Decorator	Inverter	Ticks its child and returns the opposite return state.

Note: For a complete list of available BTCpp BT nodes and their details, please refer to the BTCpp documentation.

2.2.4 Cross-Domain Application: AUV Use Case

BTs have proven highly effective in managing the complex and dynamic operations of Autonomous Underwater Vehicles (AUVs), as demonstrated in [12]. Underwater environments are characterized by unpredictable conditions, limited communication, and the need for long-duration autonomous operation. BTs have enabled modular, reusable, and robust control architectures capable of handling diverse mission phases.

Considering that AUVs operate in environments that are as, if not, more complex than those encountered by AGVs, it stands to reason that these would derive considerable benefit from the modularity, adaptability, and scalability inherent in BTs.

Chapter 3

Algorithms for Complete Path Coverage

3.1 Overview

This chapter details two algorithms designed for complete path coverage. Both share a common foundation. Including a custom BT plugin library, navigation (using Nav2) and goal computation. But differ fundamentally in the end of row detection mechanism. Here, the shared components are outlined and then each algorithm is explained in detail, highlighting the unique strategies employed to detect the end of a row.

3.2 Shared Implementation Components

3.2.1 Custom BT Nodes Plugin Library

In this section, the custom BT nodes implemented to support both algorithms are detailed. Table 3.1 summarizes each node type and its function.

Table 3.1: Overview of custom BT nodes.

Node Type	**Node Name**	**Description**
Action	ClusteringNodeAction	Clusters LiDAR data and publishes rows line equations
	CustomGoalUpdaterAction	Updates goals based on current navigation data
	FixedFramePubNodeAction	Publishes a static transform between two frames
	Nav2ClientAction	Implements a Nav2 client to call the Nav2 action server

Overview of custom BT nodes (continued)

Node Type	Node Name	Description
	RowLinesNodeAction	Publishes PointCloud2 data starting from received row line equations
	setBlackboardBoolAction	Sets a Bool blackboard value
Condition	BlackboardBoolCondition	Returns as status the specified Bool blackboard value
	RosTopicCondition	Returns as status the received Bool value from a ROS topic subscription

3.2.2 Row Lines Detection

Vineyards typically feature straight, parallel rows defined by walls of plants. Detecting these plant walls is essential for navigation and goal computation. However, the natural gaps between plants cause these walls to appear discontinuous in raw LiDAR data. As shown in Fig. 3.2a, the LiDAR sensor often captures multiple rows because of these gaps.

To address this challenge, a multi-step approach is adopted (see Fig. 3.1). First, the LiDAR data is filtered to remove outliers caused by adjacent rows. Next, the filtered data is segmented into two clusters representing the right and left walls. Finally, linear regression is applied to each cluster to approximate the line equations of the walls. This process enables the control system to extract critical navigation data, such as the distance between the robot and the row lines, which is used for goal computation.

The following subsections describe this process in further detail.

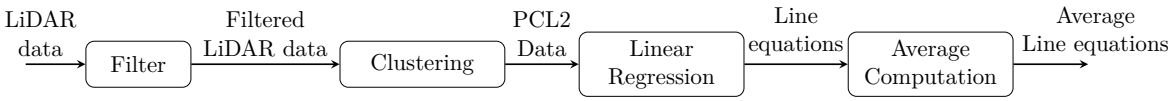
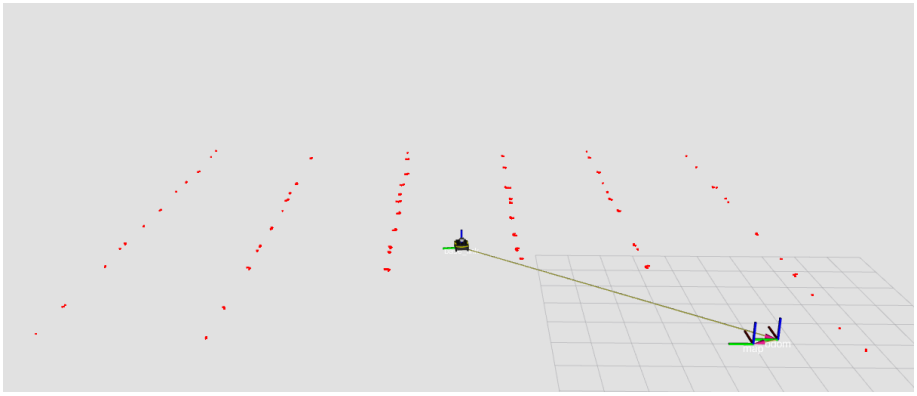


Figure 3.1: Simple diagram of the Row Lines Detection pipeline

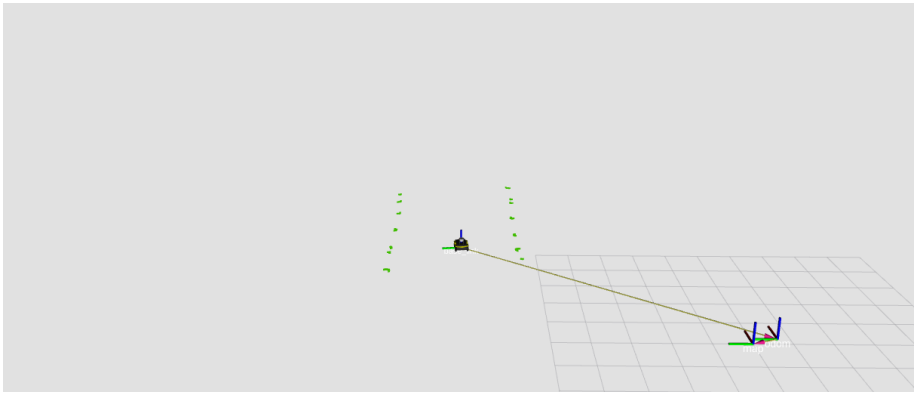
LiDAR Filtering

To filter the LiDAR data, the *LaserFilters* ROS package was employed. This package provides a variety of filters that can be chained together to achieve a customized filtering effect. To remove outliers corresponding to adjacent rows, the *LaserScanBoxFilter* was used. This filter discards any LiDAR points that fall within a specified Cartesian box, defined by minimum and maximum x, y, and z coordinates.

In this approach, four *LaserScanBoxFilter* filters were chained together to remove points detected in front of, behind, to the right, and to the left of the robot. This process effectively confines the LiDAR data to a rectangular region around the robot, as shown in Fig. 3.2b.



(a) Unfiltered data in red



(b) Filtered data in green

Figure 3.2: Visualization in Rviz of Filtered and Unfiltered LiDAR data

Clustering

To segment the filtered LiDAR data into two clusters, the custom BT node *ClusteringNodeAction* is used. First, the LiDAR scan is projected into the navigation reference frame (either Map or Local), converting the data into a PointCloud2 (PCL2) format with (x, y, z) coordinates. This transformation clarifies the spatial distribution of points, making it easier to separate those corresponding to the two walls.

Clustering is then performed using Gaussian Mixture Models (GMMs) [1], implemented via the C++ Armadillo library [9–11]. The GMM is trained on the PCL2 data to identify two distinct Gaussian distributions, which correspond to the two plant walls. Once the model is successfully trained, it is used to partition the PCL2 data into two separate clusters. These clusters are then fed into the *applyLinearRegression* method of the *ClusteringNodeAction*, as described in the next section.

Row Lines Approximation via Linear Regression

To approximate the row walls, their line equations (expressed in the navigation frame) are computed using Linear Regression. This computation is integrated into the *ClusteringNodeAction* to reduce latency and prevent the use of stale data. The process leverages the Boost C++ library to derive an approximate linear model that best fits the points of each cluster. Once computed, the line equations are published on a ROS topic for subsequent use. The line message contains along with the slope and intercept also the maximum and minimum x coordinate from the cluster associated with the line.

Exponential Moving Average with Sliding Window

Using the newly computed line equations alone would not result in a robust and reliable system, as clustering may fail to correctly identify the row walls. To mitigate this, the *RowLinesNodeAction* custom BT node implements an Exponential Moving Average (EMA) with a sliding window to smooth out variations and discard outliers. The EMA is applied to all line parameters (m, q, x_{max}, x_{min}). As new data arrives, the oldest sample is removed, and the weights are updated such that older samples have progressively less influence. This approach ensures that the system favors recent data while maintaining robustness against outliers. The weights are computed online, as follows:

$$\text{Exponential Decay Function:} \quad \begin{cases} w_i & = e^{-\lambda(N-i)}, \quad \forall i \in \{0, \dots, N-1\} \\ N & \in \mathbb{N}, \quad \lambda \in \mathbb{R}^+ \end{cases} \tag{3.1}$$

$$\text{Weighted Average Formula:} \quad x_{\text{avg}} = \frac{\sum_{i=1}^{N} x_i w_i}{\sum_{i=1}^{N} w_i} \tag{3.2}$$

Where λ is the exponential decay constant, controlling the rate at which the exponential function falls. The exponent scales by $(N - i)$, ensuring that the newest samples in the sliding window receive the highest weights, as shown in Fig. 3.3. To preserve scale, eliminate bias, and improve numerical stability the weights are normalized, ensuring their sum equals one. The computed averages are then used to generate an artificial PCL2 obstacle consisting of N equidistant points along the two lines, within their x_{max} and x_{min} bounds. This PCL2, along with the line equations, are published on ROS topics. The PCL2 allows the Nav2 stack to recognize continuous row walls as obstacles (Fig. 3.4), while the line equations are used for goal computation.

3.2.3 Navigation and Goal Computation Framework

This section explains the integration with Nav2 for navigation and outlines the goal computation methods that are shared between the two algorithms.

The overarching control architecture is based on the use of two BTs, the *MainBT* and the *NavBT*. The MainBT handles the high level control logic. It detects the end of the mission, extrapolates navigation data from sensors (GPS and/or LiDAR), computes goals and passes these to the NavBT. The NavBT is activated by the *Nav2ClientAction* node

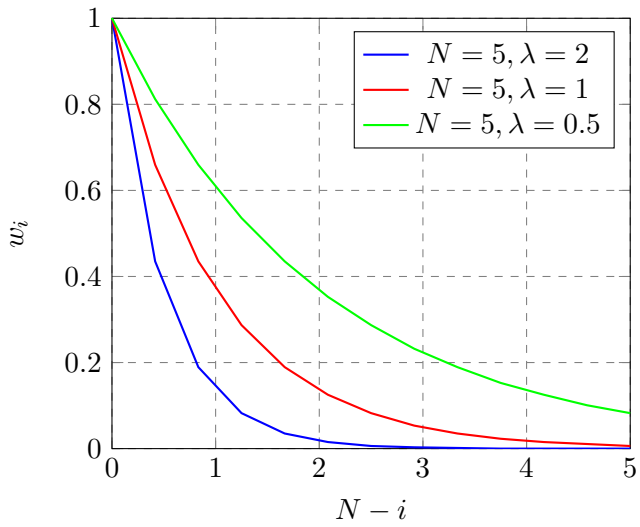


Figure 3.3: Exponential Decay Function for different decay constants.

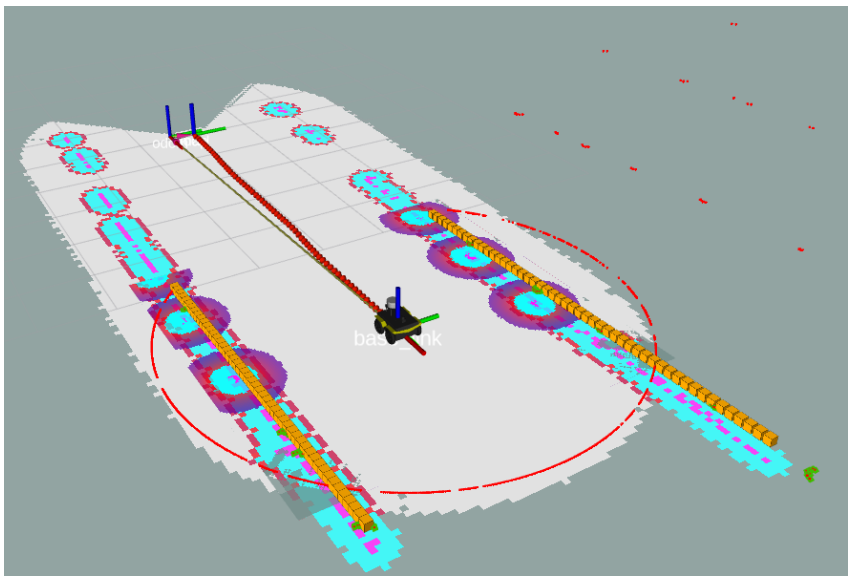


Figure 3.4: Visualization in Rviz of detected Row Lines PCL2 points in orange.

that communicates with the Nav2 action server. This node sends a navigation request specifying the use of the NavigateThroughPoses controller, along with the computed goal or goals. Additionally, it provides the XML of the NavBT, ensuring that the navigation and recovery procedures follow the designed behavior tree structure. This BT is completely made of nodes available from the Nav2 BT plugin, as shown in Fig. 3.5.

