DEVELOPMENT OF NAVIGATION SYSTEM FOR TURTLEBOT 3 USING SLAM METHOD

JOSHUA BRYAN CHEAH WERN XIEN

BACHELOR OF MECHATRONICS ENGINEERING WITH HONOURS
UNIVERSITI TEKNIKAL MALAYSIA MELAKA

2024

DEVELOPMENT OF NAVIGATION SYSTEM FOR TURTLEBOT 3 USING SLAM METHOD

JOSHUA BRYAN CHEAH WERN XIEN

A report submitted
in partial fulfilment of the requirements for the degree of
Bachelor of Mechatronics Engineering with Honours

Faculty of Electrical Technology and Engineering

UNIVERSITI TEKNIKAL MALAYSIA MELAKA

2024

DECLARATION

I declare that this thesis entitled “DEVELOPMENT OF NAVIGATION SYSTEM FOR TURTLEBOT 3 USING SLAM METHOD” is the result of my own research except as cited in the references. The thesis has not been accepted for any degree and is not concurrently submitted in the candidature of any other degree.

Signature	:	
Name	:	JOSHUA BRYAN CHEAH WERN XIEN
Date	:	20 JUNE 2024

APPROVAL

I hereby declare that I have checked this report entitled "DEVELOPMENT OF NAVIGATION SYSTEM FOR TURTLEBOT 3 USING SLAM METHOD", and in my opinion, this thesis fulfils the partial requirement to be awarded the degree of Bachelor of Mechatronics Engineering with Honours

Signature	:	
Supervisor Name	:	DR. MOHD KHAIRI BIN MOHAMED NOR
Date	:	21 JUNE 2024

DEDICATIONS

To my beloved mother, Liew Poh Yee, and father, Cheah Yew Chin, whose support has been a constant source of strength throughout this Final Year Project.

To my dearest supervisor, Dr. Mohd Khairi Bin Mohamed Nor, whose unwavering guidance has played a crucial role in shaping the success of this project.

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to my main project supervisor, Dr. Mohd Khairi Bin Mohamed Nor, for encouragement, guidance, advices and motivation. Without his continued support and interest, this project would not have been same as presented here.

I extend my sincere appreciation to Universiti Teknikal Malaysia Melaka (UTeM) for providing an exceptional academic environment that significantly contributed to the success of my Final Year Project. UTeM's commitment to excellence, innovative approach, and supportive atmosphere have been instrumental in shaping my academic journey and fostering a passion for continuous learning. I am truly grateful for the resources and dedicated faculty, which have played a crucial role in the development of my project and academic growth.

My fellow university coursemates should also be recognized for their support. My sincere appreciation also extends to all my family members and others who have provided assistance on various occasions. Their views and support are useful indeed. Unfortunately, it is not possible to list all of them in this limited space.

ABSTRACT

F1TENTH is a popular open-source platform among university students which organizes autonomous mobile robot competitions. The main navigation method used in F1TENTH competition is SLAM, also known as Simultaneous Localization And Mapping. SLAM, which is a technology developed for over 30 years, is an algorithm that builds a map of the surrounding environment of the robot through mapping process and at the same time estimating the robot’s position on the map while it is moving. Nowadays, SLAM is used in many applications such as autonomous vehicles and drones. TurtleBot 3 is an autonomous mobile robot which shares the same technology used in F1TENTH. This project focuses on developing a high-speed navigation system for TurtleBot 3 using the SLAM method in the Robot Operating System (ROS) framework. The main problem tackled is the development of a high-speed navigation system using SLAM, focusing on accurately mapping an indoor racetrack, selecting suitable path planning algorithms, and analyzing the system's performance in terms of speed and accuracy. The objective of this project is to implement SLAM algorithm in mapping and to develop the TurtleBot 3’s navigation system followed by analyzing its performance in terms of speed and accuracy. Experiments were conducted in the virtual environment, using the TurtleBot 3 Burger model in Gazebo and Rviz within the ROS framework to validate the map and analyze performance of the navigation system under various conditions. Overall, this project successfully develops the navigation system for the TurtleBot 3 and analyzes the performance parameters, establishing a foundation for future applications and enhancements.

ABSTRAK

F1TENTH merupakan platform sumber terbuka yang popular di kalangan pelajar universiti yang menganjurkan kompetisi robot mudah alih. Kaedah navigasi utama yang digunakan dalam persaingan F1TENTH ialah SLAM, juga dikenali sebagai Lokalisasi dan Peta Simultan. SLAM, yang merupakan teknologi yang telah dibangunkan selama lebih 30 tahun, ialah algoritma yang membina peta persekitaran robot melalui proses peta dan pada masa yang sama menganggarkan kedudukan robot pada peta semasa ia bergerak. Hari ini, SLAM digunakan dalam banyak aplikasi seperti kenderaan otonom dan drone. TurtleBot 3 ialah robot mudah alih autonomi yang berkongsi teknologi yang sama yang digunakan dalam F1TENTH. Projek ini memberi tumpuan kepada pembangunan sistem navigasi kelajuan tinggi untuk TurtleBot 3 menggunakan kaedah SLAM dalam kerangka Robot Operating System (ROS). Masalah utama yang ditangani ialah pembangunan sistem navigasi kelajuan tinggi menggunakan SLAM, memberi tumpuan kepada memaparkan laluan perlumbaan dalaman dengan tepat, memilih algoritma perancangan laluan yang sesuai, dan menganalisis prestasi sistem dalam hal kelajuan dan ketepatan. Objektif projek ini ialah untuk melaksanakan algoritma SLAM dalam peta dan untuk membangunkan sistem navigasi TurtleBot 3 yang diikuti dengan menganalisis prestasinya dalam hal kelajuan dan ketepatan. Eksperimen dijalankan dalam persekitaran maya, menggunakan model TurtleBot 3 Burger di Gazebo dan Rviz dalam rangka ROS untuk mengesahkan peta dan menganalisis prestasi sistem navigasi dalam pelbagai keadaan. Secara keseluruhan, projek ini berjaya membangunkan sistem navigasi untuk TurtleBot 3 dan menganalisis parameter prestasi, menubuhkan asas untuk aplikasi dan peningkatan masa depan.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

SLAM	-	Simultaneous Localization And Mapping
GNSS	-	Global Navigation Satellite System
ROS	-	Robot Operating System
AV	-	Autonomous vehicle
AI	-	Artificial Intelligence
LiDAR	-	Light Detection and Ranging
F1	-	Formula One
IMU	-	Inertial measurement unit
SBC	-	Single Board Computer
OpenCR	-	Open-source Control Module
DQN	-	Deep Q-Network
RTOD	-	Real Time Object Detection
RRT	-	Rapidly-exploring Random Tree
PPA	-	Pure pursuit algorithm
RViz	-	ROS visualization
PID	-	Proportional – Integral – Derivative
RMSE	-	Root mean square error
LV	-	Linear velocity
LD	-	Lookahead distance
Kp	-	Angular velocity proportional gain
AT	-	Angle threshold
RF	-	Linear velocity reduction factor
Ki	-	Angular velocity integral gain
Kd	-	Angular velocity derivative gain
MPC	-	Model Predictive Control

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

1.1 Background

The autonomous mobile robot competition has gained popularity among university students worldwide in recent years. F1TENTH is one of the platforms that organizes these kinds of competitions where it combines and focuses on two main aspects, namely robotics and autonomous driving. The inspiration came from the Formula 1, famously known as F1, which focuses on creating and testing algorithms for autonomous navigation and control in compact and low-priced race cars. In this competition, participants, mainly students, are required to create a 1/10th-scale race car with an autonomous navigation system. The automobile must move as quickly as it can on the designated racing course autonomously. The racing course is designed with boundaries and elements like straight lanes, curves and obstacles to be avoided by the race cars. Participants compete with one another to display their race cars with the fastest speed, most agile and highest precision in navigation [1]. Figure 1.1 below shows one of the race cars used in F1TENTH competition.

Figure 1.1: F1TENTH race car [2]

One of the navigation methods used in F1TENTH competition is SLAM, also known as Simultaneous Localization And Mapping. SLAM is a technology that builds a map of the robot’s surroundings (mapping) through data from its sensors and simultaneously estimating its position on the map while moving (localization). The main goal of using SLAM is to achieve the autonomous behavior of the mobile robot. Even though the Global Navigation Satellite System (GNSS) is commonly used for navigation, which is capable of providing an exact location, it is not always reliable or available in dark and covered up places such as in caves and tunnels where it is badly impacted and unable to finish the positioning work [3]. SLAM technology has been analyzed and developed for over 30 years. Nowadays, SLAM is used extensively in various applications such as mobile robots, autonomous vehicles as well as drones [4]. Mobile robots use SLAM technology to recognize the house environment to perform house cleaning autonomously [5]. SLAM is also applied in autonomous vehicles to create a map of the surrounding area and estimate the position of moving vehicles in real time to navigate safely on the road [6]. Besides, drones utilize SLAM technology in agriculture operations such as automated irrigation system and crop observation [7].

Autonomous mobile robot such as the TurtleBot 3 have been developed for SLAM navigation. TurtleBot 3 is one of the models in the TurtleBot series. It is small, simple, versatile, easy to assemble using consumer goods that are readily available off the shelf and at the same time it provides advanced sensors at a notably reduced cost [8]. It is commonly used for education, research and also in motion planning strategies. It utilizes the open-source Robotic Operating System (ROS) framework and is programmable in programming languages such as MATLAB and Python. Its compact size preserves its functionality and performance while making it possible to acquire a highly competitive platform for a minimal investment. Hence, the TurtleBot 3 is ideal for SLAM applications in motion planning [9].

1.2 Motivation

The F1TENTH hosts regular annual competitions, such as the most recent F1TENTH Autonomous Racing Competition, taking place at the Intelligent Vehicles Symposium (IV) 2024 [10]. One of the most notable winners of the F1TENTH competition is the group of Penn Engineering students who won the 12th Annual F1TENTH Autonomous Grand Prix hosted in San Antonio, Texas in May 2023 [11]. The TurtleBot 3 is widely recognized as one of the most popular open-source robotic platforms, particularly valued for its educational and research applications. SLAM is one of the core technologies of TurtleBot3, alongside navigation and manipulation, making it suitable for a wide range of applications from research to educational purposes [12]. SLAM navigation allows the mobile robot to navigate in an unknown environment by learning and constructing the environment's map and simultaneously localizing its own position on the map created [13]. This is also known as autonomous navigation. F1Tenth race cars also utilize SLAM techniques to autonomously navigate and map their surroundings during racing competitions [14]. To integrate F1TENTH technology into the TurtleBot 3 represents a significant challenge, yet achieving this integration would mark a substantial accomplishment. With that being said, this project has motivated me to learn and develop an autonomous navigation system for TurtleBot 3, especially to be able to apply in a fast-paced competition like the F1TENTH. Through this project, I wanted to take this opportunity to find out how does the TurtleBot 3 navigate in high-speed condition with the application of SLAM technology. Besides, I also wanted to find out how fast and accurate the TurtleBot 3 can be during navigation. To be able to find out if the of TurtleBot 3 can meet the capabilities of the F1TENTH race cars, this further sparked my curiosity and interest in completing this project.

1.3 Problem statements

The F1TENTH competition requires race cars to navigate autonomously as accurately as possible on the designated indoor racetrack. The TurtleBot 3 is used in this project to develop its navigation system with the SLAM method. This project aims to integrate the advanced navigation capabilities of the F1TENTH racing series into the TurtleBot 3 platform. Hence, the problem statement of this project is about finding out the way to implement SLAM method in developing the navigation system of TurtleBot 3. The challenges include configuring and optimizing the SLAM algorithm to accurately map the racetrack. Next, the second problem statement is to determine the suitable algorithm to develop a high-speed navigation system for TurtleBot 3, at the same time considering the accuracy of the navigation. This involves evaluating and customizing path planning and control algorithms to ensure the TurtleBot 3 can navigate efficiently around the racetrack at a comparable speed to F1TENTH race cars. Lastly, the third problem statement of this project is about determining the appropriate ways to analyze and optimize the performance of the developed navigation system of the TurtleBot 3 in terms of lap time and trajectory accuracy. This includes fine-tuning the algorithm parameters of the navigation system for optimal performance.

1.4 Objectives

1. To create a map of the surrounding environment for TurtleBot 3 using SLAM method.

2. To develop an autonomous racing navigation system for TurtleBot 3 with the map created from SLAM method.

3. To analyze the performance of the autonomous racing navigation system of TurtleBot 3 in terms of lap time and trajectory accuracy.

1.5 Scopes

1. Ubuntu 16.04.7 LTS (Xenial Xerus) version is used as the operating system foundation for this project.

2. ROS Kinetic distribution is used as the primary framework and software for this project.

3. The TurtleBot 3 model used in the development of the navigation system is the Burger model.

4. The size of the virtual racetrack used in this project is relatively smaller than the F1TENTH racetrack.

5. The environment mapping method used is the SLAM method.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

In this chapter, the overview of the F1TENTH, the components and features of the TurtleBot, the Robot Operating System (ROS), the types of navigation systems and the types of Simultaneous Localization And Mapping (SLAM) algorithms are the main topics to be discussed and reviewed.

2.2 F1TENTH

The F1TENTH autonomous racing platform was initially introduced in 2015. It is an open-source evaluation environment for continuous control and reinforcement learning that makes it easier to train, test and assess autonomous systems. The F1TENTH platform offers a 1/10th-scale, low-cost hardware and multiple virtual environments, allowing for safe and quick experimentation of autonomous vehicle (AV) algorithms [15]. The standard framework for robotics systems applications, ROS, serves as the core for the F1TENTH platform. It holds many competitions and provides engaging learning environment for those who are enthusiastic in control, autonomous driving, and artificial intelligence, especially for students. The F1TENTH platform's main objective is autonomous driving, while control is still required. The majority of research efforts in the F1TENTH community have gone into developing solutions for driving algorithms, localization, and positioning. The F1TENTH platform uses SLAM and LiDAR techniques to replicate a realistic data collection module which is used for navigation. Furthermore, it is an important tool for research and development in the field of Artificial Intelligence (AI) and autonomous driving [16].

F1TENTH is an autonomous robotics competition inspired by the well-known F1 that involves 1/10th-scale race cars developed by each team through self-driving algorithm, competing in an autonomous racing task [17]. The competition focuses on optimising these algorithms for the race cars to autonomously navigate around a randomized racetrack in the shortest amount of time. The so-called ‘randomized’ racetrack is a specially constructed with boundaries and features including straight lanes, curves, as well as static and dynamic obstacles. The racetrack can either be indoor or outdoor. Figure 2.2 below shows an example of F1TENTH competition racetrack. Algorithms such as path planning, obstacle avoidance, vehicle control, and the optimisation of racing strategies are among the challenges to be focused by the participants. Therefore, the participants are required to program their race cars for them to navigate the race course autonomously while avoiding collisions. In this case, the SLAM alogrithm is commonly used to overcome these challenges [1].

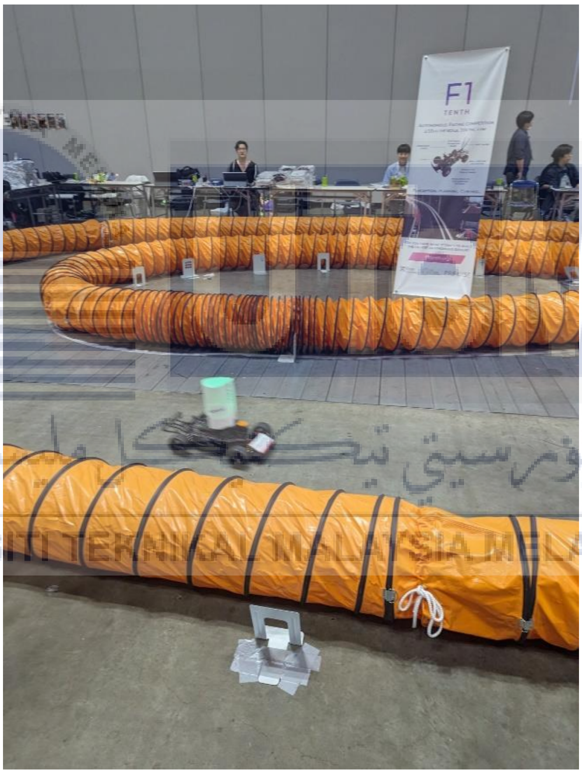


Figure 2.1: Example of F1TENTH competition racetrack [18]

The F1TENTH race car is in a 1/10th-scale, which is relatively small as compared to the size of a regular Formula One (F1) race car. These race cars are equipped with a range of sensors, including the inertial measurement unit (IMU), 2D scanning LiDAR, and camera. The F1TENTH race cars are able to sense the environment and make decisions through the data obtained from those sensors. The embedded AI computing device such as the NVIDIA Jetson TX2 as well as ROS are the default robot control software used to control the sensing and actuating components of the F1TENTH race car [19]. Figure 2.1 below shows the properties of F1TENTH race car.

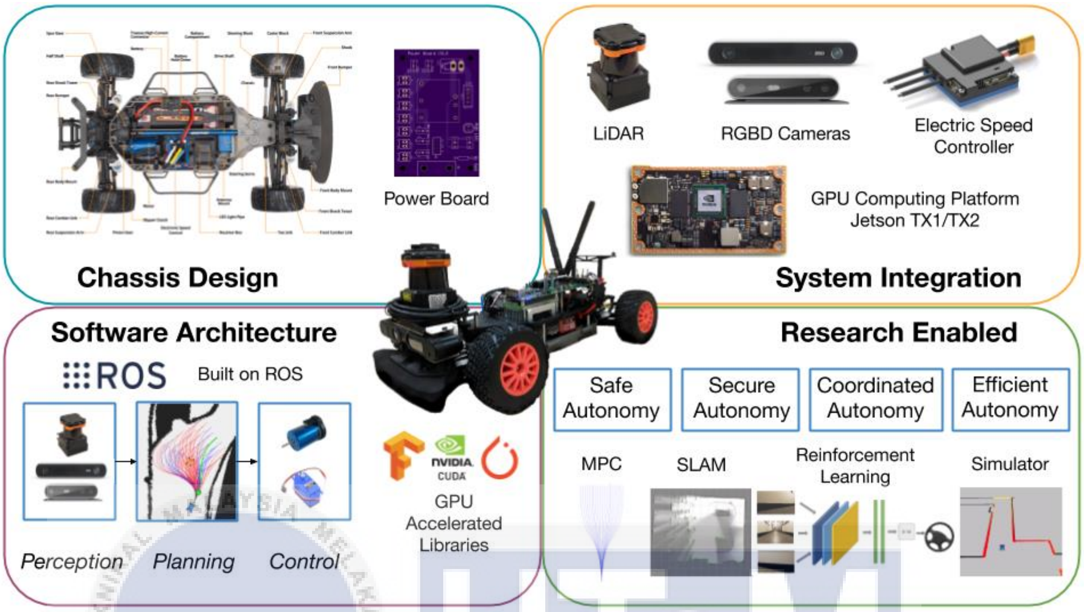


Figure 2.2: Properties of F1TENTH race car [20]

2.3 TurtleBot 3

TurtleBot 3 is a programmable, compact, low-cost mobile robot that can be used for hobby, education, research, and product prototyping. The objective of TurtleBot 3 is to provide expandability while drastically reducing the platform's size and cost without compromising its quality or usefulness. TurtleBot3 has a Single Board Computer (SBC) that is appropriate for reliable embedded systems, 360-degree distance sensors, Open-source Control Module (OpenCR) and 3D printing technology [21]. There are three variants of TurtleBot 3, namely TurtleBot 3 Burger, TurtleBot 3 Waffle and TurtleBot 3 Waffle Pi. The Waffle model can carry a heavier weight and moves ahead a little bit faster as compared to the Burger model. It is much bigger, includes an additional Pi camera sensor, and it is relatively more expensive [8]. Figure 2.6 shows the TurtleBot 3 variants.

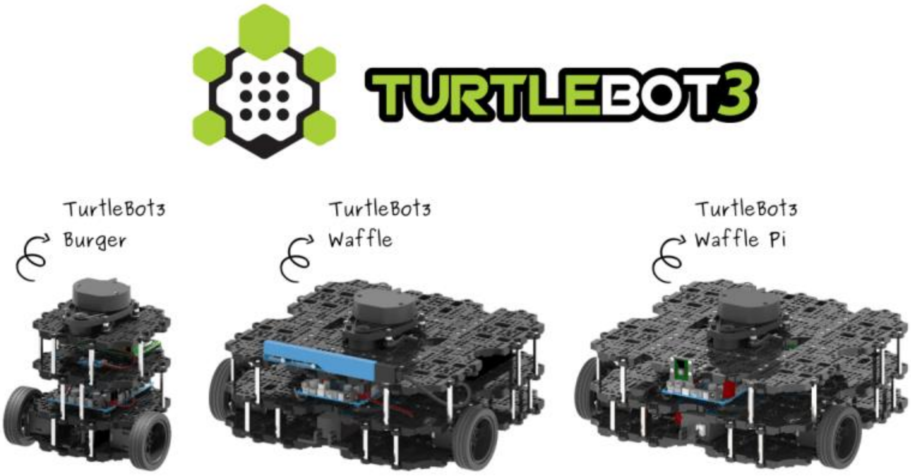


Figure 2.3: TurtleBot 3 variants [8]

Table 2.1: Specifications and features of different versions of TurtleBot

Version	References	Features
TurtleBot 1	[22]	- Cost-effective which is suitable for education and research. - Raw sensor data is accessible via open platform. - Compatible with ROS.
TurtleBot 2	[23]	- Cost-effective which is suitable for education and research. - Built for ROS. - Fully assembled and tested to be used anytime. - Wide database of tutorials that can be referred to.
TurtleBot 3	[12]	- Cost-effective which is suitable for education and research. - Compact size which is easy to carry. - Extensibility which is able to customize. - Modular actuator which makes it easy to assemble and maintain. - Open-source software which is fully open to download, modify and share.

		- Strong sensors for better detection.
TurtleBot 4	[24][25]	- Physically more solid and reliable. - Runs ROS 2 software which is faster and more reliable. - Open-source software library modules provide higher accuracy and reliability. - Improved battery with fast charging and prolonged battery life. - LiDAR provides better object recognition and higher accuracy with longer range. - More connectivity alternatives. - More diverse range of sensors to gather more accurate information around its surroundings.

Based on Table 2.1 above, it can be seen that the TurtleBot 3 has many advantages over the other variants. Firstly, its compact size and modular design makes the TurtleBot 3 more portable and customizable compared to the predecessors. Next, the TurtleBot 3 comes equipped with more advanced sensing capabilities. Additionally, like all TurtleBot versions, it runs on open-source ROS software, but the TurtleBot 3 benefited from the increased maturity and support of the ROS community by its release. In terms of reliability, the TurtleBot 3 improved upon hardware issues in older models for more robust operation. Finally, the TurtleBot 3 strikes a balance between affordability and features that made it accessible as an educational and research platform. While not the cheapest option, it provides good value for capabilities compared to TurtleBot 2 and the more expensive later model, TurtleBot 4. In summary, the blend of compact design, sensor upgrades, software maturity, hardware reliability, community support and balanced cost of TurtleBot 3 distinguish it as a versatile and capable robotics platform.

TurtleBot 3 Burger was released in 2017 and it focuses on higher education. It is one of the variants in the TurtleBot 3 series, alongside the TurtleBot 3 Waffle and TurtleBot 3 Waffle Pi [5]. The TurtleBot 3 Burger is a good choice as it offers many open-source software and libraries for users to download and share with other users, it supports ROS, which is widely used for education and research, it is relatively cheaper and smaller compared to the TurtleBot 3 Waffle variant [26]. The TurtleBot 3 Burger is able to perform various kinds of activities without adding other components. With that being said, it is capable of navigating itself to a designated location in real-time by using the SLAM algorithm [27]. Figure 2.8 below shows the components of TurtleBot 3 Burger.

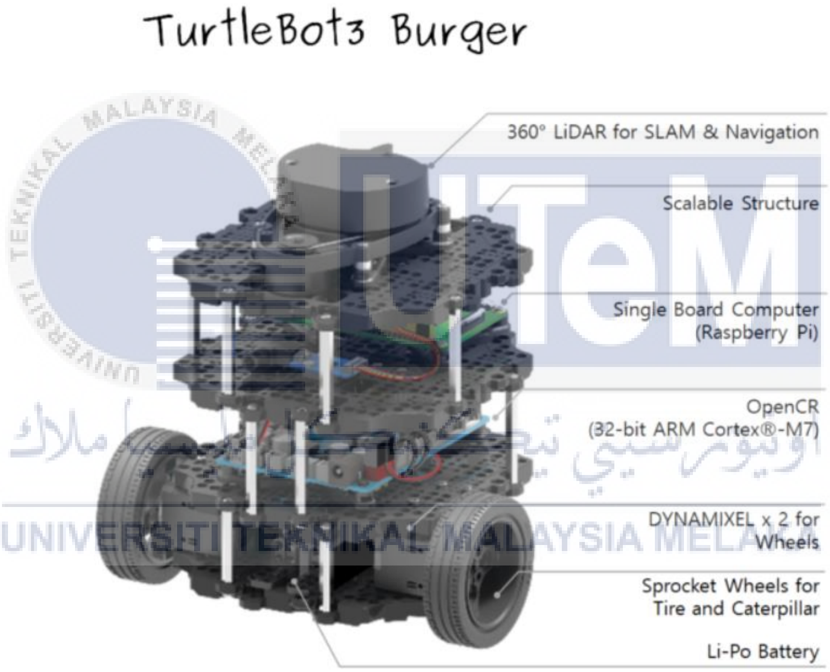


Figure 2.4: Components of TurtleBot 3 Burger [12]

2.4 Robot Operating System (ROS)

The Robot Operating System (ROS) is a widely used, popular robotics tool with open source and an active community of contributors that is easy for users to access, especially for new learners who are looking for a platform to start getting their hands on robotics. It is a versatile robotics framework which is compatible with several operating systems and it contains various tools and libraries contributed by other users that can be used to program a robot [28]. ROS was developed back in 2007 by the

Stanford Artificial Intelligence Lab and is currently actively developed and maintained by Willow Garage with the assistance from other organizations. TurtleBot is one of the standard ROS platform and it is most often used in education and research, especially in motion planning [9]. There are quite a few Linux distributions for ROS development such as Ubuntu, Debian, Fedora, Arch Linux and OpenSUSE. Ubuntu is the most preferred open-source operating system on Linux to run ROS in order to program the TurtleBot. Ubuntu comes with many versions. The Xenial Xerus Ubuntu 16.04 LTS version is the most preferred version for programming the TurtleBot 3 [29]. A ROS distribution is a versioned collection of ROS software packages. The goal of the ROS releases is to provide developers with a reasonably stable code base to work against until they are prepared to be released. Each distribution keeps a consistent collection of core packages until the distribution reaches its end of life (EOL), until then a new version of distribution will be released [30]. There are various versions of ROS distributions supported by TurtleBot 3, namely ROS Kinetic, ROS Melodic, ROS Dashing, ROS Foxy, ROS Galactic and ROS Humble. Some features are only supported by certain versions and these features can be implemented in TurtleBot 3 [12].

2.5 TurtleBot 3 navigation

Navigation is the ability to locate one's position and plan a route to reach the designated location. In order for a mobile robot to navigate autonomously, it has to determine its current location, desired destination and the best route to get to that destination [31]. It will only be considered as autonomous navigation if there is no human manipulation involved. Object detection and avoidance are also very important in autonomous navigation. Static obstacles are those that do not move such as walls, while dynamic obstacles are those that are moving such as walking human and cars moving on the road [32].

