University of Verona

Department of Computer Science and Engineering

High-Level Control with RANSAC

Submitted by:Fitsum Sereke Tedla (VR496807)
Hailemicael Lulseged Yimer (VR512127)

Supervised by: Professor Alessandro Franelli
Professor Daniele Meli
Dr. Francesco Trotti

July 25, 2024

Acknowledgements

we would like to express our deepest gratitude to Professor Alessandro Franelli and Professor Daniele Meli for their invaluable guidance, support, and mentorship throughout this project. Their expertise and insights were instrumental in shaping the direction and success of this work.

we are also profoundly grateful to all the staff members of the Robotics Department for their continuous support, encouragement, and the resources they provided. Their dedication and commitment to fostering a collaborative and innovative environment have greatly contributed to the development and completion of this project.

Thank you all for your unwavering support and for making this project a rewarding and enriching experience.

Abstract

The project began with establishing both general and specific objectives. The general objectives included achieving autonomous navigation, integrating various sensors, ensuring robust obstacle avoidance, and creating a scalable and adaptable system. Specifically, the project aimed to develop a wall-following robot that could autonomously navigate and maintain a consistent distance from walls using the RANSAC algorithm.

The methodology involved both simulation and real-world implementation. Initially, the software environment was configured using Unity Hub to simulate the robot's operations. This involved creating a virtual model of the TurtleBot3 and its environment, programming its actions, and integrating the necessary sensors and actuators. The use of Unity Hub allowed for rapid prototyping and iterative improvements in a controlled environment.

The core of the project was the implementation of the RANSAC algorithm within the TurtleBot3's control system. The RANSAC algorithm was essential for interpreting sensor data, particularly from LIDAR, to detect walls and differentiate them from other objects. A Finite State Machine was employed to manage the robot's behavior, allowing it to switch between states such as finding the wall, following the wall, and aligning itself based on real-time sensor inputs.

After successful simulations, the project transitioned to hardware testing using the actual TurtleBot3 robot. This phase involved validating the system's performance in real-world scenarios where sensor noise and environmental unpredictability are more prominent. The robot's ability to follow walls, avoid obstacles, and maintain a consistent distance was evaluated, with the RANSAC algorithm proving crucial in processing real-time sensor data and ensuring reliable navigation.

Overall, the project demonstrated the practical application of the RANSAC algorithm in mobile robotics, highlighting its effectiveness in enhancing the autonomy and reliability of robots for complex navigational tasks. The process underscored the importance of iterative testing and refinement in both simulated and real-world environments to achieve robust robotic behavior.

Dedication

This work is dedicated to every Robotics Society around the world. Your passion, innovation, and relentless pursuit of excellence in the field of robotics inspire us all. May this project contribute in some small way to the incredible advancements you continue to make every day. Thank you for your dedication and commitment to pushing the boundaries of what is possible.

Contents

List of Figures

*

Acronyms

FSM Finite State Machine

GPS Global Positioning Systems

IPD Proportional Integral Derivative

IMU Inertial measurement units

LIDAR Light Detection and Ranging

RANSAC Random sample consensus

ROS Robot Operating System

SLAM Simultaneous Location and Mapping

Chapter 1

Introduction

The discipline of mobile robotics is multidisciplinary, combining components of artificial intelligence, computer science, mechanical engineering, and electrical engineering to create robots that can move and function in a variety of contexts. These machines, sometimes referred to as mobile robots, vary in complexity from basic wheeled vehicles to intricate autonomous drones and humanoid robots. They vary from stationary robots in that they can traverse and adjust to various environments and terrains, which makes them especially useful for a variety of applications such as industrial automation, service industries, and exploratory work.

Among the most difficult problems in mobile robots is motion planning and navigation. This entails the robot's capacity to ascertain its location within the surroundings and map out an effective path to go to its target while dodging impediments. To get information about the robot's environment, sophisticated sensors like LIDAR, GPS, cameras, and inertial measurement units are frequently employed. Subsequently, sophisticated algorithms are employed to interpret this sensory data in order to generate intricate environment maps and make judgments regarding the robot's trajectory in real time.

Developing and putting into practice efficient control systems is another crucial component of mobile robots. These systems are in charge of handling sensor inputs and directing the robot's actuators to carry out activities. Depending on the necessary degree of autonomy, these control systems' complexity might vary greatly. While some robots require human supervision or are operated manually, others are completely autonomous and have artificial intelligence built in to allow them to make decisions on their own.

Robust mechanical designs that meet the physical demands of mobility in a variety of situations are also necessary for mobile robots. This includes the creation of several means of propulsion, such legs, wheels, tracks, or even hybrid systems. Each design has a unique set of benefits, and the selection process is based on the particular requirements of the intended use, including agility, speed, stability, and the capacity to navigate challenging terrain.

Mobile robots appears to have a bright future because to ongoing technological breakthroughs and increasing AI integration. These developments are expanding the applications of mobile robots

into new fields while also enhancing their efficiency and capacities. It is anticipated that future advancements will center on boosting robot autonomy, augmenting their communication with humans and other machines, and guaranteeing that their functions are morally and securely executed. With their further development, mobile robots will surely become an ever more essential aspect of contemporary life, transforming both daily tasks and whole businesses.

1.1 General Objectives

The following would be the broad goals of a project that employed LIDAR sensors and the RANSAC algorithm [2] to create a wall-following robot:

Autonomous Navigation:to create a robot that can follow walls in order to navigate complicated settings on its own. This entails the robot's capacity to autonomously identify barriers, maintain a predetermined distance, and negotiate twists and curves without assistance from a human.

Sensor Integration and Data Fusion:to efficiently combine many sensors, mainly LIDAR, with potential ultrasonic, camera, and inertial measurement units (IMUs) to collect extensive ambient data. The aim is to improve the robot's perception and decision-making abilities by combining input from several sources.

Robust Obstacle Avoidance:to put the RANSAC algorithm into practice and refine it such that sensor data may be processed in real time. In order to enable more accurate modeling of the robot's local surroundings, this entails configuring the robot to recognize and ignore outlier sensor signals.

Repeatability and Reliability:to create a system that can function consistently and dependably in a range of situations. This involves making certain the robot can operate accurately and consistently over a range of environmental configurations and situations.

Scalability and Adaptability:to create a scale-able and flexible robot and related technologies for a range of uses, including home help, inspection, and surveillance. The goal would also include addressing how well the hardware and software work with various robot kinds, sizes, and forms.

