BEng (Hons) Final Year Project Report:

TURTLEBOT(s) Navigation: Fast and Furious (but Provably Safe)

Computer Engineering
Queen's University Belfast

Student Name: Patrick McLaughlin

Student Number: 40294886

Supervisor: Dr. Nikolaos Athanasopoulos

Date: August 2025

Abstract

This project developed a custom Near-Identity Diffeomorphism (NID) enhanced local planner for TurtleBot3, improving hardware navigation success rate from 20% to 60%. my NID-DWB controller, implemented as a C++ plugin for use in the ROS 2 and Nav2 stacks, transforms single-integrator velocity commands into feasible unicycle controls using mathematical coordinate mapping, enabling us to use advanced control theory on resource-constrained hardware. Through systematic simulation validation, an optimal lookahead distance of 0.15m was identified in Gazebo simulation and successfully transferred to hardware. The NID controller demonstrated superior behavioral safety compared to the standard Dynamic Window Approach (DWB), maintaining larger obstacle clearances and mostly eliminating oscillatory motion, at the cost of reduced traversal speed when DWB did succeed.

Results reveal a fundamental safety-performance trade-off: while the NID-DWB controller achieves triple the success rate with more predictable trajectories- essential for formal safety verification-it sacrifices some path-tracking accuracy. This work demonstrates that the NID-based approach is a practical method for improving navigation on resource-constrained hardware. By generating kinematically feasible unicycle commands directly from the global path, it bypasses the computationally expensive trajectory sampling and scoring stages of the standard DWB planner. This efficiency gain is what allows for more reliable and safer performance on hardware with limited processing power, establishing a stable foundation for future work in provably safe navigation.

Project Specification

The following objectives were set for the project, as specified in the official project brief [4]:

1. Learn about mobile robot dynamics and planning methods.
2. Assemble the TurtleBot3 platform and interface it with ROS.
3. Demonstrate basic autonomous movement, e.g., straight-line motion.
4. Develop and implement algorithms to generate safe sets for the robot dynamics.
5. Set up test scenarios with corresponding performance costs and conduct systematic experiments in a test area.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr. Nikolaos Athanasopoulos, for his guidance, support, and encouragement throughout this project. I am also extremely thankful to the technical support staff on the Ninth floor and for research staff such as Stephen McIlvana and Ryan Mckee and the other colleagues in the School of EEECS at Queen's University Belfast for their assistance with laboratory resources and troubleshooting. I'd also like to extend a special thanks to my family, friends and loved ones for their patience and support during my studies.

Declaration of Originality

I, Patrick McLaughlin, declare that this dissertation is entirely my own work, and that it has not been submitted for any other degree or professional qualification. Where I have made use of the work of others, it is duly acknowledged in the text and bibliography.

Signature: 

Date: 15th August 2025

Contents

List of Figures

List of Tables

List of Symbols

v	Linear velocity of the robot
ω	Angular velocity of the robot
θ	Heading angle of the robot
l	Lookahead distance parameter in NID controller
(x, y)	Cartesian coordinates of the robot in the global frame
$\dot{s}(x)$	Velocity of the virtual lookahead point
$[u_x, u_y]^T$	Desired single-integrator velocity commands
K_{lat}	Lateral error correction gain
$SE(2)$	Special Euclidean group in 2 dimensions, representing robot configuration space

Chapter 1

Introduction

1.1 Context and Motivation

Autonomous mobile robots are increasingly deployed in complex and dynamic indoor environments, from warehouses and hospitals to domestic settings, where precise and reliable navigation is paramount for task completion and safety. The Robot Operating System 2 (ROS2) and its accompanying Navigation2 (Nav2) stack have emerged as the de facto open-source framework for developing such robotic applications, providing a robust, modular architecture for robot navigation [10]. Within this context, local planners are responsible for generating real-time velocity commands that guide the robot along a global path while avoiding immediate obstacles.

A widely used local planner is the Dynamic Window Approach (DWA) and its ROS2 variant, the DWB controller [5]. While computationally efficient, these purely reactive planners often exhibit suboptimal behaviors, particularly in confined spaces or during the final goal approach phase. Common failure modes include pronounced oscillatory motion, inefficient path following, and an inability to execute smooth trajectories in cluttered environments. These limitations become even more pronounced on differential-drive platforms like the TurtleBot3 Burger, where the nonholonomic constraints of the wheeled base (cannot instantly translate in any direction) [17]necessitate carefully generated velocity commands to achieve stable and efficient motion.

1.2 Problem Statement

While the ROS2 Nav2 stack is a production-grade framework, its real-world performance is highly dependent on the efficacy of its plugins, particularly on resource-constrained hardware. The standard DWB local planner, while effective in some scenarios, was found during preliminary investigations for this project to be fundamentally unreliable in the confined, cluttered test environment. Its purely reactive nature, which relies on short-term trajectory simulations, led to a catastrophic 80% navigation failure rate on the physical TurtleBot3,

characterised by:

- **Oscillatory Failure:** A tendency to become trapped in oscillatory loops when approaching the goal, preventing task completion.
- **Reactive Myopia:** An inability to anticipate path curvature, leading to inefficient, jagged paths that often resulted in collisions.
- **Parameter Inviability:** An exhaustive, multi-week parameter tuning process revealed that no combination of settings could reliably solve the controller's architectural limitations on the target hardware.

These findings identified a critical problem in my context: the standard controller was not a viable foundation for reliable navigation, let alone for the future implementation of formal safety methods. This was the motivation to develop a more robust local planning solution.

1.3 Project Aim and Proposed Approach

While the project's high-level ambition was the implementation of reachability analysis for provably safe navigation, my systematic empirical investigation revealed a more fundamental and pressing prerequisite. The baseline Nav2 DWB controller, when deployed on the project's resource-constrained hardware, exhibited a catastrophic 80% failure rate. This finding demonstrated that the controller's limitations were not merely parametric, but architectural; its purely reactive design is fundamentally unsuited for producing the stable, predictable trajectories required as a foundation for any formal safety verification.

Applying formal methods like reachability analysis, which require predictable system behaviour to provide meaningful guarantees, to this inherently unreliable controller setup would yield invalid results. Therefore, the project's primary aim was strategically adapted to first solve this foundational challenge. The first objective became to develop, integrate- and rigorously validate a stable and reliable controller (the NID-DWB) that could serve as the necessary precursor for future work in formal safety. While this necessary strategic pivot precluded the implementation of the final safe set objectives within the project's timeframe, it successfully established and validated the essential, reliable platform without which those objectives would be impossible.

The proposed approach replaces the reactive nature of DWB with a lookahead-based controller founded on a principled mathematical coordinate transformation known as a Near-Identity Diffeomorphism (NID), thereby providing a robust proof-of-concept for the viability of such advanced control methods on accessible robotics platforms.

1.4 Summary of Contributions

This dissertation presents the following original contributions to the field of practical mobile robot navigation:

1. **Systematic Baseline Failure Analysis:** The project's first contribution is the rigorous empirical analysis that quantitatively demonstrated the architectural failure of the standard DWB controller on resource-constrained hardware, revealing a consistent 80% navigation failure rate that could not be solved by parameter tuning alone.

2. **Practical Engineering Integration:** The second contribution is the successful implementation and integration of a theoretically complex NID-based controller into the production-grade, plugin-based C++ architecture of the ROS2 Nav2 stack, demonstrating a practical solution to the identified baseline failure.

3. **Quantitative Sim-to-Real Validation:** The project delivered a comprehensive empirical validation of this solution, including the identification of an optimal lookahead parameter (0.15m) and a tangible 200% increase in hardware navigation success rate, proving the practical viability of the approach.

4. **Open-Source Analysis Pipeline:** The design and implementation of a reusable data analysis pipeline, the methodology of which is detailed as pseudocode in **Appendix D**. This provides a valuable framework for the research community to perform systematic, quantitative evaluation of ROS2 navigation performance.

The primary contribution of this work is the demonstration that the NID-based controller provides a computationally efficient alternative to standard simulation-based planners like DWB, making it more suitable for resource-starved hardware. The NID transformation, by its mathematical nature, generates commands that inherently respect the nonholonomic constraints of a differential-drive robot, eliminating the need to sample and score a multitude of physically impossible trajectories (such as immediate lateral movements). This inherent efficiency is what leads to the improved reliability and safer performance documented in my results, providing a solid basis for future work.

1.5 Report Structure

This report is organized as follows: Chapter 2 reviews related work in robot navigation, from foundational concepts to contemporary research. Chapter 3 details the NID-based methodology and the experimental design. Chapter 4 describes the implementation and development process, software architecture, and the engineering challenges that were faced. Chapter 5 presents a mixture of quantitative and qualitative results from both simulation and hardware validation. Chapter 6 provides a deep analysis of these findings, their implications, and the project's limitations. Finally, Chapter 7 concludes with a summary of achievements, a reflection on the project's adaptive management, and directions for future work. The appendices contain supplementary materials, including extended results and other supplementary content.

Chapter 2

Background and Related Work

This chapter surveys the foundational principles and contemporary research that underpin mobile robot navigation. Here we establish a logical progression from the basic communication and perception concepts in robotics, through the architecture of a stack such as ROS2, to the research gap that this project aims to address, ultimately providing the necessary context for the proposed NID-based controller.

2.1 The ROS2 Ecosystem: A Framework for Robotics

The practical implementation of modern robotics relies on a standardised software framework. For this project, we used the Robot Operating System 2 (ROS2), following the official ROS2 Humble Hawksbill documentation [12] to inform my research and subsequent work.

2.1.1 Core Communication Concepts

ROS2 facilitates communication between different software components (called **nodes**) using a publish-subscribe model. Nodes communicate by passing messages via named **topics**. For example, a LiDAR driver node publishes sensor data on a `/scan` topic, and a navigation node subscribes to this topic to receive the data. For more complex, request-response interactions, ROS2 uses **services** (for synchronous communication) and **actions** (for asynchronous, long-running tasks like ”navigate to goal”). This modular, message-passing architecture is fundamental to my goal, building decoupled robotics systems that can communicate effectively.

2.1.2 Coordinate Frames and Transformations (TF2)

A robot must manage the spatial relationships between its various parts (e.g., its base, wheels, and sensors) and the outside world. ROS2 handles this using the **TF2 library**, which maintains a dynamic tree of coordinate frames. The key frames for mobile navigation are:

- **map:** A fixed, world-level frame that remains static over time. It is the global source of truth for the robot's absolute position.
- **odom:** A continuous, but drifting, frame originating at the robot's startup location. It provides smooth, short-term motion estimates from data taken from sources like the wheel encoders.
- **base_link:** The primary frame rigidly attached to the robot's geometric center.

The transformation from `map` to `odom` is continuously updated by the localisation system to correct for **odometry drift**—the inevitable accumulation of small errors from wheel-based motion estimates.

2.2 Perception and World Representation

For a robot to navigate, it must first perceive its environment and build an internal representation of it.

2.2.1 Sensing with LiDAR and SLAM

The primary sensor for this project is a 360-degree LiDAR, which provides an accurate point cloud of distance measurements of the surrounding environment. To perform any autonomous navigation, the robot first requires a static map of its environment to serve as its world representation.

For this project, these maps were generated as a distinct preliminary phase before any navigation took place. This process of creating a map while simultaneously tracking the robot's position within it is the fundamental challenge of **Simultaneous Localisation and Mapping (SLAM)** a challenge for which we employed Google's **Cartographer** SLAM algorithm . By manually teleoperating the robot through the test environment, we used Cartographer to process the raw LiDAR scans and odometry data. The output of this one-time process was a globally consistent and accurate 2D **occupancy grid map**, which was then saved for later use. In this map, each cell holds a probabilistic value representing the likelihood of that cell being occupied by an obstacle. This pre-generated map then served as the known, static environment for all subsequent localisation and navigation that occurred.

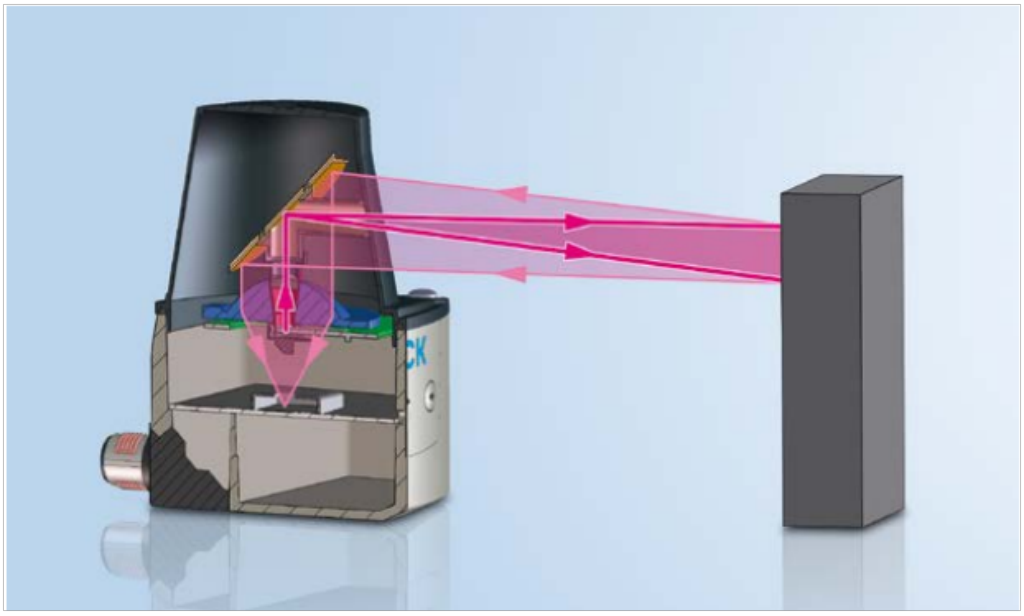


Figure 2.1: 2D scanning LiDAR operation: a single laser beam is deflected by a rotating mirror to sweep a horizontal plane, yielding distance–angle samples (r, θ) for each scan. In ROS 2 `sensor_msgs/LaserScan`, ranges are *floating-point metres* bounded by `range_min` and `range_max`; out-of-range returns are encoded as `+Inf`, too-close as `-Inf`, and invalid readings as `NaN` (REP-117) [16].

2.2.2 Localisation with AMCL

Once a map exists, the robot needs to know its position within it. **Adaptive Monte Carlo Localisation (AMCL)** [6] is the standard ROS2 algorithm for this. It uses a particle filter to estimate the robot's pose by constantly comparing live LiDAR scans to the known map. As particles (hypotheses of the robot's pose) that are inconsistent with the sensor readings are discarded, the cloud of particles converges on the robot's most likely position, correcting for odometry drift.

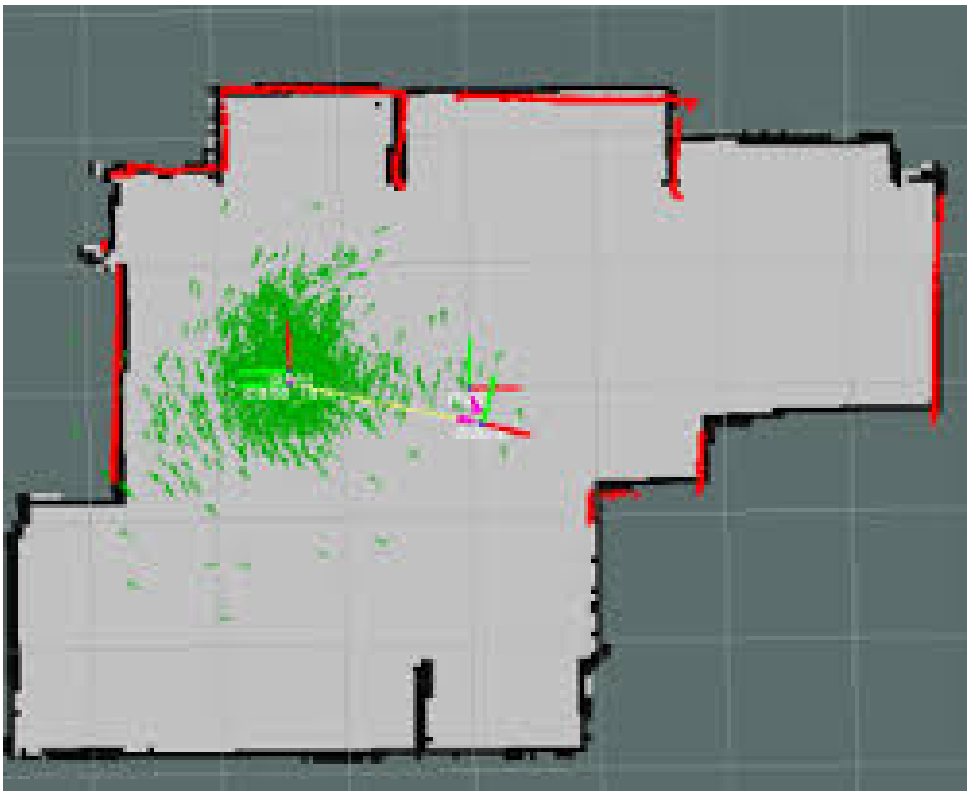


Figure 2.2: AMCL localization in RViz: the `/particlecloud` (PoseArray) visualizes pose hypotheses over the static occupancy map. A wide, dispersed cloud indicates high pose uncertainty; as laser scans and odometry are fused, particles converge to a tight cluster around the most likely pose and AMCL publishes the `map` $\rightarrow$ `odom` transform.

