

Doctoral Thesis
Shibaura Institute of Technology

**Integrating ROS with Coppeliasim Simulator for
Development Real-Time Simulation with Dynamic Control**

2022/March
Nattawat Pinrath

Abstract

Robotic simulators can be used for testing system architecture, design, and development, and, in a wider context, for artificial creatures. The visualization tools and interfaces with the robot or simulate the operation of the robotic systems in a very realistic way design and test control algorithms for different platforms. The robotic simulator is useful for real-time simulation or actual robot control in a wider context.

This study proposes a teleoperation system for assistive control of mobile robots' movement over a narrow path. That integrates simulation and the real world via the robot operating system (ROS) framework. The teleoperation system is a combination of real and virtual devices. First, we applied the object avoidance algorithm, the Braitenberg algorithm, and the path generating module (OMPL) in CoppeliaSim. The Braitenberg algorithm is a sensor-based automatic motion designed to aid the robot operator in maneuvering through a narrow path. While the OMPL module helps create a path for the operator to control the robot into narrow spaces or intersections within limited spaces, a virtual proximity sensor is used in the simulations to fulfill the Braitenberg algorithm requirement. A real laser range finder gathers the environmental data on the simulation screen. The virtual proximity sensor and Braitenberg algorithm are applied to the simulation scene with dynamic simulation. After that, simulation scripts are written to incorporate the linear and angular velocities into an ROS for real-time robot control. By simulating an actual narrow path scenario, we validated. The results indicated that the system could merge real-time dynamic simulation with real world; the proposed system can assist the operator in narrow path environments without collision.

Next, this study proposes a system for detecting small objects and surface conditions to improve environmental awareness in areas with low communication signals, proposes a teleoperation system that integrates a CoppeliaSim robotics simulator with the real world via ROS. The proposed system is a real-time display system that can provide

small object and floor conditions then visualization in simulation. A fusion of inertial measurement unit sensor and odometry data will be sent to the simulator to display the posture of a robot. CoppeliaSim constructs an algorithm to display small object and floor conditions on the robot's path. Thereafter, CoppeliaSim uses real-time and obstacle data for dynamic real-time simulation, indicating the object's movement mounted on the robot. The results show that the proposed system can display the robot posture, small object, and floor conditions during robot move over on, and real-time dynamic simulations to assess the movement and the position of objects carried by the robot...

Contents

1	Introduction	1
1.1	Motivation	1
1.1.1	Teleoperation System	1
1.1.2	Robotic Simulation	3
1.2	Objective Proposed	4
1.3	Structure of this Dissertation	8
2	Background	10
2.1	ROS	10
2.2	Simulation Comparison	12
2.2.1	Comparison Between Gazebo and CoppeliaSim	15
2.3	Physical Engine	18
2.4	CoppeliaSim	23
2.4.1	Scene Object	24
2.4.2	Simulation Techniques	25
2.4.3	Simulation Control Architecture	27
2.4.4	Real-Time Simulation Loop	28
2.4.5	Physical Engine	29
2.4.6	Friction Model in CoppeliaSim	29
2.5	Path and Motion Planing	31
2.5.1	RRT Path Planning	35

CONTENTS

2.6	Robot Navigation	37
2.6.1	Local Motion Control	38
2.6.2	Challenges, Technical Limitations, and Analytical Comparison for Local Motion Control	42
2.7	Sensor	43
2.8	Camera	45
2.9	Laser Range Finder	45
2.10	Chapter Summary	46
3	Simulation Accuracy Experiment	48
3.1	Simulation Accuracy Experiment	48
3.2	CoppeliaSim Control Algorithm	53
3.3	Chapter Summary	57
4	Real-Time Simulator for a Semiautonomous Teleoperation Robot in an Unknown Narrow Path	59
4.1	Hardware Setting	59
4.2	Developed System Framework	60
4.3	The Braitenberg Algorithm	64
4.3.1	Braitenberg Algorithm in Narrow Space	66
4.4	Entering Narrow Space and Intersection Between Narrow Path	75
4.5	Robot Moves in Narrow Space Experiment	77
4.5.1	Robot's Movement Along with a Narrow Path	77
4.5.2	Entering the Narrow Path and Intersection in the Narrow Path Experiment	79
4.6	Experiment Results and Discussion	81
4.6.1	Robot's Movement within a Narrow Path	82
4.6.2	Entering Narrow Path and the Intersection between Narrow Pathway	87
4.7	Chapter Summary	90

5	Small Object Displays in CoppeliaSim	93
5.1	Small Object Detection System Architecture	94
5.2	Small Object Displays in CoppeliaSim	96
5.3	Experiment Setting	104
5.4	Small Object Detection Experiment	105
5.4.1	Left or Right Robot Wheel Climb on a Small Object	105
5.4.2	Robot Moves on Unsmooth Surface Experiment	107
5.4.3	Real-Time Dynamic Simulation	108
5.5	Chapter Summary	113
6	Conclusions	116
A	List of Publications	128
A.1	International Journal (peer reviewed)	128
A.2	International Conference	128
B	ROS ekf_localization_node	130
B.1	Extended Kalman Filter Node	130

List of Figures

1.1	Traditional system in Robotic development.	7
1.2	Dissertation overview.	9
2.1	Major scene objects used in the simulation.	25
2.2	CoppeliaSim control architecture. Greyed items are control entities. (1) C/C++ API calls, (2) cascaded child script execution, (3) Lua API calls, (4) custom Lua API callbacks, (5) CoppeliaSim event callbacks, (6) re- mote API function calls, (7) ROS transit, (8) custom communication (socket, serial, pipes, etc.)	26
2.3	Real-time simulation loop in CoppeliaSim.	31
2.4	OMPL Overview core motion.	32
2.5	Algorithm : Extend.	36
2.6	Algorithm : RRT.	36
2.7	Algorithm : Connect.	37
2.8	Algorithm : RRT-Connect.	37
2.9	Vector field histogram algorithm.	40
2.10	Bug algorithm.	41
2.11	Usage of fuzzy logic control.	42
2.12	The structure of Hokuyo UBG-04LX-F01	46
3.1	Simulation scene in accuracy experiment.	49
3.2	Block diagram of the position control loop.	50

LIST OF FIGURES

3.3	Position control experiment.	50
3.4	The communication system between CoppeliaSim and Real robot. . . .	51
3.5	The average position error of each dynamic engine. Blue model is a real robot position; red model is robot model. (a) Bullet 2.78, (b) Bullet 2.83, (c) ODE, (d) Vortex.	52
3.6	The average orientation error of each dynamic engine. Blue model is a real robot position; red model is robot model. (a) Bullet 2.78, (b) Bullet 2.83, (c) ODE, (d) Vortex.	53
3.7	Reduce simulation error algorithm.	55
3.8	Simulation trajectory setting.	56
3.9	Robot trajectory.	56
3.10	Position error.	57
4.1	Hardware Architecture.	60
4.2	System Framework.	61
4.3	Communication to display the real-robot position into Coppeliasim scene. . .	62
4.4	Comparing robot movement.	63
4.5	Obstacle(walls) displayed in Coppeliasim.	64
4.6	Braitenberg's vehicle 2a and 2b.	65
4.7	Braitenberg's vehicle 3a and 3b.	66
4.8	Braitenberg's vehicle modified.	66
4.9	Simulation using the Braitenberg's algorithm with sensor on front and rear. .	67
4.10	Robot moves in narrow space within front and rear sensor conditions. . .	68
4.11	Robot colliding with the walls when the sensor cannot cover all area of the robot.	69
4.12	Weight factor of left-robot wheel.	70
4.13	Weight factor of right-robot wheel.	71
4.14	Pioneer Visual sensor model.	72
4.15	The description of d_{\max} and d_{\min} value.	72

LIST OF FIGURES

4.16	Comparison of speed between the left and right	73
4.17	Virtual narrow path in Coppeliasim.	74
4.18	OMPL system.	76
4.19	OMPL path module simulation in Coppeliasim.	76
4.20	OMPL State Space.	77
4.21	OMPL path generate.	78
4.22	Narrow path environment.	79
4.23	Narrow path environment in second experiment.	80
4.24	Entering narrow path experiment environment.	81
4.25	Experiment environment.	82
4.26	Trajectories of the robot move along with the narrow path.	83
4.27	Comparing the simulation scene with the real robot scene.	84
4.28	Linear and angular velocities.	85
4.29	Trajectory of the robot during move along with narrow path in experiment 2.	85
4.30	Entering and intersection narrow path environments experiment.	88
4.31	Intersection Between Narrow path Experiment	89
5.1	Proposed system.	94
5.2	CoppeliaSim simulator displays real-time robot position	95
5.3	Block diagram of the complementary filter[75]	96
5.4	Real-time robot pose in CoppeliaSim scene.	96
5.5	Teleoperation display in CoppeliaSim.	97
5.6	Created dummy position in each robot wheel and front of robot wheel.	98
5.7	Displayed the robot pose and dummy position in each scenario.	98
5.8	Relationship between the actual robot posture and dummy points W_C, W_L, and W_R.	100
5.9	Algorithm for robot posture detection.	101
5.10	CoppeliaSim displays small object.	102

LIST OF FIGURES

5.11 Experiment setting	103
5.12 Comparison between real-robot setting and robot-model setting.	105
5.13 The CoppeliaSim displays the small object when the actual robot climbs the small object.	106
5.14 D_r value during robot move on small object.	107
5.15 Robot move down from small object.	108
5.16 CoppeliaSim displays the unsmooth floor conditions.	109
5.17 CoppeliaSim displays the difference level of the floor conditions.	109
5.18 Small objects displayed in CoppeliaSim.	110
5.19 Comparison of white box movement between estimate real-world(IMU) and CoppeliaSim.	111
5.20 Comparison of the real-time error between estimate real-world(IMU) and CoppeliaSim.	114

List of Tables

2.1	Ratings for the level of user satisfaction of the most diffused tools[32] . .	15
2.2	Simulation performance. Real-time factor(R) = simulated time/real-time ($R > 1$, simulation run faster than real-time, C = CPU usage, M = Memory usage. A small scene, where robots were put on a large 2D plane. The large scene, where an industrial building model with approximately 41,6000 vertices.	19
2.3	Overall comparison between CoppeliaSim and Gazebo	20
2.4	Comparison of engines constraints support	21
2.5	Comparison of the engines geometry support	22
2.6	Comparison of engines material support	22
2.7	Algorithm comparison	44
3.1	The difference value of estimate position and simulation values for straight movement	54
3.2	The difference value of estimate position and simulation values for angular movement	54
4.1	Narrow path simulation results	74
4.2	Comparison of the robot moves along with a narrow path	86
4.3	Experimental results in entering narrow path and Intersection during the narrow path.	91
4.4	System comparison of the robot moves in a narrow path.	92

LIST OF TABLES

5.1	The difference at a final position between actual movement with simulation results.	113
B.1	The error for sensor configuration.	132

CHAPTER 1

Introduction

This chapter introduces the comprehensive study, which includes the overall discussion on the ideas that led to this study. This chapter also includes the main objectives and contributions described to clearly outline this study's scope. Finally, the organization of this dissertation is presented.

1.1 Motivation

1.1.1 Teleoperation System

In a traditional teleoperation system, the camera is the main device to provide environmental data to an operator. The visual device, usually attached on-board or in front of the robot, supplies visual data for robot direction, which allows the operator to understand the environment in front of the robot. However, the robot's limited side and rear field of view make navigating some environments challenging. The robot's limited side and rear-view also make it hard for the operator to control the robot's movements in a narrow path without collision [1]. Since the distance between the robot and walls is small, visual information from the side and rear of the robot and the environment poses challenges in operating the robot on a narrow path. For example, at Fukushima Daiichi, a nuclear power plant was severely damaged by the Great East Japan Earthquake in 2011, in which nuclear reactor buildings and sealed containers were damaged. The damage released several radioactive substances into the atmosphere, and rescue workers could not reach the worksites to undertake work and research. Although robots can conduct activities such as exploration and rescue are necessary, many disaster areas

have narrow-width routes due to the collapse of numerous structures. It is essential to design robots to navigate such sites and traverse narrow paths [2].

Navigation through narrow areas has been extensively studied, including vehicle applications, such as ships [3, 4], autonomous cars [5, 6], and autonomous mobile robot [7, 8, 9]. However, prior research focused primarily on known environments and autonomous robot control to target points. Nevertheless, in teleoperation, almost all tasks work in an unknown environment. The limited view makes it difficult for the teleoperator to be aware of the environment. Previous research and techniques to improve environmental awareness are classified into three initiatives: first, enhancing the teleinterface of autonomous systems and supervisory controls. Human operators made efforts to enhance the visual experience; second, the improvement of the teleinterface to increase situational awareness. Researchers combine multisensory or sensor fusion for teleoperation. For example, panspheric cameras use multiple cameras [10, 11], LiDAR odometry data [12], or combine UAV with LiDAR [13]. Although these methodologies help teleoperators improve environmental awareness, they require a fast communication system to send large images with minimal delay. The third initiative is virtual reality (VR) to improve environmental awareness [14, 15]. Some researchers propose autonomous systems in the teleoperation field like adjustable autonomous control and share control [16, 17, 18]. In situation awareness, the control devices technique is a common method for teleoperation, improving environmental awareness in control devices. For example, combined haptic devices with a teleoperated mobile robot system force feedback, which corresponds to the virtual repulsive forces exerted on the mobile robot from the obstacles in its environment. Some experiments involve infrared sensors attached around the robot, which detect obstacle information and compute repulsive forces to enable the teleoperator to sense force. The control robot moves while avoiding obstacles [19]. In other experiments, a proposed occupancy-grid map computes the force exerted on the operator's hand, making the teleoperator blindly control a robot (without a camera device), avoiding a collision [20]. These systems only work in a known environment.

Teleoperation, similar to remote control, is the operation of a machine from a distance. In this study, teleoperation refers to wireless networks, wherein the operator uses information from sensors or a camera attached to the robot for its operation. There are two critical objectives regarding user interface in a teleoperation system: (1) supporting efficient and accurate navigation through the remote environment and (2) providing

the operator with a sense of environmental awareness. However, the limitations of the sensors or working environment result in difficulties for the operator to sense the environment; therefore, improving the awareness environment is essential. The system can assist in the complex environment or improve awareness using a sensor fusion or navigation algorithm.

1.1.2 Robotic Simulation

A robotics simulator helps create applications for a physical robot without depending on the actual machine, thus saving cost and time. In some cases, these applications are transferable onto the physical robot (or rebuilt) without modifications

The term robotics simulator can refer to several different robotics simulation applications. For example, in mobile robotics applications, behavior-based robotics simulators allow users to create austere worlds of rigid objects and light sources and to program robots to interact with these worlds. Behavior-based simulation allows for more biological actions than more binary or computational simulators. In addition, behavior-based simulators may learn from mistakes and demonstrate the anthropomorphic quality of tenacity.

One of the most popular applications for robotics simulators is 3D modeling and rendering a robot and its environment. This type of robotics software has a simulator that is a virtual robot capable of emulating the motion of an actual robot in a natural work envelope. Some robotics simulators use a physics engine for more realistic motion generation of the robot. The use of a robotics simulator to develop a robotics control program is highly recommended regardless of whether an actual robot is available or not. The simulator allows for robotics programs to be conveniently written and debugged offline, with the final version of the program tested on an actual robot. This primarily holds for industrial robotic applications only since the success of offline programming depends on how similar the real environment of the robot is to the simulated environment. Sensor-based robot actions are much more difficult to simulate and/or program offline since the robot's motion depends on the instantaneous sensor readings in the real world.

A simulation is an essential tool in many research fields, especially robotics. Since the 20th century, robotic simulation applications have improved due to increasing CPU computing power and software and hardware performance. Simulators test system archi-

texture, design, development, visualization tools, and artificial creatures' applications. The robot operating system (ROS) recently emerged as a de-facto standard for building robot applications [21]. Robotic simulators integrate with real-world robots or devices to simulate and control robots using actual data from the real world. In 2015, a study titled "Bimanual Haptics for Humanoid Robot Teleoperation using ROS and Coppeliasim" was presented [22]. The study used an interface-bridged haptic device and a simulator based on a distributed ROS network. It demonstrates the possibility of several balancing-tasks gathering and simulation characteristics sufficiently supporting the development of an infrastructure to operate a real-world robot. In 2016, a multi-UAV simulator based on ROS and Unity3D was presented in [23]. The authors propose a system that integrates a simulator with the ROS and a game engine. Particular emphasis is placed on modeling different environments and sensors, particularly collisions and LiDARs, achieving best-virtualized reality and high frequency. A 3D modeling and simulation of a crawler robot in ROS/Gazebo, an approximation of track-terrain interaction of an underground crawler vehicle, is proposed, along with the modeling and simulation of Russian crawler robot "Engineer" with ROS/Gazebo and the visualization of its motion in ROS/RViz software. Therefore, the robot operating system framework is applied to combine the virtual world(CoppeliaSim) with the real world. The technique combines a virtual model with real-time data to assist operators in controlling the robot over long distances.

There are several robot simulation platforms. Gazebo is the official simulator platform for the robot operating system(ROS), it can easily communicate with the ROS, thus reducing the modeling steps. The CoppeliaSim simulator by CoppeliaSim Robotics can be installed and run without a powerful graphics card, and it does not require a powerful CPU. Although connecting CoppeliaSim with the ROS requires more simulation settings than Gazebo, CoppeliaSim has fewer CPU demands. Furthermore, CoppeliaSim offers path/motion planning functionality via a plugin wrapping the OMPL library. The OMPL library consists of many state-of-the-art sampling-based motion planning algorithms.

1.2 Objective Proposed

- Creating a system to help the robot operator control the robot in narrow paths.

Creating a system to help the robot operator control the robot in narrow paths. In

remote robotics, poor situational awareness is a problem in remote robot control. To enhance the operator’s situational awareness, efforts to improve visual information or enhance the ability to perceive the environment. However, increasing environmental data increases the workload of operators. Especially in situations where the robot is close to obstacles for a long time, such as moving through a long narrow path, or situations where there are many obstacles, such as Fukushima Daiichi, a nuclear power plant, was severely damaged by Great East Japan. Earthquake in 2011. Rescue workers could not reach the work sites to undertake work and research. Although robots can perform activities such as exploration and rescue, many disaster areas have narrow-width routes due to the collapse of numerous structures. Some studies propose the autonomy or intelligence system to improve teleoperation. Some autonomy-based approaches to teleoperation include shared control [2], safeguarded control [21, 22], adjustable autonomy [23, 24, 25, 26] and mixed initiatives [24, 27, 28]. One limitation of these approaches is that some robot control is taken away from humans. This limits the robot to preprogrammed behaviors and intelligence. Especially in a narrow space where the robot is close to obstacles, its effect on an operator cannot control the robot with design direction. The performance of the autonomy system requires the complex setting with sensor devices and high computation of robot that affect operator time. Therefore, we propose a system that can assist the semiautonomous system. The operator can control the robot in a narrow space, and the operator can switch the mode to control the robot into a design position in a narrow space.

- Combine real-world with robotic simulation for real-time simulation.

The simulation experiment is crucial as it is a strategic method in all fields, particularly in the robotics field [24]. The presence of robotics simulation tools as a platform for featuring the virtual robot helps the researcher and developer present the prototype of the real robot. Simulation software act as a medium to instantiate and define the robustness of the novel algorithm, and concurrently, it can reduce the development time and cost of multi-robots. It is essential to test the robustness of robots on the simulation before implementing it to the real robot. For instance, the development of robotics using simulation software helps the researcher and developer discover new approaches instead of facing hardware failures repeatedly. In other words, simulation software can aid in solving the complexity issues in robot

simulation. However, in the traditional robotic system, the simulation used in the prototype study phase, and then implemented in the real world see in Fig. 1.1a, in this system developer must be a coding process to transfer the algorithm from the simulator into the real world, ie. Coppeliasim robot simulator developed by LUA language API; however, the real-robot system support C language. In addition, the system requires a simulation setting process. The simulation model and the environment should be the same as the actual condition to improve the accuracy of the simulation results. For another system-developed setting, see in Fig 1.1b. The user-developed into another software and then interface with simulation to verify the algorithm in simulation and interface with real-robot.

However, in developing the simulation separated from the real world, developed working finished in the simulation scene and then implemented in the real world. For example, in [25] and [26] created the virtual environment as same as the real world for verified algorithm and then applied in the real world. The named tests need experiment procedures that imply imagination and materialization of various scenarios. It was evident that a strategy needed to deal with is a time-consuming one—the problem intended to be solved is how to shortcut these developing scenarios. More precisely, a use case – will be presented to exemplify the strategy that replaces the experiments with simulations. Therefore, we propose the system to integrate the robotic simulator with the real world. Coppeliasim simulation connects with the robot via a standard robot framework (ROS). We have used the dynamics engine to control the robot model in a virtual environment. The linear and angular velocities in the robot model are used to command the real robot simultaneously. We validate the simulator’s accuracy and create the algorithm for controlling the real robot.

- Applied the virtual sensor to solve the limitation of the Britenberg algorithm.

A Braitenberg algorithm is a concept that can autonomously move around based on its sensor inputs. It has primitive sensors that measure stimulus at a point, and wheels (each driven by its motor) act as actuator effects. Originally the Britenberg algorithm was used as to object avoidance algorithm in an unknown environment. However, [27] applied the Britenberg to navigate the robot into the design point. The study has shown the robot move through the straight-corridors

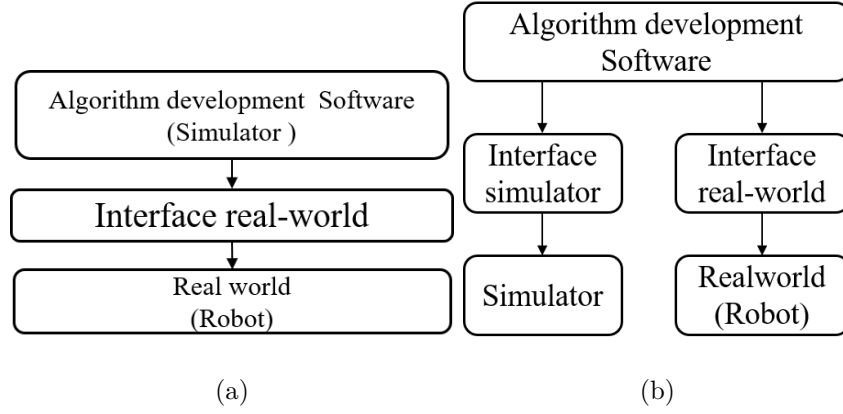


Figure 1.1: Traditional system in Robotic development.

into the design point without colliding obstacles. However, the limitations of Britengerg require more sensors, which increases the cost and power consumption of the robot. Additionally, the limits to preprogrammed activities and intelligence. The setting program depends on the number and location of sensors. Therefore, applying the Braitenberg algorithm to the robot is not easy. However, applying a virtual sensor will help developers experiment with and reduce the damage done to the robot. Additionally, virtual sensors can be placed in any robot part, which is impossible in real life.

- Applied the autonomous control(OMPL) into a teleoperation system.

The autonomy teleoperation system is designed to improve the teleoperation system. For example, safeguarded control, adjustable autonomy, and mixed initiatives. However, this approach has taken away some of the robot's control from humans. In some situations, the operators may know more than robots. In addition, it is not easy to design an automated system so that a robot can handle every conceivable situation. Therefore, we offer a system that focuses on helping operators operate the robot in narrow spaces or narrow passages. The system consists of 2 modes for the assistive operator in narrow path scenarios of the robot controller and the system for the assistive robot at intersections between narrow spaces or entering narrow path situations. The operator can change the control mode at any time via the Joystick.

- Developed the algorithm to display the small object during teleoperation robot.

Almost all related studies use sensors to determine small objects, small steps, or

unsmooth surface floors. However, vision sensors suffer from illumination change and occlusion; multiple cameras are needed to complement the incompatibility of range of view and sensing resolution. Embedding so many cameras so that everything is visible would need enormous human resources and costs due to initial setups and later maintenance of wiring, fixture, calibration, lighting control. In addition, the system requires high video resolution and three-dimensional(3D) environment data, which necessitate speedy data transfer from the robot to the operator's position and encounter problems when the operator is far from the robot or in an environment where data transmission is a problem. Furthermore, a small positioning sensor system requires appropriate environmental conditions to determine the location of smaller objects; this results in inaccuracy problems in some environments, for instance, controlling the robot inside a dark tunnel or insufficiently lit surfaces under a building. In addition, the development of a remote robot control interface focuses on improving the robot's perception of the environment. The system was designed to recognize the environment around the robot, but the robot occasionally needs to carry equipment. It is difficult for the operator to understand the state of the equipment that the robot carries, as the robot goes through different situations.

In this study, we integrate a sensor communication from an inertial measurement unit (IMU) sensor and odometry data attached to a robot to display the robot's real-time posture. We created an algorithm to detect when the robot encounters an obstacle by fusing the odometry data and data from the IMU sensor. The algorithm is applied to display the floor conditions when the robot moves over small objects or moves over rough floors. Teleoperation system is applied to the physical engine for real-time simulation of the equipment on which a robot is mounted. In this system, the CoppeliaSim simulator displays the robot's real-time posture, displaying obstacles (small objects) and the condition of the equipment the robot is mounted on, which sensor devices cannot displace.

1.3 Structure of this Dissertation

The rest of the dissertation is organized as follows: Chapter 2 explains the ROS system, discusses the current status of the robotic simulator, and compares performance between

each robotic simulation. Then describe the operation of the CoppeliaSim simulator, the simulation step, and various functions of the simulator. Chapter 3 discuss the simulation accuracy experiment. We experiment with the concept of communication between real devices with the visual world in CoppeliaSim via ROS framework and validate the accuracy of the physical engine in CoppeliaSim in terms of the robot position and orientation; after that, we test the control algorithm. Chapter 4 talks about the system to assist the operator in controlling the robot and narrow space. We discuss the Braitenberg algorithm how to apply the algorithm with a virtual proximity sensor into an experiment. The chapter demonstrates small object detection, starting with the system architecture and small object detection algorithm, and then demonstrates the experiment results. Finally, Chapter 6 concludes the study and discusses future works opportunities. The overview of the dissertation shows in Fig 1.2

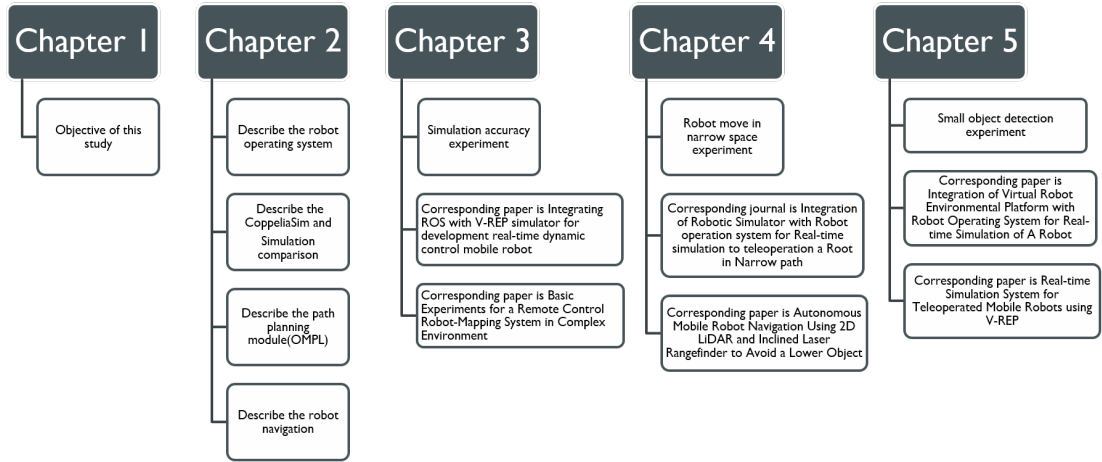


Figure 1.2: Dissertation overview.

CHAPTER 2

Background

This chapter discusses the CoppeliaSim and ROS system, which relate to the study. The chapter focuses on explaining the functions of the ROS and robotic simulation on the regions. Then, it discusses the comparison between the robotic simulator and the advantage and disadvantages of the well-known robotic simulator in the ROS system.

2.1 ROS

Robots generally employ complex electronics, mechanics, and software. The software side has the task of coordinating all of the sensing and actuation components while working on or more general objectives of the robotic system. A common approach to handle these distinct tasks is to employ a layered architecture based on complexity/abstraction level. Different layers are separable on various computing boards. At each abstraction level, there are well-studied algorithms that ensure optimal performance. Integrating each algorithm into the control architecture can be complex and error-prone due to different programming languages, communication protocols, and numeric scales.

Robotics middleware interconnects specialized software components using common communication channels. The robot operating system (ROS) is an open-source software framework that offers a structured communication layer, a packaging system for distributing ready-made algorithm implementations, and many special-purpose applications.

Interfacing a robot with ROS exposes the robot's sensors and actuators to the ROS communication network. After this step, the robot can be controlled using various control

algorithms that need not be initially designed for that particular robot. As an example, consider a two-wheel mobile robot and a drone. However, the movement mechanisms differ significantly. A generic movement message stores linear and angular velocities for all three axes within the ROS, allowing algorithm deployment on an extended range of devices without modifications.

The four essential items that compose a ROS system are nodes, messages, topics, and services. Nodes can be compared to operating system processes where both have a unique ID, receive parameters, and execute one or more subtasks to run an application. Messages represent a previously agreed-upon data format used by nodes when exchanging information. A message can be as simple as a single integer value or as complex as camera images and 3D point clouds. The ROS framework provides a set of standard messages, but any application can define custom messages if necessary, using a language-neutral paradigm. Topics are a means of identifying information flows in the publisher model used in ROS. Its name and the message type describe each topic that it allows. This identification becomes necessary when dealing with complex systems that transfer various information, as there can be multiple publishers and multiple subscribers [28]. This communication method is asynchronous.

Services are used for synchronously exchanging data using a request-reply model. A client requests a server to do a specific task and then waits for a response. As with topics, services are also identified uniquely in the system and are composed of two message types, one for the request and one for the response [28].

The ROS system allows a developer to construct a distributed robotic system by connecting multiple nodes locally or through a network. The free and open-source license of the ROS has allowed it to receive contributions from many sources. One notable example of a ROS package is `tf`. This package offers both a library and inspection/debug applications. That is used mainly for geometric tracking and transformations, at present and in the past [29].

These components and features have determined that researchers use ROS for a wide range of applications that target beginners and advanced users. One example for the former user category is `rosbridge`, a secondary middleware on top of ROS that allows remote users to interact with ROS using web technologies such as browsers, Javascript, and WebGL, in an attempt to lower the robotics knowledge entry barrier for general

application developers [30]. At the other end of the scale are robotics applications with tight requirements such as long-term autonomy and reliability, operating without human intervention. An example in this sense is the STRANDS project, which aims to provide a robotic platform capable of long-term deployment in indoor environments to monitor and model the environment while also generating alerts for unusual behavior [31].

2.2 Simulation Comparison

This section will discuss two essential simulation parameters: the simulation program and the physical engine. In robots, dynamics simulators have more stringent requirements than virtual character animations, where computational and physical accuracy may be less restrictive. One of the critical issues is numerical stability, which puts significant limitations in using simulations in real-time control settings to be helpful as a prediction tool in a real-time control loop. The simulator must be very fast in dynamic calculations and must guarantee physically possible convergence of solutions within a limited time frame. Since the 20th century, robot simulation applications have improved due to increased CPU, software, and hardware processing performance, which has minimized the problem. Nowadays, there are several robot simulation platforms.

- OpenRave is an environment for simulating motion planning algorithms for robotics. It contains several models of industrial robots and targets robotics automation.
- Robotron is software that generates symbolic models of multi-body systems, which can be analyzed and simulated in Matlab and Simulink. It is developed by the Center for Research in Mechatronics, Universite Catholique de Louvain.
- Vortex Dynamics is a software developed by CM Labs, specialized in simulating contact dynamics in different operating environments (e.g., terrain, water). It is coupled with a 3D editor for creating mechanisms, robots, and scenes and is particularly adapted to the simulation of vehicles and cable systems.
- OpenSim is a toolkit for muscle-skeletal modeling, and dynamic movement simulation developed at Stanford University and supported by the US NIH and DARPA. It is freely available, open-source, and extensible through user plugins. The physics

engine of this project is SimBody, an open-source C++ API implementing Featherstone's algorithms for rigid body mechanics with the support of different contact models.

- MuJoCo is a dynamics engine developed mainly by E. Todorov and is now the property of Roboti LLC. It is one of the most recent physics engines conceived for simulating robotics and bio-mechanical systems. It supports parallel computations, provides inverse dynamics with contacts and equality constraints, implements several contact dynamics. It is suitable for control optimization in real-time within a control loop.
- XDE is an interactive physics simulation software environment developed by CEA LIST for virtual reality applications in industrial contexts. MOBY is an open-source physics simulation library mainly developed by E. Drumwright. It is a multi-body dynamics simulation library with several features for accurate simulation of robot dynamics: multiple integrator types, two convex solvers, support for deformable bodies, several contact models. XDE is an interactive physics simulation software environment developed by CEA LIST for virtual reality applications in industrial contexts.
- MOBY is an open-source physics simulation library mainly developed by E. Drumwright. It is a multi-body dynamics simulation library with several features for accurate simulation of robot dynamics: multiple integrator types, two convex solvers, support for deformable bodies, several contact models.
- OpenERP is a system simulator developed in Japan for the HRP robots. Interestingly, it consists of several modules (a dynamics simulator, a control interface with the robot, a collision detector) and can be used for a seamless simulation/control of the robot.
- Argos is a multi-robots simulator that has been designed to simulate heterogeneous swarms of robots in real-time. Its modular approach allows the user to add custom features in a virtual environment. Its design is pretty different from the other simulators. Its most distinctive feature is that the 3D simulated world can be divided into regions, and each region can be assigned to a different physics engine. All components of Argos are plugins (robot models, sensors, actuators, physics

engines, visualizations). The users can extend or override the existing plugins to build robots and implement the simulations. Argos is available for Linux and macOS but not for Windows due to Windows' DLL format limitations.

- • Webots is a development environment used to model and simulate mobile robots. A comprehensive list of simulated sensors and actuators is available to build the robots in a virtual world. Their controllers can be programmed inside the IDE or with third-party development environments. The controller programs are transferable to commercially available real robots. Webots have been used in many universities and research centers worldwide for over 19 years. Webots are available for Linux, macOS, and Windows.
- • Gazebo started as a project at the University of Southern California. Thereafter, it was integrated into the ROS framework by John Hsu, a senior researcher at Willow Garage, the original maintainer of ROS. Since then, Gazebo has been maintained by Open Source Robotics Foundation, which is a Willow Garage spin-off and the same maintainer that takes care of ROS. The Gazebo is an entirely open-source project, available to anyone under the Apache 2.0 license.

Although many robot simulators are available, some are designed for specific purposes, and some simulators have special features. In the ROS framework, Gazebo emulators are officially supported. Therefore, we will compare the pros and cons between CoppeliaSim and Gazebo emulators.

Serena Ivaldi, 2014 (Tools for simulating humanoid robot dynamics)[32]. They're online survey for asked user opinion, comparison user rating under 4 categories.

- Document: Detail and level of understanding of manual
- Support: User support from developers. The speed of answering questions Difficulty contacting developers
- Installation: The difficulty of installing the program
- Tutorial: Number of examples on the internet

They are surveyed with 119 participants as;

- 92% Male and 8% of female.

Table 2.1: Ratings for the level of user satisfaction of the most diffused tools[32]

	Document	Support	Installation	Tutorials
CoppeliaSim	4.28 ± 0.76	4.43 ± 0.79	4.71 ± 0.76	4.14 ± 0.9
Gazebo	3.47 ± 0.99	4.00 ± 1.07	3.93 ± 1.03	3.53 ± 1.12
Webot	3.86 ± 1.07	3.57 ± 1.13	4.43 ± 0.79	3.43 ± 1.51
ARGoS	3.40 ± 0.70	3.90 ± 0.99	4.70 ± 0.48	4.20 ± 0.63

- Age average 32 ± 6 .
- 62% have PHDs. and 35% have MS and BC.
- Participants from USA 19%, France 17%, Italy 10% and Germany 9%.
- All participants simulated in robot interaction with the environment.

The results show in table 2.1 the rate each element on a scale from 1 to 5.

2.2.1 Comparison Between Gazebo and CoppeliaSim

In [33] the study compared the differences between the two robots and divided them into four categories.

- World Modeling: CoppeliaSim is easier to simulate than Gazebo (scenes and environments where robots move). CoppeliaSim is the easiest way to find its drag and drop, that is, to edit the model. It comes with many model files that can be easily inserted into these scenes. Models range from basic objects such as walls, doors, furniture, and more. It also has an easy-to-use scene graph visualization. All objects in a scene are accessible, and features are inspected and edited. Which Gazebo does not as well as CoppeliaSim. However, CoppeliaSim does not offer many world models for immediate use. However, there are tools to help design environments or infrastructure. There are also three simple geometric shapes. The shapes to be inserted into the scene- spheres, cubes, and finally cylinders. Furthermore, Gazebo has a function that allows access to online models and a database of community-developed models; this is one of its strengths. However, this database is quite fragmented and difficult to access.

The Gazebo models cannot be edited because external 3D modeling tools such as Blender or Google Sketchup are required to draw the models and then import them into the Gazebo format. Gazebo uses an XML format called SDF to use Gazebo to its full potential. Users are encouraged to learn how SDF works and relates to ROS URDF, a similar format used by ROS. CoppeliaSim has a URDF import tool included, while CoppeliaSim offers a more user-friendly feature for world modeling that does not require any XML knowledge. It is ideal for rapid prototyping setups, simulations, and use cases. More complex, if Gazebo is used instead, it is necessary to dig deeper into the SDF specification to create a non-basic setup. After understanding the learning curve, it is possible to create a very complex simulation. Following results, CoppeliaSim has better for new-user with low learning curve and is more stable than Gazebo [33].

- **Programmatic Control:** Gazebo and CoppeliaSim required for testing are available as ROS services provided by the ROS node associated with each simulator. These services include starting and stopping simulating robot gestures in scenes and setting up model gestures. In Gazebo, it is necessary to acquire and reset robot gestures due to service issues that reset simulations. This is not necessary for CoppeliaSim as the emulation reset service resets the robot's position with the ROS service, which is easily retrieved through the ROS node code. Libraries are used as part of the ROS and do not belong to any emulator. So, ROS can summarize abstract low-level applications, which facilitates user workloads. However, Gazebo and CoppeliaSim offer API code that provides complete control over the internal replication variables through user calls. The documents relating to each item are contained in the website, so it is fair to say in this criterion there is nothing to distinguish between Gazebo and CoppeliaSim[33].
- **ROS Integration:** Although it is a separate project, the Gazebo is the default emulator used in the ROS framework. However, there are also packages for Gazebo in the official ROS repository (ros-indigo-gazebo-ros). The Gazebo moderator, the Open Source Robotics Foundation, maintains this package, which contains plugins connecting ROS and Gazebo. It connects to object scenes and provides easy ROS communication methods such as topics published and subscribed by Gazebo and services. Packaging Gazebo as a ROS node also allows for integration with

ROS initiation methods. Easily for running large and complex systems known as launch files. In contrast, CoppeliaSim does not have native ROS nodes for it. This means that it still cannot run as part of the ROS system. Single executable file but on the side in other Linux Terminal, this is not much of a problem. However, a CoppeliaSim is required. LUA script to create a publisher to send the readings data into the ROS topic. Both Gazebo and ROS are open sources that can be achieved from a community of developers. The Gazebo was planned from scratch, thinking about integrating with ROS but CoppeliaSim is not far off. They offer a comprehensive API to access all code functionality, including ROS-specific features such as services and topics, subscriptions, and publishing. Another essential factor in this comparison is that Gazebo and ROS already have large community development bases. On the other hand, Plugins and code CoppeliaSim will provide duplicate capability files. However, there are no components that are ready to use. As a result, Gazebo has better features in terms of ROS integration[33].