By combining these goals, we want to push the limits of what is currently possible for robots and improve their functionality, autonomy, and capacity to navigate and operate in structured settings.

1.2 Specific Objectives

The goal of this project is to create a wall-following robot that can walk independently and keep a constant distance from walls by applying the RANSAC algorithm. The robot's first job is to find a wall and head straight for it. It makes sure to halt when it reaches a predetermined threshold

distance by continually collecting data points as it gets closer to the wall.

The robot is designed to turn left when it reaches this threshold distance, keeping the required distance from the wall to prevent any possible collisions. With this move, the robot may go along the wall in a safe and regulated manner. Here, the RANSAC technique is essential for interpreting sensor data effectively and differentiating the wall from other adjacent objects or abnormalities that might obstruct navigation.

along the wall, staying within the predetermined distance until it comes straight up against another wall. At this time, it will evaluate and carry out maneuvers once more to ensure uninterrupted, seamless navigation. The goal of this research is to show how the RANSAC approach may be used in real-world robotics applications, specifically to improve the autonomy and dependability of robots carrying out difficult navigational tasks like wall following.

Chapter 2

Methodology

In robotics, especially in the field of mobile robotics, high-level control refers to the decision-making mechanisms that allow robots to carry out difficult tasks on their own. In applications where robots must travel through or interact with dynamic, unstructured surroundings, this type of control is essential. Robotics high-level control systems use sensor data to make strategic judgments regarding navigation, avoiding obstacles, and carrying out tasks. These choices are made using a variety of inputs, including remote human inputs sometimes, pre-programmed tasks, learning experiences, and real-time sensor data.

A organized method is necessary to develop a project that involves hardware applications on TurtleBot3 and software simulation in Unity Hub. Using Unity Hub, the software environment must be configured as the initial step. To make sure you have the right version of the Unity Editor for your simulation needs, Unity Hub makes it easier to manage numerous Unity installations and projects. The TurtleBot3's operating environment may be virtually modeled using Unity. To replicate interactions in the actual world, this entails building the 3D environment, programming the robot's actions, and integrating sensors and actuators. High-level control and decision-making capabilities may be achieved by implementing the RANSAC (Random Sample Consensus) algorithm within this virtual setup for tasks like as object identification, feature matching, and motion estimation.

In this project, I implemented the RANSAC high-level control system using the TurtleBot3 robot, leveraging both simulation and real-world hardware testing. Initially, the system was developed and tested in a simulated environment using Unity Hub. This allowed for rapid prototyping and iterative improvement of the algorithm. The simulation was essential for fine-tuning the parameters and ensuring the algorithm performed as expected under controlled conditions.

Once the RANSAC algorithm demonstrated satisfactory performance in the simulated environment, I transitioned to hardware testing using the actual TurtleBot3 robot. This step was crucial for validating the system's functionality in real-world scenarios, where sensor noise and environmental unpredictability are more prominent. During both simulation and hardware testing, I measured the inlier points identified by the RANSAC algorithm. These inlier points were critical for detecting wall segments and determining the robot's position relative to the wall. The algorithm used these

points to decide when to turn left and when to follow the wall, ensuring efficient and accurate navigation.

2.1 RANSAC

Random Sample Consensus, or RANSAC, is a statistical technique essential to high-level control systems, particularly when it comes to reliably fitting models and interpreting data. RANSAC is especially helpful in settings where noise and outliers in sensor data are expected to occur, since these are frequent problems in mobile robots. The procedure selects a random portion of the original data repeatedly, fits a model to this subset, then counts the number of observations in the original dataset that agree with the proposed model to validate the model. Until the model with the greatest support is identified, this procedure is repeated.

RANSAC's strength is its ability to withstand outliers, which makes it an excellent choice for the interpretation of sensor data required for robots' autonomous navigation and mapping. RANSAC, for instance, assists in finding and matching related spots between two pictures when using stereo vision for depth perception, eliminating outliers that do not fit the model. This capability makes sure that the robot perceives its surroundings accurately and consistently, which is important for safe navigation and efficient job completion.

RANSAC is widely used in combination with algorithms such as SLAM (Simultaneous Localization and Mapping) in mobile robotics, where the objective is to map an unfamiliar environment while tracking the robot's position inside it. RANSAC improves the accuracy and dependability of the SLAM algorithm by guaranteeing that the features used to create the map and calculate the robot's location are not impacted by false readings. RANSAC's importance in high-level control systems, where accuracy and confidence in spatial knowledge are essential, is shown by this combination with SLAM [4].

RANSAC (Random Sample Consensus) is a robust iterative method used to estimate the pa-

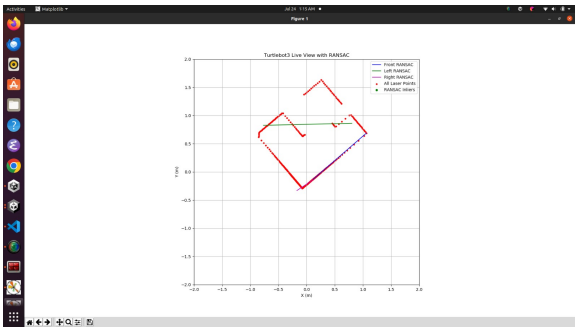

Figure 2.1: Points in green represent the RANSAC wall fit.

rameters of a mathematical model from a dataset that contains outliers. The process begins with

a set of observed data points X = $x_1, x_2, ..., x_n$, which are assumed to be modeled by a model M with parameters θ. The core idea is to repeatedly select random subsets of these data points, fit the model to these subsets, and then evaluate how well this model fits the entire dataset. The fit is evaluated by determining which data points are considered inliers—points that lie within a predefined tolerance ϵ from the model.
For each model instance, the inlier set I is defined as those data points x_i where the distance dist(x_i,M(θ)) from the model is less than or equal to ϵ :

$$I = \{x_i \mid \text{dist}(x_i, M(\theta)) \leq \epsilon\}$$

The RANSAC algorithm involves initializing parameters such as the number of iterations k, distance threshold ϵ and probability p of finding at least one good model. It then randomly samples minimal subsets of the data to fit the model, evaluates the inliers, and updates the best model if it has more inliers than previously found models. The process is repeated for k iterations or until a satisfactory model is found.

The number of iterations k required to achieve a desired probability p of finding a valid model can be computed using the formula:

$$k = \frac{\ln(1-p)}{\ln(1-(1-p)^m)} \tag{2.1}$$

where m is the number of points needed to fit the model. This iterative approach helps RANSAC effectively handle datasets with significant noise and outliers by focusing on the subset of data that best fits the model.