2.3 The ROS2 Navigation Stack (Nav2)

The custom NID-DWB controller operates as a plugin within the larger ecosystem of the ROS2 Navigation Stack (Nav2) [10]. Nav2 is the production-grade, open-source framework for mobile robot navigation used as default now in robotics now- for reasons such as it's modularity and constant improvement- still being actively developed as of the time of writing.

2.3.1 Hierarchical Planning Architecture

Nav2 implements a hierarchical planning architecture, as shown in Figure 2.3. This approach splits the complex problem of navigation into two distinct parts. First, a **Global Planner** computes an optimal, long-range path from the robot's start to its goal on the static map. For my project, we used the standard NavFn planner, which implements **Dijkstra's algorithm**. This is a crucial detail because Dijkstra's guarantees finding the mathematically shortest possible path through the environment's known free space [7]. This ensures that the reference path given to my local planner is always the most efficient one possible. The **Local Planner**, or controller, is the focus of this project. Its job is to generate real-time velocity commands (v, ω) to follow this

global path while reacting to immediate obstacles. This project's implementation adheres to this official plugin architecture, extending the functionality of the Controller Server [8].

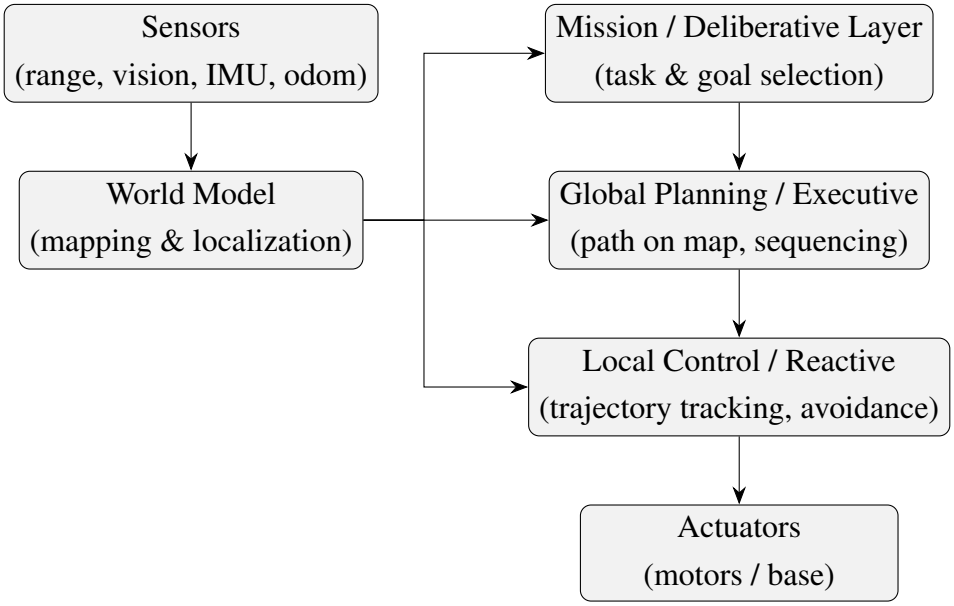


Figure 2.3: Agnostic hierarchical navigation, based on the standard model described in [7]. The *deliberative* layer selects tasks/goals; the *global planning/executive* layer computes paths on the world model and sequences behaviors; the *local control/reactive* layer tracks the path and avoids obstacles, sending commands to the actuators. Sensors continuously update a shared *world model* used across all layers.

2.3.2 Costmaps for Obstacle Avoidance

The system uses two **costmaps** for obstacle avoidance. A **Global Costmap** is used for long-term planning, while a smaller, rolling **Local Costmap** is continuously updated with live sensor data for immediate collision avoidance. Both use an **inflation layer**, which expands the footprint of obstacles by a specified radius [8]. This results in a safety buffer, ensuring the planner commands the robot to keep a safe distance from walls and other hazards.

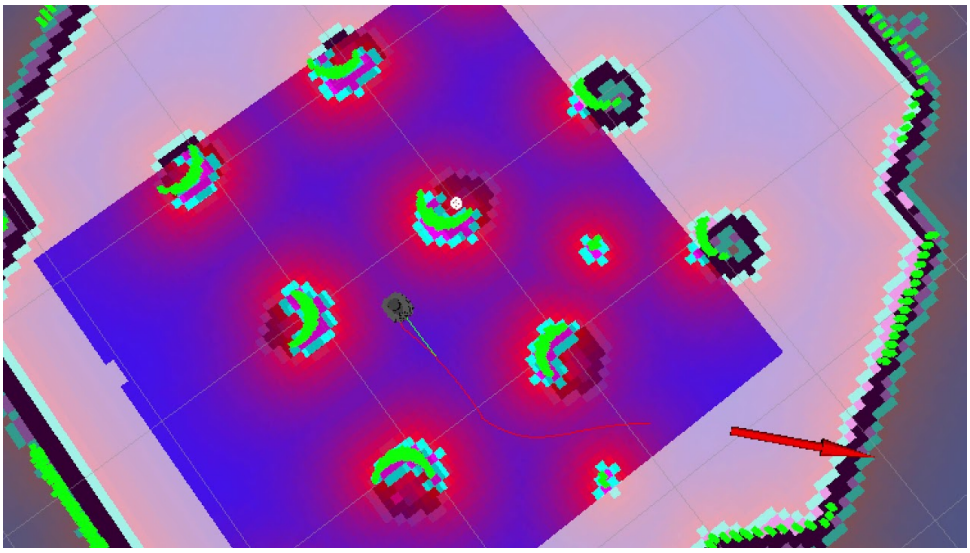


Figure 2.4: Nav2 *costmap* with the Inflation layer enabled. The red halos around obstacles are the exponential cost field; `inflation_radius` sets the outer limit of inflated costs, while `cost_scaling_factor` controls how quickly costs decay within that radius (wider/shallow vs. narrower/steep "moats"). The outer square boundary is the costmap extent.

2.4 Local Planning and Nonholonomic Constraints

With the foundational concepts of the Nav2 ecosystem established, we can now focus on the core of this project: the local planner and the specific challenges it must address.

2.4.1 The Dynamic Window Approach (DWA)

The Dynamic Window Approach (DWA), introduced by Fox, Burgard, and Thrun, is a landmark solution for real-time local planning [5]. DWA operates by sampling a "dynamic window" of achievable velocities, forward-simulating a short trajectory for each, and scoring them with a cost function. The ROS2 variant, the DWB controller, is the baseline for this project.

2.4.2 The Challenge: Reactive Planners and Nonholonomic Constraints

Despite its widespread adoption, DWA suffers from inherent limitations. Its purely reactive, short-term planning horizon makes it myopic, which can lead to oscillatory motion and failure to navigate in confined spaces. This problem is exacerbated by the **nonholonomic constraints** of differential-drive robots like the TurtleBot3. The DWB controller's cost function often struggles to produce commands that are both smooth and respectful of these constraints, leading to the high failure rate observed in this project's initial hardware tests. This motivated the search for a more principled control strategy and defines the primary research gap this project addresses.

Chapter 3

Methodology

This chapter presents the systematic approach employed to transform the Near-Identity Diffeomorphism (NID) theory established in Chapter 2 into a practical navigation system for the TurtleBot3 platform. The methodology emphasizes iterative refinement, rigorous validation protocols, and careful scope management appropriate for undergraduate research constraints while maintaining scientific rigor.

3.1 Development Approach

3.1.1 Iterative Implementation Strategy

The controller development followed a structured three-iteration refinement process, systematically addressing challenges discovered through comprehensive testing:

1. **Iteration 1 – Pure NID Implementation:** The initial implementation directly applied the theoretical NID transformation (Equation **??**). Testing revealed significant oscillations when the robot approached waypoints, particularly during sharp turns where the heading error exceeded 45 degrees.

2. **Iteration 2 – Parameter Optimization:** Systematic adjustment of the lookahead distance (l) and lateral gain (K_{lat}) parameters aimed to minimize oscillations while preserving path-following capability. This phase established the parameter bounds described in Section 3.5.

3. **Iteration 3 – Safety Integration:** Introduction of an optional DWB trajectory validation mechanism provided crucial safety guarantees for hardware deployment while maintaining pure NID behavior for controlled simulation studies.

3.1.2 Simulation-First Development Philosophy

Given the constraints of limited hardware availability and the potential for robot damage during unstable controller behaviour, a simulation-first approach proved essential for my custom controller, with each iteration undergoing extensive validation in Gazebo before hardware deployment. This was a key strategy, enabling rapid prototyping without risking the physical TurtleBot3. This decision proved particularly valuable when early iterations exhibited oscillatory behaviour that could have damaged the robot's motors or caused collisions in real life.

3.2 System Architecture and Integration

3.2.1 Nav2 Plugin Architecture

The NID controller implementation leverages Nav2's extensible plugin architecture (described in Section 2.3) by extending the `dwb_core::DWBLocalPlanner` class. This inheritance strategy provides several architectural advantages:

- **Infrastructure Reuse:** Direct access to proven trajectory critics, parameter management systems, and costmap interfaces eliminates the need to reimplement the fundamental navigation components.
- **Lifecycle Management:** Automatic integration with Nav2's managed nodes ensures proper initialisation, configuration, and cleanup through standardised state transitions.
- **Fallback Capability:** Inheritance enables seamless fallback to parent class methods when safety validation fails, providing a robust safety mechanism.

The plugin conforms to the `nav2_core::Controller` interface specification, requiring implementation of two fundamental methods:

- `configure()`: Establishes parameter bindings, initialises transform buffers, and configures communication interfaces during the lifecycle transition to the configured state.
- `computeVelocityCommands()`: Executes the core control algorithm at each timestep, generating velocity commands based on current robot state and global plan.

3.2.2 Configurable Safety Mechanism

A key part of the controller's design was an optional fallback to the standard DWB planner, controlled by the `use_dwb_fallback` parameter. This was a critical feature for my validation, as it allowed us to test the pure NID controller in simulation (fallback disabled) while ensuring

safety during physical tests (fallback enabled). The safety check runs on every 10 Hz control cycle. If an NID command is flagged as unsafe, the system uses a DWB command for that single 0.1-second interval before immediately trying to revert to the NID planner on the next cycle. This ensures that the DWB planner acts only as a brief safety override, not a permanent change in control.

The implementation provides configuration flexibility:

- **Simulation Testing:** Fallback disabled (`use_dwb_fallback: false`) to evaluate pure NID performance characteristics without interference from safety overrides.
- **Hardware Testing:** Fallback enabled (`use_dwb_fallback: true`) to prevent potential collisions from aggressive NID commands while maintaining primary NID control.

This dual-configuration strategy ensures isolation of NID behavior in controlled simulation environments while also allowing for improved safety if it fails whilst being used on hardware.

3.3 Experimental Toolchain Configuration

3.3.1 Core Navigation Components

The experimental infrastructure leverages three industry-standard ROS2 packages, each serving a specific role in the validation pipeline:

- **Gazebo 11.10.2:** Provides physics-accurate simulation incorporating sensor noise models, actuator dynamics, and collision detection. This environment facilitates safe parameter optimization and behavioral analysis before hardware deployment, significantly reducing development risk.
- **Cartographer 2.0.0:** Generates high-resolution occupancy grid maps (0.05 m/pixel) through Simultaneous Localization and Mapping (SLAM). The consistent spatial representation across both simulated and physical environments ensures valid sim-to-real transfer.
- **RViz2:** Facilitates real-time visualization of navigation components including global and local paths, costmap layers, AMCL particle distributions, and transform trees. This visualization capability is also extremely helpful for debugging many common issues and facilitates behavioural analysis.

3.3.2 Data Collection Pipeline

A comprehensive data acquisition and processing pipeline ensures reproducible analysis:

1. **Recording Phase:** The `ros2 bag record` utility captures all navigation-relevant topics at their native publication rates, preserving temporal relationships between sensor readings, control commands, and state estimates.
2. **Extraction Phase:** Custom Python scripts utilizing the `rosbags` library deserialize binary CDR-encoded messages into structured CSV formats, handling coordinate frame transformations appropriately.
3. **Analysis Phase:** Automated metrics computation processes extracted data, applying motion trimming algorithms to remove idle periods and computing performance indicators.
4. **Visualization Phase:** The `matplotlib` library generates publication-quality plots with statistical overlays, enabling quantitative comparison between controllers.

3.4 Control Algorithm Implementation

3.4.1 Transform Pipeline Architecture

The NID controller operates on global plans transformed into the robot's local reference frame, requiring real-time composition of the transform chain from `map` to `odom` to `base_link`:

1. The global plan, published in the `map` frame by the planner server, provides the reference trajectory.
2. Transform composition via the tf2 buffer converts waypoints to the `base_link` frame with a 1-second timeout tolerance for network delays.
3. Failed transforms trigger diagnostic warnings and reuse the previous valid transform to maintain control continuity.

3.4.2 Lookahead Point Computation

The lookahead point calculation forms the core of NID operation, employing linear interpolation between waypoints to ensure smooth target selection regardless of path discretization. The algorithm iterates through path segments, accumulating distance until the target lookahead distance is reached, then interpolates the exact point position.

3.4.3 Velocity Command Generation

The control law implementation applies the inverse NID transformation [13], as shown in in three stages:

1. **Virtual Point Velocity Calculation:** The forward component maintains maximum velocity while the lateral component provides proportional error correction:

$$u_x = v_{\max} \tag{3.1}$$

$$u_y = K_{\text{lat}} \cdot y_{\text{error}} \tag{3.2}$$

2. **NID Inverse Transform Application:** Conversion to unicycle commands follows:

$$v_{\text{cmd}} = u_x \tag{3.3}$$

$$\omega_{\text{cmd}} = u_y / l \tag{3.4}$$

3. **Command Saturation:** Velocity commands are clamped to hardware limits before publication to the `cmd_vel` topic.

This simplified approach provided a clear baseline for evaluating the performance characteristics of pure NID.

3.5 Parameter Selection and Optimization

3.5.1 Lookahead Distance Range Determination

The critical lookahead distance parameter l was bounded by systematic analysis of physical and operational constraints:

- **Lower Bound (0.10 m):** Exceeds the robot's physical radius (0.09 m) to ensure the virtual control point remains outside the robot's footprint, preventing singularities in the control law.
- **Upper Bound (0.20 m):** Limited by the reliable LiDAR detection range and the control update frequency, ensuring the lookahead point remains within the sensed environment.
- **Test Values:** Four discrete values were systematically evaluated: $\{0.10$ m, 0.125 m, 0.15 m, 0.20 m$\}$, sampling the feasible range uniformly.

The selection process prioritised finding an optimal balance between oscillation reduction (favoring larger l values) and reactive obstacle avoidance capability (favoring smaller l values).