- CPU usage

Reference [33] compared how much CPU power each simulator required. They created the experiment, with repeated simulation iterations, with five robots in each to compare CoppeliaSim and Gazebo. Each simulation setup consisted of a ROS master node process with 5 ROS control nodes. One ROS node process is for the genetic algorithm node and one process for the simulator. System monitoring tools are used. The results show that the CoppeliaSim emulation configuration used approximately 267 percent CPU power in its most demanding stage. The Gazebo setup continuously used 311 percent of CPU power. This shows that Gazebo has more hardware requirements. The CoppeliaSim simulation is split between active and inactive phases, while the Gazebo simulation is constantly active.

However, In [34] was found that CoppeliaSim could not be used in the case of multiple robots (more than 10), and compared to Gazebo, CoppeliaSim was found to require a higher CPU when simulating stand-alone see in Table 2.2. However, CoppeliaSim automatically spawns new threads on multiple CPU cores and, therefore, utilizes the necessary CPU power. It is, therefore, suitable for high-precision modeling of robotic applications.

Table 2.3 summarizes the comparison between CoppeliaSim and Gazebo. The table comparison of the overall performance, comparing user-friendliness in categories. The better performance is in the green column. The results show CoppeliaSim is more user-friendly than Gazebo. And the stability of the program is higher, which is suitable for use in building systems or simulating real-time operations.

2.3 Physical Engine

A dynamic physical engine plays a crucial role in the simulator’s accuracy in a robotic simulator. Recently there has been a marked increase in the number of free, publicly available physics engines. For a game developer, many aspects come into consideration, including available features, supported platforms, ease of use, and run-time performance. Most physics engines have a particular target application to which they are optimized. Most physics engines are used as middleware. A typical project will already have a target platform and budget based on other factors with a comparison matrix of the different engines licenses, costs, and supported platforms.

Table 2.4 indicates the types of constraints supported by each engine. Provided an engine supports the generic six degrees of freedom constraint, then all other constraints can be constructed. The vehicle column in the table indicates whether vehicles are supported native by the engine. This does not mean that the engine supports constraint-based vehicle models instead of ray-cast vehicles. Provided the application developer has the necessary skills, all custom constraints can be constructed from the generic constraint. However, it is uncommon for a developer to implement more than one custom constraint for their application.

All engines provide an interface for a generic constraint. ODE has no explicit support for a generic constraint; however, it can be easily modified to simulate any custom constraint as it is an open-source project. No physics engine supported all constraints. The different geometry supported by each engine is indicated in Table 2.5. indicates different geometry supported by each engine. Provided an engine supports a static triangle mesh, other meshes such as heightfields can be easily stimulated. However, it may provide inferior performance to engines that natively support height fields. Since physics engines typically require only an essential geometry representation for simulated objects, simple geometries are usually sufficient provided they can be combined in a compound

Table 2.2: Simulation performance. Real-time factor(R) = simulated time/real-time ($R > 1$, simulation run faster than real-time, C = CPU usage, M = Memory usage. A small scene, where robots were put on a large 2D plane. The large scene, where an industrial building model with approximately 41,6000 vertices.

	CoppeliaSim	Gazebo
1 robot with small scene	$R > 1$ $C = 190\%$ $M = 225 \text{ MB}$	$R > 1$ $C = 100 + 9\%$ $M = 225 \text{ MB}$
5 robots with small scene	$R = 0.37$ $C = 395\%$ $M = 360 \text{ MB}$	$R > 1$ $C = 100 + 19\%$ $M = 305 + 58 \text{ MB}$
10 robots with small scenes	$R = 0.099$ $C = 400\%$ $M = 530 \text{ MB}$	$R > 1$ $C = 100 + 30\%$ $M = 402 + 58 \text{ MB}$
50 robots with small scene	Cannot	$R = 0.87$ $C = 100 + 105\%$ $M = 402 + 58 \text{ MB}$
1 robot with large scene	$R = 0.53$ $C = 200\%$ $M = 225 \text{ MB}$	$R > 1$ $C = 100 + 10\%$ $M = 264 + 58 \text{ MB}$
5 robots with large scene	$R = 0.1$ $C = 400\%$ $M = 310 \text{ MB}$	$R > 1$ $C = 100 + 25\%$ $M = 333 + 58 \text{ MB}$
10 robots with large scene	$R = 0.036$ $C = 400\%$ $M = 460 \text{ MB}$	$R > 1$ $C = 100 + 40\%$ $M = 425 + 58 \text{ MB}$
50 robot with large scene	Cannot	$R = 0.57$ $C = 100 + 100\%$ $M = 1450 + 426 \text{ MB}$

Table 2.3: Overall comparison between CoppeliaSim and Gazebo

	CoppeliaSim	Gazebo
System requirement	Available for MacOS, Linux and Windows. Binary packages are available for all platforms.	Available for MacOS, Linux, and Windows. A binary package is only available for Linux Debian. Installed via the command line using third-party package managers on other systems.
Physical engine	Bullet 2.78, Bullet 2.83, ODE, Vortex and Newton.	Only the ODE physics engine
Object modified	Meshes can be manipulated (e.g., cut) by robots in real time.	No mesh manipulation is available.
Model	Provides a large variety of robots, including bipedal, hexapod, wheeled, flying and snake-like robots	A less diverse library of default robots, that mostly includes wheeled and flying robots. Third-party robot models are available, but their documentation is often poor.
Import model	Meshes are imported as collections of subcomponents. It is, therefore, possible to manipulate individual parts of an imported model and change their textures, materials, and other properties.	Meshes are imported as single objects. Models that contain multiple subcomponents have to be assembled in Gazebo from multiple DAE files, each corresponding to one subcomponent.
Model Modified	Able to simplify, split and combine meshes	Meshes cannot be changed
Simulation File modified	Simulation results save in coppelia file.	A scene is saved as an XML file, possible to modified in another software.
Plugin	Various options for programming functionality, including scripts attached to robots, plugins or separate programs that connect to CoppeliaSim via the RemoteAPI.	Functionality can be programmed either as compiled C++ plug-ins, python and as ROS
Model simulation setting	Scripts can be included in robot models and are often used to describe the models and their capabilities	It is difficult to recognize how a third-party robot model works.
Software Stability	No freezing issues with the interface were experienced.	The interface froze a number of times and the program, and sometimes the computer, had to be restarted
UI interface	All functionality is fairly intuitive and follows general conventions known from similar applications.	The UI usability is relatively low

Table 2.4: Comparison of engines constraints support

Engine	Corkscrew	Distance	Fixed	Generic	Prismatic	Re-volute	Spherical	Universal	Vehicle
AGEIA PhysX/Novodex	n	y	y	y	y	y	y	n	y
Bullet	n	n	y	y	y	y	y	y	y
JigLib	y	y	y	y	n	y	y	n	y
Newton	y	n	n	y	y	y	y	y	y
Open Dynamics Engine	n	n	y	n	y	y	y	n	y
Tokamak	n	n	n	y	n	y	y	n	n
True Axis	n	n	n	y	y	y	y	n	y

object that estimates the simulated object. The material properties supported by each engine are presented in Table 2.6. For gaming, applications are usually sufficient if static friction and restitution are available. The AGEIA PhysX physics engine provides several additional features not indicated in the tables above. It is the most full-featured engine provided in this comparison. Since it is a commercial engine, the implementation details are unknown. However, fixed and variable time steps are possible. It provides additional joint constraints, including cylindrical, a point on the plane, a point on the line, springs, and pulleys. Several vehicle representations are provided, and a dynamic triangular mesh geometry is also provided. An-isotropic friction is supported for the materials. The engine includes several advanced features, such as fluids, character controllers, swept geometries, soft bodies, cloth, and serialization API, and advanced hardware support AGEIA's own physics processing unit.

Historically the PhysX engine derives from a previous offering named Novodex, which was then updated to include support for AGEIA's custom hardware, and incorporated technology obtained from purchasing Meqon. However, the Novodex API naming convention was retained. For this reason it is typically referred to as Novodex. A relative newcomer to the physics scene is the Bullet physics library. For this reason it does not

Table 2.5: Comparison of the engines geometry support

Engine	Box	Capsule	Cylinder	Cone	Convex Mesh	Compound Object	Height field	Plane	Sphere	Triangle Mesh
AGEIA PhysX/Novodex	y	y	n	n	y	y	y	y	y	y
Bullet	y	y	y	n	y	y	y	y	y	y
JigLib	y	y	n	n	n	y	y	y	y	y
Newton	y	y	y	y	y	y	n	n	y	y
Open Dynamics Engine	y	y	y	n	n	y	y	y	y	y
Tokamak	y	y	n	n	y	y	n	n	y	y
True Axis	y	y	y	n	y	y	n	n	y	y

Table 2.6: Comparison of engines material support

Engine	Static Friction	Kinetic Friction	Restitution
AGEIA PhysX/Novodex	y	y	y
Bullet	y	y	y
JigLib	y	y	n
Newton	y	y	y
Open Dynamics Engine	y	y	y
Tokamak	y	n	y
True Axis	y	n	y

currently provide many additional features, the only feature included not listed in the tables above is a support for swept geometries and swept collision detection. It is a hybrid impulse and constraint-based engine that supports both variable and fixed time steps. It also includes a partial graphics processing unit (GPU) physics implementation. The Newton Game Dynamics physics engine is also a closed engine, so the implemen-

tation details are unknown. It provides a few additional features such as buoyancy, an additional “up-vector” constraint, an adaptive friction model, examples of various custom constraints, and a rag doll container.

The Open Dynamics Engine is a constraint-based physics engine that uses an Euler integrator and fixed time step. It provides an additional 2D constraint and has been ported to many platforms.

Tokamak is an impulse-based engine that uses an Euler integrator and fixed time stepping. Additional features are a container for animated bodies and support for breakable joints.

True Axis is another closed engine; however, it provides the source code in obfuscated form. This has both advantages and disadvantages since it is the simulation developers’ responsibility to build an optimal library. Microsoft’s Visual Studio 2005 was used to build True Axis for this evaluation. True Axis provides a few additional features, including a line list constraint, serialization functionality, and swept collision detection

2.4 Coppeliasim

Coppeliasim is a hybrid simulation designed around a versatile architecture combined with various relatively independent functions. Coppeliasim is no central function. Users can create simulations by enabling/disabling and calling functions at any time. For example, in a robot arm situation, the robot arm picks up some object and transports it to another location. In executing the grasping, holding, and dropping motions, Coppeliasim computes using dynamic modules, while a kinematic simulation is used for the other parts of the cycle. The approach makes it possible to quickly and precisely calculate the robot’s movement if its environment does not otherwise influence the robot. In addition, in the case of a humanoid robot, Coppeliasim handles leg movements by (a) first calculating inverse kinematics for each leg (i.e., from a desired foot position and orientation, all joint leg positions are calculated); and then (b) assigns the calculated joint positions to be used as target joint positions by the dynamic’s module. This approach allows simple simulating humanoid movement by defining the pathway while calculating the rest will be taken care of automatically.

2.4.1 Scene Object

CoppeliaSim scene contains several scene objects or elemental objects that are assembled in a tree-like hierarchy see in Fig 2.1. The following scene objects are supported in CoppeliaSim.

Joints: joints are kinematic lower pairs that link two or more scene objects together with one to three degrees of freedom. CoppeliaSim comes with four types of joints. revolute joints, Prismatic joints, Screws, and Spherical joints.

Shapes: shapes are triangular meshes used for rigid body simulation and visualization. They can be optimized for fast dynamic collision response calculation by grouping primitive or convex shapes. Other scene objects or calculation modules heavily rely on shapes (proximity sensors, the dynamics module, or the mesh-mesh distance calculation module, for example).

Proximity sensors: they perform an exact minimum distance calculation to the part of a shape contained in a configurable detection volume instead of simply performing detection based on rays. This results in a more continuous operation and thus allows for more realistic simulation.

Vision sensors: vision sensors allow the extraction of complex image information from a simulation scene (colors, object sizes, depth maps, etc.). A built-in filtering and image processing function enable the composition of filter elements blocks. Vision sensors make use of hardware acceleration for the raw image acquisition (OpenGL).

Force sensors represent rigid links between shapes that can record applied forces and torques and can conditionally break apart when a given threshold is overshot.

Graphs: graphs can record various predefined or custom data streams. Data streams can then be displayed directly (time graph of a given data type) or combined with each other to display X/Y graphs or 3D curves.

Dummies: a dummy is a reference frame for various tasks and is mainly used in conjunction with other scene objects, and as such, can be seen as a helper.

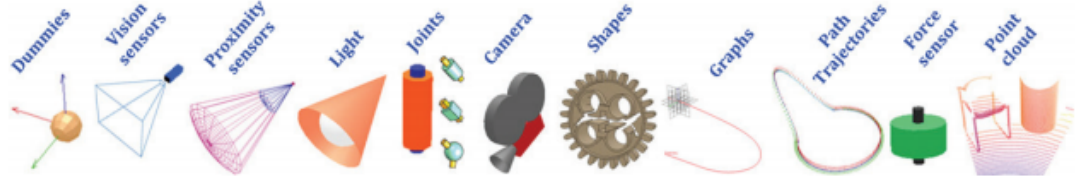


Figure 2.1: Major scene objects used in the simulation.

2.4.2 Simulation Techniques

CoppeliaSim allows the user to choose among various programming techniques simultaneously see in Fig 2.2. Generally CoppeliaSim able to users making simulation by 3 technique.

The control code is executed on another machine. This could represent a distinct machine or a robot connected to the simulator machine via a specific network (e.g., socket, serial port, etc.). The main advantage of this approach is the originality of the controller (the control code can be native and run on the original hardware). Another advantage is the reduced computing load on the simulation machine. On the other hand, this approach imposes serious limitations regarding synchronization with the simulation loop and the communication delay/lag dictated by the network.

The control code is executed on the same machine but in another process (or another thread) than the simulation loop. It can benefit from a reduced or rather balanced load on the CPU cores, but this comes accompanied by a lack of synchronization with the simulation loop. And most of the time, it comes in pair with a communication lag or thread switching delay (many resources require locking before access, or some algorithms aren't reentrant). This control technique is often implemented via external executables or plugins loaded by the simulator.

The control code is executed on the same machine and in the same thread as the simulation loop. The main advantage of this approach is the inherent synchronization with the simulation loop, and the absence of any execution, communication, or thread switching lag or delay. However, it is only possible with an increased load on the simulation loop CPU core. This control technique is often implemented via plugins loaded by the simulator.

The most common implementations of the above techniques (i.e., using external exe-

cutable or plugins) have as a direct consequence poor portability and poor scaling of simulation models: indeed, since the control code is not attached to its respective simulation model, it will have to be distributed/compiled/installed separately. This increases compatibility problems across platforms, as well as conflict/dependency issues with other libraries. Flexibility is also reduced since one must recompile and reload an executable/plugin for each small code modification. Model duplication, as in a multirobot simulation scenario, will have to be supported via hardwired mechanisms that launch new control instances for each simulation model instance.

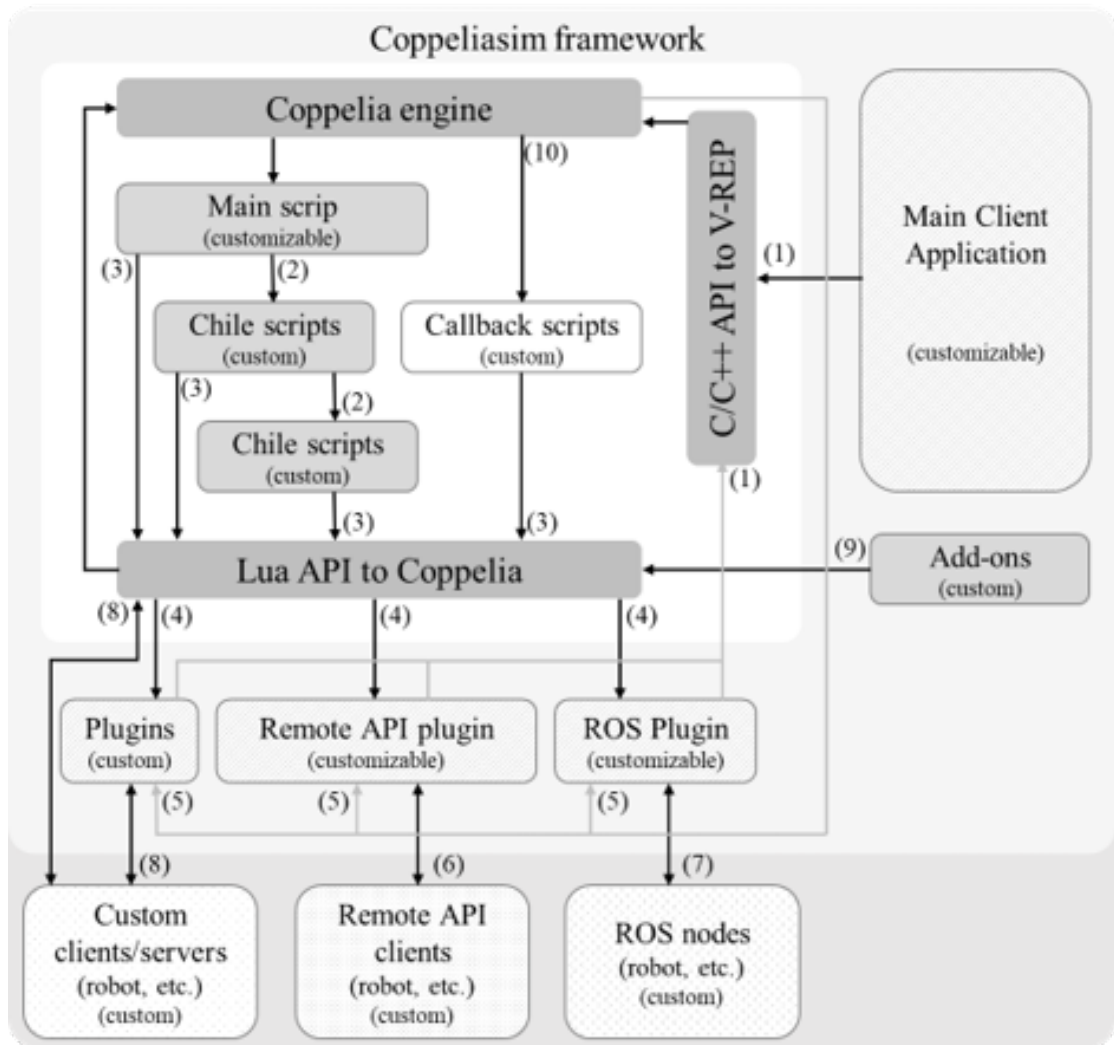


Figure 2.2: CoppeliaSim control architecture. Greyed items are control entities. (1) C/C++ API calls, (2) cascaded child script execution, (3) Lua API calls, (4) custom Lua API callbacks, (5) CoppeliaSim event callbacks, (6) remote API function calls, (7) ROS transit, (8) custom communication (socket, serial, pipes, etc.)

2.4.3 Simulation Control Architecture

- **Embedded scripts;** This represents the most powerful and distinctive feature of CoppeliaSim. The main simulation loop is a simple LUA script (called the main script), part of a given simulation scene, that handles general functionality (e.g., it will call distinct functions to handle kinematics or dynamics, for instance). The main script is also in charge of calling child scripts cascaded (for the scene hierarchy). Unlike the main script, a child script is attached to a specific object in the simulation scene and handles a particular part of the simulation. It is an integral part of its scene object and will be duplicated and serialized. As such, it represents a perfectly portable and scalable control element: there is one single file containing the model definition together with its control or functionality, there is no compatibility issue across platforms, no need for explicit compilation, no conflict among several versions of the same model, model instantiation is implicit. Child scripts can be executed in a threaded and non-threaded fashion. The threaded version of child scripts still keeps the advantages of the technique described in point 3 of section II, A: indeed, CoppeliaSim's thread scheduler handles threads in a way that makes them behave and appear as co-routines, which allow to precisely control the time at which the thread execution is switched back and forth, effectively allowing for an excellent synchronization with the main script or other child scripts. Additionally, each thread can programmatically request being set into a free-running mode (i.e., allowing them to behave as actual threads temporarily). Embedded scripts can also be seen as a glue component that binds the various supported programming techniques around CoppeliaSim: child scripts can register ROS publishers/subscribers, they can open and handle communication lines (e.g., socket or serial port), launch executable, load/unload plug-ins or start remote API server services (see point 4 hereafter). Embedded scripts also include callback scripts, used as low-level customized joint controllers. The user can extend the functionality of embedded scripts via two mechanisms: with Lua extension libraries or with custom Lua functions registered through a plug-in.
- **Plugins;** Plugins are used in CoppeliaSim as a convenient simulator customization tool. They can register custom LUA commands, allowing the execution of fast callback functions from within an embedded script. They can also extend the

functionality of a particular simulation model or object. They often implement specific importers/exporters or offer an interface to specific hardware. The remote API and ROS interfaces (see following items) are implemented via plugins.

- Remote API clients; The remote API interface in CoppeliaSim allows interacting with CoppeliaSim or a simulation from an external entity via a socket communication. It is composed of remote API server services and remote API clients. The client-side can be embedded as a small footprint code (C/C++, Python, Java, Matlab, and Urbi) in virtually any hardware, including real robots, and allows remote function call, as well as fast data streaming back and forth. On the client-side, functions are called almost regular functions, with two exceptions. However, remote API functions accept an additional argument, the operation mode, and return the same error code. The operation mode allows calling as blocking (will wait until the server replies) or non-blocking (will read streamed commands from a buffer or start/stop a streaming service on the server-side). The ease of use of the remote API, its availability on all platforms, and its small footprint make it an exciting alternative to the ROS interface (see next item).
- ROS nodes; CoppeliaSim implements a ROS node with a plug-in that allows ROS to call CoppeliaSim commands via ROS services or stream data via ROS publishers/subscribers. Publishers/subscribers can be enabled with a service call and directly enabled within CoppeliaSim via an embedded script command.

2.4.4 Real-Time Simulation Loop

CoppeliaSim is based on a distributed control architecture. Each object or model is individually controllable. CoppeliaSim allows the user to choose among various programming techniques simultaneously. The CoppeliaSim can be started with a real-time simulation loop. The system tries to keep the simulation time to match the real-time. A simulation in the CoppeliaSim can be started, paused, and stopped at any time. The simulator will use additional intermediate states to correctly inform scripts or programs about what will happen next. The CoppeliaSim allows users to modify the simulation scene during the simulation process. The simulator operates by advancing the simulation time at constant time steps. The following figure illustrates the main simulation loop. Real-time simulation is supported by trying to keep the simulation time synchronized

with the real-time see Fig. 2.3

2.4.5 Physical Engine

CoppeliaSim comes with four physical engines (Bullet, ODE, Vortex, and Newton physical engines), allowing users free selection before the simulation study.

- **ODE:** ODE is an open-source physics engine. It is probably the engine most commonly used in robotics applications. It is integrated with Gazebo and CoppeliaSim, and other robotics frameworks. ODE implements a sophisticated integrator for angular DOFs, a feature that contributes to its performance. ODE is a popular choice for robotics simulation applications, with scenarios such as mobile robot locomotion and simple grasping. ODE has some drawbacks in this field, such as approximating friction and poor support for joint-damping[35].
- **Bullet :** Bullet is a free and open-source physics engine that simulates collision detection and soft and rigid body dynamics. It has been used in video games and for visual effects in movies. Bullet can simulate in rigid body and soft body simulation with discrete and continuous collision detection, collision shapes simulation with many types of shapes such as a sphere, box, cylinder, etc.
- **Vortex :** Vortex is a complete simulation software platform. It features a high-fidelity, real-time physics engine developed by CM Labs Simulations that simulates rigid body dynamics, collision detection, contact determination, and dynamic reactions. Vortex adds accurate physical motion and interactions to objects in visual-simulation applications for operator training, mission planning, product concept validation, heavy machinery, robotics design and testing, haptics devices, and immersive and VR environments.
- **Newton Game Dynamics** is an open-source physics engine for realistically simulating rigid bodies in games and other real-time applications.

2.4.6 Friction Model in CoppeliaSim

CoppeliaSim comes with the ODE and Bullet physical engines. The physics engines track all physically active objects in the simulator, taking into account gravity, collisions,

and other constraints to determine how each object's position and orientation changes from one frame to the next. CoppeliaSim can predefine the friction coefficients of each model. In particular, the coefficient of friction is dependent on the two materials in contact. In a physics engine, define a coefficient of friction for each material. So, if materials A and B are in contact, the physics engines typically compute the coefficient of friction between materials A and B as

$$\mu_{ct} = \mu_A \times \mu_B \quad (2.4.1)$$

where; μ_A represents the friction coefficient of material A and μ_B represents the friction coefficient of material B.

Bullet uses contact dynamics, a hard contact model developed by Moreau [36],[37] and Jean[38]. Here the contacting particles are not allowed to overlap. Velocities before a collision compute the velocities after a collision. Then the Newton-Euler laws of motion are used to update the positions and orientations of the two particles and compute the contact force. Generally, the friction models of ODE are approximations to the friction cone for efficiency reasons. There are currently two approximations to choose from:

- 1) The meaning of H is changed to specify the maximum friction (tangential) force that can be present at contact in either of the tangential friction directions. This is rather nonphysical because it is independent of the normal force, but it can be useful, and it is the computationally cheapest option. In this case, h is a force limit and must be chosen appropriately for the simulation.
- 2) The friction cone is approximated by a friction pyramid aligned with the first and second friction directions. A further approximation is made: first, ODE computes the normal forces assuming that all the contacts are friction-less. Then calculates the maximum limits f_m for the friction (tangential) forces follow:

$$|f_m| = \mu \times |f_N| \quad (2.4.2)$$

Thereafter, proceeds to solve for the entire system with these fixed limits. This differs from a true friction pyramid in that the effective μ is not quite fixed. This approximation is easier to use as μ is a unit-less ratio the same as the normal Coloumb friction coefficient and thus can be set to a constant value around 1.0 without regard for the

specific simulation. However, CoppeliaSim interface can set the ODE properties with only constancy of μ , therefore in CoppeliaSim simulates with constancy of μ value only.

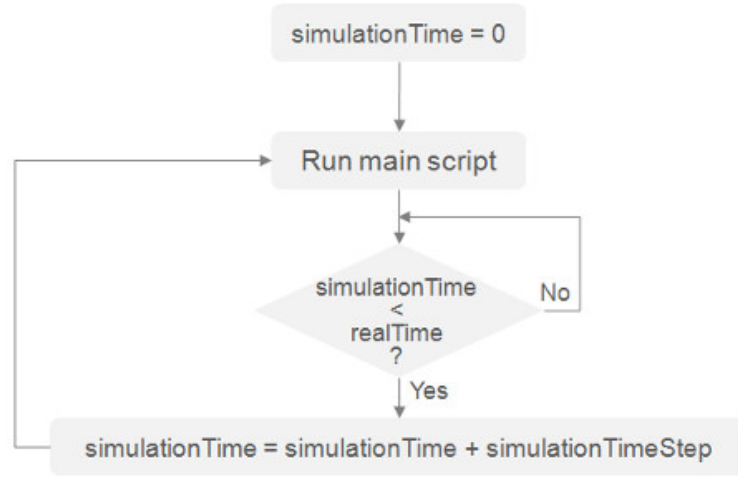


Figure 2.3: Real-time simulation loop in CoppeliaSim.

2.5 Path and Motion Planing

The open motion planning library (OMPL) is a new library for sampling-based motion planning, which contains implementations of many state-of-the-art planning algorithms. The library is designed to allow the user to solve various complex motion planning problems with minimal input easily. OMPL facilitates the addition of new motion planning algorithms, and it can be conveniently interfaced with other software components. A simple graphical user interface built on top of the library, several tutorials, demos, and programming assignments are designed to teach students about sampling-based motion planning.

OMPL is intended for use in research and education, and industry. Therefore, OMPL was designed to consist of a set of components as indicated in Fig 2.4 such that each component corresponds to general concepts in sampling-based motion planning. OMPL has been implemented entirely in C++ and is thread-safe. OMPL also offers abstract interfaces that the “host” software package can implement. Furthermore, the dependencies of OMPL are minimal: only the Boost C++ libraries are required. However, OMPL can be compiled with Python bindings, facilitating integration with Python modules. Furthermore, OMPL strives for minimalist API constraints for planning algorithms so

that new contributions can be easily integrated. As opposed to all other existing motion planning software libraries, OMPL does not include a representation of workspaces or robots; as a result, it also does not include a collision checker or any means of visualization. OMPL is reduced to only motion planning algorithms. The advantage of this minimalist approach is that it allows us to design a library for generic search in high-dimensional continuous spaces subject to complex constraints instead of defining valid states as collision-free. It would require a specific geometric representation of the environment and robot and support for a specific collision checker. OMPL leaves the definition of state validity completely up to the user.

The overview of the implementation of the core motion planning concepts in OMPL seen in Fig. 2.4 shows a high-level overview of the main classes and their relationships [39]

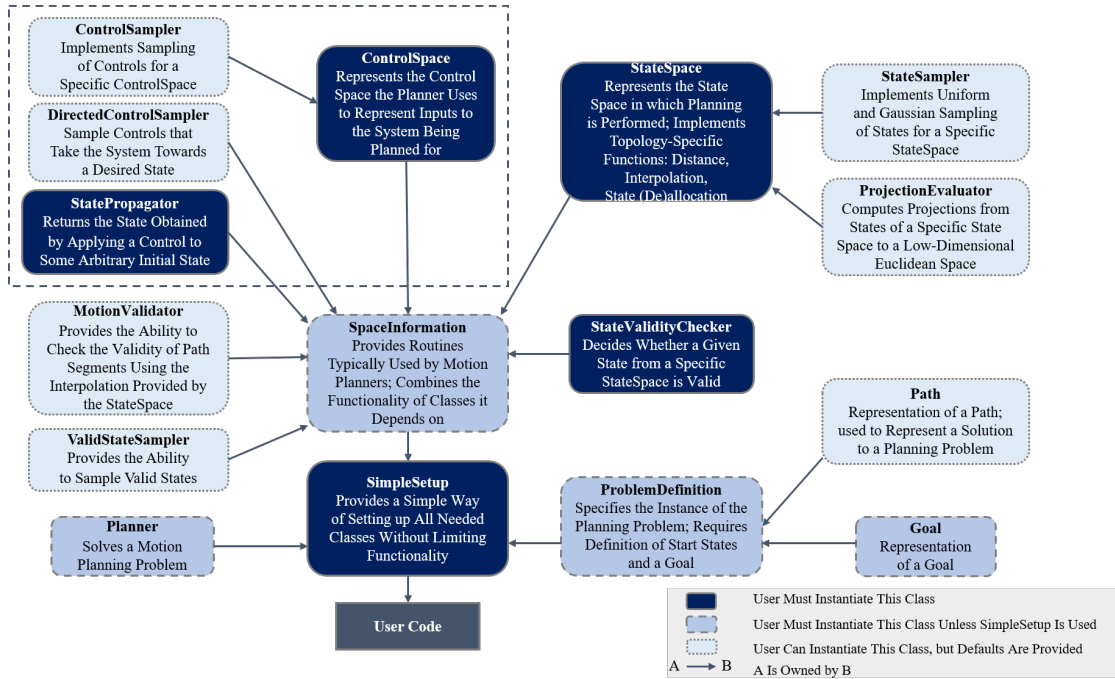


Figure 2.4: OMPL Overview core motion.

- **State Space:** represent the maximize the range of application for the included planning algorithms State spaces include operations on states such as distance evaluation, test for equality, interpolation, and memory management for states: (de)allocation and copying. Additionally, each state space has its storage format for states, which is not exposed outside the implementation of the state space itself. The planning algorithms implemented in OMPL rely only on the generic

functionality offered by state spaces to operate on states. This approach enables planning algorithms in OMPL to apply to any state spaces that may be defined, as long as the expected generic functionality is provided. OMPL includes a means of combining state spaces using the class `CompoundStateSpace`, implements the functionality of a regular state-space on top of the corresponding functionality from the maintained set of state spaces. This allows trivial construction of more complex state spaces.

State spaces optionally include specifications of projections to Euclidean spaces (`ProjectionEvaluator`). Several sampling-based planning algorithms use Low dimensional Euclidean projections (e.g., KPIECE [40], SBL [41], EST [42]) to guide their search for a feasible path, as it is much easier to keep track of coverage (i.e., which areas have been sufficiently explored and which areas should be explored further) in such low-dimensional spaces.

In addition to states and state spaces, some algorithms in OMPL require a means to represent controls. Control spaces (`ControlSpace`) mirror the structure of state spaces and provide functionality specific to controls so that planning algorithms can be implemented generically. The only available implementations of control spaces are the Euclidean space and a space for discrete modes because so far, there has not been a need for control spaces with more complex typologies. However, the API allows one to define such control spaces.

- **State Validation and Propagation:** Whether a state is valid or not depends on the planning context. In many cases, state validity means that a robot is not in collision with any obstacles, but in general, any condition on a state can be used. A user needs to specify how the system evolves when specific controls are applied for a while, starting from a given state for planning with controls. This is called state propagation in OMPL.
- **Samples:** The fundamental operation that sampling-based planners perform is sampling the explored space. Additionally, when considering controls in the planning process, sampling controls may be performed as well. To support sampling functionality, OMPL includes four types of samplers: state-space samplers (`StateSampler`), valid state samplers (`ValidStateSampler`), control samplers (`ControlSampler`), and directed control samplers (`DirectedControlSampler`). State-space

samplers are implemented as part of the StateSpace they can sample since they need to be aware of the structure of the states in that space. For instance, uniformly sampling 3-D orientations is dependent on their parameterization. Three sampling distributions are implemented by every state-space sampler: uniform, Gaussian, and uniform in the vicinity of a specified point. This first sampler is necessary to sample the entire space, but the latter two are used for sampling states near a previously generated state. This is the most basic level of sampling.

- Goal Representations: OMPL uses a hierarchical representation of goals. In the most general case, a goal can be defined by the `isSatisfied()` function that reports whether that state is a goal state or not when given a state. While this very general implicit representation is possible, it offers planners an indication of how to reach the goal region. For this reason, `isSatisfied()` optionally reports a heuristic distance from the goal region, which is not required to be a metric. `GoalRegion` is a refinement of the general goal representation, which explicitly specifies the distance to the goal using a `distanceGoal()` function. The `isSatisfied()` function is then defined to return true when `distanceGoal()` reports more petite than a user set threshold.

`GoalRegion` is still a general representation but allows planners to bias their search toward the goal. A refinement of `GoalRegion` is `GoalSampleableRegion`, which also allows drawing samples from the goal region. `GoalState` and `GoalStates` are concrete implementations of `GoalSampleableRegion`.

It is often possible to sample the goal region for practical applications, but the sampling process may be relatively slow (e.g., numerical inverse kinematics solvers). For this reason, a refinement of `GoalStates` is defined as well: `GoalLazySamples`. This refinement continuously draws samples in a separate sampling thread. It allows planners to draw samples from the goal region without waiting after at least one sample has been produced by the sampling thread.

- Planning algorithm: OMPL includes two types of motion planners that do not consider controls when planning and ones that do. If a planning algorithm can be used to plan both types of motions, with and without controls (e.g., RRT [43]), two separate implementations are provided for that algorithm: one for each type of computed motion. This choice was made for efficiency reasons. With

additional levels of abstraction in the implementation, it would have been possible to avoid separate implementations, [44]. The downside would have been that the implementation of planners would have had to follow a strict structure, which implements new algorithms more complex and possibly less efficient.

The solution path is constructed from a finite set of segments for purely geometric planning (i.e., controls are not considered). Each segment is computed by interpolation between a pair of sampled states (PathGeometric). Several geometric planning algorithms are implemented in OMPL, including KPIECE [16], bidirectional KPIECE, bidirectional lazy KPIECE, RRT [43], RRT-connect [45], lazy RRT, SBL [41], EST [42], and PRM [46]. The lazy variants listed above defer state validity checking within the robotics community. According to certain metrics, it is often challenging to demonstrate that a new motion planning algorithm is an improvement over the existing methods. manner described in [47]. In addition, there are multithreaded versions of RRT and SBL.

When controls are considered, the solution path is constructed from a sequence of controls (PathControl). Control-based planners are typically used when motion plans also need to respect differential constraints. Several algorithms for planning with differential constraints are implemented in OMPL as well, including KPIECE, SyCLoP [44], EST, and RRT.

2.5.1 RRT Path Planning

Path planning can generally be viewed as a search in configuration space(C). Each $q \in C$ specifies the position and orientation of one or more geometrically-complicated bodies in a 2D or 3D world. A metric p is assumed to be defined on C . Let C_{free} denote the set of configurations for which these bodies do not collide with any static obstacles. The obstacles are modeled entirely in the world, and an explicit representation of C_{free} is not available. However, using a collision detection algorithm, a given $q \in C$ can be tested to determine whether $q \in C_{\text{free}}$. The single query path planning task is to compute a continuous path from an initial configuration, q_{init} , to a goal configuration, q_{goal} , without performing any preprocessing. The Rapidly-exploring Random Tree (RRT) was introduced in [48] as an efficient data structure and sampling scheme to quickly search high-dimensional spaces that have both algebraic constraints (arising from obstacles)

and differential constraints (arising from nonholonomic and dynamics). The key idea is to bias the exploration toward unexplored portions of the space.

The classic RRT algorithm has the start state $x_{\text{init}} = x_{\text{start}}$. As listed in Fig.2.6 the algorithm repeatedly samples a new state $x_{\text{rand}} \in 2X$ and then tries to extend the tree from the nearest state x_{nearest} already present in V towards x_{rand} using a steer function, resulting in x_{new} (see Fig. 2.5). This extension is examined considering validity regarding externally given constraints by what is usually called a local planner. If this validity is approved, x_{new} is added to V and an edge between x_{nearest} and x_{new} is created.