2.5.1 Types of TurtleBot 3 navigation methods

2.5.1.1 Simultaneous Localization And Mapping (SLAM)

Simultaneous Localization And Mapping (SLAM) is a method used to construct an environmental map around the robot (also known as mapping), then use the known map to calculate its position (also known as localization). SLAM is a two-operation process in which these two actions occur simultaneously [33]. There are three common SLAM navigation technologies, namely Laser SLAM, visual SLAM and laser-vision fusion SLAM. Laser SLAM algorithm is implemented by mainly using LiDAR sensor. The particles follow the robot's movements, and a probability is assigned to each particle based on a comparison between the positions of the particles and the LiDAR scan. The particles eventually converge after a number of rounds and the robot's precise location can be determined. Visual SLAM algorithm is carried out by using camera sensor. It utilizes a binocular camera to capture RGB images of the environment. Feature points are detected using the FAST algorithm and their descriptors are calculated with the BRIEF algorithm. The camera pose is determined through rough matching of consecutive frame images, refined by the RANSAC algorithm for optimal matching. A local map is generated and the back end optimizes pose states and loop constraints, ensuring global consistency. Loopback constraints facilitate a return to the origin, mitigating accumulated errors and enabling the creation of a dense map. Laser-vision fusion SLAM algorithm combines the use of LiDAR and camera sensors. It employs parallel processing of laser and vision localization algorithms in its front-end. LiDAR and camera work interchangeably for robot positioning, even in extreme ambient illumination conditions. Integrating laser-scanned points with image feature points enhances the depth optimization of the pure visual SLAM's interframe localization algorithm. Visual SLAM also aids in correcting LiDAR-induced drift, improving the overall positioning accuracy of laser SLAM [34].

2.5.1.2 Deep Q-Network (DQN)

Deep reinforcement learning using a Deep Q-Network (DQN) is a method whereby the DQN agent learns to navigate towards a goal and avoid obstacles through interactions with a simulated environment in ROS Gazebo simulator. The agent selects actions based on the robot's laser scan sensor data and odometry information which represent the state. It receives positive or negative rewards based on factors like reaching the goal and collisions. This reinforcement signal trains the DQN neural network to approximate the optimal action-value function and improve its policy. The trained model enables the TurtleBot robot to determine collision-free paths autonomously in real-time. Communication between the environment, sensors and actuators is handled through the ROS. Overall, the navigation method utilizes deep reinforcement learning, specifically the DQN algorithm, to train the mobile robot via rewards from its experiences and map out optimal routes by learning [35].

2.5.1.3 Real Time Object Detection

Real Time Object Detection (RTOD) method involves training a Convolutional Neural Network (CNN) with a dataset of high number of images to enable real-time identification of specific objects within the Turtlebot's environment. The CNN is designed to categorize objects into distinct classes, such as Quadcopter, Mars Rover, Bowl, and Wheel, allowing the robot to recognize and differentiate between these objects in its surroundings. Additionally, the method incorporates the use of Haar Cascades for object detection, providing a complementary approach to the CNN-based detection. By leveraging these real-time object detection techniques, the robot is able to identify and localize itself within its environment, enabling subsequent navigation to specified locations. The integration of RTOD with the ROS framework and the utilization of depth maps further enhance the robot's ability to understand its surroundings and make informed navigation decisions. Overall, the RTOD method plays a crucial role in enabling the robot to autonomously recognize and respond to its environment in real time, facilitating its indoor localization and navigation capabilities [36].

2.5.1.4 Rapidly-exploring Random Tree (RRT)

Rapidly-exploring Random Tree (RRT) is a popular algorithm used for path planning in robotics. It is a probabilistic algorithm that generates a tree of random samples in the configuration space of a robot. The algorithm starts with an initial configuration of the robot and then randomly generates new configurations in the configuration space. The algorithm then connects the new configuration to the nearest configuration in the tree, creating a new branch. This process is repeated until a goal configuration is reached or a maximum number of iterations is reached. One of the key advantages of RRT is that it can handle high-dimensional configuration spaces and complex environments with obstacles. The algorithm is also incremental, meaning that it can be used to plan paths in real-time as the robot moves through the environment. Additionally, RRT can be extended to handle kinodynamic constraints, which makes it suitable for planning paths for robots with non-holonomic constraints. There are several variants of the RRT algorithm, including RRT*, which is an improved version of RRT that converges to the optimal path. RRT* uses a cost function to guide the growth of the tree towards the goal configuration, resulting in a more optimal path. Overall, RRT is a powerful and widely used algorithm for path planning in robotics [37].

2.5.1.5 Z-Number-based fuzzy logic

Z-Number-based fuzzy logic, as the name suggests, combines Z-numbers with fuzzy logic to address uncertainty in robot navigation. It involves creating Z-number-based fuzzy rules, converting sensor data into Z-numbers, and using Z-number arithmetic to make decisions in uncertain environments. The integration of Z-numbers into the fuzzy logic framework allows for a more accurate representation of uncertainty in robot navigation tasks. By mapping elements to degrees of certainty and uncertainty using paired membership functions, Z-numbers provide flexibility and adaptability in representing and reasoning about the robot's navigation behavior. This approach enables robots to navigate more naturally and intuitively, making decisions based on varying degrees of sensory inputs. The Z-Number-Based Fuzzy Logic Approach has shown promising results in improving mobile robot navigation in unknown and dynamic environments. It effectively handles uncertainty and imprecise information through Z-numbers, allowing for more intelligent and effective navigation. By considering the fuzzy membership function's lower and upper bounds, Z-numbers enable a more comprehensive evaluation of the robot's environment and the generation of more precise control actions. This approach has significant implications for developing autonomous robots operating in dynamic environments. It opens up new possibilities for robust and adaptive navigation systems, with potential applications in robotics-assisted healthcare, logistics, and exploration [38].

2.5.1.6 Waypoing following

Waypoint following is a navigation technique in robotics where a mobile robot follows a path defined by a sequence of waypoints. It is usually accompanied by the pure pursuit algorithm (PPA). PPA is a popular tracking algorithm that computes the robot's linear and angular velocities based on its current pose and predefined waypoints. The algorithm uses geometric equations to determine the distance and angle between the robot and the next waypoint, then dictates the robot's movement direction and speed. One of the key factors of PPA is the lookahead distance. Smaller values can improve accuracy but may cause oscillations, while larger values yield smoother paths. Depending on the environment, smaller lookahead distance can cause undesirable oscillations as the robot approached waypoints, while larger values allowed smoother paths but can cause the robot to cut corners before reaching the waypoints. An appropriate lookahead distance is needed to balance path tracking accuracy with avoiding instability and slowdowns near the waypoints [39].

2.5.2 Summary of types of navigation methods for TurtleBot 3

Table 2.2: Types of navigation methods for TurtleBot 3

Navigation method	References	Sensors used	Description
SLAM	[33][34]	- LiDAR - Odometry - IMU - Wheel encoder	Advantages: - Improve accuracy and efficiency. - Robust to noise. - Adapted to various sensors and platforms. Disadvantages:

			- Computational complexity. - Sensitivity to errors and sensor noise. - Challenging in cluttered environments.
DQN	[35]	- LiDAR - Depth sensor	Advantages: - Enables adaptation to new environments. - Deep neural network can approximate complex action-value functions for effective decision making. - Works well even with high-dimensional and continuous state spaces. Disadvantages: - Requires large amounts of training data from environment interactions. - Sensitive to hyperparameters. - Large neural network model can be computationally intensive to train.

RTOD	[36]	- Camera - LiDAR - Depth sensor	Advantages: - Able to recognize and categorize specific objects in the environment in real time. - Able to make informed decisions regarding obstacle avoidance and path planning through depth maps. - Reducing the need for manual intervention. Disadvantages: - Limited ability to identify and respond to a broader range of environmental features. - Computational overhead, potentially impacting the real-time responsiveness of the system. - Effectiveness may be influenced by variations in environmental conditions, such as changes in lighting,

			object occlusion, or the presence of unfamiliar objects.
RRT	[37]	- LiDAR - IMU - Wheel encoder - Camera	Advantages: - Can handle high-dimensional configuration spaces and complex environments with obstacles. - Can be used to plan paths in real-time - Suitable for planning paths for robots with non-holonomic constraints. - Computationally efficient and can generate paths quickly. Disadvantages: - Quality of the path generated can vary depending on the random samples generated. - May not always find the optimal path, especially in complex environments with many obstacles.

			- May not be suitable for environments with narrow passages or tight spaces.
Z-number-based fuzzy logic	[38]	- LiDAR - IMU - Wheel encoder	Advantages: - Incorporates an additional level of uncertainty modeling, allowing for more comprehensive handling of uncertainties. - Effectively handle situations where precise information is lacking or conflicting data is present. - Flexible decision-making in incomplete or ambiguous data. Disadvantages: - Complexity and expert knowledge required in assigning Z-numbers. - Room for efficiency and computational complexity enhancements.
Waypoint following	[39]	- Odometry - IMU	Advantages:

		- Wheel encoder	- Geometric simplicity for computational efficiency - Lookahead distance parameter allows tuning path tracking performance Disadvantages: - Requires tuning effort to balance accuracy and smoothness

Based on Table 2.2 above, SLAM and waypoint following, including the pure pursuit algorithm, have been selected to develop the navigation system for the TurtleBot 3. SLAM integrates LiDAR, odometry, IMU, and wheel encoder data to accurately map the racetrack and precisely determine the robot's location. This capability is vital for maintaining constant awareness of the robot's position and effectively navigating around obstacles, crucial during high-speed maneuvers. Complementing SLAM, waypoint following with the pure pursuit algorithm offers a direct yet effective method to execute predefined paths with precision. The pure pursuit algorithm continuously adjusts the robot's steering to track a predefined racing line, utilizing inputs from odometry, IMU and wheel encoders for optimized path tracking. Together, SLAM ensures robust localization and mapping accuracy, while waypoint following with the pure pursuit algorithm enables reliable path execution, essential for navigating racing lines smoothly in environments like the F1TENTH competition. This integrated approach enhances computational efficiency and adaptability, ensuring the TurtleBot 3 performs effectively in autonomous racing scenarios where precision and real-time responsiveness are critical.

2.5.3 Summary of types of sensors used in TurtleBot 3

Table 2.3: Types of sensors used in TurtleBot 3

Type	References	Advantages	Disadvantages
Vision camera	[40][41]	- Multiple object tracking	- Susceptible to environment conditions
LiDAR	[42][43]	- Higher accuracy - Large measurement range - Not affected by lightning condition - Mapping and localization	- Expensive - Narrow point detection (miss object like glass)
Odometry	[44][45]	- Inexpensive - Real-time position estimation	- Accumulate of errors - Sensitive to slippage - Not effective for featureless surface
Wheel encoder	[46]	- Inexpensive - Real-time position estimation	- Accumulation of errors - Sensitive to slippage - Not effective for featureless surface

IMU	[8]	- Pose estimation and navigation - Track robot’s orientation and heading.	- Measurements can drift over time. - Affected by sensor noise, biases, temperature fluctuations.

Based on Table 2.3 above, the LiDAR, odometry, and wheel encoders offer a balanced approach to achieving reliable autonomous navigation on the racetrack. LiDAR provides high accuracy and a broad measurement range, ensuring precise mapping and localization capabilities. Odometry and wheel encoders complement LiDAR by offering cost-effective real-time position estimation and path tracking, essential for executing predefined racing lines with accuracy. Integrating these sensors enables robust sensor fusion, enhancing the TurtleBot 3's ability to navigate autonomously while adapting to varying track conditions and obstacles. The IMU further enhances navigation by providing pose estimation and tracking the robot's orientation and heading. Since the racetrack is static, integrating a camera sensor on the TurtleBot 3 is deemed unnecessary.

2.6 Types of SLAM algorithms

SLAM is an algorithm used in TurtleBot’s navigation system. TurtleBot 3 SLAM can be conducted using ROS as it contains tools such as Gazebo and RViz. Gazebo is a simulation software used to simulate the TurtleBot in a virtual environment created by the user. RViz, also known as ROS visualization, is used to visualize robot data, particularly the map created from the LiDAR sensor. By having the navigation goals and posing estimation functionalities, RViz allows autonomous navigation of Robots in ROS [47]. There are various algorithms that can be used to implement SLAM on TurtleBot 3.

2.6.1 Gmapping

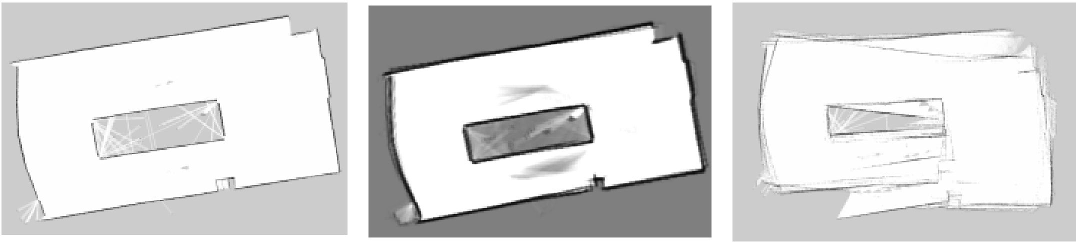
Gmapping algorithm is a laser-based SLAM algorithm utilizing Particle Filter approach. It addresses common particle filter issues, such as computational complexity and the depletion problem, by employing an adaptive resampling technique. Unlike traditional approaches, adaptive resampling is limited and performed when necessary, preventing unnecessary particle elimination. This method enhances robot localization accuracy by integrating sensor data and odometry motion model during the prediction step. The quality of laser scan matching further reduces the required number of particles [48]. Gmapping is suitable for indoor mapping [49].

2.6.2 Cartographer

Cartographer algorithm is a real-time SLAM system designed for 2D and 3D environments across various platforms and sensor configurations. As an open-source library with a ROS wrapper, it deviates from particle filter algorithms, opting for pose estimation to address error accumulation over prolonged iterations. Laser scans are matched iteratively with a recent submap, minimizing dependence on past scans and ensuring loop closure through scan matching. The conversion process from scan frame to submap frame involves representing submaps as probability grid points. Hits and misses are computed during new scan insertion, updating grid points with appropriate probabilities. Cartographer's scan matching is rooted in Ceres scan matching, maximizing probabilities for accurate scan pose determination in the submap [48].

2.6.3 Hector SLAM

Hector SLAM algorithm is a 2D SLAM system that integrates LiDAR scan matching and a 3D navigation approach using Extended Kalman Filter with an inertial sensing system. Specifically designed for onboard computations, it ensures real-time six degrees of freedom robot pose determination during motion. The system achieves high update rates for 2D LiDAR-based mapping. Laser beam endpoint alignment with the obtained map is facilitated through a Gaussian-Newton optimization approach, implicitly performing scan matching with all preceding scans [48]. Figure 2.13 below shows the examples of map made by Gmapping, Cartographer and Hector SLAM algorithm.



Gmapping　　Cartographer　　Hector SLAM

Figure 2.5: Examples of map made by Gmapping, Cartographer and Hector SLAM algorithm [48]

2.6.4 Summary of Types of SLAM algorithms

Table 2.4: Types of SLAM algorithms

Type	References	Advantages	Disadvantages
Gmapping	[48][49]	- High quality maps - Robustness in experiments	- Error accumulation - Careful parameter tuning
Cartographer	[48]	- High mapping accuracy - Wide range of configurable parameters - Precise position estimation - Real-time mapping	- Human error - Distortion in curved surface - Computational complexity - Limitations in closing large loops
Hector SLAM	[48]	- Fast execution - Less dependent on other sensors	- Relies on LiDAR-only system

			- Computational complexity - Limitations in closing large loops - Careful parameter tuning - Low quality maps in certain environment

Based on Table 2.4 above, SLAM Gmapping emerges as the optimal choice in my TurtleBot 3 project aimed at autonomous navigation on a racetrack. The algorithm is noted for its ability to generate high-quality maps and has demonstrated robustness in practical experiments, essential for ensuring accurate localization and effective path planning in dynamic environments. While it requires careful parameter tuning and may experience error accumulation over time, these challenges are manageable with proper calibration and align well with the capabilities of LiDAR sensors, which are crucial components of my sensor setup. Therefore, SLAM Gmapping promises to provide reliable performance, leveraging its proven track record in various applications to enhance the TurtleBot 3's autonomy and navigation capabilities on the racetrack.

2.7 Overall summary

F1TENTH provides various competitions and an immersive learning environment which facilitates research and development in autonomous driving and artificial intelligence [16]. The F1TENTH race car used in competition is in a 1/10th-scale and is equipped with multiple sensors used for autonomous navigation such as IMU, LiDAR and camera [19]. The F1TENTH competition is a kind of autonomous robotics competition where participants develop self-driving algorithms for F1TENTH race cars to race on a randomized racetracks. The SLAM algorithm is used in the F1TENTH navigation system [1].

TurtleBot is an open-source robotics platform compatible with ROS which is widely used for education and research. It is versatile, affordable, and just like the F1TENTH platform, it supports the SLAM technology which is used for autonomous navigation [9]. The TurtleBot 3 Burger is one of the variants of the TurtleBot 3 series which is suitable for developing an autonomous navigation system. It is relatively cheaper and smaller compared to the other variant [26]. It is able to navigate autonomously through SLAM method with just its existing components such as 360-degree LiDAR sensor and Raspberry Pi [27].

ROS offers a versatile framework compatible with various operating systems and provides a range of tools and libraries contributed by a vibrant community for programming robots [28]. ROS has many versions of distributions which contain software packages that support TurtleBot programming and features [30]. Hence, ROS Kinetic is the most preferred distribution as it supports the most features in TurtleBot 3 compared to other distributions [12].

SLAM is a method used in autonomous navigation preferably in F1TENTH and TurtleBot. Laser SLAM employs LiDAR sensor to map the environment and determine the robot's location. Since the TurtleBot 3 Burger has a built-in 360-degree LiDAR sensor, Laser SLAM would be the suitable navigation method. SLAM is able to improve accuracy and efficiency, adapt to various sensors and platforms and is robust to noise. The LiDAR sensor has high accuracy, large measurement range and is suitable for mapping and localization [33][34]. Integrating waypoint following together with the pure pursuit algorithm (PPA) allows the TurtleBot 3 to autonomously navigate the predefined racetrack path. PPA computes velocities based on current position and geometric relationships to waypoints, balancing accuracy with path smoothness using a specified lookahead distance. This sequential approach optimizes the robot's ability to navigate autonomously and accurately on the racetrack [39]. With that being said, SLAM and waypoint following are suitable to implement in TurtleBot 3 navigation for autonomous racing.

Integrating LiDAR, odometry, wheel encoders, and IMU provides robust autonomous navigation capabilities for the TurtleBot 3 on a static racetrack. LiDAR ensures accurate mapping and localization [42][43], while odometry and wheel encoders offer real-time position estimation and path tracking [44][45][46]. The IMU enhances navigation by tracking orientation and heading, contributing to precise autonomous navigation [8]. This sensor fusion approach optimizes the TurtleBot 3's ability to navigate with accuracy without the need for additional camera sensors in this application.

SLAM has many different algorithms. SLAM Gmapping produces high quality maps and is robust, but it requires careful parameter tuning. It is more suitable for indoor mapping. SLAM Cartographer is accurate in mapping and position estimation, but it has many adjustable parameters and is suitable for real-time mapping. Hector SLAM only relies on LiDAR sensor for mapping. Hence, it does not carry out large loop closure and is less accurate [48][49]. Since the designated environment is an indoor racetrack, SLAM Gmapping is suitable for mapping.

CHAPTER 3

METHODOLOGY

3.1 Introduction

In this chapter, the methods and techniques used to achieve the objectives of this project will be discussed. First of all, the steps in implementing the project will be explained based on the flowchart of project overview in the first section. Next, the overall process of the system will be described according to the flowchart of system overview.

In order to achieve the first objective of this project, which is to create a map of the surrounding environment for TurtleBot 3 using SLAM method, the overall understanding on the concept and theory of SLAM algorithm is needed to correctly implement it on the TurtleBot 3 Burger effectively. The second objective which is to develop an autonomous racing navigation system for TurtleBot 3 with the map created from SLAM method, requires the knowledge of the theoretical concept of waypoint following and pure pursuit algorithm and Proportional – Integral – Derivative (PID) controller. The third objective which is to analyze the performance of the autonomous racing navigation system of TurtleBot 3 in terms of lap time and trajectory accuracy, the knowledge of the steps of calculating the RMSE between the expected and actual path of TurtleBot 3 on the racetrack is essential.

Furthermore, this chapter will also cover the experiment setup as well as the steps of all experiments to be implemented in order to fulfill all the 3 objectives of this project.

3.2 Project overview

Figure 3.1 below shows the project flowchart. The project flowchart describes the steps to implement the Final Year Project from start to finish. The project started off by understanding the project through conducting research and literature review. This is to have a better view of the important keywords in the project. Next, the problem statements, objectives and scopes are then identified and determined. After that is to set up the experiments that will satisfy the objectives of the project. Once the setup is done, the preliminary simulation will be conducted to obtain the preliminary results. The results are recorded and written in the report for Final Year Project 1. After that, the system will be further designed and developed. System testing and all the experiments will be carried out accordingly in Final Year Project 2. The final results will be recorded and analyzed in the final report.

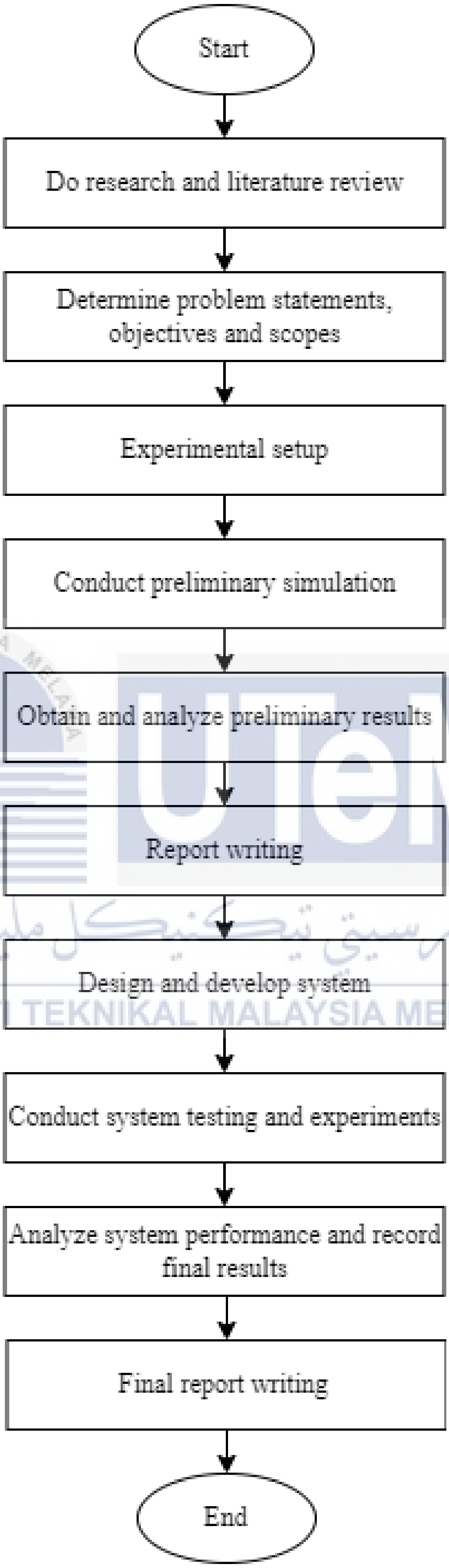


Figure 3.1: Project flowchart

3.3 System overview

Figure 3.2 below shows the overall system flowchart. The overall system flowchart describes the overall working process of the navigation system of the TurtleBot 3 Burger. Firstly, Gazebo and RViz software is opened in ROS. The model of the TurtleBot 3 is set as ‘Burger’. If the sensors on the TurtleBot 3 have successfully collected some data of the surrounding, the map of its nearby surroundings will be created and displayed in Rviz. It will then proceed to carry out the mapping process by moving around on the racetrack with the teleoperation node run in ROS. The map created can be viewed from the visualization in Rviz. After the mapping process is done, the map is saved. The saved map of the racetrack is then opened in Rviz. The autonomous racing navigation script is run and the TurtleBot 3 starts to run its navigation system on the map created to navigate autonomously around the racetrack. After the script is terminated, the TurtleBot 3 will stop moving and the graph of expected path against actual path of the TurtleBot 3 will be plotted automatically. The lap time and RMSE between the expected and actual path will also be calculated automatically and displayed on the plot.

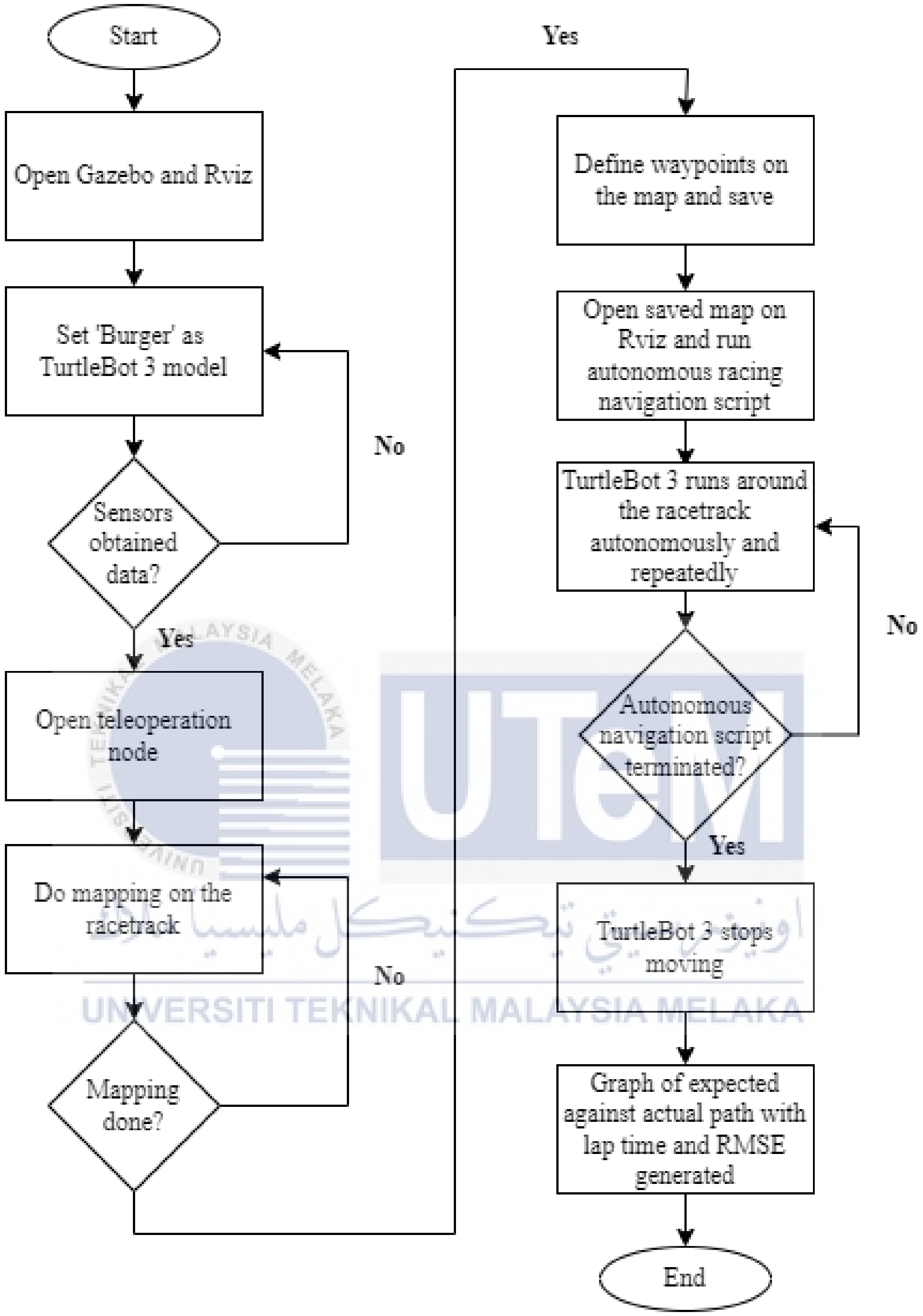


Figure 3.2: Overall system flowchart

3.4 Concept of SLAM process

SLAM is best described using probabilistic terms. Time is denoted by *t* and the robot's location is represented by x_t. For mobile robots navigating on a flat surface, the path is expressed as:

$$X_T = \{x_0, x_1, x_2, \ldots x_T\} \text{ ……… (1)}$$

T represents a terminal time. The initial location x_0 is known, while other positions remain unobservable. Odometry, denoted as u_t , furnishes relative information concerning the movement between time t−1 and time *t*. The following equation is given:

$$U_T = \{u_0, u_1, u_2, \ldots x_T\} \text{ ……… (2)}$$

Finally, the robot perceives objects within the surroundings. Let *m* represent the actual map of the environment. The robot measurements establish a connection between features in *m* and the robot location x_t. Assuming, without loss of generality, that the robot takes precisely one measurement at each time point, the sequence of measurements is represented as:

$$Z_T = \{z_0, z_1, z_2, \ldots z_T\} \text{ ……… (3)}$$

Figure 3.5 below depicts the variables central to the SLAM problem, illustrating the sequence of locations and sensor measurements, along with the causal relationships between these variables.