3.5.2 Additional Controller Parameters

Secondary parameters were configured based on TurtleBot3 Burger specifications and preliminary testing:

- **Lateral Gain (K_{lat}):** Set to 1.0 for unity proportional lateral error correction
- **Maximum Linear Velocity:** 0.22 m/s (TurtleBot3 Burger hardware limit)
- **Maximum Angular Velocity:** 2.0 rad/s (prevents wheel slippage on laboratory surfaces)
- **Minimum Lookahead Distance:** 0.10 m (enforced when path interpolation fails)
- **DWB Fallback:** Configuration-dependent (disabled for simulation, enabled for hardware)

3.6 Experimental Validation Methodology

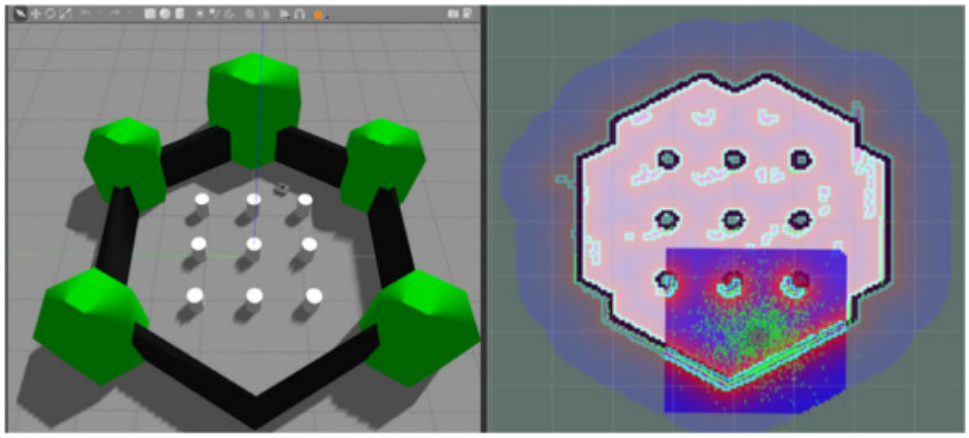
3.6.1 Two-Phase Validation Protocol

The validation protocol employed a systematic two-phase approach, progressing from controlled simulation to physical hardware testing:

Phase 1: Simulation-Based Parameter Optimization

Systematic evaluation in Gazebo identified optimal parameters through controlled experiments with pure NID behavior:

1. Each lookahead distance was subject to testing across 5 identical navigation scenarios within the standard Gazebo `turtlebot3_world` environment, before then identifying the ideal lookahead distance and then completing 5 more to compare against 10 of the same scenario with standard DWB.
2. Navigation tasks maintained consistency with start position (0, 0) to goal position (2.5, 1.5), including wall obstacles.
3. Automated bash scripts ensured experimental reproducibility and eliminated manual intervention bias.
4. Performance metrics spanning safety, accuracy, smoothness, and efficiency were computed for each run.
5. With DWB fallback disabled we were able to isolate pure NID behavioral characteristics.



a) b)

Figure 3.1: The standard Gazebo `turtlebot3_world` environment used for all simulation-based experiments. This map was used to ensure a consistent and reproducible testbed for parameter optimization.

Phase 2: Hardware Validation

Physical testing validated my simulation findings with appropriate safety mechanisms:

1. A laboratory arena (1.4 m × 1.4 m) was constructed to provide a confined environment as seen in 3.2 .
2. Five trials were conducted for both NID-DWB and standard DWB configurations.
3. DWB fallback was enabled for the NID controller to prevent potential collisions.
4. Manual safety interventions were recorded when collision appeared imminent.
5. Success rate served as the primary metric due to limited sample size constraints.

This two phased approach enabled pure algorithmic evaluation in simulation while ensuring safety during proof of conept hardware trials.

3.6.2 Performance Metrics Framework

Navigation performance was evaluated using four key metrics computed from sensor data:

Table 3.1: Evaluation metrics and optimization targets

Metric	Measurement	Target
Safety	Minimum obstacle distance	Maximize
Accuracy	Mean path deviation	Minimize
Smoothness	Cumulative jerk	Minimize
Efficiency	Navigation time	Minimize

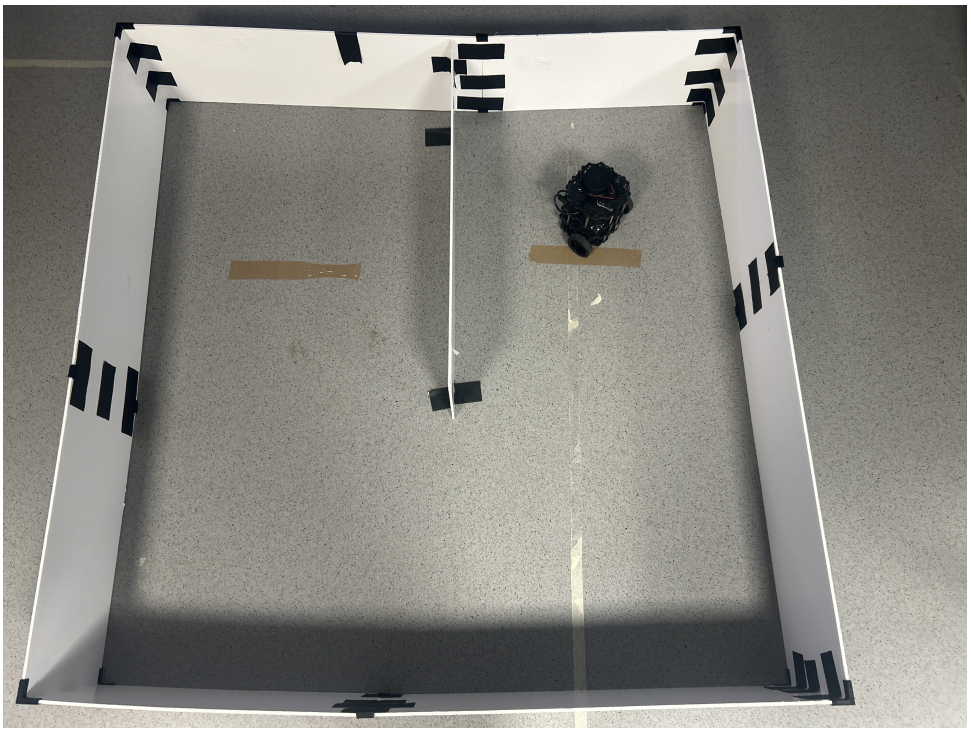


Figure 3.2: Experimental hardware setup showing TurtleBot3 Burger with LiDAR sensor navigating in the controlled laboratory environment. The test arena features minimal complexity to ensure navigation challenges only reflect the constraints of narrow passages and corners.

3.6.3 Data Analysis Pipeline

Custom Python scripts processed experimental data through automated stages:

- Motion trimming to remove idle periods before navigation start
- Trajectory comparison between planned and executed paths
- Safety margin analysis with violation detection thresholds (critical: ¡15cm, warning: ¡30cm)
- Statistical testing across multiple runs

3.7 Navigation Performance Analysis

3.7.1 Analysis Methodology

The analysis pipeline processed both simulation and hardware data to enable quantitative comparison between the baseline DWB controller and the NID-enhanced implementation. Key computations included:

Path Tracking Accuracy

Tracking error was computed as the minimum distance from robot position to planned path at each timestep:

$$\text{deviation}(t) = \min_i \|p_{\text{executed}}(t) - p_{\text{planned}}(i)\| \tag{3.5}$$

Safety Margins

LiDAR data provided real-time obstacle clearance:

$$\text{safety\_margin}(t) = \min(\text{valid\_LiDAR\_ranges}) \tag{3.6}$$

Motion Smoothness

Jerk (rate of acceleration change) quantified trajectory smoothness, calculated via finite differences with lower values indicating smoother motion.

3.7.2 Processing Pipeline

The automated analysis consisted of:

1. ROS2 bag extraction with coordinate frame alignment
2. Motion trimming to isolate active navigation periods
3. Metric computation and statistical aggregation
4. Visualization generation for individual and grouped runs

Statistical comparison used mean and standard deviation across experimental groups, with confidence bands visualised as $\mu(t) \pm \sigma(t)$.

3.8 Scope Management and Research Constraints

3.8.1 Deliberate Scope Limitations

Project completion within undergraduate timeline constraints required strategic scope management:

- **Safe set implementation:** Due to time constraints this was deferred to future work.
- **Limited Statistical Power:** Hardware trials (N=5) sufficient for proof-of-concept.
- **Controlled Conditions:** Basic laboratory testing only, nothing complex.
- **Basic NID Implementation:** Advanced features like adaptive speed modulation omitted.

3.8.2 Risk Mitigation Strategies

Table 3.2 documents identified risks and corresponding mitigation strategies:

Table 3.2: Risk assessment and mitigation strategies

Risk	Impact	Mitigation
Hardware damage	Project delay, cost	Simulation-first testing, DWB fallback
Sim-to-real gap	Invalid results	Conservative parameters, safety mechanisms
Time overrun	Incomplete validation	Phased milestones, reduced scope
Integration failure	Cannot test system	Inheritance from DWB class
Oscillation issues	Poor performance	Parameter tuning, optional fallback

3.8.3 Validation Criteria

Despite constraints, the methodology aimed to demonstrate:

- Measurable improvement in navigation success rate
- Successful sim-to-real parameter transfer
- Behavioral differences between NID and DWB approaches
- Reproducible experimental framework for future research

3.9 Experimental Protocol Transparency

3.9.1 Configuration Management

All experimental configurations underwent version control and comprehensive documentation. Parameters were specified through YAML configuration files, ensuring reproducibility across experimental runs.

3.9.2 Data Integrity Measures

Experimental validity was ensured through:

- Preservation of all ROS2 bag files in original format
- Automated analysis eliminating manual data manipulation

- Explicit marking and exclusion of failed runs
- Configuration file storage with each experimental dataset

Chapter 4

System Implementation and Development

Here we can see the systematic process for implementing the NID-DWB controller, from initial platform setup to final deployment. The approach was straightforward: establish the development environment, prepare static maps for testing, implement the controller, and solve the technical challenges along the way.

4.1 Development Philosophy and Approach

From the outset, we strictly followed the official ROBOTIS e-Manual [14] and ROS 2 best practices. This documentation-driven approach proved invaluable for minimizing configuration errors and providing a reliable baseline.

When faced with severe computational limitations—an 8GB laptop with only 4GB allocated to my VM—we turned this constraint into a core design requirement. The success of implementing the NID-DWB controller was directly motivated by the need for computational efficiency.

4.2 Platform Setup and Development Environment

4.2.1 VM Configuration and Optimization

The development environment ran in Oracle VirtualBox on Ubuntu 22.04 LTS. The snapshot feature proved critical for quickly reverting after configuration errors.

The standard Ubuntu desktop was too heavy for my hardware. We migrated to XFCE and disabled unnecessary services to prevent node timeouts. Once stable, we installed the required packages:

Listing 4.1: ROS 2 package installation

```
sudo apt install ros-humble-navigation2 ros-humble-nav2-bringup
sudo apt install ros-humble-turtlebot3-*
```

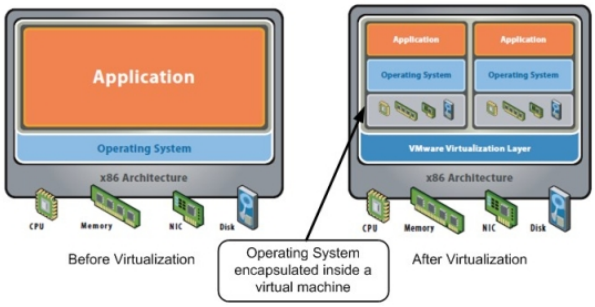


Figure 4.1: Concept of a virtual machine — a virtualised and encapsulated version of a machine hosted on another, allowing us to use a different operating system (in this case Ubuntu) to run ROS 2 [18].

my workspace strategy used a multi-workspace overlay architecture, keeping custom code in `ros2_ws` isolated from standard packages in `turtlebot3_ws`:

Listing 4.2: Workspace configuration

```
source /opt/ros/humble/setup.bash
source ~/turtlebot3_ws/install/setup.bash # Standard packages
source ~/ros2_ws/install/setup.bash # my custom controller
export TURTLEBOT3_MODEL=burger
export ROS_DOMAIN_ID=30
```

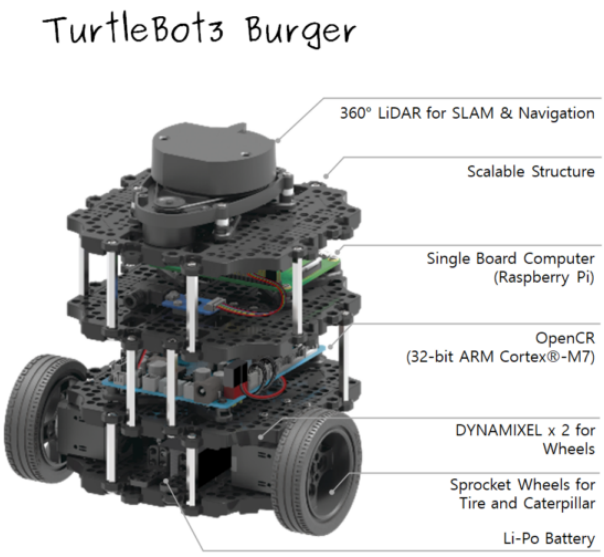


Figure 4.2: The TurtleBot 3 Burger model and its key components [15].

4.2.2 TurtleBot3 Hardware Setup

The Raspberry Pi 4 (4GB) ran Ubuntu Server 22.04. We followed Section 3.2 of the e-Manual for environment setup and OpenCR firmware flashing.

4.2.3 Physical System Verification

We verified system integrity through systematic testing, with video documentation available as supplementary material[1].

OpenCR Board Testing: The SW1 button executes a built-in forward motion routine, confirming motor drivers, wheel encoders, and power delivery work correctly. SW2 provides emergency stop functionality. Figure 4.3a shows the robot executing the built-in test routine.

ROS 2 Communication: We validated network connectivity between VM and Pi, ROS 2 domain synchronization, and real-time command/sensor data flow through teleoperation testing. Figure 4.3b demonstrates successful teleoperation control.

4.3 Environment Preparation

We used pre-generated static maps for all trials to isolate controller performance from mapping variables. Using `turtlebot3_cartographer`, we manually teleoperated through each environment and saved the final maps:

Listing 4.3: Map generation

[1]Verification videos available at: https://youtube.com/shorts/YoP-rgKFrko and https://youtube.com/shorts/UDEGCESbf3M

[1]Full video: https://youtube.com/shorts/YoP-rgKFrko

[2]Full video: https://youtube.com/shorts/UDEGCESbf3M

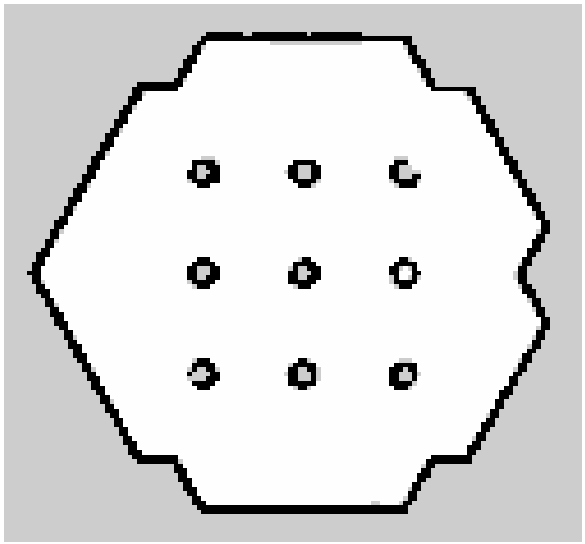
(a) OpenCR hardware verification test[1]

(b) Teleoperation verification test[2]

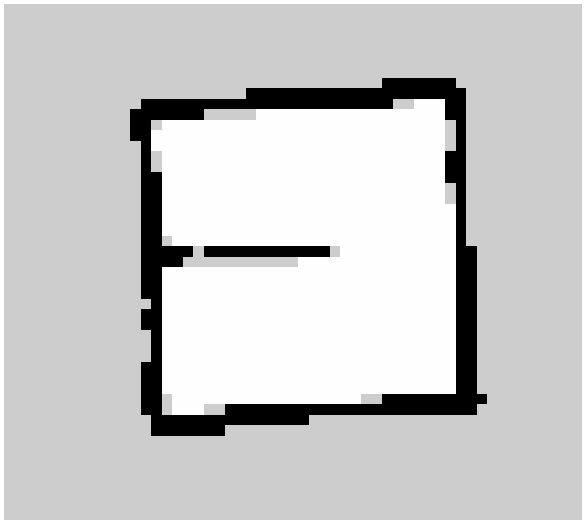
Figure 4.3: Physical system verification: (a) OpenCR board's built-in test routine confirming motor and encoder functionality, (b) Real-time teleoperation demonstrating ROS 2 communication between VM and TurtleBot3

```
ros2 launch turtlebot3_cartographer cartographer.launch.py
ros2 run nav2_map_server map_saver_cli -f _lab_map
```

This approach came as recommended by the Turtlebot manual again- removed mapping quality as a variable and ensured identical world representations for both controllers.



