Autonomous Navigation for a Two-Wheeled Unmanned Ground Vehicle: Design and Implementation

by

Tianxiang Lu
B. Eng., Donghua University, 2016

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of Mechanical Engineering

Autonomous Navigation for a Two-Wheeled Unmanned Ground Vehicle: Design and Implementation

by

Tianxiang Lu

B. Eng., Donghua University, 2016

Supervisory Committee

ABSTRACT

Unmanned ground vehicles (UGVs) have been widely used in many areas such as agriculture, mining, construction and military applications. This results from the fact that UGVs can not only be easily built and controlled, but also be featured with high mobility and handling hazardous situations in complex environments. Among the competences of UGVs, autonomous navigation is one of the most challenging problems. This is because that the success in achieving autonomous navigation depends on four factors: Perception, localization, cognition, and proper motion controller.

In this thesis, we introduce the realization of autonomous navigation for a two-wheeled differential ground robot under the robot operating system (ROS) environment from both the simulation and experimental perspectives. In Chapter 2, the simulation work is discussed. Firstly, the robot model is described in the unified robot description format (URDF)-based form and the working environment for the robot is simulated. Then we use the *gmapping* package which is one of the packages integrating simultaneous localization and mapping (SLAM) algorithm to build the map of the working environment. In addition, ROS packages including *tf*, *move_base*, *amcl*, etc., are used to realize the autonomous navigation. Finally, simulation results show the feasibility and effectiveness of the autonomous navigation system for the two-wheeled UGV with the ability to avoid collisions with obstacles.

In Chapter 3, we introduce the experimental studies of implementing autonomous navigation for a two-wheeled UGV. The necessary hardware peripherals on the UGV to achieve autonomous navigation are given. The process of implementation in the experiment is similar to that in simulation, however, calibration of several devices is necessary to adapt the scenario in a practical environment. Additionally, a proportional-integral-derivative (PID) controller for the robot base is used to handle the external noise during the experiment. The experimental results demonstrate the success in the implementation of autonomous navigation for the UGV in practice.

Table of Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

Firstly, I would like to sincerely thank my supervisor, Dr. Yang Shi, a respectable and decent professor, for providing me with this precious opportunity to conduct research in his enthusiastic group. During my MASc program, he guided me not only to take several courses, but also to read classical textbooks in areas related to my research. This has laid solid foundation for my research. He kept on offering valuable comments and suggestions in individual meetings and group meetings throughout the whole program. Especially at the end of my program when we experienced the COVID-19 pandemic, he gave me great mental encouragement. It is a great honor for me to have such a considerate supervisor.

I would also like to express my thanks to all my group members in ACIPL. Thank Huaiyuan Sheng for introducing all lab members at the first day of joining the group. Thank Tianyu Tan and Kunwu Zhang for providing help in my daily life. Dr. Chao Shen and Zhang Zhang gave me help in solving optimization problems. Jicheng Chen not only offered instructions on my research, but also sport fitness to help me stay healthy physically. Thank Chonghan Ma, Zhuo Li and Chen Ma for offering help in writing this thesis. And I really appreciate the help from Qi Sun, Qian Zhang, Henglai Wei, Yuan Yang, Changxin Liu, Xiang Sheng and Xinxin Shang.

Lastly, I would like to give my special thanks to my family and my girlfriend Rui. They always gave me unlimited love and support despite of ups and downs. I hope I can repay them with happy life in the future.

Acronyms

UGV	unmanned ground vehicle
UAV	unmanned aerial vehicle
GPS	global positioning system
SLAM	simultaneous localization and mapping
URDF	unified robot description format
ROS	robot operating system
TCP	transmission control protocol
UDP	user datagram protocol
AMCL	adaptive Monte Carlo localization
DWA	dynamic window approach
API	application programming interface
XML	extensible markup language
MCU	microcontroller unit
PID	proportional-integral-derivative
IMU	inertial measurement unit
EKF	extended Kalman filter
UKF	unscented Kalman filter
PWM	pulse width modulation

Chapter 1

Introduction

1.1 Overview of the unmanned ground vehicle (UGV)

An *unmanned ground vehicle* (UGV) refers to a vehicle that operates on the ground without human intervention [2]. Originally UGVs were developed for military applications such as exploring space with radiation levels, repairing runway under enemy fire, and processing packages with potential danger. Then during the late 1960s, components of UGVs including sensors, control systems and communication links achieved rapid development. This resulted in that an increasing amount of research efforts have been put on designing UGVs to satisfy a variety of requirements for different applications [3].

According to the operation environment, UGVs can be divided into two categories: Indoor UGVs and outdoor UGVs. Indoor UGVs are mostly prototypes of outdoor UGVs before field applications or designed for research and educational purposes. Typical applications using indoor UGVs include pattern recognition and following [4], data processing in indoor environments [5], robot soccer play and autonomous navigation [6]. Generally indoor UGVs possess the advantages of small size, high customizability and low cost. Well-known indoor UGVs include Dingo by Clearpath Robotics [7] and Turtlebot by Willow Garage [8]. Figure 1.1 shows the Turtlebot 2e which is the second edition of Turtlebot.

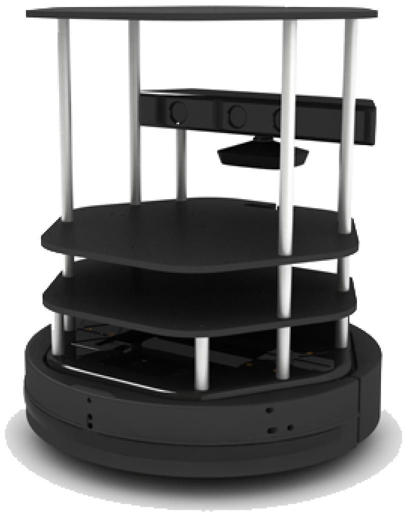


Figure 1.1: Turtlebot 2e - An indoor UGV [1].

Compared to indoor UGVs, most outdoor UGVs are more functional and versatile, and have more complex structures. This results from twofold aspects. Firstly, outdoor UGVs operate in complicated environments such as rough terrains, mine fields, and toxic or hazardous environments. As these environments may be inconvenient or impossible to have a human operator present, these UGVs are equipped with reliable long-distance communication systems, high-performance visual systems, high-precision sensors for accurate localization, and large-capacity battery. This leads to that they have large sizes and weights. Secondly, they are required to conduct complicated missions [9] such as mine detection [10], fire detection and fighting [11], farmland work using tractor-trailer systems [12], pesticide spraying [13], and selective stabilization of images when operating in terrains [14]. In addition, one single UGV can be used along with multiple UGVs or other robotic systems such as *unmanned*

[1] `https://www.turtlebot.com/turtlebot2/`

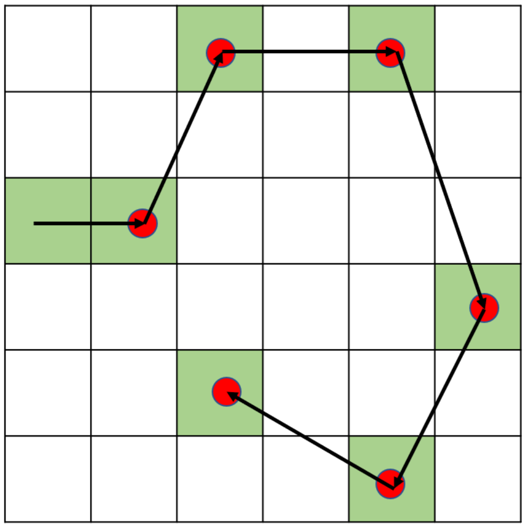
aerial vehicles (UAVs) to complete tasks such as formation operation [15], collaborative patrolling [16], and cooperative path planning for target tracking [17, 18]. Figure 1.2 shows an outdoor UGV named Warthog which is suitable for applications in mining, agriculture and environment monitoring.

Figure 1.2: Warthog - An outdoor UGV [2].

1.2 Autonomous navigation for ground vehicles

One of the most widely-used applications using ground vehicles is the autonomous navigation. Generally, there are two types of autonomous navigation for ground vehicles: Waypoint (point-to-point) navigation and path following navigation. Waypoint navigation as illustrated in Figure 1.3(a) requires a robot to reach specified locations. And a desired path is required to be followed in path following navigation as shown in Figure 1.3(b). To achieve autonomous navigation, a robot should be able to plan its paths, execute the plan without human intervention, and deal with any possible unexpected obstacles.

[2] `https://clearpathrobotics.com/warthog-unmanned-ground-vehicle-robot/`


(a) Waypoint navigation.

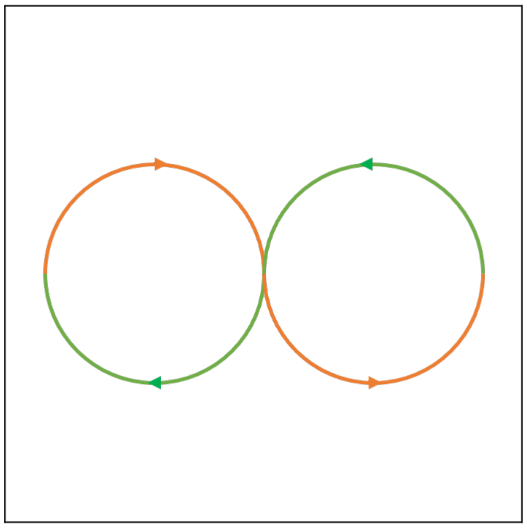

(b) Path following navigation.

Figure 1.3: Two types of autonomous navigation.

Autonomous navigation can be considered as the hardest task for a ground vehicle to deal with. However, it is the most needed ability that many ground vehicles should possess. Why is the autonomous navigation for ground vehicles challenging? This is because that the success in autonomous navigation depends on the realization of four perspectives: Perception, localization, cognition, and motion control [19].

The perception of a robot relies on whether the robot is able to extract useful data from sensors. The sensor system embedded in a ground vehicle often consists of two groups: Navigation sensors and visual sensors [20]. The navigation sensors provide the vehicles with localization abilities and visual sensors enable the vehicles to observe the environment.

Typical navigation sensors include the global positioning system (GPS), inertial measurement unit (IMU) and encoders attached to the motors on vehicles. GPS can directly provide the information on 3-dimensional (3D) position in the global range [21]. Thus it has been mostly applied in outdoor applications such as forest patrol [22], outdoor exploration [23] and autonomous driving in the urban environment [24].

Due to the advantage of lower cost than GPS, IMU and encoders are more widely used in civilian applications. Also, compared to GPS, IMU and motor encoders can only provide position information in terms of odometry measurements such as accelerations, velocities and orientation. However, as the position of a robot is determined by these measurement data, even slight inaccuracy in the raw data may deteriorate the precision of computed positions. To increase the reliability of obtaining positions using these devices, adequate calibration methods are necessary [25]. And it is also feasible to combine different sensors to integrate the data obtained from these sensors. In this process, by applying data fusion algorithms such as the Kalman filter [21, 26], the positions calculated by making use of data from different navigation sensors can be more accurate.

Essentially visual sensors endow the ground vehicles with the ability to sense the ranges during autonomous navigation. Specifically, visual sensors provide ground vehicles with the distance from themselves to the obstacles, which prevents collisions. Common visual sensors include Lidar, Radar and depth cameras. Generally, Lidar and Radar sensors achieve a larger detection range with higher precision than cameras. Depth cameras with binocular or trinocular vision are more preferable for indoor UGV applications because of relatively lower prices than Lidar and Radar. Also, cameras with monocular vision have been proved to be effective to perform autonomous navigation for UGVs [27, 28]. Similarly to the navigation sensor system, the vision sensor system can be constituted of a single visual sensor or a group of different ones. However, as the volume of data obtained from images captured by visual sensors is large, efficient data fusion techniques are necessary to avoid overhead data processing [29].

Success in localization means that a robot is aware of its position in the surrounding environment. The well-known *simultaneous localization and mapping (SLAM)* technique provides a feasible solution to handle the localization issue. The SLAM heavily relies on the perception ability since the position is determined by applying suitable SLAM algorithms to process the data extracted from sensors. Generally, the navigation data provides the odometry information which is concerned with obtaining the current position relatively to the starting point of a robot. And the visual data which shows the current view facilitates the estimation of current position with respect to the overall surrounding environment. By integrating these two groups of data, the map of an unknown environment can be built in real time [30]. To increase the efficiency of achieving SLAM, visual SLAM has been proposed. It only needs the data from visual sensors and does not need data from navigation sensors [28, 31]. However, it requires visual sensors with high precision during the map building process and reliable algorithms for position estimation and feature extraction to achieve autonomous navigation.

The cognition ability for a UGV can be defined as the act of reaching given goals, i.e., a cognitive UGV is able to plan a collision-free path from the current position to the target position. Therefore, it can be seen that the cognition is strongly connected to the localization. If the UGV does not know its position, it cannot make judgment on whether it arrives at the target position. Meanwhile, a cognitive UGV should be integrated with path planning algorithms which solve the find-path problem for the robot. The most common path planning algorithm is considered as the $A*$ algorithm [32] which is also adopted for path planning in this thesis. A brief introduction to this algorithm will be given in Chapter 2. There are also other path planning algorithms such as *genetic algorithm* [33], *particle swarm optimization (PSO)* [34], and prediction-based algorithm by utilizing Markov Decision Process [35]. Essentially they are all optimization-based algorithms which compute a feasible path by solving an optimization problem.

The motion control problem for UGVs is considerably difficult to give a brief description here as it is highly dependent on the hardware structure of UGVs. However, it is known to us that an effective motion controller enables a UGV to follow the paths computed by the path planning algorithms. As a result, the autonomous navigation can be achieved. Therefore, from above description we can see that the four elements to achieve the success in autonomous navigation: Perception, localization, cognition and motion control are strongly related to each other. This makes the autonomous navigation for ground vehicles a challenging mission.

1.3 Contributions

The implementation of autonomous navigation for our existing two-wheeled UGV is motivated by developing applications such as auto car parking management system which can utilize this functionality. Considering the economic cost of implementation, the ground vehicle is built with devices with considerably low prices and precision. However, it is feasible to achieve autonomous navigation. The contributions of this thesis are summarized as follows:

- A model of our two-wheeled differential ground vehicle described in the unified robot description format (URDF) is built for simulation.
- The SLAM with utilizing mainly gmapping package for the two-wheeled ground vehicle is realized in both simulation and experiment.
- The autonomous navigation for the two-wheeled ground vehicle with the ability to avoid obstacles is realized in both simulation and experiment.

1.4 Thesis organization

The remainder of this thesis is organized as follows:

- **Chapter 2** introduces several basic concepts in robot operating system (ROS) and the adopted framework of navigation stack. After describing the process of building a URDF-based model for the two-wheeled UGV, the simulation results are given to show the realization of autonomous navigation.
- **Chapter 3** mainly discusses the experimental results in the implementation of autonomous navigation. The hardware components installed on our UGV to facilitate the implementation and the calibration of these devices are briefly introduced. To mitigate the influence brought by the external disturbances during the experiment, a PID-based controller is used. And the experimental results demonstrate the effectiveness of the adopted framework of navigation stack and the success in realization in a practical application.
- **Chapter 4** concludes this thesis and gives future research directions.

Chapter 2

Simulation of Autonomous Navigation for a Two-Wheeled UGV

2.1 Introduction to ROS

Nowadays, many UGVs are equipped with the ROS environment [36]. With highly-integrated feature packages provided by the ROS community, many challenging competences required of UGVs can be implemented in much easier manners. Thus the efficiency of developing high-performance UGVs can be significantly improved. In our work, ROS is also adopted. Therefore, we first give a brief introduction to ROS.

As the name implies, ROS is an operating system designed for robots [37]. However, ROS is different from traditional operating systems such as *Windows* and *Linux* whose operation is directly dependent on the computer hardware. In terms of the framework, ROS can be divided into three layers: Operating system (OS) layer, intermediate layer, and application layer. The OS layer is the operation platform that

ROS runs on. Currently ROS can only run on Unix-based platforms such as *Ubuntu*, *macOS*, *Debian* etc. The intermediate layer enables the communication between the operation system and robot based on *transmission control protocol* (TCP) or *user datagram protocol* (UDP). In addition, the intermediate layer provides client libraries for the application layer. In the application layer, open-source repositories can be used to implement different applications. The modules in repositories operate in the unit of ROS node and the nodes are managed by a so-called “Master”, individually or in groups. Below we introduce some basic concepts in ROS operation.

2.1.1 ROS concepts

A package is the fundamental unit for organizing software in ROS [38]. A ROS package may contain ROS nodes, libraries, datasets, configuration and executable files, etc. A ROS package is the biggest unit for building and releasing by users.

ROS nodes are processes that perform computation and can also be called as application or software modules with each node having its functions. Nodes communicate with each other by passing ROS messages [39].