Algorithm 1: Extend

```

Function EXTEND( $G = (V, E), x, x_{\text{new}}$ )
    // Extend  $G$  towards  $x$ , creating  $x_{\text{new}}$ 
     $x_{\text{nearest}} \leftarrow \text{NEAREST}(G = (V, E), x)$ ;
     $x_{\text{new}} \leftarrow \text{STEER}(x_{\text{nearest}}, x)$ ;
    if OBSTACLEFREE( $x_{\text{nearest}}, x_{\text{new}}$ ) then
         $V \leftarrow V \cup \{x_{\text{new}}\}$ ;
         $E \leftarrow E \cup \{(x_{\text{nearest}}, x_{\text{new}})\}$ ;
        if  $x_{\text{new}} = x$  then
            return REACHED;
        else
            return ADVANCED;
    return TRAPPED;

```

Figure 2.5: Algorithm : Extend.

Algorithm 2: RRT

```

 $V \leftarrow \{x_{\text{init}}\}$ ;  $E \leftarrow \emptyset$ ;
for  $i = 1, \dots, n$  do
     $x_{\text{rand}} \leftarrow \text{SAMPLEFREE}_i$ ;
    EXTEND( $G = (V, E), x_{\text{rand}}, x_{\text{new}}$ );
return  $G = (V, E)$ ;

```

Figure 2.6: Algorithm : RRT.

The RRT algorithm has been extended towards RRT-Connect using a bidirectional search. RRT-Connect (see Fig 2.8) uses two separate trees T_a and T_b initialized respectively at the start and goal states, x_{start} and x_{goal} of the planning problem. A tree is extended just as in RRT, with the distinction that the algorithm alternates between the two trees. If an extended step is successful, the algorithm tries to connect the newly added state to the nearest neighbor of the other tree. For this connection attempt different strategies exist, see [45]. This behavior results in the trees T_a and T_b exploring their local surroundings and urging to connect with their counterpart, which leads to

paths that are usually found more quickly than with a basic RRT, especially in scenarios where barriers or narrow passages conceal the start or goal states (or both). This can be explained by the explorative strategy making it easier to escape a small gap than to enter it. Additionally, the increased set of states each tree is trying to connect to provides a greater probability of finding a valid link to the other tree.

Algorithm 3: Connect

```

Function CONNECT( $G_{\text{other}} = (V_{\text{other}}, E_{\text{other}}), x$ )
    // Repeatedly extend  $G_{\text{other}}$  towards  $x$ 
    repeat
        |  $S \leftarrow \text{EXTEND}(G_{\text{other}} = (V_{\text{other}}, E_{\text{other}}), x, x_{\text{new}})$ ;
    until  $S \neq \text{ADVANCED}$ ;
    return  $S$ ;

```

Figure 2.7: Algorithm : Connect.

Algorithm 4: RRT-Connect

```

 $V_a \leftarrow \{x_{\text{init}}\}; E_a \leftarrow \emptyset;$ 
 $V_b \leftarrow \{x_{\text{goal}}\}; E_b \leftarrow \emptyset;$ 
for  $i = 1, \dots, n$  do
     $x_{\text{rand}} \leftarrow \text{SAMPLEFREE}_i$ ;
    if  $\text{EXTEND}(G_a = (V_a, E_a), x_{\text{rand}}, x_{\text{new}}) \neq \text{TRAPPED}$  then
        if  $\text{CONNECT}(G_b = (V_b, E_b), x_{\text{new}}) = \text{REACHED}$  then
            | return  $G_a = (V_a, E_a), G_b = (V_b, E_b)$ ;
        |  $\text{SWAP}(G_a = (V_a, E_a), G_b = (V_b, E_b))$ ;
return  $G_a = (V_a, E_a), G_b = (V_b, E_b)$ ;

```

Figure 2.8: Algorithm : RRT-Connect.

2.6 Robot Navigation

Robot navigation is the ability of an autonomous robot to plan its movements in real-time and safely navigate from one place to another. An effective navigation process requires path planning, manual positioning, and motion control. Path planning is the process of finding an optimal path from a start point to the target location without any collisions. Localization means that the robot estimates its position relative to specific objects within the environment. Motion control is the robot's ability to transfer sensory information into accurate physical movement in a realistic world.

The process of robot navigation is a complex technological problem as it determines a robot's autonomy and reliability in performing assigned tasks; it has been widely

researched since the 1970s. While there are many proposed solutions and techniques, the navigation problem remains challenging. It is not because of limited navigation algorithms but because of the requirement for robust and reliable methods to acquire and extract environmental information, which is automatically related to the navigation map. Negenborn described three additional problems in robust robot navigation: limits in computational power (CPUs), difficulties in detecting and recognizing objects, and complexities in obstacle avoidance.

Navigation strategies depend on whether the environment is static (static obstacles) or dynamic (static and dynamic obstacles). Both categories are divisible into unknown and known environments. In the latter, information is provided on the location of obstacles before motion commences. Across the various environments, many navigation algorithms address the robot navigation problem. All navigation planning algorithms assume that the mobile robot knows the start and target locations. Therefore, the direction between them is to find an optimal path between these two locations and avoid obstacles. Some algorithms require extra environmental information or even a comprehensive map. Navigation algorithms are classified into global and local planning.

2.6.1 Local Motion Control

The local motion control used in the robot is in a completely unknown area and does not have information about the surrounding area. The objective of the obstacle avoidance algorithm is to move a robot toward an area free of collisions to the information handled by the sensors during motion execution. The improvement of the obstacle avoidance is to find a direction for the robot by introducing the sensor information, which is steadily updated used to control the motion in real-time. The main issue of considering the world's reality during execution is locality, which means the robot's localization during the execution of the algorithm in the environment (that, as we will see, is one of the problems encountered when the robot travels along a path). An error in the robot's location generates a series of problems that depends on the first one, as a wrong map, and trajectory that does not coincide with the real one. Notwithstanding that limitation, obstacle avoidance techniques are mandatory to deal with robotics problems in the unknown and changing environment.

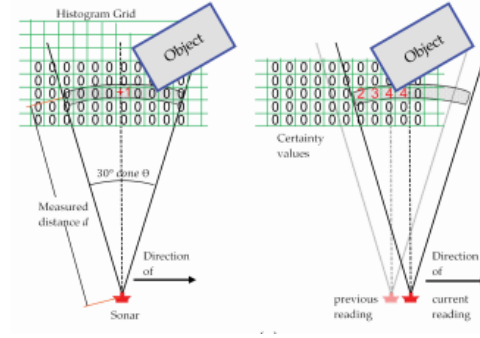
Local motion control directly uses the sensor's information in the commands that control

the robot's motion in every control cycle without constructing a global map. Therefore, these algorithms guide the robot in one straight path from the start to the target location in unknown or dynamic environments. While the robot navigates, it avoids obstacles in its path and updates important information, such as the distance between its current location and the target position. The local navigation algorithms are easy to construct and optimal for real-time applications.

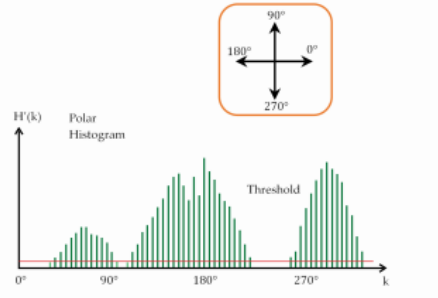
A potential field algorithm, widely used within the local navigating technique, is constructed by creating the artificial potential field around the robot. The target position's potential attracts the robot while the obstacles' potential repulses it. As the robot moves toward the target, it calculates the potential field and then determines the induced force by this field to control the robot's motion. Typically, the robot moves from a higher to a lower potential field. The optimal potential field is constructed so that the robot is not trapped into a minimum local field before reaching the target, but it is impossible to create such a field. Therefore, this method is combined with other navigation algorithms to increase efficiency.

Vector field Histogram is a three-stage method of obstacle avoidance. In the first stage, a 2D histogram is generated around the robot that represents the obstacles Fig. 2.9a. A 2D histogram is updated with new coming percepts from sensors. This 2D histogram is converted to a 1D histogram and then polar histogram in the second step Figs. 2.9b. Finally, in a third step, the algorithm selects the most suitable sector with low polar obstacle density and calculates the steering angle and velocity in that direction. The VHF algorithm improved to VFH+ [49] and VFH* [50] VFH+ reduced the original VFH parameter tuning, and the VFH* method verified that a particular candidate direction guides the robot around an obstacle. Lidar is suitable for approving VFH methods since it can take high-resolution, multiple-ranging data in two dimensions. For example, Reference [51] used VFH+ with lidar to avoid obstacles using a hexacopter, and [52] used original VHF with lidar for obstacle avoidance of automated guided vehicles.

The Bug algorithms, Lumelsky and Stepanov[53] proposed this method following the bug's movement. They made two versions of the bug algorithm: Bug1 and Bug2, which are shown in Figs. 2.10. The robot operates from "s" to target "t." For Bug1, the robot will move fully around to check the object and calculate the shortest position toward the target point. For the case of Bug2, the robot will create a line from start to target,



(a) 2D histogram grid map.



(b) converted 1D polar histogram.

Figure 2.9: Vector field histogram algorithm.

and if it finds an obstacle, it will go alongside the obstacle, and when it finds the line, it will keep moving. The Bug1 algorithm travels a long path to reach the goal point. While Bug2 uses a shorter route. However, to use even a shorter way to reach the goal, some improvements have been made on the bug algorithm, such as tangent bug [54], I-Bug [55], improved bug [56], splitting Bug [57], etc. [58]. These bug algorithms are not so reliable in a more complex environment, and in some tricky conditions, one version works better than the other version. However, the generality of the bug algorithms is that it works well with single obstacle avoidance [59].

The new hybrid navigation algorithm is based on two layers (deliberative and re-active layers). Both layers are independent of each other. The deliberative layer planned a reference path based on prior stored information. The reactive layer is an independently steers robot on the path planned by the deliberative layer. A hybrid algorithm requires prior environment information stored in a binary grid map. On the map, states of every grid are either free or occupied depending on obstacles, i.e., free for no obstacle and occupied for the obstacle. Unknown information is also taken as a free. The A* search algorithm generates a reference path in the deliberative layer. Reference path

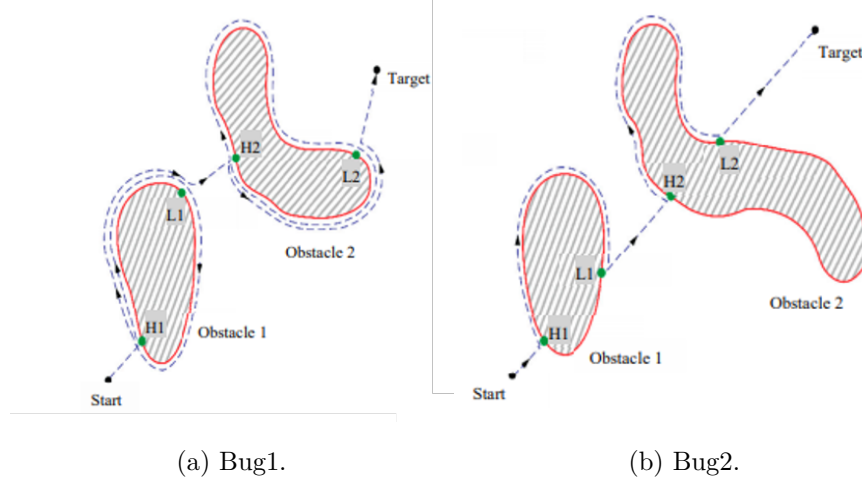


Figure 2.10: Bug algorithm.

is temporary and not necessary to follow throughout the motion; the reactive layer can change it. The reactive layer takes the reference path from the deliberative layer and controls the robot's motion. It also receives the precepts of sensors and decides to avoid an obstacle if found. This layer uses the D-H bug algorithm (distance histogram bug). This version of the bug-2 algorithm allowing the robot to rotate freely at an angle less than 90° to avoid an obstacle. If the rotation of 90° or greater is required to avoid an obstacle, it acts as a bug-2 algorithm and starts moving to the destination when the path is clear from obstacles. The reactive layer can change the path based on the current precept. Sensors provide current precepts to the reactive layer and update the prior knowledge. In case of conflict between layers, the result of the reactive layer is taken into account. Due to the present and updated nature of the results of the reactive layer, incomplete knowledge of the deliberative layer may contain errors.

A fuzzy logic algorithm was proposed by Zadeh[60], which uses the fuzzy controller. To use fuzzy logic in any system, the operator needs to assign a set of data or knowledge to create fuzzy sets that will be used to avoid obstacles or navigate the mobile robot. This process of assigning fuzzy input sets is called fuzzification. The set value usually can be anywhere between two traditional logics, such as (0, 1) or (Low, High) or (Cold, Hot). This is why those vehicles that use fuzzy logic for navigation and avoidance usually use one kind of multiple sensors or sensor fusion. [61] used the fuzzy controller to control an obstacle avoidance mobile robot. In Fig. 2.11, the usage procedure is given. This classic method is used in many vehicle navigation systems[62]. [63] used fuzzy logic

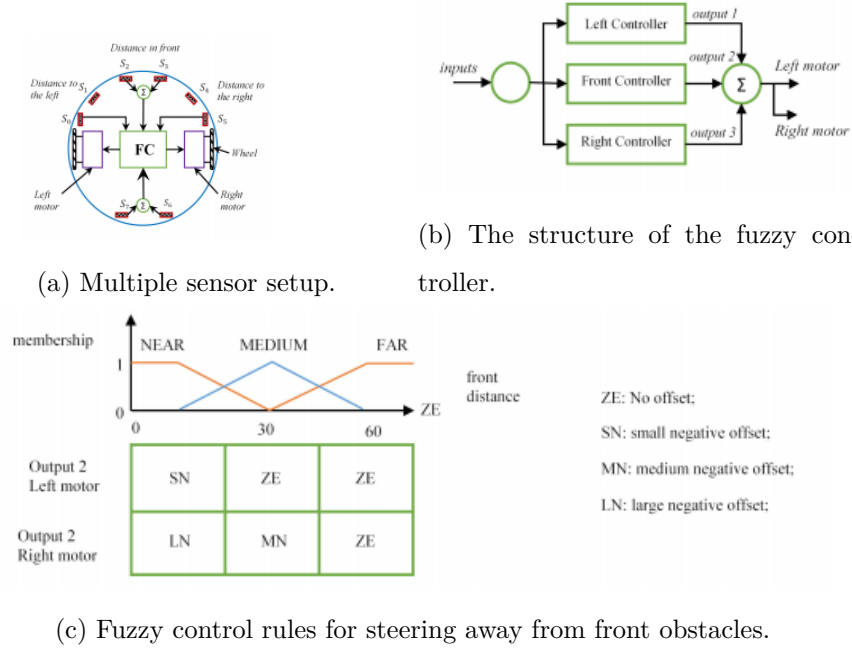


Figure 2.11: Usage of fuzzy logic control.

techniques to build a reactive navigation system and avoid obstacles. Several research works created a fuzzy logic controller using fuzzy sets to avoid obstacles in real-time. [64] used a fuzzy-based approach to track paths and avoid obstacles. [65] proposed a navigation algorithm using a fuzzy controller and sensor fusion (camera and sonar) with a mobile robot to avoid obstacles and generate trajectories. The fuzzy logic system and three-way ultrasonic sensor, [66] proposed an avoidance algorithm for a mobile robot. [67] designed a fuzzy logic controller to improve the vehicle's movement according to the obstacle's position.

2.6.2 Challenges, Technical Limitations, and Analytical Comparison for Local Motion Control

The proper obstacle avoidance technique on local motion control for teleoperation systems is based on limited or narrow environments. In traditional local motion control require the target before starting the system. However, the operator cannot determine the target's exact location in some situations. For example, survey or rescue robots searching for survivors. Additionally, it is difficult to determine the ending distance of a narrow path in unknown environments. There are limitations of some techniques which affect open environments or the limitation of the shape of the obstruction. In addition,

some techniques require high-efficiency and high-powered robots, which is not suitable for teleoperation systems. The summarized and compared the obstacle avoidance techniques in Table 2.7. The Braitenberg algorithm is an idea that makes robot characters depend on sensor inputs. The relationship between the sensor position and the number will affect the different speeds of each robot wheel. These algorithms are computationally fast can be used in any environment and any shape of obstacles (depending on the number of sensors). However, the Braitenberg algorithm requires an additional setup process between the robot and the number and sensor position. The setting time will be longer. System performance also depends on the number and location of sensors. Therefore, in this experiment, the adjusted simulation helped reduce the time and damage of robot installation. Moreover, reduce the limitation of the location and number of sensors. Use a sensor model that has the freedom to position and increase the number of sensors.

2.7 Sensor

Sensors play crucial roles in a robot. Many ways and sensors can extract meaningful information from various environments, recognize the surrounding objects using this information, and transmit it to the robot. Every information that the robot can capture is used as data to act, make a plan, or as input to perform some operation. However, in the teleoperation system, many sensors result in high energy demand and shorten the working time of the robot. Some studies have targeted systems by fusing sensors to optimize remote performance. Various data from different sensors are displayed on the screen separately to provide more accurate environmental information. Nevertheless, the information is presented differently rather than in an integrated form. This requires the operator to mentally correlate the sets of information, resulting in increased workload, decreased situation awareness, and decreased performance. The disparate information requires the operator to mentally combine the data into a holistic environment representation. This requires the operator to mentally correlate the sets of information, resulting in increased workload, decreased situation awareness, and decreased performance.

There are various sensors for getting such information, and the most used for its effectiveness and simplicity is the distance sensor. Laser-based distance sensors such as Laser Distance Sensor, LiDAR, or Laser Range Finders (LRF) and infrared-based

Table 2.7: Algorithm comparison

Avoidance Technique	Features
Bug2 Algorithm	<ul style="list-style-type: none"> • Easy and Convenient • It follows the obstacle's exact outline • It changes the heading when bypassing the obstacle • It doesn't detect the edge of the obstacle, which may take a longer path sometimes
Artificial Potential Field	<ul style="list-style-type: none"> • A simple approach for implementation • Easy to find the shorter edge of the obstacle • Local minima problem can cause process failure
Collision Cone	<ul style="list-style-type: none"> • Creates simpler avoidance path • Uses vehicle dynamics to create avoidance path • The minimum effort of guidance control • Doesn't consider the shape of the obstacle
Fuzzy Logic	<ul style="list-style-type: none"> • Robust and suitable for dynamic environments • Needs multisensory system • Polyhedral shape may increase computational calculation
Vector Field Histogram	<ul style="list-style-type: none"> • A better method for detect obstacle's shape identification • Require longer time to 2D map the obstacle • High computational requirement • Doesn't consider the vehicle's dynamics
Neural Network	<ul style="list-style-type: none"> • Good for known obstacle environment • Better performance for real-time avoidance • Needs lots of training data before performance

sensors such as RealSense, Kinect, and Xtion are widely used as distance sensors. In addition, there are various sensors depending on the information to acquire, such as color cameras used for object recognition, inertial sensors used for position estimation, microphones used for voice recognition, and torque sensors used for torque control. The study focused on camera sensors and laser rangefinders to obtain environmental data, selected IMU sensors, and odometry encoding to assess the condition and position of the robot.

2.8 Camera

The camera can be represented as the eyes of the robot, and the images taken from the camera are useful for recognizing the environment around the robot. For example, object recognition, using a camera image, facial recognition, a distance value obtained from the difference between two different images using two cameras (stereo camera), mono camera visual SLAM, color recognition using information obtained from an image, and object tracking, are very useful.

2.9 Laser Range Finder

A laser rangefinder, also known as a laser telemeter, is a rangefinder that uses a laser beam to determine the distance to an object. The most common form of laser rangefinder operates on the time of flight principle by sending a laser pulse in a narrow beam toward the object and measuring the time taken by the pulse to be reflected off the target and returned to the sender. Due to the high speed of light, this technique is not appropriate for high-precision submillimeter measurements, where triangulation and other techniques are often used.

The Hokuyo sensors are excellent options for SLAM, sense-and-avoid, and other vision applications for mobile robots. The Hokuyo URG-04LX is an LRF categorized as an Amplitude Modulated Continuous Wave sensor. As depicted in Fig. 2.12, the laser emits an infrared beam, and a rotating mirror changes the beam's direction. Then the laser hits the surface of an object and is reflected. The direction of reflected light is changed again by a rotating mirror and captured by the photodiode. The phases of the emitted received light are compared. The distance between the sensor and the object is calculated. A rotating mirror sweeps the laser beam horizontally over a range of 240° , with an angular resolution of 0.36° . As the mirror rotates at about 600 rpm, the scan rate is about 100 msec. The LRF has a quoted range of between 20 and 4,095 mm. The quoted measurement error is ± 10 mm for distances of less than 1 m. For greater distances, the error is quoted as ± 2 percent, assuming a target of a patch of white paper of size 70x70 mm.

Laser rangefinding works by detecting phase change in light reflected off a target. The sensor sends out light at a known wavelength and then reads the reflected light after it

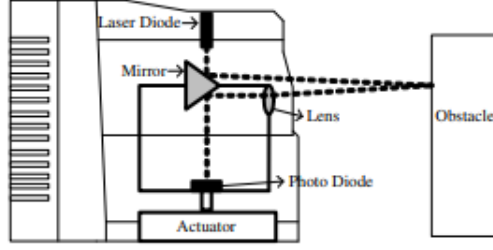


Figure 2.12: The structure of Hokuyo UBG-04LX-F01

has traveled out to the environment and back to the sensor. Any shifts in the peaks and troughs can be used to calculate phase change.

The distance D between sensor and target can be calculated from 2.9.1,

$$D = \frac{c\varphi}{2\omega} \quad (2.9.1)$$

Where c is the speed of light and, ω is the angular frequency of the light wave, and φ is the phase change between the initial and reflected light rays. We can expand our equation for D further, substituting the frequency of light $f = \omega/(4\pi)$ in Eq. 2.9.2, then substituting wavelength $\lambda = c/f$ in Eq. 2.9.3. Light is a wave, so values repeat every period. Accounting for sign changes, phase changes can be described as $(N + \Delta N)$ in Eq. 2.9.4, which N is an integer number of wave half-cycles and ΔN is the remaining fractional part of the phase change.

$$D = \frac{c\varphi}{2\omega} \quad (2.9.2)$$

$$D = \frac{c}{4\pi f}(N\pi + \Delta\varphi) \quad (2.9.3)$$

$$D = \frac{\lambda}{4}(N + \Delta N) \quad (2.9.4)$$

2.10 Chapter Summary

This chapter discusses the working principle of the ROS framework. Describe the connection and transmission of data between the robot and simulation via the ROS system.

In addition, it provides an overview of the robotic simulator and a comparison of the performance characteristics of the Gazebo simulators, which is the official emulator of the robotic operating system, with the CoppeliaSim selected in this study. CoppeliaSim is more suitable than Gazebo simulation in terms of efficiency and flexibility, although it takes the steps and expertise to connect with a physical device.

Next, we describe the role of the physical engine in the robotic simulator. Dynamic physical engine is a critical parameter affecting the simulation accuracy. CoppeliaSim comes with four different physical engines. Each engine is designed for a different purpose; therefore, we need to validate the performance of the physical engine to find a suitable physical engine before the experiment.

We demonstrated the CoppeliaSim framework and another critical parameter, such as the friction model in CoppeliaSim, how friction model simulation generated the object motion in CoppeliaSim, and description of the OMPL module to applied path planning module in this study. We also described the essential device used in the experiments.

CHAPTER 3

Simulation Accuracy Experiment

Accuracy is an essential characteristic of any simulator, particularly in robotics, aiming to replicate a real mechanic system. Thus, we have validated the accuracy of the dynamics simulation. CoppeliaSim included four different physics engines: Bullet 2.78, Bullet 2.83, ODE, and Vortex. In this experiment, the accuracy of all dynamic engines in CoppeliaSim is assessed in terms of position and orientation.

3.1 Simulation Accuracy Experiment

We created the simulation scene as shown in Fig 3.1. A simulation scene consists of 3 robots; a black robot model, a red robot model, and a blue robot model. The black robot model was used to determine the final position of the robot. Users can move the black robot model to the desired position by using the move-object function in CoppeliaSim. The blue robot model has represented the real robot position, controlled by odometry data from the ROS system. Red robot-model control by CoppeliaSim dynamics engine. We create an embedded script to call a dynamic engine for controlling the movement using control law see in Fig. 3.2. The control law generate the speed of each robot wheels by input distance(d) and the angle (α) between the center of mass of the robot $C(x_c, y_c)$ and target point ($P(x_p, y_p)$), calculated as shown in Eq. 3.1.1 and 3.1.2. With these two inputs, the control law calculates the linear velocity (ν) and the angular velocity (ω) of the robots to reach the destination point as it is shown in Eq. 3.1.3 and 3.1.4. Where ν_{\max} is the maximum linear velocity, k_r is the radius of a docking area (around the target point) and ω_{\max} is the maximum angular velocity of the robot.

$$d = \sqrt{(x_p - x_c)^2 + (y_p - y_c)^2} \quad (3.1.1)$$

$$\theta = \tan^{-1}\left(\frac{y_g - y_c}{x_g - x_c}\right) \quad (3.1.2)$$

$$\nu = \begin{cases} \nu_{max} & \text{if } |d| > k_r \\ d\left(\frac{\nu_{max}}{k_r}\right) & \text{if } |d| \leq k_r \end{cases} \quad (3.1.3)$$

$$\omega = \omega_{max} \sin(\alpha - \theta) \quad (3.1.4)$$

$$\nu_r = \frac{2\nu + \omega R}{2} \quad (3.1.5)$$

$$\nu_l = \frac{2\nu + \omega L}{2} \quad (3.1.6)$$

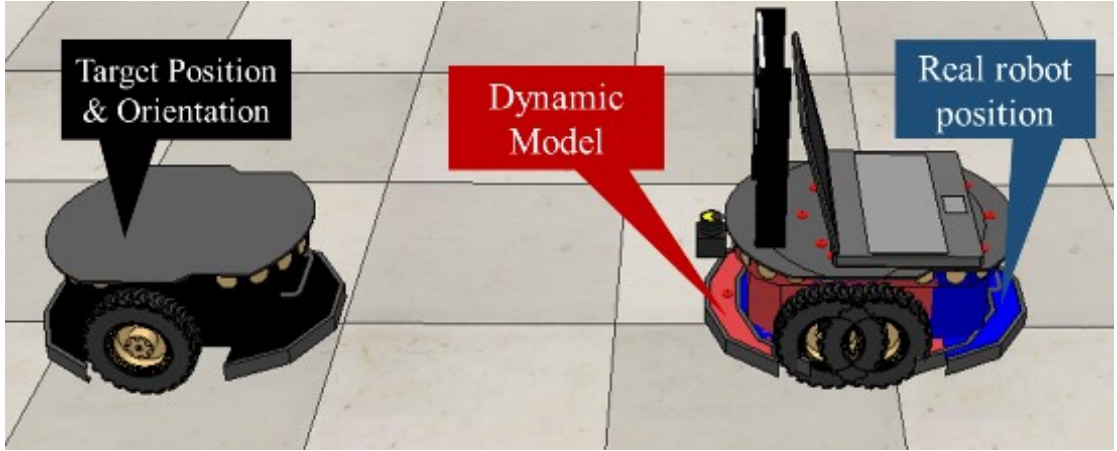


Figure 3.1: Simulation scene in accuracy experiment.

This study applied the `ekf_localization_node` for fusing the data from IMU with an odometry encoder to estimate the robot's position. ROS `ekf_localization_node` can obtain an overall position estimate whose error is less than possible by using a single sensor in isolation. The package currently contains an extended Kalman filter (EKF) implementation. It is often the case that a more significant amount of sensor input data will produce more accurate position estimates. Each state estimation node in `robot_localization` begins estimating the vehicle's state as soon as it receives a single

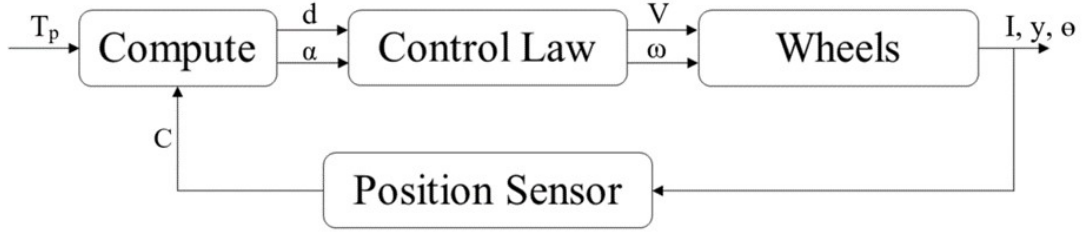


Figure 3.2: Block diagram of the position control loop.

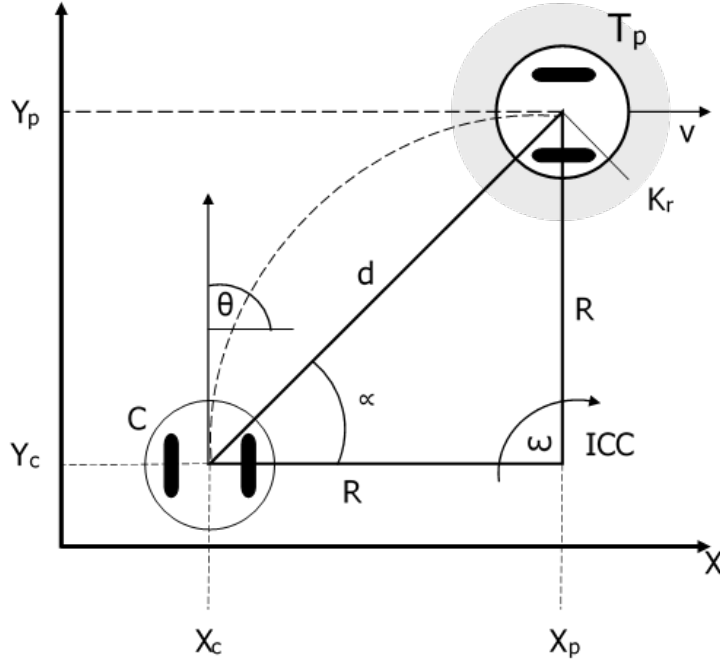


Figure 3.3: Position control experiment.

measurement. Suppose there is missing sensor data (i.e., a long period in which no data is received). In that case, the filter will continue to estimate the robot's state via an internal motion model. All state estimation `ekf_localization_node` track the 15-dimensional state of the vehicle: (X, Y, Z, roll, pitch, yaw, X, Y, Z, roll, pitch, yaw, X, Y, Z) from IMU sensor and Odometry encoder. Additionally, we set a target 1.5 m away from the starting position, reducing cumulative error due to sensor error.

The system for communication between CoppeliaSim and real robot see in Fig 3.4. The actual robot position and orientation obtained from fusion sensor (odometry via `/ROS_ARIA/odom` and IMU sensor from `IMU_complementary_node`), compared with data obtained CoppeliaSim simulation retrieved by embedded script. In the experiment,

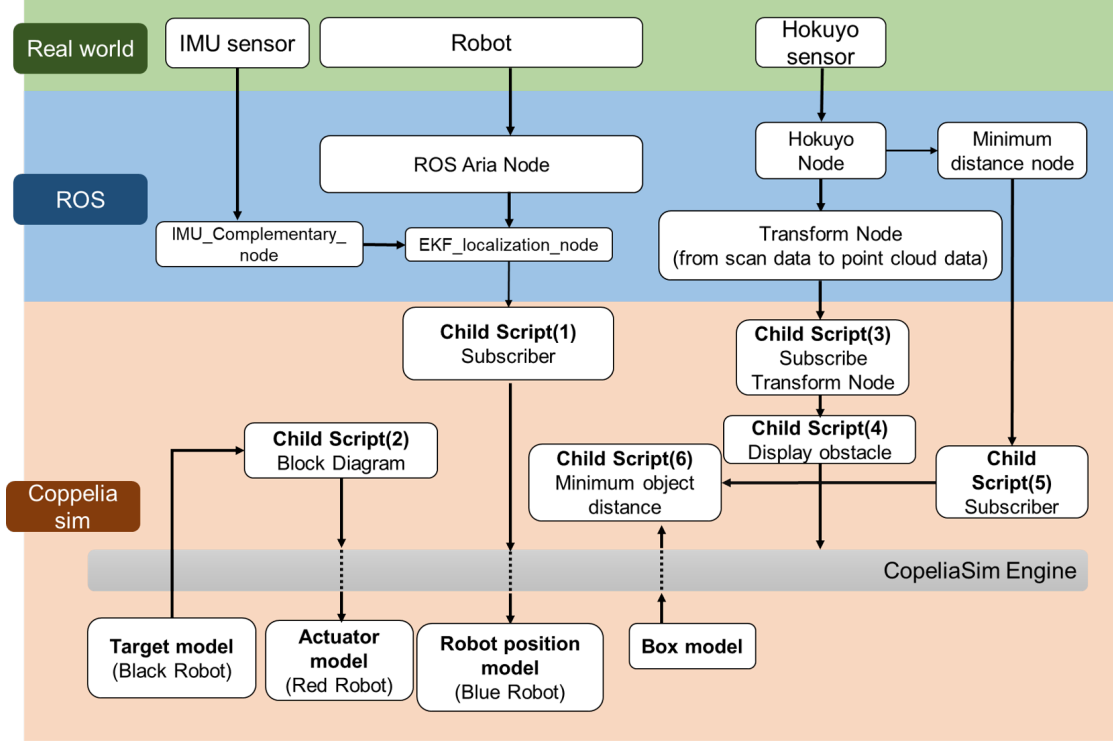


Figure 3.4: The communication system between CoppeliaSim and Real robot.

the ν_{\max} commands both actuator models (left's wheel and right's wheel) with 0.5, 1, and 2 degrees per second. In the orientation validation experiment, the LUA API commands the actuator model the same as the position validation experiment; however, the velocities of left and right robot wheels were inverses for making the rotation movement. The embedded scripts get real-time linear and angular velocities of the robot model to command the real robot movement via RosInterface, as shown in Fig. 3.4.

We compared the final position of the simulation (red robot model) with an actual robot (blue robot model), as shown in Fig 3.5. When designing target positions (black robot model) 5 meters in front of the robot, the robot moves straight towards the target. Table 3.1 shows the positional difference between the simulation and the actual robot at the goal position. Figure 3.6 shows the robot move with angular velocities. In this experiment, the robot movement with only angular velocities and comparing the final orientation of the robot as shown in table 3.2. and Fig 3.6

Table 3.1 shows the ODE dynamic engine is the most accurate in straight movement simulation; the average error of all robot speeds is 0.0623 ± 0.0227 m while the Bullet 2.83 resulted in the most position error, an average error is 0.1545 ± 0.0257 . In order of the

orientation, the results are the same as the position experiment. The results are shown in Table 3.2. The ODE dynamic engine is the most accurate with the lowest orientation error, while Bullet 2.83 has the most orientation error. In addition, we found the error will increase when speed increases.

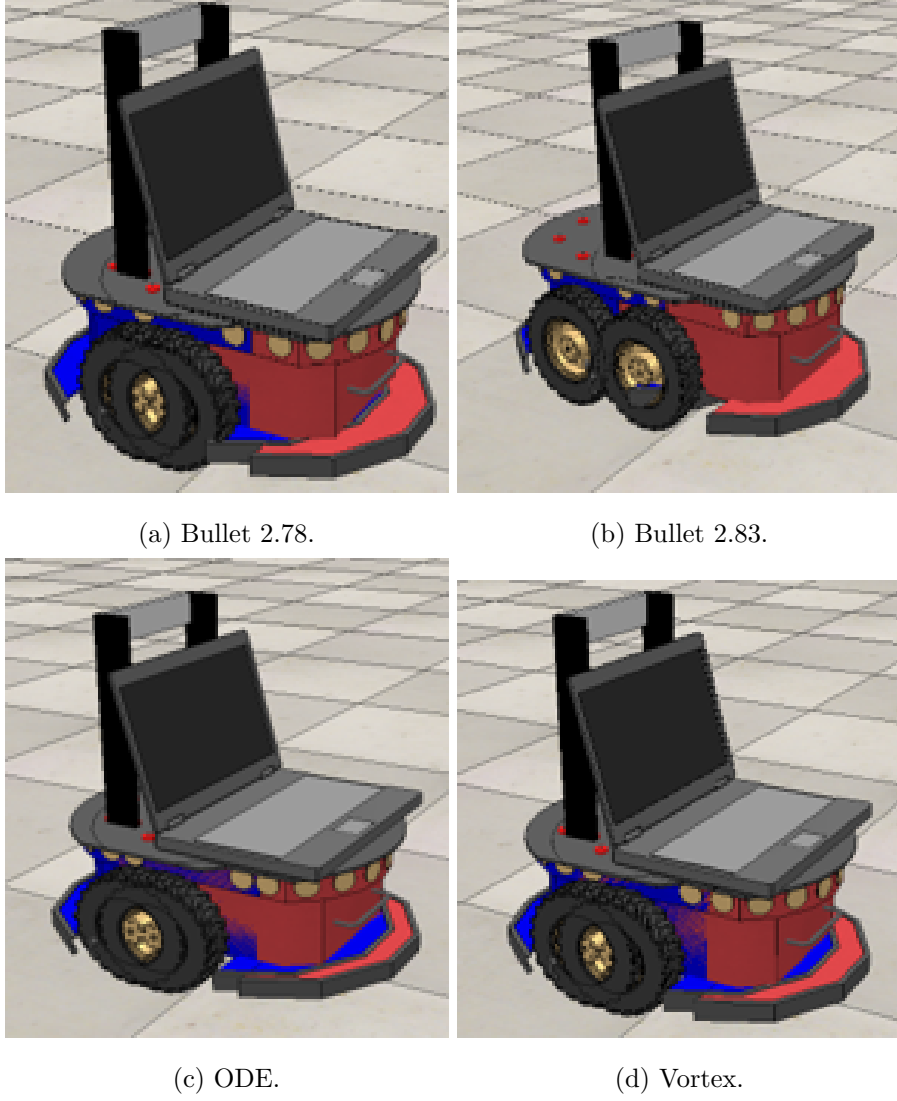


Figure 3.5: The average position error of each dynamic engine. Blue model is a real robot position; red model is robot model. (a) Bullet 2.78, (b) Bullet 2.83, (c) ODE, (d) Vortex.