RANSAC has drawbacks despite its benefits, most relating to parameter tweaking and computing efficiency. The amount of iterations required to discover the optimal model might be computationally costly, and the approach need precise threshold selection for inlier identification. RANSAC is still a vital tool for high-level robotic control, allowing robots to function more independently [1] and successfully in increasingly complicated situations, despite continuous improvements in processing power and algorithm optimization.

2.2 TurtleBot3

TurtleBot3 is a versatile and affordable Robot created by ROBOTIS and Open Robots, Which is designed primarily for education and research. Because of its small and modular design,it is simple to extend and customize with additional components to meet the needs of diverse project requirements. The open-source middle ware Robot Operating System(ROS),which offers a stable foundation for creating and managing robots,powers the platform.with its assortment of sensors,which include LiDAR,Cameras ,and an IMU,TurtleBot3 can carry out tasks including object identification,navigation, and mapping.

With varying degrees of functionality and customization, the TurtleBot3 family comprises mod-

Figure 2.2: TurtleBot3-Burger

els including Burger, Waffle, and Waffle Pi. The Waffle and Waffle Pi offer additional features and processing power for more sophisticated research and applications, while the most economical TurtleBot3 Burger is ideal for classroom use. Single-board computers, like the Raspberry Pi or Intel Joule, are commonly used to power these models. These processors have the computing capability to execute sophisticated algorithms and immediately evaluate sensor data. Due to its modular architecture, TurtleBot3 may be easily upgraded and modified to accommodate a variety of robotics applications.

Because TurtleBot3 is open-source, anybody may access and alter the software and hardware designs, encouraging experimentation and creativity. This platform is a vital resource for learning, experimenting, and creating new robotics solutions since it is extensively used in academic institutions, research laboratories, and by enthusiasts. Because of its low cost and simplicity of use, educators and students may utilize it to get practical experience with ROS, sensor integration, and robot control. Moreover, businesses and startups employ TurtleBot3 for fast prototyping, which enables the testing and iteration of new concepts in practical contexts.

2.3 Unity Hub

Unity Technologies offers a management tool called Unity Hub that makes dealing with Unity installations and projects easier. Users may create and open projects, manage different versions of the Unity Editor, and access a range of Unity resources and services by using Unity Hub, which acts as a central interface. It is intended to improve developer, artist, and designer efficiency and organization inside the Unity environment.

The Hub makes it easier to install several Unity Editor versions to guarantee compatibility with different projects and streamlines the creation and upkeep of Unity environments. Users can work with team members more effectively, rapidly move between projects, and customize project parameters. To help both novice and seasoned users develop their abilities and quickly launch new projects, Unity Hub also provides access to tutorials, templates, and learning materials. In general, Unity Hub is a vital resource for everyone working on Unity projects as it offers a unified and intuitive

interface for project management.

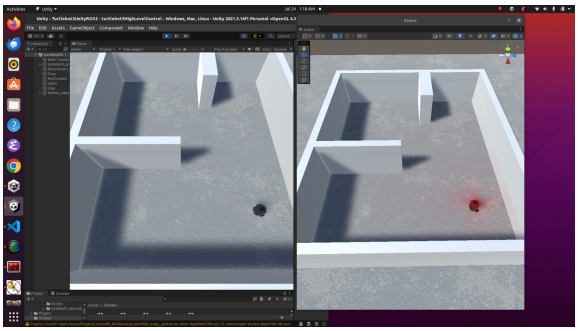

Figure 2.3: Unity Environment

2.4 High-Level Control Overview

In robotics, high-level control refers to the performance of intricate tasks by the management of several subtasks at a higher abstraction level. This sophisticated control method defines the robot's behavior in an organized way using tools like Petri Net Plans, Behavior Trees, and Finite State Machines (FSM). FSM is used for the project because it is easy to use and efficient at handling state-based control logic. The FSM enables the robot to efficiently execute tasks like wall following by allowing it to switch between multiple states based on sensor inputs and specified circumstances. The RANSAC algorithm is used in the Python code created for this project to manage the $TurtleBot3's wall-following$ behavior utilizing FSM logic. The code sets up publishers and subscribers for communication with the robot's sensors and initializes the robot using a customized ROS node. Laser scan data is processed by the $laser_callback$ function, which separates it into front, left, and right regions. Within these locations, consistent lines are found using the RANSAC method. The primary control loop uses a dictionary to handle state transitions and provide particular actions for each state as it executes on a regular basis. When the robot is in the "$FindWall$" mode, it advances and looks for barriers using RANSAC. The robot enters the "$FollowWall$" stage when it detects a wall, keeping a certain distance parallel to the wall. The robot enters the "$AlignLeft$" mode and uses RANSAC to realign and make sure the wall stays on its right side if it runs into an impediment or needs to modify its alignment.

The Python code allows the TurtleBot3 to explore complicated areas with robustness and efficiency by merging RANSAC with FSM. While the FSM offers an organized method for alternating between several behaviors based on $real_time$ sensor inputs, the RANSAC algorithm aids the robot in precisely detecting barriers and managing noisy sensor data. This combination guarantees consistent and seamless wall-following performance, proving that $high_level$ control techniques are useful in autonomous robotic systems [3]. Lasercallback, the custom ROS Node, state management through

dictionaries, and the main control loop demonstrate how state-based control logic and sophisticated algorithms can be used to accomplish complex navigation tasks, opening the door for more sophisticated robotics applications.

2.5 FSM using RANSAC

Three states of an FSM are used to construct the wall-following algorithm: Find Wall, Follow Wall, and Align Left. When in the "Find Wall" stage, the robot utilizes its sensors to detect the presence of walls. The robot moves into the Follow Wall mode when it detects a wall, maintaining a certain perpendicular distance d from it while traveling parallel to the wall.

The distance d is comuted using the formula:

$$d = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}} \tag{2.2}$$

where:

- a, b, and c are coefficients of the line equation.
- (x_0, y_0) are the coordinates of the point.
- $|ax_0 + by_0 + c|$ is the absolute value of the line equation evaluated at (x_0, y_0).
- $\sqrt{a^2 + b^2}$ normalizes the distance.

The robot enters the Align Left mode to realign itself and make sure the wall stays on its right side if it senses an obstruction or needs to change its alignment. Wall-following behavior is kept seamless and efficient by regulating the changes between these states using sensor readings and pre-established criteria.