A ROS message is simply a data structure which can be custom-built by users or one of the built-in ROS messages. The built-in ROS messages support standard primitive types such as integer, floating point, Boolean, and also arbitrarily nested structures and arrays. Messages are transmitted under the publish/subscribe semantics, i.e., a message is routed from a publisher node to a subscriber node. Specifically, the publisher node publishes messages to a given topic and the subscriber node can receive messages from the corresponding topic which it subscribes to [40].

The publish/subscribe based communication model utilizes a many-to-many, one-way transport mechanism, thus not suitable to be used in distributed systems. Instead we have ROS services to handle this situation. ROS services are based on the client/server model. Services can consist of one server node and many client nodes. Clients use a service by sending requests to the server and server replies the corresponding messages to clients [41].

2.1.2 Navigation stack

The navigation stack provides a systematic setup to achieve autonomous navigation for a robot. Figure 2.1 shows the operating principle of the navigation stack. The navigation stack takes the data from odometry source, sensor streams, and the navigation goal as inputs, while sends the desired speed signal as output to the base controller of a robot. As mentioned in Chapter 1, the success in autonomous navigation relies on the success in four aspects: Perception, localization, cognition, and motion control [19, 42]. These four building blocks for autonomous navigation can be seen from the navigation stack setup. Perception is reflected in that a robot needs messages from sensor transforms, odometry and sensor sources such as odometers and lasers. The odometry information helps the robot to localize itself. However, the odometry sources may not provide consistent reliable data. For instance, when the wheels of a ground robot skid, the data from motor encoders become inexact. Therefore, the data from other sources and localization algorithms such as adaptive Monte Carlo localization (amcl) are then necessary to ensure the reliable localization. In addition, the planners including the global and local planners endow the robot with the cognition ability and an effective base controller for the robot is the key to the success in motion control.

Now we detail the navigation stack by discussing important ROS packages used in the navigation stack setup.

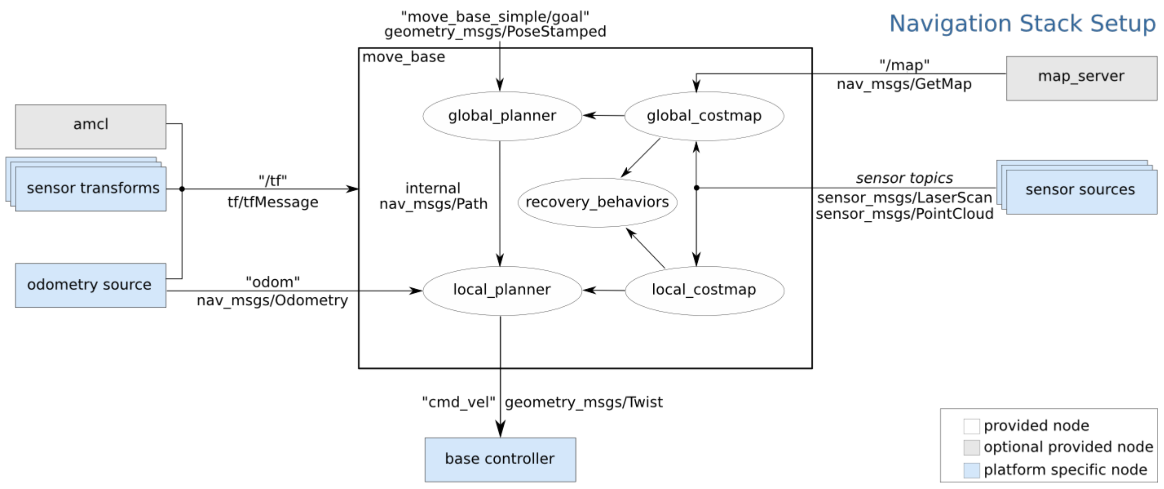


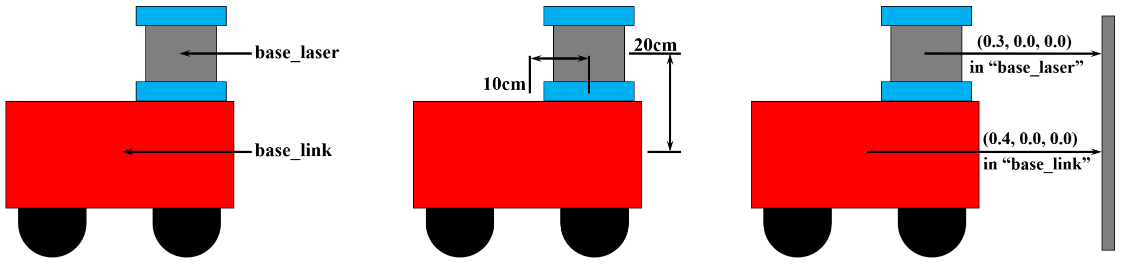
Figure 2.1: Navigation stack setup [1].

- ***TF package*** [43]

  TF package facilitates users to track coordinate frames. On the one hand, it can maintain static transformation relationships between coordinate frames. We take a simple robot as an example for illustration as shown in Figure 2.2(a). This mobile robot consists of two parts: A mobile base whose attached coordinate frame is named "base_link" and a laser whose attached frame is named "base_laser". The laser used to observe the environment is fixed on the mobile base in a manner shown in Figure 2.2(b). Figure 2.2(c) shows a normal situation in navigation where the laser detects an object at a distance of 0.3 m ahead of it. However, under such a circumstance, the mobile base which is the only controllable part of robot is not aware of the obstacle if the transformation relationship between frame "base_link" and frame "base_laser" are not built. Therefore, we need to configure the static transformation between these two frames. Then the mobile base knows that there is an obstacle at a distance of 0.4 m ahead of the robot base and can make moves to avoid the collision with

[1] `http://wiki.ros.org/navigation/Tutorials/RobotSetup?action=AttachFile&do=view&target=overview_tf.png`

the obstacle.



(a) (b) (c)

Figure 2.2: An example for illustrating configuring transform using TF package[2].

On the other hand, TF package can be used to describe dynamical transformation relationships. This contributes to publishing odometry information in navigation. Generally, for a short-term navigation, we name the coordinate frame attached to the mobile robot base as “base_link” and the world-fixed reference frame computed from odometry sources as “odom” frame. As both frames can be tracked by TF, the dynamic transformation from “odom” frame to “base_link” frame helps a robot localize itself in autonomous navigation.

- ***gmapping package*** [44]

  As aforementioned, one technique closely related to autonomous navigation is SLAM. SLAM requires a robot to be able to build a map of an unknown environment and meanwhile to be aware of its position in the environment. In other words, a robot with high intelligence can construct and update the map of its working environment during its autonomous navigation. Therefore, the robot does not need a map server to provide a map in advance which makes “map_server” an optional node in the navigation stack setup as shown in Figure 2.1. However, if a map created before navigation is not given, then the global costmap used for global path planning is needed and built using the data from

[2] `http://wiki.ros.org/navigation/Tutorials/RobotSetup/TF`

sensor sources such as lasers and cameras during the navigation. This requires a high-performance sensor system. As our robot does not possess a considerably high performance sensor system, we choose the map-based navigation approach, i.e., we build a map before navigation and need a map server to provide the map when the navigation starts.

Among the ROS packages which integrate SLAM algorithms, gmapping is one of the most widely-used and mature packages. It integrates a laser-based SLAM algorithm which is based on *Rao-Blackwellized particle filter*. Table 2.1 lists the topics that gmapping subscribes to and publishes. From the table, we can see that essentially gmapping makes use of the odometry information and data from laser scans to generate a map of the environment. The created map is a 2-D occupancy grid map, and an example is shown in Figure 2.3. The TF transforms related to gmapping package are listed in Table 2.2. Additionally, the configurable parameters in gmapping are listed in Table 2.3.

Table 2.1: Topics in gmapping package.

	Name	Type	Description
Subscribed Topics	tf	tf/tfMessage	Transforms between laser frame and "base_link" frame, also between "base_link" frame and "odom" frame
	scan	sensor_msgs/LaserScan	Data from laser scans
Published Topics	map_metadata	nav_msgs/MapMetaData	Meta data of map
	map	nav_msgs/OccupancyGrid	Data of grid map
	~entropy	std_msgs/Float64	Estimate of the entropy of the distribution over the robot pose

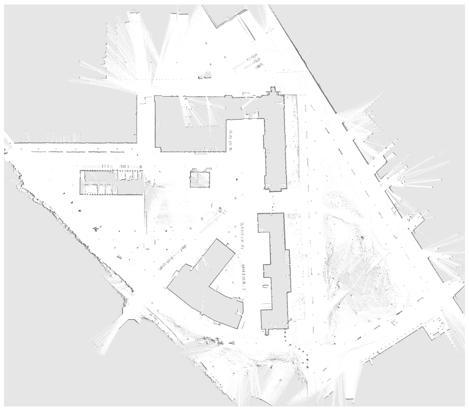


Figure 2.3: An example of 2-D occupancy grid map[3].

Table 2.2: TF transforms related to gmapping package.

	TF transforms	Description
Required TF Transforms	⟨laser scan frame⟩ → “base_link”	Transform between laser frame and “base_link” frame, usually published by the node robot_state_publisher or static_transform_publisher
	“base_link” → “odom”	Transform between “base_link” frame and “odom” frame, usually published by odometry system
Published TF Transform	“map” → “odom”	Transform between “map” frame and “odom” frame, used to estimate the robot pose in the map

- ***move_base package*** [45]

  The move_base package is the core of the navigation stack setup. It performs the path planning to let the robot reach a given navigation goal by using a global planner and a local planner. Each planner uses its own corresponding costmap to plan a path for the mobile robot base. Specifically, the global planner computes a global path based on the given goal and a global costmap. The

[3] `http://www2.informatik.uni-freiburg.de/~stachnis/research/rbpfmapper/gmapper-web/freiburg-campus/fr-campus-20040714.carmen.gfs.png`

Table 2.3: Key parameters in gmapping package.

Paramter	Type	Default value	Description
~throttle_scans	int	1	Process 1 out of n scans where n is the set value
~base_frame	string	"base_link"	Frame attached to robot base
~map_frame	string	"map"	Frame attached to map
~odom_frame	string	"odom"	Odometry frame
~map_update_interval	float	5.0	Time between two map updates (s)
~maxUrange	float	80.0	Maximum range that laser reaches (m)
~sigma	float	0.05	Standard deviation of endpoint matching
~kernelSize	int	1	Search in the nth kernel where n is the set value
~lstep	float	0.05	Step size of optimization in translational movement
~astep	float	0.05	Step size of optimization in rotational movement
~iterations	int	5	Number of iterations of scan matching
~linearUpdate	float	1.0	Translational distance between each laser scan (m)
~angularUpdate	float	0.5	Rotational distance between each laser scan (rad)
~temporalUpdate	float	-1.0	If the processing speed of the latest scan is less than the speed of update, process one scan. Stop the time-based updates if the value is negative
~particles	int	30	Number of particles in the filter
~xmin	float	-100.0	Initial map size (m)
~ymin	float	-100.0	
~xmax	float	100.0	
~ymax	float	100.0	
~delta	float	0.05	Resolution of the map (m/grid block)
~transform_publish_period	float	0.05	Time between two TF transform publications (s)
~occ_thresh	float	0.25	Threshold value of occupancy rate for map

global planning utilizes the *Dijkstra* or *A** algorithm to compute the optimal path from the current robot position to the target position and outputs this optimal path to the local planner. In many cases, the robot cannot strictly follow the global path due to physical limits and unexpected obstacles. This leads to the requirement of the local planner. The local planner takes inputs including the global path, local costmap and odometry information to plan a local path to be close to the global path as much as possible. Also the local planner takes the obstacles that may appear at any time into consideration by using the *Trajectory Rollout* and *Dynamic Window Approach (DWA)* algorithms. Below we summarize the working principles of the Trajectory Rollout and DWA algorithms as these two are fundamental to enable the ability to avoid collisions with obstacles for the robot during autonomous navigation.

The working principles of Trajectory Rollout and DWA algorithms can be divided into five steps:

1. Discretize the set of achievable velocities for the robot, thus many pairs with each consisting of translational and rotaional velocities can be formed, and each pair would result in a trajectory for the robot.

2. Determine the closest obstacle on each trajectory. Predict if the robot is able to stop without causing collisions with applying the velocities. Discard the pairs of velocities that violate.

3. Further discard the pairs of velocities that robot cannot reach due to the limitations in the accelerations.

4. Formulate an objective function which incorporates three terms. The first term is related to the effortlessness to the goal position, i.e., it reaches maximum when the robot can move straight to the goal. The second one

is related to the distance to the closest obstacle on the trajectory. Last term is related to the forward velocity of the robot.

5. Find the velocity pair that maximizes the objective function as this pair makes the robot reach the goal with the least effort, highest efficiency to handle obstacles, and shortest time. Send the velocities to the mobile base.

Trajectory Rollout and DWA algorithms share common traits, however, differ in how they discretely choose samples from the set of achievable velocities. As mentioned in the second step, both algorithms need to perform prediction, thus a prediction horizon is needed and a step size for sampling is also needed for discretization. The difference lies in that Trajectory Rollout samples over the whole prediction horizon, however DWA samples for only one sampling step. In fact, due to the requirement of real-time performance, both the prediction horizon and step size for sampling are set to a short period of time. This results in the comparable performance between Trajectory Rollout and DWA algorithms in many applications.

Next we discuss the action application programming interface (API), related topics, services, and configurable parameters in move_base which are listed in Table 2.4.

The action API is based on the *actionlib stack*. This leads to that besides the standard subscribed and published topics, move_base also possesses the action subscribed and published topics. Specifically, the user can use the SimpleActionClient and configure the move_base as SimpleActionServer if intending to track the execution status of move_base, otherwise simply use the standard API. The published topic of the move_base is the desired velocity consisting of the translational velocity and rotational velocity along the x-axis, y-axis and z-axis.

Table 2.4: Action API, topics and services in move_base package.

	Name	Type	Description
Action Subscribed Topics	move_base/goal	move_base_msgs/ MoveBaseActionGoal	Goal for move_base to reach
	move_base/cancel	actionlib_msgs/GoalID	Request to cancel a specified goal
Action Published Topics	move_base/feedback	move_base_msgs/ MoveBaseActionFeedback	Feedback that contains coordinate of mobile base
	move_base/status	actionlib_msgs/ GoalStatusArray	Information on status of goals sent to move_base
	move_base/result	move_base_msgs/ MoveBaseActionResult	No result for operation of move_base
Subscribed Topics	move_base_simple/goal	geometry_msgs/PoseStamped	Provide a non-action interface for users which do not necessarily need to track the execution status of goals
Published Topics	cmd_vel	geometry_msgs/Twist	Signal that contains desired velocity sent to mobile base
Services	∼make_plan	nav_msgs/GetPlan	Allow users to ask for the path plan to reach a given goal from move_base without making move_base execute the plan
	∼clear_unknown_space	std_srvs/Empty	Allow users to directly clear the unknown space aorund the robot
	∼clear_costmaps	std_srvs/Empty	Allow users to command move_base to clear the obstacles in the costmaps

And the positive x-axis points the forward direction of the robot, positive y-axis points left, and positive z-axis points up. As shown in Figure 2.1, the base controller of a robot receives the desired velocity and converts it to the control signals to the controllable parts in the robot, e.g., driven wheels for a wheeled ground robot. Next we list the parameters used to configure move_base in the Table 2.5.

- ***amcl package*** [46]

  The amcl package provides an approach to realizing the localization for the robot. However, it is not the only solution to localization. As aforementioned, gmapping package integrates a SLAM algorithm, and thus it can also be used for localization. This makes the amcl package an optional node in the navigation stack setup. Despite other localization algorithms such as gmapping, amcl plays a leading role in the map-based navigation due to its strong connection with the pre-given map. The amcl package takes the map, laser scans and necessary transforms as inputs to give an estimated robot pose as the output by using a particle filter. The topics and services in amcl package are listed in Table 2.6.

  From the subscribed topics of amcl, it can be seen that localization with amcl relies on the proper configuration of three key components: The particle filter, laser scans, and odometry transforms. As a result, the configurable parameters in amcl can be divided into three categories: Parameters of the laser model, the filter, and the odometry model which are shown in Table 2.7, Table 2.8 and Table 2.9, respectively.

Table 2.5: Parameters in move_base package.