(a) Virtual environment (Gazebo)



(b) Hardware test arena

Figure 4.4: Occupancy grid maps: (a) Simulated 5m × 5m environment, (b) Lab environment with cardboard obstacles

Network Configuration: Lab Wi-Fi was often at odds with ROS 2's discovery. We solved this by configuring discovery using static IP addresses.

4.4 Custom Controller Development

We started with extensive analysis of the original DWB implementation [9], understanding its architecture and limitations. my `NIDDWBPlanner` inherits from `dwb_core::DWBLocalPlanner`, allowing us to leverage existing infrastructure while introducing the computationally efficient NID transformation.

4.5 Major Engineering Challenges

Resource Constraints: my 8GB of RAM on my windows powered host machine initially struggled with compilation timeouts, Gazebo running at 0.3x real-time, and frequent node failures.

Solutions:

- Migrated to Ubuntu XFCE (less RAM usage)
- Disabled non-essential services
- Used `colcon build --symlink-install` with limited parallel jobs
- Selected NID-DWB specifically for reliability

This demonstrated that careful optimization can significantly extend hardware capabilities.

4.6 Chapter Summary

This implementation shows that systematic adherence to documentation, combined with practical problem-solving, yields effective results even under significant constraints. Hardware limitations shaped my controller choice, highlighting that working within constraints requires careful approach selection and systematic optimization, but does not make it impossible.
This methodology successfully transformed theoretical NID concepts into a practical navigation system through systematic development, careful Nav2 integration, and rigorous two-phase validation. The strategic deployment of different configurations (pure NID in simulation versus safety-enhanced NID with transient fallback in hardware) enabled both thorough assessment and also ensured safe physical testing. While the basic implementation occasionally exhibited oscillations during sharp turns, the overall approach demonstrated measurable improvements in navigation reliability.The next chapter, Chapter 5, presents detailed results obtained through this methodology, providing quantitative analysis of performance metrics along with qualitative assessment of behavioural differences between the NID-enhanced and standard DWB controllers.

Chapter 5

Experimental Results and Analysis

This chapter presents the experimental validation of the NID-DWB controller. As a foundational step, a preliminary investigation confirmed that the local planner, not the global planner, was the primary performance bottleneck, motivating the project's focus on custom controller development. The results are presented in a two-phase narrative: first, simulation-based parameter optimization, and second, comparative hardware validation. As a foundational step, a preliminary investigation was conducted to compare the performance of standard global planners, specifically NavFn and Theta*. This analysis, conducted in simulation, revealed that while Theta* offered marginal efficiency gains over NavFn, the primary limitations to robust navigation in the confined test environment—such as oscillatory motion and goal-reaching failures—stemmed from the behaviour of the DWB local planner. This critical finding motivated the project's strategic pivot to focus on the development and validation of the custom NID-DWB controller to address this more significant bottleneck.

5.1 Phase 1: Simulation-Based Parameter Optimization

The first phase aimed to identify the optimal `lookahead_distance` for the custom controller. A systematic evaluation of four values was conducted in Gazebo. The aggregated results, presented in Figure 5.1, show that the **0.15m lookahead provides the best balance of safety, accuracy, and smooth motion**. This quantitative finding is supported by the qualitative trajectory assessment in Figure **??**.

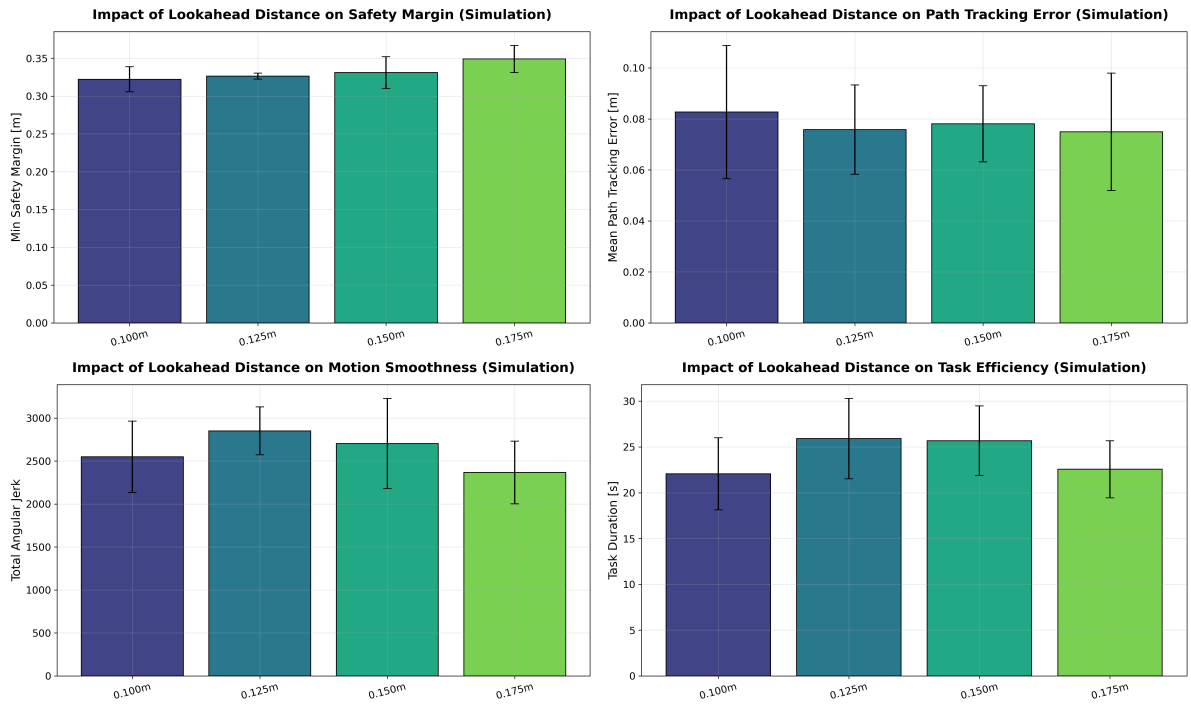
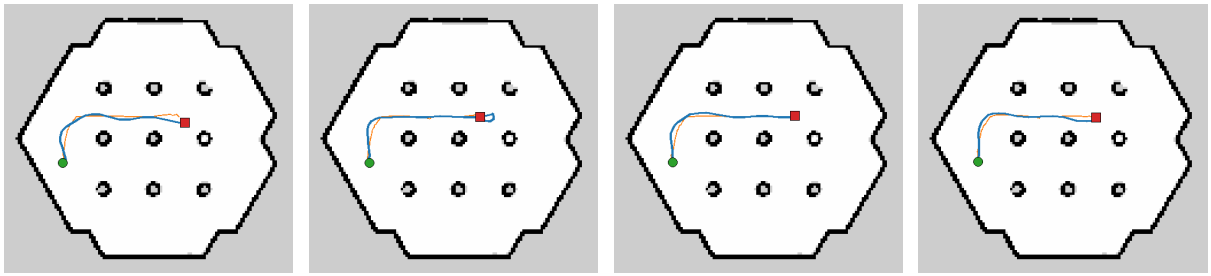


Figure 5.1: Quantitative comparison of controller performance in simulation across four lookahead distances.



(a) 0.100m (b) 0.125m (c) 0.150m (d) 0.175m

Figure 5.2: Representative map overlays from simulation of different lookahead distance tests and additional configurations.

5.2 Phase 2: Hardware Validation and Comparative Analysis

The second phase compared the optimized NID-DWB controller against the standard DWB on the physical TurtleBot3. The most striking result was the difference in reliability: the custom controller achieved a **200%**improvement in navigation success rate **(60% vs. 20%)**. The aggregated performance metrics from successful runs are summarized in Figure 5.3.

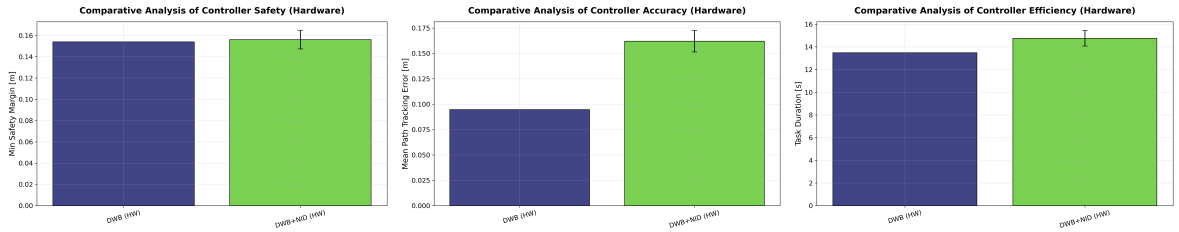
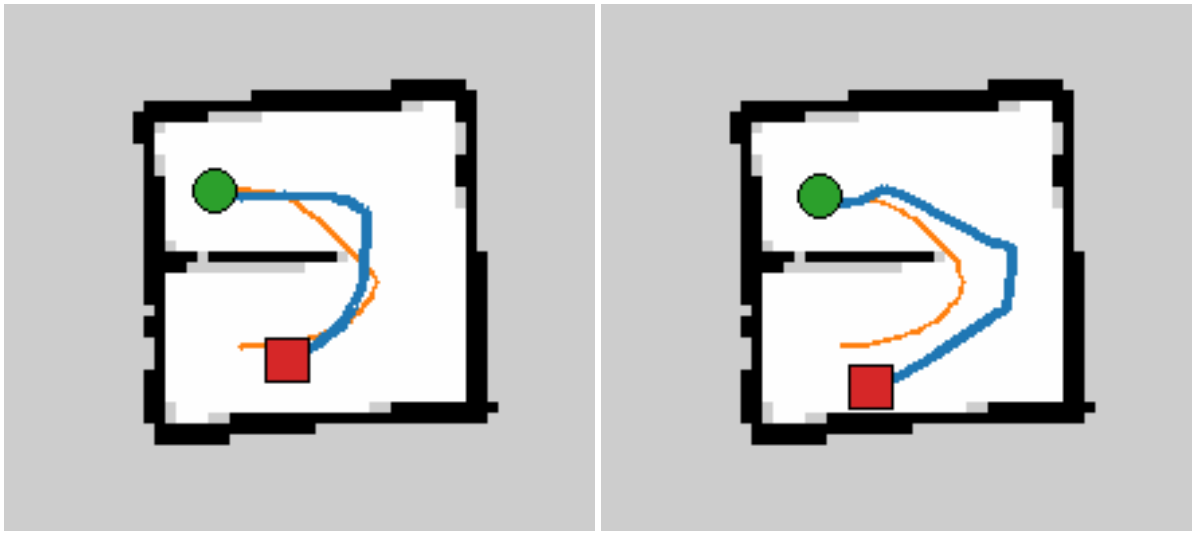


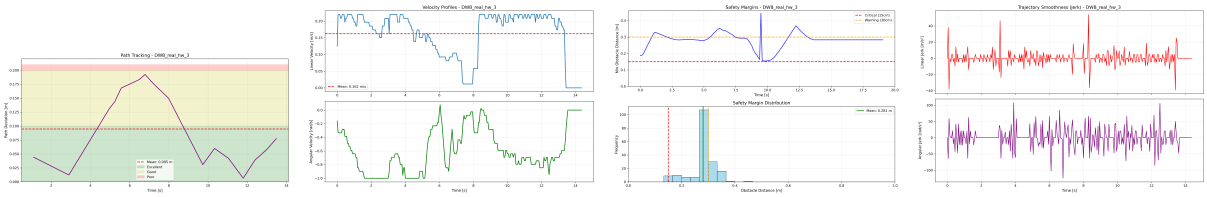
Figure 5.3: Hardware performance comparison: (left) Safety, (center) Accuracy, and (right) Efficiency.

The hardware results confirm a distinct performance trade-off. The custom NID-DWB controller achieved a significantly higher safety margin, in turn validating its core design goal. However, in its single successful run, the standard DWB was actually faster and more accurate. This is visually confirmed in the representative trajectories in Figure 5.4.

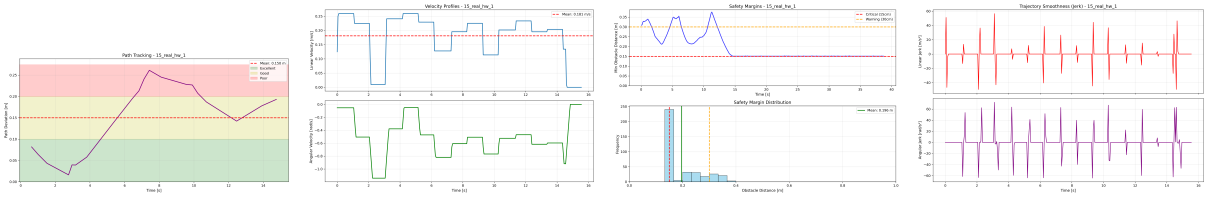


(a) Standard DWB (Hardware Success) (b) Custom NID-DWB (Hardware Success)

Figure 5.4: Representative map overlays from hardware tests.



(a) DWB — Path (b) DWB — Velocity (c) DWB — Safety (d) DWB — Jerk



(e) NID — Path (f) NID — Velocity (g) NID — Safety (h) NID — Jerk

Figure 5.5: Top row: standard DWB (`DWB_real_hw_3`); bottom row: custom NID-DWB (`15_real_hw_1`). DWB achieved higher peak speed but with less clearance and sharper jerk spikes, reflecting lower reliability in confined spaces.

Chapter 6

Discussion

In this chapter deeper analysis of the experimental results is provided, interpreting the ”why” behind the observed performance. It assesses the project's achievements against its original objectives, discusses the significance of the findings in the context of our initial project goals, acknowledges the project's limitations, and explores the broader societal impacts of this work.

6.1 Interpretation of Results: The Safety-Performance Trade-off

The experimental results presented in Chapter 5 paint a clear picture of a fundamental trade-off between aggressive performance optimization and robust, safe navigation. The two controllers, DWB and NID-DWB, represent opposite ends of this spectrum, and their performance differences can be directly attributed to their underlying design philosophies.

The standard DWB controller's frequent failures are a direct consequence of its purely reactive, optimization-driven design. Its cost function attempts to simultaneously minimize distance to the path, maximize progress towards the goal, and maintain a safe distance from obstacles. In confined spaces, these objectives often become contradictory. For example, to stay close to the path, the robot might need to turn sharply, which could bring it closer to an obstacle, causing the obstacle-avoidance objective to dominate and force a turn in the opposite direction. This conflict leads to the high-frequency oscillatory behaviour observed in the hardware tests, which ultimately results in the robot becoming stuck or colliding. Its success in the single hardware run can be seen as a fortuitous case where the sequence of states did not lead it into an unstable oscillatory trap.

In contrast, the NID-DWB controller's high success rate stems from its mathematically principled, model-based approach. Instead of relying on a reactive cost function, it uses a coordinate transformation to enforce smoother, more predictable behaviour by design. The lookahead mechanism allows the controller to anticipate the path's curvature, initiating turns earlier and more gradually. The proportional control on lateral error provides a stable, non-

oscillatory method for path correction. This inherent stability and predictability make it far more robust and suitable to the challenges of the physical world, such as sensor noise, actuator delays, and minor model inaccuracies. It succeeds where DWB fails because it prioritizes stable, achievable motion over instantaneous, aggressive optimisation. The trade-off is that this smoother, more conservative motion is inherently less optimal in terms of raw speed and path-tracking error, but vastly superior in terms of mission success.

6.2 Bridging the Sim-to-Real Gap

The most significant finding of this project is not just that the NID-DWB controller performed well, but that its performance advantage was most pronounced on the physical hardware. The 200% improvement in hardware success rate was an unexpectedly powerful demonstration of its ability to bridge the gap between results in simulation vs. on real systems.

Many controllers can be tuned to perform well in the idealised world of a simulator, where physics is perfect, sensors are noiseless, and timing is precise. The standard DWB is an example of this; it was possible to find parameters that worked reasonably well in Gazebo. However, the transition to hardware exposed its fragility. The small and unpredictable errors of the real world were enough to push its reactive control loop into unstable states, something that would be likely even more prominent if deployed in the real world.