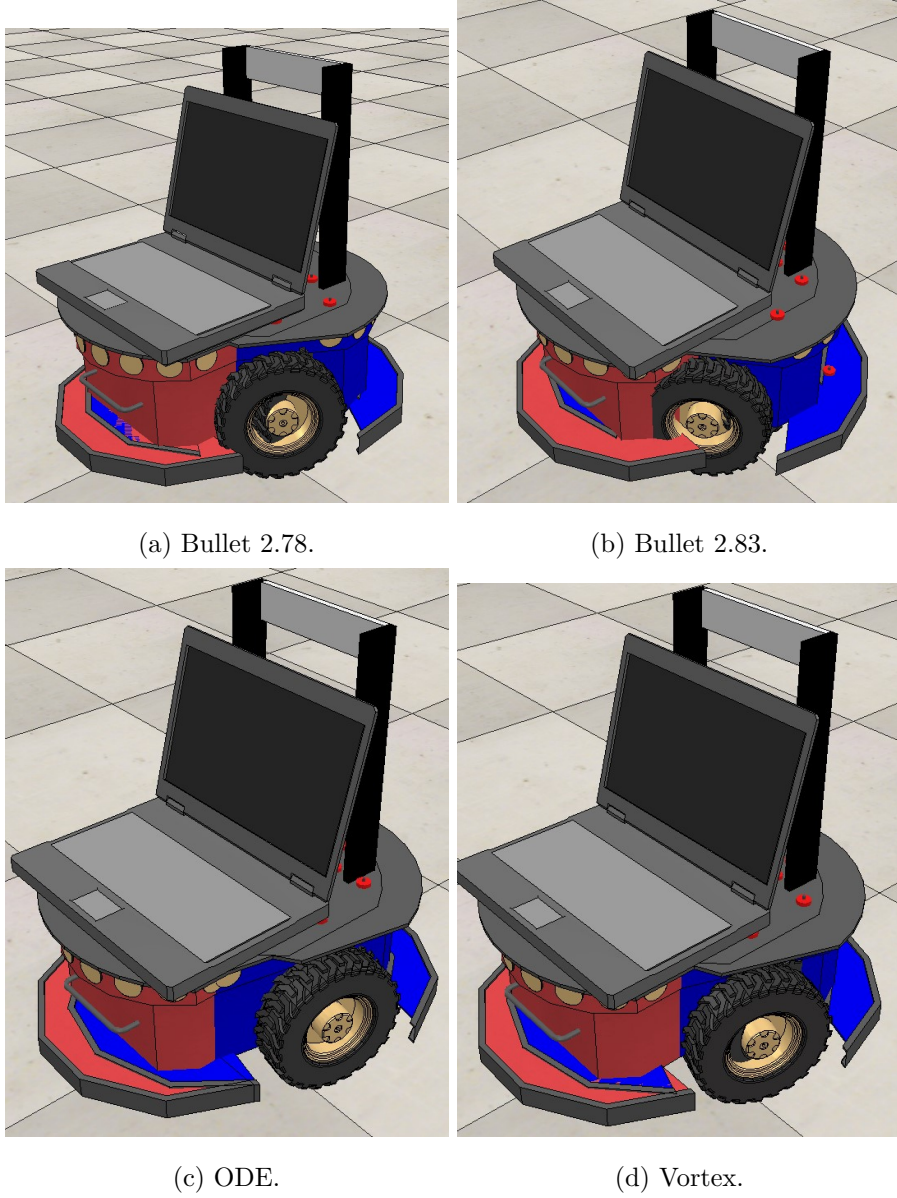


Figure 3.6: The average orientation error of each dynamic engine. Blue model is a real robot position; red model is robot model. (a) Bullet 2.78, (b) Bullet 2.83, (c) ODE, (d) Vortex.

3.2 CoppeliaSim Control Algorithm

We create an algorithm for real-time robot control by CoppeliaSim. Fig 3.7. shows the algorithm. This algorithm uses the dynamic engine in the red robot model for dynamic simulation control. Blue robot model represents the real robot position that obtains from `/ROS_Aria/odom`, transfers it to the CoppeliaSim global reference frame, and uses the LUA API to control the position of the blue robot model. The algorithm

Table 3.1: The difference value of estimate position and simulation values for straight movement

ν_{\max}	Bullet 2.78 (m)	Bullet 2.83 (m)	ODE (m)	Vortex (m)
0.5	0.064±0.003	0.162±0.012	0.046±0.018	0.052±0.044
1	0.068±0.051	0.164±0.049	0.064±0.049	0.068±0.052
2	0.072±0.014	0.137±0.015	0.07±0.018	0.07±0.014

Table 3.2: The difference value of estimate position and simulation values for angular movement

ω_{\max}	Bullet 2.78 (m)	Bullet 2.83 (m)	ODE (m)	Vortex (m)
0.5	0.1098±0.0178	0.3176±0.0249	0.074±0.01	0.152±0.064
1	0.2804±0.0679	0.308±0.031	0.0916±0.013	0.178±0.052
2	0.37±0.0171	0.303±0.0113	0.1258±0.023	0.187±0.066

matches the simulation time to ROS time and real-time, comparing the differences in position between the robot model and the real robot. We set the E_p and E_o is 0.1 m and 10 deg. Suppose the error position of the robot model and the actual robot exceeds 0.1 m, or the misalignment is greater than 10°. In that case, the dynamic model will automatically stop and adjust to the real-time robot model's position by the LUA API function. After that, the dynamic simulation resumed. In addition, we set the E_p and E_o are low value, the simulation always pauses to adjust the robot-model position to real-robot position, effects to increase simulation time(time to reach into goal position). However, resulting in less error of the simulator at the goal position. On the other hand, if the E_p and E_o values are set too high, it affects to increases the error value at the goal position.

In this experiment, we create the trajectory(goal position) as shown in Fig 3.8. The simulation model(red robot) moves to position A (black robot). After the robot reaches

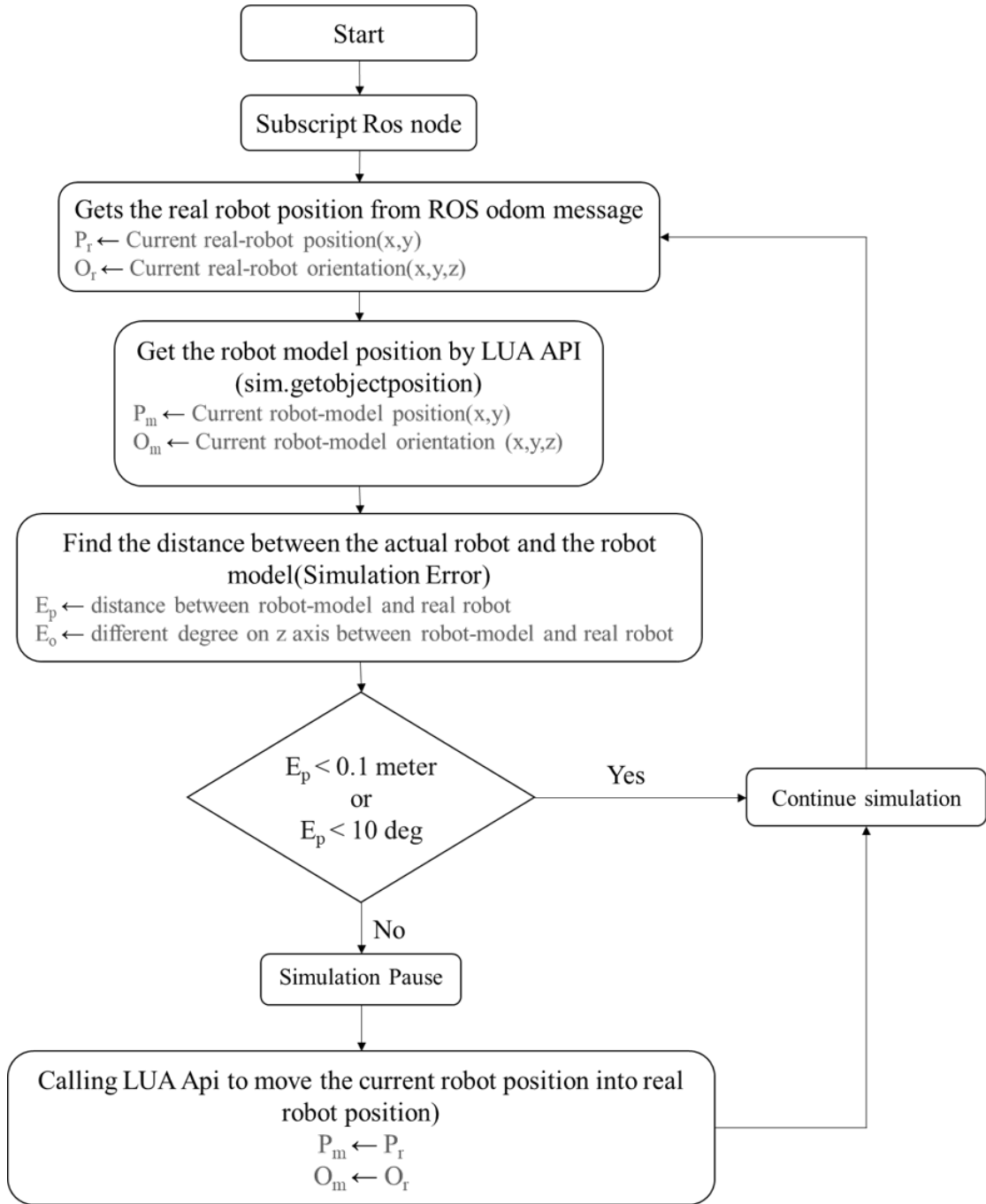


Figure 3.7: Reduce simulation error algorithm.

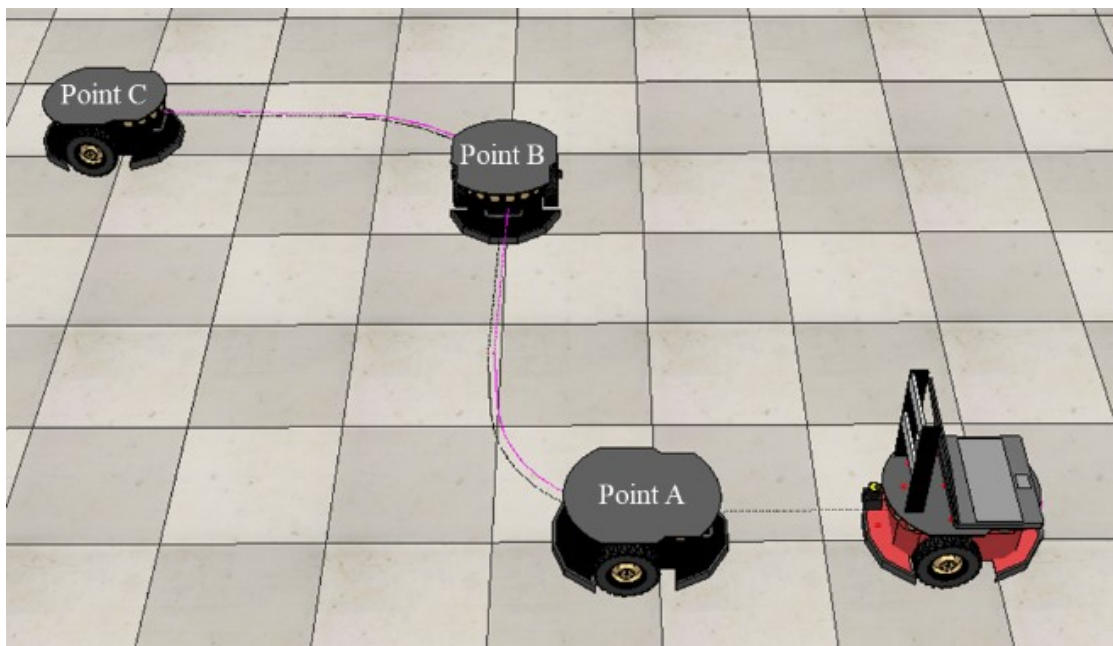


Figure 3.8: Simulation trajectory setting.

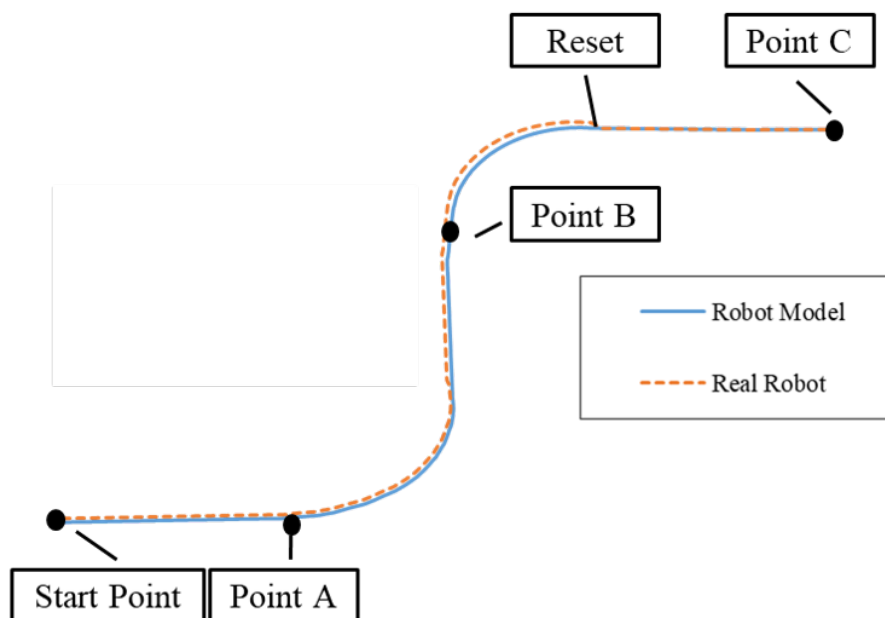


Figure 3.9: Robot trajectory.

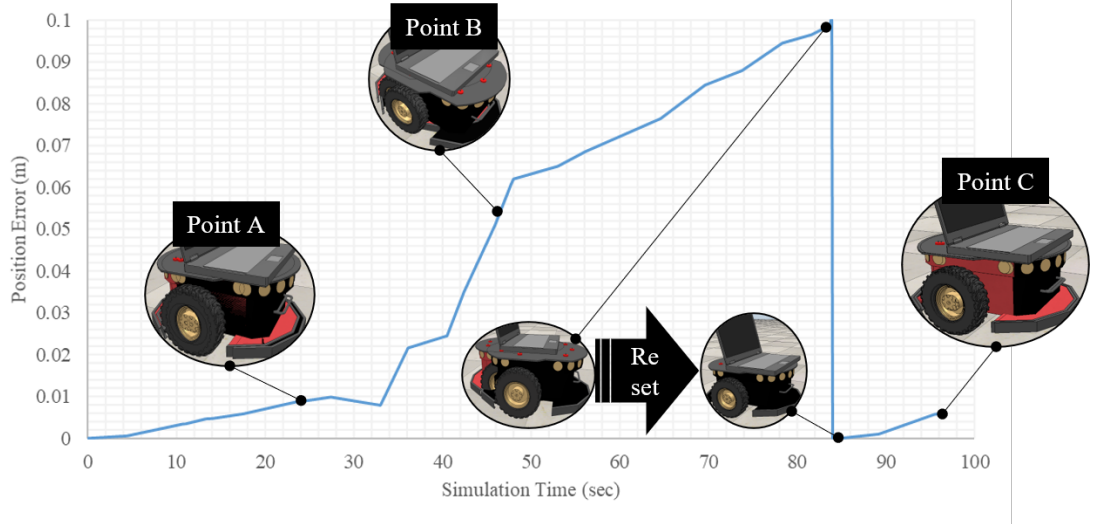


Figure 3.10: Position error.

the first target, it moves to the second target (position B). While the robot moves, the reduce error algorithm is active. The results shown in Fig 3.9 show that the blue and red curves represent the simulated and real trajectories, respectively. In the path experiment (from the start point to point A). The simulation error is slightly different. Following the accuracy test, these results show that the error in straight movement is small, as shown in Fig. Fig. 3.10. The position error between actual robot position with robot model. In the pathway from point A to point B, the position error is 0.013 m, the orientation error is 4.6° . The error in point A affects the accumulative error in trajectory in point B. The position error is 0.056 m, and the orientation error is 5.7° . The simulation automatically stops during the path from point B to C, the accumulated position error over 0.1 m. The actual robot position then continues the simulation to move to the next point, C. The error in point C is 0.094 mm, and the orientation error is 0.98° .

3.3 Chapter Summary

This chapter is presented an integrated system between CoppeliaSim and ROS system for control robot with dynamics engine. CoppeliaSim's subscribe and publish data in the ROS system. Linear and angular velocities of the robot model published to control real robots. The actual robot position and orientation were used to measure simulation error. The simulation will stop and repositions when the error is greater than 10 cm

or the orientation is greater than 10° . The result shows that CoppeliaSim can control the movement of the robot. The movement of the actual robot to positions A, B, and C and the model's movement with a few errors in the position. This error is caused by the plan flatness of the robot moving. Robots are placed on the carpet floor in the virtual environment, while the plane model is perfectly flat. This work also compares the accuracy of the dynamics engine included in CoppeliaSim.

We validated the accuracy in terms of position and orientation. The real accelerometer value provided by the ROS system is compared with the value obtained from the simulation. The results show that ODE is the most accurate while Bullet 2.83 causes more simulation errors. These results are similar to Michieletto's study[68]. Michieletto studied the accuracy of the CoppeliaSim simulator by the NAO robot. Further work involves similar adaptations for other robots with distinct characteristics that the simulation must reflect. For instance, quadrotor drones require an accurate particle simulation model that has to be coupled with IMUs and various specialized sensors 3D optical flow motion detection sensors.

CHAPTER 4

Real-Time Simulator for a Semiautonomous Teleoperation Robot in an Unknown Narrow Path

This chapter introduces the system to assist the operator in controlling the robot in a narrow path. This chapter begins with a description of the hardware setting and system framework. Next, introduce the Braitenberg algorithm and OMPL module. Thereafter, demonstration about the robot moves along with narrow path experiment, entering narrow path and intersection between the narrow path. In this experiment, we validated the experiment's performance by comparing the existing teleoperation system such as traditional system(Camera devices), 2D camera with map system, force feedback system.

4.1 Hardware Setting

The hardware architecture comprises two notebook PCs; the control PC and the simulation PC. We mounted the control PC on top of the real pioneer robot to command the real robot and receive real-time streaming data from the real robot and real devices. The simulation PC runs the simulation. Both notebook PCs operate in the same ROS system (ROS master is the control PC) and communicate via the SSH communication



Figure 4.1: Hardware Architecture.

system. In this system, the simulation PC can obtain (subscribe) real-time streaming data in the ROS system from a real robot connected with control PC (Fig. 4.1). The SSH is based on ROS multiple-machine concepts, designed with distributed computing in mind. A well-written node makes no assumptions about where in the network it runs. It allows computation to be relocated at run time to match the available resources (there are exceptions; for example, a driver node that communicates with a piece of hardware must run on the machine to which the hardware is physically connected). Deploying a ROS system across multiple machines is easy.

4.2 Developed System Framework

The communication system is shown in Fig 4.2 we developed, dividing the system into three parts: real world, ROS system, and CoppeliaSim. In the real world, the most popular differential driving the mobile robot, the Pioneer P3DX robot, is equipped with two-wheel control. We selected the Hokuyo laser rangefinder to get environmental information, and the joystick was used for robot control. The ROS system contains three ROS nodes: the ROS Aria-node connecting the simulation with the real robot, the ROS Hokuyo node used to connect the real Hokuyo sensor with the simulation, and the ROS-Joy node used to combine the integrated joystick with the simulation to the control robot. In CoppeliaSim, we used the same pioneer robot model as the real robot. We created a child script to communicate with the ROS system; each script communicated

with real devices in real-time or a control simulation scene. Child script (1) retrieved the real robot position from real-odometry data via the ROS Aria-node and included the data in the simulation scene. Child script(1) subscribing ROS Aria-node to subscribing /Aria/odom to receive a nav-msgs/odometry message. This message represents robot positions and velocity in free space with positions x, y, z, rotation about the x-, y-, and z-axes rotation, respectively. Therefore, able CoppeliaSim communicated with real-time robot position.

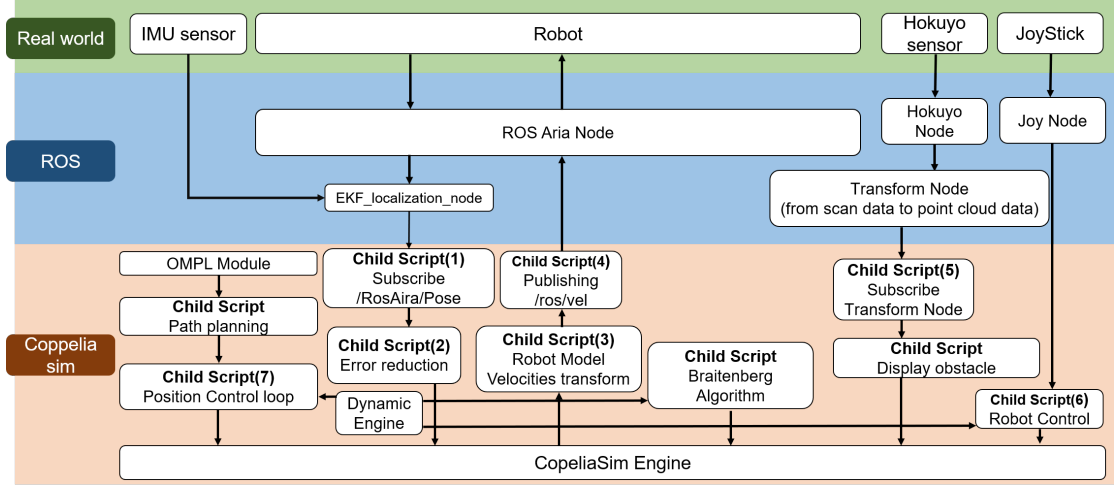


Figure 4.2: System Framework.

The topic /RosAria/pose published the odometry information. Therefore, Coppeliasim software subscribed RosAria/pose for determining the current position based on position link frame. In the Coppelia simulation, the robot model moved by following the data from the ROSARIA node (/rosaria/pose). The robot's position was moved based on the starting point frame, while the Coppelia simulation also uses the starting point frame based on the base link frame. The results indicated that in the simulation, the robot model could show the robot movement similar to the actual robot movement in the real world, as shown in Fig. 5.2. Moreover, Fig. 4.4 shows the robot trajectory between the actual robot and the simulation. The robot trajectory from the simulation and the actual trajectory was the same because, in the simulation, data from the actual and control position of the reference model were similar (i.e., starting point).

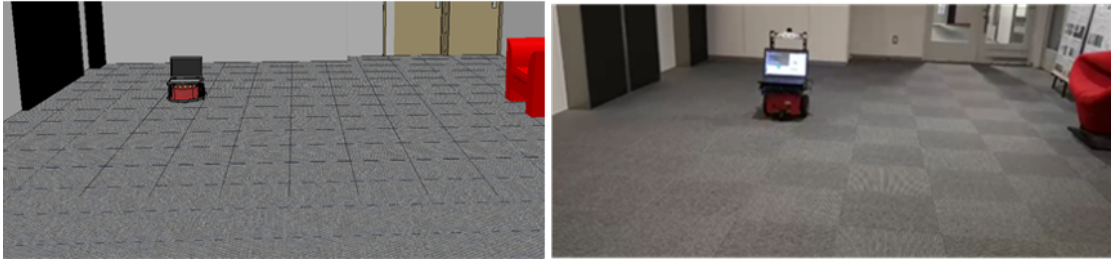
Child script(2) was an algorithm for reducing the simulation error by comparing the real robot with the model-robot positions. We applied the algorithm as explained in section 3.2. The child script(3) obtained the linear and angular velocities from the simulation



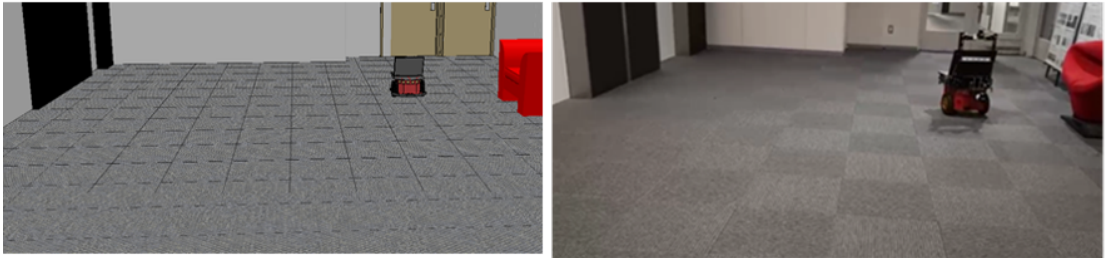
(a) A. Position 1 (Start position).



(b) B. Position 2.



(c) C. Position 3.



(d) D. Position 4.



(e) E. Position 1 (start position).

Figure 4.3: Communication to display the real-robot position into Coppeliasim scene.

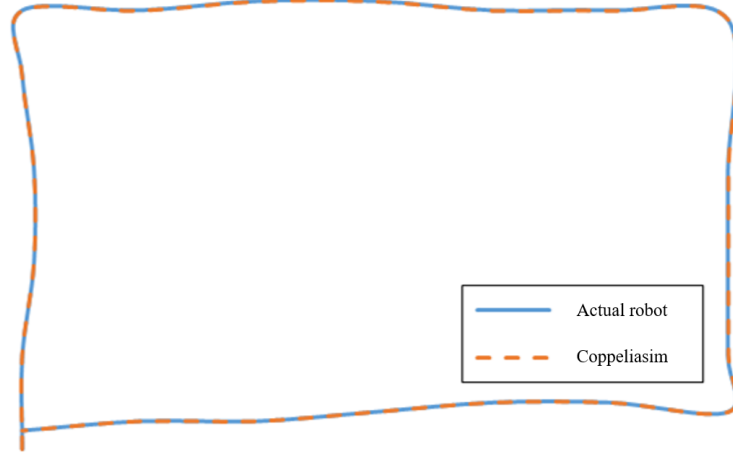


Figure 4.4: Comparing robot movement.

coordinate system and transformed them into the ROS coordinate system. The child script(4) was used to stream velocity data from child script(3) into the ROS Aria-node to control the actual robot motor. Used the LUA API command in CoppeliaSim to get the linear and angular velocities of the robot model with the base link. Generally, the LUA API in CoppeliaSim gets the linear and angular velocities of the robot based on the CoppeliaSim reference frame. Therefore, the robot base's linear velocities transforms on a robot-model base frame to control the real robot with geometry-msgs/Twist message via ROS Aria/node to control the real robot by linear and angular velocities and then we test the accuracy of the control system in section 3.1.

Child script(5) obtained environmental data from the Hokuyo node to display the obstacle information in the simulation scene via the point cloud function. CoppeliaSim communicated with ROS/Hokuyo node to get the scan data from Hokuyo sensor by sensor_msgs/LaserScan.msg, the message obtain the distance of the obstacle from equation 4.2.1. Here, D is the distance between sensor and target, c is the speed of light, ω is the angular frequency of the light wave, and φ is the phase change between the initial and reflected light rays.

$$D = \frac{c\varphi}{2\omega} \quad (4.2.1)$$

Therefore, we can get the distance between obstacle detection and sensor position. Then, we used the laser.geometry package to convert a scan message into a point cloud; the package provides two functions to convert a scan into a point cloud: project laser and

`transformLaserScanToPointCloud`. Project laser makes a straight projection from range-angle to 3D (x,y,z) without tf. It affects the converting speed fast; however, the point cloud results will be strange when the robot moves while the scan is taken because the point cloud will be in the same frame as the scan message. In contrast, `transformLaserScanToPointCloud` function uses tf to transform laser scan data into a point cloud message. The `transformLaserScanToPointCloud` uses tf to get a transform for the first and last hits in the laser scan and then interpolates to get all the transforms. This was more accurate when robots were moving in the world. So we used `transformLaserScanToPointCloud` function. We used to CopeliaSim API to call the point cloud object function and used it to display the obstacle data in Coppeliassim see in Fig. 4.5. Child script(6) received the joystick signal from the ROS-Joy node to control robot model movement. Coppeliassim communicates with Joy Node to get the joy state, and then LUA API sends a command to the Coppelia engine to make robot-model motor speed.

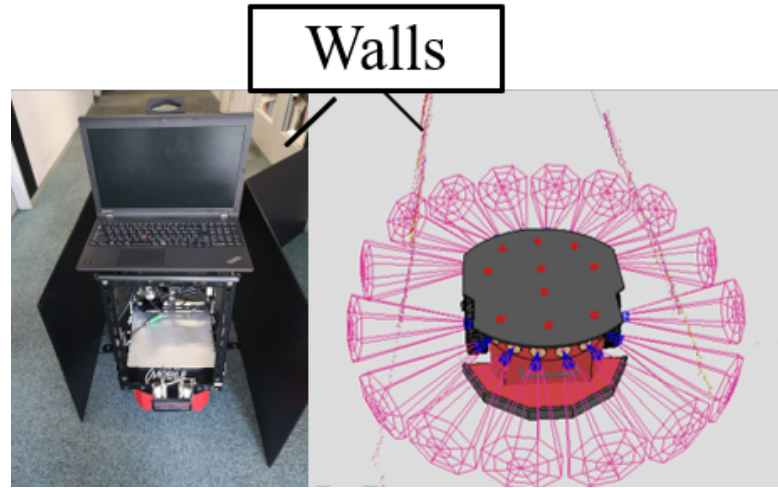


Figure 4.5: Obstacle(walls) displayed in Coppeliassim.

4.3 The Braitenberg Algorithm

A Braitenberg vehicle is conceived in a thought experiment by the Italian-Austrian cyberneticist Valentino Braitenberg. Sensors directly control the movement of the robot. A Braitenberg algorithm is a concept that can autonomously move around based on its sensor inputs. It has primitive sensors that measure some stimulus at a point and wheels (its motor drives each) that act as actuators or effectors. Generally, the connections between the sensors and actuators are divided into vehicle two and vehicle three, as

shown in Figs. 4.6 and 4.7. Vehicles are equipped with two sensors and two motors but with different connections between the sensors and the motors. Fig. 4.6 displays the concept of Braitenberg vehicle 2; the sensor in vehicle 2a is connected to the motor on the same side, called the direct connection or Permanent Love. Vehicle 2b and 3b motors are connected with the sensor on the opposite side, called the crossed-connection or Exploring Love see in Fig. 4.7 [69]. If the source is directly ahead, the corresponding motor runs faster, making vehicle 2a run away from the source (right wheel turns faster) while vehicle 2b runs toward the source (left wheel turns faster). In case the source is to one side, as shown in Fig. 4.7, the nearest sensor to the source is excited more than others; therefore, the equivalent motor runs slower to vehicle 3b, slows down, and turns toward the opposite side. Besides, vehicle 3a always slows down and turns to the opposite side. In [27], the study of combining with Braitenberg algorithm vehicle 2 and 3 for object avoidance navigation, combining sensors in the front or side sensors of the robot, in negative crossed-connection with the motor to avoid an obstacle in robot's front. Fig. 4.8 shows the connected motor by direct connection with the rear sensor to object avoidance in the back of the robot

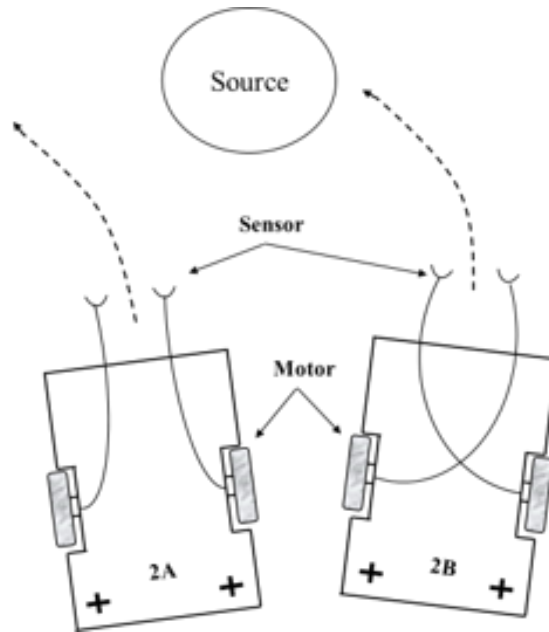


Figure 4.6: Braitenberg's vehicle 2a and 2b.

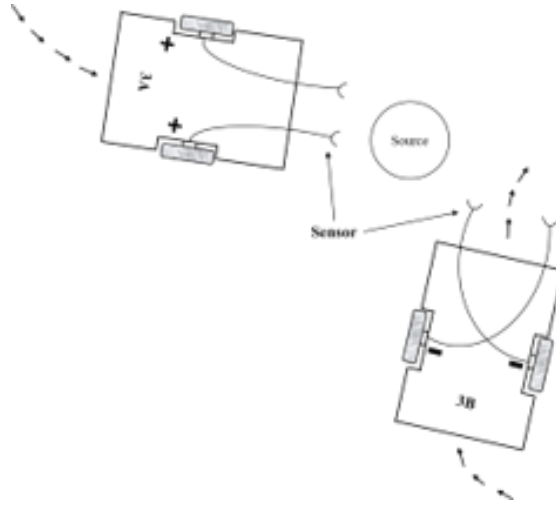


Figure 4.7: Braitenberg's vehicle 3a and 3b.

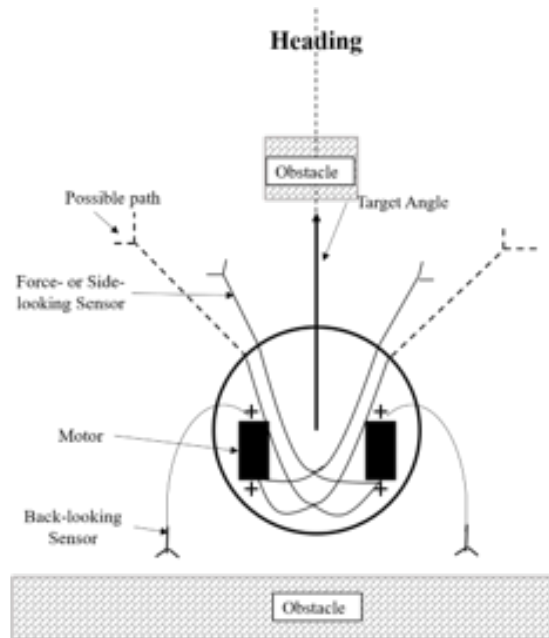


Figure 4.8: Braitenberg's vehicle modified.

4.3.1 Braitenberg Algorithm in Narrow Space

This study uses the concept of the Braitenberg algorithm to navigate the robot in a narrow path—a situation where obstacles surround the robot. Sensors are required around the robot to prevent certain robot parts from colliding. In a simulation scene, we demonstrate that when we only put a sensor in front of the robot, the rear side of the robot collides with the walls while turning within the narrow path. Fig. 4.9 shows the robot's side area colliding with the walls when we put the sensor position on the front.

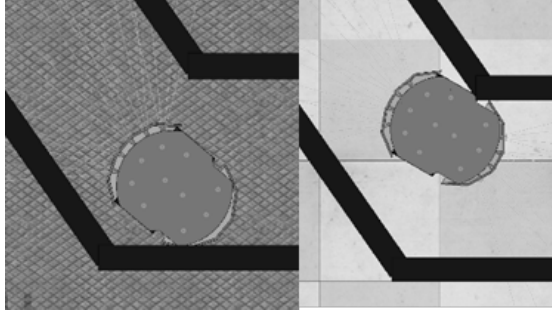
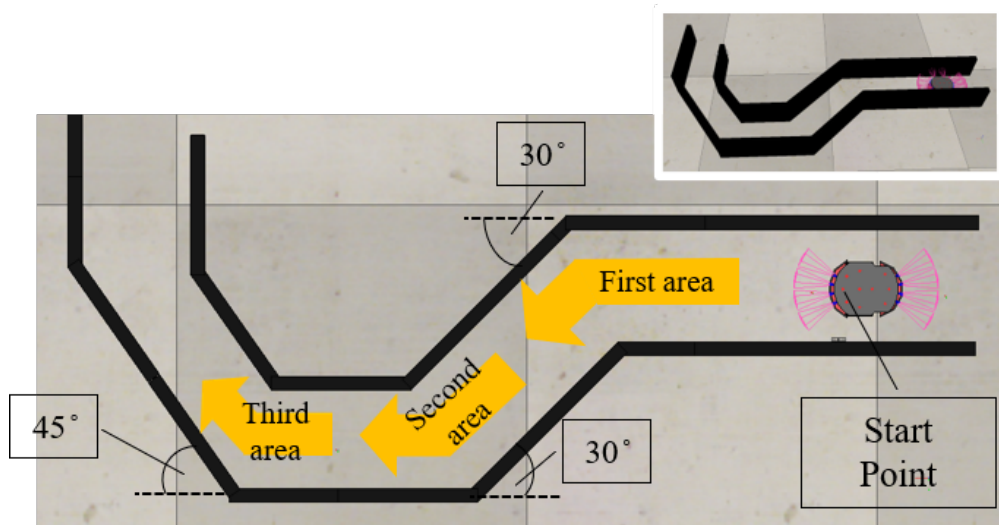


Figure 4.9: Simulation using the Braitenberg's algorithm with sensor on front and rear.

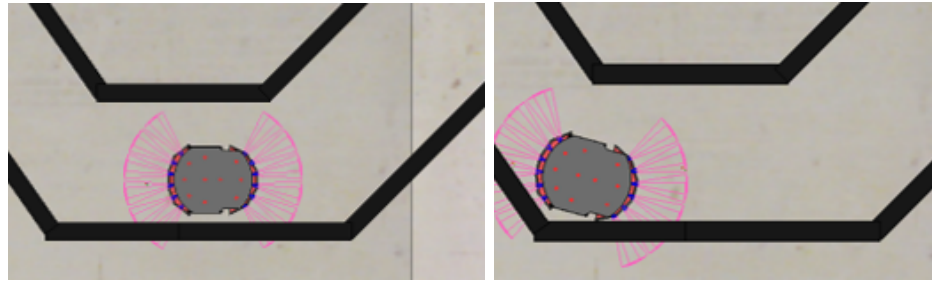
Fig. 4.10 shows the robot's side area colliding with the walls when we put the sensor only in the front and rear. The robot moves past in the first and second area of turning Figs. 4.10b and 4.10c, however in the Third area, the turning angle is high(45°), the robot collided with walls. At the same time, the robot collided with the walls when a number of the sensor could not cover around the robot in Fig. 4.10d. While another sensor condition robot colliding with the walls see Fig 4.11 when the sensor cannot cover all area of the robot because in narrow path space, the clearance distance between the robot and the walls.

The performance and efficiency of the Braitenberg algorithm depend on the number and location of sensors. However, in remote control of the robot, increasing the number of sensors affects the power requirements and shortens the run time. In addition, the limitations of the design may result in an inability to position the sensor properly. So, in this study, we applied the virtual proximity sensor in simulation to solve the problem. Virtual proximity sensors are built-in models in Coppeliasim. The simulator can run API commands to set the minimum distance between objects and sensors in CoppeliaSim. Virtual proximity sensors can be placed in any position of the simulation environment without any restrictions. As a result, it is easy to use and can solve the limitations of the Braitenberg algorithm. In additional API commands for data retrieved from virtual proximity sensors are available in the Coppelia Engine, reducing the program's computation time.