Paramter	Type	Default value	Description
~base_global_planner	string	"navfn/NavfnROS"	Name of plugin for global planner used in move_base
~base_local_planner	string	"base_local_planner/ TrajectoryPlannerROS"	Name of plugin for local planner used in move_base
~recovery_behaviors	list	[{name:conservative_reset, type:clear_costmap_recovery/ ClearCostmapRecovery}, {name:rotate_recovery, type:rotate_recovery/Rotate-Recovery}, {name:aggressive_reset, type:clear_costmap_recovery/ ClearCostmapRecovery}]	A list of plugins for recovery behaviors of move_base. When move_base fails to plan an effective path, it will start the recovery behavior in the order of this list until making a plan. Otherwise it will consider the goal unreachable and abort the mission
~controller_frequency	double	20.0	Frequency of move_base sending velocity command to the mobile base (Hz)
~planner_patience	double	5.0	Time for planner to wait for an effective plan before operation of clearing space (s)
~controller_patience	double	15.0	Time for controller to wait for an effective control signal before operation of clearing space (s)
~conservative_reset_dist	double	3.0	Obstacles within this range will be cleared in the costmap when operation of clearing space is performed (m)
~recovery_behavior_enabled	bool	true	Whether to enable recovery behavior for move_base to attempt to clear space or not
~clearing_rotation_allowed	bool	true	Whether to let mobile base attempt to rotate in-place when operation of clearing space is performed or not
~shutdown_costmaps	bool	false	Whether to shutdown costmaps when move_base becomes inactive
~oscillation_timeout	double	0.0	Time allowed for oscillation before executing recovery behaviors (s)
~oscillation_distance	double	0.5	Robot should move this far, otherwise is considered to be oscillating. Moving this far will reset the timer counting up to the parameter ~oscillation_timeout (m)
~planner_frequency	double	0.0	Frequency for loop of global planning (Hz). If the value is set to be 0.0, global planner will be used when receiving a new goal or local planner reports a invalid path
~max_planning_retries	int32_t	-1	Times allowed for the re-planning before executing recovery behaviors (s). A value of -1 represents infinite times of re-planning

Table 2.6: Topics and services in amcl package.

	Name	Type	Description
Subscribed Topics	scan	sensor_msgs/LaserScan	Data from laser scans
	tf	tf/tfMessage	Information on transforms of coordinate frames
	initialPose	geometry_msgs/ PoseWithCovarianceStamped	Mean and covariance used to initialize particle filter
	map	nav_msgs/OccupancyGrid	When the parameter use_map_topic is set to be true, this topic is subscribed to be used for laser-based localization
Published Topics	amcl_pose	geometry_msgs/ PoseWithCovarianceStamped	Estimate of robot pose in the map with covariance
	particlecloud	geometry_msgs/PoseArray	Set of estimated poses being maintained by the filter
	tf	tf/tfMessage	Transform from odom frame to map frame
Services	global_localization	std_srvs/Empty	Initialize the global localization. All particles are randomly spread in free space of the map
	request_nomotion_update	std_srvs/Empty	Manually perform update and set new particles
Services Called	static_map	nav_msgs/GetMap	Amcl calls this service to receive map data

Table 2.7: Laser model parameters in amcl package.

Paramter	Type	Default value	Description
~laser_min_range	double	-1.0	Minimum range for laser scans (m)
~laser_max_range	double	-1.0	Maximum range for laser scans (m)
~laser_max_beams	int	30	Number of evenly-spaced beams used in each scan when updating filter
~laser_z_hit	double	0.95	Mixture parameter for z_hit, z_short, z_max, and z_rand part of model
~laser_z_short	double	0.1	
~laser_z_max	double	0.05	
~laser_z_rand	double	0.05	
~laser_sigma_hit	double	0.2	Standard deviation for Gaussian model used in z_hit part of model
~laser_lambda_short	double	0.1	Parameter for exponential decay for z_hit part of model
~laser_likelihood_max_dist	double	2.0	Maximum distance to measure inflation of obstacles (m)
~laser_model_type	string	"likelihood_field"	Choice for model including beam, likelihood_field and likelihood_field_prob

Table 2.8: Overall filter parameters in amcl package.

Paramter	Type	Default value	Description
∼min_particles	int	100	Number of minimum particles allowed
∼max_particles	int	5000	Number of maximum particles allowed
∼kld_err	double	0.01	Maximum error between true and estimated distribution
∼kld_z	double	0.99	The upper standard normal quantile for (1-p) where p is the probability that estimated distribution error is less than the value for parameter kld_err
∼update_min_d	double	0.2	Translational distance required for filter to perform an update (m)
∼update_min_a	double	$3.0/\pi$	Rotational angle required for filter to perform an update (rad)
∼resample_interval	int	2	Number of updates for filter before re-sampling
∼transform_tolerance	double	0.1	Time to publish a transform, to indicate this transform is valid in the future (s)
∼recovery_alpha_slow	double	0.0	Rate of exponential decay for slow average weighted filter, used to decicde when to add random poses to recover, 0.0 represents disablement
∼recovery_alpha_fast	double	0.0	Rate of exponential decay for fast average weighted filter, used to decicde when to add random poses to recover, 0.0 represents disablement
∼initial_pose_x	double	0.0	Mean of x (m), y (m), and yaw (rad), and covariance of x*x (m), y*y (m), and yaw*yaw (rad) in initial pose, used to initialize Gaussian distribution based filter
∼initial_pose_y	double	0.0	
∼initial_pose_a	double	0.0	
∼initial_cov_xx	double	0.5 * 0.5	
∼initial_cov_yy	double	0.5 * 0.5	
∼initial_cov_aa	double	$(\pi/12) * (\pi/12)$	
∼gui_publish_rate	double	-1.0	Maximum rate of publishing information on visualization (Hz), -1.0 represents disablement
∼save_pose_rate	double	0.5	Maximum rate of storing the estimated pose and covariance in parameter server (Hz), used to subsequently initialize the filter. -1.0 represents disablement
∼use_map_topic	bool	false	When set to be true, amcl subscribes to map topic instead of receiving map from server
∼use_map_only	bool	false	When set to be true, amcl only uses the first map it subscribes to instead of the subsequent updated map

Table 2.9: Odometry model parameters in amcl package.

Paramter	Type	Default value	Description
∼odom_model_type	string	“diff”	Choice for model including diff, omni, diff-corrected, and omni-corrected
∼odom_alpha1	double	0.2	Expected noise in the estimate of odometry’s rotation based on the rotational component of robot motion
∼odom_alpha2	double	0.2	Expected noise in the estimate of odometry’s rotation based on the translational component of the robot motion
∼odom_alpha3	double	0.2	Expected noise in the estimate of odometry’s translation based on the translational component of the robot motion
∼odom_alpha4	double	0.2	Expected noise in the estimate of odometry’s translation based on the rotational component of the robot motion
∼odom_alpha5	double	0.2	Parameter for noise related to translation (only for omni)
∼odom_frame_id	string	“odom”	Coordinate frame for odometry system
∼base_frame_id	string	“base_link”	Coordinate frame for mobile base
∼global_frame_id	string	“map”	Coordinate frame which the localization system publishes
∼tf_broadcast	bool	true	When set to be false, amcl will not publish the transform between the global and odom frame

As we have finished the introduction to ROS, we are in the position of presenting the work on simulating autonomous navigation for a two-wheeled differential robot in the ROS environment.

2.2 A URDF-based model of a two-wheeled differential robot

To start the simulation of the autonomous navigation for our robot, we need to first model the robot in the ROS environment. Generally, the *Unified Robot Description Format* (URDF) is used to describe robot models in ROS. It is an *Extensible Markup Language* (XML) format and adopted to describe properties such as robot appearance, physical properties, and joint types. However, URDF does not possess the feature of code reusability, and thus becomes inefficient when adopted to describe considerably complex robots. As a result, a URDF-based format with higher efficiency is further developed, namely *xacro*. Compared to URDF, xacro supports the declaration of constant variables and code reuse by the creation of macros, and its programming provides APIs such as variables, mathematical formula, conditional statement. We also adopt the xacro format to build the robot model for simulation. Next we will detail the building process.

2.2.1 Basic visual model of the mobile base

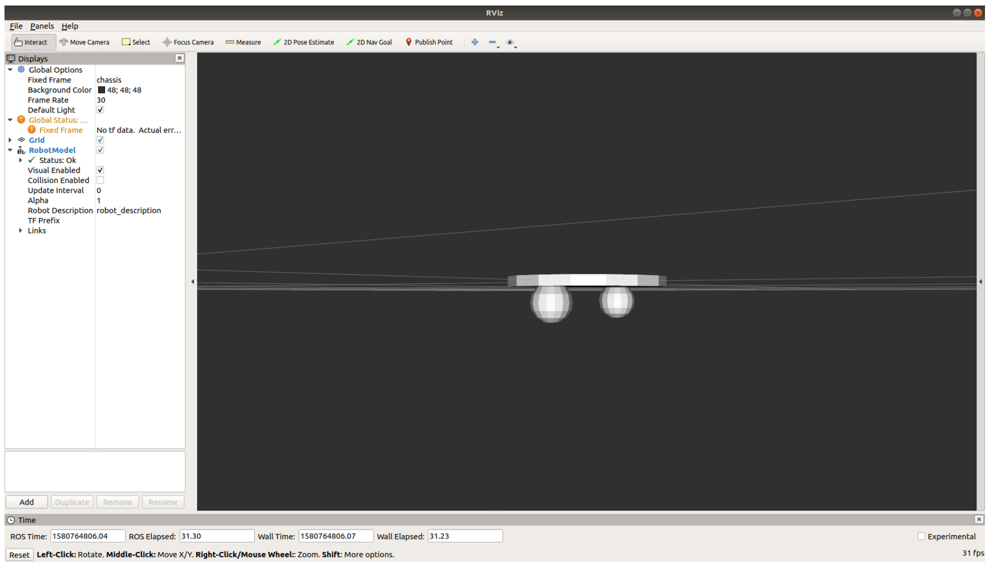
We first name our simulated robot “simbot”, describe the robot model in the file named “simbot.xacro”, and put the configuration parameters related to the robot in the file named “parameters.xacro”. Thus by calling the file “parameters.xacro” in the file “simbot.xacro”, the value of parameters can be reused.

Then we start to build our simulated robot model with the chassis of the robot and two casters which are omnidirectional wheels to support the chassis. Here we use the ⟨link⟩ label to describe the chassis along with the two casters due to the direct contact between casters and chassis. Another label used is the ⟨visual⟩ to define the appearance properties including the 3-D coordinate position, rotation pose, and specified shape. The detailed properties are shown in Table 2.10.

Table 2.10: Properties of the chassis and casters.

Property	Value
Origin of chassis	0 0 0
Roll, pitch and yaw of Chassis	0 0 0
Width of chassis (m)	0.025
Radius of chassis (m)	0.2
Origin of front caster	0.15 0 -0.05
Origin of back caster	-0.15 0 -0.05
Radius of caster (m)	0.05

The model of the chassis and its attached casters is shown in Figure 2.4. It uses rviz which is the 3-D platform for visualization in the ROS environment.



Global Options	
Fixed Frame	chassis
Background Color	■ 48; 48; 48
Frame Rate	30
Default Light	✓
Global Status: ...	
Fixed Frame	No tf data. Actual err...
Grid	✓
RobotModel	✓
✓ Status: Ok	
Visual Enabled	✓
Collision Enabled	☐
Update Interval	0
Alpha	1
Robot Description	robot_description
TF Prefix	
Links	

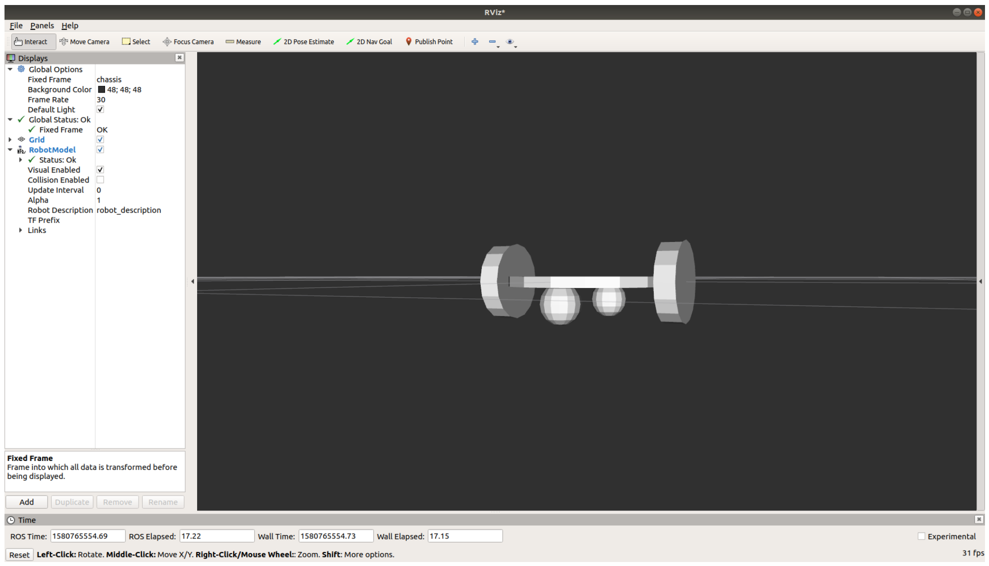
ROS Time: 1580764806.04 ROS Elapsed: 31.30 Wall Time: 1580764806.07 Wall Elapsed: 31.23 ☐ Experimental

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click/Mouse Wheel:**: Zoom. **Shift**: More options. 31 fps

Figure 2.4: The model of the chassis and casters in rviz.

Then we describe the wheels to complete the model of the robot mobile base. To connect the wheels to the chassis, two revolute joints are needed and thus should also be described. The ⟨link⟩ label is again used to describe the left and right wheels which are both in the shape of cylinder. For the hinge joints, we use the ⟨joint⟩ label and choose the type as continuous, which makes the two joints both in the revolute type. And the wheels can rotate infinitely. The types allowed to set for ⟨joint⟩ label are listed in Table 2.11. And the child and parent links for the left and right wheel hinge joints are set to be the corresponding wheel and the chassis, respectively. This enables the physical connection from the chassis, to the wheel hinge joints, to the wheels. In addition, the ⟨axis⟩ label defines the rotation axis for the joints and the ⟨limit⟩ label describes the limits of motion including the most upper and lowest position, velocity and torque limits. The detailed properties of wheels and wheel joints are shown in Table 2.12.

The model of the robot mobile base is shown in Figure 2.5.



Global Options	
Fixed Frame	chassis
Background Color	■ 48; 48; 48
Frame Rate	30
Default Light	✓
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
RobotModel	✓
✓ Status: Ok	
Visual Enabled	✓
Collision Enabled	
Update Interval	0
Alpha	1
Robot Description	robot_description
TF Prefix	
Links	

Fixed Frame
Frame into which all data is transformed before being displayed.

```
ROS Time: 1580765554.69   ROS Elapsed: 17.22   Wall Time: 1580765554.73   Wall Elapsed: 17.15
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.   31 fps
```

Figure 2.5: The model of the robot mobile base in rviz.

Table 2.11: Types for the ⟨joint⟩ label in URDF-based model.

Label Type	Joint Type	Specification
continuous	Revolute joint	Allow infinite rotation about a single axis
revolute	Revolute joint	Similarly to continuous type, however set angular limit for rotation
prismatic	Prismatic joint	Allow translational movement along a single axis with limited positions
planar	Planar joint	Allow translational or rotational movement in orthogonal direction of a plane
floating	Floating joint	Allow translational and rotational movement
fixed	Fixed joint	Not allow any movement

Table 2.12: Properties of the chassis and casters.

Property	Value
Origin of chassis	0 0 0
Width of chassis (m)	0.025
Radius of chassis (m)	0.2
Origin of front caster	0.15 0 -0.05
Origin of back caster	-0.15 0 -0.05
Radius of caster (m)	0.05

2.2.2 Physical and collision properties

Now we have created the model of our robot mobile base which can be visualized. However, necessary physical and collision properties of each part need to be configured for autonomous navigation. Specifically, the physical properties and collision properties are described using the ⟨inertial⟩ label and ⟨collision⟩ label, respectively.

The physical properties of the chassis consist of two parts: The mass and rotational inertia matrix. The mass of the chassis is 1.5 kg. As the chassis is in the shape of regular cylinder, we can compute the inertial matrix by directly using the general formula shown in Equation 2.1 [47] where M, h, and R denote the mass, height and radius of the chassis, respectively. The content in the collision description is similar to that in the visual description due to the simple structure of the chassis. Similarly the physical and collision properties of two wheels can be configured.

$$\begin{bmatrix} ixx & ixy & ixz \\ iyx & iyy & iyz \\ izx & izy & izz \end{bmatrix} = \begin{bmatrix} \frac{1}{12}Mh^2 + \frac{1}{4}MR^2 & 0 & 0 \\ 0 & \frac{1}{12}Mh^2 + \frac{1}{4}MR^2 & 0 \\ 0 & 0 & \frac{1}{2}MR^2 \end{bmatrix}. \tag{2.1}$$

Since the two casters are considerably small and light, the moments of inertia can be assumed to be all zeros and the inertia property can be neglected. To describe the collision properties, we use the ⟨surface⟩ and ⟨friction⟩ labels to model the friction between the caster and the ground or obstacles if any collisions occur. As the surface of the caster is curved, the friction cannot be easily described without using physics engine provided by the URDF library. Typical physics engines include *ode*, *orsional*, and *bullet*. The default engine is *ode* which is also the most appropriate one to be used for a curved surface. For the ode physics engine, four parameters: ⟨mu⟩, ⟨mu2⟩, ⟨slip1⟩, and ⟨slip2⟩ need to be set. Both ⟨mu⟩ and ⟨mu2⟩ are set to be zero, which makes the surface of the caster completely smooth. Parameters ⟨slip1⟩ and ⟨slip2⟩ are related to the forces which cause the slippery of casters in the horizontal and vertical direction, respectively.