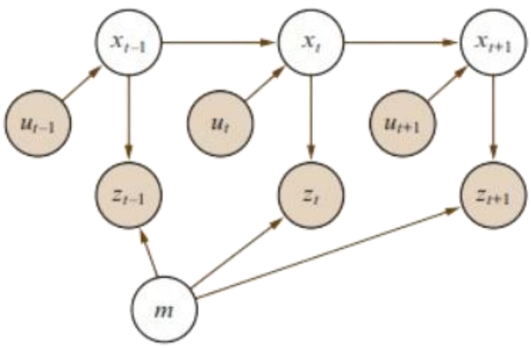


Figure 3.3: Graphical model of SLAM problem

The current challenge in SLAM is to reconstruct a model of the world, denoted as m, and the sequence of robot locations X_T using odometry and measurement data. There are two primary forms of the SLAM problem. The first is referred to as the full SLAM problem, where it estimates the posterior over the complete robot path along with the map:

$$p(X_T, m \backslash Z_T, U_T) \text{ ………. (3)}$$

The full SLAM problem entails computing the joint posterior probability over X_T and m based on the provided data. The variables to the right of the conditioning bar are directly observable to the robot, while those on the left are the sought-after variables. Offline SLAM algorithms for this problem are frequently batch-oriented, processing all data simultaneously. The second one is the online SLAM problem, which focuses on determining the current robot location through incremental algorithms, known as filters, processing one data item at a time. The online SLAM problem is defined as below:

$$p(x_t, m \backslash Z_T, U_T) \text{ ………. (4)}$$

To solve either SLAM problems, the robot relies on two models: one connecting odometry measurements u_t to robot locations x_{t-1} and x_t and another linking measurements z_t to the environment m and robot location x_t.

3.5 Concept of waypoint following and pure pursuit algorithm

Waypoint following and the pure pursuit algorithm are essential techniques in the field of autonomous vehicle navigation and robotics. Waypoint following is a navigation strategy where a vehicle or robot is directed to pass through a series of predefined points known as waypoints. These waypoints outline the desired path, and the vehicle continuously adjusts its trajectory to reach each waypoint sequentially, ensuring it stays on the intended route. This method involves recalculating the path to the next waypoint based on the vehicle's current position.

The pure pursuit algorithm is a geometric path-tracking method used in autonomous vehicles to follow a predefined path. The central concept involves calculating the curvature needed for the vehicle to steer towards a lookahead point on the path. This lookahead point is dynamically chosen based on the vehicle's current position and a specified lookahead distance. Figure 3.4 below shows the lookahead distance and lookahead point of pure pursuit algorithm.

Figure 3.4: Lookahead distance and lookahead point

The key formula for determining the curvature required to steer towards the lookahead point is given by:

$$\gamma = \frac{2 \cdot L sin(\alpha)}{{L_d}^2} \text{ .... (1)}$$

Where γ is the curvature, L is the distance between the rear axle and the lookahead point, α is the angle between the vehicle’s current heading and the line connecting the vehicle to the lookahead point and L_d is the lookahead distance, which is the distance from the vehicle to the lookahead point. From the curvature calculated from formula (1) above, the steering angle can be computed using the equation as follows:

$$\delta = \arctan(\gamma \cdot L) \text{ ………. (2)}$$

This formula ensures that the vehicle's steering is continuously adjusted to follow the path smoothly by considering the current position, heading, and the dynamic lookahead point.

3.6 Proportional – Integral – Derivative (PID) controller

A Proportional – Integral – Derivative (PID) controller is a widely used control feedback mechanism in industrial systems. It combines proportional, integral, and derivative controls to minimize the error between a desired setpoint and the actual process variable. The proportional control (P) generates an output proportional to the current error, with the proportional gain, Kp determining the response magnitude. The integral control (I) accounts for accumulated past errors, with the integral gain, Ki helping to eliminate residual steady-state errors. The derivative control (D) predicts future errors based on their rate of change, with the derivative gain, Kd adding a damping effect to improve stability and reduce overshoot. Together, these components allow the PID controller to dynamically adjust process inputs, aiming for minimal steady-state error and optimal transient response. Figure 3.5 below shows the PID controller block diagram.

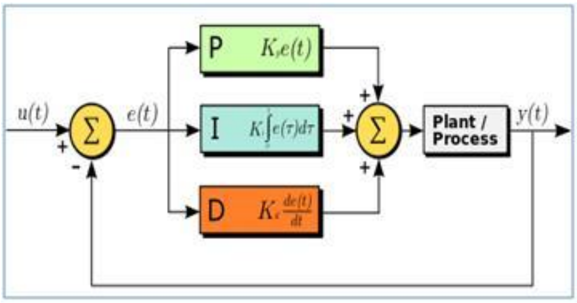


Figure 3.5: PID controller block diagram

The continuous-time formula for the PID control output, u(t) is given by:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt} \text{ .... (1)}$$

Where u(t) is the control output, e(t) is the error at time, t (difference between the desired setpoint and the actual process variable), Kp is the proportional gain, Ki is the integral gain and Kd is the derivative gain.

The discrete-time formula for the PID control output, u[n] is given by:

$$u[n] = K_p e[n] + K_i \sum_{i=0}^{n} e[i] \Delta t + K_d \frac{e[n]=e[n=1]}{\Delta t} \text{......... (2)}$$

Where u[n] is the control output at discrete time step, n, e[n] is the error at discrete time step, n, Δt is the time setep duraction. The two formulae combine the three control actions to correct the process variable and reduce the error dynamically.

3.7 Root mean square error (RMSE) calculation

The root mean square error is calculated to determine the error difference between the expected path that should be taken by the TurtleBot 3 to move around the racetrack and the actual path that is actually taken by the TurtleBot 3 itself. The RMSE provides a quantitative value representing the accuracy of the robot's path following. Figure 3.5 below shows a snippet of the RMSE calculator in the autonomous racing navigation script. The full script can be referred to Appendix D.

```
def calculate_rmse(self):
    # Ensure the lengths of the intended and actual path data are the same
    min_length = min(len(self.intended_path_data['x']), len(self.actual_path_data['x']))
    intended_path = np.array(list(zip(self.intended_path_data['x'][:min_length], self.intended_path_data['y'][:min_length])))
    actual_path = np.array(list(zip(self.actual_path_data['x'][:min_length], self.actual_path_data['y'][:min_length])))

    # Calculate RMSE
    squared_errors = np.sum((intended_path - actual_path) ** 2, axis=1)
    mse = np.mean(squared_errors)
    rmse = np.sqrt(mse)

    return rmse
```

Figure 3.6: RMSE calculator

The steps of RMSE calculation is as follows:

1. The expected path data is collected only during the first lap, consisting of the waypoints the TurtleBot 3 is supposed to follow. The actual path data, on the other

hand, is collected throughout the TurtleBot's navigation, representing the positions the robot actually reaches.

2. The lenghts of the expected and actual path data are compared and made sure that they are the same. This is done by taking the minimum length of the two datasets.
3. The expected and actual paths are then converted to numpy arrays for easier computation. Each path is represented as a series of (x, y) coordinates.
4. The difference between each element of each array are calculated and squared to get the squared errors for each coordinate pair.
5. The mean of the squared errors is calculated to get the Mean Squared Error (MSE).
6. Lastly, take the square root of the MSE to obtain the RMSE.

3.8 Experiment setup

The software used in this project is ROS Kinetic installed on Ubuntu 16.04 LTS (Xenial Xerus). Dependent ROS packages and TurtleBot 3 packages are also installed. Figure 3.5 below shows the Ubuntu 16.04 desktop interface.

Figure 3.7: Ubuntu 16.04 desktop interface

The experiments will be conducted in the virtual environment. The virtual environment is a racetrack constructed in ROS Gazebo. The outer dimensions of the racetrack is 4.5 m x 3.25 m, while the inner dimensions is 2.5 m x 1.25 m. Figure 3.6 below shows the virtual racetrack constructed in Gazebo.

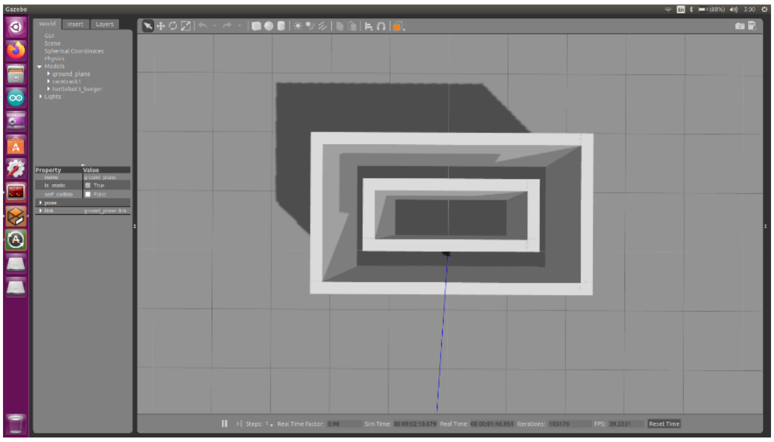


Figure 3.8: Virtual racetrack constructed in Gazebo

3.9 Experiment implementation

3.9.1 Experiment 1: Simulation of TurtleBot 3 Burger in the virtual world

This simulation is about simulating the TurtleBot 3 Burger in the virtual world. The objective of this simulation is to understand the working principles of mapping and navigation of TurtleBot 3 Burger in the virtual world. The software involved are ROS, Gazebo and Rviz. Since this simulation is just to familiarize with the working principles of the mapping and navigation process of TurtleBot 3 Burger, no parameters are being measured in this simulation. For the expected results of this simulation, the TurtleBot 3 Burger should be able to do mapping in the Gazebo simulator with the teleoperation node and the Rviz simulator should be able to visualize the mapping process through the TurtleBot's simulated sensor. After the mapping process is done, the TurtleBot 3 Burger should be able to navigate to the designated goal set in Rviz and at the same time avoid any obstacles in the way by using the '2D Pose Estimate' and '2D Nav Goal' tools.

3.9.2 Experiment 2: Analysis of the performance of TurtleBot 3 Burger in virtual racetrack

Objective: To ensure that the autonomous racing script is working properly and also to evaluate the baseline performance of the system which will be further improved in the upcoming experiments.

Tools: ROS, Gazebo, Rviz

Parameters:

Constant variable: Size of virtual racetrack

Independent variables: Linear velocity, lookahead distance, angular velocity PID gains, angle threshold, linear velocity reduction factor

Dependent variable: Lap time, RMSE

Procedure:

1. Source the bash.rc file to configure the environment.
2. Launch the virtual racetrack in Gazebo.
3. Run the SLAM node and open Rviz to visualize the map of the racetrack.
4. Run the teleoperation node to move the TurtleBot 3 Burger around to map the racetrack.
5. Save the map of the racetrack after the mapping process is done.
6. Launch the racetrack again in Gazebo and the saved map in Rviz.
7. Define the waypoints of the map using the ‘Publish Point’ tool as shown in Figure 3.8 below:

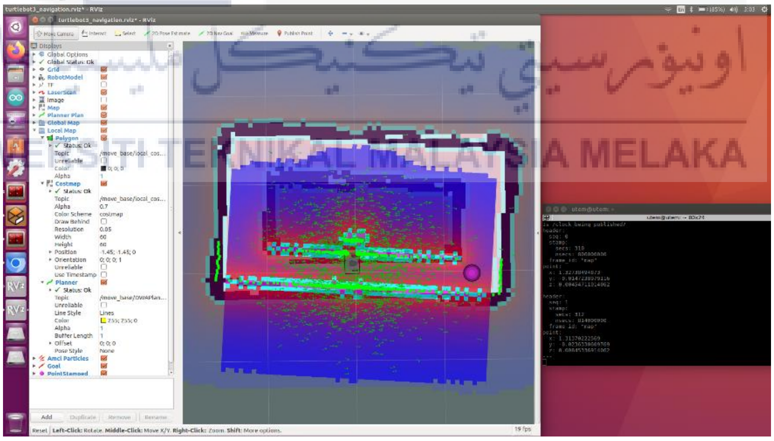


Figure 3.9: Defining waypoints on the map

8. Record the coordinates of the waypoint displayed on the terminal in the autonomous racing navigation script in Appendix D.
9. The waypoints are listed in the format as shown in Figure 3.9 below:

```python
# Define the waypoints
WAYPOINTS = [
    (1.41405820847, 0.0293627232313),
    (1.40997290611, 1.12405860424),
    (-1.15272676945, 1.28899729252),
    (-1.23371112347, 0.0824709683657)
]
```

Figure 3.10: Waypoints on the racetrack

10. The parameters of the autonomous racing navigation script are initialized as shown in Figure 3.10 below:

```python
# Pure Pursuit parameters
LOOKAHEAD_DISTANCE = 0.2  # meters
LINEAR_VELOCITY = 0.2  # m/s
ANGLE_THRESHOLD = 0 # radians
VELOCITY_REDUCTION = 0

# PID parameters for angular velocity
ANGULAR_PID_PARAMS = {
    'Kp': 0.7,
    'Ki': 0,
    'Kd': 0
}
```

Figure 3.11: Autonomous racing navigation parameters

11. Run the autonomous racing navigation script and observe the behavior of the TurtleBot 3 while racing around the racetrack.
12. The lap time is calculated from the moment it crosses the first waypoint until it crosses the first waypoint again after completing a lap.
13. The lap time and RMSE of each lap are displayed on the terminal as shown in Figure 3.8 below.

```
utem@utem: ~
utem@utem: ~ 80x24
utem@utem:~$ rosrun pure_pursuit pure_pursuit9.py
[INFO] [1718034928.623529, 60.198000]: Lap timer started.
[INFO] [1718034977.740805, 98.037000]: Lap 1: RMSE: 1.395 m, Lap Time: 49.12 sec
onds
```

Figure 3.12: Terminal displaying the lap time and RMSE

14. After 5 laps of race, terminate the script.
15. Analyze the graph of expected path against actual path of the TurtleBot 3 for every lap.

16. Tabulate and analyze the lap time and RMSE.

3.9.3 Simulation 3: Analysis of the performance of TurtleBot 3 Burger in virtual racetrack with different waypoint density levels

Objective: To analyze the effect of different waypoint density levels on the time taken for the TurtleBot 3 Burger to finish a lap as well as the RMSE between the expected path and the actual path of its motion in the virtual racetrack.

Tools: ROS, Gazebo, Rviz

Parameters:
Constant variable: Size of virtual racetrack, Linear velocity, lookahead distance, angular velocity PID gains, angle threshold, linear velocity reduction factor
Independent variables: Waypoint density level
Dependent variable: Lap time, RMSE

Procedure:

1. Source the bash.rc file to configure the environment.
2. Launch the virtual racetrack in Gazebo and the its map in Rviz.
3. Define the waypoints of the map using the ‘Publish Point’ tool, this time increasing the waypoints from the previous experiment.
4. Record the coordinates of the waypoint displayed on the terminal in the autonomous racing navigation script in Appendix D. Figure below shows the medium level of waypoint density.

```
# Define the waypoints
WAYPOINTS = [
    (1.37903225422, -0.0234580878168),
    (1.41338634491, 0.578707754612),
    (1.41644704342, 1.14961564541),
    (0.121521875262, 1.27317237854),
    (-1.2282140255, 1.28206908703),
    (-1.2495187521, 0.705962061882),
    (-1.30736768246, 0.0490701533854),
    (0.219430014491, -0.0479167960584)
]
```

Figure 3.13: Medium waypoint density

5. Run the autonomous racing navigation script and observe the behavior of the TurtleBot 3 while racing around the racetrack.
6. After 5 laps of race, terminate the script.
7. Analyze the graph of expected path against actual path of the TurtleBot 3 for every lap.
8. Tabulate and analyze the lap time and RMSE.
9. Repeat step 1 to 9 by increasing the waypoint density as shown in Figure below:

```
# Define the waypoints
WAYPOINTS = [
    (0.55785882473, -0.0744303613901),
    (0.986305236816, -0.0964283570647),
    (1.37430500984, -0.041693713516),
    (1.4085495472, 0.167969807982),
    (1.42460131645, 0.574233055115),
    (1.43535208702, 0.907108724117),
    (1.43031644821, 1.16660523415),
    (1.07386648655, 1.22979664803),
    (0.652593672276, 1.22451078892),
    (-0.745338320732, 1.30347430706),
    (-1.01950228214, 1.32738614082),
    (-1.23254036903, 1.32750248909),
    (-1.22658610344, 1.10366272926),
    (-1.23305869102, 0.677461087704),
    (-1.26714420319, 0.232884287834),
    (-1.26731300354, 0.0213141478598),
    (-0.595813214779, -0.0466784350574),
    (0.0376093015075, -0.0449083335698)
]
```

Figure 3.14: High waypoint density

10. Compare the lap time and RMSE with the previous experiment.

3.9.4 Simulation 4: Analysis of the performance of TurtleBot 3 Burger in virtual racetrack with varying linear velocities, lookahead distances, and angular velocity proportional gains

Objective: To analyze the effect of varying linear velocities, lookahead distances and angular velocity proportional gains on the TurtleBot 3 Burger's behavior in terms of its ability to complete the lap, the time taken for it to finish a lap as well as the RMSE between the expected path and the actual path of its motion in the virtual racetrack.

Tools: ROS, Gazebo, Rviz

Parameters:

Constant variable: Size of virtual racetrack, angular velocity integral and derivative gains, angle threshold, linear velocity reduction factor

Independent variables: Linear velocity, lookahead distance, angular velocity proportional gain

Dependent variable: Lap time, RMSE

Procedure:

1. Source the bash.rc file to configure the environment.
2. Launch the virtual racetrack in Gazebo and the its map in Rviz.
3. Increase the value of the linear velocity from the previous experiment and at the same time, tune the lookahead distance and angular velocity to balance the system.
4. Run the autonomous racing navigation script in Appendix D and observe the behavior of the TurtleBot 3 while racing around the racetrack.
5. After 3 laps of race, terminate the script.
6. Analyze the graph of expected path against actual path of the TurtleBot 3 for every lap.
7. Tabulate and analyze the lap time and RMSE.
8. Repeat step 1 to 7 by increasing the value of the linear velocity, tune the lookahead distance and angular velocity in a trial-and-error way until satisfactory result in terms of lap time and RMSE is obtained.
9. Compare the lap time and RMSE with the previous experiment.

3.9.5 Simulation 5: Analysis of the performance of TurtleBot 3 Burger in real world racetrack with varying angle thresholds and linear velocity reduction factors

Objective: To analyze the effect of varying angle thresholds and linear velocity reduction factors on the TurtleBot 3 Burger’s behavior in terms of its ability to complete the lap, the time taken for it to finish a lap as well as the RMSE between the expected path and the actual path of its motion in the virtual racetrack.

Tools: ROS, Gazebo, Rviz

Parameters:

Constant variable: Size of virtual racetrack, Linear velocity, lookahead distance, angular velocity PID gain

Independent variables: Angle threshold, linear velocity reduction factor

Dependent variable: Lap time, RMSE

Procedure:

1. Source the bash.rc file to configure the environment.
2. Launch the virtual racetrack in Gazebo and the its map in Rviz.
3. Adjust the angle threshold and linear velocity reduction factor to balance the system.
4. Run the autonomous racing navigation script in Appendix D and observe the behavior of the TurtleBot 3 while racing around the racetrack.
5. After 3 laps of race, terminate the script.
6. Analyze the graph of expected path against actual path of the TurtleBot 3 for every lap.
7. Tabulate and analyze the lap time and RMSE.
8. Repeat step 1 to 7 by adjusting the angle threshold and linear velocity in a trial-and-error way until satisfactory result in terms of lap time and RMSE is obtained.
9. Compare the lap time and RMSE with the previous experiment.

3.9.6 Simulation 6: Analysis of the performance of TurtleBot 3 Burger in real world racetrack with varying angular velocity integral and derivative gains

Objective: To analyze the effect of varying angular velocity integral and derivative gains on the TurtleBot 3 Burger's behavior in terms of its ability to complete the lap, the time taken for it to finish a lap as well as the RMSE between the expected path and the actual path of its motion in the virtual racetrack.

Tools: ROS, Gazebo, Rviz

Parameters:

Constant variable: Size of virtual racetrack, linear velocity, lookahead distance, angular velocity proportional gain, angle threshold, linear velocity reduction factor
Independent variables: Angular velocity integral and derivative gains
Dependent variable: Lap time, RMSE

Procedure:

1. Source the bash.rc file to configure the environment.
2. Launch the virtual racetrack in Gazebo and the its map in Rviz.
3. Adjust the angular velocity integral and derivative gains to balance the system.
4. Run the autonomous racing navigation script in Appendix D and observe the behavior of the TurtleBot 3 while racing around the racetrack.
5. After 3 laps of race, terminate the script.
6. Analyze the graph of expected path against actual path of the TurtleBot 3 for every lap.
7. Tabulate and analyze the lap time and RMSE.
8. Repeat step 1 to 7 by adjusting the angular velocity integral and derivative gains in a trial-and-error way until satisfactory result in terms of lap time and RMSE is obtained.
9. Compare the lap time and RMSE with the previous experiment.

3.9.7 Summary of simulations:

As a summary, Experiments 1 and 2 are conducted to meet Objective 1, which is to create a map of the surrounding environment for TurtleBot 3 using SLAM method. Experiments 1 and 2 are conducted to map the surrounding environment of the TurtleBot 3. Next, Experiments 2, 3, 4, 5 and 6 are carried out to meet Objective 2, which is to develop an autonomous racing navigation system for TurtleBot 3 with the map created from SLAM method. Experiment 2 is carried out to evaluate the baseline performance, Experiment 3 is carried out by implementing different waypoint density levels, Experiment 4 is carried out by tuning the linear velocity, lookahead distance and angular velocity proportional gain, Experiment 5 is carried out by tuning the angle threshold and linear velocity reductor factor, Experiment 6 is carried out by tuning the angular velocity integral and derivative gains. Lastly, Objective 3 which is to analyze the performance of the autonomous racing navigation system of TurtleBot

3 in terms of lap time and trajectory accuracy, is fulfilled through the implementation of Experiments 2, 3, 4, 5 and 6, whereby Experiments 2, 3, 4, 5 and 6 are implemented by analyzing the performance of the navigation system in terms of lap time and RMSE.

Table 3.1: Objectives fulfilment for each experiment

Experiment	Objective 1	Objective 2	Objective 3
1			
2			
3			
4			
5			
6			

Colour	Description
	Fulfill
	Partially-fulfill

CHAPTER 4

RESULTS AND DISCUSSIONS

4.1 Introduction

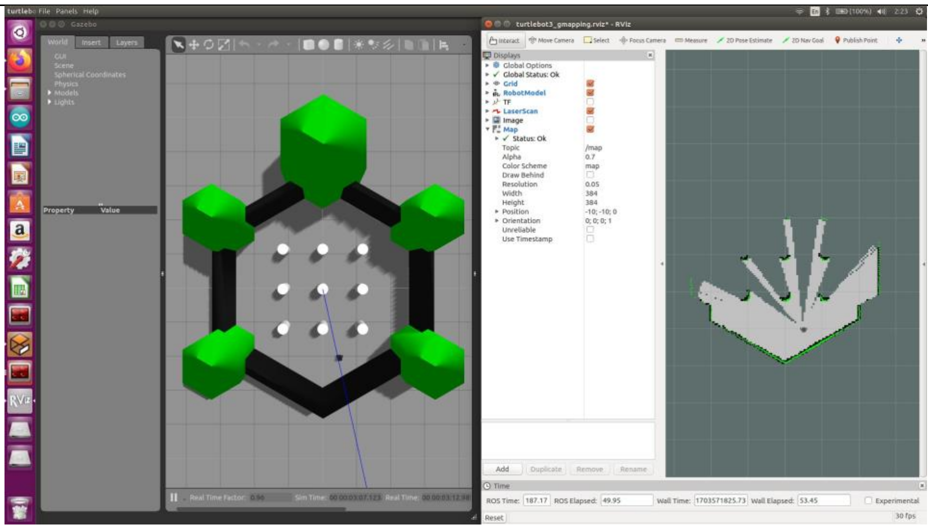
In this chapter, the results obtained will be tabulated and shown in graph. The results will also be discussed accordingly. A total of 6 experiments have been planned and conducted.