The number of sensors covering the entire area of the robot is 16 virtual ultrasonic sensors, as shown in Fig. 4.14, the positions of a virtual sensor on the side-surface of the robot. Eq. 4.3.5 and Eq. 4.3.4 were used to implement the Braitenberg concept to generate the left and right motor speeds, where, ω_L and ω_R are the weight factors of



(a) Virtual Environment

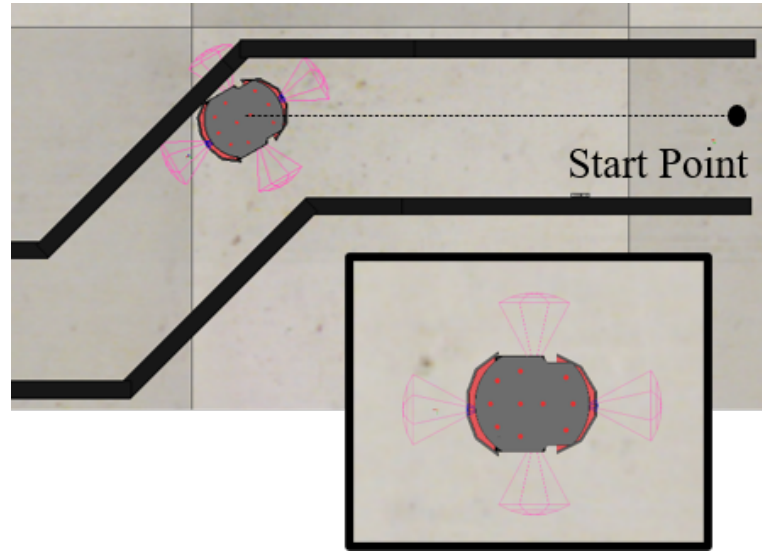


(b) Robot moves pass first and second (c) Robot starts to move along in sec-
ond section into second section

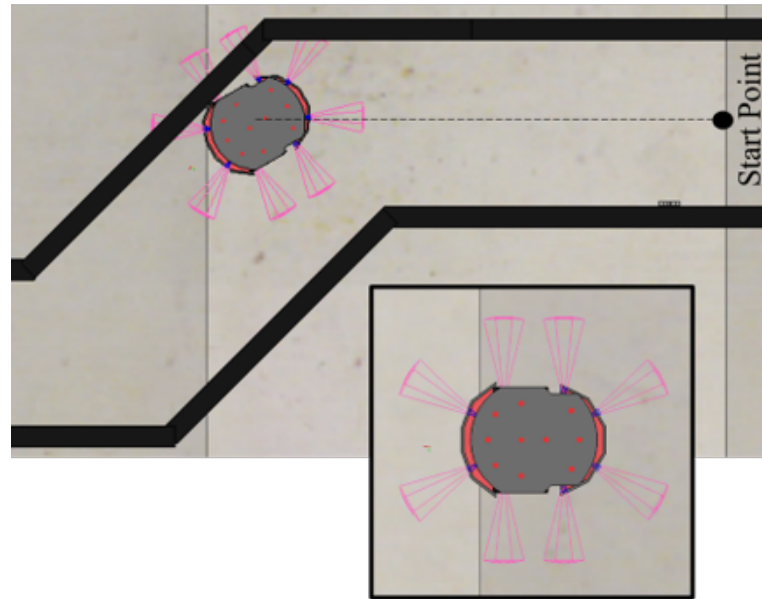


(d) robot collided with walls in third area.

Figure 4.10: Robot moves in narrow space within front and rear sensor conditions.



(a) Robot Colliding with the walls in first area of the turn with 4 sensor conditions (front, side and rear).



(b) Robot Colliding with the walls in first area of the turn with 8 sensor conditions (2 at the front area, 2 at side area and 2 at the rear area).

Figure 4.11: Robot colliding with the walls when the sensor cannot cover all area of the robot.

left and right motors respectively. Fig. 4.12 shows the weight factor of the left robot in Eq. ?? and Fig. 4.13 display the weight factor of the right robot in Eq. ??; The weight factor used to generate the speed of the robot, if the robot moves closer to an object on the front, the robot slows down and turns away from the object (weight factor of sensor number 1 to 8) on the other hand, it moves faster away from the object behind it (weight factor of sensor number 9-16). d_i is the detected distance of the virtual sensor i ; v_r is the velocity of the right wheel, and v_L is the velocity of the left wheel. d_{\min} is minimum distance that the robot can detected and d_{\max} is maximum distance that the sensor can detect see in Fig .4.15 v_{\max} for the maximum velocity of the robot is equal to 2 m/s (maximum speed for pioneer p3dx robot).

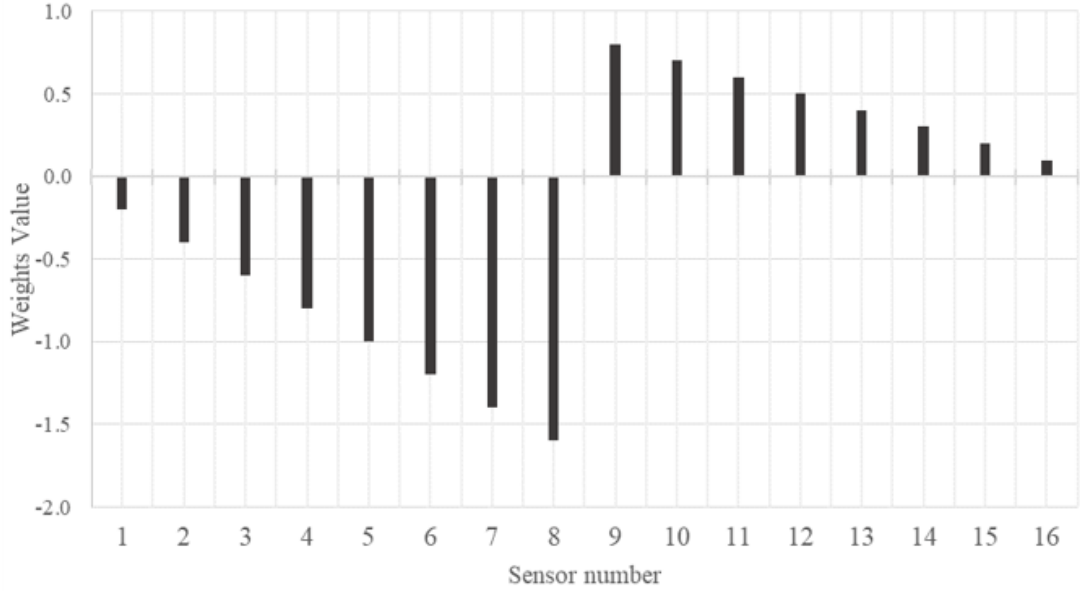


Figure 4.12: Weight factor of left-robot wheel.

$$\begin{aligned}
 &\{-0.2, -0.4, -0.6, -0.8, -1.0, \\
 wl_i = &-1.2, -1.4, -1.6, 0.8, \\
 &0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1 \}
 \end{aligned} \tag{4.3.1}$$

$$\begin{aligned}
 &\{-1.6, -1.4, -1.2, -1.0, -0.8, \\
 wr_i = &-0.6, -0.4, -0.2, 0.1, \\
 &0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8 \}
 \end{aligned} \tag{4.3.2}$$

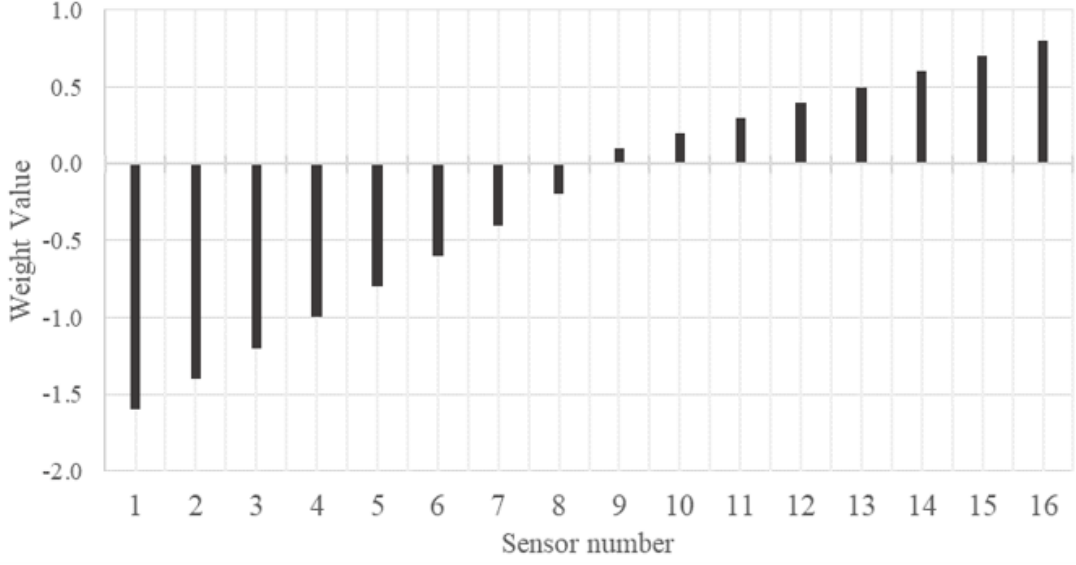


Figure 4.13: Weight factor of right-robot wheel.

$$d_i = \{d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8, d_9, d_{10}, d_{11}, d_{12}, d_{13}, d_{14}, d_{15}, d_{16}\} \quad (4.3.3)$$

$$v_r = \begin{cases} v_r = v_{max} & \text{if } d_i = d_{max} \\ v_r = v_{max} + \sum_{i=1}^{16} [wr_i (1 - \frac{d_i - d_{min}}{d_{max} - d_{min}})] & \text{if } d_i \leq d_{max} \end{cases} \quad (4.3.4)$$

$$v_l = \begin{cases} v_l = v_{max} & \text{if } d_i = d_{max} \\ v_l = v_{max} + \sum_{i=1}^{16} [wl_i (1 - \frac{d_i - d_{min}}{d_{max} - d_{min}})] & \text{if } d_i \leq d_{max} \end{cases} \quad (4.3.5)$$

We create the simulation scenes to preliminary test the concept of Britenberg in narrow space in Coppeliasim, shown in Fig 4.17. The robot moves from the start point into the goal point, and then we try to find the parameter for move-in narrow path space. The results are shown in Fig 4.16 the speed comparison between the left and right wheels at the start position. The sensor on the robot's left side (sensor numbers 2,1,16, and 15) detected the walls. The right speed was lower than the left speed (effect of the weighted parameter in Eq. 4.3.5 and 4.3.4 resulting in the robot moving away from the wall.

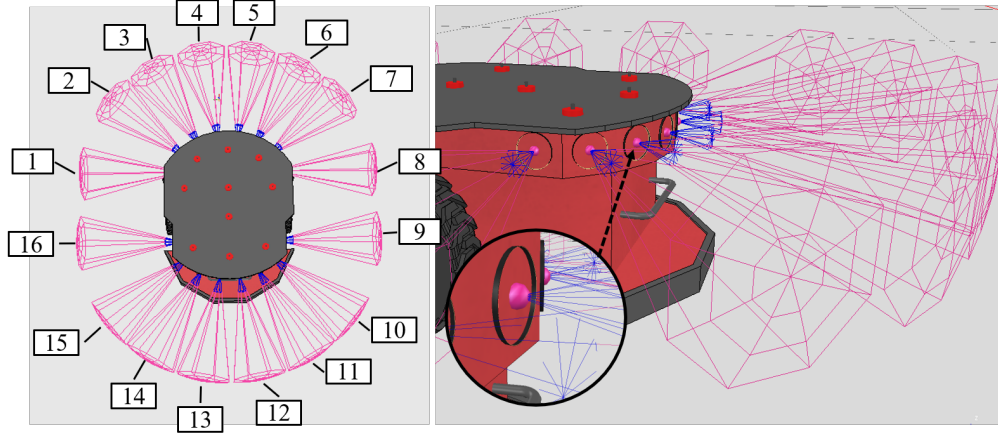


Figure 4.14: Pioneer Visual sensor model.

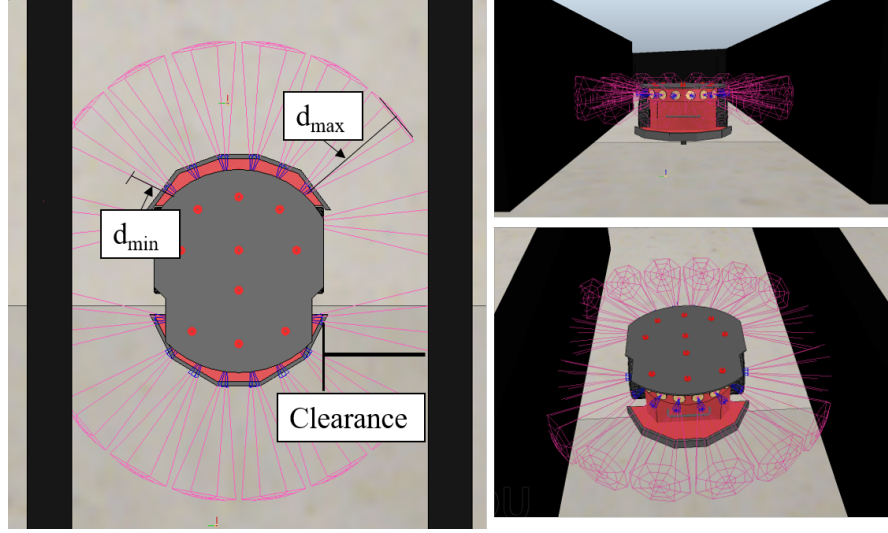


Figure 4.15: The description of d_{\max} and d_{\min} value.

After that, the speed of the right wheel increased, the robot moved away from the wall, one of the sensors on front was unable to detect the wall. In Fig. 4.16b the left wheel spins backward while the right wheel spins forward at the first turning area with 60° . When the front and right sensors detect a wall, as a result, the robot does not collide with the wall.

We found the parameters d_{\max} and d_{\min} for Eq. 4.3.5 and 4.3.4 by Coppeliasim, creating the small paths shown in Fig. 4.17. The distance between the walls is 0.6 m. The robot-model size is 0.4 m. Therefore, we had a clearance of 0.2 m between the robot and the walls. The angles of the narrow path are 60° , 30° , 90° , and 30° , respectively. Table 4.1 shows the results of the simulation when simulated with the d_{\max} and d_{\min}

factors. Fig.4.15 shown the relationship of each parameter. In the simulation results, we observed that if the values of factor d_{\max} are more than the clearance value between the robot and walls, the robot cannot move along with narrow space (speed of the robot are decreases). The results from the primary simulation suggested d_{\max} would be less than the clearance between the robot and walls. while d_{\min} are the minimum distance between the robot and obstacle, due to we used the simulation error algorithm as explain in topic 3.1 therefore, the d_{\min} are 0.1 m ; hence, we used d_{\max} and d_{\min} as 0.15 and 0.1 m, respectively.

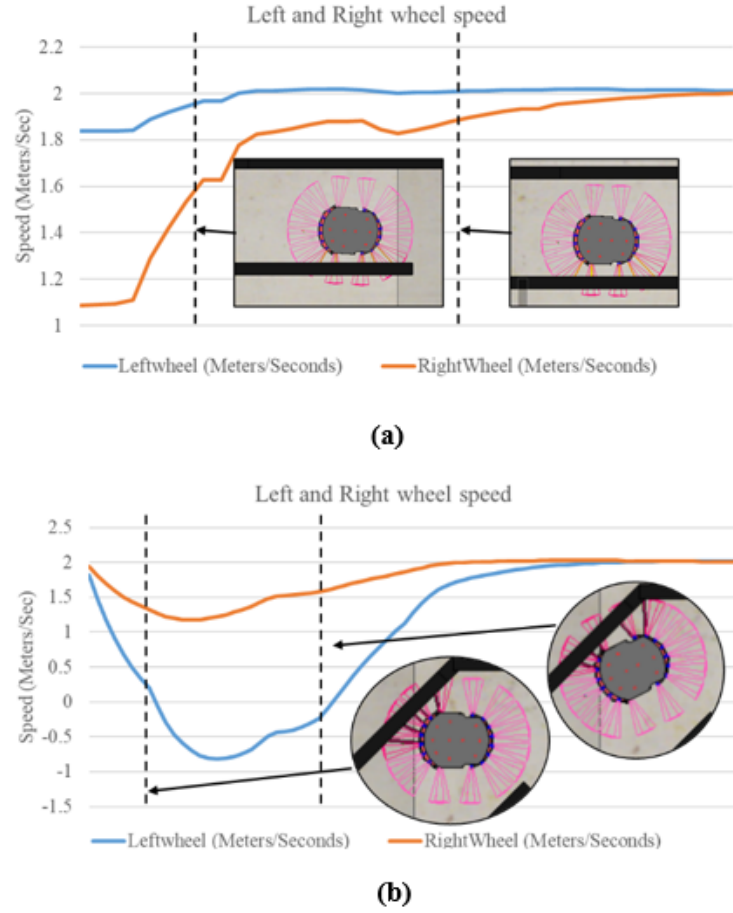


Figure 4.16: Comparison of speed between the left and right .

The primary tested the system's performance for operator-assisted control robot movements in narrow paths using the simulator. Fig. 4.17 shows the narrow path environment. In the experiment, we fixed the width of the narrow path to 0.55 m. In the first turning area section, the narrow path turns left 60° relative to the pathway, then moves straight 1.5 m into the second turn area section and turns right 30° . The narrow path

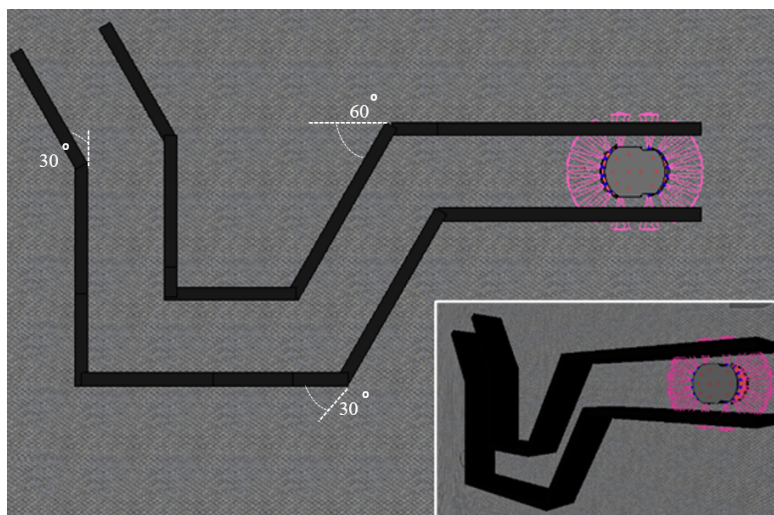


Figure 4.17: Virtual narrow path in Coppeliasim.

continues to move straight into the third turning area, 90° relative to the straightway, and then moves straight with the final turning area 30° relative to the goal position.

Table 4.1: Narrow path simulation results

Codition		Path width(mm) ,clearance		
dmax	dmin	550 , 20	600 , 35	700 , 45
0.25	0.1	cannot moves	Pass	Pass
0.2	0.1	Cannot move	Pass	Pass
0.15	0.1	Pass	Pass	Pass
0.1	0.1	Pass	Pass	Pass

4.4 Entering Narrow Space and Intersection Between Narrow Path

One of the limitations of the teleoperation system and the Braitenberg algorithm is the difficulty of directly controlling the robot into narrow pathways or intersections between narrow pathways. Since the robot's speed depends on the value obtained from the sensor, this makes it hard to direct the robot to the desired location. Therefore, we selected the OMPL module in CoppeliaSim to solve these situations. The flowchart seen in Fig 4.18. We developed a child script in CoppeliaSim for calling the OMPL path planning module when the operator called the OMPL path planning system to move into the narrow path. The OMPL modules required four parameters: start position, goal position, state-space, and collide object. We set the current robot position to the start position in the start position. At goal position, an operator sets the goal position by robot dummy seen in Fig 4.19. In state-space, seen in Fig 4.20, the size of state space is $2.3D_x$ and $2.3D_y$. Hence, D_x is the distance between current robot position with goal point in the x-axis, and D_y is the distance between current robot position and goal point in the y-axis. We used 2.5 times of distance between the robot and the goal point because the state space should be cover the area between the robot and the goal point. However, the robot does not require large state space in narrow path simulation, reducing simulation calculation time. We designed every object in the simulation scene for colliding objects, except the robot model, via the applied collection function in CoppeliaSim. After that, the OMPL module autonomously generated the path from the robot start point into the goal point Fig 4.21. Then we create the child script for use path following module to control the robot model move following the path generated by OMPL and publishing the speed into the ROS system. The actual robot subscribes the speed for moving into the design point(goal point). We compared the real-time robot model position and real-robot position to reduce the simulation error, as we explain in section 3.1. The simulation pauses and automatically adjusts the robot-model position into the actual robot position, regenerates the path, and moves following the path until the robot reaches the goal point.

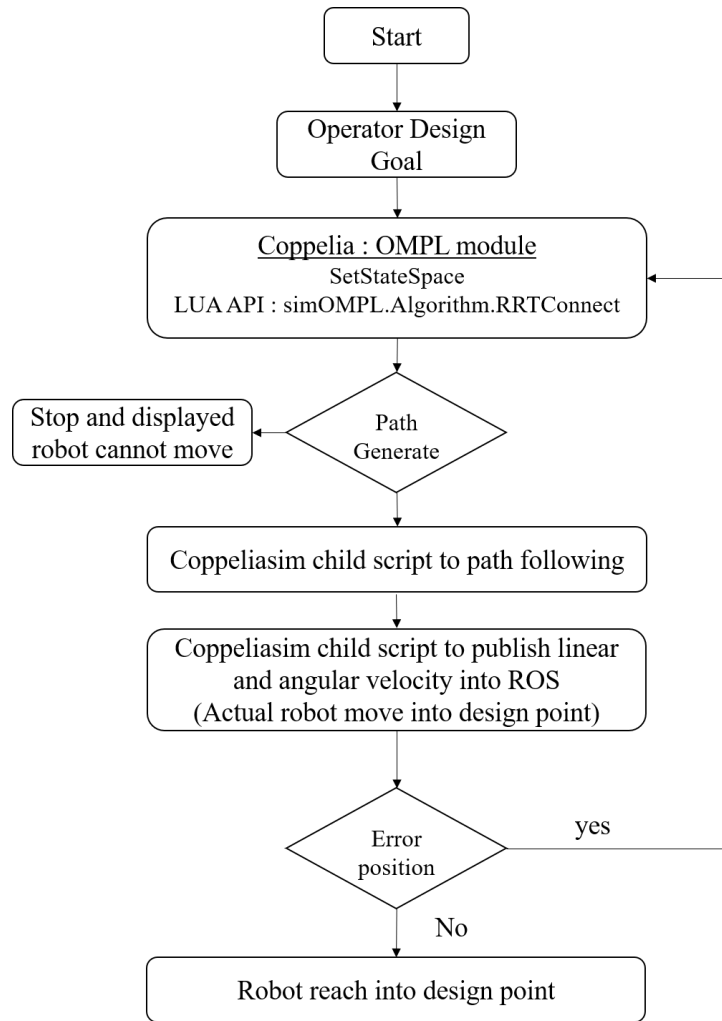


Figure 4.18: OMPL system.

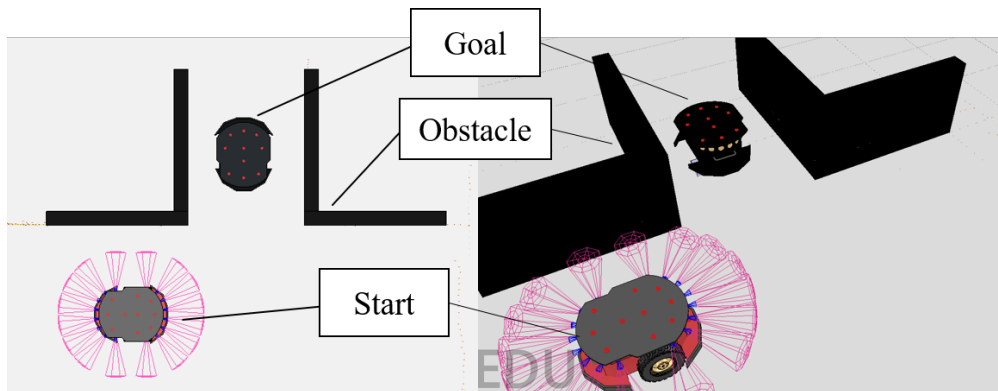


Figure 4.19: OMPL path module simulation in Coppeliasim.

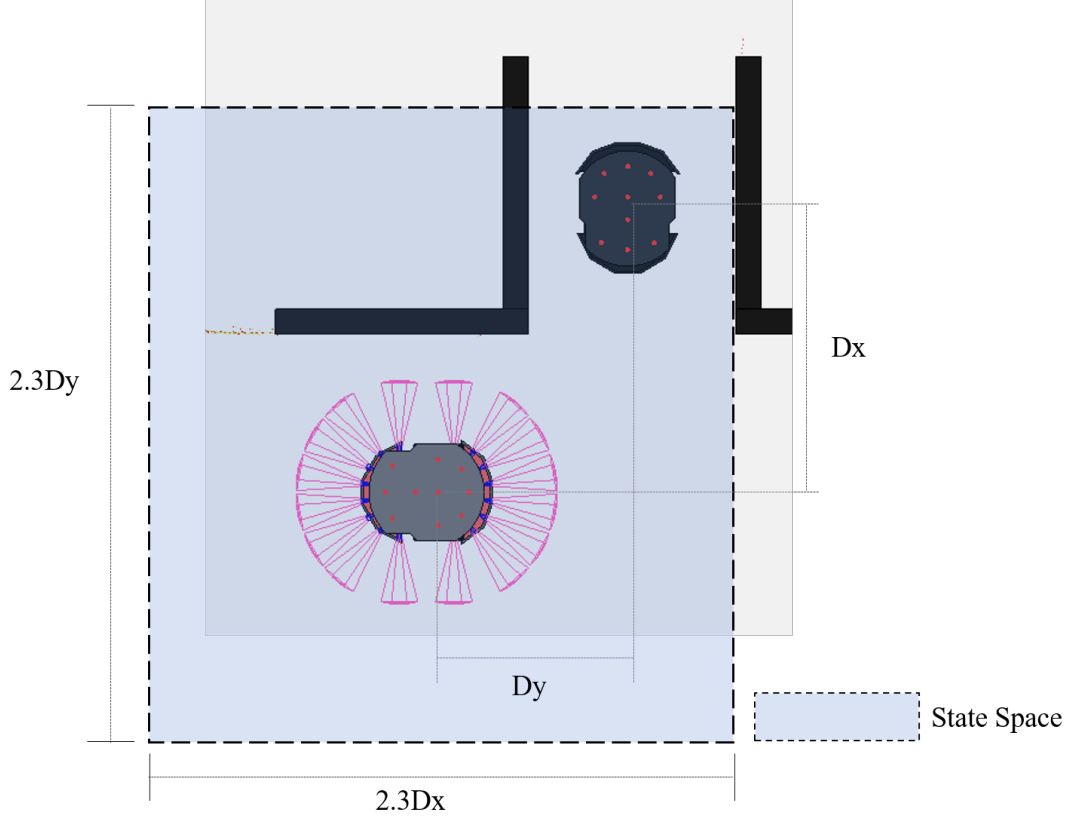


Figure 4.20: OMPL State Space.

4.5 Robot Moves in Narrow Space Experiment

It is essential to investigate the concept of a real-time simulation that employs virtual sensors for the Braitenberg algorithm, which aids robots in the narrow pathway in real-world environments. We created a real-world narrow path environment to put the Braitenberg algorithm to the test. Following that, narrow path entrances and intersections between narrow paths were created to validate the OMPL module's ability to control the robot's movement into the appropriate position in narrow paths.

4.5.1 Robot's Movement Along with a Narrow Path

This section describes the experiments conducted to evaluate the proposed system's performance for operator-assisted control robot movements in a narrow path. Fig. 4.22 depicts the created narrow path environment at the first experiment. We fixed the width of the narrow path to 0.55 m, at the first section of the turning area turns right at 60° relative to the pathway. Then moves straight for 1.5 m into the second section of the

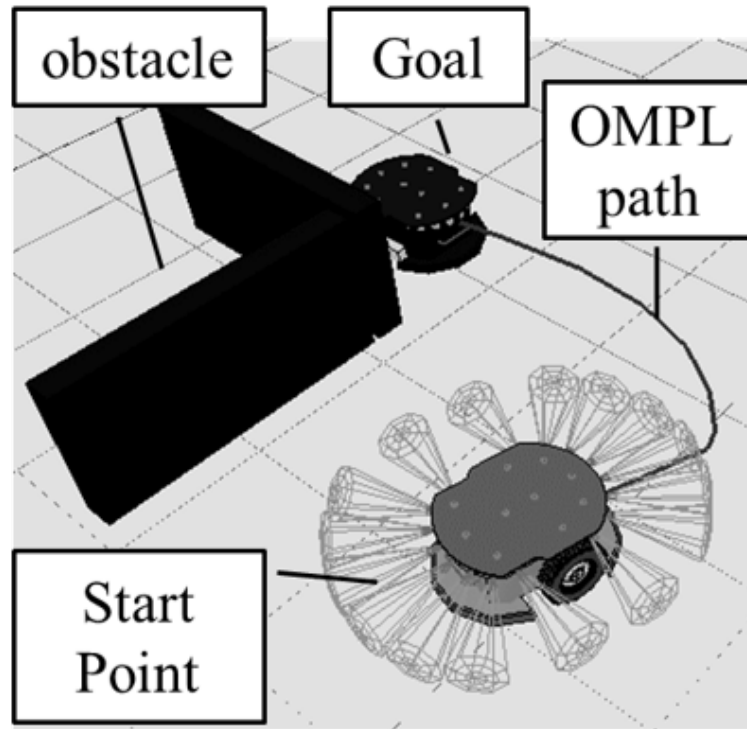


Figure 4.21: OMPL path generate.

turning area and turns left at 30° . The narrow path continues straight into the third turning area, which is 90° relative to the straightway, then moves straight into the final turning area, which is 30° relative to the goal position.

The operators could control the robot from the teleoperation room. Before the experiment, the operators were unaware of the surroundings. The operators call the Braitenberg algorithm to control robot movements along the narrow path into the goal position. During the manipulation of the robot with the Braitenberg algorithm, the operator controls the robot by pressing the forward button in the joystick, which makes the system generate the robot speed moving along a narrow path. Next, we created the narrow path environments (Fig. 4.23). We tested the wall's constancy width by gradually narrowing the environment's width with distance. At the starting position, the wall width was 0.75 m, the width gradually narrows to 0.55 m in the existing area. In addition, we used the same algorithm and parameters used in the previous experiment.

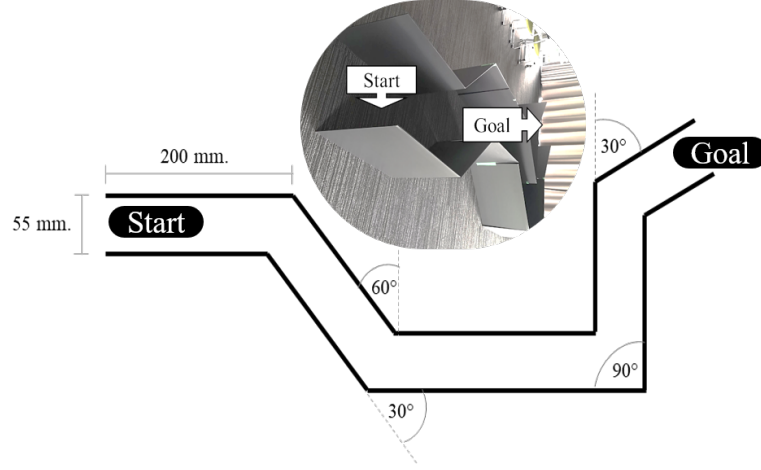


Figure 4.22: Narrow path environment.

4.5.2 Entering the Narrow Path and Intersection in the Narrow Path Experiment

The Braitenberg algorithm for the assistance-control robot, which generates velocities autonomously, makes it difficult for the operator to control the robot in certain environments, such as the entrance of the narrow path or the intersection during narrow path movement. We applied the OMPL module in the CoppeliaSim to generate linear and angular velocities for a goal position to solve that problem. This experiment tested two conditions; entering a narrow path situation and an intersection in a narrow path.

In the entering narrow path experiment, we created the environment as shown in Fig. 4.24. The width of the narrow path was 0.55 and 0.6 m. We also created the entrance of the narrow path environment by blending the narrow pathway 45° for turning left and right. The operator defined the goal to move into the narrow path after starting the simulation. The system calls OMPL for the generated path to move the robot into the goal position.

We were creating the experiment environment in Fig. 4.25. In this experiment, we combine the narrow path with turning area and intersection inside the narrow path. The operator controls the robot from open space(start position) moves into a narrow pathway with an entrance area path width of 0.75 m. The narrow path contains two junction areas inside a narrow path. The first junction area consists of two junctions (left and right) at an angle of 45° relative to the narrow path. The second junction area consists of three junctions (straight, left, and right); the operator should control a

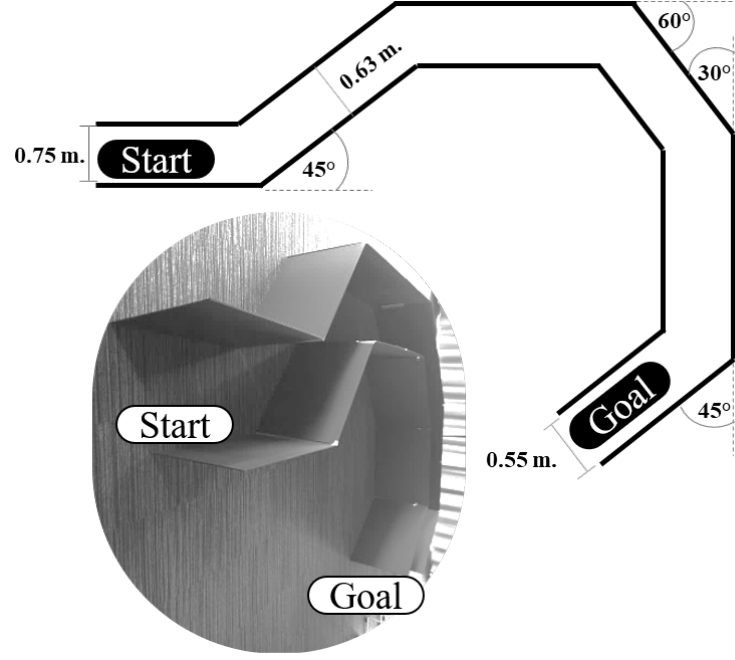
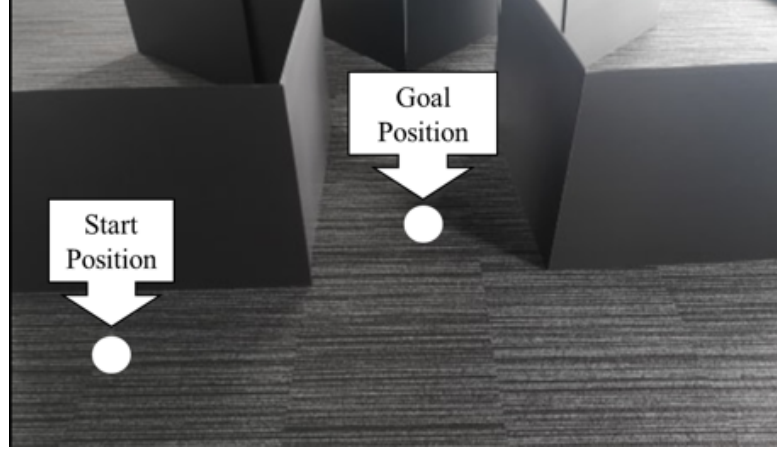


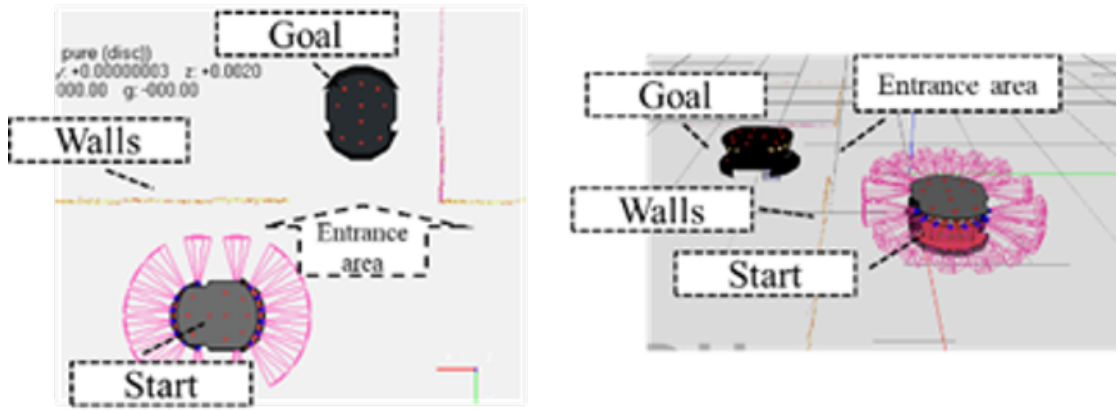
Figure 4.23: Narrow path environment in second experiment.

robot with a turn left with 90° relative to the pathway into goal position. In addition, a narrow path consists of three turning areas, with the first turning area was 30° after the robot moves straight from the first junction area. At the second turning area, the narrow path blending with 45° and 60° in the last turning area.

In Fig. 4.25 we are creating the experiment environment. We combine the narrow path with a turning area and an intersection inside the narrow path. The operator controls the robot from open space(start position) onto a narrow pathway with a path width of 0.75 m. Inside the narrow path, there are two junction areas. The first junction area consists of two junctions (left and right) at a 45° angle to the narrow path. The second junction area consists of three junctions (straight, left, and right); the operator should control a robot with a 90° turn left relative to the pathway into the goal position. In addition, a narrow path consists of three turning areas; the first turning area was 30° after the robot moves straight from the first junction area. The narrow path blends 45° and 60° in the last turning area at the second turning area.



(a) Real scene.



(b) CoppeliaSim scene display when robot move in entering narrow space.

Figure 4.24: Entering narrow path experiment environment.