In addition, the coefficients of damping and static friction of the two wheel joints are both set to be 1.0.

2.2.3 Sensor model

For a ground robot with the ability to perform autonomous navigation, visual sensors are indispensable for environment observation. Therefore, we need to add visual sensors to our simulated robot. Generally, to achieve autonomous navigation, an indoor ground robot is equipped with a RGB-D camera such as a Kinect camera or a Lidar such as RPLidar and Hokuyo scanning Lidar. Here we choose the Hokuyo Lidar due to its advantages of higher accuracy, faster response and lower computational complexity. To fix the Hokuyo Lidar to the robot, a joint is needed to link the Lidar to the chassis. In addition, we make use of the ROS plugin for the Hokuyo Lidar provided by the ROS community to facilitate the adding of the sensor to the simulated robot and its proper functioning.

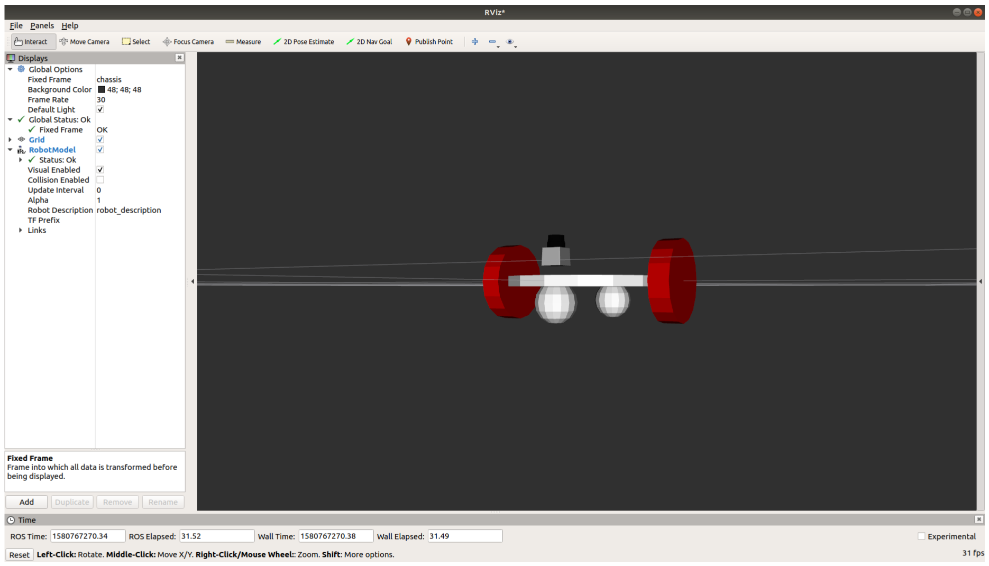
Though the Hokuyo Lidar is in a irregular shape, its collision model is configured as a box which is slightly larger than its original size. This simplification will decrease the computational demand and increase the smoothness of performing simulation. The properties of the Hokuyo Lidar and its linked joint are shown in Table 2.13. Now a URDF-based model with incorporating visual, physical and collision properties is built and shown in Figure 2.6.

2.2.4 Properties required by Gazebo

To simulate the autonomous navigation for our two-wheeled differential ground vehicle, we use the Gazebo which is a 3-D simulation platform for the ROS environment.

Table 2.13: Properties of the Hokuyo Lidar and its linked joint.

	Property	Value
Hokuyo joint	Axis	0 1 0
	Origin	0.15 0 0.07
	Roll, pitch and yaw angle	0 0 0
	Parent link	“chassis”
	Child link	“hokuyo”
Hokuyo Lidar	Origin	0 0 0
	Roll, pitch and yaw angle	0 0 0
	Size (m×m×m)	0.1×0.1×0.1
	Mass (kg)	10^{-5}
	Inertia matrix	diag(10^{-6}, 10^{-6}, 10^{-6})



```
Global Options
  Fixed Frame          chassis
  Background Color     48; 48; 48
  Frame Rate           30
  Default Light        ✓
Global Status: Ok
  Fixed Frame          OK
Grid                   ✓
RobotModel             ✓
  Status: Ok
  Visual Enabled       ✓
  Collision Enabled
  Update Interval      0
  Alpha                1
  Robot Description    robot_description
  TF Prefix
  Links
```

Fixed Frame
Frame into which all data is transformed before being displayed.

ROS Time: 1580767270.34 ROS Elapsed: 31.52 Wall Time: 1580767270.38 Wall Elapsed: 31.49

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click/Mouse Wheel:**: Zoom. **Shift**: More options. 31 fps

Figure 2.6: The model of a two-wheeled ground vehicle in rviz.

Although we have created a model of the ground robot which satisfies the basic requirement for simulation in Gazebo, it cannot move in Gazebo due to the lack of extensional properties needed by Gazebo. To add these properties, the ⟨gazebo⟩ label is used for necessary parts of robot. We put the configuration for these properties in the file named “simbot.gazebo” and declare the reuse of them in the “simbot.xacro” file similarly to the “parameters.xacro” file.

Since Gazebo is able to read the visual, physical and collision properties only except the material parameters configured in the visual property, we only need to set the material type, i.e., the color in the ⟨gazebo⟩ description for the chassis and two driven wheels.

Next, we need to configure the base controller for our robot. As aforementioned, the base controller takes the desired velocity as its input which in our case consists of two parts: The translational velocity along x-axis and rotational velocity about z-axis. The base controller is then indispensable for our robot since its input cannot directly drive the wheels. Here, the base controller is added using the Gazebo controller plugin. A typical controller plugin used for differential ground robots provided by Gazebo is named “libgazebo_ros_diff_drive.so”. It converts the desired linear velocity in the forward direction and planar angular velocity to the velocity of left and right wheels as:

$$\begin{aligned} V_{left} &= V_x - V_z * d/2 \\ V_{right} &= V_x + V_z * d/2, \end{aligned} \tag{2.2}$$

where V_{left} and V_{right} denote the linear velocities of the left and right wheels, respectively. And V_x and V_z represent the linear velocities in the forward direction and planar angular velocity about z-axis, and d is the distance between two wheels.

We directly call the plugin by configuring some parameters specified for our robot as shown in Table 2.14.

Table 2.14: Parameters for the base controller.

Parameter	Description	Value
updateRate	Update rate of sending control signal to robot (Hz)	30
leftJoint	Joint connected to left wheel	“left_wheel_hinge”
rightJoint	Joint connected to right wheel	“right_wheel_hinge”
wheelSeparation	Distance between two wheels (m)	0.46
wheelDiameter	Diameter of wheels (m)	0.2
torque	Maximum torque that wheels can operate (Nm)	10
commandTopic	Desired velocity command containing in a ROS topic that controller subscribes to	“cmd_vel”
odometryTopic	Odometry information containing in a ROS topic that controller publishes	“odom”
odomteryFrame	Odometry frame	“odom”
robotBaseFrame	Frame attached to mobile base	“chassis”

Finally, we add the plugin information about the Hokuyo Lidar as shown in Table 2.15.

Table 2.15: Parameters for the Hokuyo Lidar.

Parameter	Description	Value
sensor type	Type of sensor, there are two types for lasers: "gpu_ray" and "ray". The former utilizes GPU of master device and the latter does not.	"gpu_ray"
update_rate	Rate of updating one scan (Hz)	40
scan-samples	Number of samples for performing one scan	720
scan-resolution	Resolution of scan in horizontal direction (mm)	1
scan-min_angle	Scan angle (rad)	-1.570796
scan-max_angle		1.570796
range-min	Detection range (m)	0.10
range-max		30
range-resolution	Accuracy of detection	0.01 (1%)
noise-type	Type of noise exerted on the Lidar	"gaussian"
noise-mean	Mean of distribution for noise	0.0
noise-stddev	Standard deviation of noise distribution (m)	0.01
plugin-topicName	ROS topic which the ROS node linked to Lidar publishes	"/simbot/laserScan"
plugin-frameName	Frame attached to Hokuyo Lidar	"hokuyo"

Note that the parameter values for the simulated Hokuyo Lidar such as the update frequency, scanning resolution and range, and accuracy are set based on the values for the actual Hokuyo Lidar. However, the values can be changed to observe to what extent does the autonomous navigation rely on the performance of the Hokuyo Lidar. In addition, the ROS topic published by the ROS node linked to Hokuyo Lidar is named "/simbot/laserScan". It is also the subscribed topic for nodes used to localize the robot. And by using the ⟨material⟩ label to make our robot colored, we can display the model of our robot in Gazebo as shown in Figure 2.7.

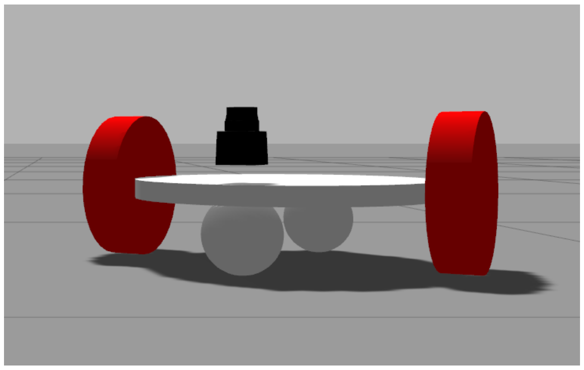


Figure 2.7: The model of a two-wheeled ground vehicle in Gazebo.

2.3 Simulation setup

2.3.1 Working environment

To simulate the autonomous navigation, we firstly create a simulated working environment. The environment can be built using *Gazebo Building Editor* tool. Figure 2.8 shows our robot model within the created working environment.

2.3.2 Setup for gmapping, move_base and amcl

Then we configure some parameters for the gmapping, move_base and amcl packages to realize the SLAM and navigation using our simulated robot model. The configured parameters to run the gmapping node are shown in Table 2.16.

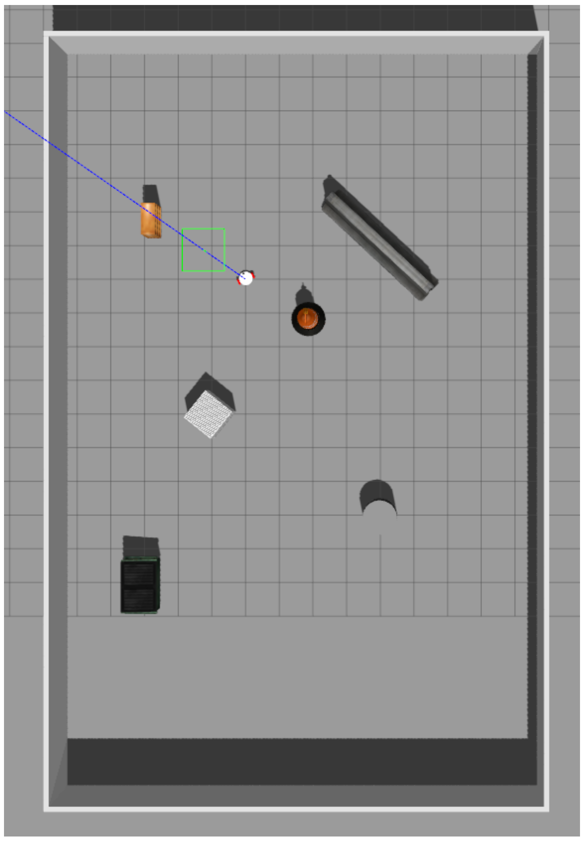


Figure 2.8: The two-wheeled UGV within its working environment in Gazebo.

Table 2.16: Parameter values in gmapping configuration.

Parameter	Description	Value
delta	Resolution of the map (m/grid block)	0.05
xmin	Map size (m)	-20
xmax		20
ymin		-20
ymax		20
base_frame	Frame attached to robot base	“chassis”
linearUpdate	Translational distance between each laser scan (m)	0.5
angularUpdate	Rotational distance between each laser scan (rad)	0.5
particles	Number of particles in the filter	80

Here, we set the size of created map as 20×20 m^2 instead of using the default value of 100×100 m^2 to accommodate the size of the simulated working environment. Additionally, it is worth noting the settings of two frames here. Firstly, the "base_frame" is set to be "chassis" to maintain the consistency of frame set in the URDF-based model. Secondly, since gmapping subscribes to the topic for lase scan named "scan", however the topic for the Hokuyo Lidar of our robot is named "simbot/laserScan", we then need to set the remapping from "scan" to "simbot/laserScan" with the ⟨remap⟩ label.

For the setup of the move_base package, three parts including the global and local costmaps, and the local planner need to be configured. The global and local costmaps share some common configuration shown in Table 2.17 and use exclusive configuration shown in Table 2.18 and Table 2.19, respectively.

Table 2.17: Adopted parameter values in common configuration for both costmaps.

Parameter	Description	Value
obstacle_range	Maximum range for robot to detect an obstacle (m)	1.5
raytrace_range	Maximum range for robot to clear out free space (m)	1.5
robot_radius	Radius of robot if setting the robot center as origin (m)	0.3
inflation_radius	Minimum safety distance between robot and obstacles (m)	0.5

Table 2.18: Adopted parameter values in global configuration for global costmap.

Parameter	Description	Value
global_frame	Coordinate frame the costmap runs in	"map"
robot_base_frame	Coordinate frame the costmap refers for the robot base	"chassis"
update_frequency	Update frequency of the costmap (Hz)	1.0
static_map	If the initialization of the costmap is based on the map served by the map_server	true

Table 2.19: Adopted parameter values in local configuration for local costmap.

Parameter	Description	Value
global_frame	Coordinate frame the costmap runs in	“odom”
update_frequency	Update frequency of the costmap (Hz)	3.0
publish_frequency	Frequency of costmap publishing visualization information (Hz)	1.0
static_map	If initialization of costmap is based on the map served by the map_server	false
rolling_window	If robot needs costmap to remain centered around robot	true
width	The size of the costmap (m)	6.0
height		6.0
resolution	Resolution of the costmap (m/cell)	0.01

Once the global costmap is set properly, the global planner does not need to be configured. The reason lies in that the *Dijkstra* and *A** algorithms utilized by the global planner can immediately compute the optimal path from the current robot position to the given goal by using the global costmap. Thus we do not necessarily need to configure the global planner due to its high efficiency. Nevertheless, the local planner is faced with much more complex situations than the global planner and needs to be configured properly to achieve collision-free navigation. The configuration of the local planner is shown in Table 2.20 below.

At last, we set parameters for the amcl node as shown in Table 2.21. Apart from the settings for the remapping of laser scan topic, and necessary coordinate frames, we set the triggering conditions of updating the localization.

Table 2.20: Adopted parameter values in base local planner configuration.

Parameter	Description	Value
acc_lim_x	The x acceleration limit of the robot (m/s^2)	0.5
acc_lim_y	The y acceleration limit of the robot (m/s^2)	0.5
acc_lim_theta	Rotational acceleration limit of the robot (rad/s^2)	1.5
max_vel_x	Maximum forward velocity allowed for the base (m/s)	0.5
min_vel_x	Minimum forward velocity allowed for the base (m/s)	0.01
max_vel_theta	Maximum rotational velocity allowed for the base (rad/s)	1.5
min_in_place_vel_theta	Minimum rotational velocity allowed for the base when performing in-place rotations (rad/s)	0.01
escape_vel	Speed used for driving during escapes (m/s)	-0.12
holonomic_robot	If the robot is holonomic	false

Table 2.21: Parameter values in amcl configuration.

Parameter	Description	Value
odom_frame_id	Coordinate frame for odometry system	“odom”
odom_model_type	Type of model for odometry system	“diff”
base_frame_id	Coordinate frame for mobile base	“chassis”
update_min_d	Translational distance required for filter to perform an update (m)	0.5
update_min_a	Rotational angle required for filter to perform an update (rad)	1.0

2.4 Simulation results

2.4.1 Creating a map of the environment

As mentioned in Section 2.1.2, we aim to realize the map-based autonomous navigation for our robot. Thus we need to create a map of the working environment in advance of performing navigation. We first initialize the simulated environment and robot model and run the gmapping node with pre-set configuration.