Goal computation is performed in the MainBT, through the *CustomGoalUpdaterAction* node. Depending on the navigation state, i.e. inrow or endrow, the system employs

two distinct goal computation strategies. The following subsections describe these approaches.

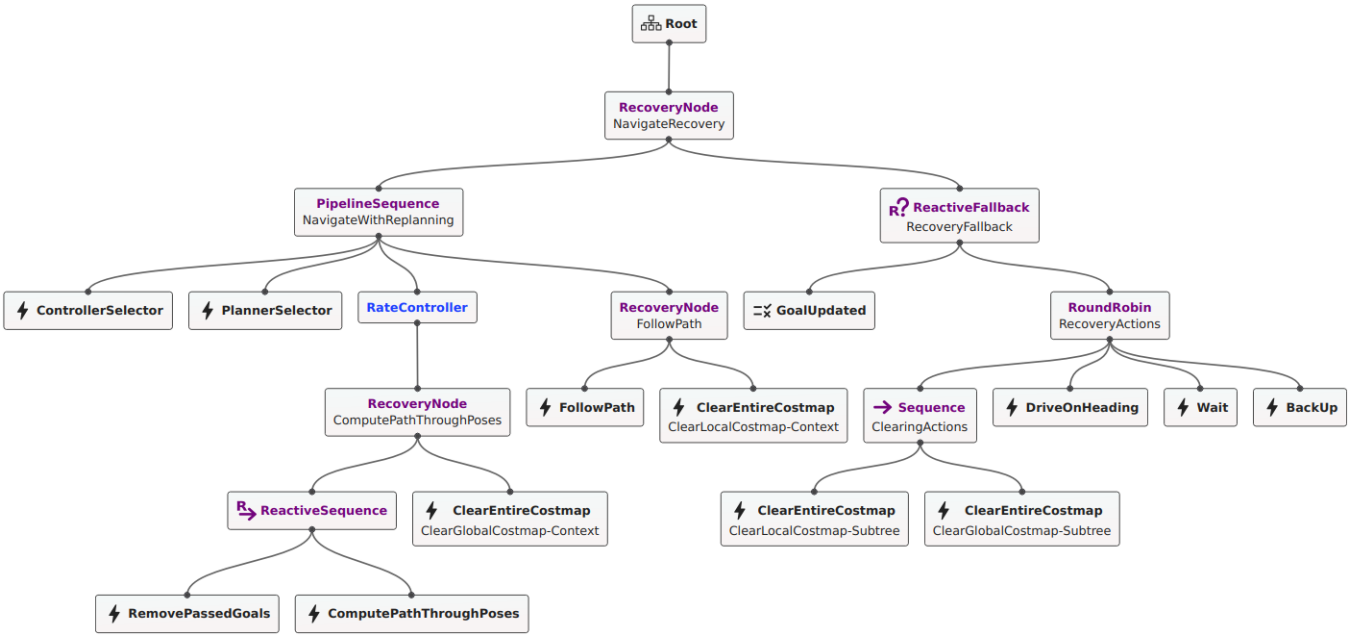


Figure 3.5: Illustration of the NavBT configuration for navigation with recovery.

Goal Computation In Row

While the robot is detected as inside of a row, the latest odometry of the robot is used to compute the next goal as follows:

$$(x, y) = (x_{odom} + \Delta x, y_{odom} + \Delta y_{drift}) \tag{3.3}$$

$$\Delta y_{drift} = \frac{d_0 - d_1}{d_0 + d_1}, \quad \begin{cases} \Delta x & > 0, \quad \text{if positively aligned with the } x \text{ axis} \\ \Delta x & < 0, \quad \text{otherwise} \end{cases} \tag{3.4}$$

Where d_0 and d_1 are the distances between the robot and the two detected row lines, as shown in Fig. 3.6.

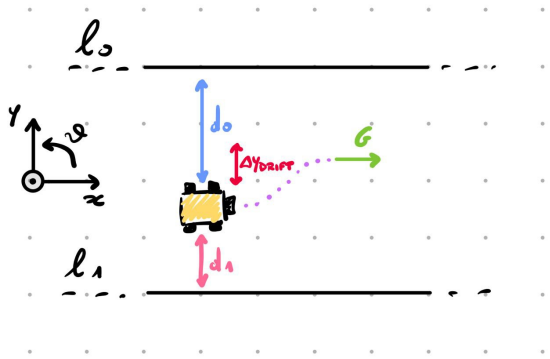


Figure 3.6: Computation of goal along positive x axis for in-row navigation.

Goal Computation End Row

If the robot is detected to be at the end of a row, a more complex approach is used. This is to ensure the robot enters the next row aligned while keeping clear of the vineyard plants. Again the latest odometry and estimated robot-row lines distance are used, computing N equidistant goals $[G_1, \ldots, G_i, \ldots, G_N]$ along a circumference centered on the top row line. Assuming the example shown in Fig. 3.7, the goals will be computed as follows:

$$\text{The equation of a circumference is } \gamma : (x-a)^2 + (y-b)^2 = r^2$$

$$\text{Then set } \begin{cases} a = x_{odom} \\ b = y_{odom} + d_0 \\ r = d_0 \end{cases} \quad \text{to obtain the translated circumference of interest.}$$

To follow the trajectory described by this circumference, consider a reference frame translated to the center of the circumference. Then apply a rotation around its z-axis by an angle θ to the relative position of the robot w.r.t. the new reference frame:

$$R(\hat{k},\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}, \begin{pmatrix} x_{rel} \\ y_{rel} \end{pmatrix} = \begin{pmatrix} x_{odom} - a \\ y_{odom} - b \end{pmatrix}$$

$$\Rightarrow G_{rel} : \begin{pmatrix} x_{G_{rel}} \\ y_{G_{rel}} \end{pmatrix} = R(\hat{k},\theta) \cdot \begin{pmatrix} x_{rel} \\ y_{rel} \end{pmatrix} = \begin{pmatrix} x_{rel}\cos\theta - y_{rel}\sin\theta \\ x_{rel}\sin\theta + y_{rel}\cos\theta \end{pmatrix}$$

Translate the Goal coordinates back to the navigation reference frame :

$$G = G_{rel} + \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} x_{rel}\cos\theta - y_{rel}\sin\theta + a \\ x_{rel}\sin\theta + y_{rel}\cos\theta + b \end{pmatrix}$$

Using the derived equations, defining the total angle of rotation, it is then possible to compute N equidistant goals:

$$\Delta\theta = \frac{\theta_{tot}}{N}, \theta_i = \theta_{i-1} + \Delta\theta, \forall i = \{1,\dots,N-1\}, N \in \mathbb{N} \text{ with } \theta_0 = \Delta\theta$$

$$\Rightarrow G_i = \begin{pmatrix} x_{rel}\cos\theta_i - y_{rel}\sin\theta_i + a \\ x_{rel}\sin\theta_i + y_{rel}\cos\theta_i + b \end{pmatrix}$$

Finally to ensure smooth navigation, the orientation of each goal pose is set to the following Roll-Pitch-Yaw configuration: $R(\hat{k},\theta_i) \cdot R(\hat{j},0) \cdot R(\hat{i},0)$. No additional rotations are needed since the navigation frame and the relative frame defined by the circumference have parallel axes.

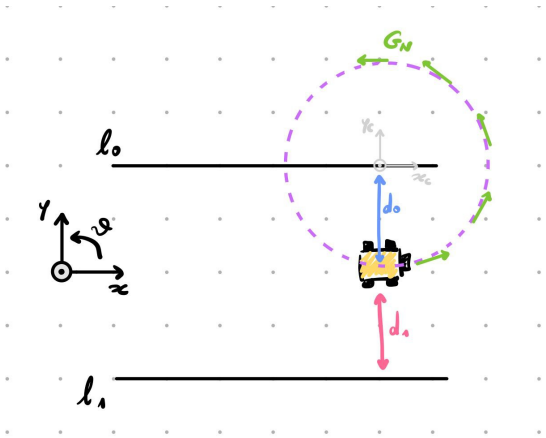


Figure 3.7: Visualization of equidistant goals along a translated circumference for end-of-row navigation.

3.3 Baseline Algorithm (GNSS-Dependent)

This algorithm leverages GPS data to guide navigation.

3.3.1 Main BT Structure (Fig. 3.8)

This section describes the high-level control actions performed by the Main BT and its subtrees.

The control flow is divided into two parts. The first part contains actions that occur at every tick, while the second part comprises actions that depend on the navigation mode (inrow or endrow).
At the root of the Main BT is a Pipeline sequence, ensuring that the tree re-ticks starting from its first child if any return *Running*. In the first part, the end-of-mission condition is checked, then through a Fallback control node it determines if the navigation state should be detected from the current GPS position, and finally performs clustering. These tasks are implemented by the *RosTopicCondition*, *GPSCheckerNodeAction*, and *ClusteringNodeAction* custom BT nodes, respectively.

The second part is managed by the *Reactive Fallback Nav*, which switches between the two navigation subtrees depending on the detected navigation mode. The following subsections provide a detailed explanation of these subtrees.

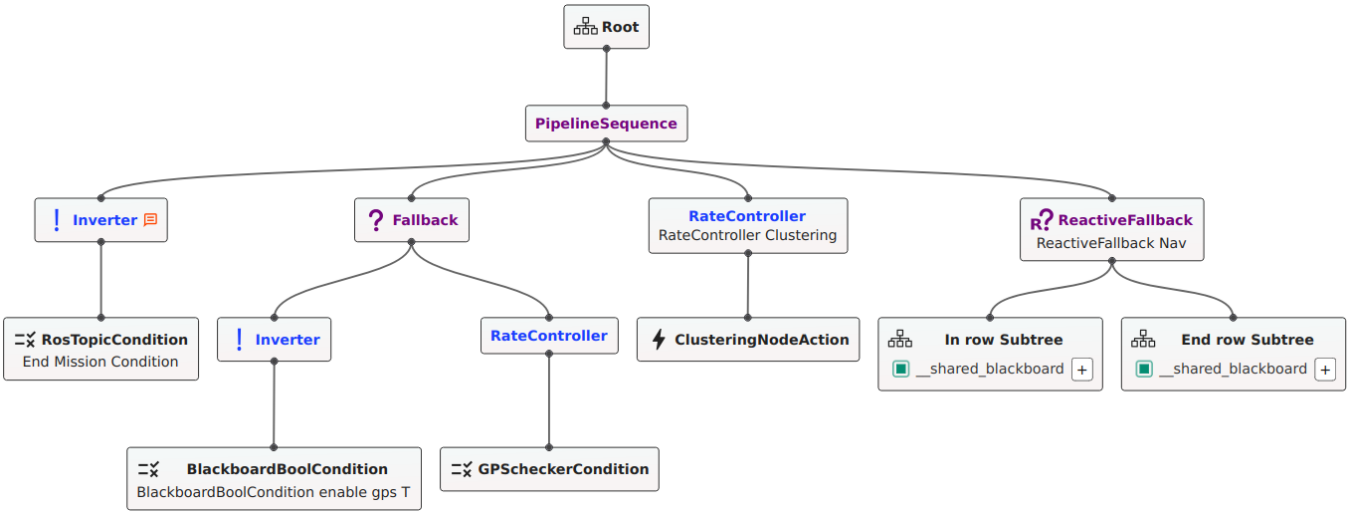


Figure 3.8: Illustration of the Position Agnostic Main BT.

In row Subtree (Fig. 3.9)

The control flow of this subtree is rather simple, with a Pipeline Sequence control node at its root. First, it checks if the current navigation mode is set to inrow. If it is, clustering is enabled by setting the *Enable Clustering* blackboard key to True. Next, the *RowLinesNodeAction* is ticked to compute and publish the row lines and associated

PCL2. Finally, the navigation goal is computed and stored on the blackboard for the *Nav2ClientAction* node to access. This node sends the navigation request to the Nav2 action server and its return state reflects the navigation result: goal reached, running or failed/aborted.