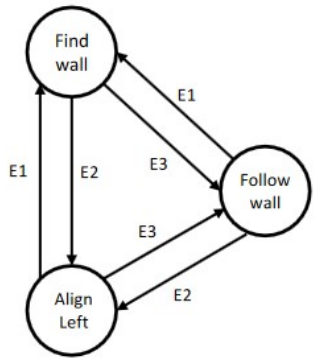

Figure 2.4: The three States of FSM and the transitions

Where

- E1: If right region is greater than Th

- E2: If front region is less than Th and other regions are greater, If front and right regions are less than Th and other region is greater, If all Lidar regions are less than Th
- E3: If front and left regions are greater than Th and other regions are less.

2.6 Parameters and Execution Flow

The required distance from the wall, linear and angular velocities, and lidar range thresholds are important factors for the FSM. Three zones are identified by the robot's lidar readings: front, left, and right. Based on these data, the FSM transitions are created. For instance, if the right lidar area above a certain threshold, the robot switches from Find Wall to Follow Wall. Each state's fundamental actions are motions, such forward motion, rotation, or alignment. The best wall distance for TurtleBot3 Burger is 0.15 meters, while its maximum linear and angular velocities are 0.22 and 2.84 rad/s, respectively.

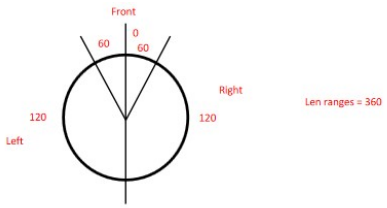

Figure 2.5: The three Zones

2.7 Simulation and Real-World Implementation

First, the FSM and wall-following behavior are designed and evaluated in a virtual environment using Unity Hub, a software simulation tool. Before deployment, this stage makes that the control logic is reliable and working. After the simulation is successful, TurtleBot3 receives the FSM logic through the Robot Operating System (ROS). The robot has all the ROS nodes it needs to process sensor data and carry out control orders. The FSM-based high-level control is proven in both simulated and real-world contexts through iterative testing and development, guaranteeing the robot can follow walls and explore places on its own.

The combination of simulation in Unity Hub and hardware testing on the TurtleBot3 robot provided a comprehensive approach to developing and validating the RANSAC high-level control system. By iteratively refining the algorithm in a simulated environment and then rigorously testing it in real-world conditions, I ensured the system's robustness and reliability. The use of inlier points for wall

Figure 2.6: TurtleBot3 on testing

detection and navigation decisions proved effective, enabling the TurtleBot3 to maintain a consistent and appropriate distance from walls while navigating efficiently. This methodology not only demonstrated the feasibility of the RANSAC-based control system but also highlighted its practical applicability in real-world robotic navigation tasks.

Chapter 3

Analysis and Results

The high-level control mechanism based on RANSAC is being effectively used by TurtleBot3 to follow walls. To find and fit lines that represent the walls in the robot's surroundings, laser scan data points are gathered, examined, and the RANSAC algorithm is used. These identified lines serve as the robot's movement guides, enabling it to travel successfully by keeping a constant distance from the walls. Through the use of real-time data processing and decision-making, the TurtleBot3 is accurately following the walls, as demonstrated by the visual plots that display the detected lines in different colors for the front, left, and right areas, as well as the robot's route.

The following terminal output demonstrates the effectiveness of the $RANSAC - based$ high-level control system in guiding the TurtleBot3 through a wall-following task. By continuously detecting walls, aligning itself, and following the wall, the robot showcases robust navigation capabilities. The detailed log entries provide insight into the $real - time\ decision - making$ process of the FSM and the RANSAC algorithm, highlighting the system's responsiveness to environmental changes and obstructions.

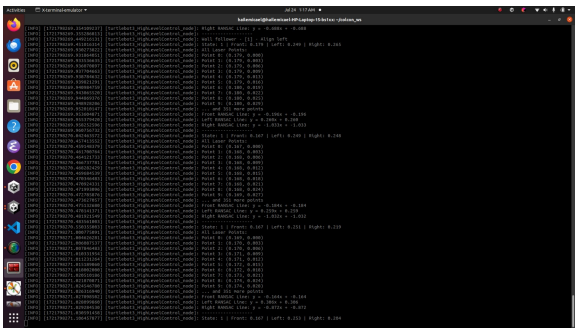

Figure 3.1: The output from laser scan in each region

It can keep a constant distance from barriers thanks to this system's processing of laser scan data. Plots that display the robot's course and the walls it has identified graphically validate this, demonstrating how precisely and instantly the trajectory of the robot can be adjusted to follow the walls. This illustrates how RANSAC may be effectively used for real-time robotic navigation.States that

have changed and the areas where lines have been discovered by the RANSAC algorithm are shown in the terminal output logs. The logs offer comprehensive insights into the robot's assessment of its surroundings. The log shows the fitted line parameters and the locations of the observed points when RANSAC detects a line. Understanding how the robot perceives its environment and modifies its behavior is much easier with the use of this real-time data.

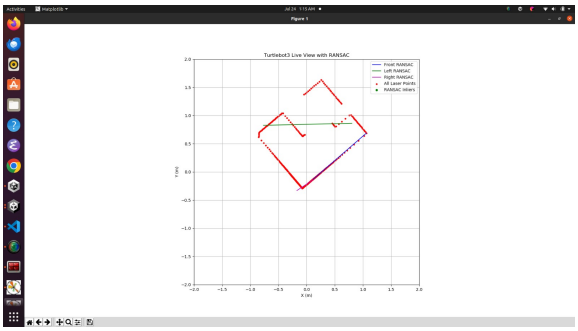

Figure 3.2: The Laser Scan data points

The robot changes states according to the lines that the RANSAC algorithm finds. For instance, the robot searches for walls while it is in state 0 (find the wall), which implies that no distinct line has been spotted. Depending on the direction and position of the observed wall, the status changes to either 1 (align left) or 2 (follow the wall) when a line is detected. The robot can effectively travel and follow barriers because to these state changes that guarantee the robot's ability to dynamically adapt to its surroundings.

Chapter 4

Discussion

The RANSAC algorithm demonstrates great accuracy in detecting and maintaining an acceptable distance from the wall, accurately detecting the wall and the three areas. As it reaches the threshold, the robot always turns to the left, effectively locating and following the wall. Its robust and reliable wall-following performance is ensured by its exceptional ability to manage uneven wall surfaces and deal curves.