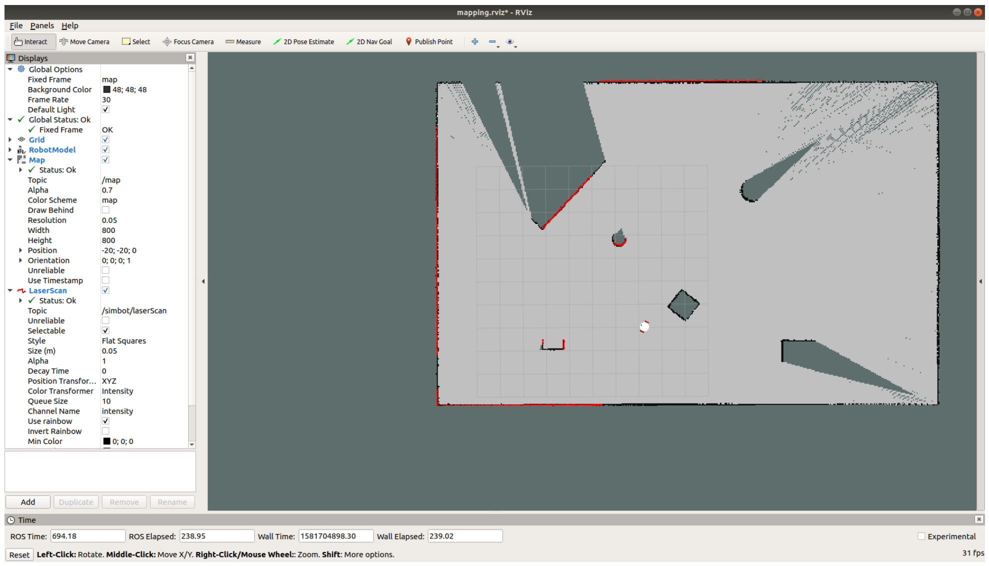
Then we run the teleoperation node to control the robot to move in the working environment. The teleoperation node is an existing node for controlling the sample robots such as the turtlebot, thus here we can directly use it by changing some parameters and remapping its original published topic "turtlebot3_teleop_keyboard/cmd_vel" to the corresponding topic for our robot "cmd_vel". Figure 2.9 illustrates an intermediate state of the map creation process in rviz.

And the obtained map of the working environment is shown in Figure 2.10 which is named "map.pgm". We also obtain its configuration file named "map.yaml" which will be used by the map server at the beginning of the autonomous navigation.

We can see that though there are some overlaps between the free space and objects, the map illustrates the positions of objects in the working environment. Specifically, the outlines of objects are precisely displayed, which can let the robot easily identify the objects during the autonomous navigation. Therefore, the quality of the created map is acceptable.

2.4.2 Autonomous navigation

With the obtained map, we can start the autonomous navigation for our ground vehicle. Similarly to Section 2.4.1, we first initialize the working environment along with our robot model. And then we need to start the map server and move_base.



Global Options	
Fixed Frame	map
Background Color	■ 48; 48; 48
Frame Rate	30
Default Light	✓
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
RobotModel	✓
Map	✓
✓ Status: Ok	
Topic	/map
Alpha	0.7
Color Scheme	map
Draw Behind	☐
Resolution	0.05
Width	800
Height	800
Position	-20; -20; 0
Orientation	0; 0; 0; 1
Unreliable	☐
Use Timestamp	☐
LaserScan	✓
✓ Status: Ok	
Topic	/simbot/laserScan
Unreliable	☐
Selectable	✓
Style	Flat Squares
Size (m)	0.05
Alpha	1
Decay Time	0
Position Transfor...	XYZ
Color Transformer	Intensity
Queue Size	10
Channel Name	intensity
Use rainbow	✓
Invert Rainbow	☐
Min Color	■ 0; 0; 0

ROS Time: 694.18 ROS Elapsed: 238.95 Wall Time: 1581704898.30 Wall Elapsed: 239.02 ☐ Experimental

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click/Mouse Wheel:**: Zoom. **Shift**: More options. 31 fps

Figure 2.9: An intermediate state of the map creation process in rviz.

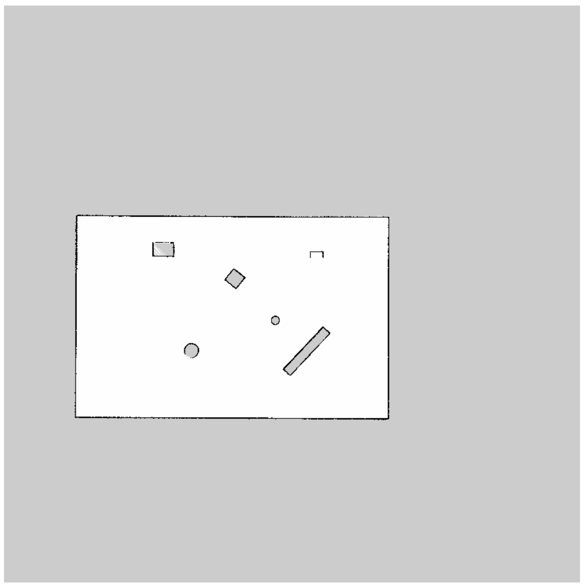


Figure 2.10: The created map of the simulated working environment.

To perform the autonomous navigation, a series of target points for the vehicle to reach in order is set as shown in Figure 2.11. In other words, the robot is required to reach position 1, then positions 2, 3 and so on. After arriving at position 6, it again goes to position 1 and continues the navigation task until we end the task.

There are two interfaces when the autonomous navigation is performed: The rviz and Gazebo as shown in Figure 2.12. The rviz interface shown in Figure 2.12(a) enables us to see the planned paths generated by the global and local planners, and to send additional navigation goals to our robot with the “2D Nav Goal” button. The Gazebo interface shown in Figure 2.12(b) is for observing the actual robot states in the working environment.

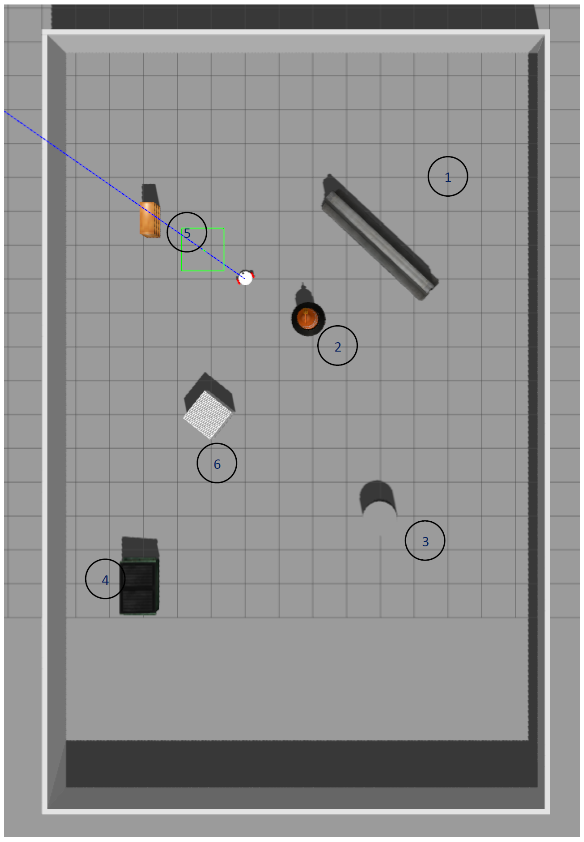
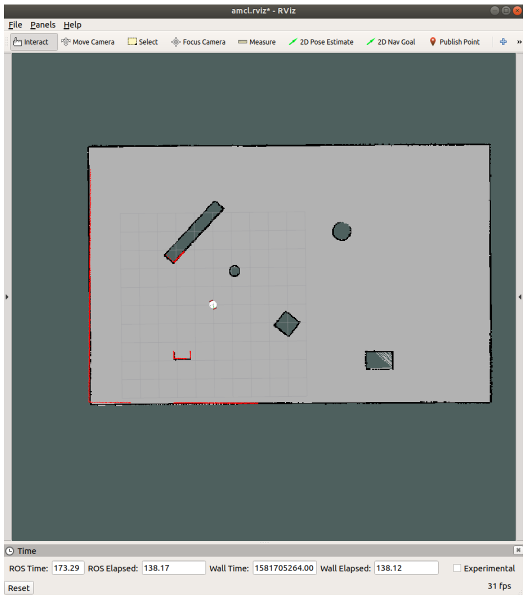


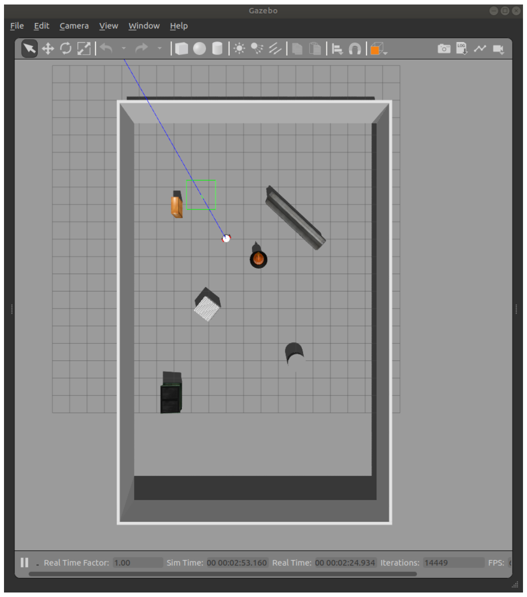
Figure 2.11: Six target waypoints for the robot to reach.


amcl.rviz* - RViz
File Panels Help
Interact Move Camera Select Focus Camera Measure 2D Pose Estimate 2D Nav Goal Publish Point
Time
Experimental
Reset

ROS Time: 173.29 ROS Elapsed: 138.17 Wall Time: 1581705264.00 Wall Elapsed: 138.12

31 fps

(a) The rviz interface.



Real Time Factor: 1.00 Sim Time: 00 00:02:53.160 Real Time: 00 00:02:24.934 Iterations: 14449 FPS:

(b) The Gazebo interface.

Figure 2.12: Two interfaces used during the simulation.

Figure 2.13 shows the robot movement towards position 1. The red line segments in rviz show the results of laser scans using the Hokuyo Lidar. The blue curve is the global path computed by the global planner and the short green curve represents the local path computed by the local planner.

To achieve a given goal, say reaching position 1 for example, the robot will plan feasible global and local paths to the target point with avoiding colliding with the objects. And after reaching the target point, it will perform in-place rotation until achieving the desired pose. This process can be shown in Figures 2.14, 2.15 and 2.16.

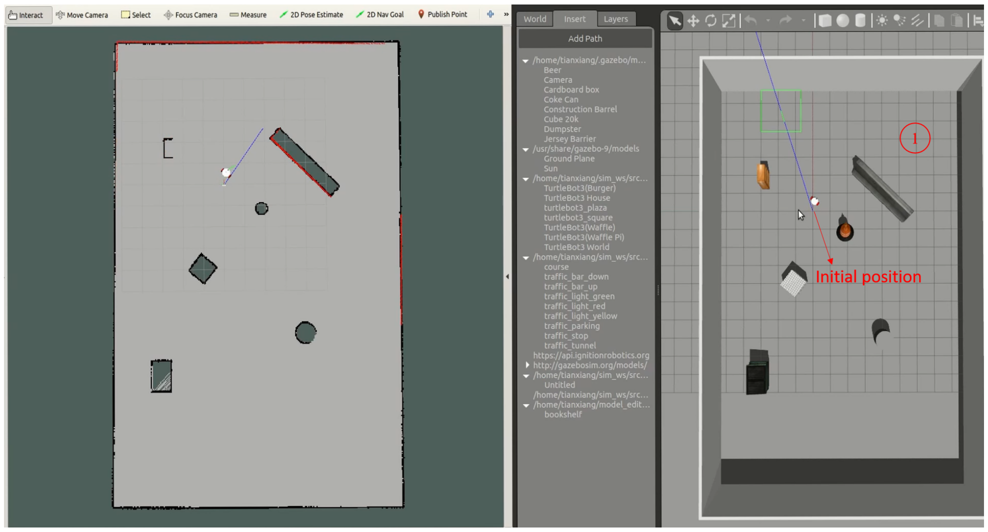


Figure 2.13: An intermediate state of robot moving towards position 1.

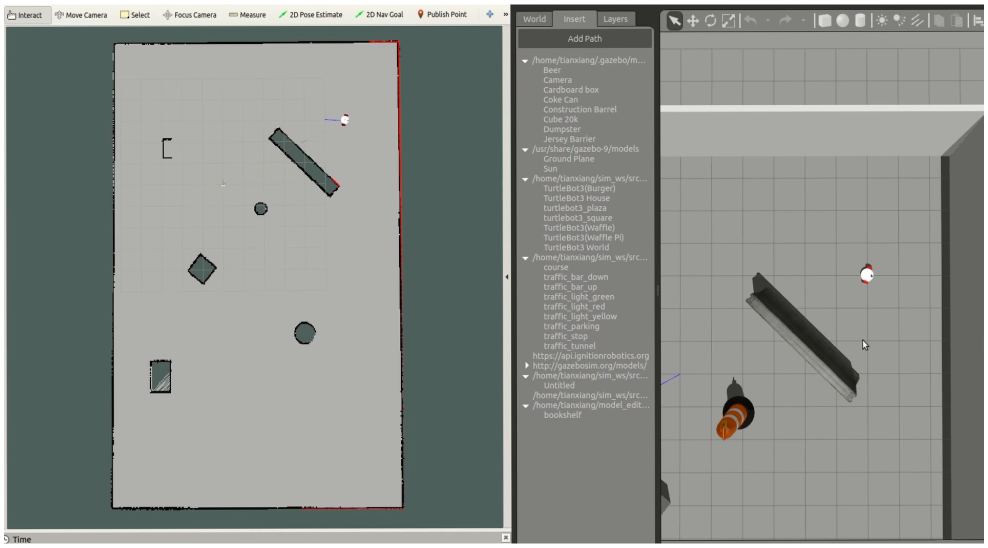


Figure 2.14: The robot reaches the position 1.

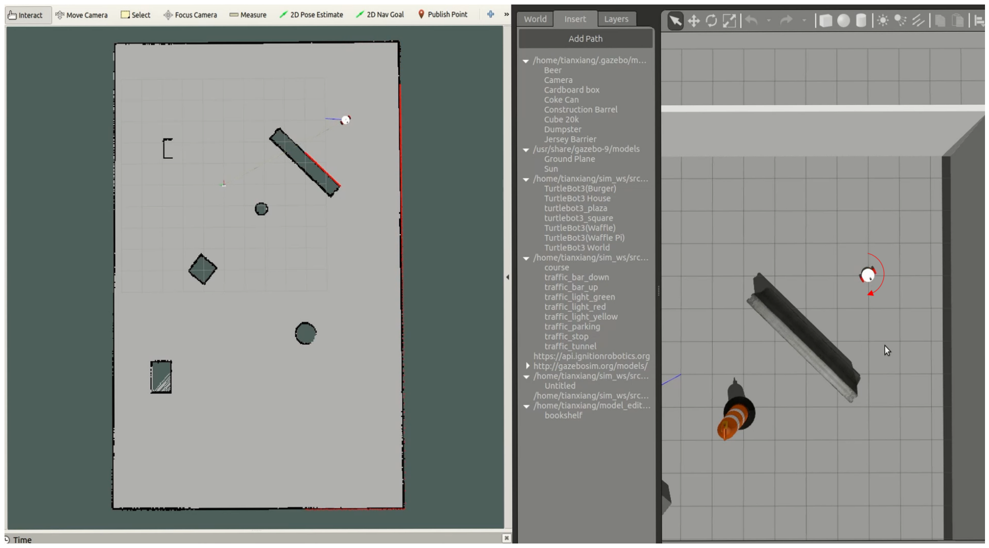


Figure 2.15: The in-place rotation of the UGV after reaching position 1.

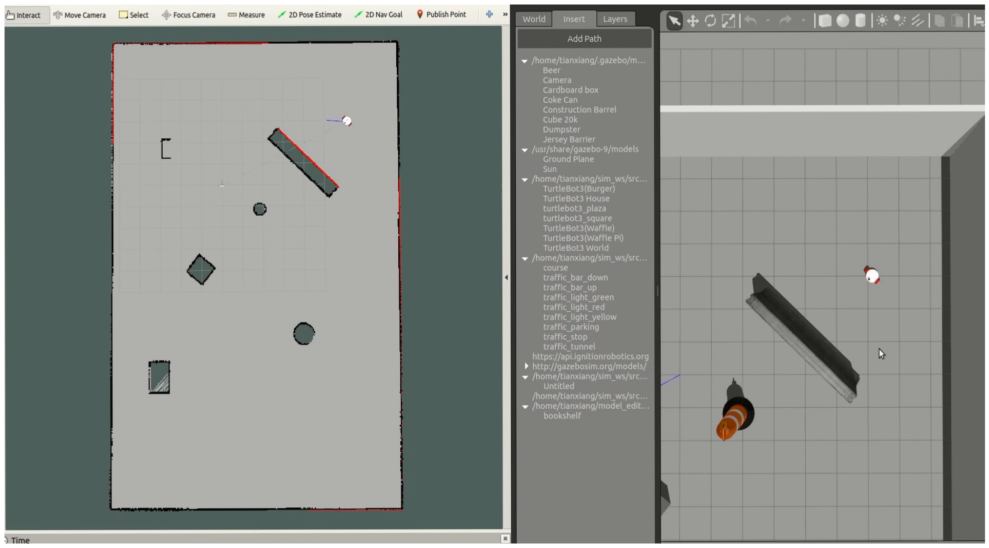


Figure 2.16: The robot achieves the navigation goal for position 1.