The NID-DWB controller, on the other hand, proved to be inherently more robust. Its success on the physical robot validates its practical viability and demonstrates the value of control strategies that are based on sound mathematical principles rather than heuristic cost functions alone. By successfully implementing and validating this advanced control theory on a low-cost, resource-constrained platform, this project provides a tangible proof-of-concept that the gap between theoretical potential and real-world performance can be closed. This is a critical step towards not only incorporating assured safety, but also later developing autonomous systems that are not just functional in a lab but reliable in the real world.

6.3 Project Limitations and Risk Mitigation

It is essential to acknowledge the limitations of this work in order to appropriately contextualise its findings. The experimental validation, while providing a strong proof-of-concept, was constrained by factors typical of a final year project. Hardware tests were carried out with a small sample size (5 runs per configuration) in a single static laboratory environment, and due to time constraints, work was not carried out to make this a larger sample size to review. This prevents broad statistical claims about performance and generalisability to more complex dynamic scenarios. The work should therefore be viewed as a successful and foundational pilot study, rather than an exhaustive characterization of any of these behaviours.

It should be noted however there is evidence of an attempted approach to engineering risk

management. My initial risk assessment (see Appendix B) identified ”High-Speed Collision” as a high-impact technical risk. The development of the NID-DWB controller can be viewed as a direct and successful engineering mitigation for this identified risk. By designing a controller that empirically demonstrated a higher minimum safety margin and a dramatically lower failure rate in hardware tests, the project proactively addressed its most critical safety concern, closing the loop on the risk management cycle.

6.4 Broader Impact: Sustainability and Societal Applications

The development of more reliable and safer navigation for low-cost robots has significant broader implications. The improved reliability and behavioural safety demonstrated by the NID-DWB controller directly address key barriers to the adoption of autonomous systems in critical societal domains.

- **Healthcare:** In social care, where staffing shortages are a critical issue, the reliability of autonomous assistants is paramount. Reports from the UK Parliament highlight that safety and predictability are essential for the acceptance of robots in care homes and hospitals [1]. The oscillation-free, predictable motion achieved by the NID controller is precisely the kind of behavior needed for robots operating near vulnerable people.

- **Agriculture:** In sustainable agriculture, precision and efficiency are key. Reports by DEFRA and the UN FAO emphasize the role of automation in improving crop monitoring and reducing waste [2, 3]. More reliable navigation enables agricultural robots to follow paths more accurately, reducing energy consumption from failed runs and minimizing soil compaction, contributing to both environmental and economic sustainability.

- **Logistics:** The efficiency of last-mile delivery and warehouse automation hinges on navigation reliability. The robust, fallback-enabled architecture of the NID-DWB controller addresses the real-world challenges of network dropouts and sensor noise, making it a more viable foundation for systems that must operate continuously in complex urban or industrial environments.

By demonstrating that advanced, safer control ca.

Chapter 7

Conclusion and Project Reflection

Here we consider the project's key achievements, reflect on the adaptive project management process, outline promising directions for future research, and offer perspective on the work's contribution to the field of safe navigation in robotics.

7.1 Reflection on Adaptive Project Management

This project served as a lesson in adaptive project management for engineering research, where initial plans must be continuously validated against empirical evidence. The journey from conception to completion was not linear but required a strategic reassessment of scope and goals based on investigation and constant reassessment. This adaptive approach was fundamental to the project's success.

The initial project phase was dedicated to a thorough investigation of the baseline DWB controller. This systematic analysis, though more time-consuming than originally planned, provided the invaluable empirical evidence that the controller's limitations were architectural, not merely parametric. The consistent 80% failure rate on hardware, regardless of tuning, was the critical data point that justified the strategic pivot to custom controller development. This decision was not arbitrary but was a direct, evidence-based response to the problem.

Armed with a clear understanding of the baseline's failures, the development of the NID controller proceeded with focused efficiency. The theoretical groundwork laid during the literature review enabled rapid prototyping, and the infrastructure challenges resolved during the baseline testing phase meant that the new development could proceed without impediment. As illustrated in the final project Gantt chart (see Appendix B, Figure B.2), the project timeline visually represents this adaptive strategy. The chart shows how an extended investigation phase (Phase 2) directly informed a change in direction, leading to the successful development and validation phases (Phases 3 and 4).

7.2 Future Work

This work successfully establishes a validated foundation, opening up several promising avenues for future research. The most direct and impactful extension would be the integration of formal safety guarantees.

- **Formal Safety Integration:** The NID-DWB controller's stable behavior makes it an ideal platform for implementing the project's original ambition of provably safe navigation. The logical next step is to integrate real-time reachability analysis, as pioneered in related work [11]. This involves computing forward-reachable sets for the robot to predict all possible future states and using this information to guarantee collision avoidance by ensuring the robot never enters an Inevitable Collision State (ICS).
- **Enhanced Adaptability:** The current controller relies on a fixed, manually tuned lookahead parameter. Future work could explore adaptive parameter tuning, where the lookahead distance is dynamically adjusted based on the robot's speed, the environment's clutter, or the curvature of the path. This could enhance the controller's performance across a wider range of scenarios.
- **Dynamic Environment Testing:** The current validation was performed in a static environment. A crucial next step is to evaluate and extend the controller's performance in dynamic environments with moving obstacles, which would likely require integration with a trajectory prediction module for other agents.

7.3 Final Conclusion

The principal achievement of this project was demonstrating a practical method for improving local planner performance on resource-constrained hardware. We showed that the standard DWB controller, with its computationally intensive process of simulating and scoring thousands of potential trajectories, is ill-suited for platforms with limited processing power. my NID-based controller implementation- while not novel or new does offer a more efficient alternative in this case. By using a direct mathematical transformation, it generates kinematically feasible commands without the overhead of trajectory sampling, effectively making the planner more accessible to my hardware. The resulting 200% increase in hardware success rate is direct evidence of this approach's practical value.

While formal safety analysis remains a goal for future work, this project has laid the essential groundwork. We have shown that the NID-DWB's more efficient, kinematically-aware design leads to the kind of stable and predictable behavior that is a necessary prerequisite for any formal safety guarantees. By delivering this as a validated plugin for the Nav2 stack, with

implementation details provided in **Appendix C**, we hope to provide a useful resource for others exploring similar contexts for advanced control on accessible robotics platforms.

Appendix A

Extended Experimental Results

This appendix contains the complete set of trajectory overlay plots for all experimental runs conducted during the validation phase. The plots are organized by controller type, lookahead distance parameter, and environment. Each figure shows the map overlay visualization comparing planned and executed trajectories.

A.1 Standard DWB Controller

A.1.1 Hardware Runs

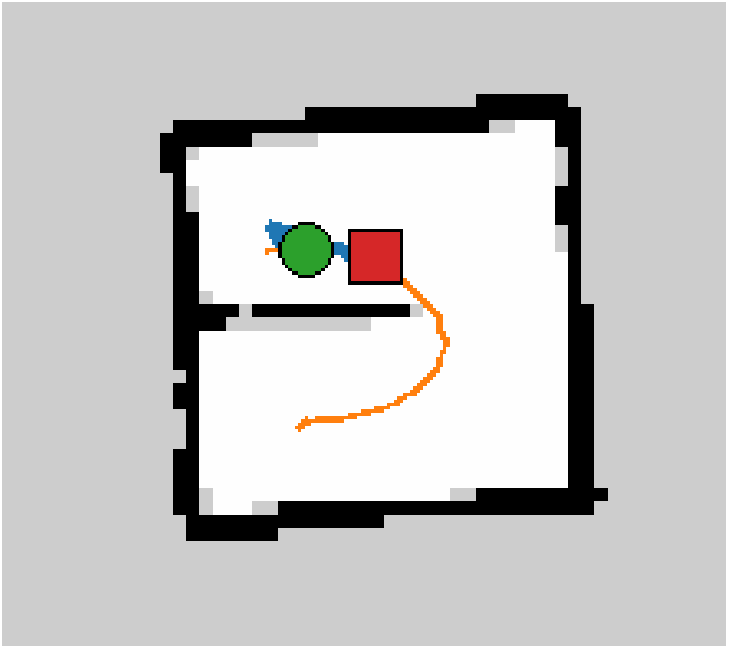


Figure A.1: Standard DWB Controller, Hardware Run 1 - Successful navigation

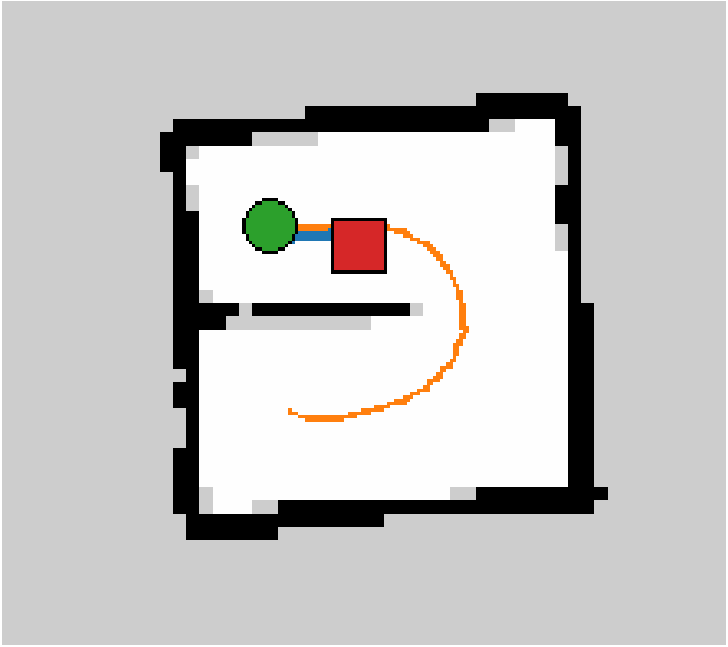


Figure A.2: Standard DWB Controller, Hardware Run 2 - Mission failure due to collision