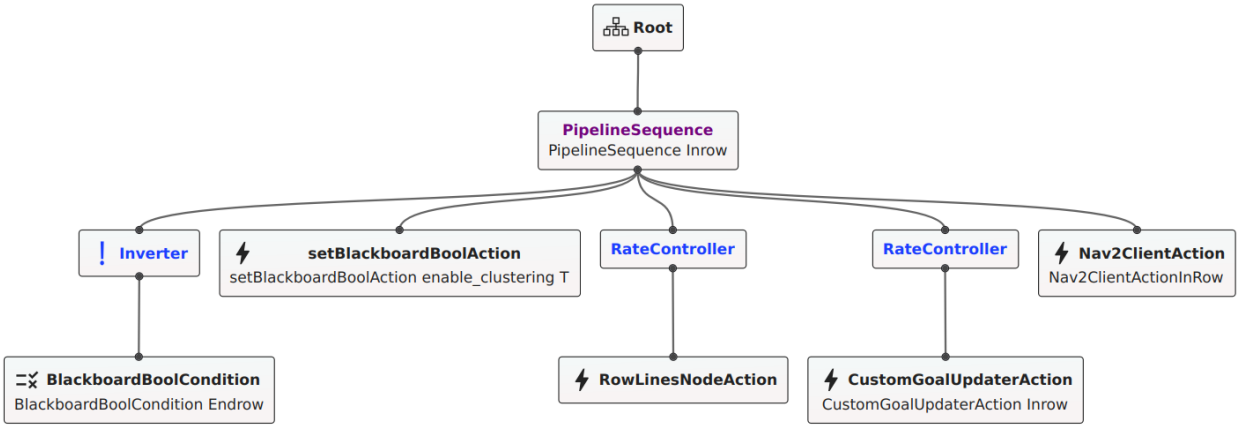


Figure 3.9: Illustration of In Row Subtree from the Baseline MainBT.

End row Subtree (Fig. 3.10)

The control flow in the End Row subtree is more complex, with a Pipeline Sequence control node at its root. Upon entering the subtree, the system first verifies whether the current navigation mode is set to end row. If it is, a Reactive Fallback control node is ticked.

The *Reactive Fallback Endrow* node selects between two child branches based on the goal navigation mode. If the goal has not been reached (or is unset), the *Sequence Endrow Nav* control node is ticked. At this stage, clustering is disabled to prevent artifacts during end-of-row navigation, and the GPS navigation state is turned off to avoid unexpected navigation mode switches. Subsequently, the system computes new goal poses and sends a navigation request using the *CustomGoalUpdaterAction* and the *Nav2ClientAction* custom BT nodes.

Once navigation is completed, by reaching the final goal in the computed path, the *Nav2ClientAction* node sets the *Goal reached endrow* key to True. This triggers the alternate branch of the Reactive Fallback in the next iteration, where clustering and GPS checks are re-enabled through their respective blackboard keys.

3.3.2 Goal Computation and Navigation

The Baseline algorithm uses the *Map* frame, a world fixed frame, for navigation and goal computation. Within its BT structure, the algorithm employs the goal computation and navigation strategy described extensively in **Section 3.2.2**. Utilizing the detected row lines to identify row walls as continuous obstacles (see Fig. 3.4), to keep the robot centered during in row navigation, and to compute the of the end-of-row trajectory.

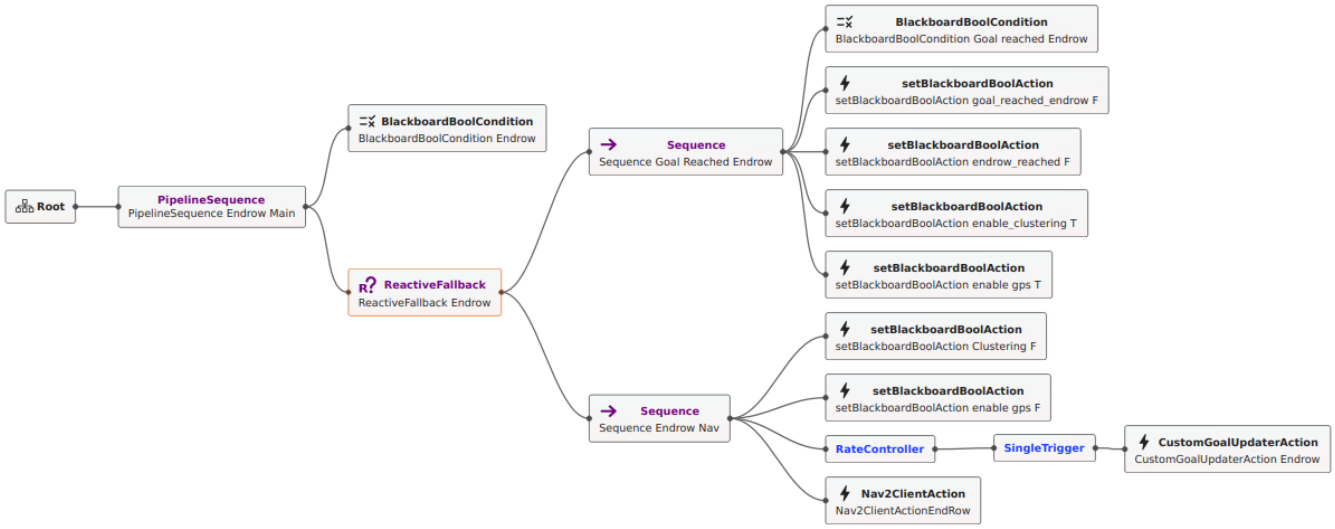


Figure 3.10: Illustration of End Row Subtree from the Baseline MainBT.

3.3.3 End of Row Detection

The algorithm relies on GPS data to detect the end of a row, this is implemented in the *GPScheckerNodeAction*. A set of GPS coordinates are loaded from a configuration file, marking the start and end of each row. At each tick, the node checks if the distance between the latest GPS position and the next waypoint falls below than a predefined threshold. When this condition is met, the navigation state is switched and the reference waypoint is updated. To compute the distance between the robot and the reference waypoint, the *Haversine formula* is used. This formula allows the computation of the *great-circle distance* (see Fig. 3.11) between two points on a sphere, given their latitudes and longitudes.

$$\text{Central angle: } \theta = \frac{d}{r}, \quad \begin{cases} d : \text{great-circle distance} \\ r : \text{radius of the sphere} \end{cases} \tag{3.5}$$

$$\tag{3.6}$$

$$\text{Haversine formula: } \begin{cases} \text{hav}(\theta) = \text{hav}(\Delta\phi) + \cos\phi_1 \cos\phi_2 \text{hav}(\Delta\lambda) \\ \phi_1, \phi_2 : \text{latitude of point 1 and 2} \Rightarrow \Delta\phi = \phi_2 - \phi_1 \\ \lambda_1, \lambda_2 : \text{longitude of point 1 and 2} \Rightarrow \Delta\lambda = \lambda_2 - \lambda_1 \end{cases} \tag{3.7}$$

By definition the Haversine function is $\text{hav}(\theta) = \sin^2\frac{\theta}{2} = \frac{1-\cos\theta}{2}$, then solving for the distance d:

$$\begin{aligned} d = r \ \text{archav}(\text{hav}(\theta)) = 2r \arcsin\sqrt{\text{hav}(\theta)} = \cdots = \\ = 2r \arcsin\sqrt{\frac{1 - \cos(\Delta\phi) + \cos\phi_1 \cdot \cos\phi_2 \cdot (1 - \cos(\Delta\lambda))}{2}} \end{aligned} \tag{3.8}$$

This formula is valid only when $0 \leq \text{hav}(\theta) \leq 1$, making it not viable for points at the opposite sides of the sphere. However for our application this not a concern, as the robot and waypoint are in the range of tens on meters at most.

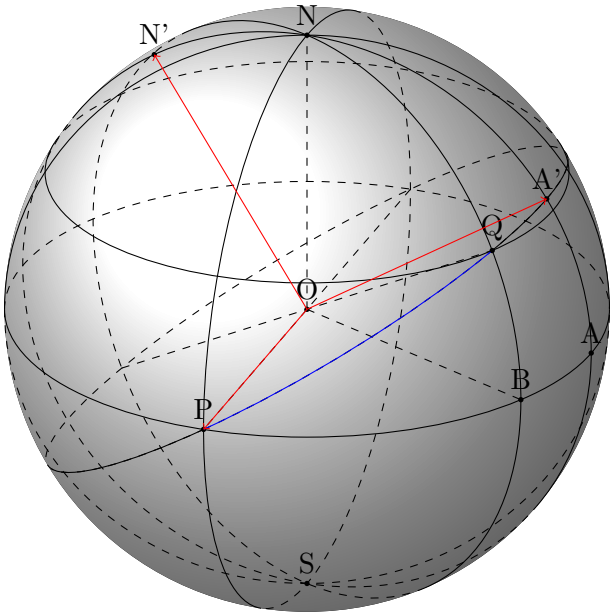


Figure 3.11: Visualization of the great-circle distance between two points (P and Q) in blue.

3.4 Position Agnostic Algorithm

Focused on overcoming GNSS limitations, this approach relies on sensor data other than GPS.

3.4.1 Main BT Structure (Fig. 3.12)

This section describes the high-level control actions performed by the Main BT and its subtrees.

The control flow is divided into two parts. The first part contains actions that occur at every tick, while the second part comprises actions that depend on the navigation mode (inrow or endrow).

At the root of the Main BT is a Pipeline sequence, which ensures that the tree re-ticks starting from its first child if any return *Running*. In the first part, the end-of-mission condition is checked then a dynamic reference frame for navigation is published and computed, and finally clustering isperformed. These tasks are implemented by the *RosTopicCondition*,*FixedFramePubNodeAction* and the *ClusteringNodeAction* custom BT nodes,

respectively. Both nodes offer configurable functionalities through their input ports, such as enabling the computation of a new frame transform or enabling clustering.

The second part involves two *ReactiveFallbacks* that, based on blackboard values, activate or switch to specific control branches. For instance, the *Reactive Fallback Settling* checks whether settling is enabled. If it is, the BT refrains from engaging in navigation for a predefined time interval. This delay is managed by the *Delay* node, which returns *Running* until the specified time has elapsed, at which point it returns *Success*. Combined with the root Pipeline sequence, this behavior causes the BT to restart until the delay period is over. This approach prevents the use of clusters derived from noisy LiDAR data that may be gathered during the final section of the end-of-row trajectory, which often includes data from adjacent rows.

The second Reactive Fallback, called *Reactive Fallback Nav*, is responsible for switching between the two navigation subtrees. The following subsections provide a detailed explanation of these subtrees.

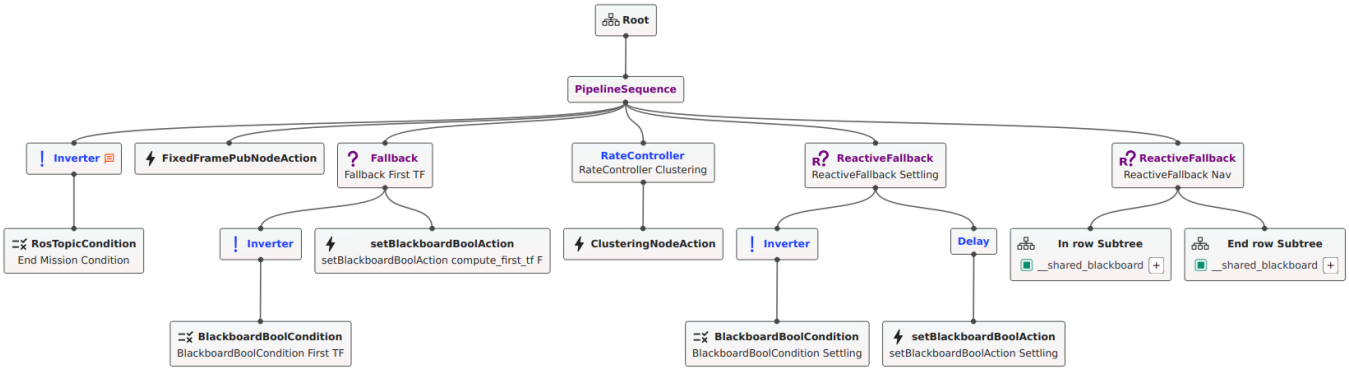


Figure 3.12: Illustration of the Position Agnostic Main BT.

In row Subtree (Fig. 3.13)

The control flow of this subtree is rather simple, with a Pipeline Sequence control node at its root. First, it checks if the current navigation mode is set to inrow. If it is, clustering is enabled by setting the *Enable Clustering* blackboard key to True. Next, the *RowLinesNodeAction* is ticked to compute and publish the row lines and associated PCL2. This node also checks wether the end of a row has been reached and updates the blackboard key accordingly, ensuring that the correct navigation subtree is ticked in the next iteration. Finally, the navigation goal is computed and stored on the blackboard for the *Nav2ClientAction* node to access. This node sends the navigation request to the Nav2 action server and its return state reflects the navigation result: goal reached, running or failed/aborted.