The robot will follow the similar procedures described above to achieve the navigation goal for other five positions. Video results of the simulation can be referred to the links given in the Appendix.

Then to test whether our robot has the ability to avoid unexpected obstacles, we add obstacles in the working environment as shown in Figure 2.17.

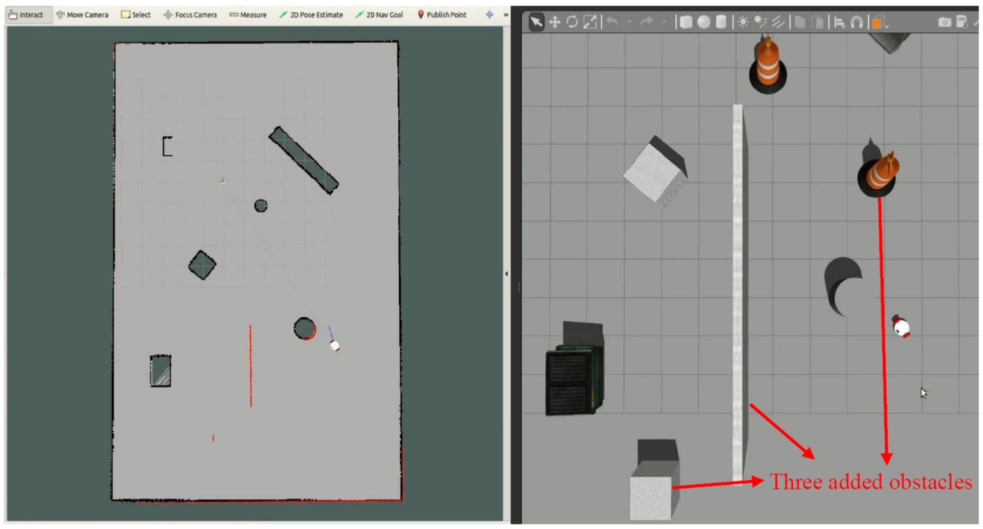


Figure 2.17: Add three obstacles in the working environment.

The robot then efficiently computes the global and local paths and starts to move as shown in Figure 2.18.

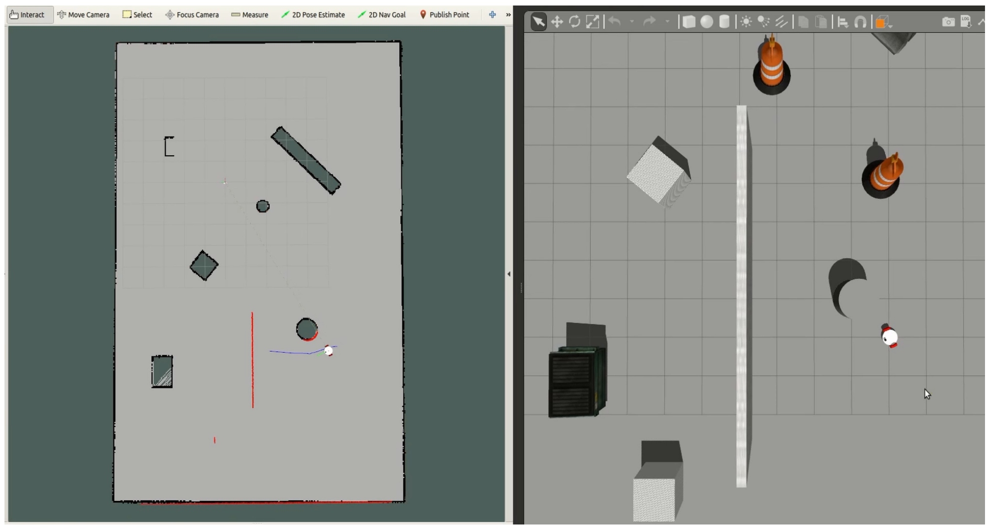


Figure 2.18: The robot finishes computing the global and local paths.

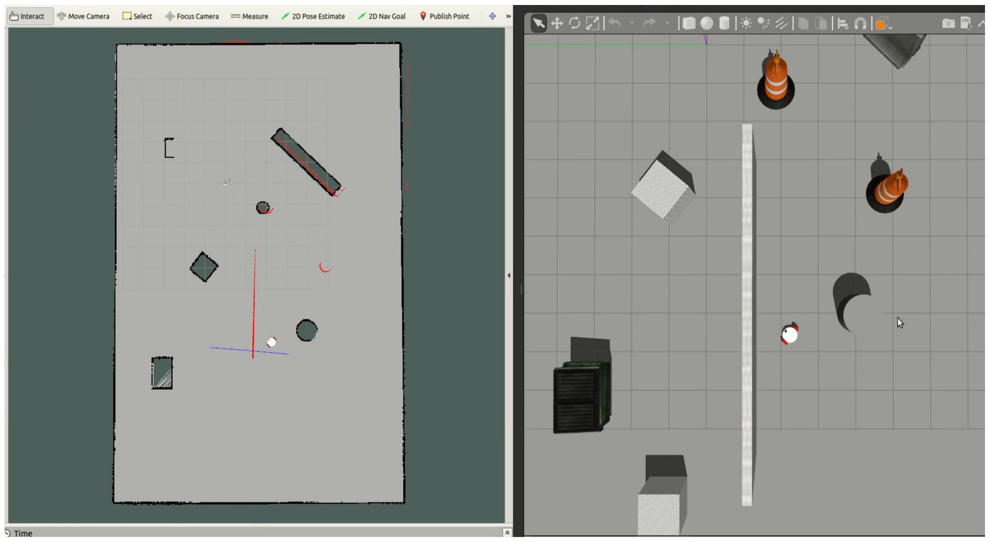


Figure 2.19: The robot starts to re-plan the global and local paths.

However, when the robot observes the unexpected obstacles, it will discard the global path and start to re-plan the global path as shown in Figure 2.19. Additionally, when it comes to a position which is considerably close to two or more objects, it will stop and perform in-place rotation to search for feasible global and local paths. This can be seen in Figure 2.20.

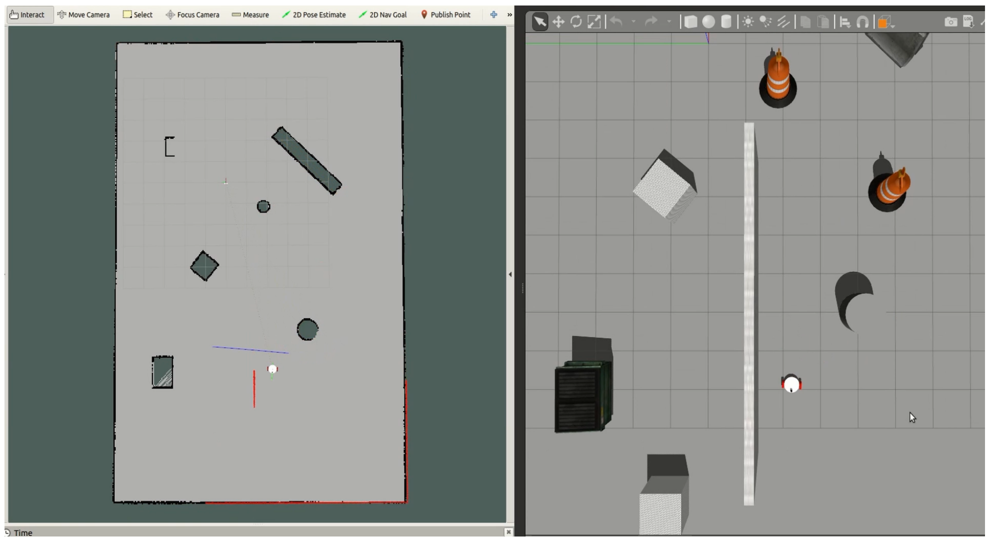


Figure 2.20: The in-place rotation of the UGV to search for feasible paths.

When the robot successfully plans a local path which will avoid collisions with objects, it will follow the local path to move without causing collisions. This is shown in Figures 2.21 and 2.22.

Lastly, Figures 2.23 and 2.24 show that the robot is able to achieve autonomous navigation and to continue its navigation task without colliding with any unexpected obstacles.

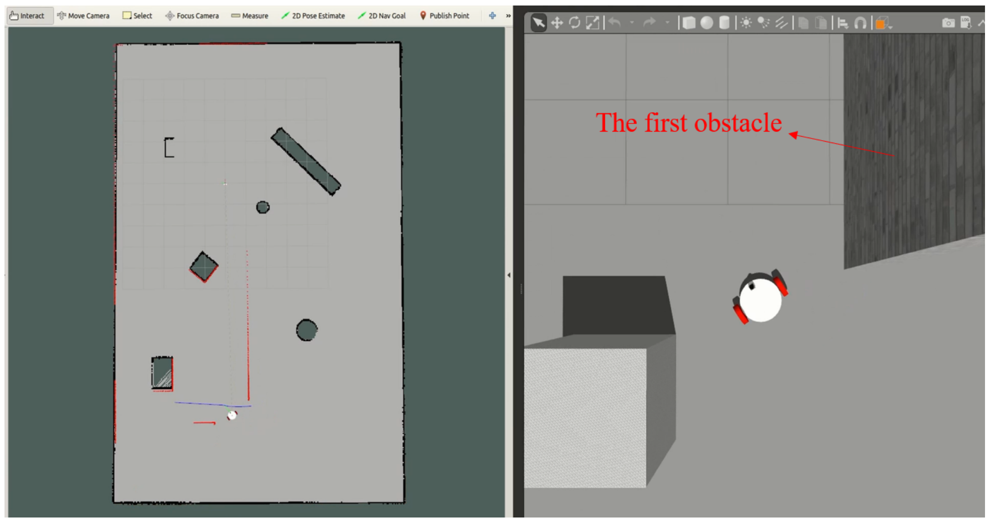


Figure 2.21: The robot avoids the collision with the first added obstacle.

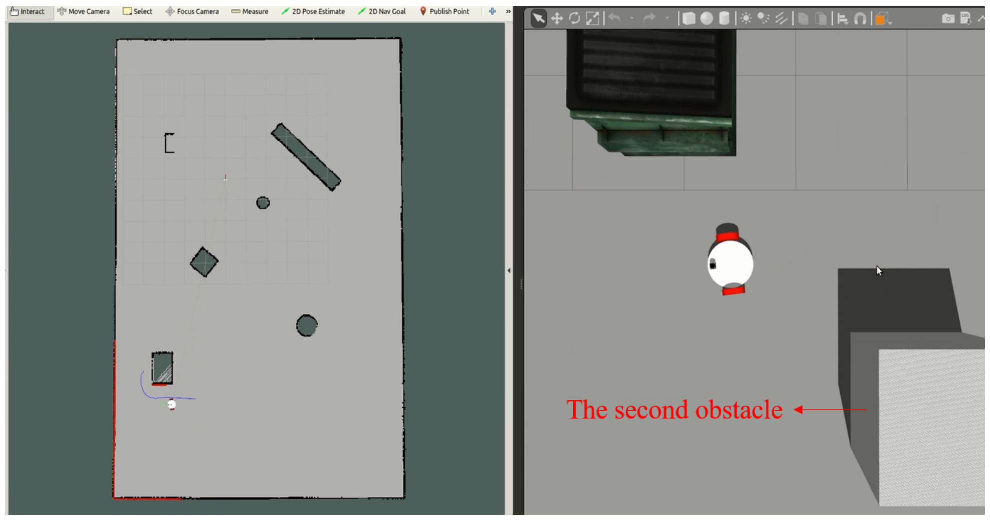


Figure 2.22: The robot avoids the collision with the second added obstacle.

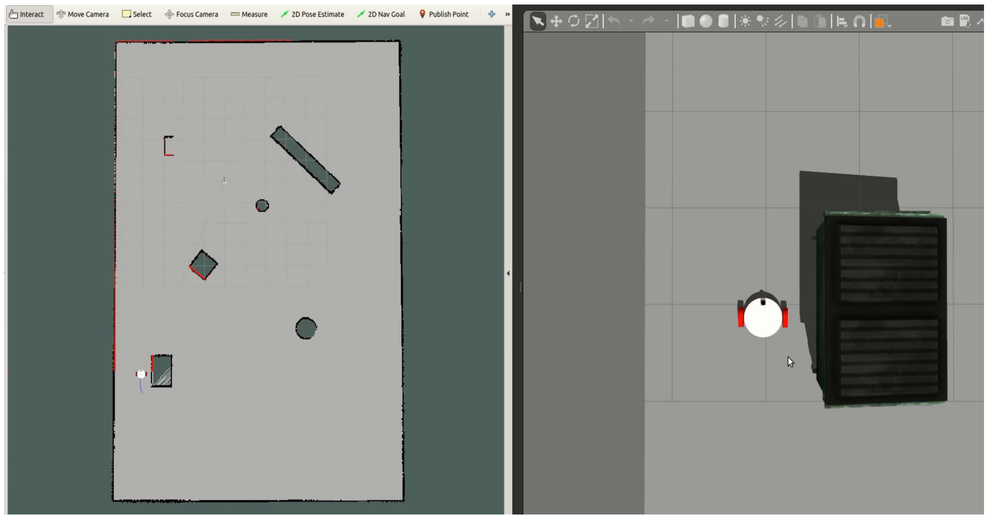


Figure 2.23: The robot achieves the navigation goal for position 4 with avoiding unexpected obstacles.

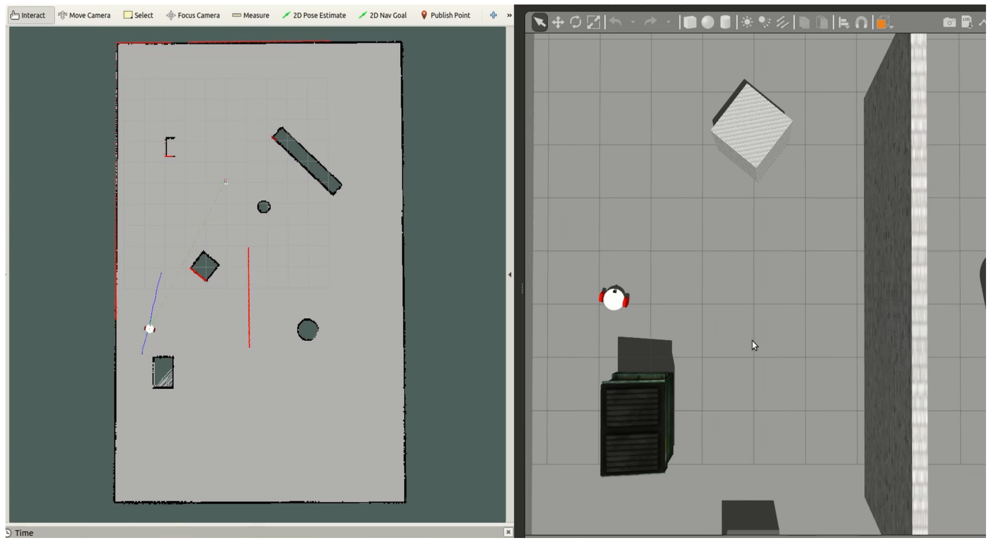


Figure 2.24: The robot continues the navigation.

2.5 Conclusion

In this chapter, we present the simulation of autonomous navigation for a two-wheeled differential ground robot in the ROS environment. The simulated URDF-based robot model is built. And the SLAM of the simulated robot can be realized by properly using the gmapping package. Moreover, by following the navigation stack setup and using move_base and amcl with suitable configuration, the autonomous navigation can be achieved with the functionality of collision avoidance.

Chapter 3

Experiment of Autonomous Navigation for a Two-Wheeled UGV

3.1 Overview

In Chapter 2, we conduct the simulation of autonomous navigation for a two-wheeled differential ground vehicle. In this chapter, we present the realization of autonomous navigation for a two-wheeled UGV in experiments. The remaining part of this chapter is organized as follows. Section 3.2 describes the necessary hardware components of the UGV to realize autonomous navigation. Sections 3.3 and 3.4 introduce the implementation of SLAM and autonomous navigation for the UGV, respectively. Section 3.5 concludes this chapter.

3.2 Hardware components of the UGV

A UGV is always faced with sensor errors caused by the external disturbances from the environment, and also unexpected situations such as wheel skidding during the practical operation. Therefore, a practical UGV to realize autonomous navigation should have more hardware components compared to the simulated UGV shown in Chapter 2. Below we introduce both the existing hardware components of the UGV and peripheral devices which can be added to enhance the performance of the existing robot.