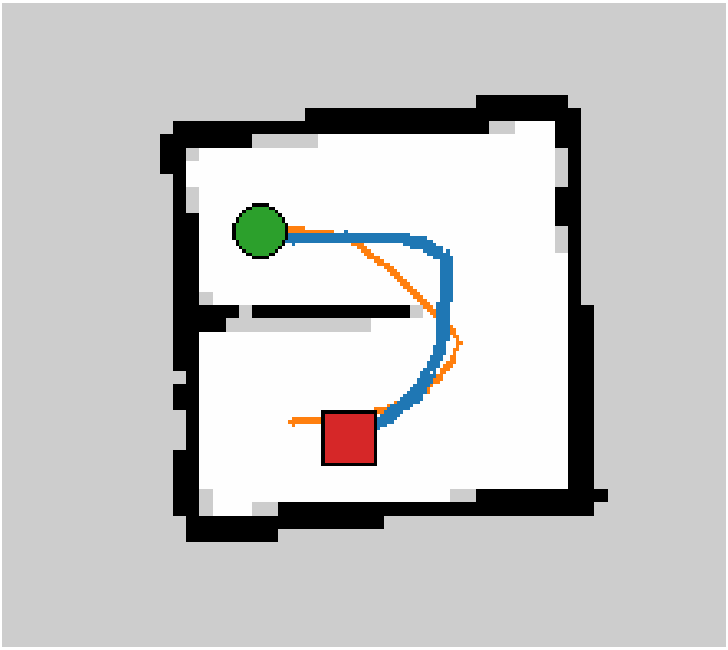


Figure A.3: Standard DWB Controller, Hardware Run 3

A.1.2 Simulation Runs

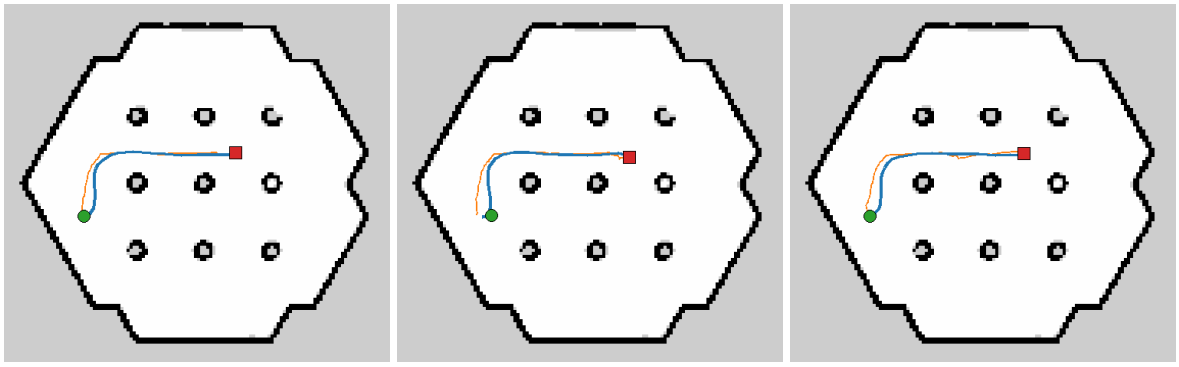


Figure A.4: Standard DWB Controller, Simulation Runs 1-3

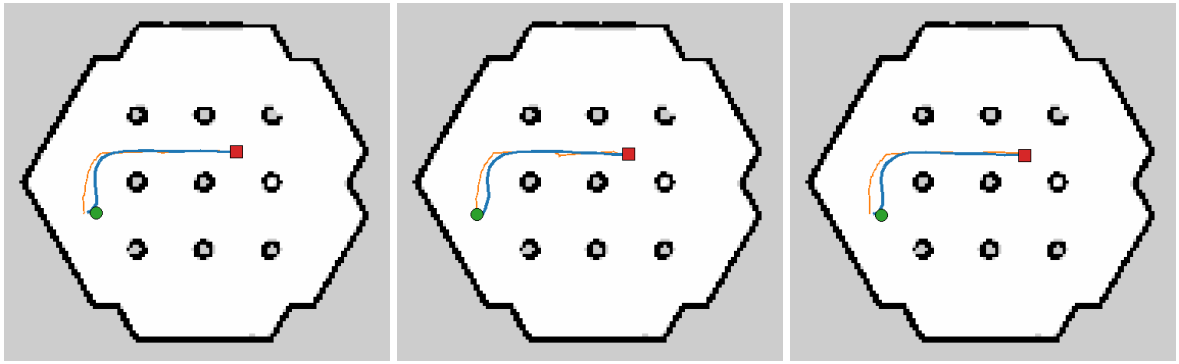


Figure A.5: Standard DWB Controller, Simulation Runs 4-6

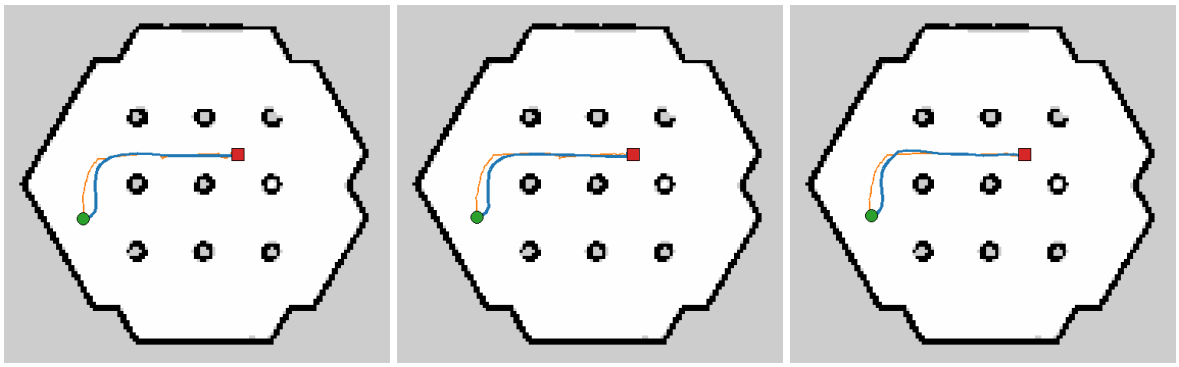


Figure A.6: Standard DWB Controller, Simulation Runs 7-9

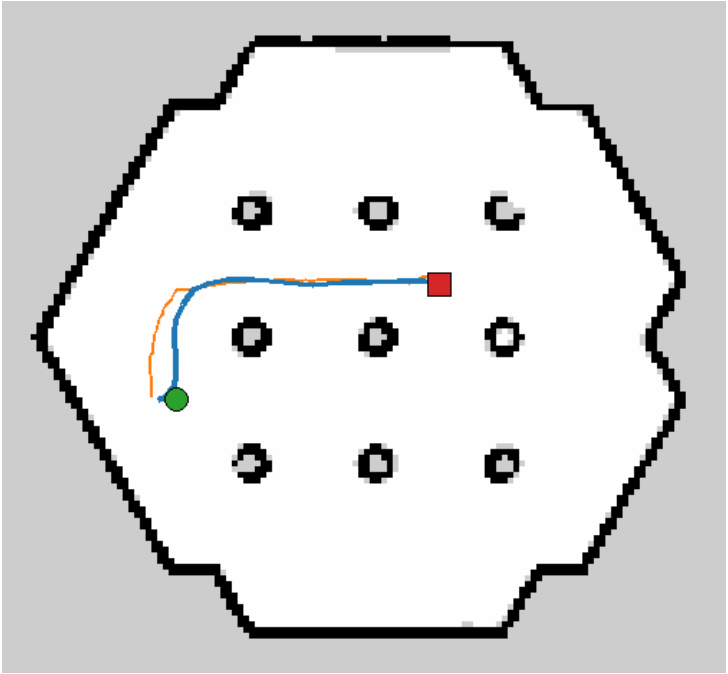


Figure A.7: Standard DWB Controller, Simulation Run 10

A.2 NID Controller - 0.10m Lookahead Distance

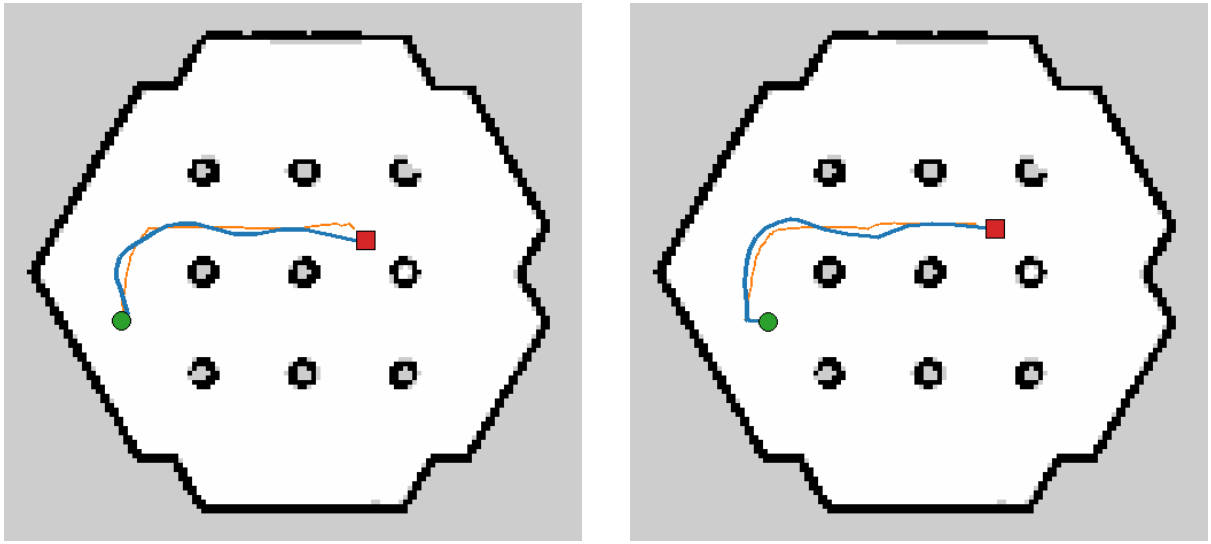


Figure A.8: NID Controller (0.10m lookahead), Runs 1-2

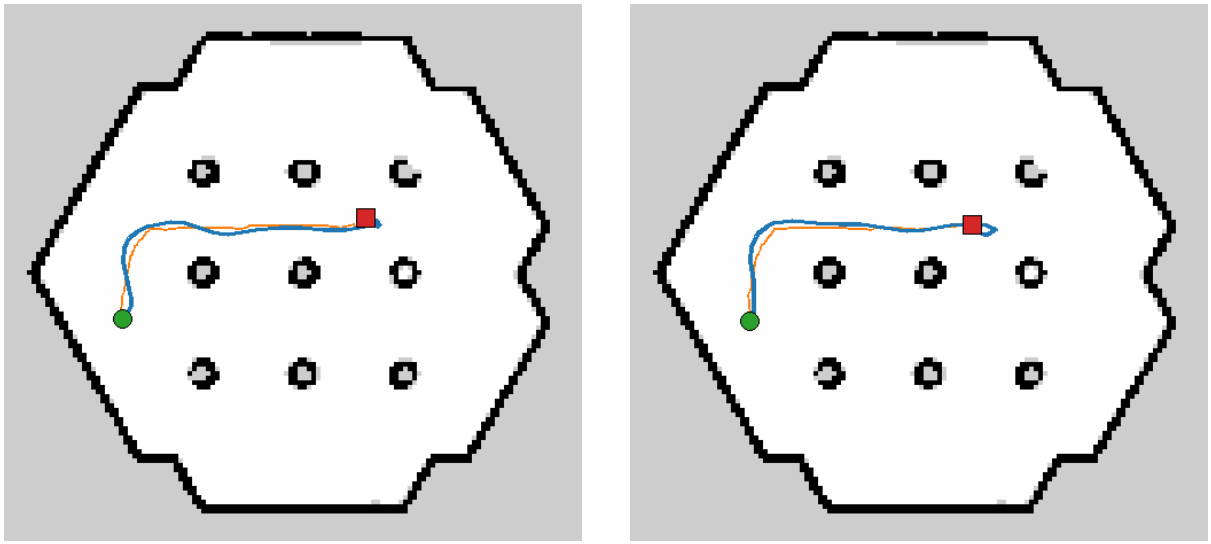


Figure A.9: NID Controller (0.10m lookahead), Runs 3-4

A.3 NID Controller - 0.15m Lookahead Distance

A.3.1 Hardware Runs

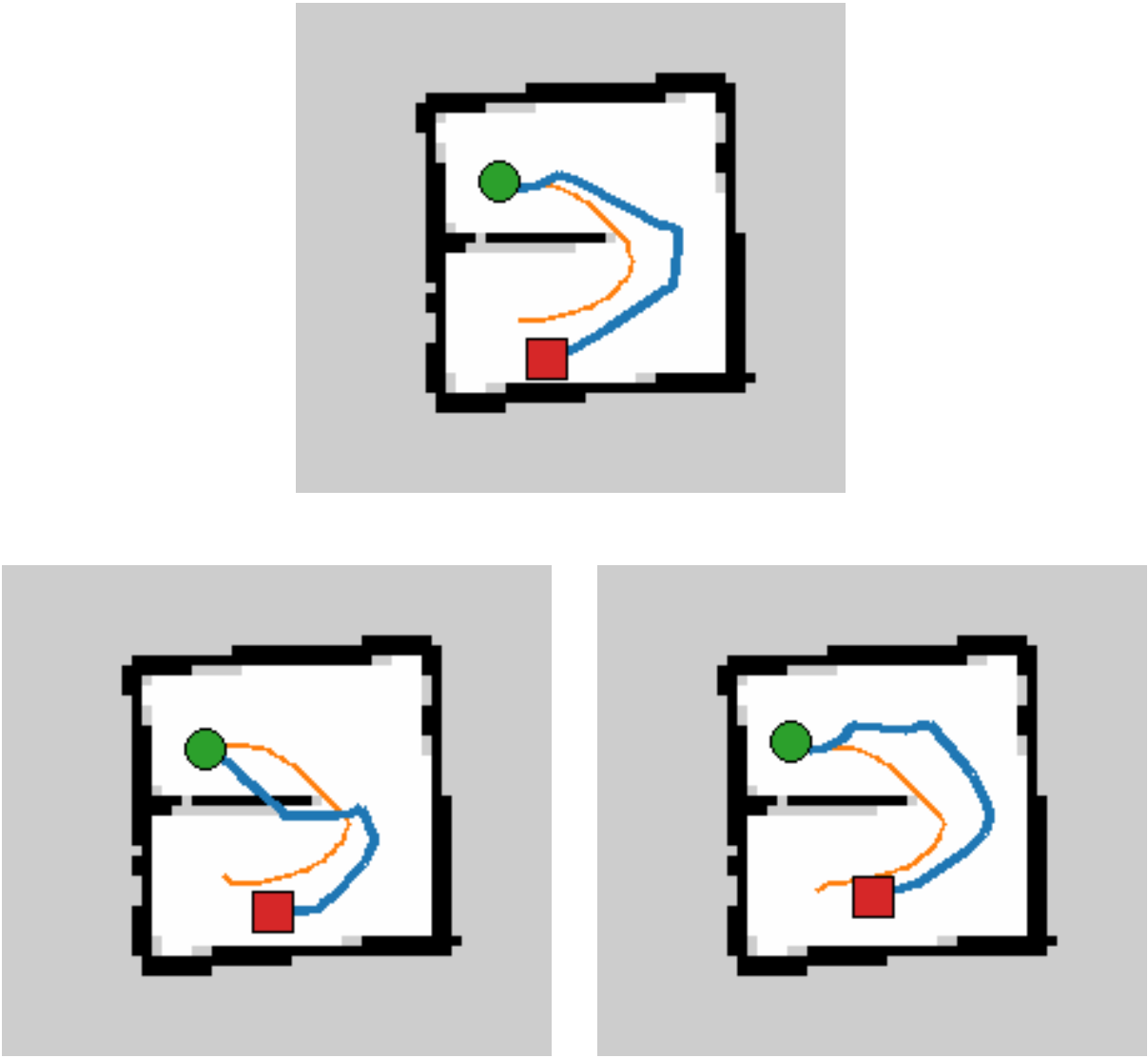


Figure A.10: NID Controller (0.15m lookahead), Hardware Runs 3-4

A.3.2 Standard Environment Runs

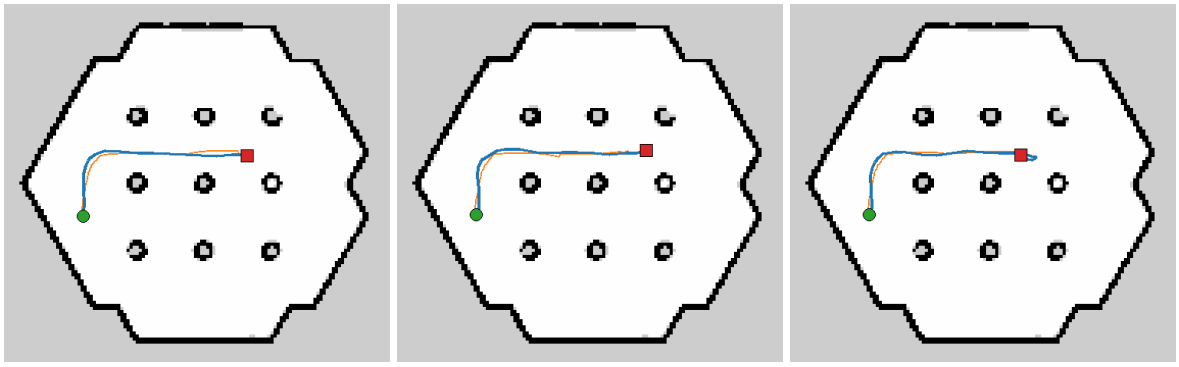


Figure A.11: NID Controller (0.15m lookahead), Standard Runs 1-3

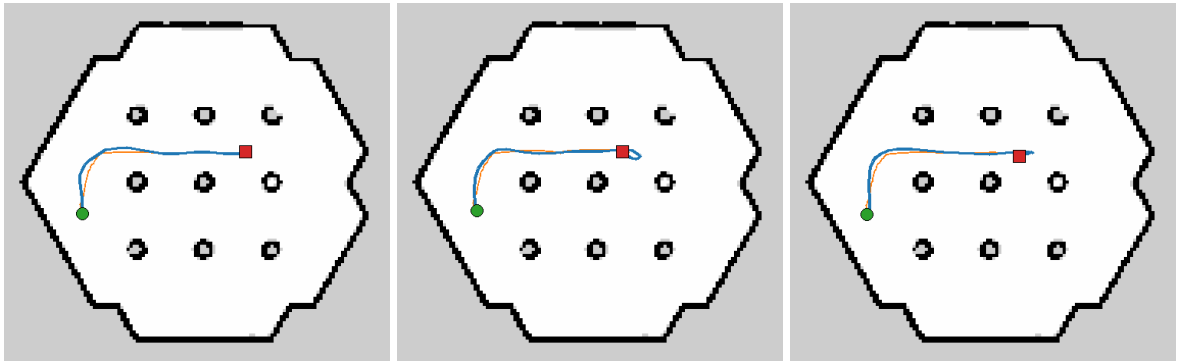


Figure A.12: NID Controller (0.15m lookahead), Standard Runs 4-6

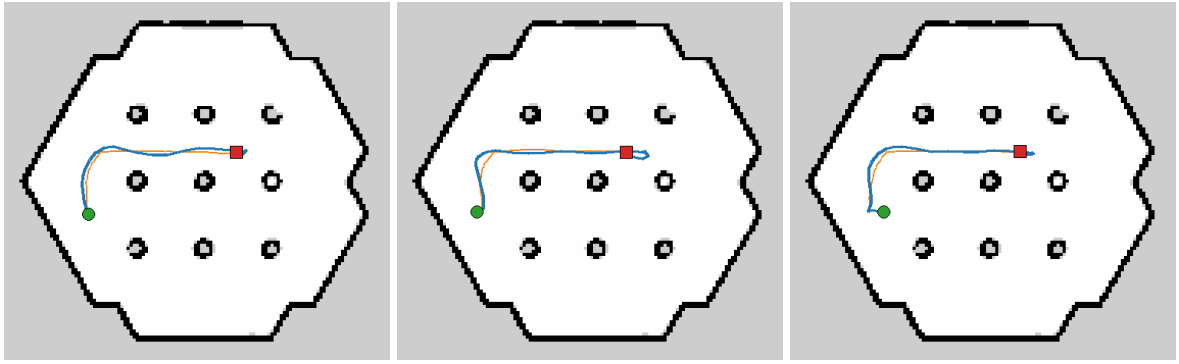


Figure A.13: NID Controller (0.15m lookahead), Standard Runs 7-9

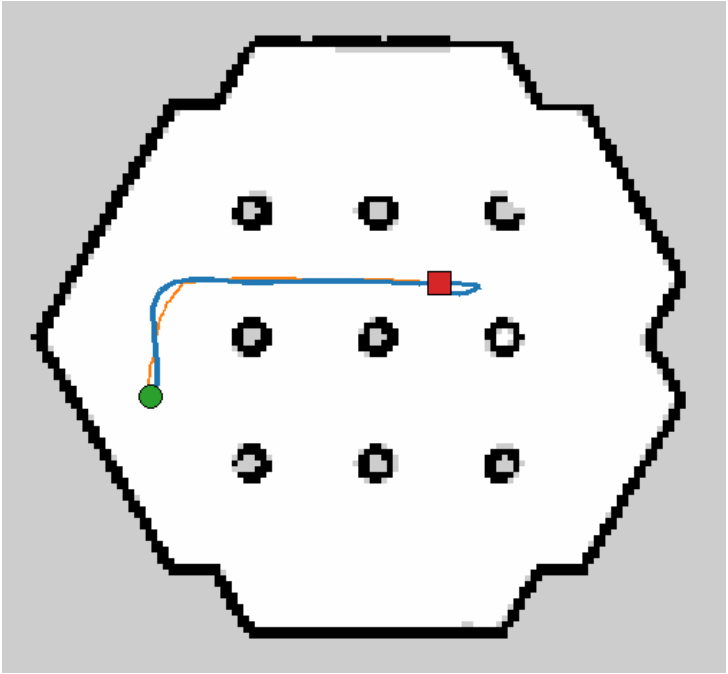


Figure A.14: NID Controller (0.15m lookahead), Standard Run 10

A.4 NID Controller - 1.25m Lookahead Distance

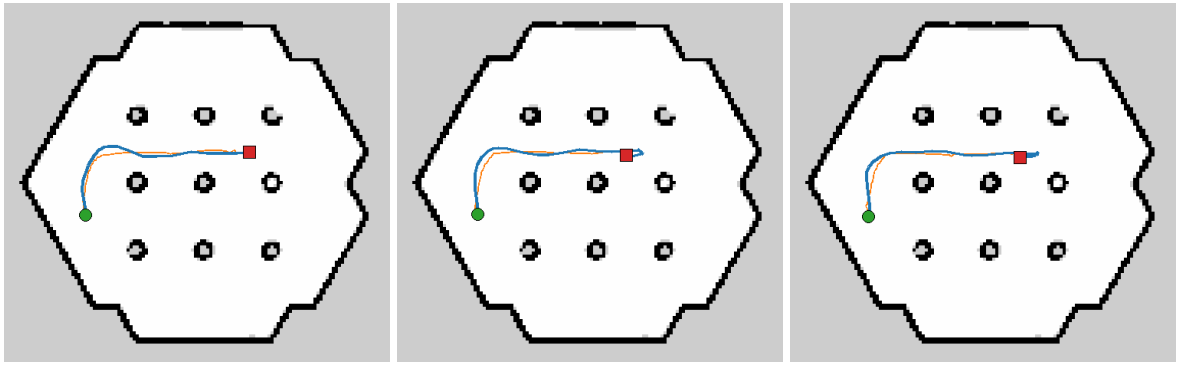


Figure A.15: NID Controller (1.25m lookahead), Runs 4-6

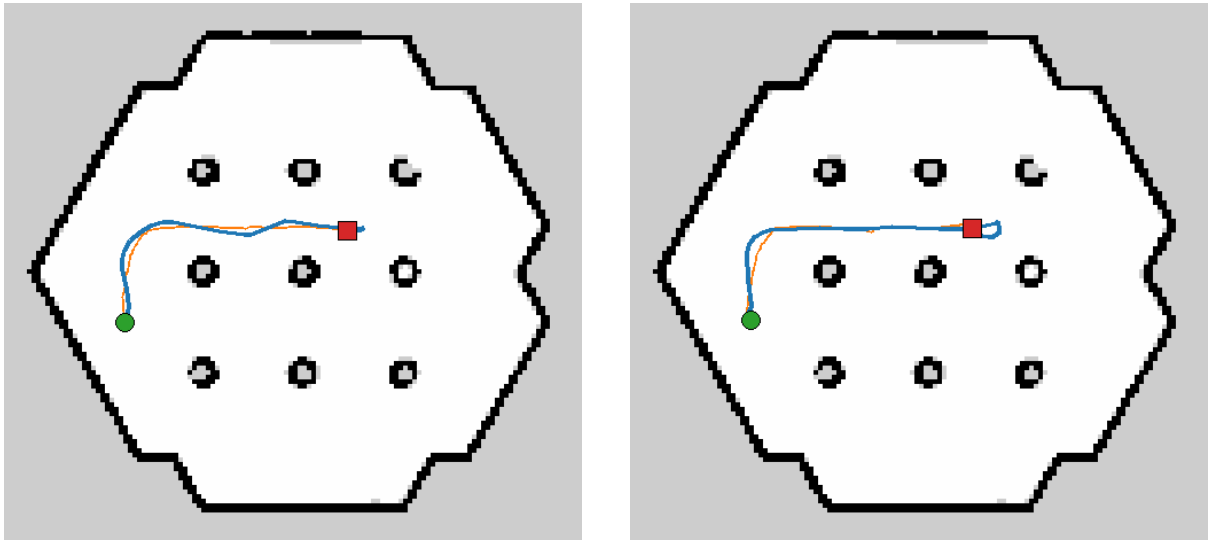


Figure A.16: NID Controller (1.25m lookahead), Runs 7-8

A.5 NID Controller - 1.75m Lookahead Distance

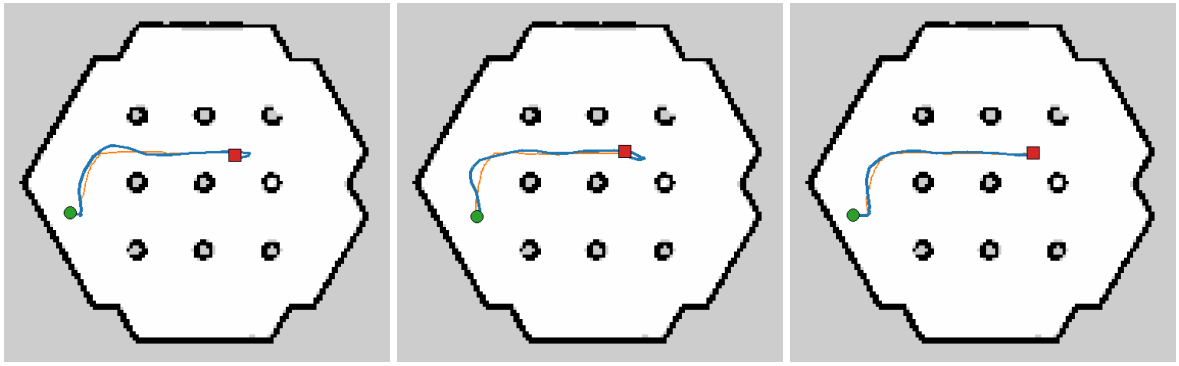


Figure A.17: NID Controller (1.75m lookahead), Runs 1-3

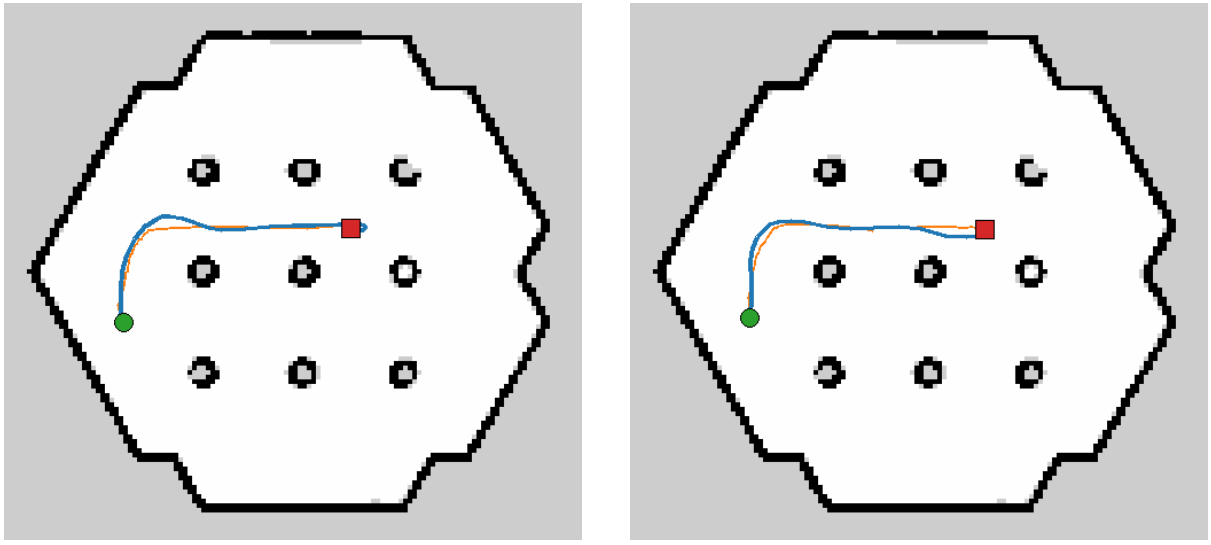


Figure A.18: NID Controller (1.75m lookahead), Runs 4-5

Appendix B

Project Management Artifacts

This provides the formal project management documentation, including the final project timeline and the risk register. This was to document the project's adaptive management process and its proactive approach to risk mitigation.

B.1 Risk Assessment and Mitigation

A formal risk assessment was conducted at the project's outset and revisited again towards the final stages before testing. Table B.1 details the most significant identified risks, the planned mitigation strategies, and the actual outcomes, demonstrating a complete risk management cycle from identification to successful mitigation.

Table B.1: Project Risk Register with Mitigation Strategies and Outcomes

Risk	Impact	Planned Mitigation Strategy	Actual Outcome & Implemented Mitigation
High-Speed Collision	High	Develop a controller with inherently safer characteristics; conduct all initial testing in simulation.	**Successfully Mitigated.** The custom NID-DWB controller was developed and validated, achieving a measurably higher minimum safety margin and a 200% improvement in success rate in hardware tests.