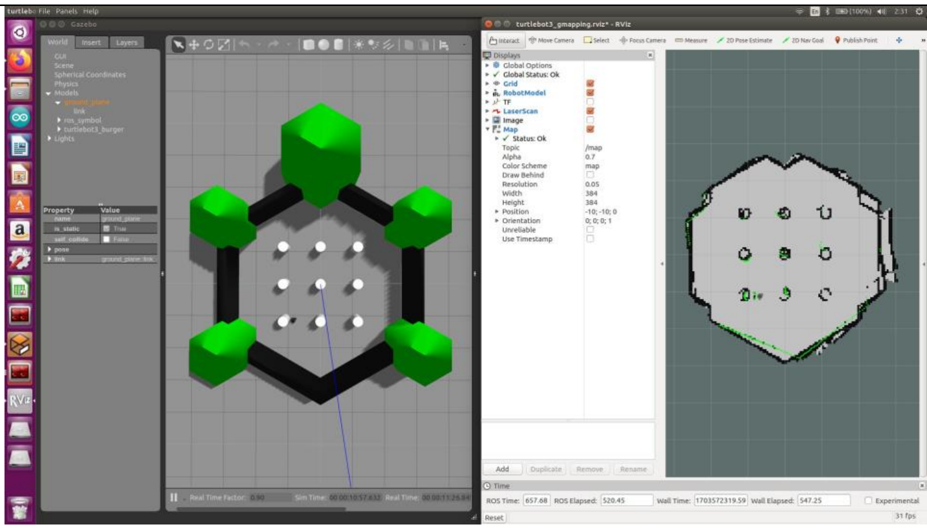
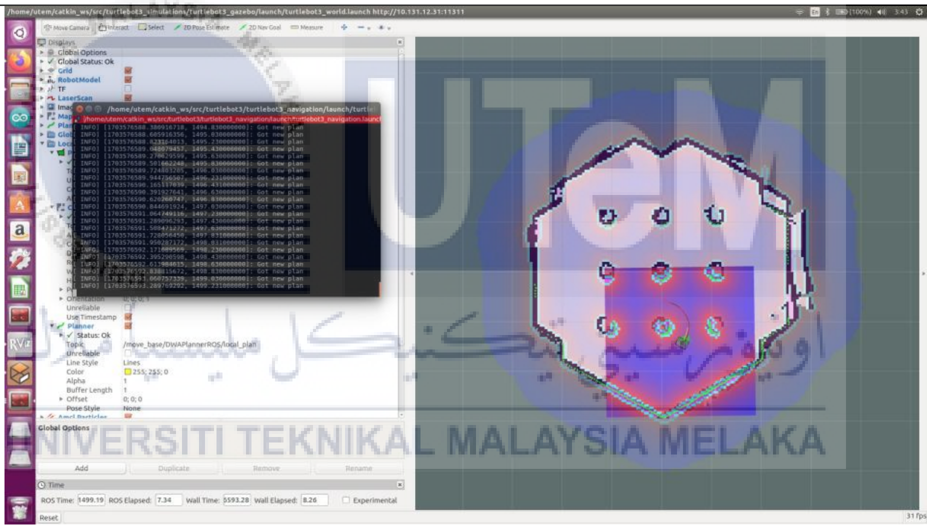
4.2 Results

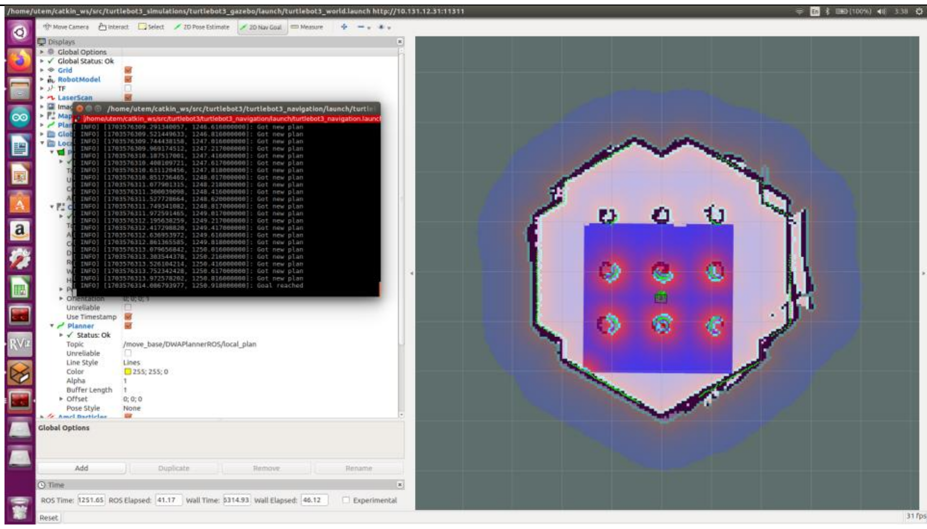
4.2.1 Experiment 1: Simulation of TurtleBot 3 Burger in the virtual world

In this experiment, the TurtleBot 3 Burger is simulated in a virtual environment, which is the TurtleBot 3 World, shown in ROS Gazebo. Rviz visualizes the process of map creation by the TurtleBot 3 Burger. The objective of this experiment is to understand the working principle of SLAM mapping and navigation of TurtleBot 3 Burger in the virtual world. The results of this experiment are shown in Table 4.1 below.

Table 4.1: Process of mapping and navigation by TurtleBot 3 Burger

 TurtleBot 3 Burger starts mapping process.

 TurtleBot 3 Burger done mapping process.
 TurtleBot 3 Burger starts navigation process.



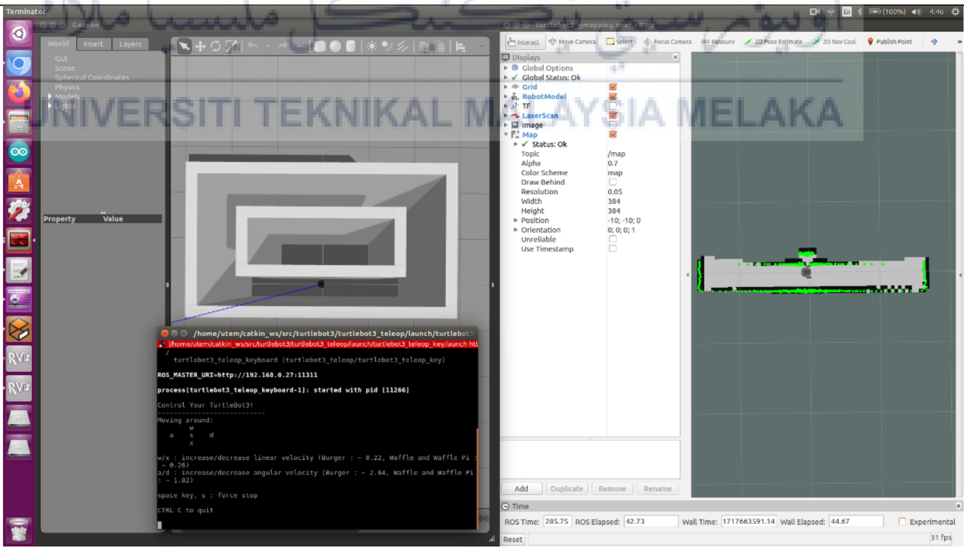
TurtleBot 3 Burger done navigation process.

Based on Table 4.1, it can be seen that the TurtleBot 3 Burger has completed the mapping process by moving and exploring around the environment using SLAM Gmapping and the teleoperation node. It is then able to navigate itself to the designated location on the map created earlier using the 2D Nav Goal function. The 2D Nav Goal function is used to set the target position and orientation for the TurtleBot. Based on the results, the TurtleBot 3 Burger is able to plan its own path and move from its starting position to the designated goal autonomously. Hence, objective 1 has been partially fulfilled.

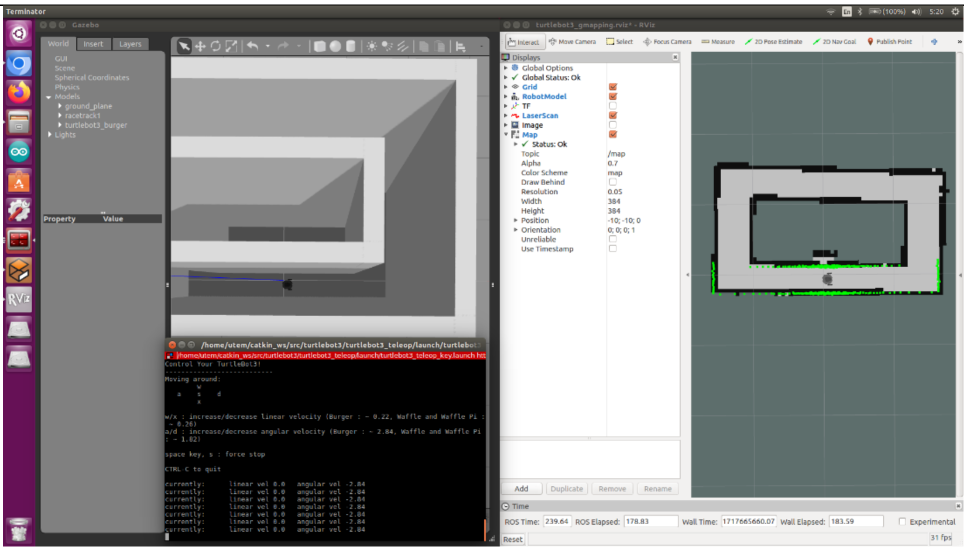
4.2.2 Experiment 2: Analysis of the performance of TurtleBot 3 Burger in virtual racetrack

In this experiment, the TurtleBot 3 Burger is simulated in the virtual racetrack created in ROS Gazebo. The racetrack in Gazebo is mapped by the TurtleBot using SLAM and is visualized in Rviz. Table 4.2 below shows the mapping process of the racetrack in Gazebo, visualized in Rviz. Figure 4.1 below shows the TurtleBot 3 Burger navigating around the racetrack, visualized in Rviz. The objective of this experiment is to ensure that the autonomous racing script is working properly and also to evaluate the baseline performance of the system which will be further improved in the upcoming experiments. This experiment is conducted to analyze the time taken for the TurtleBot to finish a lap and to analyze the graph of expected path against the actual path of its motion in the virtual racetrack. This is done by recording the time taken for the TurtleBot to finish a lap on the racetrack and calculating the RMSE between the expected and actual path for each lap after the autonomous racing script is run. A total of 5 repeated laps are completed by the TurtleBot to obtain the results. The results of this experiment are shown in Table 4.3 and 4.4 below.

Table 4.2: Mapping process of the racetrack by TurtleBot 3 Burger in Gazebo



TurtleBot 3 Burger starts mapping process.



TurtleBot 3 Burger done mapping process.

Figure 4.1: Navigation process of TurtleBot 3 Burger around the racetrack

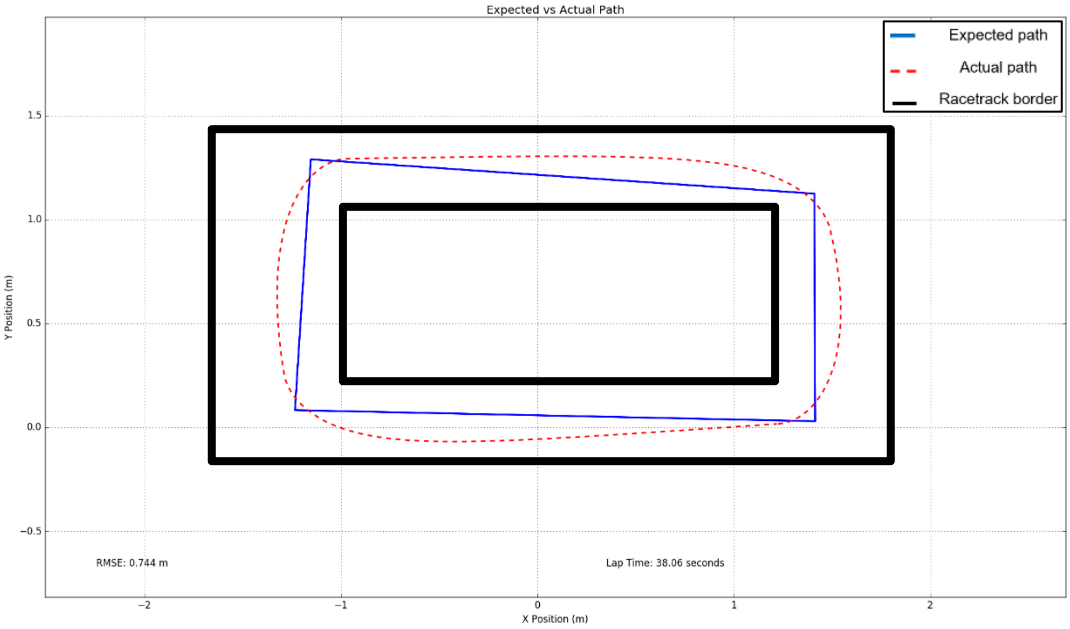
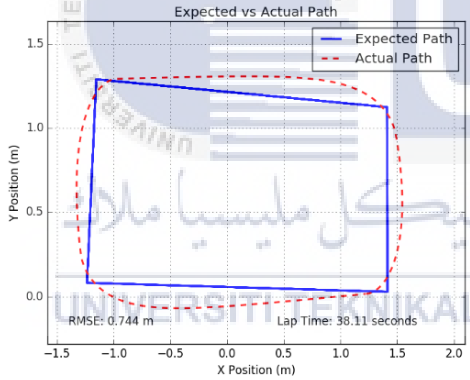
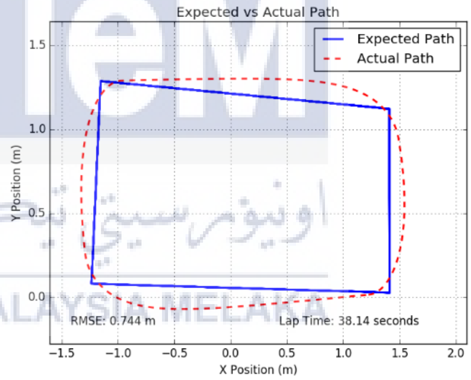
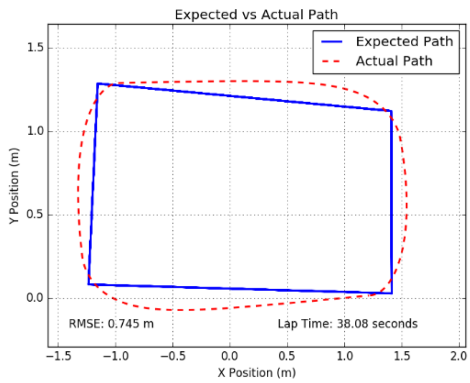
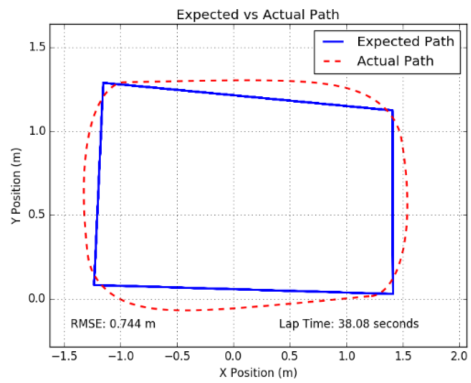
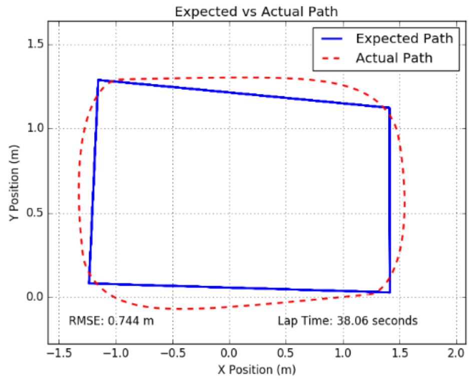


Figure 4.2: Sample graph of expected path against actual path

Table 4.3: Graph of expected path against actual path for each lap

 Lap 1	 Lap 2
 Lap 3	 Lap 4



Lap 5

Table 4.4: Lap time and RMSE for respective laps

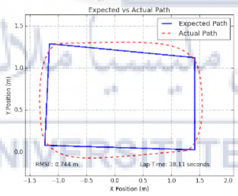
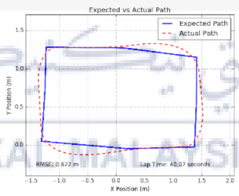
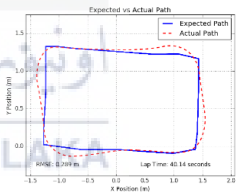
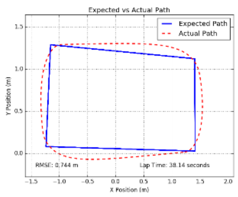
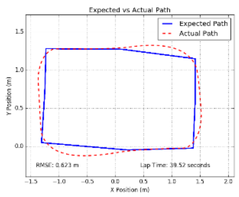
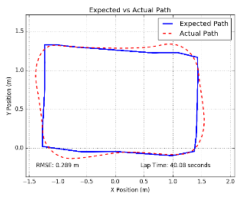
Lap	Lap time (seconds)	RMSE (meters)
1	38.11	0.744
2	38.14	0.744
3	38.08	0.745
4	38.08	0.744
5	38.06	0.744
Average	(38.11 + 38.14 + 38.08 + 38.08 + 38.06) / 5 = **38.0940**	(0.744 + 0.744 + 0.745 + 0.744 + 0.744) / 5 = **0.7442**

Based on Table 4.2, it shows that the map created by the TurtleBot 3 Burger is well defined with all the borders shown clearly on Rviz. Figure 4.1 shows that the TurtleBot is running autonomously on the map, which means the autonomous racing navigation script is working. Figure 4.2 displays the sample graph of expected path against actual path of the TurtleBot on the map with the racetrack border drawn. Based on Table 4.3, it can be seen that there are deviations between the expected path and the actual path of the TurtleBot’s motion on the racetrack. There are several factors that causes this deviation, such as dynamic and kinematic constraints, including wheel slippage and the TurtleBot's inertia and momentum. The TurtleBot’s path planning and execution issues, such as the chosen lookahead distance in the pure pursuit algorithm also contributes to this error. Table 4.4 shows that the time taken for the TurtleBot to complete a lap for lap 1 to 5 are approximately the same, with an average lap time of **38.0940 seconds**. A shorter lap time corresponds to a higher speed of navigation by the TurtleBot. Table 4.4 also shows that the RMSE values between the expected path and the actual path of the TurtleBot are approximately the same, with an RMSE of **0.7442 m**. The lower the RMSE value, the closer the actual path matches the expected path, indicating a higher accuracy in following the desired trajectory. The lap time and RMSE values are then used as a baseline for further improvements in the upcoming experiments. As a result of this experiment, objective 1, 2 and 3 have been fulfilled.

4.2.3 Experiment 3: Analysis of the performance of TurtleBot 3 Burger in virtual racetrack with different waypoint density levels

This experiment is conducted on the virtual racetrack similar to the one in Experiment 2. The objective of this experiment is to analyze the effect of different waypoint density levels on the time taken for the TurtleBot 3 Burger to finish a lap as well as the RMSE between the expected path and the actual path of its motion in the virtual racetrack. This is done by recording the time taken for the TurtleBot to finish a lap on the racetrack and calculating the RMSE between the expected and actual path for each lap after the autonomous racing script is run with 3 different levels of waypoint density. A total of 5 repeated laps are completed by the TurtleBot for each waypoint density level to obtain the results. The results of this experiment are shown in Table 4.5, 4.6 and 4.7 below. Figure 4.2, 4.3 and 4.4 visualizes the results in graphs.

Table 4.5: Graph of expected path against actual path for each waypoint density level

Lap	Waypoint density level		
	Low	Medium	High
1			
2			
3			

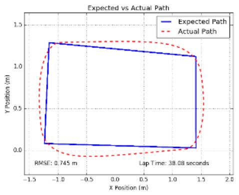
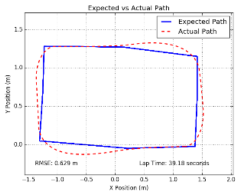
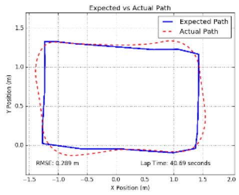
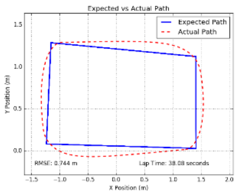
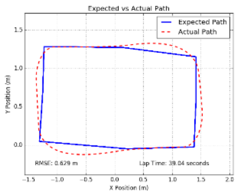
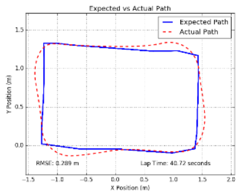
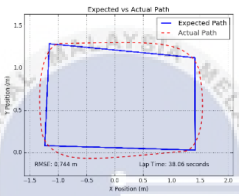
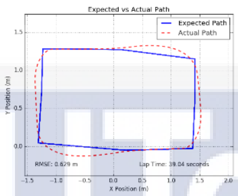
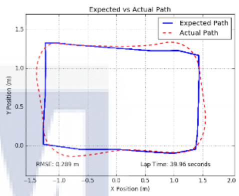
			
4			
5			

Table 4.6: Lap time and RMSE for respective laps of each waypoint density level

Waypoint density level	Lap	Lap time (seconds)	RMSE (meters)
Low	1	38.11	0.744
	2	38.14	0.744
	3	38.08	0.745
	4	38.08	0.744
	5	38.06	0.744
	Average	(38.11 + 38.14 + 38.08 + 38.08 + 38.06) / 5 = **38.0940**	(0.744 + 0.744 + 0.745 + 0.744 + 0.744) / 5 = **0.7442**
	1	40.07	0.622
	2	39.52	0.623

Medium	3	39.18	0.629
	4	39.04	0.629
	5	39.04	0.629
	Average	(40.07 + 39.52 + 39.18 + 39.04 + 39.04) / 5 = **39.3700**	(0.622 + 0.623 + 0.629 + 0.629 + 0.629) / 5 = **0.6264**
High	1	40.14	0.289
	2	40.08	0.289
	3	40.69	0.289
	4	40.72	0.289
	5	39.96	0.289
	Average	(40.14 + 40.08 + 40.69 + 40.72+ 39.96) / 5 = **40.3180**	(0.289 + 0.289 + 0.289 + 0.289 + 0.289) / 5 = **0.2890**

Table 4.7: Percentage of decrease in lap time and RMSE compared to last experiment

Waypoint density level	Percentage of decrease in lap time (%)	Percentage of decrease in RMSE (%)
Low	[(38.0940 - 38.0940) / 38.0940] x 100 = **0**	[(0.7442 – 0.7442) / 0.7442] x 100 = **0**
Medium	[(38.0940 - 39.3700) / 38.0940] x 100 = **-3.3496**	[(0.7442 – 0.6264) / 0.7442] x 100 = **15.8291**
High	[(38.0940 - 40.3180) / 38.0940] x 100 = **-5.8382**	[(0.7442 – 0.2890) / 0.7442] x 100 = **61.1664**

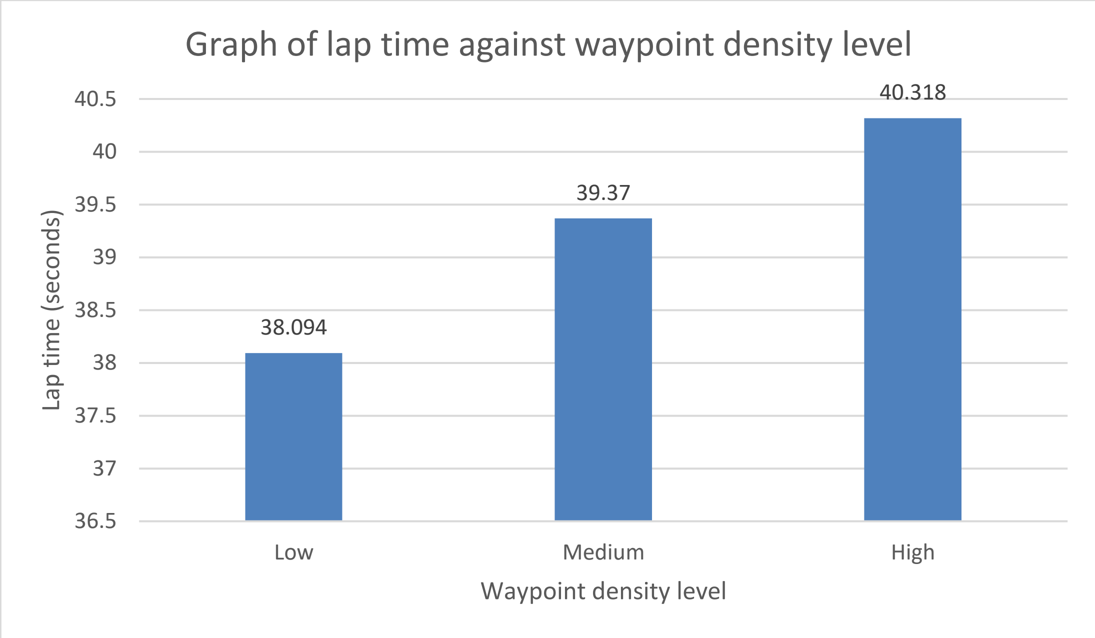


Figure 4.3: Graph of lap time against waypoint density level

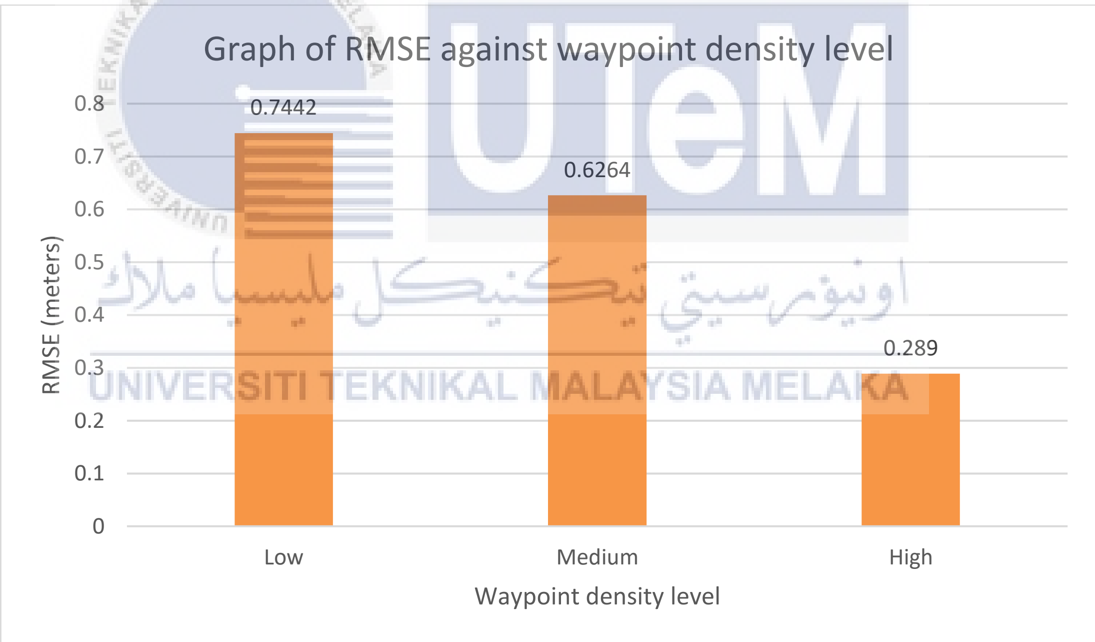


Figure 4.4: Graph of RMSE against waypoint density level

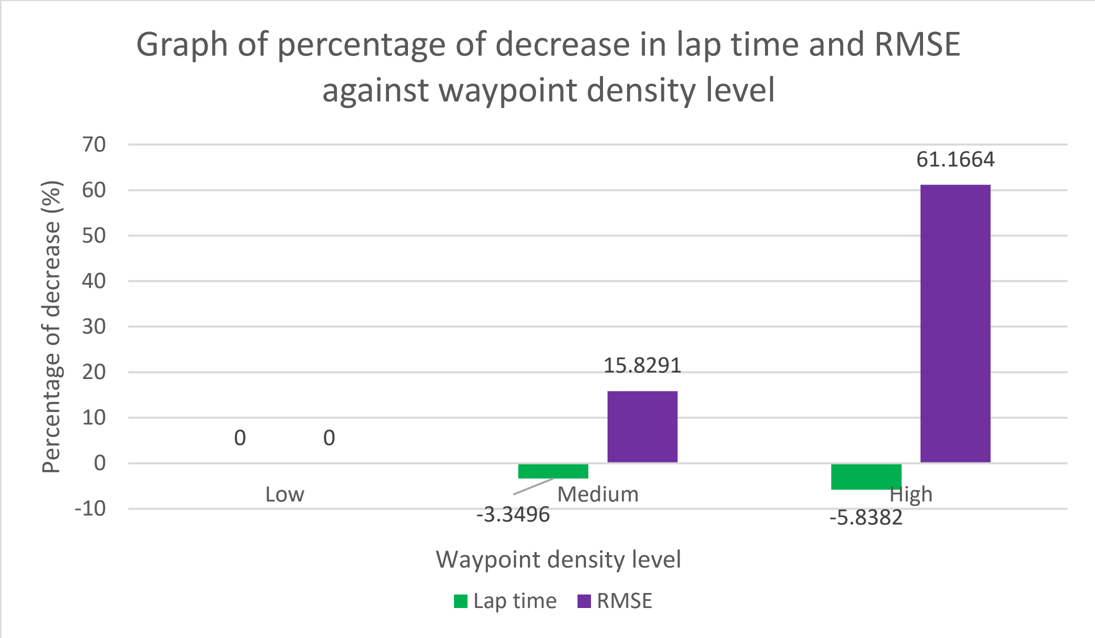


Figure 4.5: Graph of percentage of decrease in lap time and RMSE against waypoint density level

Based on Table 4.5, it can be seen that there are deviations between the expected path and the actual path of the TurtleBot's motion on the racetrack. Just like in previous experiment, it may be caused by wheel slippage, the TurtleBot's inertia and momentum as well as the chosen lookahead distance in the pure pursuit algorithm. A waypoint is a set of coordinates that the TurtleBot follows to navigate the track. The waypoints guide the TurtleBot's trajectory, helping it to adjust its speed and steering to optimize lap time and avoid obstacles. For this experiment, the waypoints around the map of the racetrack are increased and separated into three density levels, namely low, medium and high.