3.2.1 Master computer

The master computer provides a platform for ROS operation. It performs most of the computational work for autonomous navigation. Specifically, it computes odometry, makes data from visual sensor system compatible with the move_base node if necessary, and plans paths by move_base. Accordingly, a Dell XPS 15 model is used with an Intel Core i5-6300HQ CPU @ 2.30 GHz $\times 4$ processor and 8GB RAM. It runs Ubuntu 18.04 64-bit with ROS Melodic.

3.2.2 Microcontroller unit (MCU)

MCU acts as the base controller in Figure 2.1 to drive the DC motors loaded with the wheels of the robot. Thus it should be compatible with both the ROS and Ubuntu environments installed on the master computer. It also generates pulse-width modulation (PWM) signals for driving the motors connected to the robot wheels. In addition, to handle the external disturbances during the navigation, a more robust control strategy is implemented in MCU. In this experiment, we apply the classical proportional-integral-derivative (PID) control strategy. To satisfy the above requirements, the Elegoo Atmega2560 R3 board with built-in Atmel Atmega2560 microcontroller is chosen and its technical specifications are listed in Table 3.1.

Table 3.1: Technical specifications of the Elegoo Atmega2560 R3 board [1].

Microcontroller	Atmel Atmega2560-16au and Atmega 16u2
Operating Voltage (V)	5
Input Voltage (recommended) (V)	7-12
Input Voltage (limits) (V)	6-20
Digital I/O Pins	54 (of which 15 provide PWM output)
Analog Input Pins	16
DC Current per I/O Pin (mA)	40
DC Current for 3.3V Pin (mA)	50
Flash Memory (KB)	256 KB of which 8 KB used by bootloader
SRAM (KB)	8
EEPROM (KB)	4
Clock Speed (MHz)	16
PWM Specifications	Pin number: 2-13, 44-46
	Default frequency: 980 Hz for pins 13 and 4, 490 Hz for others
	Scale of duty cycle: 0-255

3.2.3 Sensor system

As aforementioned in Section 1.2, the whole sensor system of the UGV can be divided into two parts: The odometry sensors and the visual sensors. For odometry sensors, one can choose to use a single sensor or a combination of several sensors along with utilizing the data fusion algorithms such as the extended Kalman filter (EKF) and unscented Kalman filter (UKF) supported by ROS. Here the wheel encoder which is generally attached to the motor is selected. Unlike the use of the Hokuyo Lidar in simulation, the Kinect V1 camera which is a RGB-D camera is used due to its low cost. Though its performance is not comparable to a Lidar, it suffices for most indoor applications [48]. Additionally, the use of IMU is discussed below in Section 3.2.5.

3.2.4 Vehicle platform

The vehicle platform consists of three layers: The base layer, medium layer and top layer. Most hardware peripherals are put on the chassis on the base layer. The master computer and the visual sensor can be put on the medium layer and top layer, respectively.

Similarly to the simulated ground robot, the chassis is supported by an omnidirectional caster. The chassis is made of wood due to the low cost and machining difficulty. The wheels with one on each side of the chassis are linked to the chassis by the DC motors. And the DC motors are connected to the MCU board through the DC motor driver board. Considering the compatibility of the four parts: MCU, DC motor driver board, motors, and wheels, the Cytron MDD10A dual channel 10A DC motor driver, Pololu 70:1 metal gearmotor 37D×70L mm 12V with 64 CPR encoder, and wheels with diameter of 72 mm are chosen respectively. As aforementioned, the wheel encoder is required to provide odometry information, thus the DC motor with attached quadrature encoder is selected. The caster is then chosen as the Pololu ball caster with the diameter of 1 inch according to the size of wheels. These peripherals constitute the base layer of vehicle platform for the UGV.

For the hardware connection on the base layer, the MCU board provides the ground reference for the motor driver board and the two encoders, and also outputs 5V voltage to power the encoders. Except the pin for ground, the motor driver board has 4 input pins of which 2 for direction input and other 2 for PWM input. The two direction input pins can be linked to any 2 digital pins on the MCU board. And the two PWM input pins need to be connected to pins on the MCU board which are able to generate PWM signal. The motor driver board has 4 output pins with 2 for each motor and has 2 pins for external power source. For each encoder, apart from 2 pins mentioned before, 2 pins are connected to digital pins on MCU board to enable the communication. The ground robot is built up as shown in Figure 3.1. Additionally, the specifications for the UGV are shown in the Table 3.2.

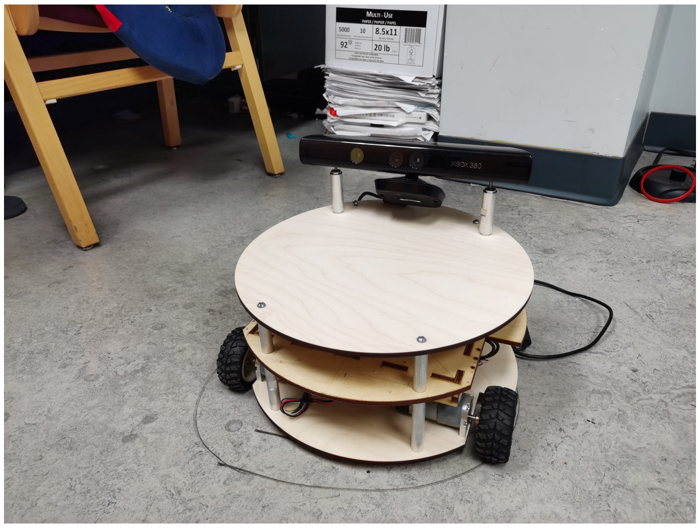


Figure 3.1: The two-wheeled UGV used in the experiment.

3.2.5 Inertial Measurement Unit

The IMU can be an auxiliary device to provide odometry information which helps the localization of the ground robot. This results from that the data error from wheel encoders leads to the cumulative inaccuracy of computing the robot position, especially when a ground vehicle works in a complex outdoor environment to achieve a long-term task. Under such a circumstance, the localization can be reliable by making use of more in-depth measurements such as the magnetic field, angular velocities and linear acceleration. And these measurements can only be provided by the IMU. However, in our experiment, the UGV works in a considerably small indoor space and is faced with a simple short-term navigation task, and it is not necessary to use an IMU to achieve autonomous navigation.

Table 3.2: Technical specifications of the two-wheeled UGV.

Specification	Value
Wheel Diameter (mm)	72
Wheel Width (mm)	34
Distance between Wheels (m)	0.3
Motor Rated Voltage (V)	12
Motor No-load Speed (rpm)	150
Motor No-load Current (A)	0.2
Motor Speed at Max Efficiency (rpm)	130
Motor Current at Max Efficiency (A)	0.68
Gear Ratio	70:1
The Number of Encoder Counts per Revolution of Shaft	4480
The Maximum Linear Velocity for UGV (m/s)	0.565
The Maximum Angular Velocity for UGV (rad/s)	3.77

3.2.6 Power supply

The power supply is determined by the power requirements from each component of the UGV. As the microcontroller board is powered by the master computer through a USB cable, the capacity of external power relies on the requirements from the DC motors, motor driver board and Kinect camera. Accordingly the 25C-50C, 2500mAh, 11.1V LiPo Battery is chosen due to its considerably large capacity and the maximum voltage of 12.60V.

3.3 Implementation of SLAM for the UGV

In this section, we introduce how to implement SLAM for our two-wheeled UGV, i.e., create a map of the surrounding environment where our robot operates. We first discuss the setup and calibration of the hardware devices in Section 3.3.1. Then, experimental results on implementing SLAM for our UGV are shown in Section 3.3.2.

3.3.1 Robot setup

The robot setup can be separated into four parts: To calibrate the DC motors loaded with wheels, implement speed control algorithm on the base controller, configure necessary transformation among coordinate frames, and make the Kinect camera prepared for the map creation process.

- Calibration of DC motors loaded with wheels

  The calibration of two DC motors each loaded with a wheel is investigated to obtain the relationship between the duty cycles of PWM signals sent to the DC motors and the resulting linear velocities. The calibration needs be finished before the implementation of the control algorithm on the base controller. Specifically in this procedure, we first vary the duty cycles of PWM signals sent from the microcontroller board to the DC motors. Secondly, the linear velocities of two wheels are calculated by using the number of counts from encoders. Lastly, the relationship between the duty cycles of PWM signals and the linear velocities of the two wheels is determined by solving two optimization problems.

  We start with setting the operation direction of both motors as forward, and generating an approximately 3.92% duty cycle PWM signal in both channels of DC motors by writing a value of 10 into the function *analogWrite*. And then we increase the value by 10 each time until reaching 240 which corresponds to an approximately 94.12% duty cycle. To measure the linear velocities of two wheels, we sample the read from two quadrature encoders in the period of 40 ms and compute the linear velocity as

$$v = x * \frac{\pi * 0.072}{4480 * 40 * 10^{-3}} \ \ (\text{m/s}), \tag{3.1}$$

where x denotes the number of counts from each encoder in 40 ms, and v is the linear velocity. The linear velocity of each wheel for corresponding duty cycle is determined as the median of 100 linear velocity samples taken in a time period of 4 seconds after the stable operation of motor is achieved. Therefore, we have 24 sets of data for each DC motor linked to a wheel as shown in Table 3.3.

Table 3.3: Measured linear velocity of wheels in forward direction for corresponding duty cycle of PWM signal.

Value Written to analogWrite Function	Linear Velocity of Left Wheel (m/s)	Linear Velocity of Right Wheel (m/s)
$dc_1 = 10$	$vl_1 = 0.016409210860729218$	$vr_1 = 0.016409210860729218$
$dc_2 = 20$	$vl_2 = 0.04039190709590912$	$vr_2 = 0.04039190709590912$
$dc_3 = 30$	$vl_3 = 0.06311235576868057$	$vr_3 = 0.06311235576868057$
$dc_4 = 40$	$vl_4 = 0.08835729211568832$	$vr_4 = 0.08835729211568832$
$dc_5 = 50$	$vl_5 = 0.11107774078845978$	$vr_5 = 0.11233998090028763$
$dc_6 = 60$	$vl_6 = 0.13506042957305908$	$vr_6 = 0.13127368688583374$
$dc_7 = 70$	$vl_7 = 0.16030538082122803$	$vr_7 = 0.15904313325881958$
$dc_8 = 80$	$vl_8 = 0.17923907935619354$	$vr_8 = 0.1754523366689682$
$dc_9 = 90$	$vl_9 = 0.2019595354795456$	$vr_9 = 0.20069728791713715$
$dc_{10} = 100$	$vl_{10} = 0.2284667193889618$	$vr_{10} = 0.2259422093629837$
$dc_{11} = 110$	$vl_{11} = 0.2549739181995392$	$vr_{11} = 0.25118714570999146$
$dc_{12} = 120$	$vl_{12} = 0.2638096213340759$	$vr_{12} = 0.2638096213340759$
$dc_{13} = 130$	$vl_{13} = 0.30420154333114624$	$vr_{13} = 0.3029392659664154$
$dc_{14} = 140$	$vl_{14} = 0.33070874214172363$	$vr_{14} = 0.32818421721458435$
$dc_{15} = 150$	$vl_{15} = 0.35721591114997864$	$vr_{15} = 0.3509046733379364$
$dc_{16} = 160$	$vl_{16} = 0.3824608623981476$	$vr_{16} = 0.37867411971092224$
$dc_{17} = 170$	$vl_{17} = 0.4064435660839081$	$vr_{17} = 0.40265679359436035$
$dc_{18} = 180$	$vl_{18} = 0.4304262399673462$	$vr_{18} = 0.42663952708244324$
$dc_{19} = 190$	$vl_{19} = 0.4569334387779236$	$vr_{19} = 0.44557321071624756$
$dc_{20} = 200$	$vl_{20} = 0.4809161126613617$	$vr_{20} = 0.4670313894748688$
$dc_{21} = 210$	$vl_{21} = 0.5048988461494446$	$vr_{21} = 0.4809161126613617$
$dc_{22} = 220$	$vl_{22} = 0.5276192426681519$	$vr_{22} = 0.5023742914199829$
$dc_{23} = 230$	$vl_{23} = 0.5503396987915039$	$vr_{23} = 0.5276192426681519$
$dc_{24} = 240$	$vl_{24} = 0.5755846500396729$	$vr_{24} = 0.5276192426681519$

Then we can determine the relationship between the duty cycle of PWM signal in each channel and the linear velocity of the corresponding wheel by finding a function which best fits the data. We first analyze the data by making plots in

MATLAB. The data is shown in the shape of circle in Figure 3.2 which appears considerably accurate pattern of linear relationship. Thus finding a function to best fit the data for each wheel becomes two linear regression problems as:

$$\underset{k_l, b_l}{\text{minimize}} \quad \sum_{i=1}^{24} (k_l \cdot dc_i + b_l - vl_i)^2$$

and

$$\underset{k_r, b_r}{\text{minimize}} \quad \sum_{i=1}^{24} (k_r \cdot dc_i + b_r - vr_i)^2$$

for left and right wheels, respectively. Solving two optimization problems yields the solution: k_l = 0.002452, b_l = -0.01273, k_r = 0.002328, and b_r = -0.004788 which form two linear functions shown in the shape of line in Figure 3.2. And the results can also be validated with the *Curve Fitting Tool* in MATLAB.

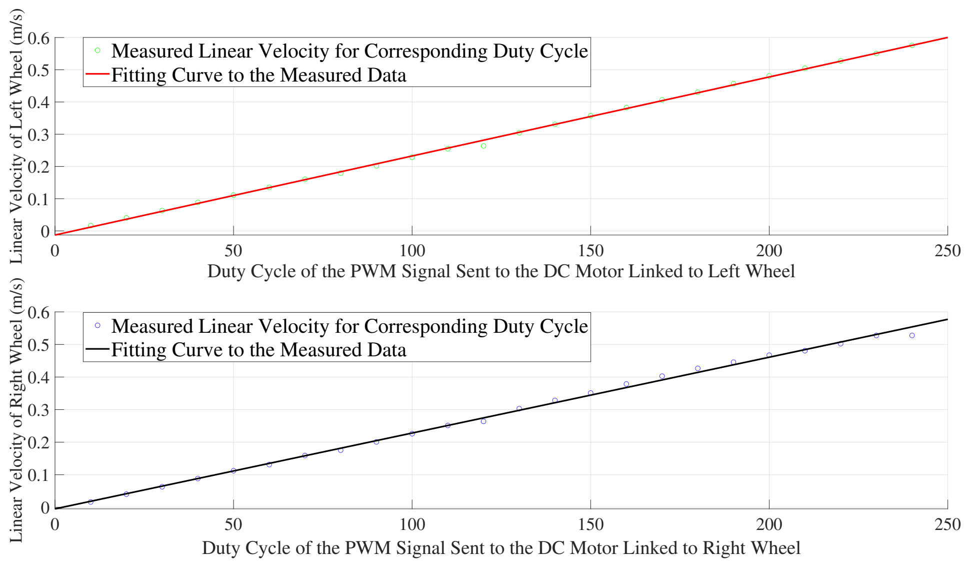


Figure 3.2: The relationship between the duty cycle of PWM signal and the operation speed of wheels in forward direction.

This gives us the relationship between the duty cycle of PWM signal in each channel and the linear velocity of corresponding wheel when motors rotate in forward direction as:

$$\begin{aligned} dc_l &= \frac{1}{k_l}(v_l - b_l) \\ &= \frac{1}{0.002452}(v_l + 0.01273), \end{aligned} \tag{3.2}$$

and

$$\begin{aligned} dc_r &= \frac{1}{k_r}(v_r - b_r) \\ &= \frac{1}{0.002328}(v_r + 0.004788), \end{aligned} \tag{3.3}$$

where dc_l and dc_r denote the value written to *anologWrite* function to generate certain duty cycle of PWM signal sent to the DC motors connected to the left and right wheels, respectively. v_l and v_r denote the linear velocities of left and right wheels, respectively. And the positiveness of v_l and v_r indicate the rotation of the left and right wheels in the forward direction, respectively. The negativeness of v_l and v_r represent the rotation in the backward direction.

When motors rotate in the backward direction, we follow similar procedures and obtain similar results as:

$$\begin{aligned} dc_l &= \frac{1}{k_l}(v_l - b_l) \\ &= \frac{1}{-0.00246}(v_l - 0.009901) \end{aligned} \tag{3.4}$$

and

$$\begin{aligned} dc_r &= \frac{1}{k_r}(v_r - b_r) \\ &= \frac{1}{-0.002306}(v_r - 0.006586). \end{aligned} \tag{3.5}$$

Related data and figures can be referred to Table A.1 and Figure A.1 in the

Appendix.

- Mobile base control setup

After the calibration of DC motors loaded with wheels, we program the MCU board to configure the base controller for the UGV. First we adopt the same controller used in our simulation. By using Equation 2.2, the velocity command sent from the master computer is split into the desired velocity of each wheel. Consider the external disturbances exerting on the UGV, we then implement the classical PID control method in the MCU board for the DC motor speed control. Additionally, the PID controller can eliminate the overshoot and decrease the settling time to meet the real-time requirements. A diagram to illustrate the speed control of a DC motor using PID method is shown in Figure 3.3.