Timeline Overrun	Medium	Allocate buffer time in the project plan; identify critical path tasks early; adopt an adaptive management style.	**Successfully Managed.** The strategic pivot to NID development utilized the buffer time. The project successfully adapted to the new timeline, meeting the final submission deadline.
Computational Limitations	Medium	Use a virtual machine to standardize the environment; monitor system performance.	**Unsuccessful.** The initial development PC proved insufficient. A breakthrough was eventually made, however this was too late in the project to allow for the other objectives to be attempted.
Network Issues	High	Use static IP addresses; create scripts for network recovery; ensure a consistent ROS_DOMAIN_ID.	**Partially Mitigated.** Intermittent network issues were a recurring challenge. The use of static IPs and robust error handling in the software provided a partial mitigation, though it remained a source of occasional test failures.

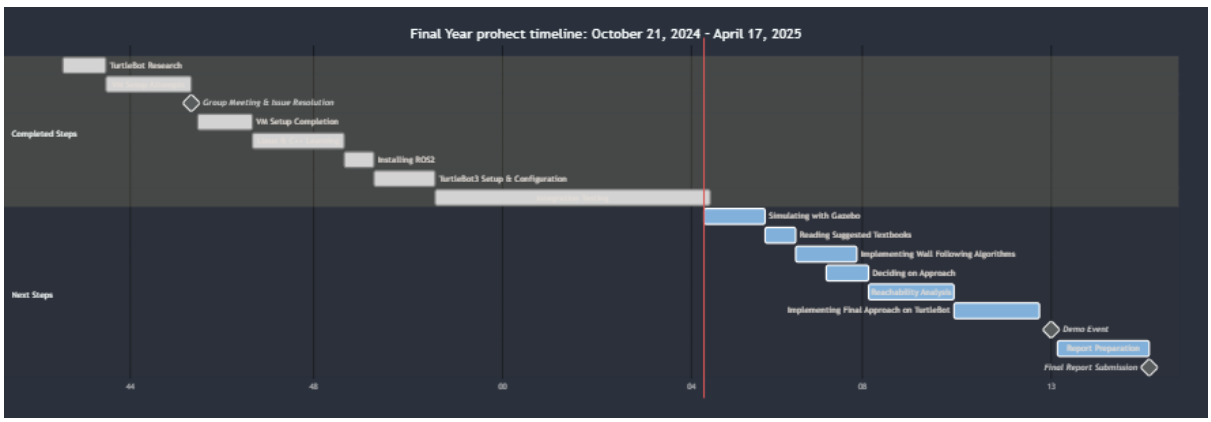


Figure B.1: The planned project timeline.

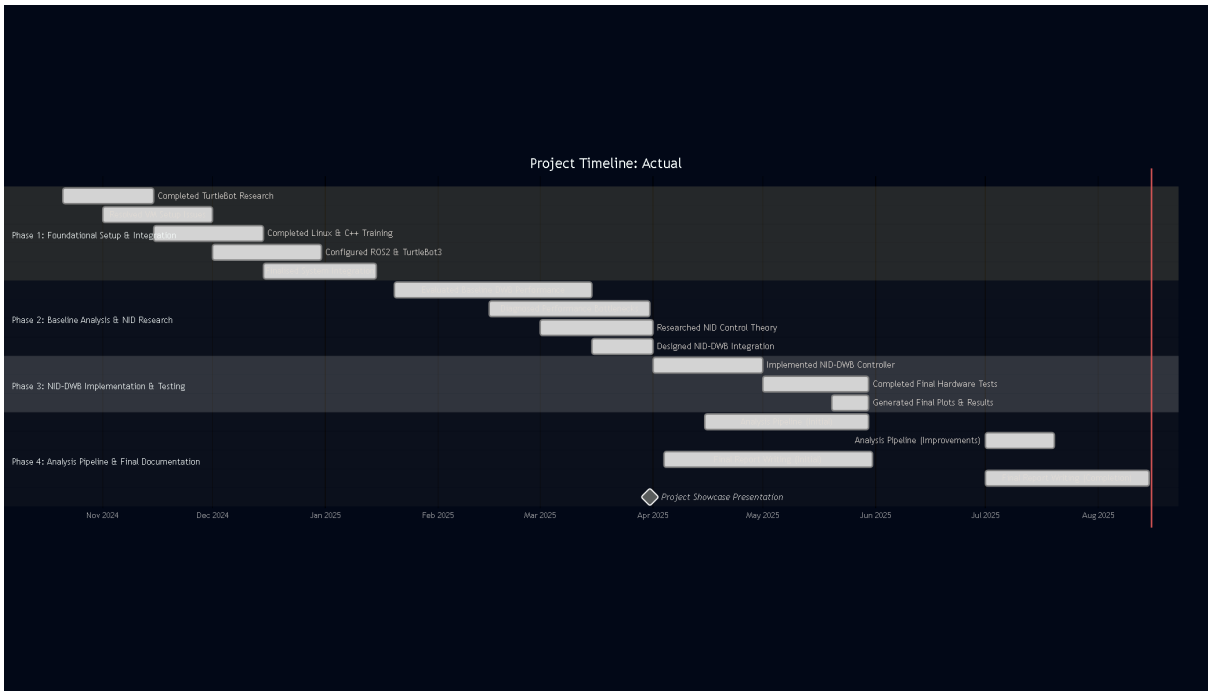


Figure B.2: The final project timeline, showing the actual four-phase adaptive workflow from initial setup to final validation and reporting.

B.2 Ethical Considerations

This project adhered to ethical guidelines for autonomous systems research throughout its lifecycle:

- **Safety First:** All hardware testing was conducted in a controlled, isolated laboratory environment with a option to cancel navigation or stop readily available to us. The controller was also designed with velocity and acceleration limits that respected the hardware's capabilities.
- **Honest Reporting:** This report provides a transparent account of the project's findings, including a full disclosure of failures, limitations, and the small sample size of hardware tests. No results were fabricated or altered.
- **Environmental Impact:** The focus on improving navigation reliability and reducing failed runs has a positive, albeit small, environmental impact by improving the energy efficiency of the battery-powered robot.
- **Open Source Commitment:** The developed code for the Nav2 plugin is intended to be shared with the research community to encourage further development, validation, and innovation in safe navigation.

Appendix C

Custom Controller Code Listings

This appendix provides the core source code and configuration for the custom `NIDDWBPlanner`. It is organised into three sections: the C++ header file defining the planner's structure, the main C++ implementation file detailing the control logic, and the YAML file showing how the planner and its parameters are integrated into the Nav2 stack.