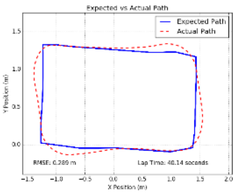
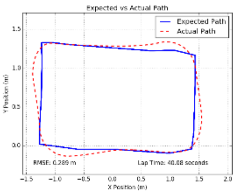
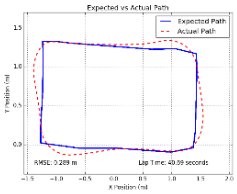
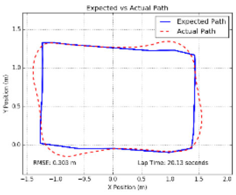
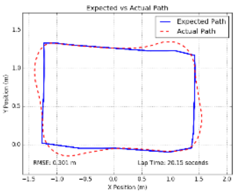
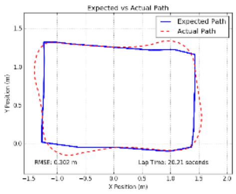
Table 4.6 and Figure 4.2 show that the average time taken for the TurtleBot to complete a lap for each waypoint density level are approximately the same, where the percentage of decrease in lap time as compared to the previous experiment are 0%, -3.3496% and -5.8382% for low, medium and high waypoint density level respectively as shown in Table 4.7 and Figure 4.4. The negative percentage values indicate that the lap time increased compared to the one in the previous experiment. This shows that the difference in waypoint densities does not really affect the lap time.

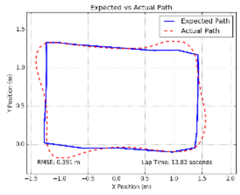
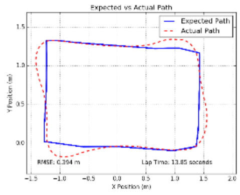
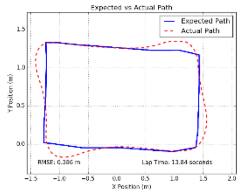
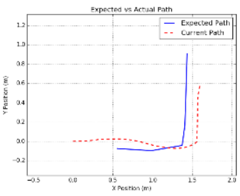
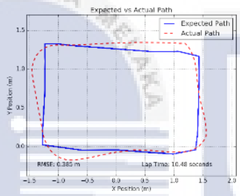
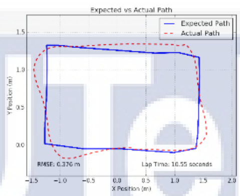
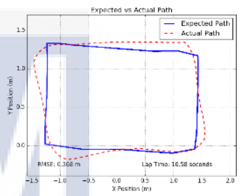
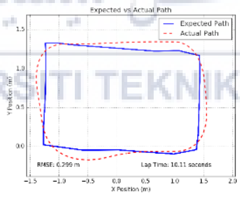
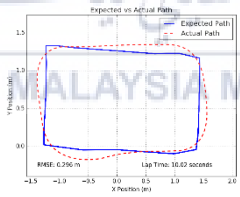
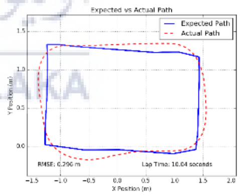
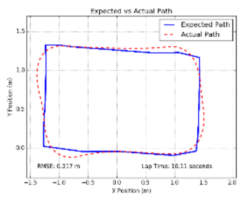
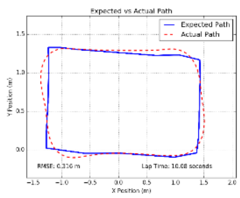
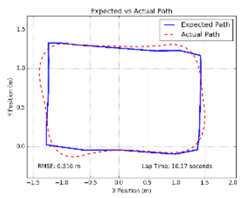
Table 4.6 and Figure 4.3 show that the average RMSE value between the expected path and the actual path of the TurtleBot decreases from low to high waypoint density level, where the percentage of decrease in RMSE as compared to the previous experiment are 0%, 15.8291% and 61.1664% respectively as shown in Table 4.7 and Figure 4.4. From here, we can see that the lap time slightly increases but the RMSE decreases when the waypoint density increases. This is because when the waypoint density increases, the TurtleBot follows a more accurate path (low RMSE) due to more frequent reference points for trajectory correction. However, this can slightly increase the lap time due to more frequent adjustments, computational overhead, and cautious movement around sharp turns. This experiment concludes that the increase in waypoint density can significantly improve the RMSE between the expected and actual path lines. Although there is a slight trade-off between the lap time and RMSE, the minor increase in lap time (-5.8382%) can be neglected, since the RMSE can be greatly reduced (61.1664%) on the flip side. Overall, a high waypoint density produced the best result with an average lap time of **40.3180 seconds** and average RMSE of **0.2890 m**. Hence, the **high waypoint density** is chosen to be conducted in the upcoming experiments for further improvements. As a result of this experiment, objective 2 and 3 have been fulfilled.

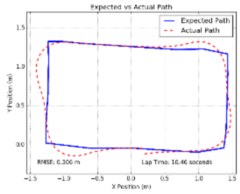
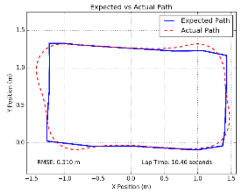
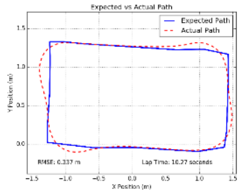
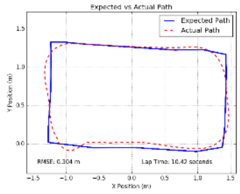
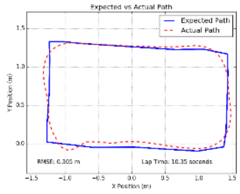
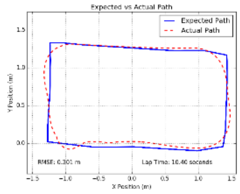
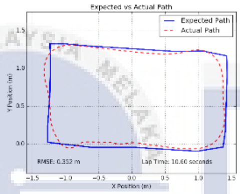
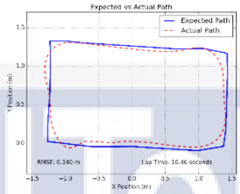
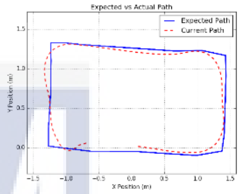
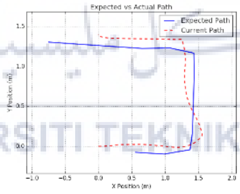
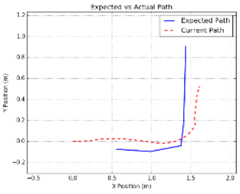
4.2.4 Experiment 4: Analysis of the performance of TurtleBot 3 Burger in virtual racetrack with varying linear velocities, lookahead distances, and angular velocity proportional gains

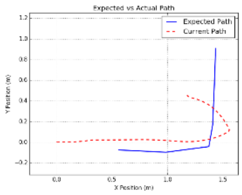
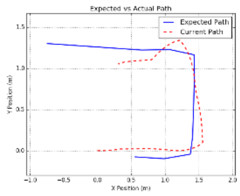
This experiment is conducted on the virtual racetrack similar to the one in Experiment 2. The objective of this experiment is to analyze the effect of varying linear velocities, lookahead distances and angular velocity proportional gains on the TurtleBot 3 Burger's behavior in terms of its ability to complete the lap, the time taken for it to finish a lap as well as the RMSE between the expected path and the actual path of its motion in the virtual racetrack. This is done by recording the time taken for the TurtleBot to finish a lap on the racetrack and calculating the RMSE between the expected and actual path for each lap after the autonomous racing script is run with multiple combinations of linear velocities, lookahead distances and angular velocity proportional gains. 3 repeated laps are completed by the TurtleBot for each combination of the respective parameters to obtain the results. The results of this experiment are shown in Table 4.8, 4.9 and 4.10 below. Figure 4.5, 4.6 and 4.7 visualizes the results in graphs.

Table 4.8: Graph of expected path against actual path for each parameter combination

<table>
<tr><th>Parameters</th><th>Lap 1</th><th>Lap 2</th><th>Lap 3</th></tr>
<tr><td>LV = 0.2 m/s
LD = 0.2 m
Kp = 0.7</td>
<td>
</td>
<td>
</td>
<td>
</td></tr>
<tr><td>LV = 0.4 m/s
LD = 0.2 m
Kp = 1.4</td>
<td>
</td>
<td>
</td>
<td>
</td></tr>
</table>

LV = 0.6 m/s LD = 0.2 m Kp = 2			
LV = 0.8 m/s LD = 0.2 m Kp = 2		-	-
LV = 0.8 m/s LD = 0.3 m Kp = 2			
LV = 0.8 m/s LD = 0.4 m Kp = 2			
LV = 0.8 m/s LD = 0.4 m Kp = 2.5			
LV = 0.8 m/s LD = 0.4 m			

Kp = 3			
LV = 0.8 m/s LD = 0.45 m Kp = 2.5			
LV = 0.8 m/s LD = 0.5 m Kp = 3			
LV = 0.9 m/s LD = 0.4 m Kp = 3		-	-
LV = 0.9 m/s LD = 0.4 m Kp = 3.5		-	-
LV = 1 m/s LD = 0.4 m Kp = 3		-	-

			
LV = 1 m/s LD = 0.4 m Kp = 4		-	-

* LV = Linear velocity

LD = Lookahead distance

Kp = Angular velocity proportional gain

Table 4.9: Lap time, RMSE and observation for respective laps of each parameter combination

LV (m/s)	LD (m)	Kp	Lap	Lap time (s)	RMSE (m)	Observation
0.2	0.2	0.7	1	40.14	0.289	Completed all laps
			2	40.08	0.289	
			3	40.69	0.289	
			Average	(40.14 + 40.08 + 40.69) / 3 = **40.3033**	(0.289 + 0.289 + 0.289) / 3 = **0.2890**	
0.4	0.2	1.4	1	20.13	0.303	Completed all laps
			2	20.15	0.301	
			3	20.21	0.302	
			Average	(20.13 + 20.15 + 20.21) / 3 = **20.1633**	(0.303 + 0.301 + 0.302) / 3 = **0.3020**	
			1	13.83	0.391	
			2	13.85	0.394	

0.6	0.2	2	3	13.84	0.386	Completed all laps
			Average	(13.83 + 13.85 + 13.84) / 3 = **13.8400**	0.391 + 0.394 + 0.386) / 3 = **0.3903**	
0.8	0.2	2	1	-	-	Crashed on the first lap
			2			
			3			
			Average			
0.8	0.3	2	1	10.48	0.385	Completed all laps but touched border
			2	10.55	0.376	
			3	10.58	0.368	
			Average	(10.48 + 10.55 + 10.58) / 3 = **10.5367**	(0.385 + 0.376 + 0.368) / 3 = **0.3763**	
0.8	0.4	2	1	10.11	0.299	Completed all laps but touched border
			2	10.02	0.296	
			3	10.04	0.296	
			Average	(10.11 + 10.02 + 10.04) / 3 = **10.0567**	(0.299 + 0.296 + 0.296) / 3 = **0.2970**	
0.8	0.4	2.5	1	10.11	0.317	Completed all laps (best)
			2	10.08	0.316	
			3	10.17	0.316	
			Average	(10.11 + 10.08 + 10.17) / 3 = **10.1200**	(0.317 + 0.316 + 0.316) / 3 = **0.3163**	
0.8	0.4	3	1	10.46	0.306	Completed all laps but unstable
			2	10.46	0.310	
			3	10.27	0.337	
			Average	(10.46 + 10.46 + 10.27) / 3 = **10.3967**	0.306 + 0.310 + 0.337) / 3 = **0.3177**	

0.8	0.45	2.5	1	10.42	0.304	Completed all laps but nearly crashed
			2	10.35	0.305	
			3	10.40	0.301	
			Average	(10.42 + 10.35 + 10.40) / 3 = **10.3900**	(0.304 + 0.305 + 0.301) / 3 = **0.3033**	
0.8	0.5	3	1	10.60	0.352	Crashed on the last lap
			2	10.46	0.340	
			3	-	-	
			Average	-	-	
0.9	0.4	3	1	-	-	Crashed on the first lap
			2			
			3			
			Average			
0.9	0.4	3.5	1	-	-	Crashed on the first lap
			2			
			3			
			Average			
1	0.4	3	1	-	-	Crashed on the first lap
			2			
			3			
			Average			
1	0.4	4	1	-	-	Crashed on the first lap
			2			
			3			
			Average			

* LV = Linear velocity

LD = Lookahead distance

Kp = Angular velocity proportional gain

Table 4.10: Percentage of decrease in lap time and RMSE compared to last experiment

LV (m/s)	LD (m)	Kp	Percentage of decrease in lap time (%)	Percentage of decrease in RMSE (%)
0.2	0.2	0.7	[(40.3180 – 40.3033) / 40.3180] x 100 = **0.0365**	[(0.2890 - 0.2890) / 0.2890] x 100 = **0**
0.4	0.2	1.4	[(40.3180 – 20.1633) / 40.3180] x 100 = **49.9893**	[(0.2890 - 0.3020) / 0.2890] x 100 = **-4.4983**
0.6	0.2	2	[(40.3180 – 13.8400) / 40.3180] x 100 = **65.6729**	[(0.2890 - 0.3903) / 0.2890] x 100 = **-35.0519**
0.8	0.2	2	-	-
0.8	0.3	2	[(40.3180 – 10.5367) / 40.3180] x 100 = **73.8660**	[(0.2890 - 0.3763) / 0.2890] x 100 = **-30.2076**
0.8	0.4	2	[(40.3180 – 10.0567) / 40.3180] x 100 = **75.0566**	[(0.2890 - 0.2970) / 0.2890] x 100 = **-2.7682**
0.8	0.4	2.5	[(40.3180 – 10.1200) / 40.3180] x 100 = **74.8995**	[(0.2890 - 0.3163) / 0.2890] x 100 = **-9.4464**
0.8	0.4	3	[(40.3180 – 10.3967) / 40.3180] x 100 = **74.2133**	[(0.2890 - 0.3177) / 0.2890] x 100 = **-9.9308**
0.8	0.45	2.5	[(40.3180 – 10.3900) / 40.3180] x 100 = **74.2299**	[(0.2890 - 0.3033) / 0.2890] x 100 = **-4.9481**
0.8	0.5	3	-	-
0.9	0.4	3	-	-
0.9	0.4	3.5	-	-
1	0.4	3	-	-

1	0.4	4	-	-

* LV = Linear velocity

LD = Lookahead distance

Kp = Angular velocity proportional gain

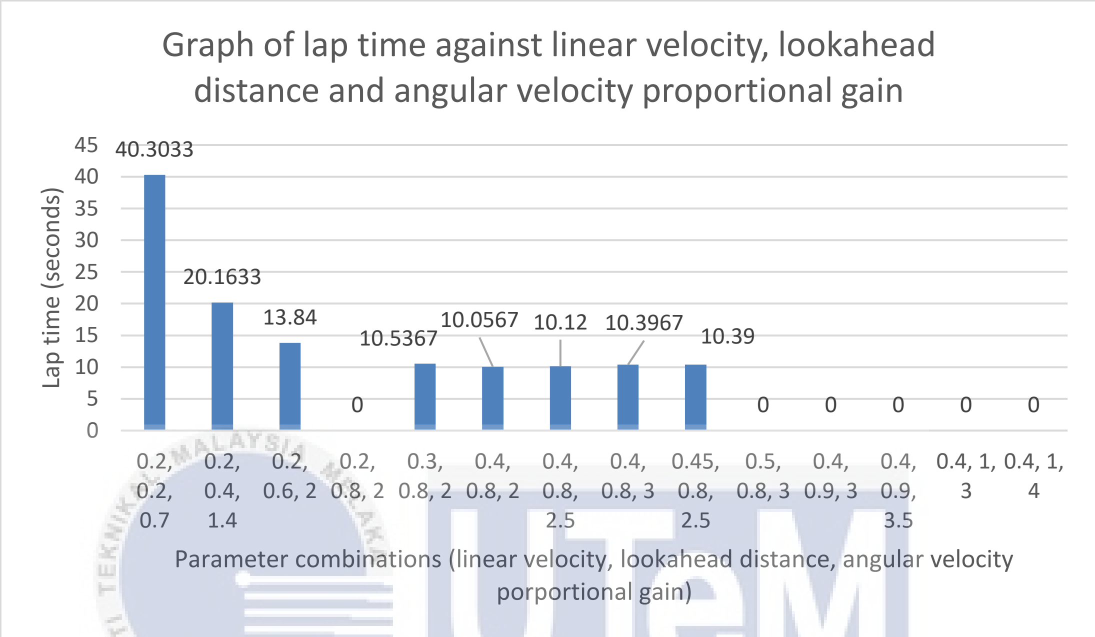


Figure 4.6: Graph of lap time against linear velocity, lookahead distance, and angular velocity proportional gain

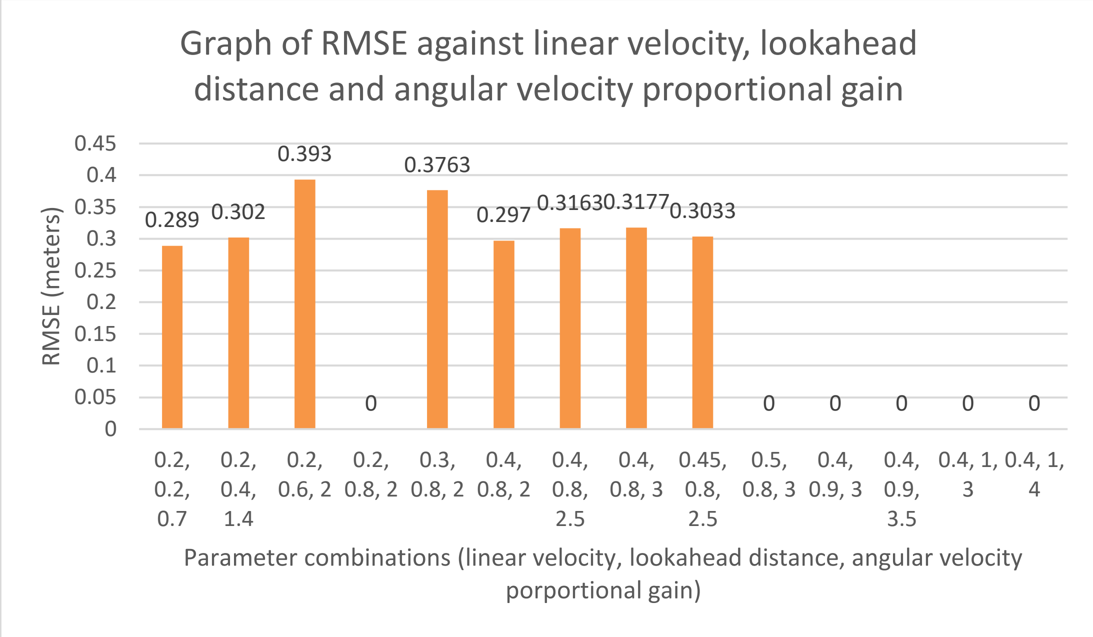


Figure 4.7: Graph of RMSE against linear velocity, lookahead distance and angular velocity proportional gain

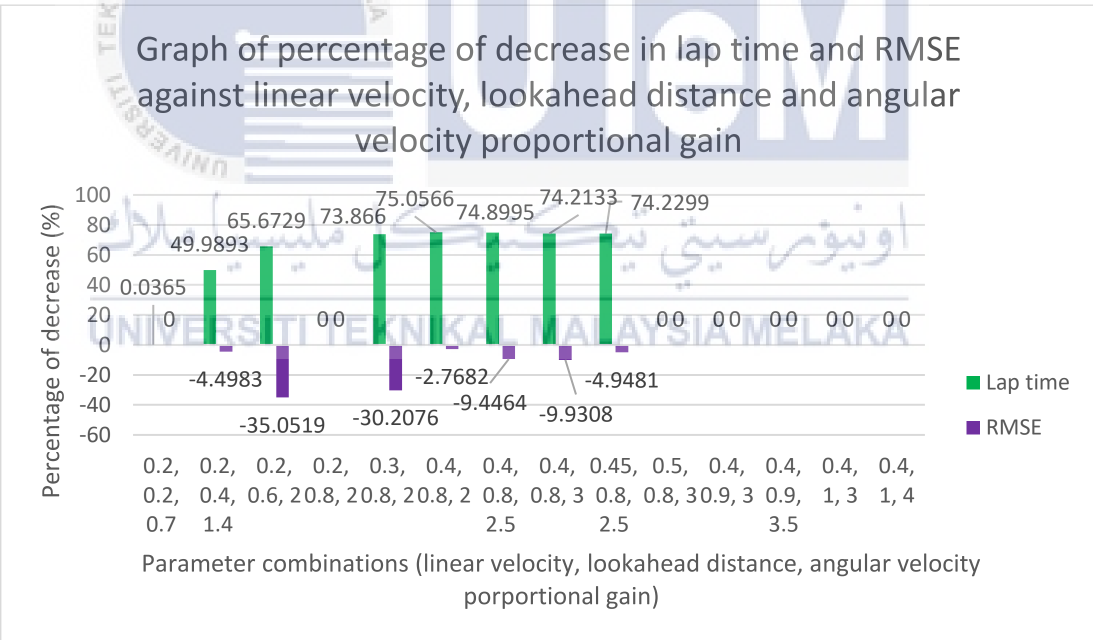


Figure 4.8: Graph of percentage of decrease in lap time and RMSE against linear velocity, lookahead distance and angular velocity proportional gain

Based on Table 4.8, it can be seen that there are deviations between the expected path and the actual path of the TurtleBot’s motion on the racetrack. Just like in previous experiments, it may be caused by wheel slippage, the TurtleBot's inertia and momentum as well as the chosen lookahead distance in the pure pursuit algorithm.

The linear velocity is the forward speed of the TurtleBot, measured in meters per second. The lookahead distance, measured in meters, is the distance ahead of the TurtleBot where it aims to move towards, ensuring smooth and accurate path following. Angular velocity proportional gain (Kp) determines the TurtleBot's turning response to orientation errors. It balances responsive turning with stability. These pure pursuit parameters are interdependent and can vary depending on various factors such as the size of racetrack. This means that the parameters must be fine-tuned among each other to maintain stable and accurate path tracking. If one of the parameter values is too high or too low, it might affect the TurtleBot's performance.

For this experiment, the linear velocity of the TurtleBot is gradually increased while adjusting the lookahead distance and the angular velocity proportional gain in a trial-and-error way to analyze and optimize the TurtleBot's performance. Table 4.9 and Figure 4.5 show that the average time taken for the TurtleBot to complete a lap decreases when its linear velocity increases. This also leads to an increase in the percentage of decrease in lap time compared to the previous experiment as shown in Table 4.10 and Figure 4.7. For some parameter combinations, the TurtleBot touched or crashed against the border because the chosen lookahead distance and angular velocity proportional gain were not appropriate at certain linear velocities. When the TurtleBot moves faster, it requires a larger lookahead distance to anticipate and react to upcoming waypoints effectively, otherwise it may be unstable in its motion. The angular velocity gain must also be fine-tuned to prevent abrupt changes in direction that can lead to collisions.

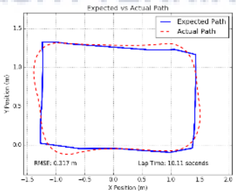
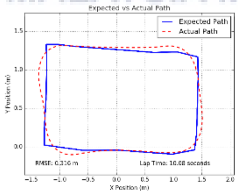
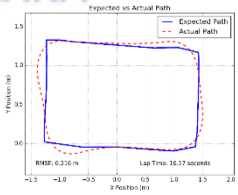
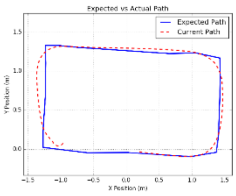
Table 4.9 and Figure 4.5 show that the average RMSE between the expected and actual path of the TurtleBot are approximately the same. The percentage of decrease in lap time compared to the previous experiment are also approximately the same as shown in Table 4.10 and Figure 4.7. The negative percentage values indicate that the lap time increased compared to the one in the previous experiment. This experiment proves that the increase in linear velocity of the TurtleBot does not significantly affect the RMSE but can greatly improve the lap time, provided that the lookahead distance and the angular velocity proportional gain must be tuned properly. Although there is a slight trade-off between the lap time and RMSE, the minor increase in RMSE (-9.4464%) can be neglected, since the lap time can be greatly reduced

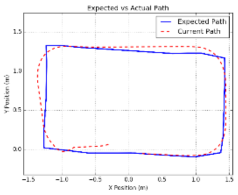
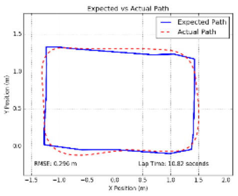
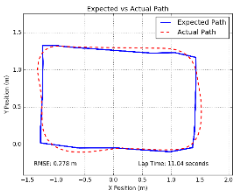
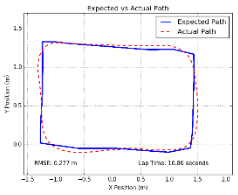
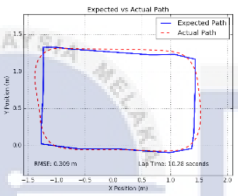
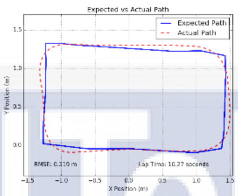
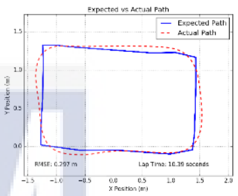
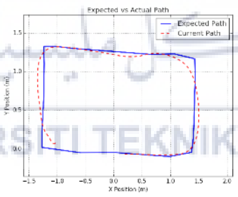
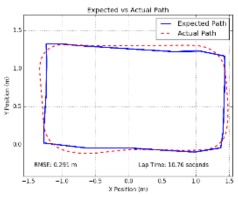
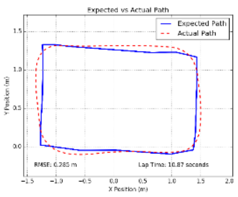
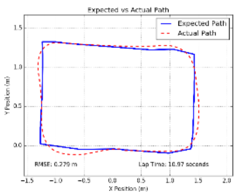
(74.8995%) on the flip side. Overall, the linear velocity of **0.8 m/s**, lookahead distance of **0.4 m** and angular velocity proportional gain of **2.5** produced the best result without crashing or touching the border with an average lap time of **10.1200 seconds** and average RMSE of **0.3163 m**. Hence, these parameter values are chosen to be conducted in the upcoming experiments for further improvements. As a result of this experiment, objective 2 and 3 have been fulfilled.