End row Subtree (Fig. 3.14)

The control flow in the Endrow subtree is more complex, as it must coordinate multiple actions based on navigation progress. To manage this complexity, several control nodes

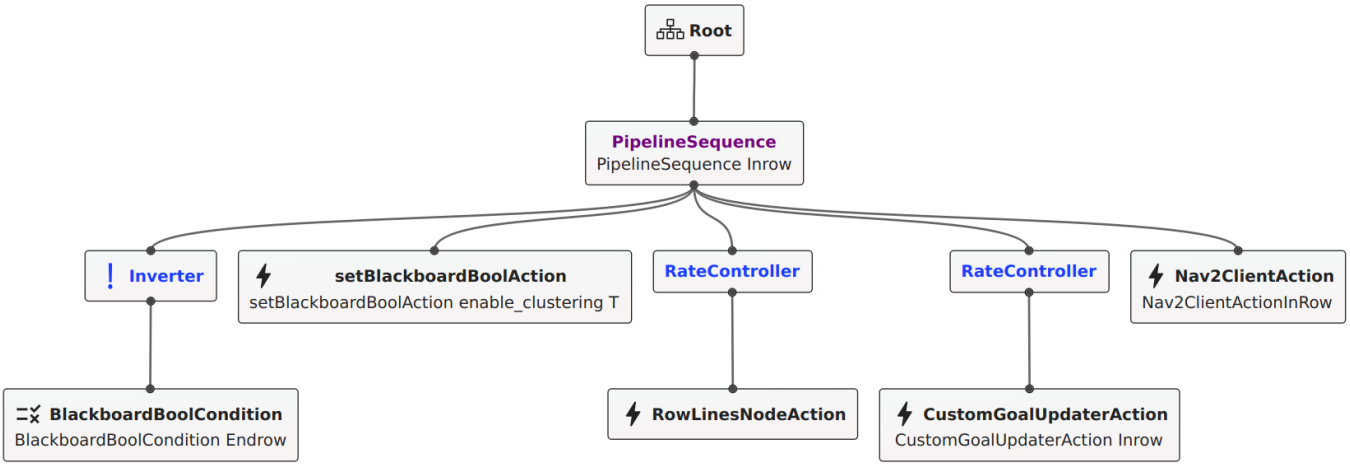


Figure 3.13: Illustration of Inrow Subtree from the Position Agnostic MainBT.

are nested within the subtree.

Upon entering the subtree, it first checks if the current navigation mode is set to endrow. If so, a Reactive Fallback control node is ticked. This node switches between two branches whose root nodes are respectively *Sequence Endrow Goal Reached* and *Sequence Endrow Nav*. In the *Sequence Endrow Goal Reached*, the node first checks wether the last goal of the end-of-row trajectory has been reached through the *Goal Reached* blackboard key. If so, the *Local* frame is updated and published, and both clustering and settling are enabled by updating their blackboard keys. Finally, the navigation mode is switched to inrow by setting its blackboard key. In the *Sequence Endrow Nav*, first clustering is disabled to prevent artifacts during end of row navigation. Next, ticking its children the new goal poses are computed and the navigation request sent. To determine wether the last goal of the end-of-row trajectory has been reached, the *Nav2ClientActionNode* uses its output port to set the blackboard key *Goal Reached* with the feedback received by the Nav2 action server.

3.4.2 Extrapolating Navigation Data from LiDAR

Unlike the Baseline algorithm, the Position Agnostic approach primarily extrapolates navigation data from LiDAR. As detailed in **Section 3.2.2**, row lines are detected directly from raw LiDAR data. These detections allow for the computation of both the distances between the robot and the row walls and the estimated endpoint of the row within the navigation frame.

3.4.3 Goal Computation and Navigation

A key difference from the Baseline algorithm is the use of a dynamic navigation frame called *Local*. This frame is updated every time the robot enters a new row. The dynamic frame ensures that the robot is always positively aligned with the x-axis of the navigation

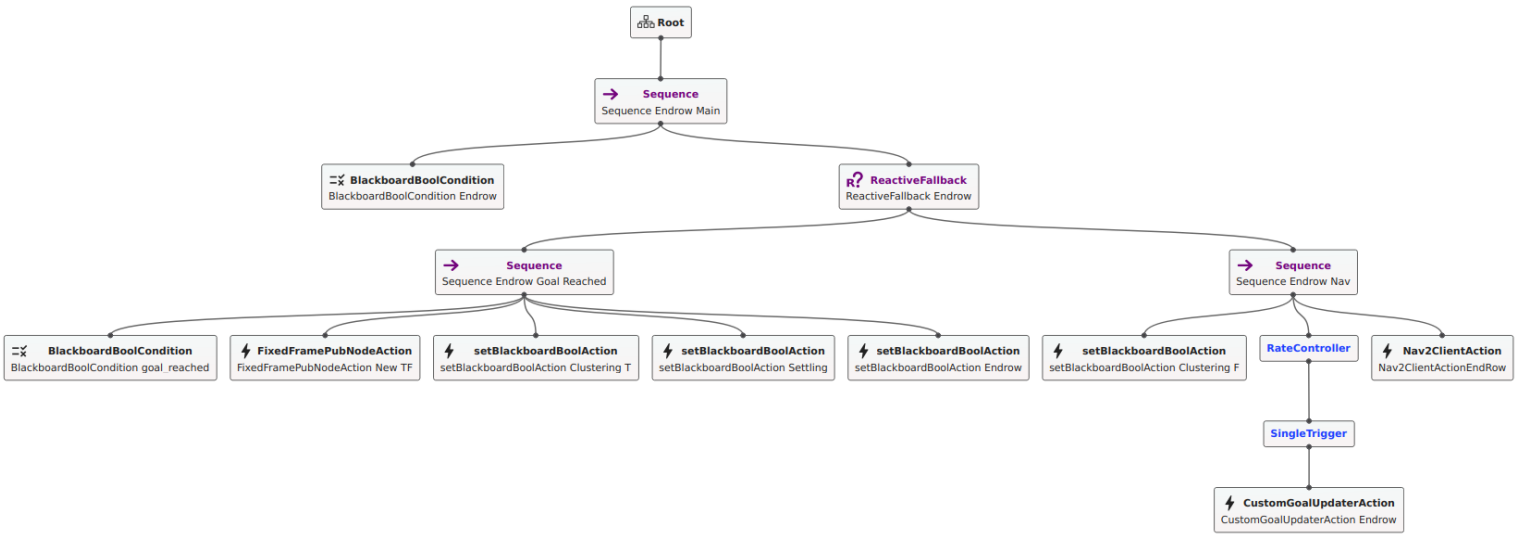


Figure 3.14: Illustration of the Endrow Subtree from the Position Agnostic MainBT.

frame and addresses the issue that the fixed *Map* frame may not be aligned with the rows, which could lead to drift during navigation.

The publishing and updating of the *Local* frame is handled by the *FixedFramePubNodeAction* custom BT node. This node copies the transform between two existing frames, stores it in a blackboard key, and publishes it when ticked. For this application, it copies the transform from the *Map* frame to the *Base Link* frame, with the latter representing the robot's current position and orientation.

The node is deployed in two parts of the system. At the top of the MasterBT (see Fig. 3.12), it publishes the saved transform at each tick. Initially, it also computes the transform at startup, this functionality is later disabled by the *Fallback First TF* so that the node only publishes the transform. In the Endrow Subtree (see Fig. 3.14), the node is placed as a child of the *Sequence Endrow Goal Reached* control node and is ticked only when the final goal of the end row trajectory is reached. At that point, it updates and publishes the transform.

Finally, two additional goals are appended to the computed circular trajectory to ensure that the robot completely exits the current row and enters the next one. This additional step is essential to guarantee that sufficient LiDAR data is available for effective clustering when going into in row navigation.

3.4.4 End of Row Detection

The switch between the Inrow and Endrow Subtrees is based on the detection performed by the *RowLinesNodeAction*. In addition to its other functionalities described in **Section 3.2.2**, this node also detects if the robot has reached the end of row. It does so through two metrics: the deviation from the current slope averages and distance from the the current x_{max} values. When the slopes of the incoming row lines deviate from the average beyond a predetermined threshold, it indicates that the clustering was unable to properly

segment the data. This situation typically arises when too few data points are given to the clustering algorithm, which in turn tends to happen as the robot approaches the end of a row. To enhance reliability, the current odometry is compared to the average x_{max} values, checking if their distance falls between a certain threshold. This way, even if the clustering does not completely degenerate, the algorithm can still reliably detect the end of a row.

Chapter 4

Simulation Evaluation of Algorithms

4.1 Introduction

To evaluate the performance of the two developed control algorithms, simulation experiments were conducted. Data were collected and specific metrics defined to assess the algorithms' robustness and reactivity.

4.2 Simulation Setup

4.2.1 Environment and Tools

The simulations were conducted in Gazebo Classic, an open-source software for robot and environment simulation that is compatible with ROS Humble.

4.2.2 Experimental Configuration

The Clearpath Jackal Unmanned Ground Vehicle (UGV) was employed. Its available GitHub repositories (*Jackal* and *Jackal Simulator*) provide an interface to launch Nav2 and run simulations in Gazebo. Additionally, also providing a highly configurable robot description. Which was used to integrate the Jackal's 3D model and multiple sensors (e.g., cameras, IMUs, LiDARs, and GPS antennas) via its Universal Robot Description File (URDF). For this application, the IMU, LiDAR (Hukyo st100), and GPS antenna (Novatel Smart6) were enabled, as shown in Fig. 4.1. All these sensors were simulated using their respective standard Gazebo libraries.

Experimental Set up

The simulations were conducted in a custom Gazebo world featuring multiple straight vineyard rows (see Fig. 4.2). The AGV was tasked with navigating three consecutive rows, detecting the row walls for navigation and the end of a row, switching accordingly between in-row and end-of-row navigation. The AGV, is positioned at the start of the

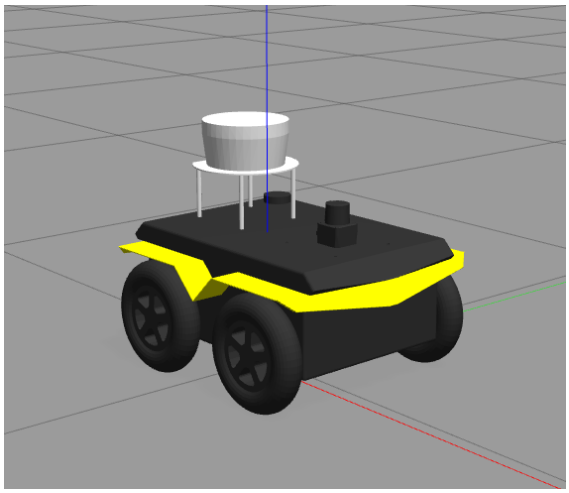


Figure 4.1: Clearpath Jackal model with enabled sensors in Gazebo.

right-most row. Running the *Tree executor node* the control BT, corresponding to one of the two algorithms is chosen via a terminal interface, loaded and ticked at a rate of 10Hz. The end mission condition, is triggered differently depending on the algorithm at hand. With the Baseline, if the last waypoint is reached navigation is considered successful. On the other hand the Position Agnostic algorithm, relies on a down counter which is called each time a new end-of-row is detected. The starting value of the down counter is defined before hand, and once it reaches zero the navigation is considered as completed. Which then needs to be verified, as either a successful or failing total path coverage. Relevant data were collected using the *Rosbag2* ROS package and subsequently visualized with *PlotJuggler*. More importantly the recorded ros bags, are then processed by the *Metrics node*. This node is subscribed to the odometry, row lines and end mission topics. From these then it computes the performance metrics described in the next section. These are then visualized through plots via the *Plot metrics node*, levaraging the *matplotlib* Python library.

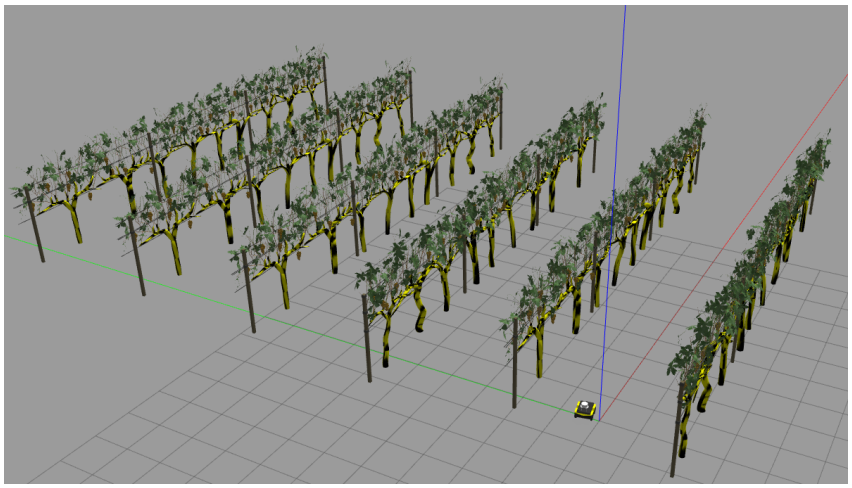
4.2.3 Baseline Algorithm Simulation

Standard Conditions (Fig. 4.3)

The AGV is spawned in the center of the first row and no manipulation is performed to inject errors.
The algorithm performs somewhat well, remaining centered with some margin and detecting correctly end-of-row conditions with no false positives or negatives. However it failed one of the attempts for full path navigation.