To guarantee constant and effective navigation, these state changes are made to be as seamless as possible. However, delays or inaccurate line or state change detection might affect the system's performance. The robot may momentarily lose sight of the wall if the state change is not rapid, which might cause oscillations or sudden movements. Environmental complexity, algorithmic processing time, and sensor noise can all cause these kinds of delays. In spite of these possible problems, the robot performs admirably overall, following and aligning with the wall and managing corners and uneven terrain with impressive accuracy. The state transition logic's responsiveness determines the overall performance, which in turn determines how smoothly and consistently the wall-following behavior operates.

It has been demonstrated that the RANSAC algorithm is very resilient in handling noise and anomalies in the laser scan data, which is necessary for precise wall recognition and tracking. RANSAC efficiently removes noise and unnecessary data by concentrating on finding the most consistent data points that meet the predicted line models, enabling the TurtleBot3 to keep a trustworthy understanding of its surroundings. This feature guarantees that even in the presence of high sensor noise or abnormalities in the wall surfaces, the robot can recognize and follow walls with accuracy. The total stability and efficacy of the robot's navigation system is enhanced by RANSAC's effectiveness in real-time situations, which show its adaptability for dynamic and unpredictable settings.

Chapter 5

Conclusion

The TurtleBot3's RANSAC-based high-level control system has shown to be highly effective in simulated environments for wall-following tasks. The durability and dependability of the RANSAC algorithm in handling noisy and irregular laser scan data is demonstrated by the robot's ability to recognize barriers and navigate around them with accuracy. The method maintains accurate wall identification and the proper distance from the walls by concentrating on the most consistent data points and eliminating noise and outliers. In situations where surface imperfections and sensor noise are common, this skill is essential for autonomous navigation.

A key component of the TurtleBot3's overall navigation behavior is the state transition logic built into its control system. To keep a steady and effective navigation course, the robot seamlessly switches between locating the wall, aligning with it, and following it. The state changes are sensitive to the observed lines, as seen by the terminal output logs, indicating that the robot can swiftly adjust to changing environmental conditions. Even if there may be a few small delays or errors in state transitions, the robot is still able to function normally since the system immediately corrects itself and modifies its course.

The RANSAC algorithm's strength in handling noise and outliers in laser scan data is demonstrated by evaluating its resilience. An important benefit for autonomous robotic systems is that the algorithm works well in dynamic and unpredictable contexts, as demonstrated by its efficacy in real-time scenarios. Still, there's potential for improvement. The algorithm's performance may be enhanced by including adaptive thresholding and a multi-model hypothesis method, which would enable it to recognize many wall segments at once and dynamically adapt to changing noise levels. With these upgrades, the system's precision and dependability would increase, guaranteeing even more resilience in a variety of difficult situations.

Overall, it has been successful to achieve precise and dependable wall-following behavior with the TurtleBot3 with the integration of the RANSAC-based high-level control system. The potential for autonomous robots to navigate complicated surroundings effectively is demonstrated by the system's ability to tolerate noise and outliers, smoothly transition between states, and maintain the proper distance from barriers. The TurtleBot3's potential as a dependable and strong autonomous navigation system will probably be cemented by future improvements to the state transition logic

and RANSAC algorithm, which will probably further optimize performance. More complex and adaptable autonomous systems will be possible thanks to this effort, which shows that employing complicated algorithms like RANSAC in practical robotic applications is both feasible and effective.

5.1 Future Work

To enhance the performance of high-level control on TurtleBot3, implementing more sophisticated control algorithms, such as Proportional-Integral-Derivative (PID) controllers, represents a promising avenue for future work. PID control algorithms are known for their robustness and versatility in tuning system responses to achieve desired performance. By integrating PID controllers into our control system, we can improve the precision of navigation and obstacle avoidance tasks, leading to smoother and more reliable autonomous movements. Future research will focus on designing and fine-tuning PID parameters specifically tailored to the TurtleBot3's dynamic environment to optimize its control responses.

Incorporating Simultaneous Localization and Mapping (SLAM) technology into the TurtleBot3 system offers significant advancements in mapping and localization capabilities. SLAM algorithms will enable the TurtleBot3 to build comprehensive maps of its surroundings while simultaneously tracking its position within those maps. This integration will enhance the robot's ability to navigate complex and dynamic environments more effectively. Future work will involve selecting and implementing an appropriate SLAM technique, evaluating its performance in various scenarios, and integrating it seamlessly with the existing control and navigation systems.

Extending the visualization of the TurtleBot3's environment with 3D mapping and path planning displays represents another critical area for future development. Advanced 3D visualization can provide a more intuitive understanding of the robot's operational environment and improve real-time decision-making. By developing and integrating 3D mapping and path planning visualizations, users will gain enhanced insights into the robot's navigation strategies and environmental interactions. Future research will focus on implementing these visualization tools, ensuring they integrate effectively with the robot's control systems, and refining them based on user feedback and performance data.