4.6 Experiment Results and Discussion

We divided the results into three parts. First, we examined the Braitenberg algorithm, which can help operators control the robot's movement along narrow paths. The second experiment examined at the OMPL module involved entering the narrow path area and the narrow path situation intersecting. Third, we examined a proposed system for combining Braitenberg and OMPL to move robots from open spaces to narrow paths. We compare the proposed system to a traditional teleoperation system, a camera and map data obtained by the ROS framework with LRF, and a force feedback system. In the traditional system, the operator controls the robot to finish the experiment by obtaining environmental data using the 2D camera attached to the robot and obstacle information from LRF. The concept of this system applied from [70][71][17]. In a 2D camera with a

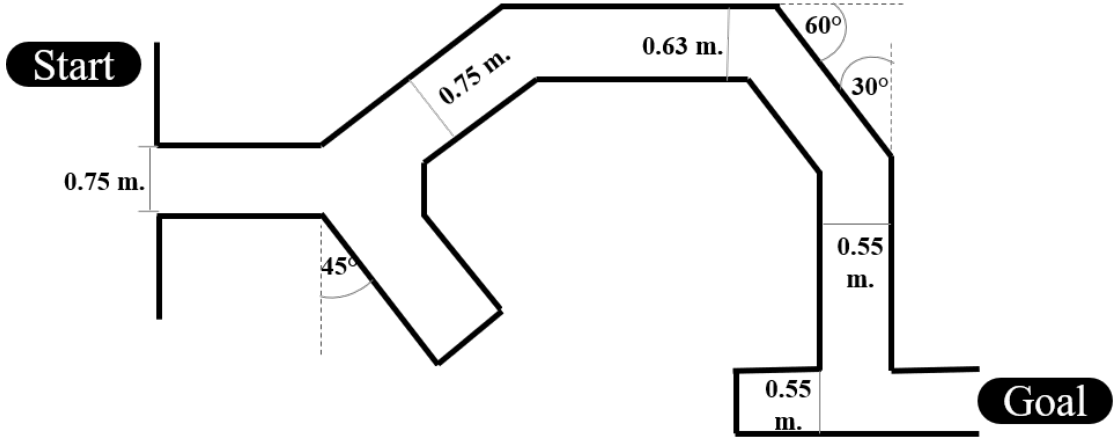


Figure 4.25: Experiment environment.

map data system, the operator controls the robot to finish the experiment by obtaining data from the 2D camera and real-time map built by the ROS framework with LRF as presented in [12][72][73]. We also compared the proposed system to the variable force feedback teleoperation system. In the force feedback experiment, the operator senses the obstacle by the vibration of the joystick when the robot moves in along narrow space and also obtains the force feedback value in teleoperation-PC screen the concept as presented in [9]. Finally, we compared our results to those of a previous experiment in our lab called the traversing teleoperation system. The traversing teleoperation is a concept to generate the velocities of the robot to move along a narrow path. The system obtains data from LRF in front of the robot to auto-generate the robot move along with narrow path as presented in [2].

4.6.1 Robot's Movement within a Narrow Path

Figure 4.26 shows that five experiments tested the robot's trajectory. The operator pushed the forward key on the gamepad; the proposed system assisted the operator in controlling the robot's movement from the start point to the goal without collision. We observed a slightly different trajectory at the starting position because we could not place the robot in the same position during the experimental setup step. However, d_{\max} and d_{\min} are suitable for the Braitenberg section, so the robot adjusted the trajectories; the trajectories were stable after the robot moved 0.5 m. The robot's turns along each narrow path in each turning area were almost identical in every experiment. At the

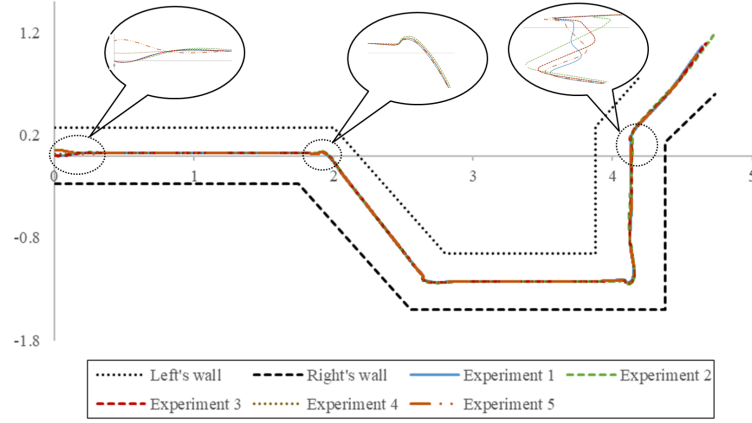


Figure 4.26: Trajectories of the robot move along with the narrow path.

first turning area with 60° , the proposed system could assist the real robot movements along the narrow path as shown in the simulation model in Fig. 4.27a. Fig. 4.27b shows the direction of the robot in the second turning area. Fig. 4.27c shows the results when the robot moves with a turning angle of 90° ; the results showed that the real robot moved along the narrow path without any part of the robot colliding with the walls. Fig. 4.27d shows the final turning area; the robot also moved along the narrow path without collision.

Figure 4.28 shows the linear and angular velocities of the robot during movement within the narrow path. The linear and angular velocities generated by the Braitenberg algorithm were too high at the starting point because the robot adjusted the trajectory; afterward, during straight movement, the angular velocities decreased, whereas linear velocities remained stable at 0.15 m/s. At the first turning area, the robot's linear velocities decreased. By contrast, the robot's angular velocities increased for the robot turning direction to move within the narrow path at a change direction of 60° . The angular velocities became small, and linear velocities were stable at about 0.15 m/s when the robot moved straight along the narrow path. After using the Braitenberg algorithm, the linear velocities rapidly decreased, whereas the angular velocities increased. At the third turning area, the linear velocities were minimal because the section's turning angle was 90° , so all virtual proximity sensors in front of the robot detected the walls, making velocities generated from the algorithm reduce.

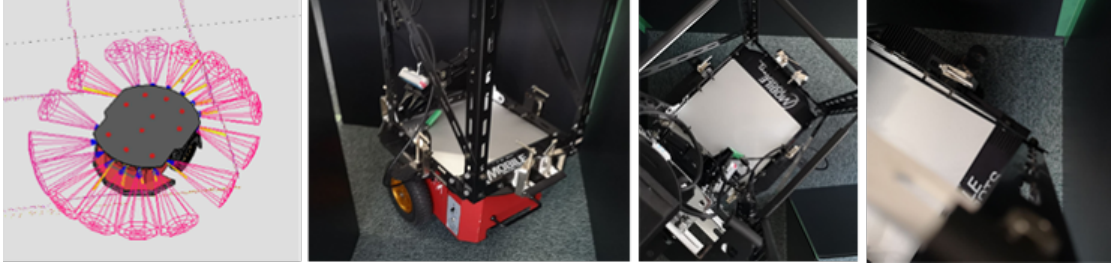
Figure 4.29 shows the proposed system's results, with the robot moving with assistance along the narrow path without collision, unlike the previous experiment environment.



(a) First turning area with blending angle of 60° .



(b) Second turning area with blending angle of 30° .



(c) Third turning area with blending angle of 90° .



(d) Fourth turning area with blending angle of 30° .

Figure 4.27: Comparing the simulation scene with the real robot scene.

However, the robot movement had the same characteristics as in the previous experiment.

Table 4.2 compares the proposed system's teleoperation time in the narrow path scenario. Previously, the operator was unaware of the environment (with ten operators con-

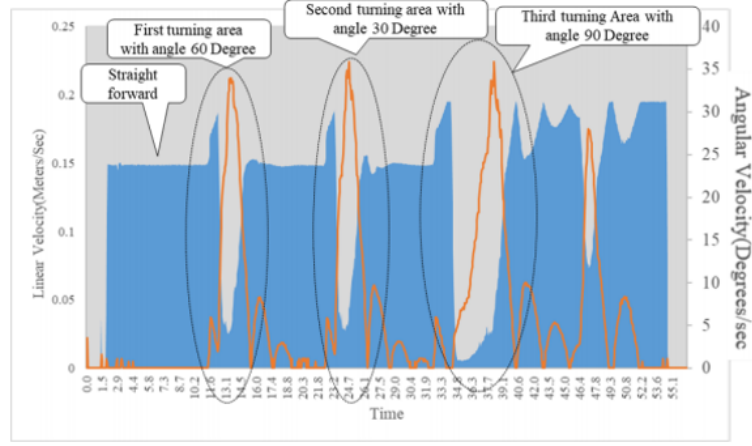


Figure 4.28: Linear and angular velocities.

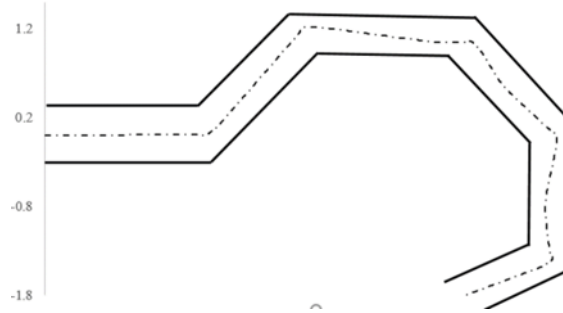


Figure 4.29: Trajectory of the robot during move along with narrow path in experiment 2.

trolling robot movement). The experimenter experimented with controlling the robot in the first and second environments (Figs. 4.22 and 4.23). The results showed that with the 2D camera system, the operator's difficult to control robot moves along narrow paths with high uptime and high standards with each trial. It is difficult for operators to recognize the distance between the wall and the robot's sides. As a result, compared to the planned system, the number of collisions with the wall was significant. There were no collisions with the walls when the system was controlled by a 2D camera and a map information system. However, the time required to finish the course is long compared to the proposed system because the distances between the walls around the robot are small. As a result, the operator must carefully maneuver the robot in the narrow space. The operator focused the robot's movement along the path, particularly where the path changes direction. As a result, the operator must pay close attention to the

Table 4.2: Comparison of the robot moves along with a narrow path

Environment	Teleoperation System	Time to finish (sec/std)	Number of Collided (Avg/std)
Environment 1	2D camera (only)	126.4 / 14.34	8 / 2.64
	2D camera + obstacle information	91.8 / 5.42	0.96 / 0.45
	Force feedback system	100.3 / 13.5	0.13 / 0.33
	Traversing a narrow path	67.3 / 6.5	2.32 / 1.35
	Propose system	54.8 / 3.96	0
Environment 2	2D camera (only)	150.6 / 12.31	9 / 1.34
	2D camera + obstacle information	110.4 / 9.31	0.87 / 0.31
	Force feedback system	123.3 / 5.6	0.15 / 0.33
	Traversing a narrow path	68.4 / 3.5	2.33 / 0.65
	Propose system	63.4 / 3.13	0

robot's position and the distance between the robot and the obstacles. There were very few collisions with the walls because of the force feedback system. The quality of the robot's motion, on the other hand, was different. Because the distances to the walls around the robot were modest during teleoperation with conventional force feedback, the human operator reflected a substantial quantity of force feedback. These high force feedback values had a significant impact on the position of the control device, which was often unexpected by the human operator. This resulted in rather large change in the linear and angular velocities of the robot during the control robot and narrow space. Traversing a narrow path system allows the operator to finish the experiment with the same up-time as a proposed system. However, the proposed system has lower collision rates than the existing system because it only uses sensors in front of the robot. As a result, when the robot moves and changes direction in the narrow path, the robot's

sides and back collide with obstructions.

4.6.2 Entering Narrow Path and the Intersection between Narrow Pathway

In both experiments, the robot moved smoothly from the starting position to the goal position without colliding with the walls. Table 4.3 shows the comparisons between the OMPL module with the traditional system. The results show that the traditional system spends more time than the OMPL module at the narrow entrance path with widths of 0.50 m and 0.6 m. When the path width was 0.55 m with a traditional system, the operator spent more time controlling the robot into goal position than when the traditional narrow path width was 0.6 m. The operator controls the robot to face the same direction as the entrance area (to prevent the robot's side from colliding with the wall). The robot then moves into the narrow path. The time to finish the path is the same in the OMPL module because the distance between the start and goal positions is the same, and CoppeliaSim generates the robot's trajectory.

We created the intersection between the narrow paths (Figs. 4.31a and 4.31b). The operator desired the goal position and then started the simulation for teleoperation control in a similar way to entering the narrow path experiment. The results are shown in Figs. 4.31a and 4.31b. Alternatively, as in the previous experiment, the robot could move from the starting point in the narrow path to the goal position junction by junction while avoiding colliding with the wall. We experimented in two environments. In the first experiment, the robot moved to the right side of the intersection with a blending angle of 45° . In the second experiment, the robot moved to the left side of the intersection with a blending angle of 90° .

Table 4.3 shows the comparison between the teleoperation system when entering narrow space and the intersection of narrow space in a 2D camera system (traditional system). The operator could control the robot into the design position. Nevertheless, we found that in some experiments, the robot collided with the obstacle due to the limited perspective in the 2D camera + map information system, force feedback, and traversing system. The operator achieved the design position without colliding with the walls. However, it is high uptime when compared with the proposed system. In a 2D camera + map information system, the operator considers the distance between the robot and

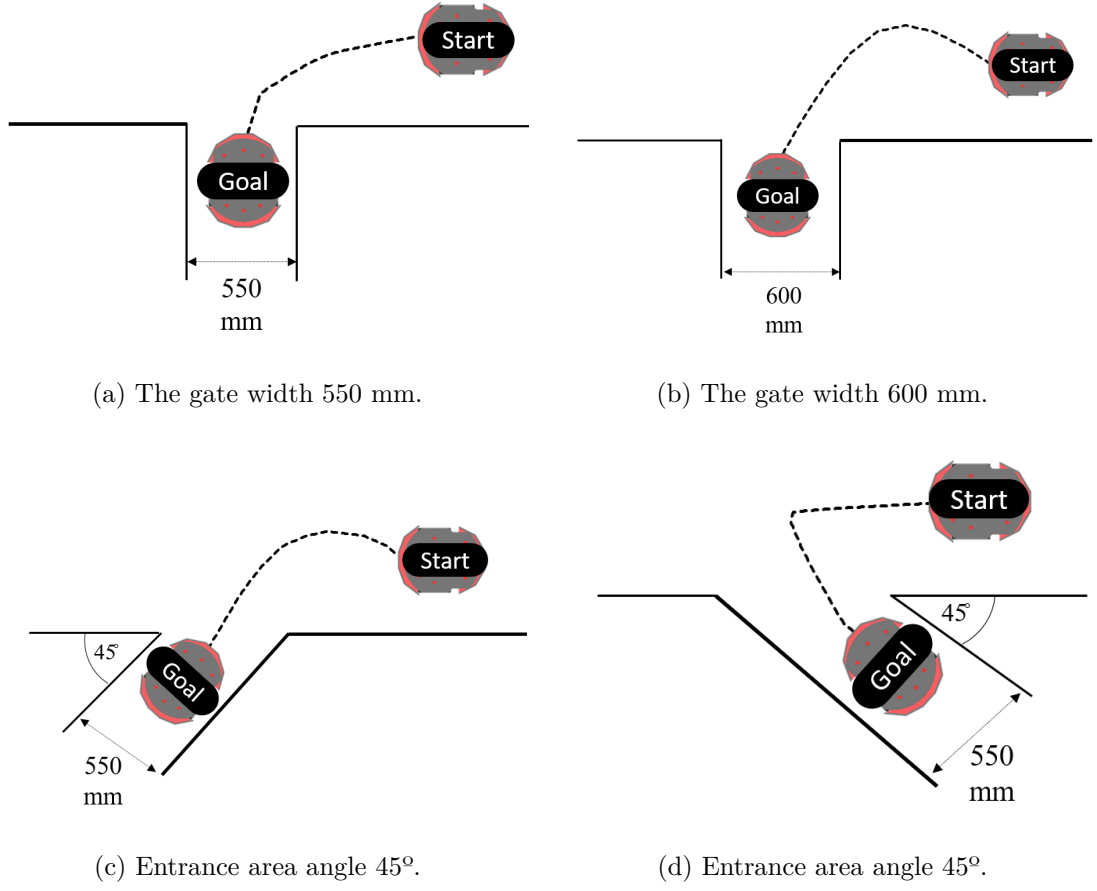
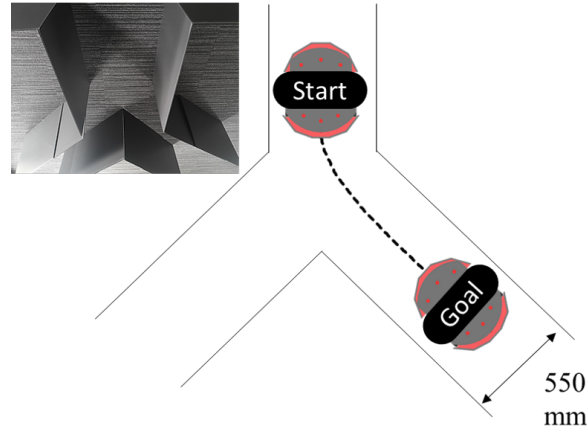


Figure 4.30: Entering and intersection narrow path environments experiment.

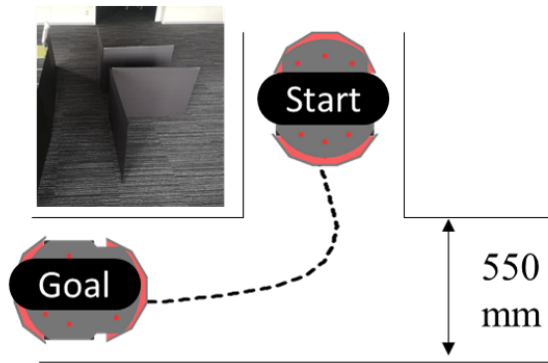
the obstacle before deciding to control the robot's movement. In traversing systems and force feedback systems, operators spend more time when compared with the proposed system because the distance between the robot and the obstacle is low, making the operator spend the time adjusting the robot.

Table 4.4 compares the operation time to finish a task between the traditional teleoperation system, the Braitenberg algorithm, the OMPL module, the traversing narrow path system, and the proposed system (with ten operators). Zhefu created the traversing narrow path system by developing an LRF detecting area algorithm to generate the robot's motion as it moves along the narrow path.

The results showed that the traditional system required a long time to complete the pathway and had many collisions. Operators found it challenging to control robot movements in the desired direction when encountering intersections and facing other problems. The robot moves along narrow paths during control when the narrow path is in the intended



(a) Intersection with the path blending 45° .



(b) Intersection with the path blending 90° .

Figure 4.31: Intersection Between Narrow path Experiment

direction. As a result of the operator's trying to change the robot's direction within a narrow path, the sides and back of the robot collide with impediments. The narrow path traversal system encountered a problem when the path changed 60 degrees. The robot's backside collided with the wall because it uses data from the LRF, which only receives data from the environment in front of the robot. We also encountered an issue at the second intersection where the robot was moving in the wrong direction because its movement depended on the LRF data. While the Braitenberg algorithm encountered a problem at the narrow entrance area, the operator must adjust the robot's direction in the same direction as the entrance. The Braitenberg algorithm automatically generates speed based on the sensor, causing the robot to move away from the entrance area (the sensor detects the wall at the entrance area). In addition, when the robot encountered intersections within narrow pathways, the operator spent additional time. It was found

that the operator was unable to direct the robot in the desired direction (the movement of the robot depends on the sensor readings), resulting in the robot's coming to a halt at the first and second junction. On the other hand, the operator had no issues while moving along a narrow path. The results of the OMPL module showed that the operator could control the robot without obstacles' collision or desired direction problems (as in the Braitenberg algorithm). The OMPL approach, on the other hand, took the longest time to complete the narrow path. The operator had to gradually set the robot's goal position according to the map updates (LRF); as a result, the operator had to repeat the same work, increasing the workload and decreasing environmental awareness. As a result of controlling the robot using the proposed system, it was found that the operator could direct the robot to the goal position in the least amount of time while avoiding obstacles' collision because an operator can choose different control methods according to the situation.

4.7 Chapter Summary

This chapter proposes a teleoperation system for assistive controlling the movement of a mobile robot in a narrow path. The teleoperation system combines data from the real world and virtual devices. We apply the Braitenberg algorithm and OMPL module to create the robot's movement along with a narrow path. The Braitenberg algorithm is an automatic motion to aid the robot operator maneuver through the narrow path. While the OMPL is used to create a path for the operator to control the robot entering narrow paths or intersections within a narrow path, the simulation uses a visual proximity sensor to fulfill the Braitenberg algorithm requirement. LRF obtains the environmental data and displays it on the simulation screen with dynamic simulation. The simulation scene is subjected to the visual proximity sensor, and the Braitenberg algorithm is applied to the simulation scene. Afterward, simulation publishes the linear and angular velocities to real-time robot control via ROS framework. The results showed that the system could combine real-time dynamic simulation with the real world. Furthermore, the proposed system could aid the operator in narrow path environments while avoiding the collision.

Table 4.3: Experimental results in entering narrow path and Intersection during the narrow path.

Environment	Teleoperation System	Time to finish (Sec / std)	Number of Collided (Avg / std)
Entering narrow path with path's width 550 m	2D camera (Only)	28.63 / 5.46	0.41 / 0.33
	2D camera + map information	20.11 / 2.26	0
	Force feedback	17.34 / 2.45	0
	Traversing a narrow path	23.41 / 4.53	0
	Proposed system	13.0 / 2.3	0
Entering narrow path with path's width 600 m	2D camera (Only)	33.6 / 8.31	0.33 / 0.3
	2D camera + map information	20.13 / 0.96	0
	Force feedback	16.93 / 2.54	0
	Traversing a narrow path	21.56 / 8.15	0
	Proposed system	13.4 / 2.13	0
Intersection between narrow path with 45	2D camera (Only)	29.08 / 9.66	0.48 / 0.2
	2D camera + map information	23.49 / 1.33	0
	Force feedback	24.67 / 5.63	0.13 / 0.15
	Traversing a narrow path	25.65 / 1.43	0.38 / 0.33
	Proposed system	18.0 / 1.26	0
Intersection between narrow path with 90	2D camera (Only)	41.01 / 4.36	1.3 / 0.33
	2D camera + map information	28.04 / 4.13	0.43 / 0.13
	Force feedback	31.67 / 3.25	0.33 / 0.14
	Traversing a narrow path	33.33 / 5.33	0.83 / 0.33
	Proposed system	18.65 / 1.33	0

Table 4.4: System comparison of the robot moves in a narrow path.

Teleoperation System	Time to finish(sec) (Avg/std)	Number of Collided (Avg/std)	Robot moves wrong way (Avg/std)
2D Camera (Traditional System)	170.90 / 6.90	9.2 / 1.23	0 / 0
2D Camera + map information	151.11 / 5.63	0.9 / 0.33	0 / 0
Traversing narrow path sytem[2]	99.10 / 6.54	2.2 / 1.03	0.4 / 0.52
Braitenberg algorithm	90.60 / 5.06	0 / 0	0.6 / 0.52
OMPL module	233.60 / 9.40	0 / 0	0 / 0
Proposed system (OMPL+Braitenberg)	66.40 / 4.00	0 / 0	0 / 0

CHAPTER 5

Small Object Displays in CoppeliaSim

To effectively control a robot from a distance, the operator must be aware of the environment around the robot so that the operator can give accurate instructions to the robot. Conventional teleoperation works by transmitting video feeds to a remote operator from a vehicle. Looking at the world through a video camera's "soda straw" decreases or leads to loss of peripheral vision. Once an object leaves the field of view, the operator must rely on memory and motion perception to assess an object's location. In addition, operators have limited ways of judging the relative size or the position concerning environmental elements of the vehicle. However, some context is possible if part of the vehicle is visible in the image. Small objects and slightly different surface steps make it difficult for the operator to perceive during teleoperation, especially when the camera's resolution is low or in environments with insufficient light. In case of image quality problems, it is difficult for the operator to understand the robot's condition and sense the surface on which the robot is moving.

This chapter begins with an introduction to the small object system architecture. Next, the proposed system algorithm for detecting small objects is described, then discuss displaying small objects in CoppeliaSim and test setup and end with the experiment results and discussion.

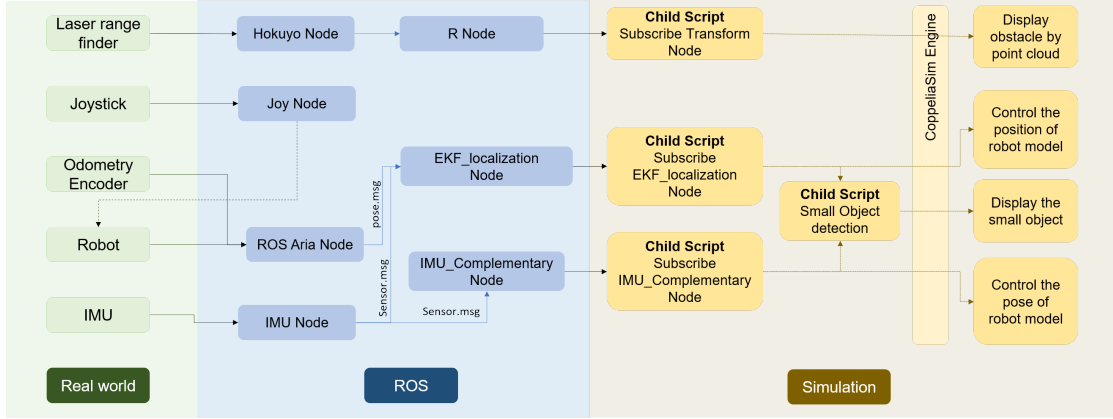


Figure 5.1: Proposed system.

5.1 Small Object Detection System Architecture

We have developed a teleoperation system that integrates the CoppeliaSim simulator and ROS, as shown in Fig. 5.1. We developed a child script in the CoppeliaSim simulator to estimate a robot's real-time position from the fusion data of odometry and IMU data through the EKF_localization node. The EKF_localization node implements an extended Kalman filter(EKF) to reduce the odometry encoder(ODE) error. The EKF_localization node is published to estimate real-time robot position. Therefore, the CoppeliaSim simulator subscribed EKF_localization node to determine the robot's current position based on a base link frame. In the CoppeliaSim simulator, the robot model moves based on the data from the EKF_localization node. The robot's position is moved on the basis of the starting point frame, whereas the Coppelia simulator also uses the starting point frame based on the base link frame. The results indicated that in the simulation, the robot model could show the robot movement similar to the actual robot movement in the real world, as shown in Fig. 5.2.

Another child script is used to obtain quaternion data from the IMU sensor to display the real-time robot pose. The IMU sensor is connected to a computer through a USB port. The ROS system between the IMU sensor and the CoppeliaSim simulator displays the real-time robot pose. The IMU node obtains information from the IMU sensor in the ROS system. The ROS system publishes angular information that includes angular velocity, linear velocity, and magnetic field. This study used the `ros/complementary` package to transform the original data from an IMU sensor into a quaternion coordinate system to represent the robot model pose. Fig. 5.3 shows the block diagram

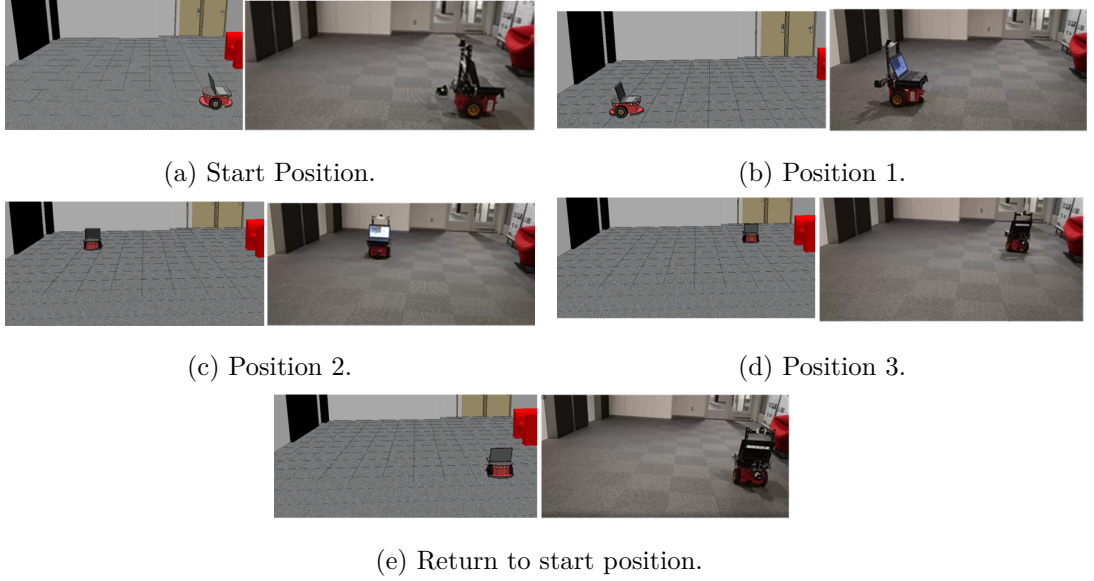


Figure 5.2: Coppeliasim simulator displays real-time robot position

for transferring the original IMU data into the quaternion coordinate system. Hence, ${}^L\omega_t$ is the angular vector arranged as pure quaternion at the time instant, Lm_t is a local magnetic field at time instant, q_{mag} represents a rotation around the z-axis, q_{acc} rotates a vector from the sensor frame to the horizontal plane of the global frame, ${}^Lgq_{t-1}$ is the previous estimate of the orientation, $\Delta\hat{q}_{acc}$ is computed data from the accelerometer, $\Delta\hat{q}_{mag}$ is reading data from a magnetic field, ${}^L_Gq_\omega$ predicted quaternion, La_t is a normalized body frame gravity vector obtained from the accelerometer, L_gq is the orientation quaternion of the global frame(G) relative to the local frame(L). Therefore, we can display the real-time robot pose with Coppeliasim as shown in Fig. 5.4 where errors from ros/complementary package in roll pitch and yaw are 0.0267, 0.0211, and 0.1876 rad, respectively. The ROS Hokuyo node is used to connect the real Hokuyo LRF sensor with the simulator. The Hokuyo LRF was performed at 180° with a rate of 512 samples per reading. Generally, the ROS_Hokuyo node publishes the scan data from the real Hokuyo sensor into the ROS with 2D data. However, Coppeliasim is not suitable for displaying 2D sensor data. Therefore, the node is used to transform the 2D scan data message into a point cloud system to display the scanning data (obstacle data) into Coppeliasim. The transforms node subscribes to sensor_msgs/LaserScan messages (from the Hokuyo sensor) on the scan topic. These scan data are processed by the projector and transformer, which project the scan data into Cartesian space and then transform into the fixed_frame. This results in a sensor_msgs/PointCloud that

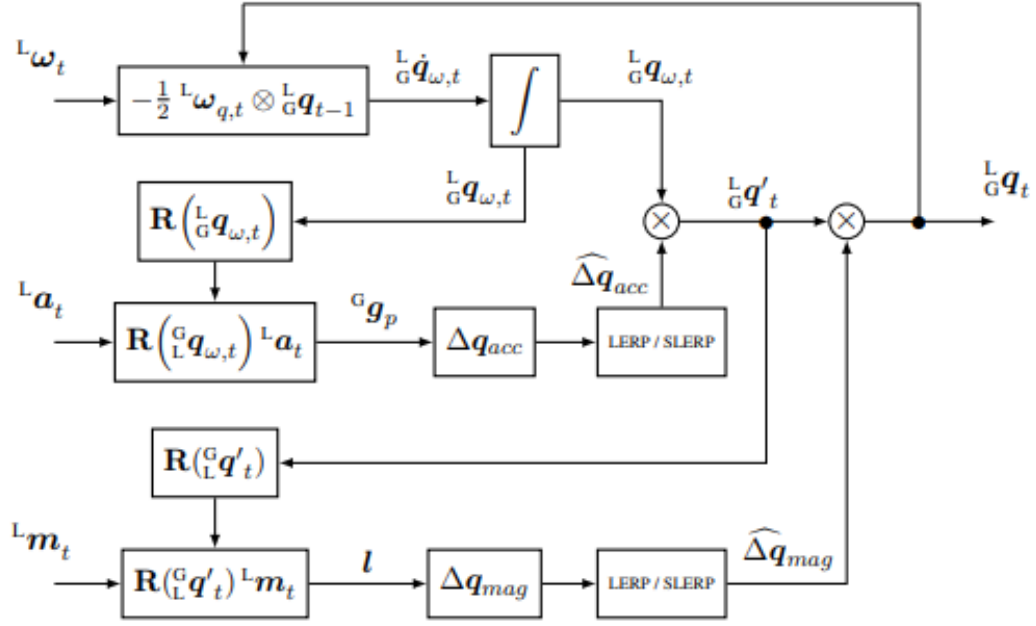
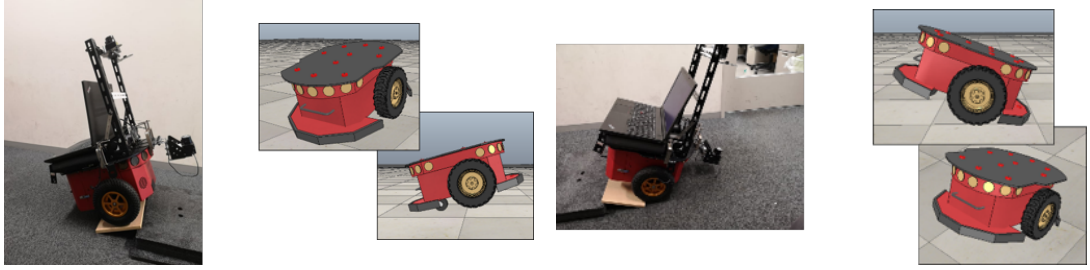


Figure 5.3: Block diagram of the complementary filter[75]



(a) Move up

(b) Move down

Figure 5.4: Real-time robot pose in CoppeliaSim scene.

can be added to the rolling buffer. Clouds in the rolling buffer are then assembled on service calls. The results are shown in Fig. 5.5. The ROS-Joy node is connected to a joystick is used to control the robot.

5.2 Small Object Displays in CoppeliaSim

CoppeliaSim simulator uses IMU sensor data to display obstacles (small objects). We used this algorithm based on the quaternion-based complementary filter to obtain quaternion data from IMU devices, developing an algorithm by applying a CoppeliaSim function and visualizing obstacles (small objects) in CoppeliaSim using dummy point. The

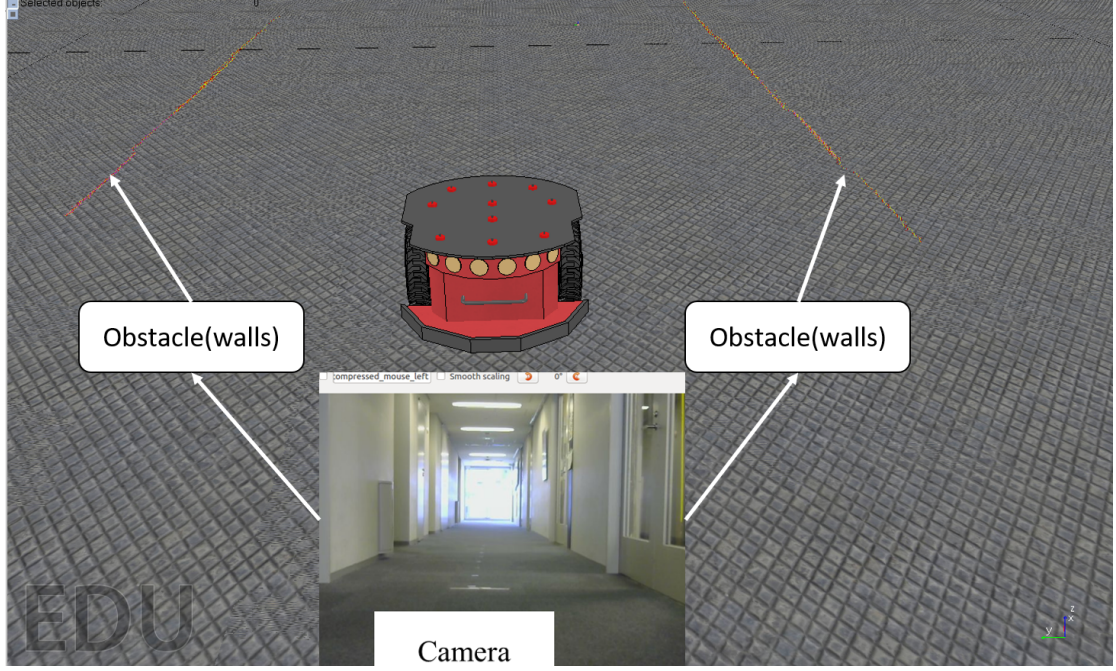


Figure 5.5: Teleoperation display in CoppeliaSim.

dummy points (W_C , W_R , and W_L), are shown in Fig. 5.6. The dummy point W_C is located in front of the robot, whereas the dummy points W_R and W_L are contact points between the floor and left and right wheels, respectively. Fig. 5.7 shows different scenarios when the robot encounters an obstacle or moves on a rough surface. We will find that when the left side of the robot moves over a small obstacle, the robot will have a left tilt (see Fig 5.7a) and the location of the dummy W_L will be higher than the reference frame. Conversely, when the right wheel of the robot moves on an obstacle, the dummy is right-tilted (Fig 5.7b), and the dummy W_R and W_C are higher than the reference frame. Thus, we derive the relationship between each dummy in each scenario as shown in Fig 5.8. Fig. 5.9 illustrates the algorithm used in the CoppeliaSim simulator to display the robot posture when it encounters obstacles or moves on rough surfaces. At the start of the teleoperation system, the CoppeliaSim simulator obtains the current robot posture and starting positions of the dummy points W_C , W_L , and W_R . If the robot encounters an obstacle or moves on rough surface conditions, the position of W_C , W_L , and W_R will be changed, as shown in Figs. 5.7 and Fig. 5.8. Therefore, we use this relationship to calculate the difference between the starting positions of the dummy points W_C , W_L , and W_R and their current positions to obtain difference values D_c , D_r , and D_l , respectively (see Fig 5.10a).

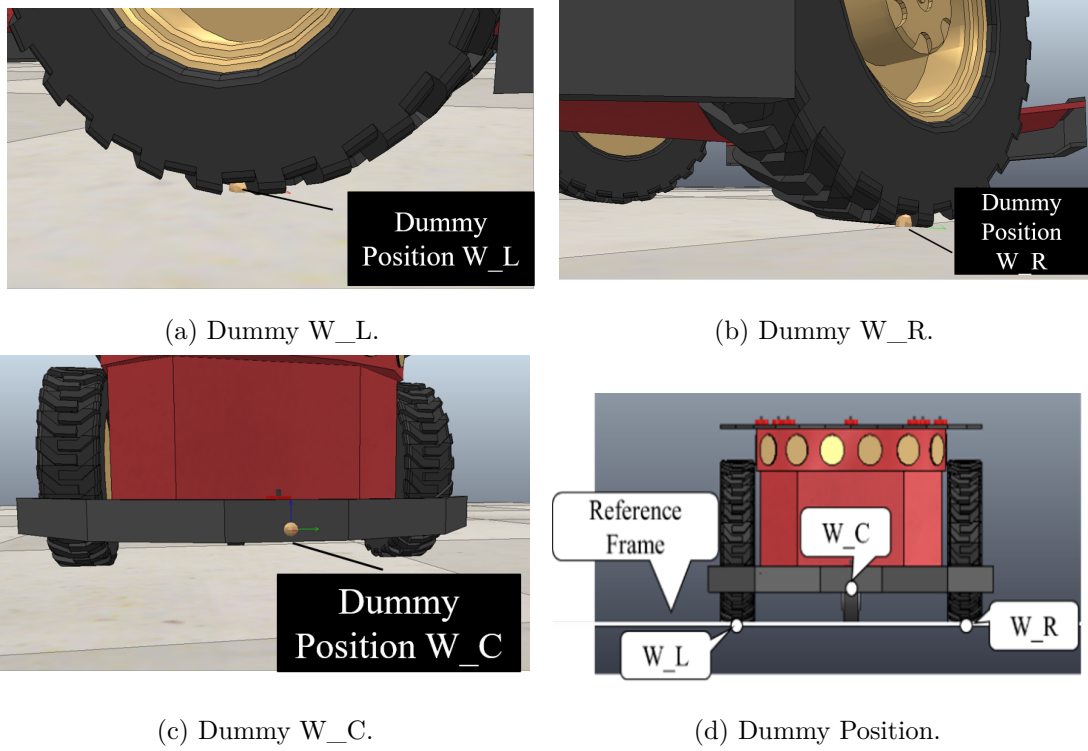
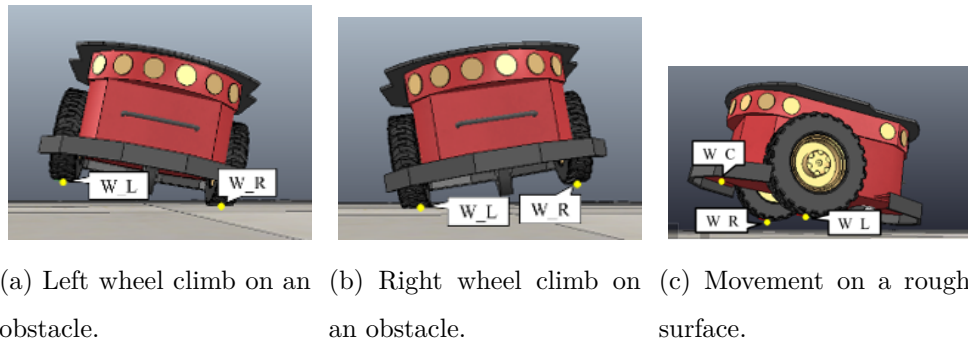


Figure 5.6: Created dummy position in each robot wheel and front of robot wheel.