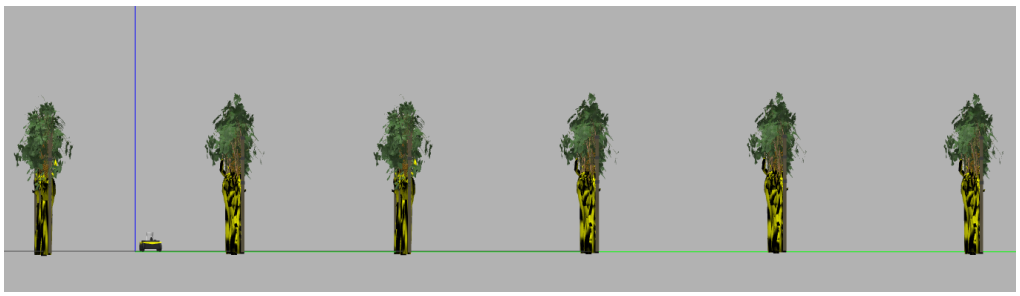
Misaligned Start Condition (Fig. 4.4)

The AGV is spawned with an offset, of -1 meters along y, w.r.t. the center of the first row and no manipulation is performed to inject errors.



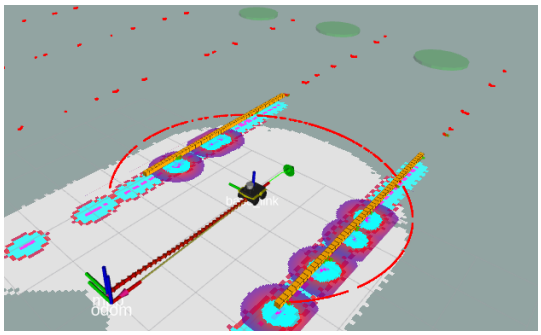
(a) Perspective view

(b) Top view

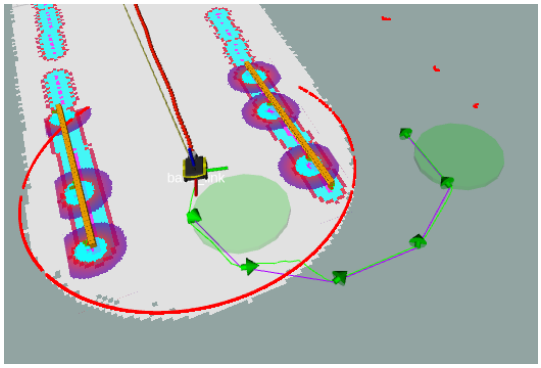


(c) Front view

Figure 4.2: Multiple views of the utilized Gazebo world.



(a) In-Row behavior



(b) End-Row behavior

Figure 4.3: Visualization in Rviz of the Baseline algorithm typical simulation behavior, in green the GPS waypoints.

The algorithm performs sufficiently, it is able to somewhat correct the initial offset however it never truly reaches an acceptable distance from the center line. It detected correctly end-of-row conditions with no false positives or negatives. However it failed only one of the attempts for full path navigation.

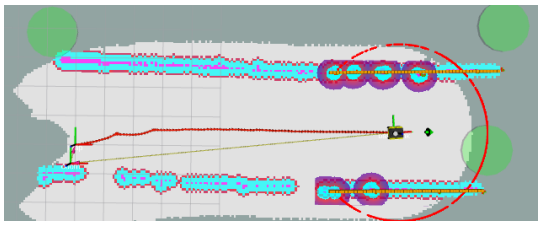
Navigation Error Injection Via Teleop (Fig. 4.5)

The AGV is spawned in the center of the first row and manipulation is performed to inject errors during navigation, by overwriting the control action through teleoperation. The algorithm performs sufficiently, it is able to somewhat correct the initial offset however it never truly reaches an acceptable distance from the center line. It detected correctly end-of-row conditions with no false positives or negatives. However it failed one of the attempts for full path navigation.

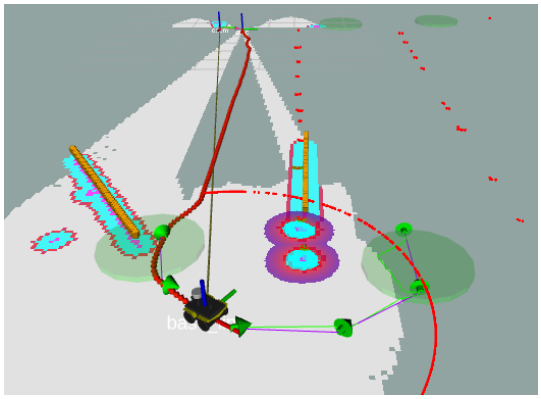
4.2.4 Position Agnostic Algorithm Simulation

Standard Conditions (Fig. 4.6)

The AGV is spawned in the center of the first row and no manipulation is performed to inject errors.



(a) In-Row behavior



(b) End-Row behavior

Figure 4.4: Visualization in Rviz of the Baseline algorithm simulation behavior for a misaligned start, in green the GPS waypoints.

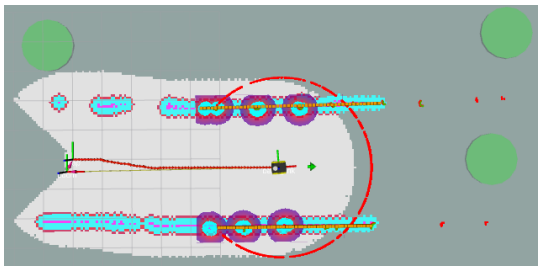
The algorithm performs well, remaining centered and detecting correctly end-of-row conditions with no false positives or negatives. Moreover all attempts resulted in successful full path navigation.

Misaligned Start Condition (Fig. 4.7)

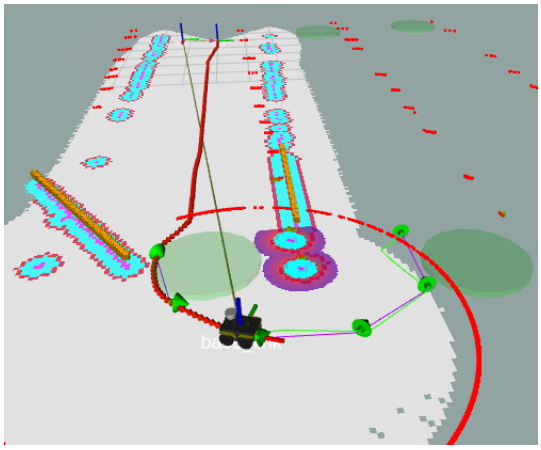
The AGV is spawned with an offset, of -1 meters along y, w.r.t. the center of the first row and no manipulation is performed to inject errors.
The algorithm performs well, it is able to correct the initial offset while remaining centered for all rows thanks to the dynamic navigation frame. It detected correctly end-of-row conditions with no false positives or negatives. Moreover all attempts resulted in successful full path navigation.

Navigation Error Injection Via Teleop (Fig. 4.8)

The AGV is spawned in the center of the first row and manipulation is performed to inject errors during navigation, by overwriting the control action through teleoperation.
The algorithm performs well, it is able to correct the initial offset while remaining centered for all rows thanks to the dynamic navigation frame. It detected correctly end-of-row conditions with no false positives or negatives. Moreover all attempts resulted in successful full path navigation.



(a) In-Row behavior



(b) End-Row behavior

Figure 4.5: Visualization in Rviz of the Baseline algorithm simulation behavior for an injected disturbance through teleop, in green the GPS waypoints.

4.3 Evaluation Metrics

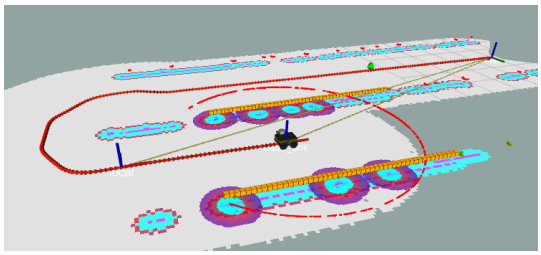
This section describes the metrics computed from the simulation data. Some metrics are evaluated in terms of both accuracy and precision. Accuracy quantifies how close the measured values are to the true or accepted values, while precision assesses the repeatability or consistency of the measurements, independent of their proximity to the true value.

4.3.1 Deviation from Row Center

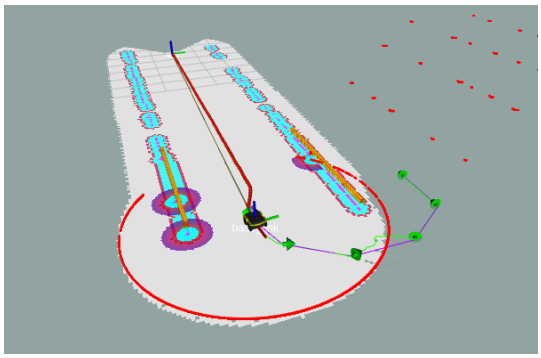
This metric is defined as the distance, in meters, between the robot's odometry position at time t and the ideal line representing the center of a row.

4.3.2 Average Time for Full Path Coverage

This metric represents the average time required for the robot to navigate the target path, which consists of three vine rows.



(a) In-Row behavior



(b) End-Row behavior

Figure 4.6: Visualization in Rviz of the Position Agnostic algorithm typical simulation behavior, note the dynamic navigation frame *Local*.

4.3.3 Endrow Detection Accuracy and Precision

The metrics for endrow detection, at the end of a row, are defined as follows:

$$\text{Accuracy (\%)}: \quad A_{endrow} = \frac{\text{Number of Correct Detections}}{\text{Total Detections}} \times 100, \tag{4.1}$$

$$\text{Precision (\%)}: \quad P_{endrow} = \frac{\text{Number of Correct Detections}}{\text{Number of Correct Detections} + \text{Number of Incorrect Detections}} \times 100. \tag{4.2}$$

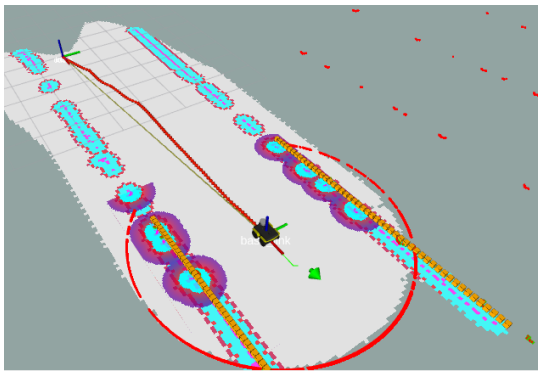
4.3.4 Row Lines Detection Accuracy and Precision

For each row line parameter, p (where p represents parameters such as m and q), the metrics are defined as:

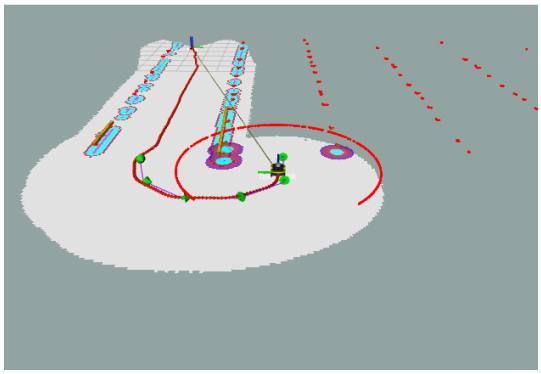
$$\text{Accuracy}: \quad A(p) = \frac{|p_{avg} - p_{\text{true}}|}{p_{\text{true}}}, \tag{4.3}$$

$$\text{Precision}: \quad P(p) = \sqrt{\frac{\sum_{i=1}^{N} (p_i - p_{\text{avg}})^2}{N}}, \tag{4.4}$$

where p_{true} is the true parameter value, p_{avg} is the average of the measured values, and N is the number of measurements.



(a) In-Row behavior



(b) End-Row behavior

Figure 4.7: Visualization in Rviz of the Position Agnostic algorithm simulation behavior for a misaligned start, note the dynamic navigation frame *Local*.

4.3.5 Success Rate

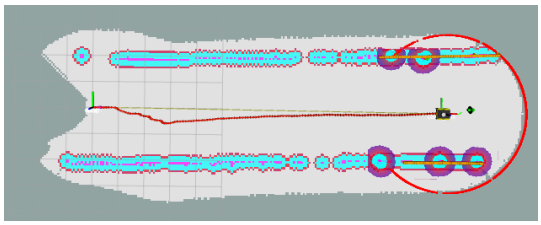
The success rate is defined as the percentage of simulations in which the robot successfully completed the full exploration of the target path (i.e., all three vine rows).