Chapter 6

Appendix

.1 Python Code: high_Level_Control.py

```python
import rclpy  # Import the rclpy library to work with ROS 2.
from rclpy.node import Node  # Import the Node class from rclpy to create a ROS 2 node.
from geometry_msgs.msg import Twist  # Import the Twist message from geometry_msgs to publish velocity commands.
from sensor_msgs.msg import LaserScan  # Import the LaserScan message from sensor_msgs to subscribe to laser scan data.
import numpy as np  # Import the numpy library for numerical operations.
import matplotlib  # Import matplotlib for plotting.
matplotlib.use('TkAgg')  # Set the backend for matplotlib to 'TkAgg' for interactive plots.
import matplotlib.pyplot as plt  # Import pyplot from matplotlib for plotting.
from matplotlib.animation import FuncAnimation  # Import FuncAnimation from matplotlib for animated plots.
from sklearn.linear_model import RANSACRegressor  # Import RANSACRegressor from sklearn for robust line fitting.
import time  # Import time for handling time-related tasks.

plt.ion()  # Turn on interactive mode for matplotlib to allow dynamic updates of plots.


class Turtlebot3HighLevelControl(Node):

    """
    Turtlebot3 High-Level Control System
    ------------------------------------
    This ROS 2 node is designed for controlling a Turtlebot3 robot to perform wall-following behavior.
    Developed by Hailemicael Lulseged Yimer and Fitsum Sereke Tedla, this project integrates several advanced libraries and techniques to achieve autonomous navigation.

    Key Features:
    1. **ROS 2 Integration**: Utilizes ROS 2 for node creation, data publishing, and subscribing to sensor data.
```

```
    2. **Sensor Data Handling**: Receives and processes LaserScan data to detect and analyze obstacles around the robot.
    3. **Finite State Machine (FSM)**: Implements an FSM to handle different states: finding the wall, aligning with the wall, and following the wall.
    4. **RANSAC Algorithm**: Applies the RANSAC algorithm to fit lines to the laser scan data, which helps in detecting and tracking walls.
    5. **Real-time Visualization**: Uses Matplotlib for plotting and visualizing the laser scan data and the fitted lines, providing real-time feedback of the robot's environment.
    6. **Dynamic Control**: Adjusts the robot's movement based on the state of the FSM and the detected wall distance, ensuring consistent navigation and obstacle avoidance.

    Functionality:
    - **Initialization**: Sets up publishers, subscribers, and timers. Initializes plotting configurations.
    - **Laser Data Processing**: Converts laser scan data from polar to Cartesian coordinates, filters valid data, and performs RANSAC line fitting.
    - **Control Logic**: Based on the FSM state and sensor data, decides the robot's movement actions (e.g., move forward, turn left, follow the wall).
    - **Real-time Updates**: Continuously updates the plot with new sensor data and RANSAC line fittings.
    - **Error Handling**: Logs errors and warnings related to data processing and RANSAC fitting.
    - **State Transitions**: Changes the robot's state based on sensor readings and predefined conditions to adapt to different navigation scenarios.
    """

    def __init__(self):
        super().__init__('turtlebot3_HighLevelControl_node')  # Initialize the node with the name 'turtlebot3_HighLevelControl_node'.

        # Create a publisher to publish Twist messages to the '/cmd_vel' topic with a queue size of 1.
        self.publisher_ = self.create_publisher(Twist, '/cmd_vel', 1)
        # Create a subscriber to the '/scan' topic to receive LaserScan messages and call the laser_callback method.
        self.subscription = self.create_subscription(LaserScan, '/scan', self.laser_callback, rclpy.qos.qos_profile_sensor_data)

        self.state_ = 0  # Initialize the state of the finite state machine (FSM) to 0 (Find wall).
        self.state_dict_ = {  # Dictionary mapping state numbers to state descriptions.
            0: 'Find wall',
            1: 'Align left',
            2: 'Follow the wall'
        }

        self.msg = Twist()  # Initialize the Twist message for velocity commands.
        self.th = 0.15  # Set the distance threshold to the wall.

        timer_period = 0.1  # Set the timer period to 0.1 seconds.
        # Create a timer to call the control_loop method at regular intervals defined by timer_period.
        self.timer = self.create_timer(timer_period, self.control_loop)

        self.setup_plot()  # Initialize the plotting setup.
```

```
64
65          # Add a timer to update the plot every 0.5 seconds.
66          self.plot_timer = self.create_timer(0.5, self.update_plot)
67
68          # Initialize data structures
69          self.all_laser_points = []  # Initialize a list to store all laser points.
70          self.plot_data = {  # Initialize a dictionary to store data for plotting.
71              'x': [], 'y': [],
72              'front_fit_x': [], 'front_fit_y': [],
73              'left_fit_x': [], 'left_fit_y': [],
74              'right_fit_x': [], 'right_fit_y': []
75          }
76          self.regions = {'front': 10, 'left': 10, 'right': 10}  # Initialize the
                distances to the walls in different regions.
77
78      def setup_plot(self):
79          self.fig, self.ax = plt.subplots()  # Create a new figure and axis for
                plotting.
80          self.ax.set_xlim(-2, 2)  # Set the x-axis limits.
81          self.ax.set_ylim(-2, 2)  # Set the y-axis limits.
82          self.ax.set_aspect('equal')  # Set the aspect ratio to be equal.
83          self.ax.grid(True)  # Enable grid on the plot.
84          # Initialize scatter plots and lines for plotting.
85          self.scatter = self.ax.scatter([], [], c='r', s=10, label='All Laser Points'
                )
86          self.inlier_scatter = self.ax.scatter([], [], c='g', s=20, label='RANSAC
                Inliers')
87          self.line_front, = self.ax.plot([], [], 'b-', label='Front RANSAC')
88          self.line_left, = self.ax.plot([], [], 'g-', label='Left RANSAC')
89          self.line_right, = self.ax.plot([], [], 'm-', label='Right RANSAC')
90          self.ax.set_title('Turtlebot3 Live View with RANSAC')  # Set the title of
                the plot.
91          self.ax.set_xlabel('X (m)')  # Set the x-axis label.
92          self.ax.set_ylabel('Y (m)')  # Set the y-axis label.
93          self.ax.legend()  # Add a legend to the plot.
94
95
96      def update_plot(self):
97          try:
98              if len(self.all_laser_points) > 0:  # If there are laser points
                    available,
99                  x_data, y_data = zip(*self.all_laser_points)  # Unzip the laser
                        points into x and y data.
100                 self.scatter.set_offsets(np.column_stack((x_data, y_data)))  #
                        Update the scatter plot with new data.
101             else:
102                 self.scatter.set_offsets(np.empty((0, 2)))  # Clear the scatter plot
                        if no laser points are available.
103
104             # Update the RANSAC lines for each region.
105             for region in ['front', 'left', 'right']:
106                 if len(self.plot_data[f'{region}_fit_x']) > 0:
107                     getattr(self, f'line_{region}').set_data(self.plot_data[f'{
                            region}_fit_x'], self.plot_data[f'{region}_fit_y'])
108                 else:
109                     getattr(self, f'line_{region}').set_data([], [])
110
111             self.fig.canvas.draw()  # Redraw the figure.
```

```
112             self.fig.canvas.flush_events()  # Process the GUI events.
113
114             self.display_terminal_output()  # Display the terminal output.
115
116         except Exception as e:
117             self.get_logger().error(f'Error in update_plot: {str(e)}')  # Log any