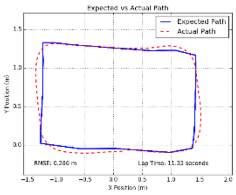
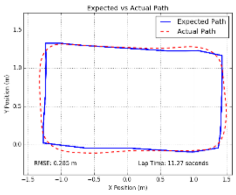
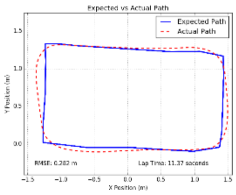
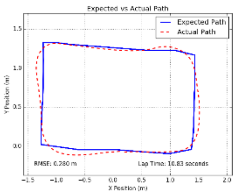
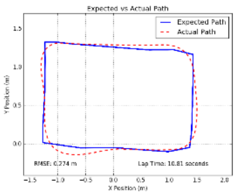
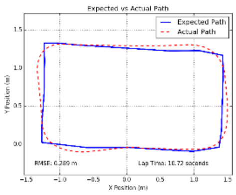
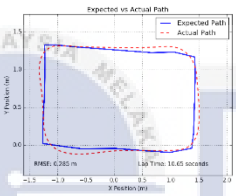
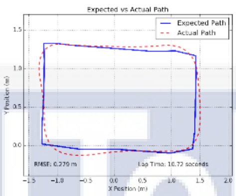
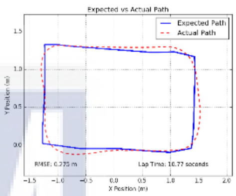
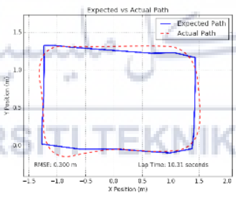
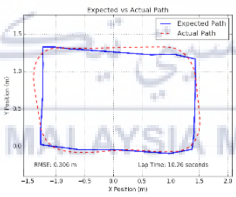
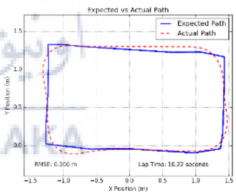
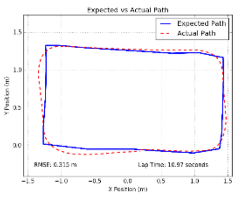
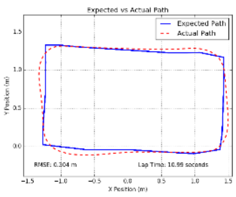
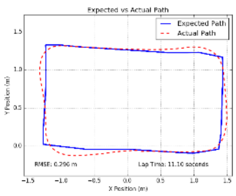
4.2.5 Experiment 5: Analysis of the performance of TurtleBot 3 Burger in real world racetrack with varying angle thresholds and linear velocity reduction factors

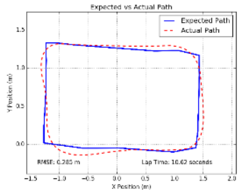
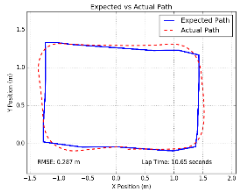
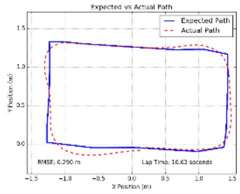
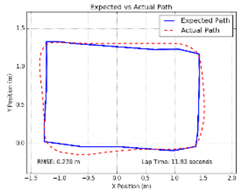
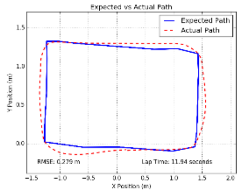
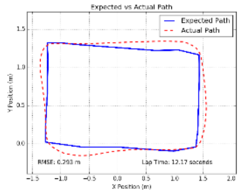
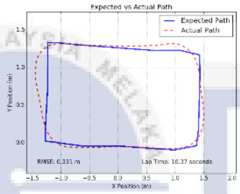
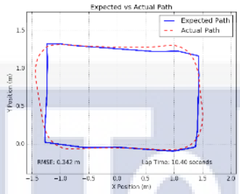
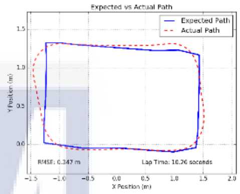
This experiment is conducted on the virtual racetrack similar to the one in Experiment 2. The objective of this experiment is to analyze the effect of varying angle thresholds and linear velocity reduction factors on the TurtleBot 3 Burger’s behavior in terms of its ability to complete the lap, the time taken for it to finish a lap as well as the RMSE between the expected path and the actual path of its motion in the virtual racetrack. This is done by recording the time taken for the TurtleBot to finish a lap on the racetrack and calculating the RMSE between the expected and actual path for each lap after the autonomous racing script is run with multiple combinations angle thresholds and linear velocity reduction factors. 3 repeated laps are completed by the TurtleBot for each combination of the respective parameters to obtain the results. The results of this experiment are shown in Table 4.11, 4.12 and 4.13 below. Figure 4.8, 4.9 and 4.10 visualizes the results in graphs.

Table 4.11: Graph of expected path against actual path for each parameter combination

Parameters	Lap 1	Lap 2	Lap 3
AT = 0 rad RF = 1			
AT = 0.05 rad RF = 0.8		-	-

AT = 0.05 rad RF = 0.9		-	-
AT = 0.09 rad RF = 0.8			
AT = 0.09 rad RF = 0.9			
AT = 0.1 rad RF = 0.7		-	-
AT = 0.1 rad RF = 0.8			
AT = 0.15 rad RF = 0.7			

			
AT = 0.15 rad RF = 0.8			
AT = 0.2 rad RF = 0.8			
AT = 0.2 rad RF = 0.9			
AT = 0.3 rad RF = 0.7			
AT = 0.3 rad RF = 0.8			

			
AT = 0.5 rad RF = 0.5			
AT = 1 rad RF = 0.5			

* AT = Angle threshold

RF = Linear velocity reduction factor

Table 4.12: Lap time, RMSE and observation for respective laps of each parameter combination

AT (rad)	RF	Lap	Lap time (s)	RMSE (m)	Observation
0	1	1	10.11	0.317	Completed all laps
		2	10.08	0.316	
		3	10.17	0.316	
		Average	(10.11 + 10.08 + 10.17) / 3 = **10.1200**	(0.317 + 0.316 + 0.316) / 3 = **0.3163**	
0.05	0.8	1	-	-	Crashed on the first lap
		2			
		3			

		Average			
0.05	0.9	1	-	-	Crashed on the first lap
		2			
		3			
		Average			
0.09	0.8	1	10.82	0.296	Completed all laps
		2	11.04	0.278	
		3	10.86	0.277	
		Average	(10.82 + 11.04 + 10.86) / 3 = **10.9067**	(0.296 + 0.278 + 0.277) / 3 = **0.2837**	
0.09	0.9	1	10.28	0.309	Completed all laps
		2	10.27	0.319	
		3	10.39	0.297	
		Average	(10.28 + 10.27 + 10.39) / 3 = **10.3133**	(0.309 + 0.319 + 0.297) / 3 = **0.3083**	
0.1	0.7	1	-	-	Crashed on the first lap
		2			
		3			
		Average			
0.1	0.8	1	10.76	0.291	Completed all laps
		2	10.87	0.285	
		3	10.97	0.279	
		Average	(10.76 + 10.87 + 10.97) / 3 = **10.8667**	(0.291 + 0.285 + 0.279) / 3 = **0.2850**	
0.15	0.7	1	11.33	0.286	Completed all laps but unstable
		2	11.27	0.285	
		3	11.37	0.282	
		Average	(11.33 + 11.27 + 11.37) / 3	(0.286 + 0.285 + 0.282) / 3	

			= **11.3233**	= **0.2843**	
0.15	0.8	1	10.83	0.280	Completed all laps
		2	10.81	0.274	
		3	10.72	0.289	
		Average	(10.83 + 10.81 + 10.72) / 3 = **10.7867**	(0.280 + 0.274 + 0.289) / 3 = **0.2810**	
0.2	0.8	1	10.65	0.285	Completed all laps (best)
		2	10.72	0.279	
		3	10.77	0.275	
		Average	(10.65 + 10.72 + 10.77) / 3 = **10.7133**	(0.285 + 0.279 + 0.275) / 3 = **0.2797**	
0.2	0.9	1	10.31	0.300	Completed all laps
		2	10.26	0.306	
		3	10.22	0.306	
		Average	(10.31 + 10.26 + 10.22) / 3 = **10.2633**	(0.300 + 0.306 + 0.306) / 3 = **0.3040**	
0.3	0.7	1	10.97	0.315	Completed all laps
		2	10.99	0.304	
		3	11.10	0.296	
		Average	(10.97 + 10.99 + 11.10) / 3 = **11.0200**	(0.315 + 0.304 + 0.296) / 3 = **0.3050**	
0.3	0.8	1	10.62	0.285	Completed all laps
		2	10.65	0.287	
		3	10.63	0.290	
		Average	(10.62 + 10.65 + 10.63) / 3 = **10.6333**	(0.285 + 0.287 + 0.290) / 3 = **0.2873**	
		1	11.93	0.278	

0.5	0.5	2	11.94	0.279	Completed all laps
		3	12.17	0.293	
		Average	(11.93 + 11.94 + 12.17) / 3 = **12.0133**	(0.278 + 0.279 + 0.293) / 3 = **0.2833**	
1	0.5	1	10.37	0.331	Completed all laps
		2	10.40	0.342	
		3	10.26	0.347	
		Average	(10.37 + 10.40 + 10.26) / 3 = **10.3433**	(0.331 + 0.342 + 0.347) / 3 = **0.3400**	

* AT = Angle threshold

RF = Linear velocity reduction factor

Table 4.13: Percentage of decrease in lap time and RMSE compared to last experiment

AT (rad)	RF	Percentage of decrease in lap time (%)	Percentage of decrease in RMSE (%)
0	1	[(10.1200 – 10.1200) / 10.1200] x 100 = **0**	[(0.3163 – 0.3163) / 0.3163] x 100 = **0**
0.05	0.8	-	-
0.05	0.9	-	-
0.09	0.8	[(10.1200 – 10.9067) / 10.1200] x 100 = **-7.7737**	[(0.3163 – 0.2837) / 0.3163] x 100 = **10.3067**
0.09	0.9	[(10.1200 – 10.3133) / 10.1200] x 100 = **-1.9101**	[(0.3163 – 0.3083) / 0.3163] x 100 = **2.5292**
0.1	0.7	-	-
0.1	0.8	[(10.1200 – 10.8667) / 10.1200] x 100 = **-7.3785**	[(0.3163 – 0.2850) / 0.3163] x 100 = **9.8957**
0.15	0.7	[(10.1200 – 11.3233) / 10.1200] x 100 = **-11.8903**	[(0.3163 – 0.2843) / 0.3163] x 100 = **10.1170**
0.15	0.8	[(10.1200 – 10.7867) / 10.1200] x 100 = **-6.5879**	[(0.3163 – 0.2810) / 0.3163] x 100 = **11.1603**

0.2	0.8	[(10.1200 – 10.7133) / 10.1200] x 100 = **-5.8626**	[(0.3163 – 0.2797) / 0.3163] x 100 = **11.5713**
0.2	0.9	[(10.1200 – 10.2633) / 10.1200] x 100 = **-1.4160**	[(0.3163 – 0.3040) / 0.3163] x 100 = **3.8887**
0.3	0.7	[(10.1200 – 11.0200) / 10.1200] x 100 = **-8.8933**	[(0.3163 – 0.3050) / 0.3163] x 100 = **3.5726**
0.3	0.8	[(10.1200 – 10.6333) / 10.1200] x 100 = **-5.0721**	[(0.3163 – 0.2873) / 0.3163] x 100 = **9.1685**
0.5	0.5	[(10.1200 – 12.0133) / 10.1200] x 100 = **-18.7085**	[(0.3163 – 0.2833) / 0.3163] x 100 = **10.4331**
1	0.5	[(10.1200 – 10.3433) / 10.1200] x 100 = **-2.2065**	[(0.3163 – 0.3400) / 0.3163] x 100 = **-7.4929**

* AT = Angle threshold

RF = Linear velocity reduction factor

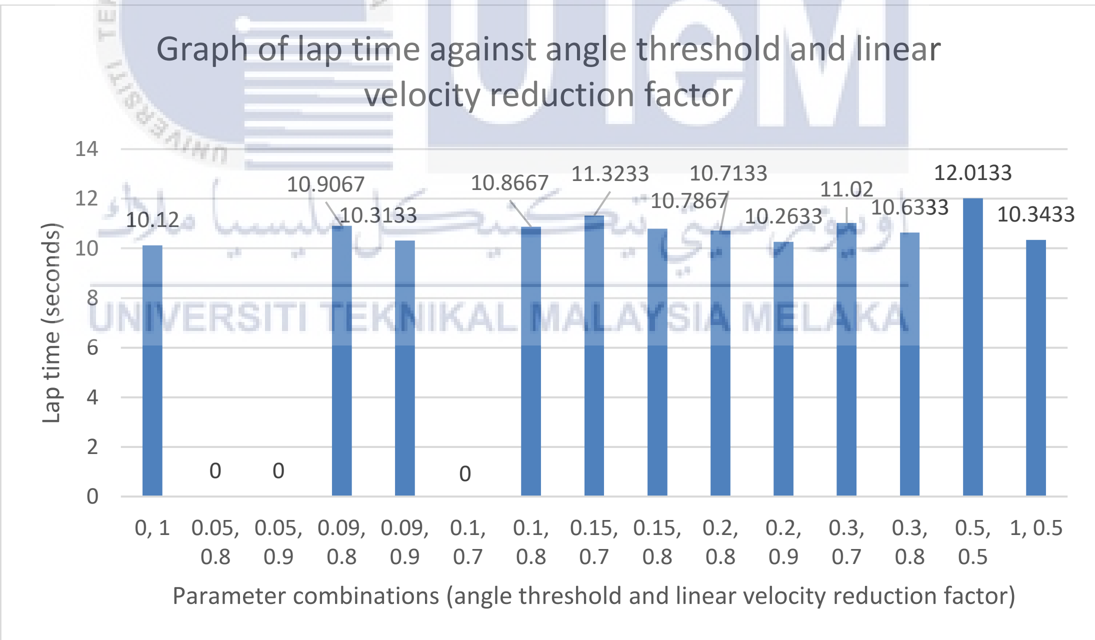


Figure 4.9: Graph of lap time against angle threshold and linear velocity reduction factor

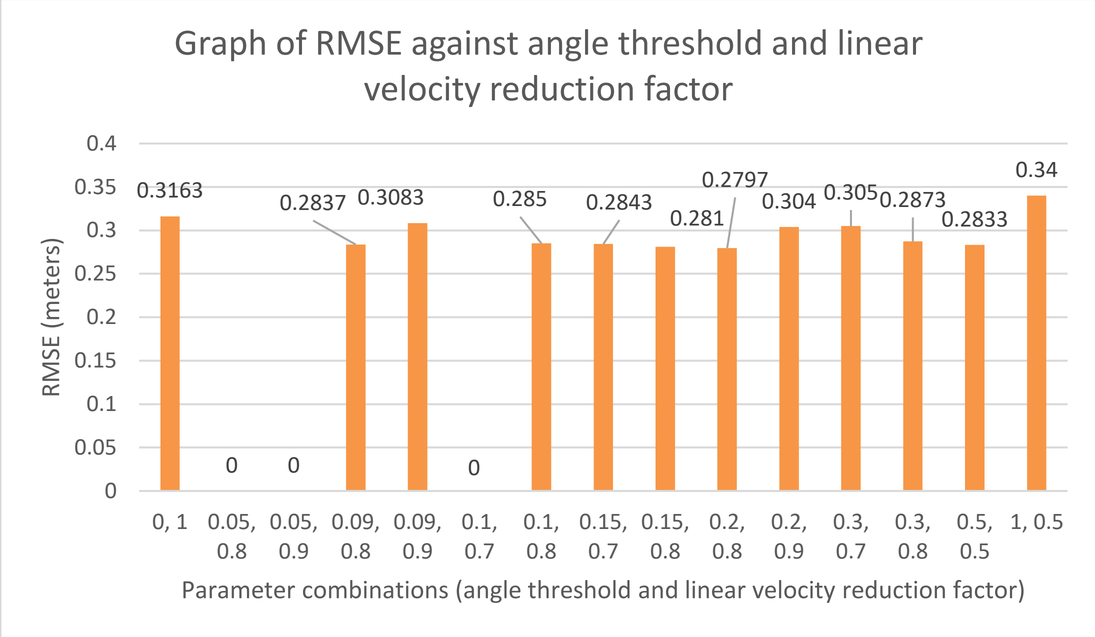


Figure 4.10: Graph of RMSE against angle threshold and linear velocity reduction factor

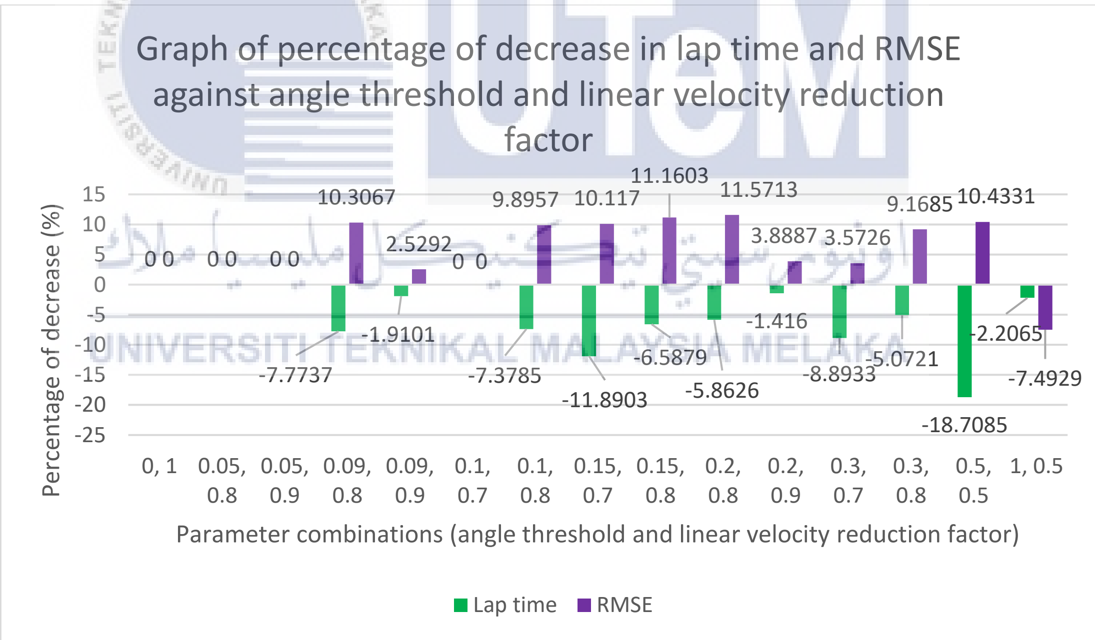


Figure 4.11: Graph of percentage of decrease in lap time and RMSE against angle threshold and linear velocity reduction factor

Based on Table 4.11, it can be seen that there are deviations between the expected path and the actual path of the TurtleBot's motion on the racetrack. Just like in previous experiments, it may be caused by wheel slippage, the TurtleBot's inertia and momentum as well as the chosen lookahead distance in the pure pursuit algorithm.

The angle threshold is a predefined limit that determines the maximum allowable deviation in the TurtleBot's orientation before corrective actions are taken, measured in radians. The linear velocity reduction factor is a scaling factor applied to the TurtleBot's speed to ensure safe and controlled movement when navigating sharp turns or avoiding obstacles. The angle threshold and linear velocity reduction factor are interdependent and can vary depending on various factors such as the size of racetrack. This means that the parameters must be fine-tuned among each other to maintain stable and accurate path tracking, just like in Experiment 4.

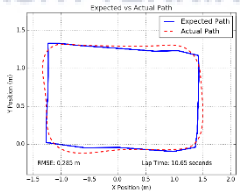
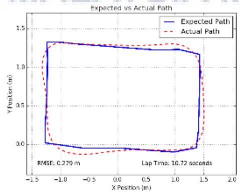
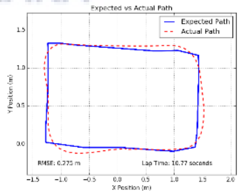
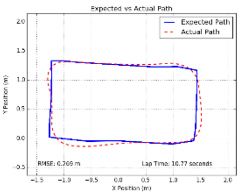
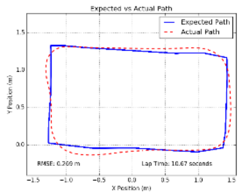
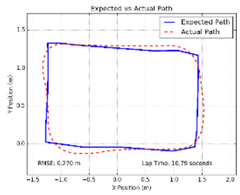
For this experiment, the angle threshold and linear velocity reduction factor of the TurtleBot are adjusted in a trial-and-error way to analyze and optimize the TurtleBot's performance. Table 4.12 shows that for some combinations of angle threshold and linear velocity reduction factor, the TurtleBot touched or crashed against the border because they were not appropriate to be matched with the pure pursuit parameters (linear velocity, lookahead distance and angular velocity proportional gain). At higher speeds, precise tuning of angle threshold and linear velocity reduction factors is critical to prevent path deviations and collisions with borders. The linear velocity reduction factor slows the TurtleBot during turns to ensure that it stays on track with the expected path.

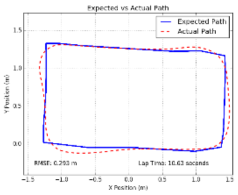
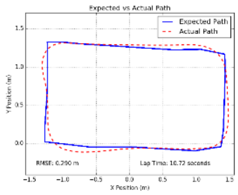
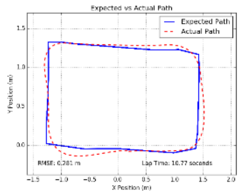
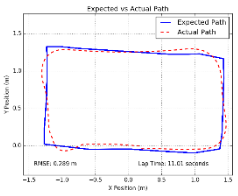
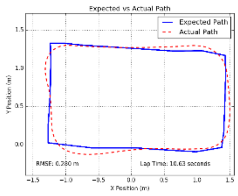
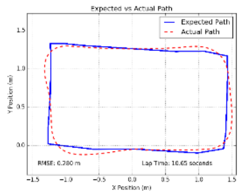
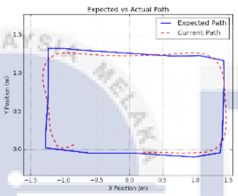
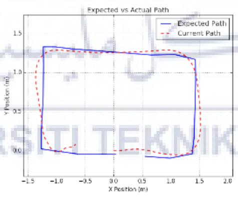
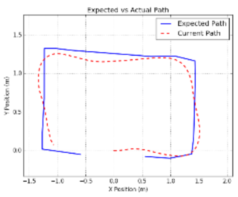
Figure 4.8 and 4.9 show that the average lap time and average RMSE between the expected and actual path of the TurtleBot are approximately the same. From Table 4.13 and Figure 4.10, it is evident that when there is an improvement (+%) in the lap time, there will be an increase (-%) in the RMSE value and vice versa for every combination of the angle threshold and linear velocity reduction factor. However, when the angle threshold = **0.2 rad** and linear velocity reduction factor = **0.8**, it yields the best result with an average lap time of **10.7133 seconds** and average RMSE of **0.2797 m**, where the percentage of decrease in RMSE (11.5713%) is higher than the percentage of increase in the lap time (-5.8626%). This experiment proves that a proper tuning of these two parameters can improve the RMSE, while suffering a slight trade-off from the increase in lap time. Hence, these parameter values are chosen to be conducted in the upcoming experiments for further improvements. As a result of this experiment, objective 2 and 3 have been fulfilled.

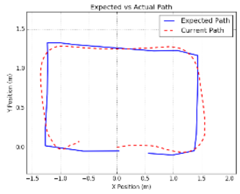
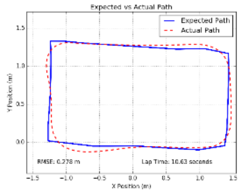
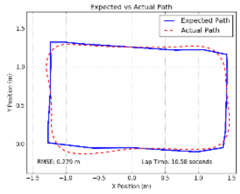
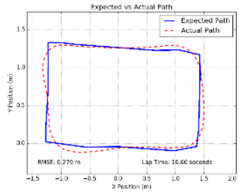
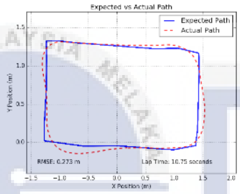
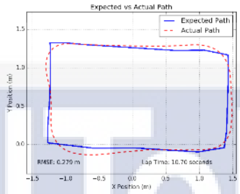
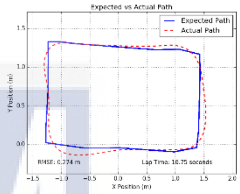
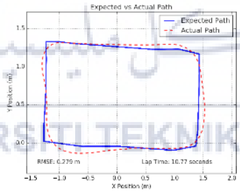
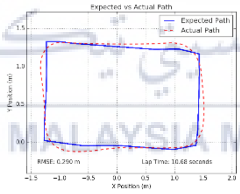
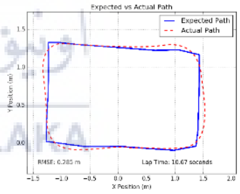
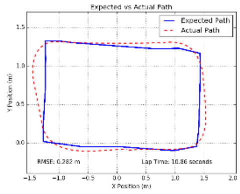
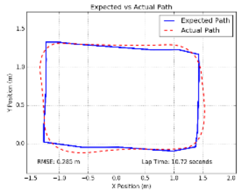
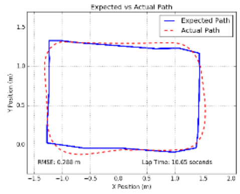
4.2.6 Experiment 6: Analysis of the performance of TurtleBot 3 Burger in real world racetrack with varying angular velocity integral and derivative gains

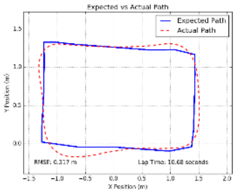
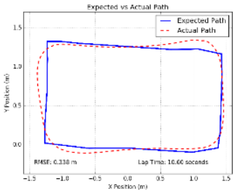
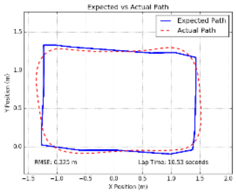
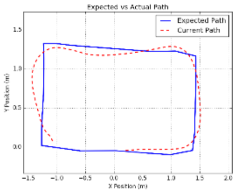
This experiment is conducted on the virtual racetrack similar to the one in Experiment 2. The objective of this experiment is to analyze the effect of varying angular velocity integral and derivative gains on the TurtleBot 3 Burger's behavior in terms of its ability to complete the lap, the time taken for it to finish a lap as well as the RMSE between the expected path and the actual path of its motion in the virtual racetrack. This is done by recording the time taken for the TurtleBot to finish a lap on the racetrack and calculating the RMSE between the expected and actual path for each lap after the autonomous racing script is run with multiple combinations of angular velocity integral and derivative gains. 3 repeated laps are completed by the TurtleBot for each combination of the respective parameters to obtain the results. The results of this experiment are shown in Table 4.14, 4.15 and 4.16 below. Figure 4.11, 4.12 and 4.13 visualizes the results in graphs.