4.4 Results

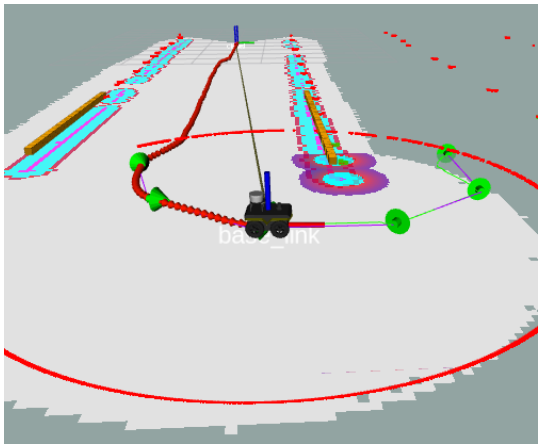
4.4.1 Algorithm 1 Results

The Baseline algorithm performed well under the ideal simulation conditions. It achieved a success rate of 75% across all test scenarios, with the only failures due to an artifact of clustering which led to a failure of the navigation controller. The detection accuracy and precision for the end-of-row identification were 100% for all test scenarios, thanks to the high precision of the simulated GNSS signals. In standard conditions,the average traveled path was of $70.94, m$ and completed, on average in approximately $254.91 s$ (around 4 minutes and 25 seconds), as shown in Fig. 4.9. For the other two scenarios the travel time and distances were higher due to the introduced disturbances.

Issues arose in the detection of row lines. Clustering artifacts near the row endings



(a) In-Row behavior



(b) End-Row behavior

Figure 4.8: Visualization in Rviz of the Position Agnostic algorithm simulation behavior for an injected disturbance through teleop, note the dynamic navigation frame *Local*.

resulted in significant performance degradation, particularly affecting the q parameter across all scenarios (see Fig.4.11, Fig.4.14 and Fig.4.17). In contrast, the metrics for the m parameter remained robust, with accuracy maintained at or above 95% and precision slightly below 80% (see Fig.4.10, Fig.4.13 and Fig.4.16). Being the Average Center Deviation reliant on good estimation of both the line parameters, it is expected that this metric will also suffer. Leading in the worst case, during the navigation of Row 3, to a deviation of nearly one meter (see Fig.4.12, Fig.4.15 and Fig.4.18).

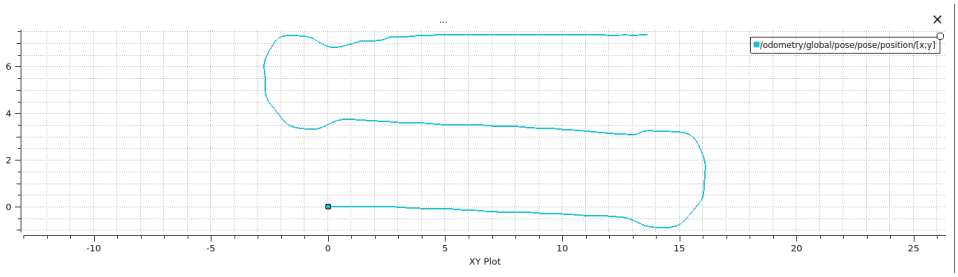
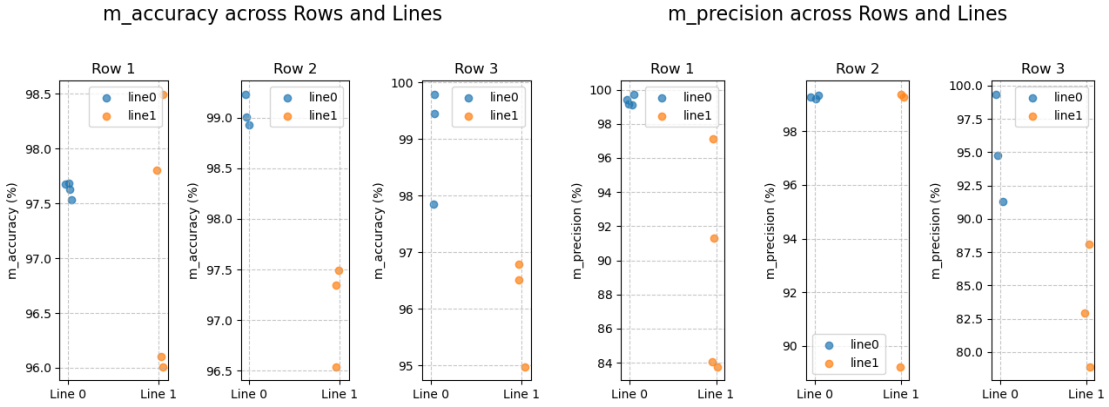


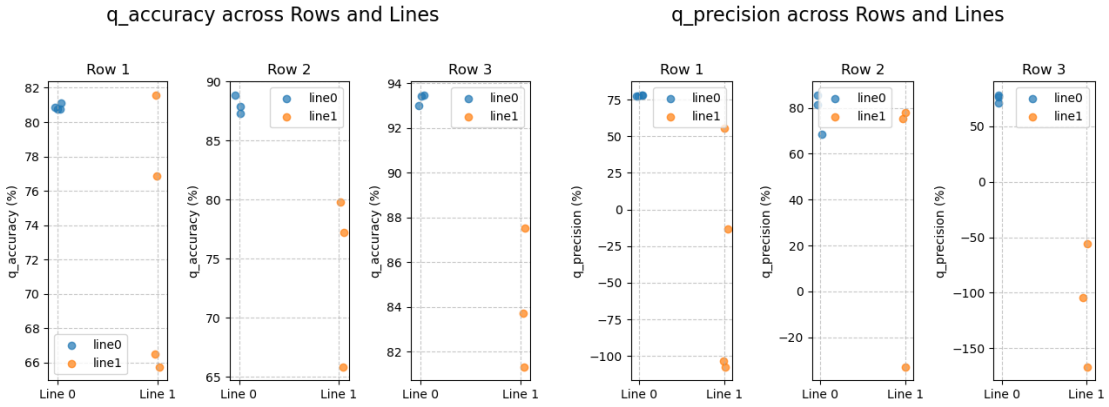
Figure 4.9: Baseline Algorithm Standard Conditions: Odometry data from simulation, representing the trajectory of the AGV.

Standard Conditions Results (Figs. 4.10, 4.11, 4.12)



(a) (b)

Figure 4.10: Baseline algorithm Standard Conditions: computed metrics in % for detected line slope (m), from the simulated row walls of the three vineyard rows.



(a) (b)

Figure 4.11: Baseline algorithm Standard Conditions: computed metrics in % for detected line intercept (q), from the simulated row walls of the three vineyard rows.

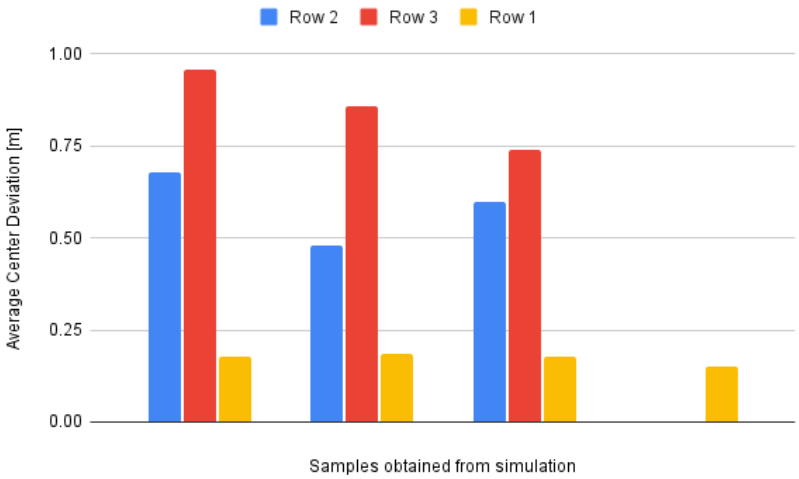
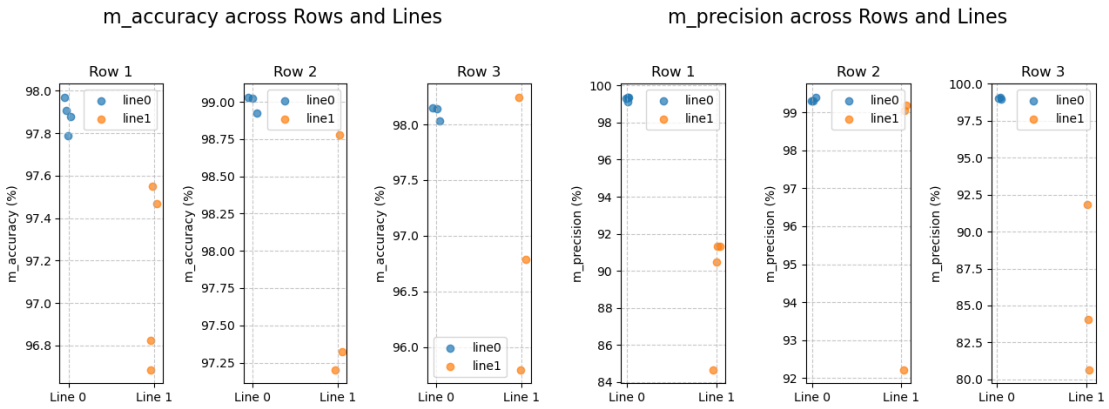


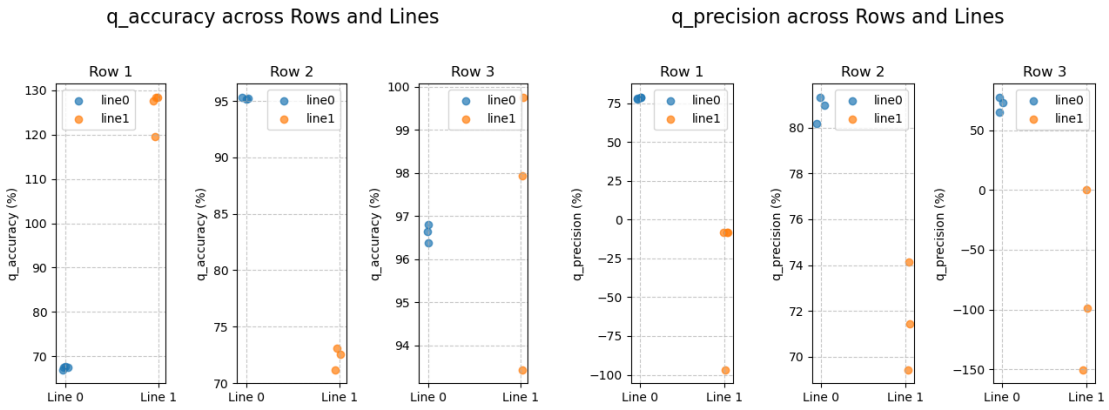
Figure 4.12: Baseline algorithm Standard Conditions: Average deviation from row center.

Misaligned Start Results (Figs. 4.13, 4.14, 4.15)



(a) (b)

Figure 4.13: Baseline algorithm Misaligned Start: computed metrics in % for detected line slope (m), from the simulated row walls of the three vineyard rows.



(a) (b)

Figure 4.14: Baseline algorithm Misaligned Start: computed metrics in % for detected line intercept (q), from the simulated row walls of the three vineyard rows.