(a) Left wheel climb on an obstacle. (b) Right wheel climb on an obstacle. (c) Movement on a rough surface.

Figure 5.7: Displayed the robot pose and dummy position in each scenario.

In this algorithm, if the difference values D_c and D_r are greater than the reference frame, the robot's right wheel has encountered an obstacle (small object). The CoppeliaSim simulator displays a blue box at the dummy point W_R position, and the height of the blue box is the same as the D_r value. This blue box represents the small object (obstacle) that the robot encountered. Otherwise, if the difference values D_l and D_c are greater than zero and the difference value D_r is zero or less than zero, the robot's left wheel has encountered an obstacle (small object). The CoppeliaSim simulator displays a blue box at the dummy point W_L position, with the height of the blue box equal to the D_l value. Furthermore, if the difference values D_l and D_r are greater than zero, the robot's right and left wheels have both encountered an obstacle (small object), or the robot is moving on a rough floor. We use LUA API to create and display floor conditions in the CoppeliaSim simulator. We create small cuboids to represent small objects (obstacles) or rough surface conditions. The child script measures and calculates the D_r , D_l , and D_c values using the algorithm as shown in Fig. 5.9. If the algorithm indicates that the robot's left or right wheels encounters an obstacle, the embedded script will call a function to create the obstacle as small cuboids. The size of the small cuboid is the same as the size of the robot wheel. If the algorithm determines that the robot moves on different floor levels, the embedded script will call a function to create obstacles as small cuboids.

When the robot posture indicates that the actual robot is climbing small objects or unsmooth surface conditions, we develop the function for creating obstacles represented by small cuboids. The LUA API in CoppeliaSim obtains the current dummy positions $PDL_{x\ y\ z}$ and $PDR_{x\ y\ z}$. Hence, PDL_x is a dummy position of dummy W_L in the x-axis. PDL_y is dummy position of dummy W_L in the y-axis, and PDL_z is dummy position of dummy W_L in Z-axis. While PDR_x is dummy position of dummy W_R in the x-axis. PDR_y is dummy position of dummy W_R in the y-axis, and PDR_z is dummy position of dummy W_R in Z-axis. The LUA API command is used to generate the small cuboids at dummy positions. The size of a small cuboid is $0.01 \times 2D \times D_l$ in the left wheel, the position of the small cuboid is $PDR_{x,y,z}$ and $0.01 \times 2D \times D_r$ in the right wheel, and the position of the small cuboid is $PDR_{x,y,z}$ as shown in Fig.5.10b.

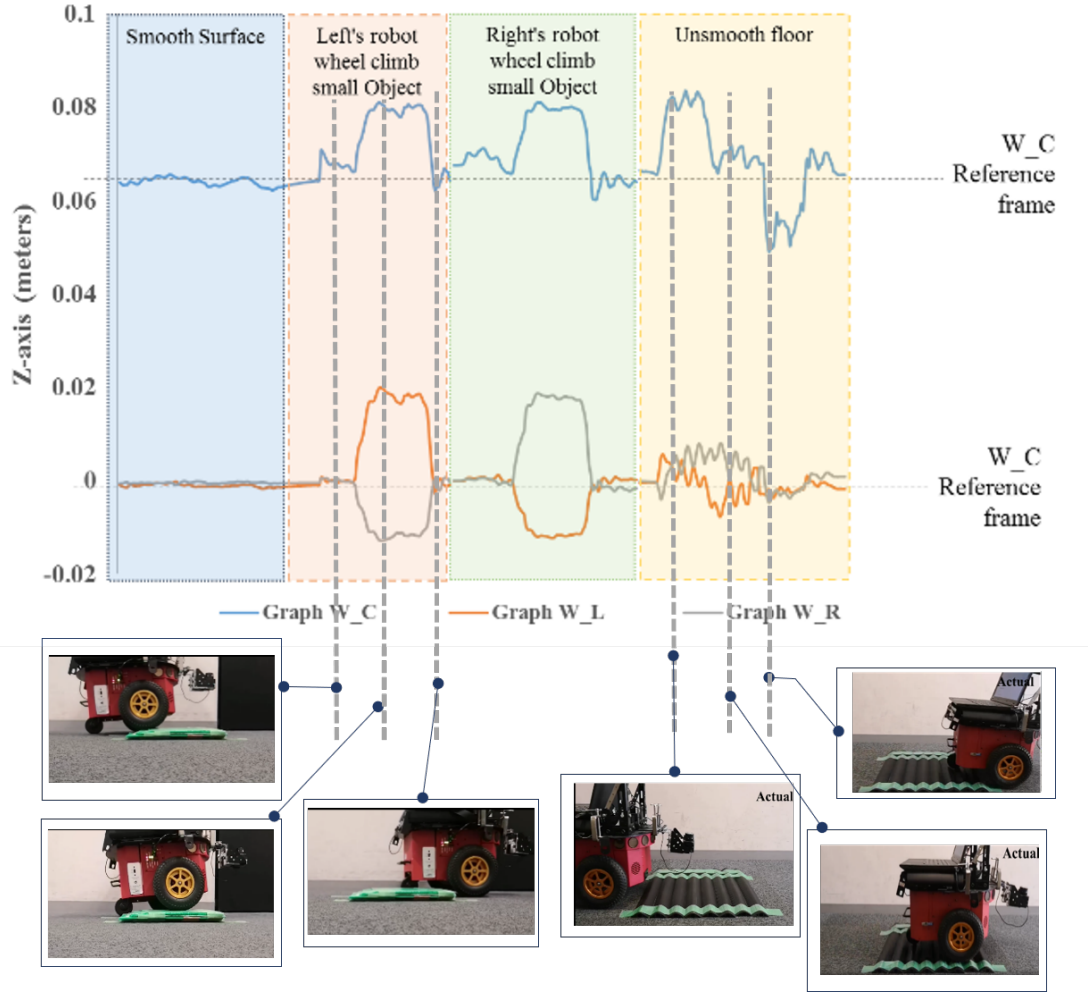


Figure 5.8: Relationship between the actual robot posture and dummy points W_C , W_L , and W_R .

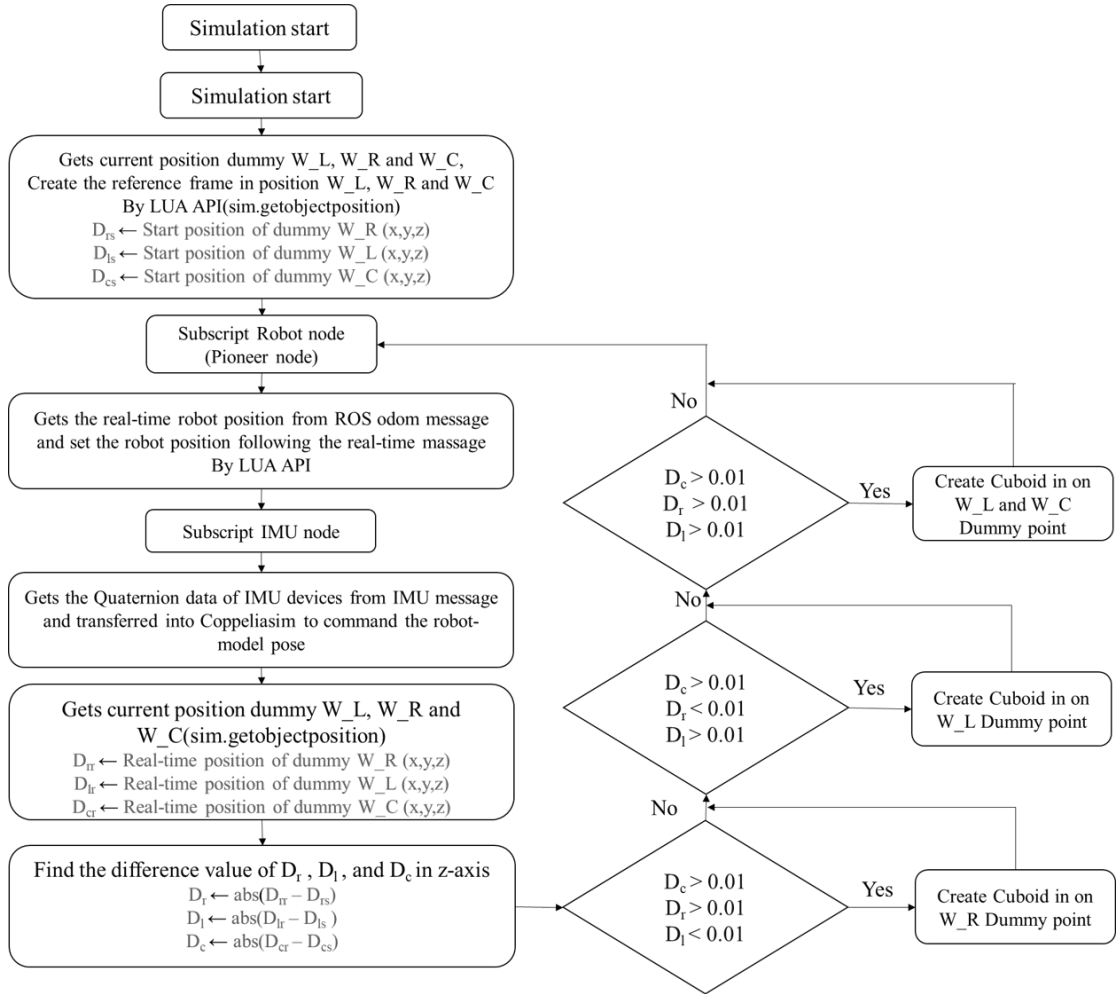
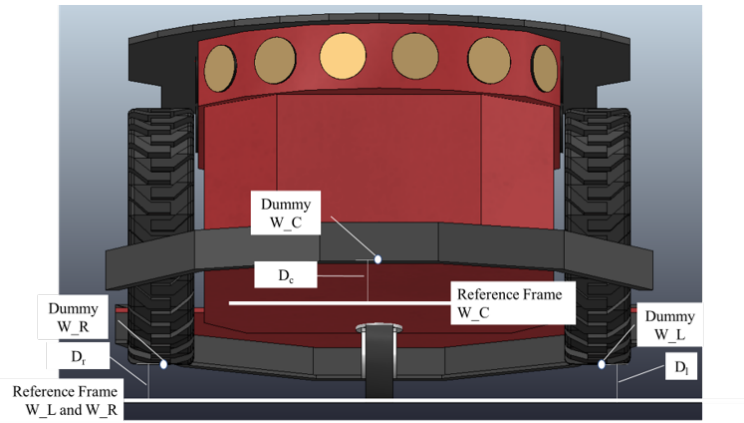
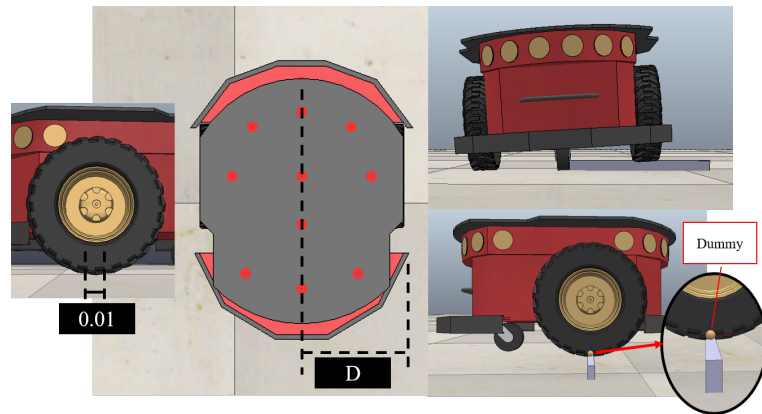


Figure 5.9: Algorithm for robot posture detection.

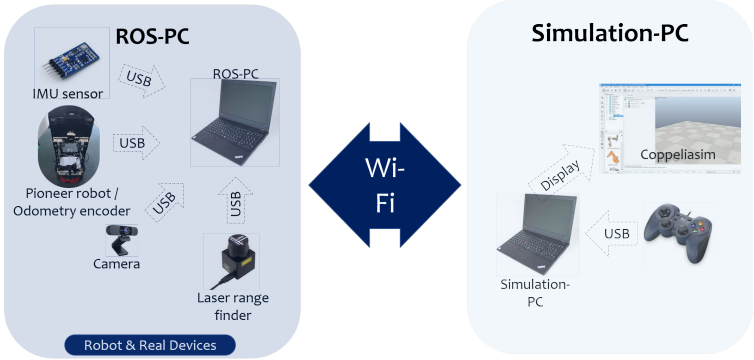


(a) Display the relationship between Dummy point W_C, W_L, and W_R with value D_l, D_r and D_c .

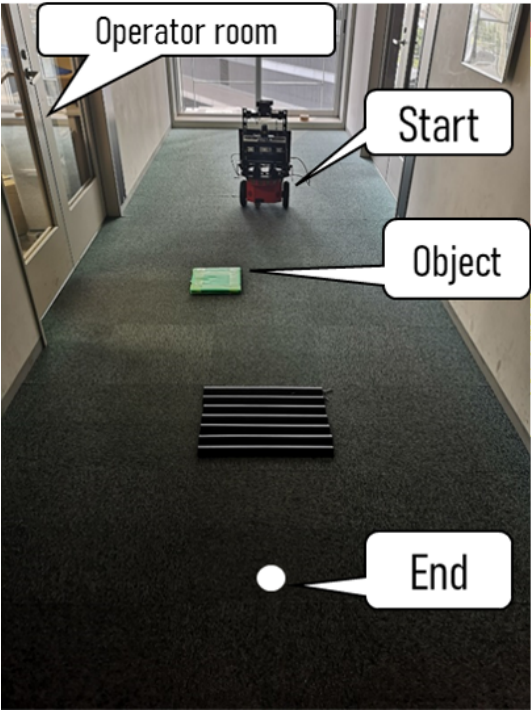


(b) Create cuboid for represent the small object.

Figure 5.10: CoppeliaSim displays small object.



(a) Hardware.



(b) Experiment environment.

The screenshot shows the 'Physics Engines Properties - Material' window in Coppeliasim. It displays a table of properties for a material, with 'Bullet properties' and 'ODE properties' sections.

Property	Value
Apply predefined settings:	None
Bullet properties	
Friction (only Bullet V2.78)	1.0000e+00
Friction (after Bullet V2.78)	1.0000e+00
Restitution	0.0000e+00
Linear damping	0.0000e+00
Angular damping	0.0000e+00
Sticky contact (only Bullet V2.78)	<input type="checkbox"/> False
Auto-shrink convex mesh	<input type="checkbox"/> False
Custom collision margin	<input type="checkbox"/> False
Custom collision margin factor	1.0000e-01
ODE properties	
Friction	1.0000e+00
Maximum contacts	64
Soft ERP	2.0000e-01
Soft CFM	0.0000e+00
Linear damping	0.0000e+00
Angular damping	0.0000e+00

(c) Friction setting scene in Coppeliasim.

Figure 5.11: Experiment setting

5.3 Experiment Setting

To investigate the concept of small object detection and integrated real-time dynamic simulation, this study created a small object, difference floor level and unsmooth surface to investigate the system and create a the white box mounted on top of the robot; this represent the object that the robot carries. The teleoperation system is shown in Fig. 5.12a. Pioneer robots are equipped with ROS-PC to establish communication between the robot and IMU sensor, camera and laser rangefinder to obtain environmental information. ROS-PC connects the teleoperation-PC with the same ROS master. In addition, the teleoperation-PC connects with the joystick to obtain the operator's command to control the robot. The teleoperation-PC runs CoppeliaSim to receive data for simulation. Fig.5.12b depicts the experiment environment, along with the operator controls room. The operator remote control the robot to move forward and climb an object in different scenario(the robot's left or right wheel climb on a small object, robot move in unsmooth surface and robot move in different floor level). With environmental information displayed by the CoppeliaSim simulator, the operator is made aware of the environment conditions.

We create a realistic object that the robot can performs on a CoppeliaSim model. Fig. 5.12 shows the comparison between the actual and simulation model, using a notebook PC and a white box. Set the model to the same conditions as actual conditions. CoppeliaSim comes with the defaults setting friction coefficient, operators could selected using predefined setting function(Fig. 5.11c); The defaults friction coefficient is 0.71, In addition, an object's weight can be set before simulation in `rigid_body_dynamic_properties` function in CoppeliaSim. The friction between the robot and the white box is the default value obtained by CoppeliaSim, which defined the white box model weight as 0.0283 kg. The white box represents the object that the robot carries on. As in the experiment, the operator controls the robot's movement over a small object, unsmooth surface and at different floor level conditions. We measured the real white box movement by attaching the IMU sensor to the small white box(Fig. 5.12c) and using the `IMU_complementary_filter` node to obtain the movement of real small objects. We compared real-time white-box movements to real-time white-box models and measured the difference in their final positions. The reference frame of the simulation setup is the same as the actual coordinates provided by the IMU. The

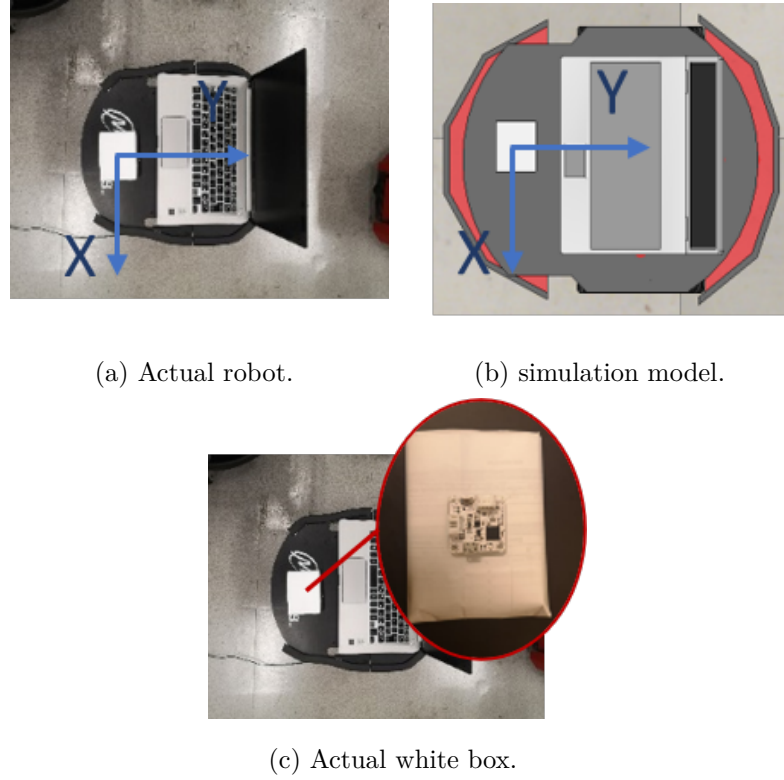


Figure 5.12: Comparison between real-robot setting and robot-model setting.

+y-axis is represents the robot's forward movement, as shown in Fig. 5.12c.

5.4 Small Object Detection Experiment

5.4.1 Left or Right Robot Wheel Climb on a Small Object

In the obstacle detection experiment, we create an obstacle with a 35 cm length, and height is 28 centimeters. In the experiment, a teleoperator remote control is used to move the robot to climb the obstacle. The robot starts moving on a smooth surface, and then its left wheel position climbs the obstacle. The experimental results are shown in Fig. 5.13. The Coppeliasim simulator creates an obstacle when the robot's left wheel climbs the obstacle. Fig. 5.13a shows a simulator creating a small cuboid in the left wheel position when the robot climbs the obstacle. Next, as shown in Fig. 5.13b, the Coppeliasim simulator continues to create the small cuboids as the actual robot's left wheel climbs over the obstacle. Finally, as shown in Fig. 5.13c, once the robot has moved over the obstacle, the Coppeliasim simulator also stops creating a small cuboid as in

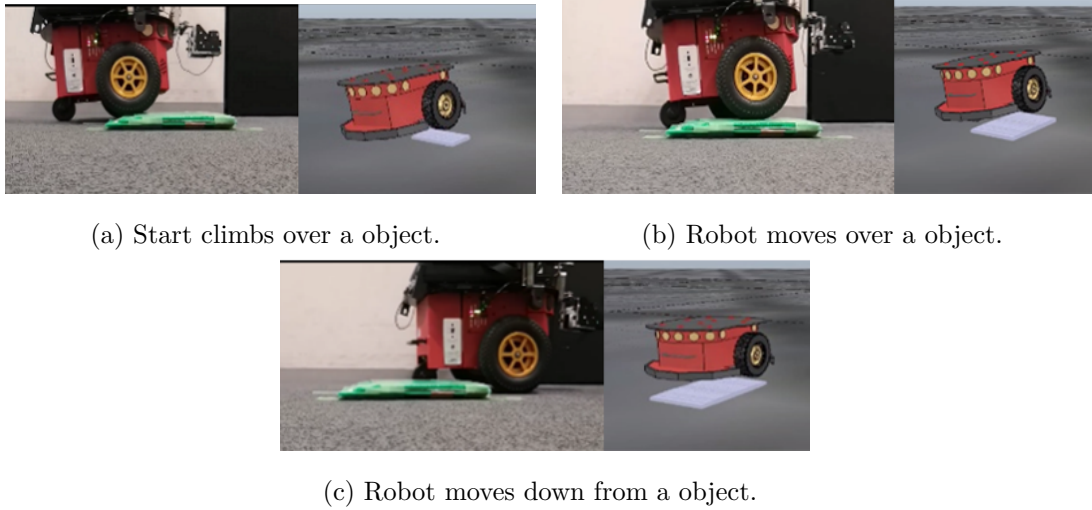
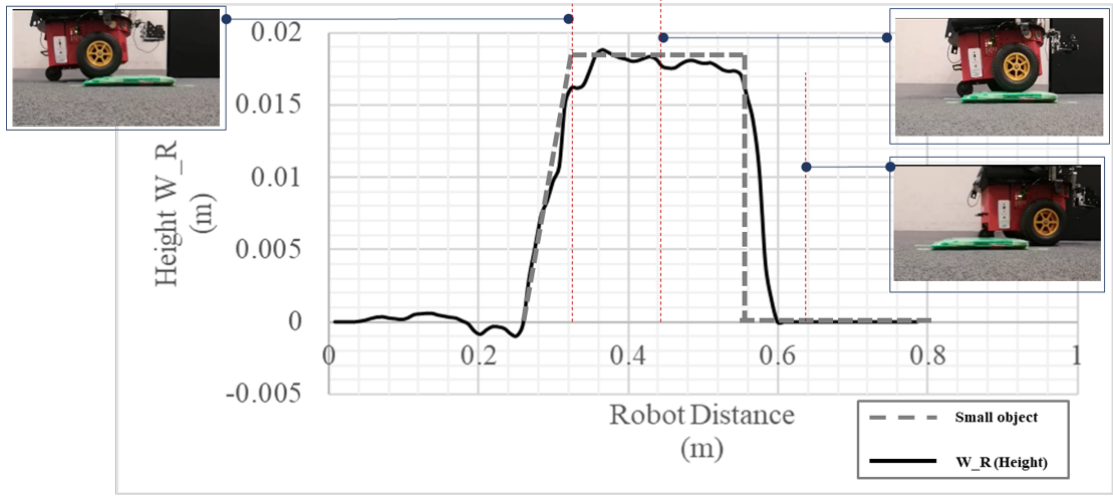


Figure 5.13: The CoppeliaSim displays the small object when the actual robot climbs the small object.

the actual situation.

Fig. 5.14 shows the D_r values or small object height data with the robot's distance. The D_r value point starts to change position at 0.3 m. The slope of the graph increases until the robot's distance is 0.36 m. The peak of the robot's height is 10.8 mm. In this phase, represents that the robot is starting to climb the small object, as shown in Fig. 5.13a. Afterward, the left wheel continues to move over small objects, resulting into no significant change in the slope. The D_r starts to decrease when the robot's distance is 0.53 m, and the height of the robot becomes zero. When the robot's distance is 0.6 m, the robot moves on a flat surface, as shown in Fig. 5.13c. The results of the robot's right wheel are the same as those of the robot's left wheel, and the CoppeliaSim simulator can display the small object when the robot's right wheel is climbing over the small object. The D_r value represents the small object height, and it is compared to the height of the actual object that has similar characteristics. The incremental characteristics of D_r were close to the slope of the actual object. D_r reaches the maximum value, and the D_r value is slightly less than the actual object height and not constant due to the fact the actual object is not a rigid body; it can be compressed by the weight of the robot. As a result, the maximum altitude of D_r is less than the actual value and not constant. At the last position, the actual robot rapidly lower its height to zero. However, D_r decreases with the slope until it reaches zero because the wheel of the robot is large and the robot moves quickly, as shown in Figs 5.15b and 5.15c. The effect of the slope on the D_r value

Figure 5.14: D_r value during robot move on small object.

cannot represent an object that rapidly decreases height.

5.4.2 Robot Moves on Unsmooth Surface Experiment

We created the unsmooth floor as in the small object detection experiment. The operator remotely controls the robot over the flat floor through unsmooth floor conditions. The operator observes data from the camera devices attached to the front of the mobile robot when the robot moves over the unsmooth floor, making it difficult for the operator to understand the current floor condition. The CoppeliaSim simulator displays the unsmooth floor condition, as shown in Fig. 5.16. In Fig. 5.16a, at the start point, the actual robot moves over the flat floor. In Fig. 5.16b, the actual robot starts to climb the unsmooth floor, CoppeliaSim simulator starts to display the unsmooth floor and continues to display the floor condition as the actual robot moves over the unsmooth floor, as shown in Fig. 5.16c. In Fig. 5.16d, the CoppeliaSim simulator stops displaying the unsmooth floor condition when the actual robot moves over a flat floor. In this experiment, we created different floor levels, as shown in Fig 5.17. The results show that the CoppeliaSim simulator displays the floor difference level as in the experiment. The average length of the unsmooth surface shown in the simulation was 0.61 ± 0.035 m (in 10 experiments). However, the actual length of the unsmooth surface is 0.58 m.

Fig. 5.18c shows the height and length of a small object displayed by the CoppeliaSim simulator. Fig. 5.18a shows the small object displayed by the CoppeliaSim simulator, the same as the actual object. The small object model is 19 x 4.13 x 4.46 cm whereas the

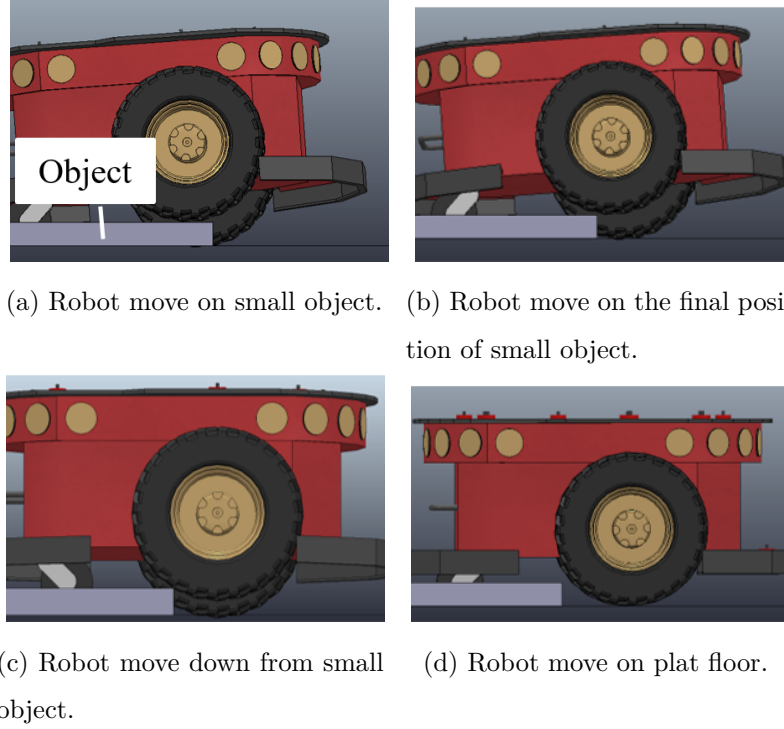
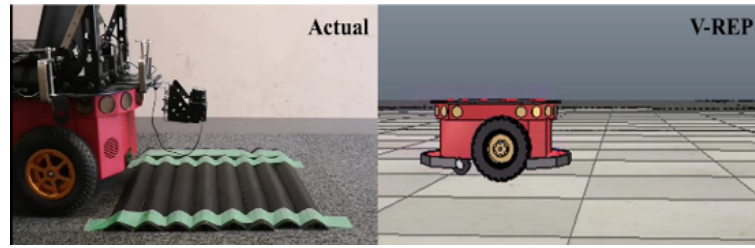


Figure 5.15: Robot move down from small object.

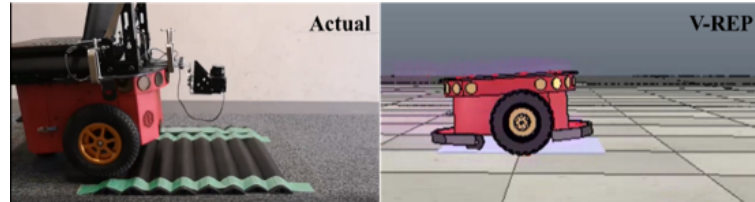
actual model is 29 x 4 x 4.5 cm. However, the width of the small object is preset before starting the system because the limitation of the system cannot perceive the width of the small object as shown in Fig. 5.18d. CoppeliaSim shows an object width larger than the object's actual size; the small object model is 19 cm, whereas the actual model is 25 cm (small model objects depending on the width of the robot). In addition, the object's preset type is cuboid, so CoppeliaSim shows small objects in a cuboid form. In contrast, the real object is cylindrical (Fig. 5.18b) but CoppeliaSim displays the height and length of the small object similar to the actual value (the height and length of small objects are 5.11 x 4.98 cm whereas the actual size is 5 x 5 cm). CoppeliaSim can display objects with curved surfaces if the curvature is sufficient, as shown in Fig. 5.18d. However, the system can display a real-time robot pose and the visual small in CoppeliaSim, improving operators' understanding of the robot's characteristics and surface environment.

5.4.3 Real-Time Dynamic Simulation

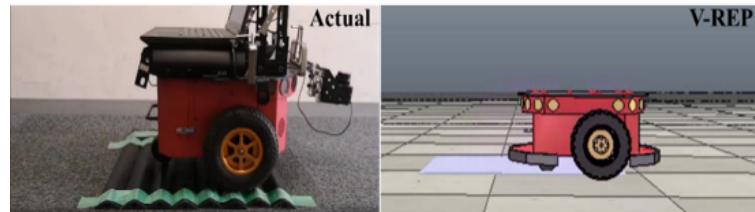
In general, in teleoperation systems, a camera is mounted on the robot to obtain environmental information in the front/around the robot from which the operator understands



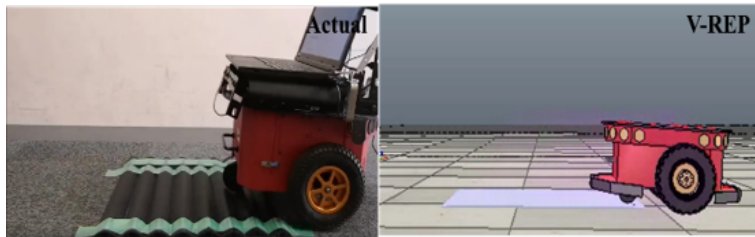
(a) Robot moves on the flat floor



(b) Robot begins over move on unsmooth floor.

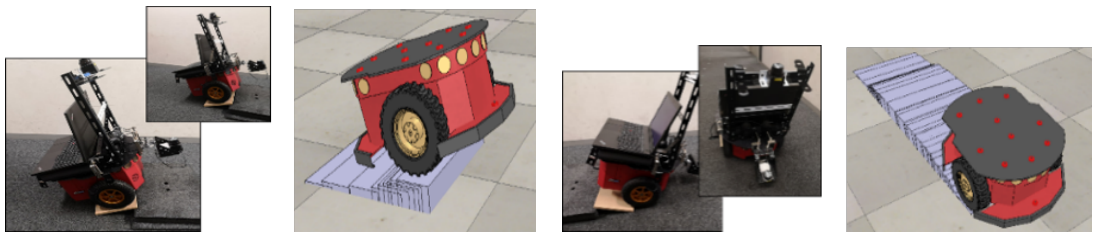


(c) Robot moves over unsmooth floor.



(d) Robot move down into flat floor.

Figure 5.16: CoppeliaSim displays the unsmooth floor conditions.



(a) Starts climbing the sloped floor.

(b) Robot moves down from slope floor.

Figure 5.17: CoppeliaSim displays the difference level of the floor conditions.

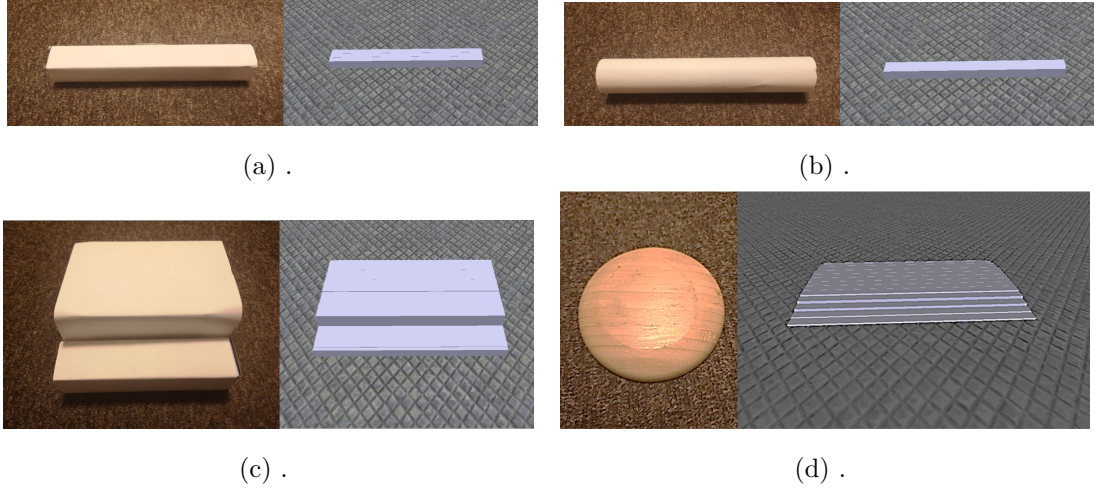
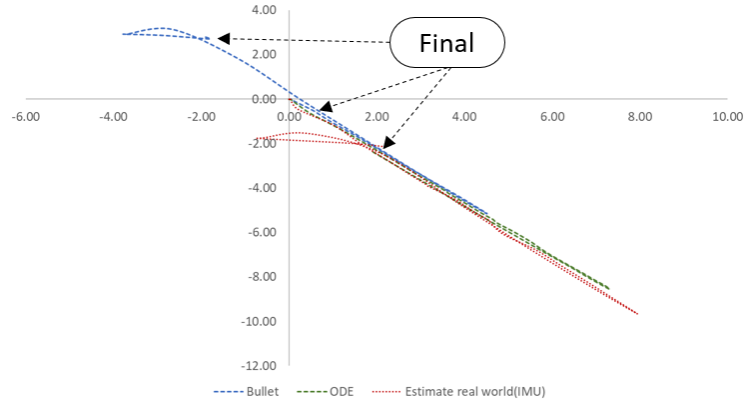


Figure 5.18: Small objects displayed in CoppeliaSim.

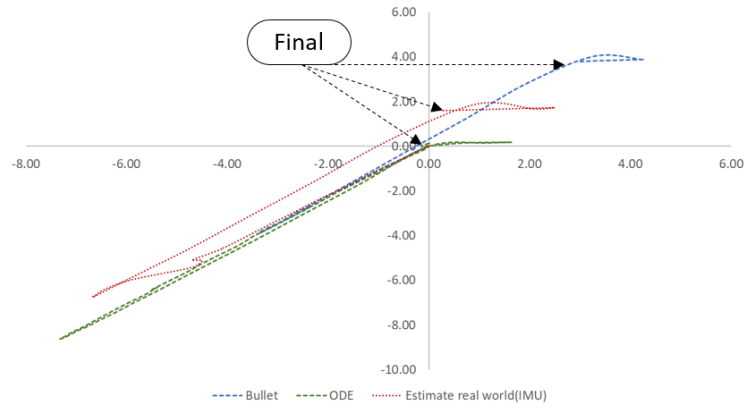
the environment to control the robot's movements. Thus, the camera has a limited field of the view and cannot obtain the robot condition preventing an operator from sensing the robot during the teleoperation process. In some situations, the robot is required to carry the object; for example, a survey robot with a robotic arm to collect samples in unknown environments. Therefore, it is difficult for operators to understand the object's state when the robot moves on unsmooth surfaces. This experiment combines real-time dynamic simulation and proposes a system to assess object posture during remote operations. The proposed system uses a real-time robot to control the robotic model in CoppeliaSim.