Table 4.14: Graph of expected path against actual path for each parameter combination

Parameters	Lap 1	Lap 2	Lap 3
Ki = 0 Kd = 0			
Ki = 0.0001 Kd = 0			

Ki = 0.001 Kd = 0			
Ki = 0.01 Kd = 0			
Ki = 0.05 Kd = 0		-	-
Ki = 0.1 Kd = 0		-	-
Ki = 0.5 Kd = 0		-	-
Ki = 0.1 Kd = 0.0001		-	-

			
Ki = 0.01 Kd = 0.0001			
Ki = 0.001 Kd = 0.0001			
Ki = 0.0001 Kd = 0.0001			
Ki = 0.001 Kd = 0.001			
Ki = 0.01 Kd = 0.01			

			
Ki = 0.1 Kd = 0.1		-	-

* Ki = Angular velocity integral gain

Kd = Angular velocity derivative gain

Table 4.15: Lap time, RMSE and observation for respective laps of each parameter combination

Ki	Kd	Lap	Lap time (s)	RMSE (m)	Observation
0	0	1	10.65	0.285	Completed all laps
		2	10.72	0.279	
		3	10.77	0.275	
		Average	(10.65 + 10.72 + 10.77) / 3 = **10.7133**	(0.285 + 0.279 + 0.275) / 3 = **0.2797**	
0.0001	0	1	10.77	0.269	Completed all laps (best)
		2	10.67	0.269	
		3	10.79	0.270	
		Average	(10.77 + 10.67 + 10.79) / 3 = **10.7433**	(0.269 + 0.269 + 0.270) / 3 = **0.2693**	
0.001	0	1	10.63	0.293	
		2	10.72	0.290	
		3	10.77	0.281	

		Average	(10.63 + 10.72 + 10.77) / 3 = **10.7067**	(0.293 + 0.290 + 0.281) / 3 = **0.2880**	Completed all laps
0.01	0	1	11.01	0.289	Completed all laps
		2	10.63	0.280	
		3	10.65	0.280	
		Average	(11.01 + 10.63 + 10.65) / 3 = **10.7633**	(0.289 + 0.280 + 0.280) / 3 = **0.2830**	
0.05	0	1	-	-	Crashed on the first lap
		2			
		3			
		Average			
0.1	0	1	-	-	Crashed on the first lap
		2			
		3			
		Average			
0.5	0	1	-	-	Crashed on the first lap
		2			
		3			
		Average			
0.1	0.0001	1	-	-	Crashed on the first lap
		2			
		3			
		Average			
0.01	0.0001	1	10.63	0.278	Completed all laps
		2	10.58	0.279	
		3	10.66	0.279	
		Average	(10.63 + 10.58 + 10.66) / 3 = **10.6233**	(0.278 + 0.279 + 0.279) / 3 = **0.2787**	
		1	10.75	0.273	

0.001	0.0001	2	10.70	0.279	Completed all laps
		3	10.75	0.274	
		Average	(10.75 + 10.70 + 10.75) / 3 = **10.7333**	(0.273 + 0.279 + 0.274) / 3 = **0.2753**	
0.0001	0.0001	1	10.77	0.279	Completed all laps
		2	10.68	0.290	
		3	10.67	0.285	
		Average	(10.77 + 10.68 + 10.67) / 3 = **10.7067**	(0.279 + 0.290 + 0.285) / 3 = **0.2847**	
0.001	0.001	1	10.86	0.282	Completed all laps
		2	10.72	0.285	
		3	10.65	0.288	
		Average	(10.86 + 10.72 + 10.65) / 3 = **10.7433**	(0.282 + 0.285 + 0.288) / 3 = **0.2850**	
0.01	0.01	1	10.68	0.317	Completed all laps
		2	10.60	0.338	
		3	10.53	0.325	
		Average	(10.68 + 10.60 + 10.53) / 3 = **10.6033**	(0.317 + 0.338 + 0.325) / 3 = **0.3267**	
0.1	0.1	1	-	-	Crashed on the first lap
		2			
		3			
		Average			

* Ki = Angular velocity integral gain

Kd = Angular velocity derivative gain

Table 4.16: Percentage of decrease in lap time and RMSE compared to last experiment

Ki	Kd	Percentage of decrease in lap time (%)	Percentage of decrease in RMSE (%)
0	0	[(10.7133 – 10.7133) / 10.7133] x 100 = **0**	[(0.2797 – 0.2797) / 0.2797] x 100 = **0**
0.0001	0	[(10.7133 – 10.7433) / 10.7133] x 100 = **-0.2800**	[(0.2797 – 0.2693) / 0.2797] x 100 = **3.7183**
0.001	0	[(10.7133 – 10.7067) / 10.7133] x 100 = **0.0616**	[(0.2797 – 0.2880) / 0.2797] x 100 = **-2.9675**
0.01	0	[(10.7133 – 10.7633) / 10.7133] x 100 = **-0.4667**	[(0.2797 – 0.2830) / 0.2797] x 100 = **-1.1798**
0.05	0	-	-
0.1	0	-	-
0.5	0	-	-
0.1	0.0001	-	-
0.01	0.0001	[(10.7133 – 10.6233) / 10.7133] x 100 = **0.8401**	[(0.2797 – 0.2787) / 0.2797] x 100 = **0.3575**
0.001	0.0001	[(10.7133 – 10.7333) / 10.7133] x 100 = **-0.1867**	[(0.2797 – 0.2753) / 0.2797] x 100 = **1.5731**
0.0001	0.0001	[(10.7133 – 10.7067) / 10.7133] x 100 = **0.0616**	[(0.2797 – 0.2847) / 0.2797] x 100 = **-1.7876**
0.001	0.001	[(10.7133 – 10.7433) / 10.7133] x 100 = **-0.2800**	[(0.2797 – 0.2850) / 0.2797] x 100 = **-1.8949**
0.01	0.01	[(10.7133 – 10.6033) / 10.7133] x 100 = **1.0268**	[(0.2797 – 0.3267) / 0.2797] x 100 = **-16.8037**
0.1	0.1	-	-

* Ki = Angular velocity integral gain
  Kd = Angular velocity derivative gain

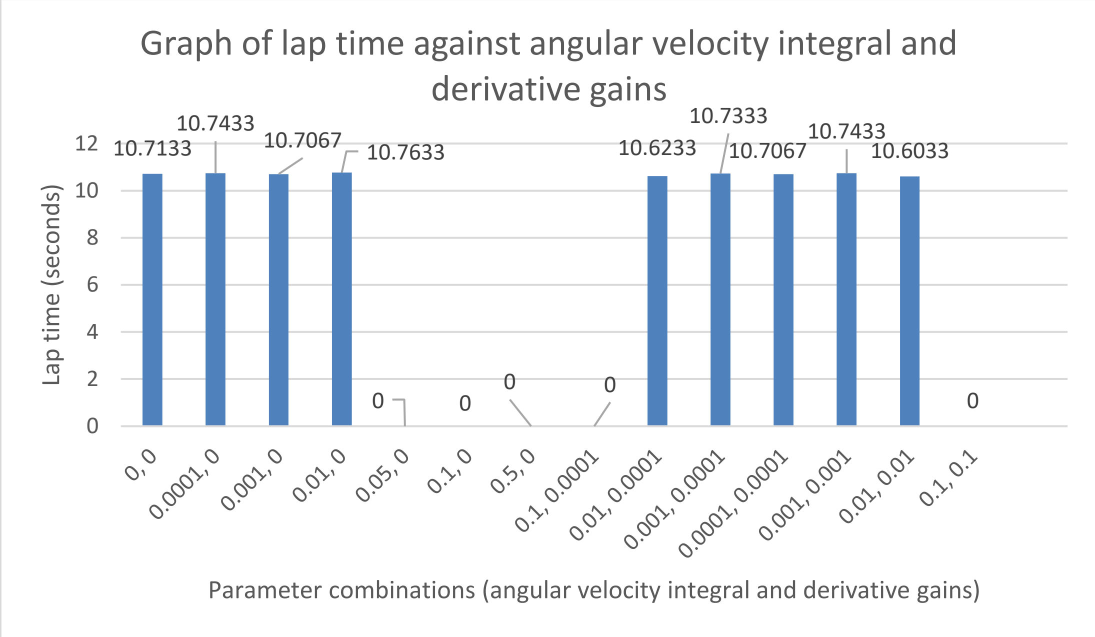


Figure 4.12: Graph of lap time against angular velocity integral and derivative gains

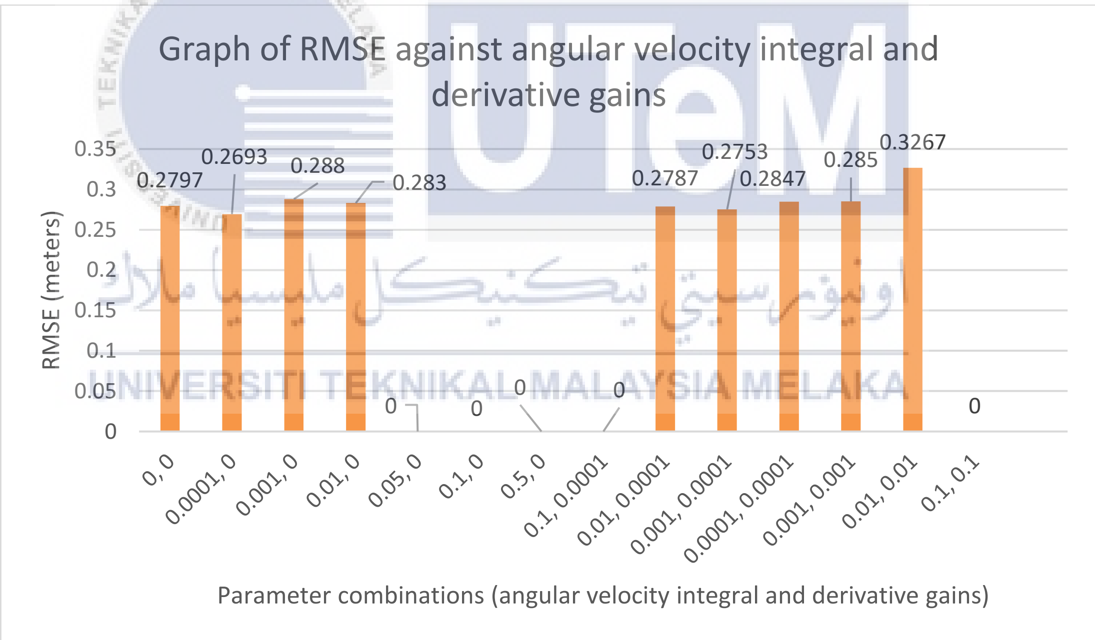


Figure 4.13: Graph of RMSE against angular velocity integral and derivative gains

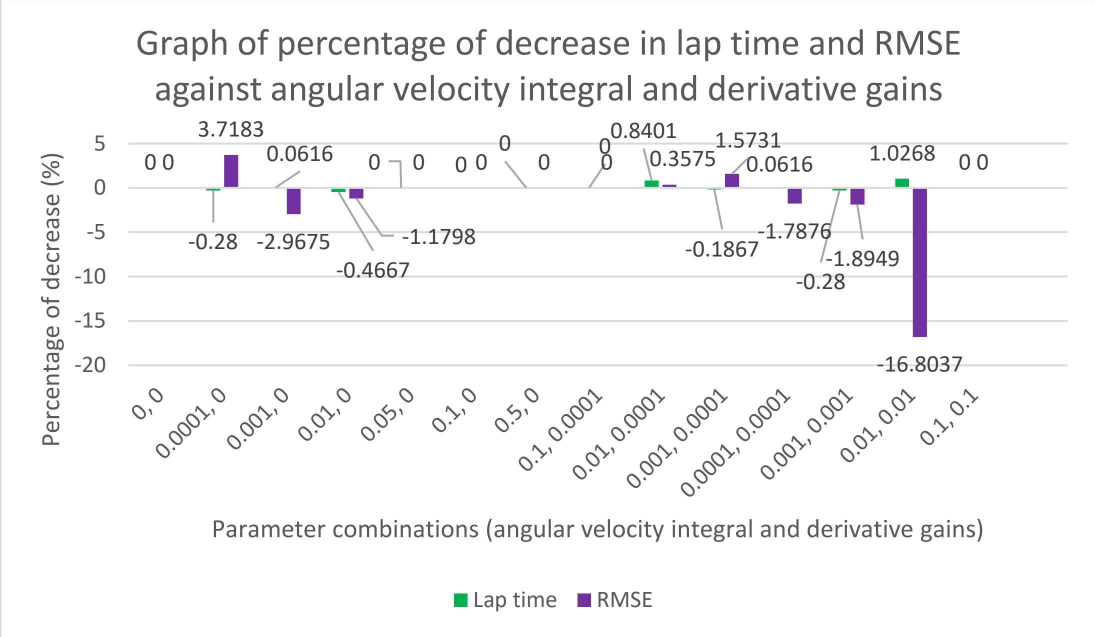


Figure 4.14: Graph of percentage of decrease in lap time and RMSE against angular velocity integral and derivative gains

Based on Table 4.14, it can be seen that there are deviations between the expected path and the actual path of the TurtleBot's motion on the racetrack. Just like in previous experiments, it may be caused by wheel slippage, the TurtleBot's inertia and momentum as well as the chosen lookahead distance in the pure pursuit algorithm. In PID control, the integral gain (Ki) for angular velocity integrates the error over time to eliminate steady-state error, while the derivative gain (Kd) predicts and responds to the rate of change of the error, enhancing stability and reducing oscillations in the system. These parameters are crucial for tuning the PID controller to achieve accurate and stable control of the TurtleBot's angular velocity during navigation tasks. In this experiment, the integral and derivative gains are combined with the proportional gain (Kp) used in the previous experiments to form a complete PID. These parameters are interdependent and can vary depending on various factors such as the size of racetrack. This means that the parameters must be fine-tuned among each other to maintain stable and accurate path tracking, just like in Experiment 4 and 5.

For this experiment, the angular velocity integral (Ki) and derivative (Kd) gains of the TurtleBot are adjusted in a trial-and-error way to analyze and optimize the TurtleBot's performance. Table 4.15 shows that for some combinations of Ki and Kd, the TurtleBot crashed against the border because they were not appropriate to be matched with the pure pursuit parameters (linear velocity, lookahead distance and angular velocity proportional gain). An inappropriate Ki and Kd combination can cause instability, while a well-tuned combination ensures a smoother and more accurate trajectory at higher speeds. Therefore, precise tuning of these gains is crucial for optimal high-speed navigation performance.

Figure 4.11 and 4.12 show that the average lap time and average RMSE between the expected and actual path of the TurtleBot are approximately the same. From Table 4.16 and Figure 4.13, it is evident that when there is an improvement (+%) in the lap time, there will be an increase (-%) in the RMSE value and vice versa for every combination of Ki and Kd. However, when Ki = **0.0001** and Kd = **0**, it yields the best result with an average lap time of **10.7433 seconds** and average RMSE of **0.2693 m**, where the percentage of decrease in RMSE (3.7183%) is higher than the percentage of increase in the lap time (-0.2800%). This experiment proves that a proper tuning of Ki and Kd with the existing Kp can further improve the RMSE, even by a small bit. Besides, the minor trade-off from the small increase in lap time is also inevitable, but the increase can be minimized through appropriate tuning. Since the improvement in RMSE is minimal, it shows that the system can already perform well by tuning just the Kp value. Hence, extensive adjustments to Ki and Kd may not be necessary for achieving significant performance gains and it might even negatively affect the TurtleBot's performance if not carefully balanced with the existing Kp value. As a result of this experiment, objective 2 and 3 have been fulfilled.

CHAPTER 5

CONCLUSION AND FUTURE WORKS

5.1 Conclusion

As a conclusion, the navigation system of TurtleBot 3 has been successfully developed using SLAM method. A total of six experiments have been conducted and the results are obtained successfully to meet all the three objectives of this project. Objective 1 which is to create a map of the surrounding environment for TurtleBot 3 using SLAM method is partially fulfilled through Experiment 1 and fulfilled through Experiment 2. Objective 2 which is to develop an autonomous racing navigation system for TurtleBot 3 with the map created from SLAM method is fulfilled through Experiment 2, 3, 4 ,5 and 6. Objective 3 which is to analyze the performance of the autonomous racing navigation system of TurtleBot 3 in terms of lap time and trajectory accuracy is fulfilled through Experiment 2, 3, 4, 5 and 6.

Based on the results, it can be seen that the overall layout of the virtual racetrack mapped using SLAM Gmapping is well defined with all the borders clearly recognized by the TurtleBot 3 Burger. The autonomous racing navigation system demonstrated effective path planning, ensuring that the TurtleBot is able to navigate on the racetrack accurately and efficiently. The experimental results show that the time taken for the TurtleBot to finish a lap and the RMSE between the expected and actual path of the TurtleBot are affected by its linear velocity, lookahead distance, PID gain of angular velocity as well as the angle threshold and linear velocity reduction factor while turning against sharp corners. The final optimized lap time and RMSE are 10.7433 seconds and 0.2693 m respectively. Further optimization may be achieved through detailed fine-tuning of the respective parameters. However, there might be a trade-off between lap time and RMSE, where reducing the lap time may lead to an increase in RMSE and vice versa. Hence, careful tuning is essential to balance both objectives and achieve optimal performance in the TurtleBot's autonomous navigation system.

5.2 Future Works

For the current system, the TurtleBot 3 is able to navigate around a known environment with static obstacles that it maps prior to autonomous navigation. In order to further improve the system, integrating dynamic obstacle avoidance capabilities is crucial. This enhancement would enable the TurtleBot 3 to detect and respond to moving obstacles in real-time, ensuring safe navigation in dynamic environments. Implementing this feature involves enhancing the perception system with sensors capable of detecting changes in the environment, such as cameras for visual recognition or LIDAR for precise distance measurements. Additionally, transitioning from simulation to real-world hardware deployment requires optimization on the system's algorithms and parameters for robustness and efficiency. This includes fine-tuning motion planning algorithms to account for real-time data from sensors and ensuring hardware reliability. Lastly, expanding the system with more advanced algorithms such as Model Predictive Control (MPC) would further elevate the TurtleBot 3's autonomy, enabling it to handle complex tasks and diverse real-world applications effectively.

REFERENCES

[1] Shah, P., Maheshwari, M., Ramane, S., Chandra, P., Valvi, M. S., & Mehendale, N. (2023). Autonomous Racing Vehicle. SSRN Electronic Journal. https://doi.org/10.2139/ssrn.4464632

[2] Project, C. a. V. (2023, October 29). landing_car-HEIC.jpg. Roboflow. https://universe.roboflow.com/curc-autonomous-vehicle-project/f1tenth-car-detection

[3] Chghaf, M., Rodriguez, S., & Ouardi, A. el. (2022). Camera, LiDAR and Multi-modal SLAM Systems for Autonomous Ground Vehicles: a Survey. Journal of Intelligent and Robotic Systems: Theory and Applications, 105(1). https://doi.org/10.1007/s10846-022-01582-8

[4] Mu, L., Yao, P., Zheng, Y., Chen, K., Wang, F., & Qi, N. (2020). Research on SLAM Algorithm of Mobile Robot Based on the Fusion of 2D LiDAR and Depth Camera. IEEE Access, 8, 157628–157642. https://doi.org/10.1109/ACCESS.2020.3019659

[5] Xuexi, Z., Guokun, L., Genping, F., Dongliang, X., & Shiliu, L. (2019). SLAM algorithm analysis of mobile robot based on lidar. Chinese Control Conference, CCC, 2019-July. https://doi.org/10.23919/ChiCC.2019.8866200

[6] Cheng, J., Zhang, L., Chen, Q., Hu, X., & Cai, J. (2022). A review of visual SLAM methods for autonomous driving vehicles. In Engineering Applications of Artificial Intelligence (Vol. 114). https://doi.org/10.1016/j.engappai.2022.104992

[7] Krul, S., Pantos, C., Frangulea, M., & Valente, J. (2021). Visual slam for indoor livestock and farming using a small drone with a monocular camera: A feasibility study. Drones, 5(2). https://doi.org/10.3390/drones5020041

[8] Amsters, R., & Slaets, P. (2020). Turtlebot 3 as a robotics education platform. Advances in Intelligent Systems and Computing, 1023. https://doi.org/10.1007/978-3-030-26945-6_16

[9] Martínez, F. H. (2021). TurtleBot3 robot operation for navigation applications using ROS. Tekhnê, 18(2).

[10] Brown, M. (2024, February 14). IEEE IV 2024 Call for Participation: F1tenth Autonomous Racing Competition. IEEE ITSS. https://ieee-itss.org/ieee-iv-2024-call-for-participation-f1tenth-autonomous-racing-competition/

[11] Wojcik, H. (2023, June 16). Penn Engineering Students Win the 12th Annual F1Tenth Autonomous Grand Prix. Penn Engineering Blog. https://blog.seas.upenn.edu/penn-engineering-students-win-the-12th-annual-f1tenth-autonomous-grand-prix/

[12] Robotis. (2018). ROBOTIS e-Manual. ROBOTIS e-Manual. https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/

[13] Sun, J., Zhao, J., Hu, X., Gao, H., & Yu, J. (2023). Autonomous Navigation System of Indoor Mobile Robots Using 2D Lidar. Mathematics, 11(6). https://doi.org/10.3390/math11061455

[14] View of Development of F1 Tenth-specific Autonomous Navigation Algorithm. (2024). https://www.propulsiontechjournal.com/index.php/journal/article/view/4759/3258

[15] O'Kelly, M., Zheng, H., Karthik, D., & Mangharam, R. (2019). F1TENTH: An Open-source Evaluation Environment for Continuous Control and Reinforcement Learning. Proceedings of Machine Learning Research, 123

[16] Tatulea-Codrean, A., Mariani, T., & Engell, S. (2020). Design and simulation of a machine-learning and model predictive control approach to autonomous race

driving for the F1/10 platform. IFAC-PapersOnLine, 53(2). https://doi.org/10.1016/j.ifacol.2020.12.166

[17] Patton, R., Schuman, C., Kulkarni, S., Parsa, M., Mitchell, J. P., Haas, N. Q., Stahl, C., Paulissen, S., Date, P., Potok, T., & Sneider, S. (2021). Neuromorphic Computing for Autonomous Racing. ACM International Conference Proceeding Series. https://doi.org/10.1145/3477145.3477170

[18] Verardi, M. (2024, May 16). Martina Verardi on LinkedIn: #icra2024 #f1tenth #autonomousracing #innovation #cultureexchange. https://www.linkedin.com/posts/martinaverardi_icra2024-f1tenth-autonomousracing-activity-7196842834908344321-1u95/

[19] Babu, V. S., & Behl, M. (2020). F1tenth.dev-An Open-source ROS based F1/10 Autonomous Racing Simulator. IEEE International Conference on Automation Science and Engineering, 2020-August. https://doi.org/10.1109/CASE48305.2020.9216949

[20] Learn. (2024, March 25). https://f1tenth.org/learn.html

[21] Soebhakti, H., Yulianti, R., Risi, F., & Pratiwi, Y. (2023). Obstacle Avoidance System Using LiDAR on Robot Turtlebot3 Burger. https://doi.org/10.4108/eai.5-10-2022.2327479

[22] Team, R. (2023, April 22). TurtleBot. ROBOTS: Your Guide to the World of Robotics. https://robotsguide.com/robots/turtlebot

[23] TurtleBot 2 - Open source personal research robot. (2021, November 16). Clearpath Robotics. https://clearpathrobotics.com/turtlebot-2-open-source-robot/#:~:text=MOBILE%20ROBOT%20PLATFORM&text=This%20second%20generation%20personal%20robot,Pro%20Sensor%20and%20a%20gyroscope.

[24] TurtleBot 4 - Clearpath Robotics. (2023, April 26). Clearpath Robotics. https://clearpathrobotics.com/turtlebot-4/

[25] Senecat, M. (2023, December 19). TurtleBot4 vs. TurtleBot3 : What are the differences and improvements? - Génération Robots - Blog. Génération Robots - Blog. https://www.generationrobots.com/blog/en/turtlebot4-vs-turtlebot3-what-are-the-differences-and-improvements/#:~:text=TurtleBot%204%20is%20a%20significant,new%20challenges%20in%20mobile%20robotics.

[26] Khnissi, K., Seddik, C., & Seddik, H. (2018). Smart Navigation of Mobile Robot Using Neural Network Controller. 2018 International Conference on Smart Communications in Network Technologies, SaCoNeT 2018. https://doi.org/10.1109/SaCoNeT.2018.8585616

[27] de Assis Brasil, P. M., Pereira, F. U., de Souza Leite Cuadros, M. A., Cukla, A. R., & Tello Gamarra, D. F. (2020). A Study on Global Path Planners Algorithms for the Simulated TurtleBot 3 Robot in ROS. 2020 Latin American Robotics Symposium, 2020 Brazilian Symposium on Robotics and 2020 Workshop on Robotics in Education, LARS-SBR-WRE 2020. https://doi.org/10.1109/LARS/SBR/WRE51543.2020.9307003

[28] Pietrzik, S., & Chandrasekaran, B. (2019). Setting up and Using ROS-Kinetic and Gazebo for Educational Robotic Projects and Learning. Journal of Physics: Conference Series, 1207(1). https://doi.org/10.1088/1742-6596/1207/1/012019

[29] Thakur, R. (2023, February 18). Choosing the Best Linux Distribution for ROS - RAVI THAKUR - Medium. Medium. https://medium.com/@ravi.deepak.thakur/choosing-the-best-linux-distribution-for-ros-278e82d52eb5

[30] Robot Operating System Cookbook. (2023). https://subscription.packtpub.com/book/iot-and-hardware/9781783987443/1/ch01lvl1sec03/installing-ros-on-desktop-systems

[31] Alatise, M. B., & Hancke, G. P. (2020). A Review on Challenges of Autonomous Mobile Robot and Sensor Fusion Methods. In IEEE Access (Vol. 8). https://doi.org/10.1109/ACCESS.2020.2975643

[32] Teleweck, P. E., & Chandrasekaran, B. (2019). Path Planning Algorithms and Their Use in Robotic Navigation Systems. Journal of Physics: Conference Series, 1207(1). https://doi.org/10.1088/1742-6596/1207/1/012018

[33] Dai, Y. (2022). Research on Robot Positioning and Navigation Algorithm Based on SLAM. Wireless Communications and Mobile Computing, 2022. https://doi.org/10.1155/2022/3340529

[34] Zhou, B., Du, M., Chen, Z., Liu, Y., Zhang, Y., & Wang, Y. (2022). Design and Implementation of Intelligent Security Robot Based on Lidar and Vision Fusion. Journal of Physics: Conference Series, 2216(1). https://doi.org/10.1088/1742-6596/2216/1/012013

[35] Escobar-Naranjo, J., Caiza, G., Ayala, P., Jordan, E., Garcia, C. A., & Garcia, M. v. (2023). Autonomous Navigation of Robots: Optimization with DQN. Applied Sciences (Switzerland), 13(12). https://doi.org/10.3390/app13127202

[36] Nandkumar, C., Shukla, P., & Varma, V. (2021). Simulation of Indoor Localization and Navigation of Turtlebot 3 using Real Time Object Detection. Proceedings of IEEE International Conference on Disruptive Technologies for Multi-Disciplinary Research and Applications, CENTCON 2021. https://doi.org/10.1109/CENTCON52345.2021.9687937

[37] Gurel, C., Sathyam, R., & Guha, A. (2018). ROS-based Path Planning for Turtlebot Robot using Rapidly Exploring Random Trees (RRT*). Researchgate.Net, May.