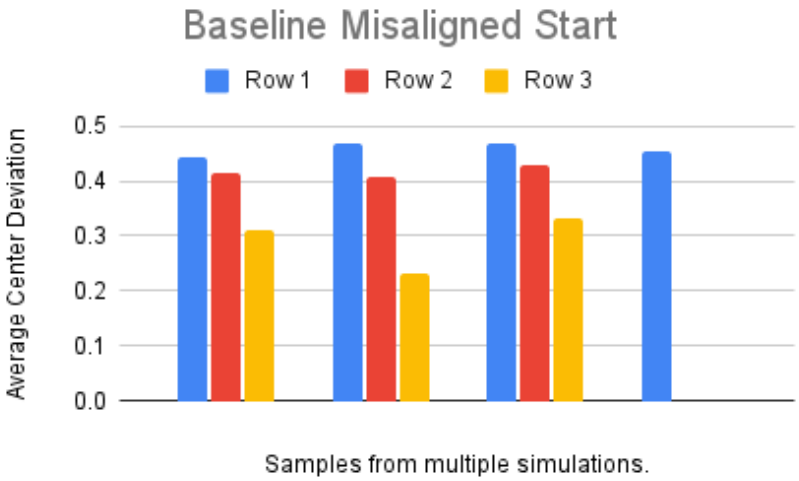
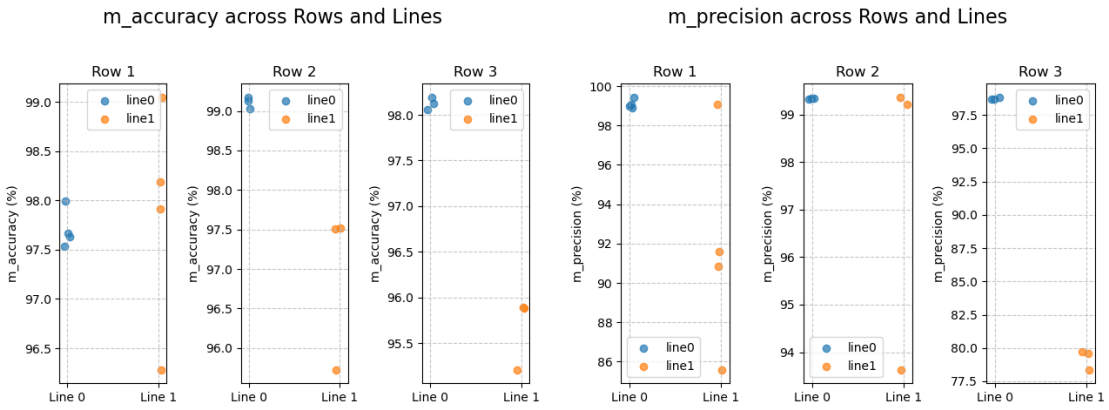


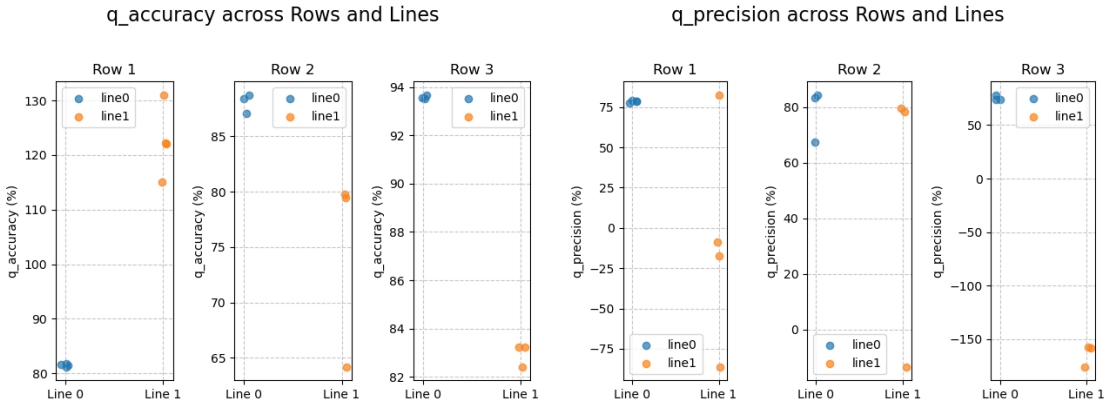
Figure 4.15: Baseline algorithm Misaligned Start: Average deviation from row center.

Teleop Injected Error Results (Figs. 4.16 4.17, 4.18)



(a) (b)

Figure 4.16: Baseline algorithm Teleop Injected Error: computed metrics in % for detected line slope (m), from the simulated row walls of the three vineyard rows.



(a) (b)

Figure 4.17: Baseline algorithm Teleop Injected Error: computed metrics in % for detected line intercept (q), from the simulated row walls of the three vineyard rows.

4.4.2 Algorithm 2 Results

The Position Agnostic algorithm performed quite well, relying primarily on LiDAR data. It achieved a 100% success rate for all scenarios. Consequently, both the accuracy and precision for end-of-row detection scored 100% for all. In standard conditions, the average traveled path was of $74.5m$, on average in approximately $264.14s$ (around 4 minutes and 35 seconds), as shown in Fig. 4.19.

Regarding the line row parameters, the algorithm generally performed well. However, the worst performance was observed in the precision and accuracy of the q parameter, which at times dropped below 80% (see Fig.4.21, Fig.4.24 and Fig.4.27). On the other hand, the metrics for the m parameter remained very robust, with accuracy staying above

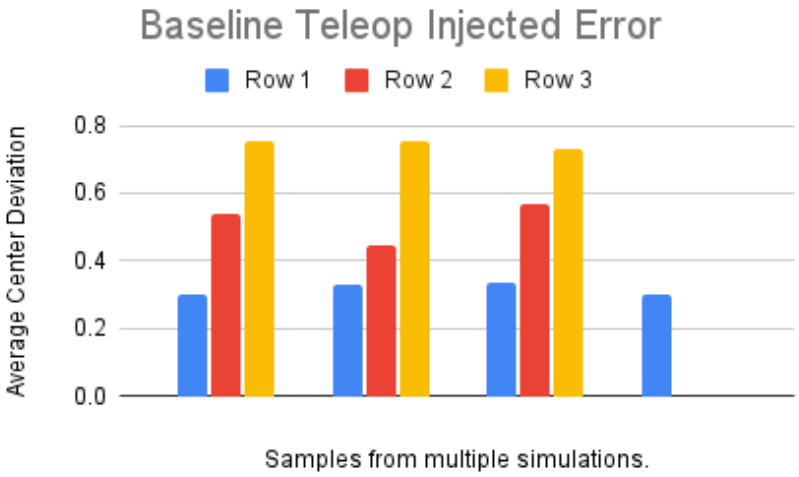


Figure 4.18: Baseline algorithm Teleop Injected Error: Average deviation from row center.

93% and precision above 80% even in the worst cases (see Fig.4.20, Fig.4.23 and Fig.4.26). Due to the general better performance of the line parameter identification, the Average Center Deviation remained at most $0.25m$ across all scenarios.

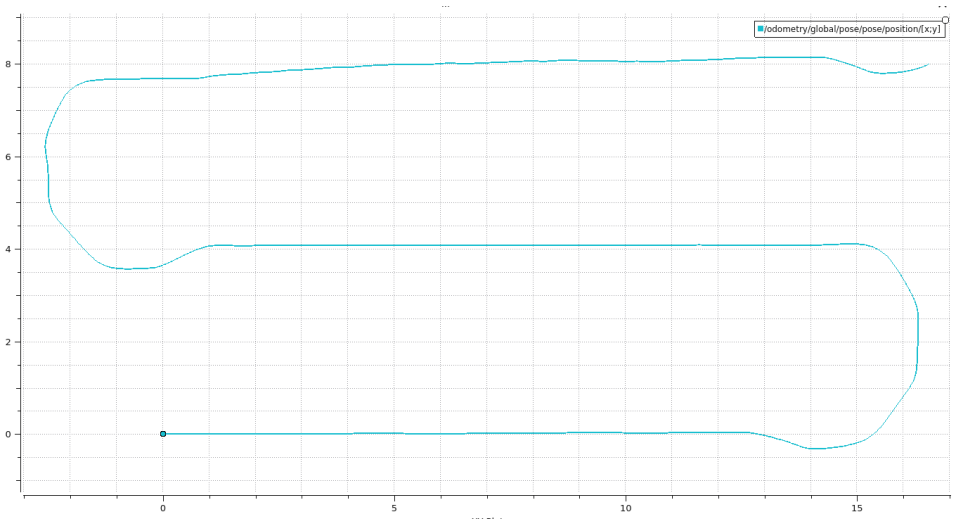
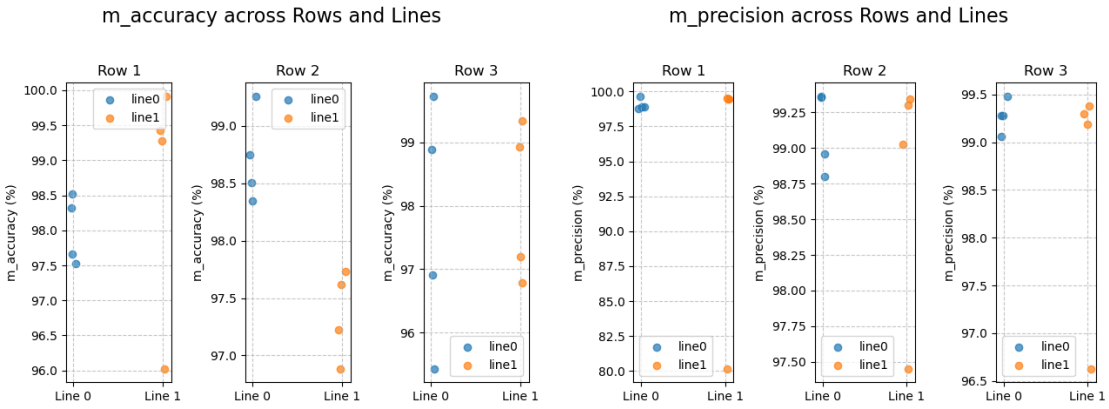


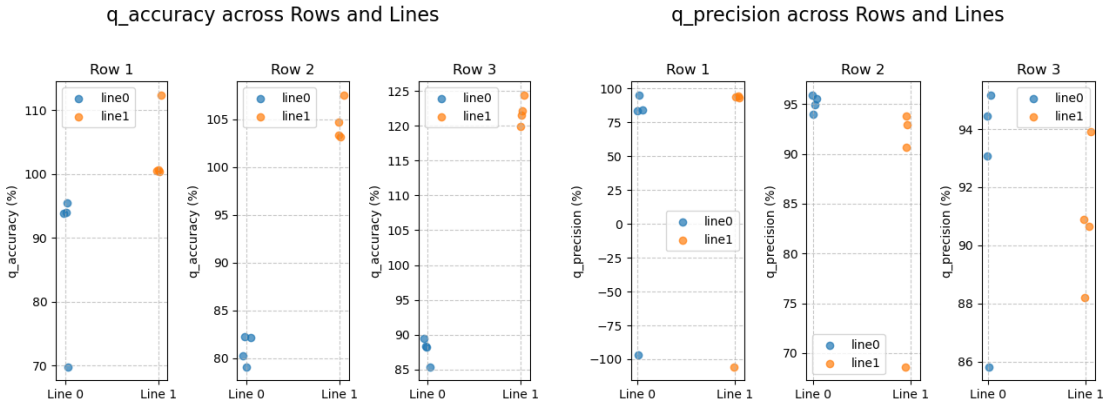
Figure 4.19: Position Agnostic Algorithm Standard Condition: Odometry data from simulation, representing the trajectory of the AGV.

Standard Conditions Results (Fig. 4.20, 4.21, 4.22)



(a) (b)

Figure 4.20: Position Agnostic algorithm Standard Conditions: computed metrics in % for detected line slope (m), from the simulated row walls of the three vineyard rows.



(a) (b)

Figure 4.21: Position Agnostic algorithm Standard Conditions: computed metrics in % for detected line intercept (q), from the simulated row walls of the three vineyard rows.

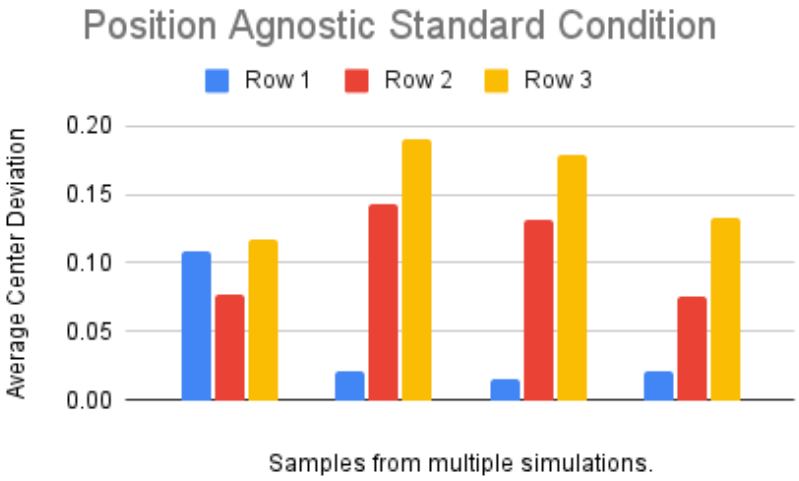
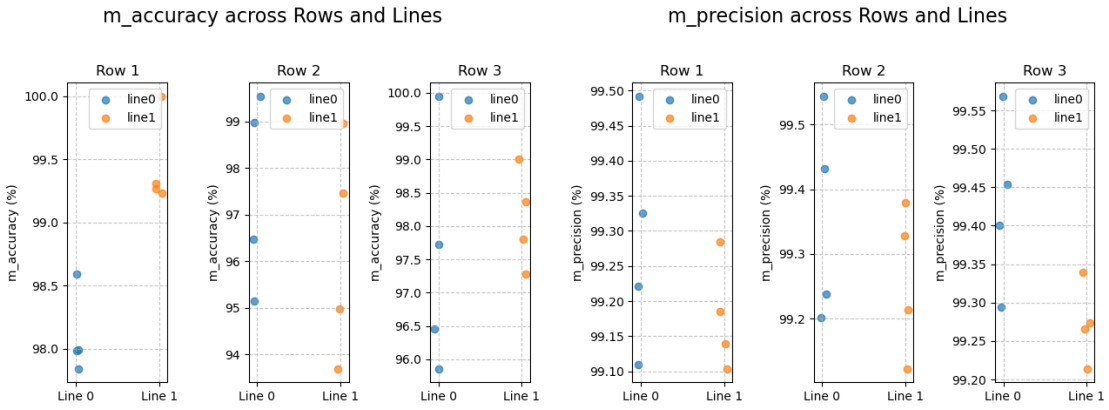