Dynamic Simulation when the Robot's Left/Right Wheels Climbs a Small Object

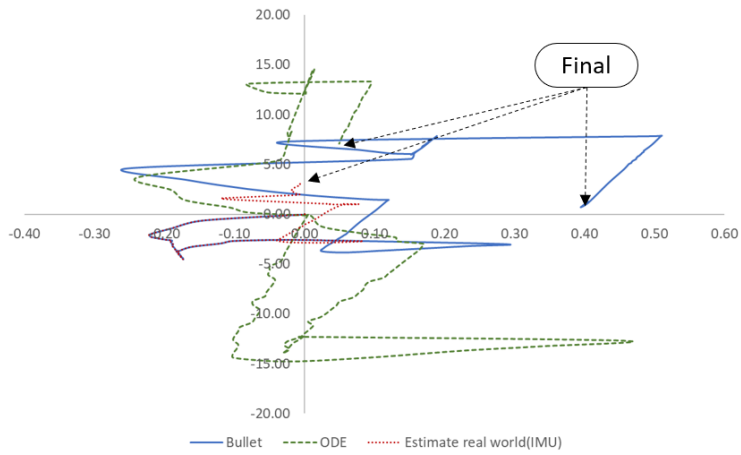
Fig. 5.19 shows the comparison results of the actual white box movement with the simulation movement. The real white box movement is obtained from the IMU sensor attached to the actual white box. When the robot's left or right wheel climbs a small object, the characteristics of the white box movement between the actual and the simulation movement are the same. When the robot's left wheel climbs a small object, as shown in Fig. 5.19a, the white box begins to move down in the $-x$ and $-y$ axes at the start of the climb and continues to move down slightly when the robot's left wheel moves over the small object. The small object rushes into $-x$ and $+y$ axes when the robot's left wheel moves down from the small object. The simulation results showed that the



(a) White box movement when the left wheel climbs on a small object (cm).



(b) White box movement when the right wheel climbs on a small object (cm).



(c) White box movement when the robot moves on difference level of the floor (cm).

Figure 5.19: Comparison of white box movement between estimate real-world(IMU) and CoppeliaSim.

movement of the white model in the simulation program with Bullet and ODE was the same as that of the actual white box movement. The error at the last position of the small white box between actual movement is shown in table 5.1. The results of the ODE physics engine are more closely related to the approximate actual white box location obtained from the IMU sensor than the Bullet's physical engine. When the right wheel climbs a small object (Fig. 5.19b), the simulation could simulate the trajectory of the white box model with the same characteristic as the actual white box movement. The white box starts to move down in the $+x$ and $-y$ axes when the right wheel moves up on the small object and continues to move down slightly as the right wheel moves down on the small object. Then the small white box rapidly moves into the $-x$ and $+y$ axes when the right wheel moves down from the small object. The error at the last position of the small white box between the simulation and actual movements is shown in Table 5.1. The result is that the ODE physics engine is more closely related to the approximate actual white box location obtained from the IMU sensor than the Bullet physical engine, which is moving in the same direction as the experimental robot's left wheel on the object.

Dynamic Simulation when the Robot Moves on Different Floor Levels

When the robot moved on different floor levels and unsmooth surface conditions, there was a difference in the trajectory of the white box movement between the simulation movement and the movement of the real object. At different floor levels, the Bullet had an extensive trajectory error on the x -axis, whereas ODE shows the big trajectory difference on the y -axis (Fig. 5.19c). However, the error of the last position in unsmooth surface conditions is less than left and right wheel climb on the small object conditions. The error at the last position of the small white box between actual shown in table 5.1. The error when the left or right wheel climbs on the small object is greater than the difference floor level and unsmooth floor condition as shown in Fig. 5.20. We compare the data from the IMU sensor to the simulation results by matching the simulation time step with the ROS time step. In the experiment, the right or left wheel of the robot is constantly moving on a small object, causing the robot to tilt. As a result, the white box is continuously moving according to the slope of the robot (accumulating error). In the experiment, when the robot moves to a different floor level, it found that initially, the error in the y -axis is initially high (Figs. 5.20b and 5.20d). Because the robot is moving

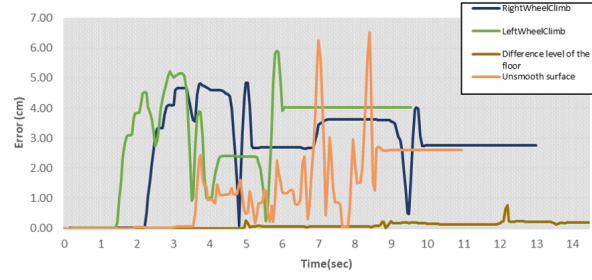
Table 5.1: The difference at a final position between actual movement with simulation results.

Conditions	Bullet		ODE	
	x-axis	y-axis	x-axis	y-axis
Left's wheel climb on small object	4.02 ± 1.15	4.90 ± 1.31	0.21 ± 0.13	0.24 ± 0.09
Right wheel climb on small object	2.75 ± 0.95	2.21 ± 0.83	0.26 ± 0.13	1.46 ± 0.078
Robot moves in difference level of the floors	1.49 ± 0.05	1.49 ± 1.18	0.16 ± 0.09	1.18 ± 0.1
Robot moves in unsmooth surface floors	2.59 ± 0.1	0.03 ± 0.1	1.55 ± 0.13	0.07 ± 0.13

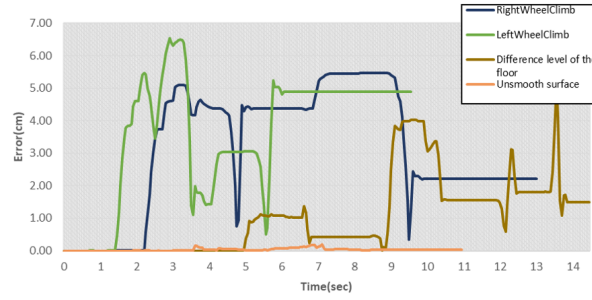
up a steep slope and the error remains constant as it moves on a different floor level, the tolerance will rise again as it descends from the slope. In comparison, the error in the x-axis is small because the robot is moving in a straight line (as shown in the robot's motion in Fig. 5.17). In the unsmooth surface condition, we found a slight difference between actual and simulation movements because the height of the unsmooth surface is low. This prevents the robot from tilting at high altitudes, so the movement of the white box in the y-axis is low. Most of the time, the white box moves along x-axis according to the robot's tilt while moving on an unsmooth surface, resulting in the tolerance in the x-axis being higher than that of y-axis.

5.5 Chapter Summary

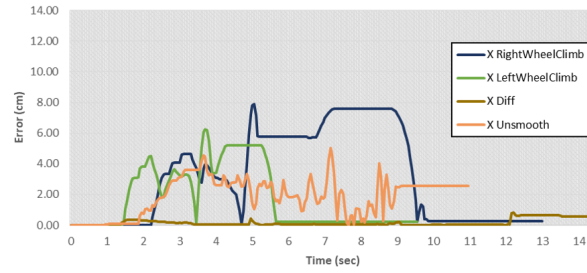
This chapter proposes a teleoperation system for detecting small objects and surface conditions to improve environmental awareness in areas with low communication signals. It proposes a teleoperation system that integrates a CoppeliaSim robotics simulator with the real world via the ROS framework. The proposed system is a real-time display system that can provide small object and floor conditions then visualization in simulation. A fusion of inertial measurement unit sensor and odometry data will be sent to the simulator to display the posture of a robot. CoppeliaSim constructs an algorithm to



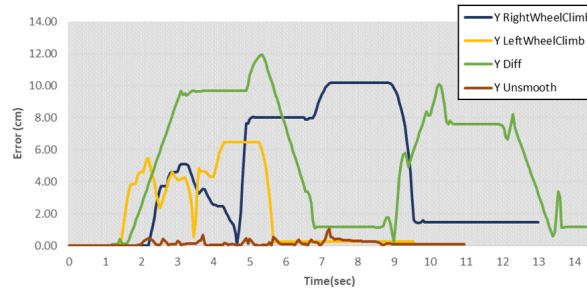
(a) Error position between simulator and estimate position Bullet in x-axis (cm).



(b) Error position between simulator and estimate position Bullet in y-axis (cm).



(c) Error position between simulator and estimate position ODE in x-axis (cm).



(d) Error position between simulator and estimate position ODE in y-axis (cm).

Figure 5.20: Comparison of the real-time error between estimate real-world(IMU) and CoppeliaSim.

display small object and floor conditions on the robot's path. Then, CoppeliaSim uses real-time and obstacle data for dynamic real-time simulation, displaying the object's movement mounted on the robot. The results show that the proposed system can display the robot posture, small object, and floor conditions during robot move over on, and real-time dynamic simulations to assess the movement and position of objects carried by the robot.

Conclusions

This study proposes a system that can integrate the real world with the robotic simulator. The main objective of this study is to create two systems. First, the system assists the operator in controlling the robot's movement in a narrow space, and second, the system detects and displays the floor condition and small objects on the floor. However, In the robotic simulation, the accuracy of the robot depends on the physical engine.

In this study, we experiment to test the concept of connection between real world and simulator via ROS and the accuracy of CoppeliaSim in Chapter 3. CoppeliaSim embedded script subscribed and published data in the ROS system. Linear and angular velocities of the robot model published to control real robots. The real robot position and orientation were used for measured simulation error. We validated the accuracy in terms of position and orientation. The sensor fusion by odometry data and IMU sensor provided by the ROS system are compared with the value obtained from the simulation. The results show that ODE is the most accurate while Bullet 2.83 causes more simulation errors. These results are similar to Michieletto's study [68]. Next, purpose the algorithm that makes the simulation stop and repositions when the error is high. The result shows CoppeliaSim can control the robot movement from positions A, B, and C and the model's movement with a few errors in the position. This error is caused by the plan flatness of the robot moving. Robots are placed on the carpet floor in the virtual environment, while the robot model is perfectly flat.

In Chapter 4, we propose the system to assist the operator in controlling the robot move along with a narrow path. Poor situational awareness is one of the problems in teleoperation robot control. To enhance the operator's situational awareness, efforts to improve

visual information or enhance the ability to perceive the environment. However, increasing environmental data increases the workloads of the operators, especially in situations where the robot is close to obstacles for a long time, such as moving through a long narrow path, or situations where there are many obstacles, such as Fukushima Daiichi, a nuclear power plant that was severely damaged by an earthquake in 2011. Many disaster areas have narrow-width routes due to the collapse of numerous structures. Some studies purpose the autonomy or intelligence system to improve teleoperation. Some autonomy-based approaches to teleoperation include shared control, safeguarded control, adjustable autonomy, and mixed initiatives. One limitation of these approaches is that some robot control is taken away from humans. This limits the robot to the behaviors and intelligence that have been preprogrammed. Especially in a narrow space where the robot is close to obstacles, its effect on an operator cannot control the robot with design direction. The performance of the autonomy system requires the complex setting with sensor devices and high computation of robot that affect operator time. Therefore, in this study, we propose an approach that can assist the semiautonomous system in controlling the robot in a narrow space. The operator can switch the mode to control the robot into a design position in a narrow space.

The Braitenberg algorithm is the concept for explaining the robot movement, such as avoid—ing an obstacle or moving to a target point. Some studies applied for autonomous robot move in designed point. However, one limitation requires more sensors to con—trol it. For example, we need the sensor around the robot to prevent the robot from colliding with an obstacle. In addition, some studies in the teleoperation system pur—pose the force feedback to improve environmental awareness. However, in a narrow space, the distance between obstacles is high. Therefore, the fore feedback is high, making it the operator difficult to control the robot in a narrow space. So, in this study, we applied the Braitenberg algorithm. We proposed the system that used a virtual sensor in CoppeliaSim to fill the limited Braitenberg algorithm. The results show the proposed system can help the robot move along with narrow path without collision and when compared with other systems such as 2D camera, 2D camera with map, force feedback system, and traversing the narrow path. The proposed system can achieve the goal with the lowest uptime.

We also employed the OMPL module to solve the limitations of the Braitenberg algorithm. Generally, the Braitenberg algorithm has difficulty directly controlling the robot

passing through narrow pathways or intersections between narrow pathways because the robot's movement depends on the sensor information. The OMPL module requires environmental data to determine a path that does not cause the robot to collide with obstacles. However, the environmental data are not exhaustive in the unknown environment. We only use a laser rangefinder to obtain the environmental data in this study. The space for target design is limited depending on the range of the laser rangefinder. So, the OMPL module was used in some cases (entering into narrow space and intersection between narrow spaces). The results showed that the OMPL module autonomously controlling robot moved smoothly from the start to the design position without colliding with walls. The operator could use the OMPL module to slightly move the robot from entering narrow space into the destination in the final narrow space. However, the operator needed to set the target goal depending on the sensor and camera data. Then, the robot moved to the target goal. The map was updated. The operator set the new robot targets; this increased the teleoperation time, increased workload, decreased situation awareness, and reduced performance. While the Braitenberg algorithm, the operator pushed the forward key to control the robot moves along with narrow space.

In Chapter 5, in environments with poor signal quality or limited illumination intensity. This makes the operator less aware of the environment, especially the surface or small objects on the floor. The lower the amount of communication one key factor to making the system. Therefore, in Chapter 5, we described the CoppeliaSim simulator for communicating with the IMU sensor odometry encoder to obtain data in the real world and use the data in real-time simulation. The simulator applied the IMU_complementary to filter the sensor noise and error to estimate the real-time robot pose. The estimated position of the robot using fusion data from the IMU and odometry encoder to estimate the robot position. Afterward, develop an algorithm in CoppeliaSim using dummy points to detect the robot model pose and then create a virtual object as the robot moves on a small object or not a flat floor. The results show that the CoppeliaSim can display the small object in three scenarios: robot's Left/right wheel climb on a small object, rough floor, and different floor levels. CoppeliaSim constructs a virtual cuboid model to represent the object on which the robot moves on. In this system, the height and length of the virtual object could be estimable using data from real devices. However, the system cannot obtain the object's width (we preset the object's width by a ratio of the robot's width). In addition, the system preset model object type is cuboid, so

CoppeliaSim shows small objects in a cuboid form. In contrast, the real object is cylindrical, but CoppeliaSim displays the height and length of the small object as they are in reality. However, CoppeliaSim can display objects with curved surfaces if sufficient curvature.

In addition, the limited number of data transmissions between the robot and operator results in a limitation on the number of sensors, so we use a simulator to fill the problem. If the actual cargo behavior could be estimated in advance, it would be possible to develop a suitable method of fixing the cargo. In this sense, simulation is essential. We merge dynamic simulation to estimate the object carried by the robot. However, contact modeling is one of the most challenging tasks in the simulator. We can use the real-time simulation to assist the operator in a fog environment during the teleoperation process. The simulator obtains real-time robot posture. Then the physical engine simulates the characteristic of the object model to help the operator understand the object that the robot is carrying when the robot is not moving on a flat floor. When the robot's right or left wheel climbs on a small object, the simulation could simulate the object trajectory the same as the actual object. When the robot moves on different floor levels and rough surfaces, there is a difference in the object's trajectory between the simulation and the real world. However, the trajectory characteristics have similar, and the differences are less in the simulator's final positions with the real world situation in each condition.

In this study, users did not use a robotic simulator to validate the initial problem or simulate different situations. It is also integrated into the real world to use the program's ability to overcome limitations that are not possible in the real world—for example, using simulated sensors to replace the requirement of Braitenberg algorithms that require a large number of sensors. Furthermore, applying a path planning module to design in certain situations where the environment information is limited or even using simulation programs to predict surface characteristics. Display the movement of objects carried on the robot these difficult or limited in the real world. Although this experiment used less data from the sensors to compute in the simulator, the system still needed a more stable connection. However, with the increased ability of internet connection, for example, the 5G technology or rapidly growing computer performance real-time simulation applications could be applied to other problems.

Bibliography

- [1] Daniel Saakes et al. “A teleoperating interface for ground vehicles using autonomous flying cameras”. In: *2013 23rd International Conference on Artificial Reality and Telexistence (ICAT)*. IEEE. 2013, pp. 13–19.
- [2] Zhefu Du et al. “Experimental evaluation of a tele-operated robot system in traversing a narrow path”. In: *2018 IEEE International Conference on Mechatronics and Automation (ICMA)*. IEEE. 2018, pp. 821–826.
- [3] Zbigniew Pietrzykowski. “Ship’s fuzzy domain-a criterion for navigational safety in narrow fairways”. In: *The Journal of Navigation* 61.3 (2008), p. 499.
- [4] Chaojian Shi, Mingming Zhang, and Jing Peng. “Harmonic potential field method for autonomous ship navigation”. In: *2007 7th International Conference on ITS Telecommunications*. IEEE. 2007, pp. 1–6.
- [5] Domokos Kiss and Dávid Papp. “Effective navigation in narrow areas: A planning method for autonomous cars”. In: *2017 IEEE 15th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*. IEEE. 2017, pp. 000423–000430.
- [6] Francisco-Angel Moreno et al. “Automatic waypoint generation to improve robot navigation through narrow spaces”. In: *Sensors* 20.1 (2020), p. 240.
- [7] Chaoqun Wang et al. “Autonomous mobile robot navigation in uneven and unstructured indoor environments”. In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2017, pp. 109–116.
- [8] Yong Nyeon Kim, Dong Wook Ko, and Il Hong Suh. “Confidence random tree-based algorithm for mobile robot path planning considering the path length and safety”. In: *International Journal of Advanced Robotic Systems* 16.2 (2019), p. 1729881419838179.

- [9] Ildar Farkhatdinov, Jee-Hwan Ryu, and Jinung An. “A preliminary experimental study on haptic teleoperation of mobile robot with variable force feedback gain”. In: *2010 IEEE Haptics Symposium*. IEEE. 2010, pp. 251–256.
- [10] Stephen Hughes et al. “Camera control and decoupled motion for teleoperation”. In: *SMC’03 Conference Proceedings. 2003 IEEE International Conference on Systems, Man and Cybernetics. Conference Theme-System Security and Assurance (Cat. No. 03CH37483)*. Vol. 2. IEEE. 2003, pp. 1339–1344.
- [11] Brenden Keyes et al. “Camera placement and multi-camera fusion for remote robot operation”. In: *Proceedings of the IEEE international workshop on safety, security and rescue robotics*. National Institute of Standards and Technology Gaithersburg, MD. 2006, pp. 22–24.
- [12] Maxim Sokolov, Oleg Bulichev, and Ilya Afanasyev. “Analysis of ROS-based Visual and Lidar Odometry for a Teleoperated Crawler-type Robot in Indoor Environment.” In: *ICINCO (2)*. 2017, pp. 316–321.
- [13] Abel Gawel et al. “Aerial-Ground collaborative sensing: Third-Person view for teleoperation”. In: *2018 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*. IEEE. 2018, pp. 1–7.
- [14] Patrick Stotko et al. “A VR system for immersive teleoperation and live exploration with a mobile robot”. In: *arXiv preprint arXiv:1908.02949* (2019).
- [15] Tomáš Kot and Petr Novák. “Application of virtual reality in teleoperation of the military mobile robotic system TAROS”. In: *International journal of advanced robotic systems* 15.1 (2018), p. 1729881417751545.
- [16] Jeffrey M Bradshaw et al. “Adjustable autonomy and human-agent teamwork in practice: An interim report on space applications”. In: *Agent autonomy*. Springer, 2003, pp. 243–280.
- [17] David J Bruemmer, Donald D Dudenhofer, and Julie L Marble. “Dynamic-Autonomy for Urban Search and Rescue.” In: *AAAI mobile robot competition*. Menlo Park, CA. 2002, pp. 33–37.
- [18] Nathan J Kanyok. “Situational Awareness Monitoring for Humans-In-The-Loop of Telepresence Robotic Systems”. PhD thesis. Kent State University, 2019.

- [19] Terrence W Fong et al. “Novel interfaces for remote driving: gesture, haptic, and PDA”. In: *Mobile Robots XV and Telemanipulator and Telepresence Technologies VII*. Vol. 4195. International Society for Optics and Photonics. 2001, pp. 300–311.
- [20] Nicola Diolaiti and Claudio Melchiorri. “Teleoperation of a mobile robot through haptic feedback”. In: *IEEE International Workshop HAVE Haptic Virtual Environments and Their*. IEEE. 2002, pp. 67–72.
- [21] Miguel A Olivares-Mendez, Somasundar Kannan, and Holger Voos. “Vision based fuzzy control autonomous landing with UAVs: From V-REP to real experiments”. In: *2015 23rd Mediterranean Conference on Control and Automation (MED)*. IEEE. 2015, pp. 14–21.
- [22] João José Oliveira Barros, Vitor Manuel Ferreira dos Santos, and Filipe Miguel Teixeira Pereira da Silva. “Bimanual haptics for humanoid robot teleoperation using ROS and V-REP”. In: *2015 IEEE International Conference on Autonomous Robot Systems and Competitions*. IEEE. 2015, pp. 174–179.
- [23] François Ferland et al. “Egocentric and exocentric teleoperation interface using real-time, 3D video projection”. In: *Proceedings of the 4th ACM/IEEE international conference on Human robot interaction*. 2009, pp. 37–44.
- [24] Aaron Staranowicz and Gian Luca Mariottini. “A survey and comparison of commercial and open-source robotic simulator software”. In: *Proceedings of the 4th International Conference on Pervasive Technologies Related to Assistive Environments*. 2011, pp. 1–8.
- [25] Chenguang Yang et al. “Teleoperated robot writing using EMG signals”. In: *2015 IEEE International Conference on Information and Automation*. IEEE. 2015, pp. 2264–2269.
- [26] Omar Adjali et al. “Multimodal fusion, fission and virtual reality simulation for an ambient robotic intelligence”. In: *Procedia Computer Science* 52 (2015), pp. 218–225.
- [27] Xiaoyu Yang, Rajnikant V Patel, and Mehrdad Moallem. “A fuzzy–braitenberg navigation strategy for differential drive mobile robots”. In: *Journal of Intelligent and Robotic Systems* 47.2 (2006), pp. 101–124.
- [28] Morgan Quigley et al. “ROS: an open-source Robot Operating System”. In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.

- [29] Tully Foote. “tf: The transform library”. In: *2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA)*. IEEE. 2013, pp. 1–6.
- [30] Christopher Crick et al. “Rosbridge: Ros for non-ros users”. In: *Robotics Research*. Springer, 2017, pp. 493–504.
- [31] Nick Hawes et al. “The strands project: Long-term autonomy in everyday environments”. In: *IEEE Robotics & Automation Magazine* 24.3 (2017), pp. 146–156.
- [32] Serena Ivaldi et al. “Tools for simulating humanoid robot dynamics: a survey based on user feedback”. In: *2014 IEEE-RAS International Conference on Humanoid Robots*. IEEE. 2014, pp. 842–849.
- [33] Lucas Nogueira. “Comparative analysis between gazebo and v-rep robotic simulators”. In: *Seminario Interno de Cognicao Artificial-SICA* 2014.5 (2014).
- [34] Lenka Pitonakova et al. “Feature and performance comparison of the V-REP, Gazebo and ARGoS robot simulators”. In: *Annual Conference Towards Autonomous Robotic Systems*. Springer. 2018, pp. 357–368.
- [35] Evan Drumwright et al. “Extending open dynamics engine for robotics simulation”. In: *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. Springer. 2010, pp. 38–50.
- [36] Jean J Moreau. “Unilateral contact and dry friction in finite freedom dynamics”. In: *Nonsmooth mechanics and Applications*. Springer, 1988, pp. 1–82.
- [37] Jean Jacques Moreau. “Some numerical methods in multibody dynamics: application to granular materials”. In: *European Journal of Mechanics-A/Solids* 13.4-suppl (1994), pp. 93–114.
- [38] Michel Jean. “Frictional contact in collections of rigid or deformable bodies: numerical simulation of geomaterial motions”. In: *Studies in Applied Mechanics*. Vol. 42. Elsevier, 1995, pp. 463–486.
- [39] Ioan A Sucan, Mark Moll, and Lydia E Kavraki. “The open motion planning library”. In: *IEEE Robotics & Automation Magazine* 19.4 (2012), pp. 72–82.
- [40] Ioan A Sucan and Lydia E Kavraki. “A sampling-based tree planner for systems with complex dynamics”. In: *IEEE Transactions on Robotics* 28.1 (2011), pp. 116–131.

- [41] Gildardo Sánchez and Jean-Claude Latombe. “A single-query bi-directional probabilistic roadmap planner with lazy collision checking”. In: *Robotics research*. Springer, 2003, pp. 403–417.
- [42] David Hsu, J-C Latombe, and Rajeev Motwani. “Path planning in expansive configuration spaces”. In: *Proceedings of International Conference on Robotics and Automation*. Vol. 3. IEEE. 1997, pp. 2719–2726.
- [43] Steven M LaValle and James J Kuffner Jr. “Randomized kinodynamic planning”. In: *The international journal of robotics research* 20.5 (2001), pp. 378–400.
- [44] Erion Plaku, Kostas E Bekris, and Lydia E Kavraki. “Oops for motion planning: An online, open-source, programming system”. In: *Proceedings 2007 IEEE International Conference on Robotics and Automation*. IEEE. 2007, pp. 3711–3716.
- [45] James J Kuffner and Steven M LaValle. “RRT-connect: An efficient approach to single-query path planning”. In: *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*. Vol. 2. IEEE. 2000, pp. 995–1001.
- [46] Lydia E Kavraki et al. “Probabilistic roadmaps for path planning in high-dimensional configuration spaces”. In: *IEEE transactions on Robotics and Automation* 12.4 (1996), pp. 566–580.
- [47] Robert Bohlin and Lydia E Kavraki. “Path planning using lazy PRM”. In: *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*. Vol. 1. IEEE. 2000, pp. 521–528.
- [48] Steven M LaValle et al. “Rapidly-exploring random trees: A new tool for path planning”. In: (1998).
- [49] Iwan Ulrich and Johann Borenstein. “VFH/sup*: Local obstacle avoidance with look-ahead verification”. In: *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*. Vol. 3. IEEE. 2000, pp. 2505–2511.
- [50] Iwan Ulrich and Johann Borenstein. “VFH+: Reliable obstacle avoidance for fast mobile robots”. In: *Proceedings. 1998 IEEE international conference on robotics and automation (Cat. No. 98CH36146)*. Vol. 2. IEEE. 1998, pp. 1572–1577.

- [51] Indri Purwita Sary et al. “Design of Obstacle Avoidance System on Hexacopter Using Vector Field Histogram-Plus”. In: *2018 IEEE 8th International Conference on System Engineering and Technology (ICSET)*. IEEE. 2018, pp. 18–23.
- [52] SS Bolbhat et al. “Intelligent Obstacle Avoiding AGV Using Vector Field Histogram and Supervisory Control”. In: *Journal of Physics: Conference Series*. Vol. 1716. 1. IOP Publishing. 2020, p. 012030.
- [53] Vladimir J Lumelsky and Alexander A Stepanov. “Path-planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape”. In: *Algorithmica* 2.1 (1987), pp. 403–430.
- [54] Ishay Kamon, Elon Rimon, and Ehud Rivlin. “Tangentbug: A range-sensor-based navigation algorithm”. In: *The International Journal of Robotics Research* 17.9 (1998), pp. 934–953.
- [55] Kamilah Taylor and Steven M LaValle. “I-Bug: An intensity-based bug algorithm”. In: *2009 IEEE International Conference on Robotics and Automation*. IEEE. 2009, pp. 3981–3986.
- [56] Muhammad Zohaib et al. “IBA: Intelligent Bug Algorithm—A novel strategy to navigate mobile robots autonomously”. In: *International Multi Topic Conference*. Springer. 2013, pp. 291–299.
- [57] Nishant Sharma, Jose Pinto, and PB Sujit. “BugFlood: A bug inspired algorithm for efficient path planning in an obstacle rich environment”. In: *AIAA Infotech@Aerospace*. 2016, p. 0254.
- [58] James Ng and Thomas Bräunl. “Performance comparison of bug navigation algorithms”. In: *Journal of Intelligent and Robotic Systems* 50.1 (2007), pp. 73–84.
- [59] Alpaslan Yufka and Osman Parlaktuna. “Performance comparison of bug algorithms for mobile robots”. In: *Proceedings of the 5th international advanced technologies symposium, Karabuk, Turkey*. 2009, pp. 13–15.
- [60] JA Goguen. “LA Zadeh. Fuzzy sets. Information and control, vol. 8 (1965), pp. 338–353.-LA Zadeh. Similarity relations and fuzzy orderings. Information sciences, vol. 3 (1971), pp. 177–200.” In: *The Journal of Symbolic Logic* 38.4 (1973), pp. 656–657.

- [61] Sng Hong Lian. “Fuzzy logic control of an obstacle avoidance robot”. In: *Proceedings of IEEE 5th International Fuzzy Systems*. Vol. 1. IEEE. 1996, pp. 26–30.
- [62] Dimiter Driankov and Alessandro Saffiotti. *Fuzzy logic techniques for autonomous vehicle navigation*. Vol. 61. Physica, 2013.
- [63] Patrick Reignier. “Fuzzy logic techniques for mobile robot obstacle avoidance”. In: *Robotics and Autonomous Systems* 12.3-4 (1994), pp. 143–153.
- [64] Tao Dong et al. “Path tracking and obstacles avoidance of uavs-fuzzy logic approach”. In: *The 14th IEEE International Conference on Fuzzy Systems, 2005. FUZZ’05*. IEEE. 2005, pp. 43–48.
- [65] Tae-Seok Jin. “Obstacle avoidance of mobile robot based on behavior hierarchy by fuzzy logic”. In: *International Journal of Fuzzy Logic and Intelligent Systems* 12.3 (2012), pp. 245–249.
- [66] Xi Li and Byung-Jae Choi. “Design of obstacle avoidance system for mobile robot using fuzzy logic systems”. In: *International Journal of Smart Home* 7.3 (2013), pp. 321–328.
- [67] Anish Pandey et al. “Path planning navigation of mobile robot with obstacles avoidance using fuzzy logic controller”. In: *2014 IEEE 8th international conference on intelligent systems and control (ISCO)*. IEEE. 2014, pp. 39–41.
- [68] Stefano Michieletto, Davide Zanin, and Emanuele Menegatti. “Nao robot simulation for service robotics purposes”. In: *2013 European Modelling Symposium*. IEEE. 2013, pp. 477–482.
- [69] Achim J Lilienthal and Tom Duckett. “Experimental analysis of smelling Braitenberg vehicles”. In: *IEEE international conference on advanced robotics (ICAR 2003), Coimbra, Portugal, June 30-July 3, 2003*. Vol. 1. Coimbra, University. 2003, pp. 375–380.
- [70] Curtis W Nielsen, Michael A Goodrich, and Robert W Ricks. “Ecological interfaces for improving mobile robot teleoperation”. In: *IEEE Transactions on Robotics* 23.5 (2007), pp. 927–941.
- [71] Holly A Yanco, Jill L Drury, and Jean Scholtz. “Beyond usability evaluation: Analysis of human-robot interaction at a major robotics competition”. In: *Human-Computer Interaction* 19.1-2 (2004), pp. 117–149.

- [72] Xieyuanli Chen et al. “Robust SLAM system based on monocular vision and LiDAR for robotic urban search and rescue”. In: *2017 IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*. IEEE. 2017, pp. 41–47.
- [73] Michael Baker et al. “Improved interfaces for human-robot interaction in urban search and rescue”. In: *2004 IEEE International Conference on Systems, Man and Cybernetics (IEEE Cat. No. 04CH37583)*. Vol. 3. IEEE. 2004, pp. 2960–2965.
- [74] Brian J Odelson, Murali R Rajamani, and James B Rawlings. “A new autocovariance least-squares method for estimating noise covariances”. In: *Automatica* 42.2 (2006), pp. 303–308.
- [75] T. Moore and D. Stouch. “A Generalized Extended Kalman Filter Implementation for the Robot Operating System”. In: *Proceedings of the 13th International Conference on Intelligent Autonomous Systems (IAS-13)*. Springer, July 2014.

APPENDIX A

List of Publications

A.1 International Journal (peer reviewed)

- Nattawat Pinrath and Nobuto Matsuhira, Integration of Robotic Simulator with Robot operation system for Real-time simulation to teleoperation a Root in Narrow path, Journal of Robotics and Mechatronics.(to be publish in vol.34, No.3, Jun. 20, 2022.)

A.2 International Conference

- Nattawat Pinrath and Nobuto Matsuhira, Integration of Virtual Robot Environmental Platform with Robot Operating System for Real-time Simulation of A Robot, The SICE Annual Conference 2020, 2020 (DOI: 10.23919/SICE48898.2020.9240370)
- Phang Darren Ren Yee, Nattawat PinrathNobuto MatsuhiraAutonomous Mobile Robot Navigation Using 2D LiDAR and Inclined Laser Rangefinder to Avoid a Lower Object, Proceedings of the SICE Annual Conference 2020September 23-26, 2020 9 (DOI: 10.23919/SICE48898.2020.9240417)
- Nattawat Pinrath and Nobuto Matsuhira, Intregrating ROS with V-REP simulator for development real-time dynamic control mobile robot, 14th South East Asian Technical University Consortium,2020 (Proceeding of seatuc 2020, page 26)
- Li Ke, Tingxin Song, Nattawat Pinrath, Darren Phang Ren Yee, Nobuto Matsuhira, Basic Experiments for a Remote Control Robot-Mapping System in Com-

APPENDIX A: LIST OF PUBLICATIONS

- plex Environment, poster 174, 2019 IEEE International Conference on Mechatronics and Automation (ICMA 2019) (DOI: 10.1109/ICMA.2019.8816250)
- Nattawat Pinrath and Nobuto Matsuhira, Real-time Simulation System for Tele-operated Mobile Robots using V-REP, ADVANCES IN ROBOTICS 2019, 2019, Chennai, India (DOI:10.1145/3352593.3352598)

APPENDIX B

ROS ekf__localization__node

In this study applied the ekf_localization_node for fusing the data from IMU with odometry encoder to reduce the error. ROS ekf_localization_node can obtain an overall position estimate whose error is less than would be possible by using a single sensor in isolation. The package currently contains an implementation of an extended Kalman filter (EKF) It is often the case that a greater amount of sensor input data will produce more accurate position estimates. It is therefore critical that any software that performs sensor fusion on a mobile robot platform is able to take in any and all available data on the platform to continuous estimation. Each state estimation node in robot_localization begins estimating the vehicle's state as soon as it receives a single measurement. If there is a holiday in the sensor data (i.e., a long period in which no data is received), the filter will continue to estimate the robot's state via an internal motion model. All state estimation ekf_localization_node track the 15-dimensional state of the vehicle: (X, Y, Z, roll, pitch, yaw, \dot{X} , \dot{Y} , \dot{Z} , \dot{roll} , \dot{pitch} , \dot{yaw} , \ddot{X} , \ddot{Y} , \ddot{Z}) from IMU sensor and Odometry encoder.

B.1 Extended Kalman Filter Node

The EKF formulation and algorithm are well-known [3, 4, 5]. We detail them here to convey important implementation details. Our goal is to estimate the full 3D (6DOF) pose and velocity of a mobile robot over time. The process can be described as a nonlinear dynamic system, with

$$x_k = f(x_{k-1}) + W_{k-1} \quad (\text{B.1.1})$$

where x_k is the robot's system state (i.e., 3D pose) at time k , f is a nonlinear state transition function, and w_{k-1} is the process noise, which is assumed to be normally distributed. Our 12-dimensional state vector, x , comprises the vehicle's 3D pose, 3D orientation, and their respective velocities. Rotational values are expressed as Euler angles. Additionally, we receive measurements of the form

$$z_k = h(x_k) + \nu_k \quad (\text{B.1.2})$$

where z_k is the measurement at time k , h is a nonlinear sensor model that maps the state into measurement space, and ν_k is the normally distributed measurement noise.

The first stage in the algorithm, shown as equations B.1.3 and B.1.4, is to carry out a prediction step that projects the current state estimate and error co-variance forward in time:

$$\hat{x}_k = f(x_{k-1}) \quad (\text{B.1.3})$$

$$\hat{P}_k = F P_{k-1} F^T + Q. \quad (\text{B.1.4})$$

For our application, f is a standard 3D kinematic model derived from Newtonian mechanics. The estimate error covariance, P , is projected via F , the Jacobian of f , and then perturbed by Q , the process noise covariance.

We then carry out a correction step in equations B.1.5 through B.1.7:

$$K = \hat{P}_k H^T (H \hat{P}_k H^T + R)^{-1} \quad (\text{B.1.5})$$

$$\hat{x}_k = \hat{x} + K(z - H \hat{x}_k) \quad (\text{B.1.6})$$

$$P_k = (I - KH) \hat{P}_k (I - KH)^T + K R K^T \quad (\text{B.1.7})$$

We calculate the Kalman gain using our observation matrix, H our measurement covariance, R , and \hat{P}_k use the gain to update the state vector and covariance matrix. We

Table B.1: The error for sensor configuration.

Sensor set	Loop Closure error $x,y(m)$	Estimate Std. Dev. $x,y(m)$
Odometry	69.65, 160.33	593.09, 359.08
Odometry + one IMU	10.23, 47.09	5.25, 5.25

employ the Joseph form covariance update equation B.1.6 to promote filter stability by ensuring that P_k remains positive semi-definite.

Jacobian matrix of the observation model function H . To support a broad array of sensors, we make the assumption that each sensor produces measurements of the state variables we are estimating. As such, H is simply the identity matrix. A core feature of `ekf_localization_node` is that it allows for partial updates of the state vector, which is also a requirement of any future state estimation nodes that are added to `robot_localization`. This is critical for taking in sensor data that does not measure every variable in the state vector, which is nearly always the case. In practice, this can be accomplished through H . Specifically, when measuring only m variables, H becomes an m by 12 matrix of rank m , with its only nonzero values (in this case, ones) existing in the columns of the measured variables. Because the process noise covariance, Q can be difficult to tune for a given application[74], `ekf_localization_node` exposes this matrix as a parameter to users, allowing for an additional level of customization.

In [75], report the performance of `ekf_localization_node` by loop closure experiment, the loop closure error is reported for x and y position show in Table B.1.

Acknowledgments

Though my name appears on the front cover of this dissertation, all of the compliments and my gratitude must go to Professor Nobuto Matsuhira. My supervisor throughout my years at Shibaura Institute of Technology. I have learned so much, yet, I still feel there is so much more I could learn from him. I could not have imagined having a better and friendlier supervisor. I am fortunate to have been her student for these five years of my time at Shibaura Institute of Technology. Also thanks to committee members, Prof. Yutaka Uchimura, Prof. Takashi Yoshimi and Prof. Akira Shimada at Shibaura institute of technology and Prof. Gen Endo at Tokyo Institute of Technology who kindly guidance and support. Your encouraging words and thoughtful, detailed feedback have been very important to me.

Besides, I would like to offer my special thanks to all other brave participants at Shibaura Institute of Technology and also from outside, who helped me out with my experiment and offered themselves in the name of science and academic pursuit.

I am grateful to my father, mother, and younger brother. Being away from home was never easy, and they have been excellent support for that. Not only telling me that I will be doing well, but they have also made sure that I have nothing to worry about at home. They have made sure that I will focus 100% on my path, my journey, and my dream. Moreover, for that, I can never be thankful enough.

My special gratitude goes down to the MEXT scholarship and all of the university staff. These organizations and personnel have provided enormous support for me to develop cutting-edge innovations and research.

Nattawat Pinrath