[38] Abdelwahab, M., Parque, V., Fath Elbab, A. M. R., Abouelsoud, A. A., & Sugano, S. (2020). Trajectory tracking of wheeled mobile robots using Z-Number based fuzzy logic. IEEE Access, 8. https://doi.org/10.1109/ACCESS.2020.2968421

[39] Boztaş, G., & Aydoğmuş, Ö. (2021). Implementation of Pure Pursuit Algorithm for Nonholonomic Mobile Robot using Robot Operating System. Balkan Journal of Electrical and Computer Engineering, 9(4), 337-341. https://doi.org/10.17694/bajece.983350

[40] A. Ali, "10+ Advantages and Disadvantages of Night Vision Technology » Hubvela," Hubvela, Jun. 20, 2023. https://hubvela.com/hub/technology/advantages-disadvantages/night-vision-technology/

[41] Daneshyar, S. A., & Nahvi, M. (2017). Moving objects tracking based on improved particle filter algorithm by elimination of unimportant particles. Optik, 138. https://doi.org/10.1016/j.ijleo.2017.03.100

[42] FlyGuys, "LiDAR vs. Sonar: What's the Difference? - FlyGuys," FlyGuys, Jul. 07, 2023. https://flyguys.com/lidar-vs-sonar-whats-the-difference/

[43] T. Test, "Ultrasonic Sensors vs. LiDAR: Which One Should You Use?," MaxBotix, Apr. 20, 2021. https://maxbotix.com/blogs/blog/ultrasonic-sensors-vs-lidar-which-one-should-you-use

[44] Yang, M., Sun, X., Jia, F., Rushworth, A., Dong, X., Zhang, S., Fang, Z., Yang, G., & Liu, B. (2022). Sensors and Sensor Fusion Methodologies for Indoor Odometry: A Review. Polymers, 14(10). https://doi.org/10.3390/polym14102019

[45] Zhang, J., & Singh, S. (2015). Visual-lidar odometry and mapping: Low-drift, robust, and fast. Proceedings - IEEE International Conference on Robotics and Automation, 2015-June(June). https://doi.org/10.1109/ICRA.2015.7139486

[46] Jiang, F., Chen, J., & Ji, S. (2021). Panoramic Visual-Inertial SLAM Tightly Coupled with a Wheel Encoder. ISPRS Journal of Photogrammetry and Remote Sensing, 182. https://doi.org/10.1016/j.isprsjprs.2021.10.006

[47] Olalekan, A. F., Sagor, J. A., Hasan, M. H., & Oluwatobi, A. S. (2021). Comparison of two SLAM algorithms provided by ROS (Robot Operating System). 2021 2nd International Conference for Emerging Technology, INCET 2021. https://doi.org/10.1109/INCET51464.2021.9456164

[48] Yagfarov, R., Ivanou, M., & Afanasyev, I. (2018). Map Comparison of Lidar-based 2D SLAM Algorithms Using Precise Ground Truth. 2018 15th International Conference on Control, Automation, Robotics and Vision, ICARCV 2018. https://doi.org/10.1109/ICARCV.2018.8581131

[49] Tian, C., Liu, H., Liu, Z., Li, H., & Wang, Y. (2023). Research on Multi-Sensor Fusion SLAM Algorithm Based on Improved Gmapping. IEEE Access, 11. https://doi.org/10.1109/ACCESS.2023.3243633

APPENDICES

APPENDIX A: GANTT CHART FOR FINAL YEAR PROJECT

Final Year Project 1

FYP 1 Gantt Chart

Duration	OCT'23				NOV'23				DEC'23				JAN'24	
Activities	**W1**	**W2**	**W3**	**W4**	**W5**	**W6**	**W7**	**W8**	**W9**	**W10**	**W11**	**W12**	**W13**	**W14**
Selection of project title														
Project title registration and submission														
Literature review														
i. Search for relevant references														
ii. Summarize references														
Introduction														
i. Determine problem statements, objectives and scopes								**Mid Semester Break**						
Methodology														
i. Install Ubuntu OS and ROS														
ii. Conduct simulation on TurtleBot 3 mapping and navigation														
iii. Experiment implementation and analysis														
Results														
i. Obtain early results from experiment														
FYP 1 Seminar														
FYP 1 report writing														
FYP 1 report submission														

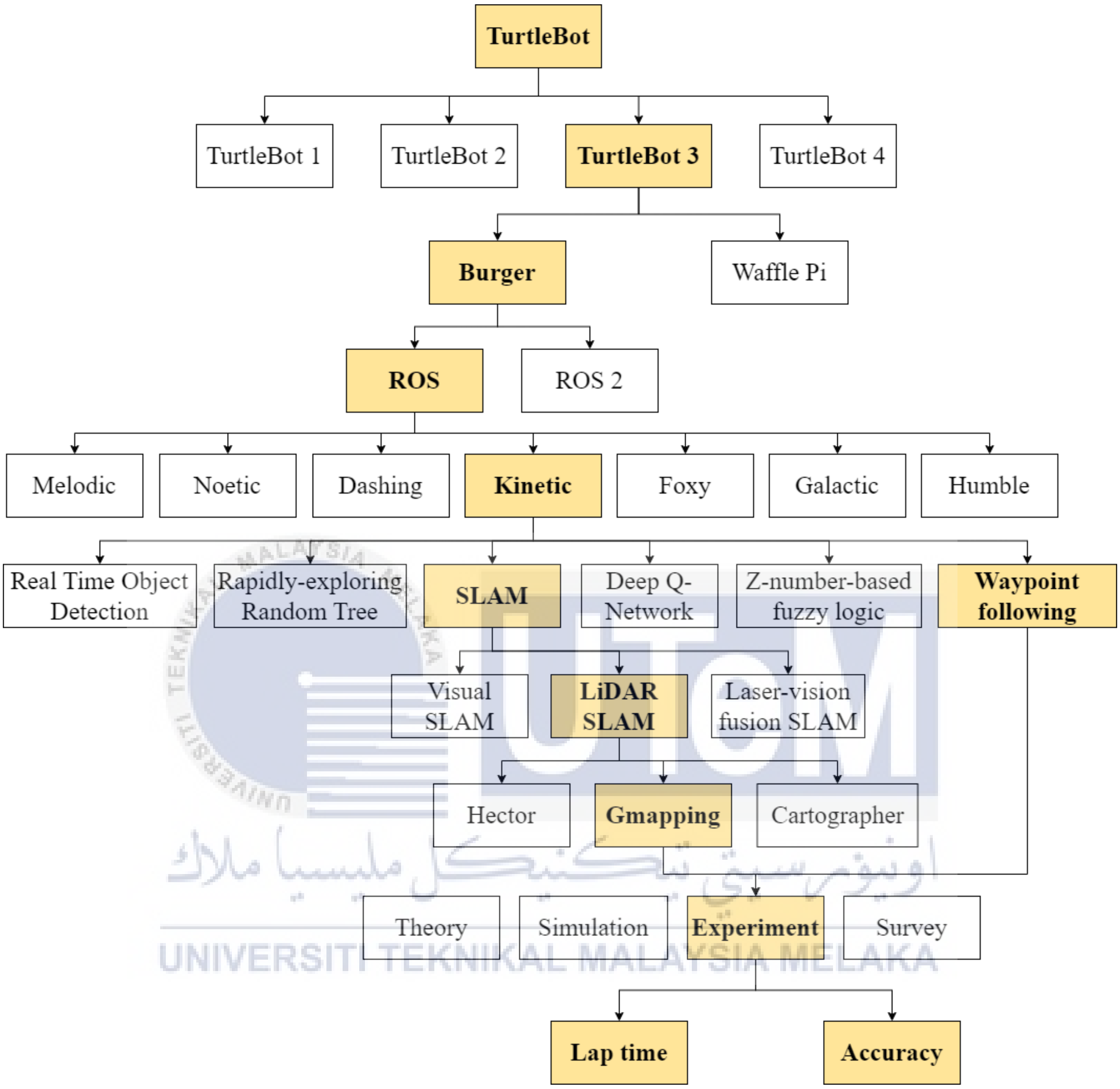
Final Year Project 2

FYP 2 Gantt Chart

Duration	MAC'24			APR'24				MAY'24				JUN'24		
Activities	**W1**	**W2**	**W3**	**W4**	**W5**	**W6**	**W7**	**W8**	**W9**	**W10**	**W11**	**W12**	**W13**	**W14**
Methodology														
i. Implement TurtleBot 3 mapping and navigation in real environment														
ii. Experiment implementation and analysis														
iii. Improve parameters of previous experiments														
Results								**Mid Semester Break**						
i. Collect results from experiment														
ii. Results analysis and discussion														
FYP 2 presentation														
FYP 2 report writing														
FYP 2 report submission														

	Completed
	Delayed
	In Progress

APPENDIX B: K-CHART



APPENDIX C: CODING OF TURTLEBOT 3

TurtleBot 3 SLAM launch file

```
<launch>
  <!-- Arguments -->
  <arg name="model" default="$(env TURTLEBOT3_MODEL)" doc="model type [burger, waffle, waffle_pi]"/>
  <arg name="slam_methods" default="gmapping" doc="slam type [gmapping, cartographer, hector, karto, frontier_exploration]"/>
  <arg name="configuration_basename" default="turtlebot3_lds_2d.lua"/>
  <arg name="open_rviz" default="true"/>

  <!-- TurtleBot3 -->
  <include file="$(find turtlebot3_bringup)/launch/turtlebot3_remote.launch">
    <arg name="model" value="$(arg model)" />
  </include>

  <!-- SLAM: Gmapping, Cartographer, Hector, Karto, Frontier_exploration, RTAB-Map -->
  <include file="$(find turtlebot3_slam)/launch/turtlebot3_$(arg slam_methods).launch">
    <arg name="model" value="$(arg model)"/>
    <arg name="configuration_basename" value="$(arg configuration_basename)"/>
  </include>

  <!-- rviz -->
  <group if="$(arg open_rviz)">
    <node pkg="rviz" type="rviz" name="rviz" required="true"
          args="-d $(find turtlebot3_slam)/rviz/turtlebot3_$(arg slam_methods).rviz"/>
  </group>
</launch>
```

TurtleBot 3 teleoperation launch file

```
<launch>
  <arg name="model" default="$(env TURTLEBOT3_MODEL)" doc="model type [burger, waffle, waffle_pi]"/>
  <param name="model" value="$(arg model)"/>

  <!-- turtlebot3_teleop_key already has its own built in velocity smoother -->
  <node pkg="turtlebot3_teleop" type="turtlebot3_teleop_key" name="turtlebot3_teleop_keyboard"  output="screen">
  </node>
</launch>
```

TurtleBot 3 teleoperation node

```python
import rospy
from geometry_msgs.msg import Twist
import sys, select, os
if os.name == 'nt':
  import msvcrt, time
else:
  import tty, termios

BURGER_MAX_LIN_VEL = 0.22
BURGER_MAX_ANG_VEL = 2.84

WAFFLE_MAX_LIN_VEL = 0.26
WAFFLE_MAX_ANG_VEL = 1.82

LIN_VEL_STEP_SIZE = 0.01
ANG_VEL_STEP_SIZE = 0.1

msg = """
Control Your TurtleBot3!
---------------------------
Moving around:
        w
   a    s    d
        x

w/x : increase/decrease linear velocity (Burger : ~ 0.22, Waffle and Waffle Pi : ~ 0.26)
a/d : increase/decrease angular velocity (Burger : ~ 2.84, Waffle and Waffle Pi : ~ 1.82)

space key, s : force stop

CTRL-C to quit
"""

e = """
Communications Failed
"""

def getKey():
    if os.name == 'nt':
        timeout = 0.1
        startTime = time.time()
        while(1):
            if msvcrt.kbhit():
                if sys.version_info[0] >= 3:
                    return msvcrt.getch().decode()
                else:
                    return msvcrt.getch()
            elif time.time() - startTime > timeout:
                return ''

    tty.setraw(sys.stdin.fileno())
    rlist, _, _ = select.select([sys.stdin], [], [], 0.1)
    if rlist:
        key = sys.stdin.read(1)
    else:
        key = ''

    termios.tcsetattr(sys.stdin, termios.TCSADRAIN, settings)
    return key

def vels(target_linear_vel, target_angular_vel):
    return "currently:\tlinear vel %s\t angular vel %s " % (target_linear_vel,target_angular_vel)

def makeSimpleProfile(output, input, slop):
    if input > output:
        output = min( input, output + slop )
    elif input < output:
        output = max( input, output - slop )
    else:
        output = input

    return output

def constrain(input, low, high):
    if input < low:
      input = low
    elif input > high:
      input = high
    else:
      input = input

    return input
```

```python
def checkLinearLimitVelocity(vel):
    if turtlebot3_model == "burger":
      vel = constrain(vel, -BURGER_MAX_LIN_VEL, BURGER_MAX_LIN_VEL)
    elif turtlebot3_model == "waffle" or turtlebot3_model == "waffle_pi":
      vel = constrain(vel, -WAFFLE_MAX_LIN_VEL, WAFFLE_MAX_LIN_VEL)
    else:
      vel = constrain(vel, -BURGER_MAX_LIN_VEL, BURGER_MAX_LIN_VEL)

    return vel

def checkAngularLimitVelocity(vel):
    if turtlebot3_model == "burger":
      vel = constrain(vel, -BURGER_MAX_ANG_VEL, BURGER_MAX_ANG_VEL)
    elif turtlebot3_model == "waffle" or turtlebot3_model == "waffle_pi":
      vel = constrain(vel, -WAFFLE_MAX_ANG_VEL, WAFFLE_MAX_ANG_VEL)
    else:
      vel = constrain(vel, -BURGER_MAX_ANG_VEL, BURGER_MAX_ANG_VEL)

    return vel

if __name__=="__main__":
    if os.name != 'nt':
        settings = termios.tcgetattr(sys.stdin)

    rospy.init_node('turtlebot3_teleop')
    pub = rospy.Publisher('cmd_vel', Twist, queue_size=10)

    turtlebot3_model = rospy.get_param("model", "burger")

    status = 0
    target_linear_vel   = 0.0
    target_angular_vel  = 0.0
    control_linear_vel  = 0.0
    control_angular_vel = 0.0

    try:
        print(msg)
        while not rospy.is_shutdown():
            key = getKey()
            if key == 'w' :
                target_linear_vel = checkLinearLimitVelocity(target_linear_vel + LIN_VEL_STEP_SIZE)
                status = status + 1
                print(vels(target_linear_vel,target_angular_vel))
            elif key == 'x' :
                target_linear_vel = checkLinearLimitVelocity(target_linear_vel - LIN_VEL_STEP_SIZE)
                status = status + 1
                print(vels(target_linear_vel,target_angular_vel))
            elif key == 'a' :
                target_angular_vel = checkAngularLimitVelocity(target_angular_vel + ANG_VEL_STEP_SIZE)
                status = status + 1
                print(vels(target_linear_vel,target_angular_vel))
            elif key == 'd' :
                target_angular_vel = checkAngularLimitVelocity(target_angular_vel - ANG_VEL_STEP_SIZE)
                status = status + 1
                print(vels(target_linear_vel,target_angular_vel))
            elif key == ' ' or key == 's' :
                target_linear_vel   = 0.0
                control_linear_vel  = 0.0
                target_angular_vel  = 0.0
                control_angular_vel = 0.0
                print(vels(target_linear_vel, target_angular_vel))
            else:
                if (key == '\x03'):
                    break

            if status == 20 :
                print(msg)
                status = 0

            twist = Twist()

            control_linear_vel = makeSimpleProfile(control_linear_vel, target_linear_vel, (LIN_VEL_STEP_SIZE/2.0))
            twist.linear.x = control_linear_vel; twist.linear.y = 0.0; twist.linear.z = 0.0

            control_angular_vel = makeSimpleProfile(control_angular_vel, target_angular_vel, (ANG_VEL_STEP_SIZE/2.0))
            twist.angular.x = 0.0; twist.angular.y = 0.0; twist.angular.z = control_angular_vel

            pub.publish(twist)

    except:
        print(e)

    finally:
        twist = Twist()
        twist.linear.x = 0.0; twist.linear.y = 0.0; twist.linear.z = 0.0
        twist.angular.x = 0.0; twist.angular.y = 0.0; twist.angular.z = 0.0
```

```
        pub.publish(twist)

    if os.name != 'nt':
        termios.tcsetattr(sys.stdin, termios.TCSADRAIN, settings)
```

TurtleBot 3 navigation launch file

```
<launch>
  <!-- Arguments -->
  <arg name="model" default="$(env TURTLEBOT3_MODEL)" doc="model type [burger, waffle, waffle_pi]"/>
  <arg name="map_file" default="$(find turtlebot3_navigation)/maps/map.yaml"/>
  <arg name="open_rviz" default="true"/>
  <arg name="move_forward_only" default="false"/>

  <!-- Turtlebot3 -->
  <include file="$(find turtlebot3_bringup)/launch/turtlebot3_remote.launch">
    <arg name="model" value="$(arg model)" />
  </include>

  <!-- Map server -->
  <node pkg="map_server" name="map_server" type="map_server" args="$(arg map_file)"/>

  <!-- AMCL -->
  <include file="$(find turtlebot3_navigation)/launch/amcl.launch"/>

  <!-- move_base -->
  <include file="$(find turtlebot3_navigation)/launch/move_base.launch">
    <arg name="model" value="$(arg model)" />
    <arg name="move_forward_only" value="$(arg move_forward_only)"/>
  </include>

  <!-- rviz -->
  <group if="$(arg open_rviz)">
    <node pkg="rviz" type="rviz" name="rviz" required="true"
          args="-d $(find turtlebot3_navigation)/rviz/turtlebot3_navigation.rviz"/>
  </group>
</launch>
```

APPENDIX D: AUTONOMOUS RACING NAVIGATION SCRIPT

```python
#!/usr/bin/env python

import rospy
import math
import time
import matplotlib.pyplot as plt
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
from tf.transformations import euler_from_quaternion
import numpy as np

# Define the waypoints
WAYPOINTS = [
    (0.55785882473, -0.0744303613901),
    (0.986305236816, -0.0964283570647),
    (1.37430500984, -0.041693713516),
    (1.4085495472, 0.167969807982),
    (1.42460131645, 0.574233055115),
    (1.43535208702, 0.907108724117),
    (1.43031644821, 1.16660523415),
    (1.07386648655, 1.22979664803),
    (0.652593672276, 1.22451078892),
    (-0.745338320732, 1.30347430706),
    (-1.01950228214, 1.32738614082),
    (-1.23254036903, 1.32750248909),
    (-1.22658610344, 1.10366272926),
    (-1.23305869102, 0.677461087704),
    (-1.26714420319, 0.232884287834),
    (-1.26731300354, 0.0213141478598),
    (-0.595813214779, -0.0466784350574),
    (0.0376093015075, -0.0449083335698)
]

# Pure Pursuit parameters
LOOKAHEAD_DISTANCE = 0.4  # meters
VELOCITY = 0.8  # m/s
ANGLE_THRESHOLD = 0.2 # radians
VELOCITY_REDUCTION = 0.8 # percentage

# PID parameters for angular velocity
ANGULAR_PID_PARAMS = {
    'Kp': 2.5,
    'Ki': 0.0001,
    'Kd': 0
}

class PIDController:
    def __init__(self, Kp, Ki, Kd):
        self.Kp = Kp
        self.Ki = Ki
        self.Kd = Kd
        self.prev_error = 0
        self.integral = 0

    def update_params(self, Kp, Ki, Kd):
        self.Kp = Kp
        self.Ki = Ki
        self.Kd = Kd

    def compute(self, error, dt):
        self.integral += error * dt
        derivative = (error - self.prev_error) / dt if dt > 0 else 0.0
        output = self.Kp * error + self.Ki * self.integral + self.Kd * derivative
        self.prev_error = error
        return output

class PurePursuit:
    def __init__(self):
        rospy.init_node('turtlebot3_pure_pursuit')
        self.velocity_publisher = rospy.Publisher('/cmd_vel', Twist, queue_size=10)
        self.odom_subscriber = rospy.Subscriber('/odom', Odometry, self.odom_callback)
        self.current_position = (0.0, 0.0)
        self.current_orientation = 0.0
        self.current_waypoint_index = 0
        self.lap_data = []
        self.intended_path_data = {'x': [], 'y': []}
        self.actual_path_data = {'x': [], 'y': []}
        self.timer_started = False
        self.start_time = None
        self.lap_time = None
        self.latest_rmse = None
        self.lap_count = 0
        self.first_lap = True

        # PID controller for angular velocity
        self.angular_pid = PIDController(ANGULAR_PID_PARAMS['Kp'], ANGULAR_PID_PARAMS['Ki'], ANGULAR_PID_PARAMS['Kd'])
        self.prev_time = time.time()

    def odom_callback(self, data):
        position = data.pose.pose.position
        orientation_q = data.pose.pose.orientation
        _, _, yaw = euler_from_quaternion([orientation_q.x, orientation_q.y, orientation_q.z, orientation_q.w])
        self.current_position = (position.x, position.y)
        self.current_orientation = yaw

        # Append actual position data
        self.actual_path_data['x'].append(position.x)
        self.actual_path_data['y'].append(position.y)

```

```python
        if self.first_lap:
            # Append intended position data only during the first lap
            self.intended_path_data['x'].append(WAYPOINTS[self.current_waypoint_index][0])
            self.intended_path_data['y'].append(WAYPOINTS[self.current_waypoint_index][1])

        self.pursue_waypoints()

    def pursue_waypoints(self):
        target_waypoint = WAYPOINTS[self.current_waypoint_index]
        while True:
            if self.reached_waypoint(target_waypoint):
                if self.current_waypoint_index == 0:
                    if not self.timer_started:
                        self.start_time = time.time()
                        self.timer_started = True
                        rospy.loginfo("Lap timer started.")
                    else:
                        self.lap_time = time.time() - self.start_time
                        self.latest_rmse = self.calculate_rmse()  # Calculate RMSE for the latest lap
                        self.lap_count += 1
                        rospy.loginfo("Lap {}: RMSE: {:.3f} m, Lap Time: {:.2f} seconds".format(self.lap_count, self.latest_rmse, self.lap_time))

                        # Store lap data
                        self.lap_data.append({
                            'intended_path': self.intended_path_data.copy(),
                            'actual_path': self.actual_path_data.copy(),
                            'lap_time': self.lap_time,
                            'rmse': self.latest_rmse
                        })

                        # Reset actual path data for the next lap
                        self.actual_path_data = {'x': [], 'y': []}
                        self.start_time = time.time()  # Restart the timer for the next lap

                self.current_waypoint_index = (self.current_waypoint_index + 1) % len(WAYPOINTS)
                target_waypoint = WAYPOINTS[self.current_waypoint_index]
            else:
                break
        self.drive_to_waypoint(target_waypoint)

    def reached_waypoint(self, waypoint):
        distance = math.sqrt((waypoint[0] - self.current_position[0]) ** 2 + (waypoint[1] - self.current_position[1]) ** 2)
        return distance < LOOKAHEAD_DISTANCE

    def drive_to_waypoint(self, waypoint):
        current_time = time.time()
        dt = current_time - self.prev_time
        self.prev_time = current_time

        # Angular PID control
        angle_to_waypoint = math.atan2(waypoint[1] - self.current_position[1], waypoint[0] - self.current_position[0])
        angle_diff = self.normalize_angle(angle_to_waypoint - self.current_orientation)
        twist = Twist()
        twist.angular.z = self.angular_pid.compute(angle_diff, dt)

        # Calculate distance to waypoint
        distance_to_waypoint = math.sqrt((waypoint[0] - self.current_position[0]) ** 2 + (waypoint[1] - self.current_position[1]) ** 2)

        # Adjust linear velocity based on angular error (or curvature)
        if abs(angle_diff) > ANGLE_THRESHOLD:  # Threshold for sharp turns
            twist.linear.x = VELOCITY_REDUCTION * VELOCITY  # Reduce linear velocity during turns
        else:
            twist.linear.x = VELOCITY  # Normal linear velocity for straight paths

        self.velocity_publisher.publish(twist)

    @staticmethod
    def normalize_angle(angle):
        while angle > math.pi:
            angle -= 2 * math.pi
        while angle < -math.pi:
            angle += 2 * math.pi
        return angle

    def calculate_rmse(self):
        # Ensure the lengths of the intended and actual path data are the same
        min_length = min(len(self.intended_path_data['x']), len(self.actual_path_data['x']))
        intended_path = np.array(list(zip(self.intended_path_data['x'][:min_length], self.intended_path_data['y'][:min_length])))
        actual_path = np.array(list(zip(self.actual_path_data['x'][:min_length], self.actual_path_data['y'][:min_length])))

        # Calculate RMSE
        squared_errors = np.sum((intended_path - actual_path) ** 2, axis=1)
        mse = np.mean(squared_errors)
        rmse = np.sqrt(mse)

        return rmse

    def plot_path(self):
        if self.lap_data:
            for i, lap in enumerate(self.lap_data):
                plt.figure()
                plt.plot(lap['intended_path']['x'], lap['intended_path']['y'], color='blue', linestyle='-', linewidth=2, label='Expected Path')
                plt.plot(lap['actual_path']['x'], lap['actual_path']['y'], color='red', linestyle='--', linewidth=2, label='Actual Path')
                plt.xlabel('X Position (m)')
                plt.ylabel('Y Position (m)')
                plt.title('Expected vs Actual Path'.format(i + 1, VELOCITY))
                plt.legend()
                plt.grid(True)
```

```python

                # Display RMSE and lap time on the plot
                plt.text(0.05, 0.05, 'RMSE: {:.3f} m'.format(lap["rmse"]), transform=plt.gca().transAxes, fontsize=12, verticalalignment='bottom')
                plt.text(0.55, 0.05, 'Lap Time: {:.2f} seconds'.format(lap["lap_time"]), transform=plt.gca().transAxes, fontsize=12, verticalalignment='bottom')
        else:
            # Plot the path for the incomplete lap if no complete laps exist
            plt.figure()
            plt.plot(self.intended_path_data['x'], self.intended_path_data['y'], color='blue', linestyle='-', linewidth=2, label='Expected Path')
            plt.plot(self.actual_path_data['x'], self.actual_path_data['y'], color='red', linestyle='--', linewidth=2, label='Actual Path')
            plt.xlabel('X Position (m)')
            plt.ylabel('Y Position (m)')
            plt.title('Expected vs Actual Path')
            plt.legend()
            plt.grid(True)

        # Plot current lap data if the lap is not complete
        if self.actual_path_data['x']:
            plt.figure()
            plt.plot(self.intended_path_data['x'], self.intended_path_data['y'], color='blue', linestyle='-', linewidth=2, label='Expected Path')
            plt.plot(self.actual_path_data['x'], self.actual_path_data['y'], color='red', linestyle='--', linewidth=2, label='Current Path')
            plt.xlabel('X Position (m)')
            plt.ylabel('Y Position (m)')
            plt.title('Expected vs Actual Path')
            plt.legend()
            plt.grid(True)

        plt.show()

if __name__ == '__main__':
    try:
        pp = PurePursuit()
        rospy.on_shutdown(pp.plot_path)
        rospy.spin()
    except rospy.ROSInterruptException:
        pass
```