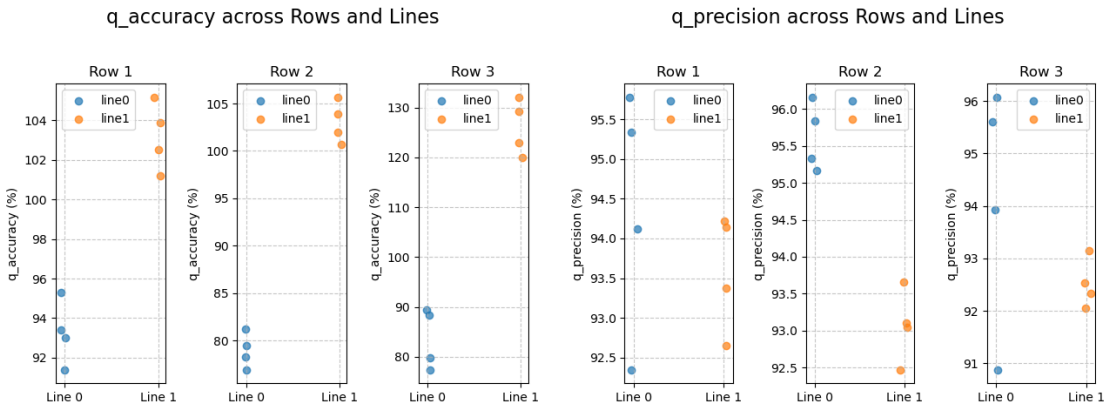
Figure 4.22: Position Agnostic algorithm Standard Conditions: Average deviation from row center.

Misaligned Start Results (Fig. 4.23, 4.24, 4.25)



(a) (b)

Figure 4.23: Position Agnostic algorithm Misaligned Start: computed metrics in % for detected line slope (m), from the simulated row walls of the three vineyard rows.



(a) (b)

Figure 4.24: Position Agnostic algorithm Misaligned Start: computed metrics in % for detected line intercept (q), from the simulated row walls of the three vineyard rows.

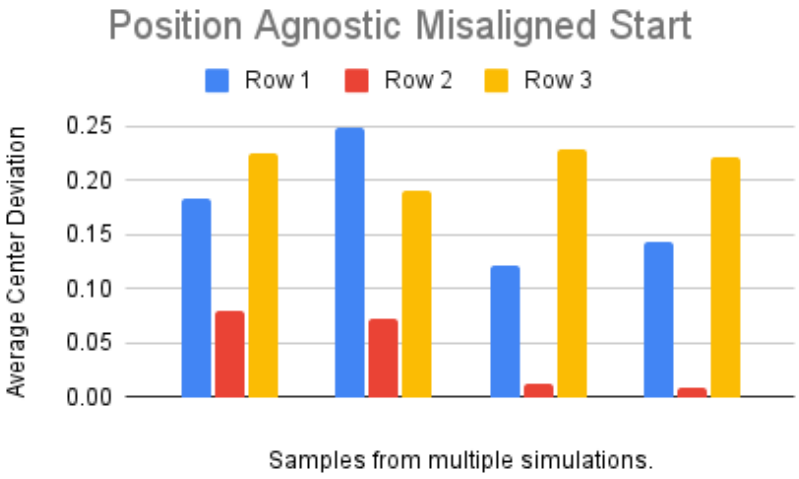
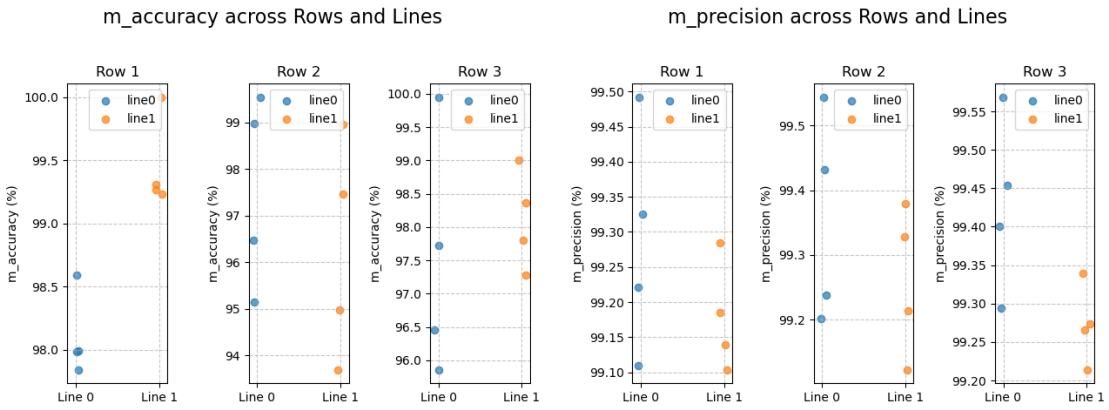


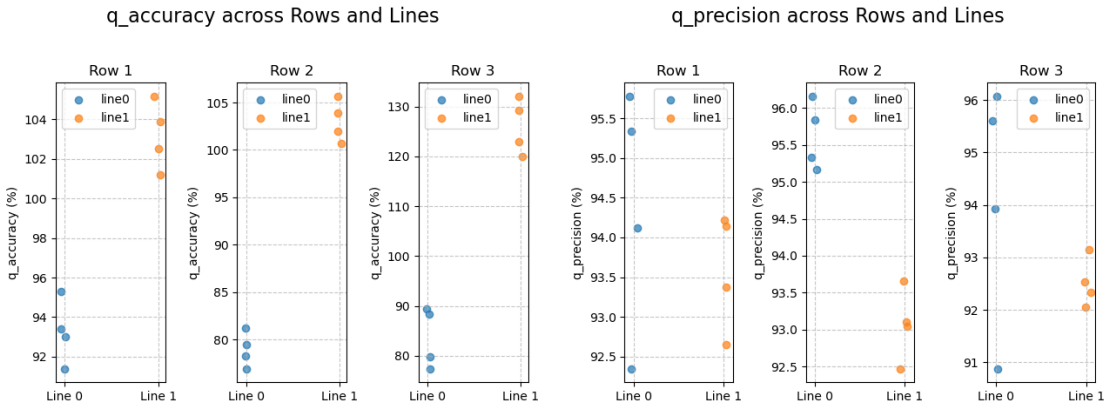
Figure 4.25: Position Agnostic algorithm Misaligned Start: Average deviation from row center.

Teleop Injected Error Results (Fig. 4.26, 4.27, 4.28)



(a) (b)

Figure 4.26: Position Agnostic algorithm Teleop Injected Error: computed metrics in % for detected line slope (m), from the simulated row walls of the three vineyard rows.



(a) (b)

Figure 4.27: Position Agnostic algorithm Teleop Injected Error: computed metrics in % for detected line intercept (q), from the simulated row walls of the three vineyard rows.

4.4.3 Comparative Analysis

The Baseline algorithm leverages GNSS data, which under ideal simulation conditions yields high precision for end-of-row detection. However, this could become a vulnerability in real-world applications where GNSS signals may be less reliable. On the other hand, the Position Agnostic algorithm, primarily driven by LiDAR data, demonstrates superior consistency in maintaining a centered path and in row line detection.

These observations suggest that while GNSS has exceptional accuracy under controlled conditions, its performance might degrade in environments with signal interference or obstructions, such as vineyards. On the other hand, the Position Agnostic method shows promise for real-world scenarios by mitigating these issues.

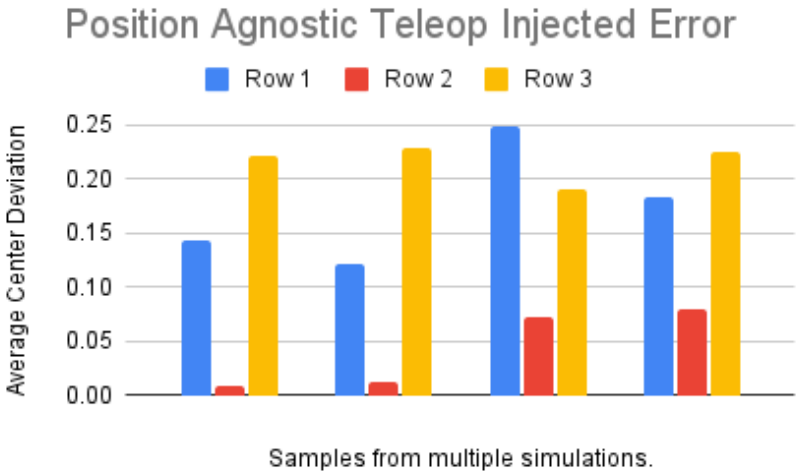


Figure 4.28: Position Agnostic algorithm Teleop Injected Error: Average deviation from row center.

Chapter 5

Conclusion

In summary, this thesis compared two distinct algorithms for row line detection in agricultural navigation. The Baseline algorithm, leveraging GNSS data, achieved high precision under ideal conditions but may face challenges in real-world applications. Particularly in challenging environments such as orchards or vineyards, which due to dense vegetation could weaken GNSS signals, rendering them unreliable. In contrast, the Position Agnostic algorithm, primarily based on LiDAR, maintained a more centered path and showed robust performance despite some metric fluctuations. For this reason a viable approach would consider fusing the two, leveraging the modular structure of BTs, switching or fusing the two for an even more robust end-of-row navigation. Leveraging GNSS where conditions are favorable while defaulting to a LiDAR-based approach when challenging environments are encountered. Such integration could potentially enhance overall system reliability and performance. Additionally, further testing under varied environmental conditions would help to refine the algorithms and validate their practical applicability in a real world scenario.

Bibliography

[1] Ieee standard for neural network-based image coding. *IEEE Std 1857.11-2024*, pages 1–159, 2024.

[2] Riccardo Bertoglio, Veronica Carini, Stefano Arrigoni, and Matteo Matteucci. A map-free lidar-based system for autonomous navigation in vineyards. In *2023 European Conference on Mobile Robots (ECMR)*, pages 1–6, 2023.

[3] Simone Cerrato, Vittorio Mazzia, Francesco Salvetti, Mauro Martini, Simone Angarano, Alessandro Navone, and Marcello Chiaberge. A deep learning driven algorithmic pipeline for autonomous navigation in row-based crops. *IEEE Access*, 12:138306–138318, 2024.

[4] Michele Colledanchise and Petter Ögren. Behavior trees in robotics and ai, July 2018.

[5] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[6] Alessandro Navone, Mauro Martini, Marco Ambrosio, Andrea Ostuni, Simone Angarano, and Marcello Chiaberge. Gps-free autonomous navigation in cluttered tree rows with deep semantic segmentation, 2024.

[7] Petter Ögren and Christopher I Sprague. Behavior trees in robot control systems. *Annual Review of Control, Robotics, and Autonomous Systems*, 5(1):81–107, 2022.

[8] F.M. Rico. *A Concise Introduction to Robot Programming with ROS2 (1st ed.)*., volume 1. Chapman and Hall/CRC., 2022.

[9] Conrad Sanderson and Ryan Curtin. An open source c++ implementation of multithreaded gaussian mixture models, k-means and expectation maximisation. In *2017 11th International Conference on Signal Processing and Communication Systems (ICSPCS)*, pages 1–8, 2017.

[10] Conrad Sanderson and Ryan Curtin. Practical sparse matrices in c++ with hybrid storage and template-based expression optimisation. *Mathematical and Computational Applications*, 24(3):70, July 2019.

[11] Conrad Sanderson and Ryan Curtin. Armadillo: An efficient framework for numerical linear algebra, 2025.

[12] Christopher Iliffe Sprague, Özer Özkahraman, Andrea Munafo, Rachel Marlow, Alexander Phillips, and Petter Ögren. Improving the modularity of auv control systems using behaviour trees. In *2018 IEEE/OES Autonomous Underwater Vehicle Workshop (AUV)*, pages 1–6, 2018.

[13] Michiel Van Dijk, Tom Morley, Marie Luise Rau, and Yashar Saghai. A meta-analysis of projected global food demand and population at risk of hunger for the

period 2010–2050. *Nature Food*, 2(7):494–501, 2021.