                    errors that occur.
118
119     def control_loop(self):
120         # Perform actions based on the current state of the FSM.
121         if self.state_ == 0:
122             self.find_wall()
123         elif self.state_ == 1:
124             self.align_left()
125         elif self.state_ == 2:
126             self.follow_the_wall()
127         else:
128             self.get_logger().error('Unknown state!')  # Log an error if the state
                    is unknown.
129
130         self.publisher_.publish(self.msg)  # Publish the velocity command.
131
132     def laser_callback(self, msg):
133         ranges = np.array(msg.ranges)  # Convert the laser scan ranges to a numpy
                array.
134         angles = np.array([msg.angle_min + i * msg.angle_increment for i in range(
                len(ranges))])  # Calculate the angles for each range.
135
136         valid_indices = (ranges > msg.range_min) & (ranges < msg.range_max)  #
                Filter out invalid ranges.
137         valid_ranges = ranges[valid_indices]  # Get the valid ranges.
138         valid_angles = angles[valid_indices]  # Get the valid angles.
139
140         x = valid_ranges * np.cos(valid_angles)  # Convert polar coordinates to
                Cartesian coordinates (x).
141         y = valid_ranges * np.sin(valid_angles)  # Convert polar coordinates to
                Cartesian coordinates (y).
142
143         self.all_laser_points = list(zip(x, y))  # Store all valid laser points.
144
145         data = np.column_stack((x, y))  # Prepare data for RANSAC.
146
147         front_angle_half = 50 * np.pi / 180  # Convert 50 degrees to radians for the
                 front region.
148         side_angle = 85 * np.pi / 180  # Convert 85 degrees to radians for the side
                regions.
149
150         angles_rad = np.arctan2(y, x)  # Calculate angles in radians.
151
152         # Compute indices for each region.
153         front_indices = np.abs(angles_rad) < front_angle_half
154         left_indices = (angles_rad >= front_angle_half) & (angles_rad <
                front_angle_half + side_angle)
155         right_indices = (angles_rad <= -front_angle_half) & (angles_rad > -
                front_angle_half - side_angle)
156
157         # Fit RANSAC lines for each region.
```

```
158         for region, indices in zip(['front', 'left', 'right'], [front_indices,
                left_indices, right_indices]):
159             if np.sum(indices) > 2:  # Ensure there are at least 3 points for RANSAC
                     fitting.
160                 region_x = x[indices]  # Extract x coordinates for the current
                        region.
161                 region_y = y[indices]  # Extract y coordinates for the current
                        region.
162                 try:
163                     ransac = RANSACRegressor()  # Initialize the RANSAC regressor.
164                     ransac.fit(region_x.reshape(-1, 1), region_y)  # Fit the RANSAC
                            model to the data.
165                     x_fit = np.linspace(min(region_x), max(region_x), 100)  #
                            Generate x values for the fit line.
166                     y_fit = ransac.predict(x_fit.reshape(-1, 1))  # Predict y values
                             for the fit line.
167                     self.plot_data[f'{region}_fit_x'] = x_fit  # Store x values for
                            plotting.
168                     self.plot_data[f'{region}_fit_y'] = y_fit  # Store y values for
                            plotting.
169                 except Exception as e:
170                     self.get_logger().warn(f'RANSAC fitting failed for {region}: {
                            str(e)}')  # Log warning if RANSAC fails.
171                     self.plot_data[f'{region}_fit_x'] = []  # Clear fit data on
                            failure.
172                     self.plot_data[f'{region}_fit_y'] = []  # Clear fit data on
                            failure.
173
174                 # Update region distance.
175                 self.regions[region] = np.min(np.linalg.norm(np.column_stack((
                        region_x, region_y)), axis=1)) if len(region_x) > 0 else 10  #
                        Compute minimum distance to the wall for the current region or
                        set default distance.
176
177         self.take_action()  # Decide the action based on the updated region
                distances.
178
179     def take_action(self):
180         """
181         Determines the appropriate action for the robot based on the current state
                and
182         the distances to obstacles in the front, left, and right regions.
183         """
184         # If the robot is in state 1 or 2, and the distances to the right and front
185         # are greater than the threshold, change state to 0 (Find wall).
186         if (self.state_ == 1 or self.state_ == 2) and self.regions['right'] > self.
                th and self.regions['front'] > self.th:
187             self.change_state(0)  #  Move to find the wall.
188
189         # If the robot is in state 0 or 2, and the distance to the front is less
                than
190         # the threshold but the distances to the left and right are greater than the
                 threshold,
191         # change state to 1 (Align left).
192         elif (self.state_ == 0 or self.state_ == 2) and self.regions['front'] < self
                .th and self.regions['left'] > self.th and self.regions['right'] > self.
                th:
193             self.change_state(1)  #  Align left with the wall.
```

```
194
195         # If the robot is in state 0 or 2, and the distance to the front is less
                than
196         # the threshold, the distance to the left is greater than the threshold, and
                 the
197         # distance to the right is less than the threshold, change state to 1 (Align
                 left).
198         elif (self.state_ == 0 or self.state_ == 2) and self.regions['front'] < self
                .th and self.regions['left'] > self.th and self.regions['right'] < self.
                th:
199             self.change_state(1)  # Align left with the wall.
200
201         # If the robot is in state 0 or 2, and the distance to the front is less
                than
202         # the threshold, the distances to the left and right are also less than the
                threshold,
203         # change state to 1 (Align left).
204         elif (self.state_ == 0 or self.state_ == 2) and self.regions['front'] < self
                .th and self.regions['left'] < self.th and self.regions['right'] < self.
                th:
205             self.change_state(1)  # Align left with the wall.
206
207         # If the robot is in state 0 or 2, and the distance to the front is less
                than
208         # the threshold, the distance to the left is less than the threshold, and
                the
209         # distance to the right is greater than the threshold, change state to 1 (
                Align left).
210         elif (self.state_ == 0 or self.state_ == 2) and self.regions['front'] < self
                .th and self.regions['left'] < self.th and self.regions['right'] > self.
                th:
211             self.change_state(1)  #Align left with the wall.
212
213         # If the robot is in state 0 or 1, and the distance to the front is greater
                than
214         # the threshold, the distance to the left is greater than the threshold, and
                 the
215         # distance to the right is less than the threshold, change state to 2 (
                Follow wall).
216         elif (self.state_ == 0 or self.state_ == 1) and self.regions['front'] > self
                .th and self.regions['left'] > self.th and self.regions['right'] < self.
                th:
217             self.change_state(2)  # Follow the wall.
218
219         # Log the current state and distances to the front, left, and right.
220         self.get_logger().info(f"State: {self.state_} | Front: {self.regions['front
                ']:.3f} | Left: {self.regions['left']:.3f} | Right: {self.regions['right
                ']:.3f}")
221
222 def change_state(self, state):
223     """
224     Changes the state of the robot and logs the transition.
225     """
226     if state != self.state_:
227         # Log the transition to the new state.
228         self.get_logger().info(f'Wall follower - [{state}] - {self.state_dict_[state
                ]}')
229         self.state_ = state  # Update the state.
```

```
230
231 def find_wall(self):
232     """
233     Set the robot to move forward to find the wall.
234     """
235     self.msg.linear.x = 0.021  # Move forward with a small speed.
236     self.msg.angular.z = 0.0  # No turning, just move straight.
237
238 def align_left(self):
239     """
240     Set the robot to stop moving forward and turn left to align with the wall.
241     """
242     self.msg.linear.x = 0.0  # Stop moving forward.
243     self.msg.angular.z = 0.21  # Turn left to align with the wall.
244
245 def follow_the_wall(self):
246     """
247     Set the robot to move forward while following the wall.
248     """
249     self.msg.linear.x = 0.021  # Move forward with a small speed.
250     self.msg.angular.z = 0.0  # Move straight, no turning.
251
252 def stop_robot(self):
253     """
254     Stop the robot's movement by setting linear and angular velocities to zero.
255     """
256     self.msg.linear.x = 0.0  # Stop moving forward by setting the linear velocity to
          0.
257     self.msg.angular.z = 0.0  # Stop turning by setting the angular velocity to 0.
258     self.publisher_.publish(self.msg)  # Publish the stop command to the '/cmd_vel'
          topic to execute the stop.
259
260 def display_terminal_output(self):
261     """
262     Display diagnostic information about the robot's laser scan data and RANSAC line
          fitting.
263     """
264     # Log the first 10 laser points to the terminal.
265     self.get_logger().info("All Laser Points:")
266     for i, point in enumerate(self.all_laser_points[:10]):  # Display first 10
          points from the laser scan data.
267         self.get_logger().info(f"Point {i}: ({point[0]:.3f}, {point[1]:.3f})")
268
269     # If there are more than 10 points, log how many more points there are beyond
          the first 10.
270     if len(self.all_laser_points) > 10:
271         self.get_logger().info(f"... and {len(self.all_laser_points) - 10} more
              points")
272
273     # Log RANSAC line fitting results for each region (front, left, right).
274     for region in ['front', 'left', 'right']:
275         if len(self.plot_data[f'{region}_fit_x']) > 0:  # Check if there is RANSAC
              line data for the region.
276             # Log the equation of the RANSAC line fitted to the data for the region.
277             self.get_logger().info(f"{region.capitalize()} RANSAC Line: y = {self.
                  plot_data[f'{region}_fit_y'][0]:.3f}x + {self.plot_data[f'{region}
                  _fit_y'][0]:.3f}")
278         else:
```

```
            # Log a message if no RANSAC line was fitted for the region.
            self.get_logger().info(f"No RANSAC line for {region}")

    # Log a separator line for clarity.
    self.get_logger().info("--------------------")

def main(args=None):
    """
    Main function to initialize and run the Turtlebot3 High-Level Control Node.
    """
    # Initialize the ROS client library
    rclpy.init(args=args)  # Initialize ROS client library for Python, passing command-line arguments if any.

    # Create an instance of the Turtlebot3HighLevelControl node
    turtlebot3_HighLevelControl_node = Turtlebot3HighLevelControl()

    try:
        # Enter a loop where the node will keep running and processing callbacks
        rclpy.spin(turtlebot3_HighLevelControl_node)  # This keeps the node active and responsive to callbacks until interrupted.
    except KeyboardInterrupt:
        # Handle a keyboard interrupt (Ctrl+C) gracefully
        turtlebot3_HighLevelControl_node.stop_robot()  # Stop the robot if interrupted by the user.
        print("Node terminated by user!")  # Inform the user that the node was terminated.
        time.sleep(0.5)  # Pause briefly to ensure all commands are processed before exiting.

    # Clean up after the node has been stopped
    turtlebot3_HighLevelControl_node.destroy_node()  # Cleanly destroy the node, releasing any resources.
    rclpy.shutdown()  # Shut down the ROS client library, terminating all communications.
    plt.close()  # Close the Matplotlib plot when the node shuts down to prevent hanging or errors.

if __name__ == '__main__':
    """
    Entry point for the script. Calls the main function to start the ROS node.
    """
    main()  # Execute the main function if this script is run directly.

# ==============================
# Code Summary and Explanation
# ==============================

# This script defines a ROS 2 node for high-level control of a Turtlebot3 robot. The robot is designed to perform wall-following behavior
# using laser scan data. Below is a summary of the key components and their functions:

# 1. **Imports:**
#    - `rclpy`: The ROS 2 client library for Python, used for creating nodes and managing ROS communication.
#    - `Node`, `Twist`, `LaserScan`: ROS 2 classes for creating nodes, publishing velocity commands, and subscribing to laser scan data.
```

```
#     - `numpy`: Library for numerical operations and data manipulation.
#     - `matplotlib`, `FuncAnimation`: Libraries for real-time plotting and animation of data.
#     - `RANSACRegressor`: For robust line fitting to detect walls in laser scan data.
#     - `time`: For handling time-related tasks.

# 2. **Class `Turtlebot3HighLevelControl`:**
#     - **Initialization (`__init__` method):** Sets up publishers, subscribers, timers, and initializes plotting.
#     - **Laser Data Processing (`laser_callback` method):** Converts laser scan data from polar to Cartesian coordinates, performs RANSAC line fitting, and updates region distances.
#     - **Control Logic (`control_loop` method):** Determines robot actions based on the current FSM state.
#     - **Action Handling (`take_action` method):** Decides and executes robot actions based on region distances and FSM state.
#     - **Plotting (`setup_plot` and `update_plot` methods):** Initializes and updates real-time plots of laser scan data and RANSAC lines.
#     - **State Management (`change_state` method):** Handles transitions between different robot states.

# 3. **Main Function (`main` method):**
#     - Initializes the ROS client library and creates an instance of the Turtlebot3HighLevelControl node.
#     - Runs the node in a loop to process callbacks.
#     - Handles keyboard interrupts to stop the robot and perform cleanup.

# 4. **Script Execution:**
#     - The `if __name__ == '__main__':` block ensures the `main` function is executed if the script is run directly.
```

Bibliography

[1] Adam Caccavale and Mac Schwager. Trust but verify: A distributed algorithm for multi-robot wireframe exploration and mapping. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3294–3301. IEEE, 2019.

[2] Sudarshan Srinvasa Rangan, Ion Barosan, and Mr Pasquale Van Heumen. Indoor localization of mobile robots using magnetic fields and lidar.

[3] Xiaolei Sun, Yu Zhang, and Jing Chen. High-level smart decision making of a robot based on ontology in a search and rescue scenario. *Future Internet*, 11(11):230, 2019.

[4] Paolo Vanella. *Implementation of ROS-based Multi-Agent SLAM Centralized and Decentralized Approaches*. PhD thesis, Politecnico di Torino, 2023.