C.1 Planner Header File (`nid_dwb_planner.hpp`)

This header file defines the `NIDDWBPlanner` class, inheriting from the base `DWBLocalPlanner`. It declares the necessary parameters, helper methods, and overrides the core `computeVelocityCommands` function.

```
#pragma once
#include <dwb_core/dwb_local_planner.hpp>
#include <pluginlib/class_list_macros.hpp>
#include <tf2/utils.h>
#include <tf2_geometry_msgs/tf2_geometry_msgs.hpp>
#include <memory>
#include <string>
#include <vector>

namespace nid_dwb_controller
{
class NIDDWBPlanner : public dwb_core::DWBLocalPlanner
{
public:
  void configure(const rclcpp_lifecycle::LifecycleNode::WeakPtr &
     parent,
             std::string name, std::shared_ptr<tf2_ros::Buffer> tf,
             std::shared_ptr<nav2_costmap_2d::Costmap2DROS>
                costmap_ros) override;

  geometry_msgs::msg::TwistStamped computeVelocityCommands(
```

```
    const geometry_msgs::msg::PoseStamped & pose,
    const geometry_msgs::msg::Twist & velocity,
    nav2_core::GoalChecker * goal_checker) override;

private:
  // --- NID parameters ---
  double lookahead_dist_{0.20};
  double K_lat_{1.0};
  double max_vel_x{0.26};
  double max_vel_theta{2.0};
  bool use_dwb_fallback_{true};
  double min_lookahead_dist_{0.10};

  // --- Frame identifiers ---
  std::string robot_frame_{"base_link"};
  std::string global_frame_;

  // --- Logger ---
  rclcpp::Logger logger_{rclcpp::get_logger("NID-DWB")};

  // --- Helper methods ---
  void transformGlobalPlanToRobotFrame(
    const geometry_msgs::msg::PoseStamped & robot_pose,
    std::vector<geometry_msgs::msg::PoseStamped> & plan_robot_frame);

  double findLookAheadPoint(
    const std::vector<geometry_msgs::msg::PoseStamped> & plan,
    double target_distance,
    geometry_msgs::msg::PoseStamped & lookahead_pose);

  bool validateCommand(
    const geometry_msgs::msg::PoseStamped & pose,
    const geometry_msgs::msg::Twist & velocity,
    double v_cmd, double w_cmd);
};
} // namespace nid_dwb_controller
```

C.2 Planner Implementation File (`nid_dwb_planner.cpp`)

This file contains the main logic for the controller. The `configure` method initialises parameters, while the `computeVelocityCommands` method executes the core NID control law at each time step. Helper functions for path transformation and lookahead point calculation are also included.

```
#include "nid_dwb_controller/nid_dwb_planner.hpp"
#include <algorithm>
#include "nav2_util/node_utils.hpp"
#include "tf2/utils.h"

namespace nid_dwb_controller
{

void NIDDWBPlanner::configure(
  const rclcpp_lifecycle::LifecycleNode::WeakPtr & parent,
  std::string name,
  std::shared_ptr<tf2_ros::Buffer> tf,
  std::shared_ptr<nav2_costmap_2d::Costmap2DROS> costmap_ros)
{
  // Let base class handle most configuration
  dwb_core::DWBLocalPlanner::configure(parent, name, tf, costmap_ros);

  auto node = parent.lock();
  robot_frame_ = "base_link";
  global_frame_ = costmap_ros->getGlobalFrameID();

  // Declare and read parameters
  nav2_util::declare_parameter_if_not_declared(node, name +
      ".lookahead_distance", rclcpp::ParameterValue(0.20));
  nav2_util::declare_parameter_if_not_declared(node, name +
      ".lateral_gain", rclcpp::ParameterValue(1.0));
  nav2_util::declare_parameter_if_not_declared(node, name +
      ".max_vel_x", rclcpp::ParameterValue(0.26));
  nav2_util::declare_parameter_if_not_declared(node, name +
      ".max_vel_theta", rclcpp::ParameterValue(2.0));
  nav2_util::declare_parameter_if_not_declared(node, name +
      ".use_dwb_fallback", rclcpp::ParameterValue(true));
  nav2_util::declare_parameter_if_not_declared(node, name +
      ".min_lookahead_distance", rclcpp::ParameterValue(0.10));

  node->get_parameter(name + ".lookahead_distance", lookahead_dist_);
  node->get_parameter(name + ".lateral_gain", K_lat_);
  node->get_parameter(name + ".max_vel_x", max_vel_x);
  node->get_parameter(name + ".max_vel_theta", max_vel_theta);
```

```
  node->get_parameter(name + ".use_dwb_fallback", use_dwb_fallback_);
  node->get_parameter(name + ".min_lookahead_distance",
      min_lookahead_dist_);
}

geometry_msgs::msg::TwistStamped
NIDDWBPlanner::computeVelocityCommands(const
    geometry_msgs::msg::PoseStamped & pose,
                                const geometry_msgs::msg::Twist &
                                    velocity,
                                nav2_core::GoalChecker * goal_checker)
{
  if (global_plan_.poses.empty()) {
    RCLCPP_WARN(logger_, "Received empty global plan");
    return geometry_msgs::msg::TwistStamped();
  }

  try {
    // 1. Transform global plan to robot's local frame
    std::vector<geometry_msgs::msg::PoseStamped> plan_robot_frame;
    transformGlobalPlanToRobotFrame(pose, plan_robot_frame);

    // 2. Find the lookahead point on the local plan
    geometry_msgs::msg::PoseStamped target_pose;
    double current_lookahead = findLookAheadPoint(plan_robot_frame,
        lookahead_dist_, target_pose);

    // 3. Compute single-integrator velocity commands (u_x, u_y)
    double u_x = max_vel_x; // Simplified for clarity
    double u_y = K_lat_ * target_pose.pose.position.y; // Proportional
        lateral correction

    // 4. Apply the NID inverse transformation to get (v, w)
    double v_cmd = u_x;
    double w_cmd = u_y / current_lookahead;

    // 5. Apply velocity limits and validate command
    v_cmd = std::clamp(v_cmd, -max_vel_x, max_vel_x);
    w_cmd = std::clamp(w_cmd, -max_vel_theta, max_vel_theta);

    if (use_dwb_fallback_ && !validateCommand(pose, velocity, v_cmd,
        w_cmd)) {
      RCLCPP_WARN(logger_, "NID command invalid, falling back to
          DWB");
      return DWBLocalPlanner::computeVelocityCommands(pose, velocity,
          goal_checker);
    }
```

```
    // 6. Construct and return the command message
    geometry_msgs::msg::TwistStamped cmd_vel;
    cmd_vel.header.stamp = this->clock_->now();
    cmd_vel.header.frame_id = robot_frame_;
    cmd_vel.twist.linear.x = v_cmd;
    cmd_vel.twist.angular.z = w_cmd;
    return cmd_vel;

  } catch (const tf2::TransformException & ex) {
    RCLCPP_WARN(logger_, "Transform exception: %s", ex.what());
    // Fallback to DWB if transforms fail
    return DWBLocalPlanner::computeVelocityCommands(pose, velocity,
        goal_checker);
  }
}
// ... Other helper methods like findLookAheadPoint etc. would follow
    ...
} // namespace nid_dwb_controller

// Register the plugin with class loader
#include "pluginlib/class_list_macros.hpp"
PLUGINLIB_EXPORT_CLASS(nid_dwb_controller::NIDDWBPlanner,
    nav2_core::Controller)
```

C.3 Nav2 Parameter Configuration (`nav2_params.yaml`)

This YAML file is used to configure the entire Nav2 stack. The critical section for this project is within `controller_server`, where the `FollowPath` controller plugin is set to my `NIDDWBPlanner` and its specific parameters, such as `lookahead_distance`, are defined.

```
controller_server:
  ros__parameters:
    controller_frequency: 10.0
    goal_checker_plugins: ["general_goal_checker"]
    controller_plugins: ["FollowPath"]

    # ... other parameters ...

    # NID-DWB Controller Parameters
    FollowPath:
      plugin: "nid_dwb_controller::NIDDWBPlanner"

      # Core NID Parameters
      lookahead_distance: 0.15 # Optimal distance found in simulation
      lateral_gain: 1.0      # Proportional gain for path correction
      min_lookahead_distance: 0.10 # Safety floor for lookahead distance
      use_dwb_fallback: false # Disabled for pure NID behaviour in tests

      # Velocity Limits
      max_vel_x: 0.22
      max_vel_theta: 1.5
      min_vel_x: 0.0

      # DWB Critic settings (used only if fallback is enabled)
      critics: ["RotateToGoal", "Oscillation", "BaseObstacle",
         "GoalAlign",
             "PathAlign", "PathDist", "GoalDist"]
      # ... critic weights ...
      BaseObstacle.scale: 0.02
      PathAlign.scale: 32.0
      PathDist.scale: 32.0
      GoalAlign.scale: 24.0
      GoalDist.scale: 24.0
```

Appendix D

Data Analysis Pipeline

This appendix provides pseudocode for the data processing pipeline, detailing the workflow from raw experimental data to the final aggregated performance charts presented in the report.

Listing C.1: Per-Run Data Processing and Analysis

```
// Accepts:
//  - A set of directories containing raw ROS2 bag files.
//  - A 'trusted_runs.txt' file listing the specific bag names to
   process.
//
// Outputs:
//  - For each trusted run, a subdirectory (e.g., 'run_name_csv/')
   containing
//    CSV files for key topics (/amle_pose, /plan, /scan, etc.).
//  - For each trusted run, a second subdirectory (e.g.,
   'run_name_analysis/') containing
//    individual analysis plots and a 'run_metrics.json' summary file.

PROCEDURE ProcessAllExperimentalRuns
   // 1. INITIALISATION
   Load list of trusted run names from 'trusted_runs.txt' into a Set.
   Define paths for input bag directories and the main output
      directory.

   // 2. MAIN PROCESSING LOOP
   FOR EACH directory containing bag files:
      FOR EACH bag_file in the directory:
         bag_name <- GetBaseName(bag_file)

         // 3. FILTERING
         IF bag_name is NOT IN trusted_run_names THEN
            CONTINUE to next bag_file // Skip non-trusted runs.
```

```
        END IF

        // 4. DATA EXTRACTION (from bag to CSV)
        csv_output_path <- "output_dir/" + bag_name + "_csv/"
        IF csv_output_path does NOT exist THEN
           Initialise data containers for key topics.
           Open bag_file for reading.
           FOR EACH message in bag_file:
              // Extract data and append to the relevant container.
              // CRITICAL: Use /amcl_pose for the robot's true
                 global position.
           END FOR
           Save each data container to its own CSV file.
        END IF

        // 5. INDIVIDUAL RUN ANALYSIS
        analysis_output_path <- "output_dir/" + bag_name +
           "_analysis/"
        Load all generated CSV files into data frames.
        Apply motion trimming to data.

        // Generate and save individual analysis plots.
        PlotTrajectoryComparison(planned_path, executed_path)
        PlotSafetyAnalysis(obstacle_distance_over_time)

        // Calculate key metrics and save to 'run_metrics.json'.
        CalculateMetric(total_duration)
        CalculateMetric(min_safety_margin)
        CalculateMetric(mean_path_tracking_error)
      END FOR
   END FOR
END PROCEDURE
```

Listing C.2: Trajectory Map Overlay Visualisation

```
// Accepts:
//  - The path to a directory containing a run's CSV data (e.g.,
   'run_name_csv/').
//  - The path to the relevant 'map.yaml' file for the environment.
//  - The full desired path for the output image file.
//
// Outputs:
//  - A single high-resolution PNG image showing the planned and
   executed
//   trajectories overlaid on the environment map.

PROCEDURE GenerateMapOverlay
   // 1. LOAD MAP DATA
   Load map metadata (resolution, origin) from the 'map.yaml' file.
   Load map image file (.pgm) specified in the metadata.

   // 2. UPSCALE CANVAS FOR HIGH-RESOLUTION DRAWING
   Upscale map_image by a defined factor (e.g., 4x).
   Adjust map_resolution to match the new upscaled dimensions.

   // 3. LOAD TRAJECTORY DATA
   Load executed path coordinates from 'amcl_pose.csv'.
   Load planned path coordinates from 'plan_detailed.csv'.

   // 4. COORDINATE TRANSFORMATION
   executed_pixels <- WorldToPixel(executed_path, map_origin,
      adjusted_resolution).
   planned_pixels <- WorldToPixel(planned_path, map_origin,
      adjusted_resolution).

   // 5. DRAWING
   Initialise a drawing context on the upscaled map image.
   Draw planned_pixels as a dashed orange line.
   Draw executed_pixels as a solid blue line.
   Draw a green circle at the start and a red square at the end.

   // 6. SAVE IMAGE
   Save the final composite image to the specified output path.
END PROCEDURE
```

Listing C.3: Aggregate Performance Analysis and Plotting

```
// Accepts:
//  - The base directory containing all the individual '..._analysis'
    folders.
//
// Outputs:
//  - A set of summary PNG bar charts saved to the base directory,
    comparing
//    performance metrics across different experimental groups.

PROCEDURE GenerateSummaryPlots
   // 1. DATA AGGREGATION
   all_run_metrics <- empty List.
   Find all 'run_metrics.json' files within the base directory.

   FOR EACH json_file found:
      metrics <- LoadJSON(json_file).
      run_name <- GetRunNameFromFilePath(json_file).

      // Categorise the run based on keywords in its name.
      metrics.controller <- CategoriseController(run_name)
      metrics.lookahead <- GetLookaheadValue(run_name)

      Append metrics to all_run_metrics.
   END FOR

   // 2. SIMULATION ANALYSIS: LOOKAHEAD OPTIMISATION
   sim_data <- FILTER all_run_metrics where controller is 'DWB+NID
      (Sim)'.
   sim_summary <- GROUP sim_data BY 'lookahead' and AGGREGATE('mean',
      'std').

   // Generate and save a bar chart for each metric of interest.
   PlotBarChart(sim_summary, metric='min_safety_margin_m',
      title='Lookahead Impact on Safety').
   PlotBarChart(sim_summary, metric='path_tracking_error_m',
      title='Lookahead Impact on Accuracy').

   // 3. HARDWARE ANALYSIS: CONTROLLER COMPARISON
   hw_data <- FILTER all_run_metrics where controller CONTAINS '(HW)'.
   hw_summary <- GROUP hw_data BY 'controller' and AGGREGATE('mean',
      'std').

   // Generate and save comparative bar charts.
```

```
   PlotBarChart(hw_summary, metric='min_safety_margin_m',
      title='Hardware Controller Safety').
   PlotBarChart(hw_summary, metric='duration_s', title='Hardware
      Controller Efficiency').
END PROCEDURE
```

Bibliography

[1] Robotics in social care. Parliamentary Office of Science and Technology Report POST-PN-0656, UK Parliament POST, November 2021. URL https://post.parliament.uk/research-briefings/post-pn-0656/.

[2] Automation in horticulture review. Technical report, Department for Environment, Food & Rural Affairs, July 2022. URL https://www.gov.uk/government/publications/automation-in-horticulture-review. GOV.UK Technical Report.

[3] Automation in agriculture can improve sustainability, but requires investment and policies, November 2022. URL https://www.fao.org/newsroom/detail/automation-in-agriculture-can-improve-sustainability-but-requires-investment-and-policies/en. Press Release.

[4] Nikolaos Athanasopoulos. TURTLEBOT(s) navigation: Fast and furious (but provably safe) - project brief. Technical report, School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, 2024.

[5] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine*, 4(1):23–33, March 1997. doi: 10.1109/100.580977.

[6] Dieter Fox, Wolfram Burgard, Frank Dellaert, and Sebastian Thrun. Monte carlo localization: Efficient position estimation for mobile robots. In *Proceedings of AAAI/IAAI*, pages 343–349. AAAI Press, 1999.

[7] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. ISBN 0521862051.

[8] Steven Macenski. *Navigation2 Documentation*, 2025. URL https://navigation.ros.org/. Accessed: January 2025.

[9] Steven Macenski and Carl Delsey. nav2_dwb_controller, 2025. URL https://github.com/ros-planning/navigation2. ROS 2 Humble Package.

[10] Steven Macenski, Francisco Martin, Ruffin White, and Jonatan Ginés. The marathon 2 navigation stack. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1–8, 2020. doi: 10.1109/IROS45743.2020.9341207.

[11] Sean McGovern, Nikolaos Athanasopoulos, and Seán McLoone. Safe set based trajectory planning for robotic manipulators underactuated in gravity. In *Proceedings of the American Control Conference (ACC)*, pages 1–6, 2024. doi: 10.23919/ACC60939.2024.10644920.

[12] *ROS 2 Humble Hawksbill Documentation*. Open Source Robotics Foundation, 2025. URL https://docs.ros.org/en/humble/index.html. Accessed: January 2025.

[13] Dong-hyung Park and Benjamin Kuipers. A smooth control law for graceful motion of differential drive mobile robots. *Robotics and Autonomous Systems*, 59(6):414–427, 2011. doi: 10.1016/j.robot.2011.02.005.

[14] *ROBOTIS e-Manual for TurtleBot3*. ROBOTIS, 2025. URL https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/. Accessed: January 2025.

[15] ROBOTIS. Turtlebot3 models, n.d. URL https://emanual.robotis.com/assets/images/platform/turtlebot3/hardware_setup/turtlebot3_models.png. Accessed: 2025-08-15.

[16] SICK AG. Lidar technology white paper. Technical report, SICK AG, 2017. URL https://www.sick.com/media/docs/3/63/963/whitepaper_lidar_en_im0079963.pdf. Accessed: 2025-08-15.

[17] Mark W. Spong, Seth Hutchinson, and M. Vidyasagar. *Robot Modeling and Control*. Wiley, Hoboken, NJ, USA, 2005. ISBN 0471649902.

[18] YoYoClouds. Five best vm creation applications, 2012. URL https://yoyoclouds.wordpress.com/2012/04/24/five-best-vm-creationapplications/. Accessed: 2025-08-15.