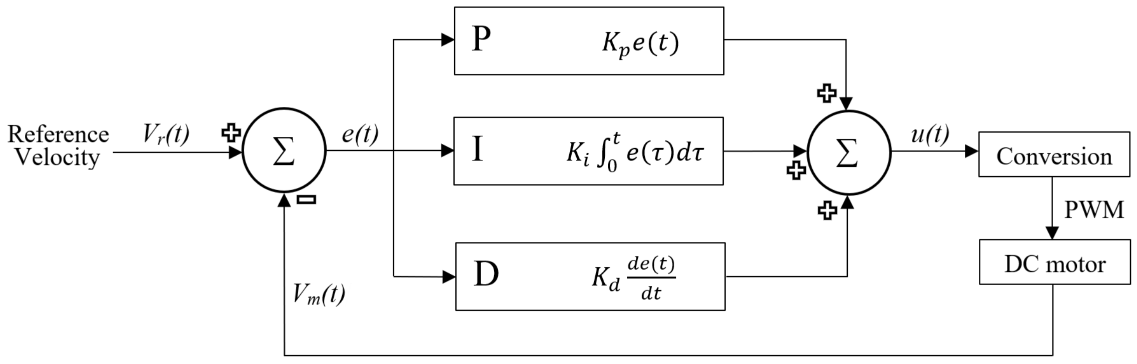


Figure 3.3: A PID controller for the speed control of a DC motor.

The reference velocity signal V_r is transformed from the velocity command sent from the master computer using Equation 2.2. The measured velocity V_m of each motor is computed by the data from motor encoder using Equation 3.1. And the error between the reference and measured linear velocity of a wheel can be computed as

$$e(t) = V_r(t) - V_m(t), \tag{3.6}$$

and the control input is given by

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt}, \tag{3.7}$$

where K_p, K_i and K_d denote the proportional, integral and derivative gains, respectively.

In order to achieve satisfying control performance, the tuning of these three parameters is of great importance. There are several classical methods for tuning PID parameters such as Ziegler-Nichols (ZN) method [49], Tyreus Luyben (TL) method, Cohen-Coon method, and relay (Åström-Hägglund) method [50, 51]. Consider the similarity between the ZN and TL methods, accuracy of physical model required by the Cohen-Coon method, and possible dangerous oscillation caused by the relay method, we choose to follow the process of ZN and TL methods. The tuning process for each motor loaded with a wheel can be described in the following four steps:

1. Set a sufficient small proportional gain K_p, and zero integral and derivative gains, i.e., $K_p = 0.1$, $K_i = K_d = 0$. Set the reference velocity to 0.08 m/s for each wheel in ROS.

2. Keep $K_i = K_d = 0$ and gradually increase the proportional gain K_p until a stable oscillation of the output linear velocity is achieved, as shown in Figures 3.4 and 3.5. Record the critical proportional gain at this point as K_{pc}.

3. Set the proportional gain $K_p = 0.6 * K_{pc}$, keep $K_d = 0$ and gradually increase the integral gain K_i with keeping K_d until the steady-state error is approximately zero. Record the integral gain at this point denoted as K_{ic}.

4. Set the derivative gain as

$$K_d = \frac{3.6 * k_{pc}^2}{40 * K_{ic}}, \tag{3.8}$$

and slightly adjust the integral and derivative gains K_i and K_d until a considerably fast output response with negligible overshoot and steady-state error is achieved as shown in Figure 3.6.

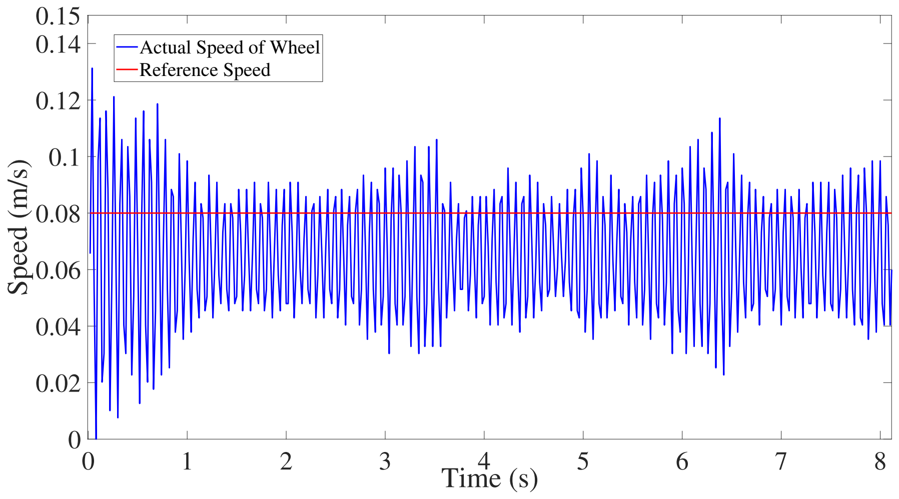


Figure 3.4: The oscillation of linear velocity of the left wheel during PID tuning process.

Compare with standard ZN and TL methods, some modifications are considered here. Due to the existence of inherent error of the quadrature encoders, in step 2, it is difficult to evaluate whether stable oscillations of rotation for both wheels are accurately achieved according to data from encoders. This can also be seen in Figure 3.4 and 3.5. Therefore, K_{pc} for each motor is the value set to the proportional gain K_p that causes the oscillation for the first time. The inherent error also leads to that the oscillation period (generally denoted as T_{pc} in ZN and TL method) for each wheel is hard to be determined. Therefore, the formula for the integral gain K_i shown in Equation 3.9 cannot be utilized.

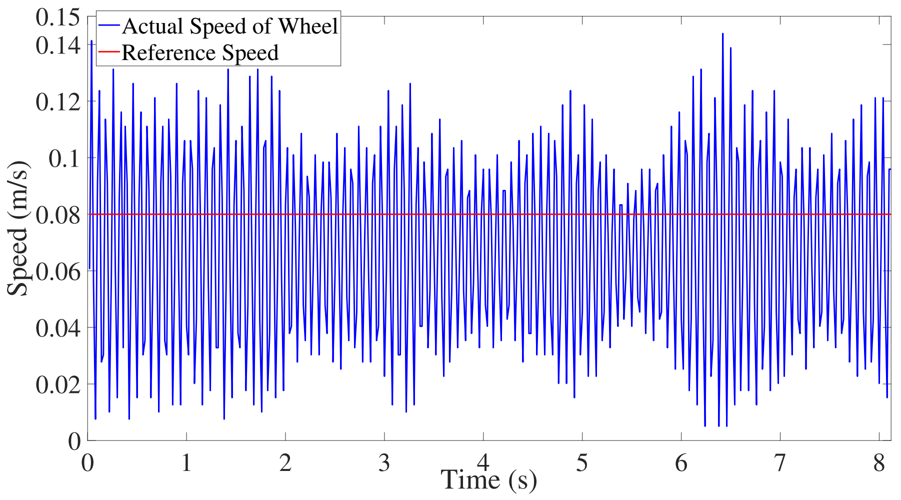


Figure 3.5: The oscillation of linear velocity of the right wheel during PID tuning process.

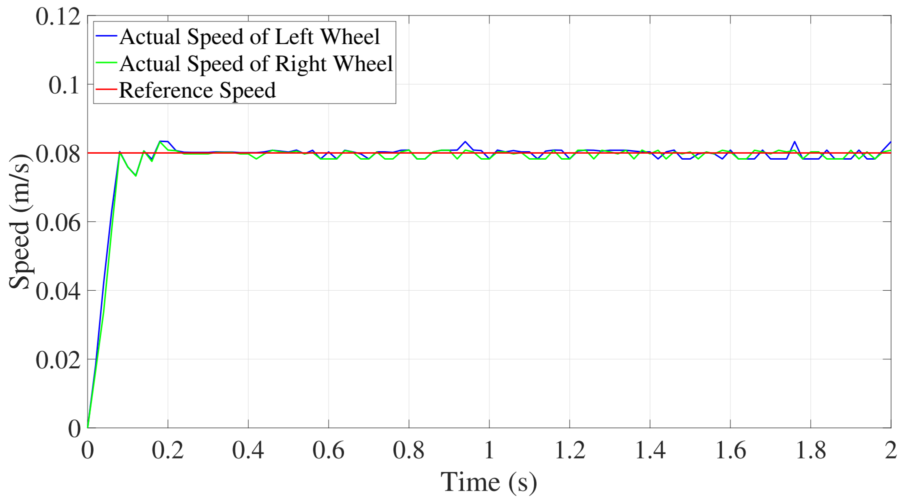


Figure 3.6: The output response in PID tuning process.

An appropriate integral gain K_i is considered as the value to best eliminate the steady-state error. Then with a settled integral gain, we can make use of the formulas for the integral and derivative gain in ZN method in Equation 3.9 to derive Equation 3.8.

$$\begin{aligned} K_i &= 1.2\frac{K_{pc}}{T_{pc}} \\ K_d &= \frac{3K_{pc}T_{pc}}{40}. \end{aligned} \tag{3.9}$$

After obtaining the suitable integral gain $K_i = K_{ic}$ and derivative gain K_d using Equation 3.8, we slightly adjust the two gains to achieve better control performance. After the adjustment process, for the motor loaded with the left wheel, the final values of the proportional, integral and derivative gains are set to be $K_p = 3.6$, $K_i = 0.8$, and $K_d = 0.8$, respectively. For the motor loaded with the right wheel, the final value of the proportional, integral and derivative gains are set to be $K_p = 3.6$, $K_i = 1$, and $K_d = 1$, respectively. In conclusion, from Figure 3.6 we can see that using PID controllers for the speed control of DC motors loaded with two wheels in our UGV is acceptable and effective.

- Transform configuration

  To implement SLAM using gmapping and autonomous navigation with navigation stack for our two-wheeled UGV, we next need to configure both static and dynamical transforms to establish relationships between coordinate frames. We first build the static transform relationship between the frames attached to the mobile base and visual sensor system. In this experiment, we name the coordinate frame attached to the base layer of UGV as “base_link” and the frame attached to the Kinect camera as “base_laser”. We follow a similar procedure for the example shown in Figure 2.2. The Kinect camera is placed 9cm backward and 17.3cm above the center of base layer. Therefore, if we let the “base_link” frame as the parent node of the link and “base_laser” frame as the child node, from the “base_laser” frame to the “base_link” frame, we have the offsets of -0.09m, 0, and 0.173m in the x, y and z directions, respectively. Thus a 3-D vector (-0.09, 0.0, 0.173) is used to set up the translational relationship from the “base_laser” frame to the “base_link” frame.

Additionally, the dynamical transform in terms of the odometry information needs to be configured and published in the ROS environment. This is both required by the gmapping package when implementing SLAM and by the navigation stack for autonomous navigation. Specifically, the transform from the “odom” frame to the “base_link” frame needs to be published. As aforementioned, the “odom” coordinate frame is a world-fixed reference frame which can be used to localize the UGV. Here we briefly introduce how the localization using these two coordinate frames works. Initially the “base_link” frame coincides with the “odom” frame, with the moving of UGV, the “base_link” frame moves relative to the “odom” frame. With using the data from quadrature encoders we can compute the relative position of UGV with respect to the “odom” frame and thus the position in the global working environment can be inferred. This process is also known as *dead reckoning* in which the current position of UGV is determined by using the previous position [52]. We illustrate the computation by Figure 3.7.

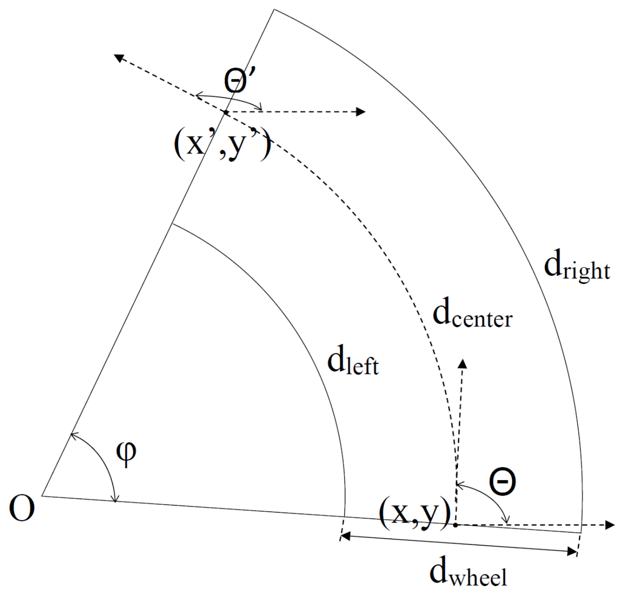


Figure 3.7: The odometry geometry.

Figure 3.7 shows the movement of the robot over a small time period from position (x, y) to position (x', y'). The moving distance for the left and right wheels during the time period are respectively denoted as d_{left} and d_{right}. The rotation angle of the robot during the period is denoted as φ. And θ and θ' denote the heading, i.e., the yaw angle of robot in position (x, y) and (x', y'), respectively. The distance between two wheels of robot is denoted as d_{wheel}. As d_{left} and d_{right} can be obtained with using the velocity of the corresponding wheel computed with Equation 3.1, the length of the moving trajectory for the center of two wheels (approximately the center of the robot) denoted as d_{center} can be computed as

$$d_{center} = \frac{1}{2}(d_{left} + d_{right}). \tag{3.10}$$

And with basic geometry, we can obtain

$$\varphi = \frac{d_{right} - d_{left}}{d_{wheel}}. \tag{3.11}$$

Additionally, in our case the sampling period is set to be 40ms which is a considerably small value, we can approximately have the relationship between the current position of the robot and the previous position as

$$\begin{aligned} \theta' &= \theta + \varphi \\ x' &= x + d_{center}\cos(\theta) \\ y' &= y + d_{center}\sin(\theta). \end{aligned} \tag{3.12}$$

More detailed derivation for the odometry can be referred in [53].

To match the form of publishing rotation information in the transform, the yaw angle needs to be transformed into quaternion. This is realized using the built-in function *createQuaternionMsgFromYaw* in TF package. And by Equation 3.12, the position of UGV can be continually updated and thus the transform from the "odom" frame to the "base_link" frame can be published in ROS.

- Kinect setup

  To match the type of subscribed topic for the data from visual sensor system, we need to transfer the image-type data obtained from Kinect camera to the laser-scan-type data to be used for gmapping and navigation stack nodes. Here we make use of the *depthimage_to_laserscan* package. Figure 3.8 shows how the depth data is converted to the laser scan data.

  The image on the left shows the original image in color taken by the Kinect camera. The image on the upper right corner is the depth image in grey scale processed by the depthimage_to_laserscan package. We can see that there is a straight line in the middle of the image. The color of the segment on this line implies the distance of the corresponding position to the camera. Specifically, red is close to camera and purple is far from camera. Additionally, the image on the lower right corner shows the image in the type of laser scan which is converted from the image on the upper right corner.

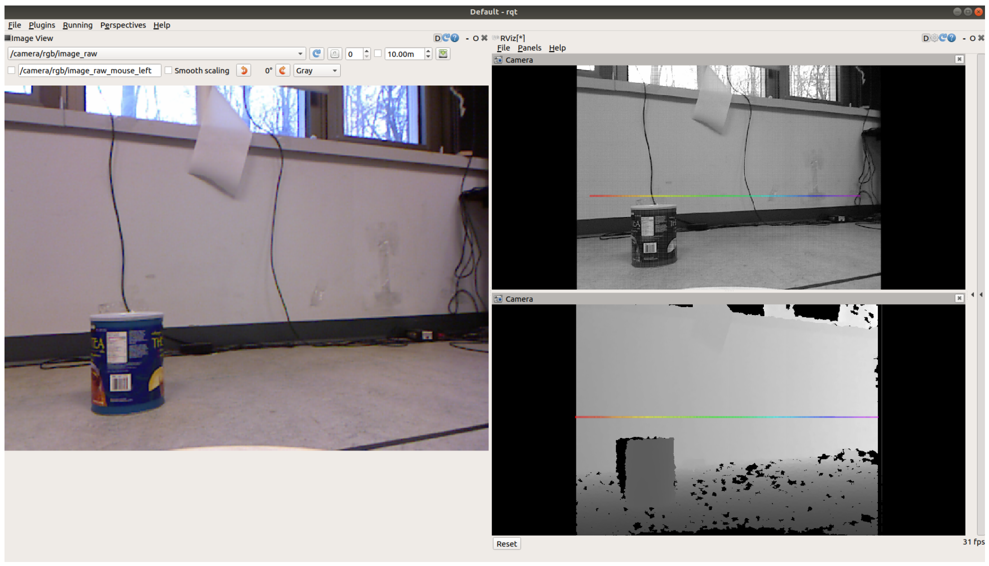


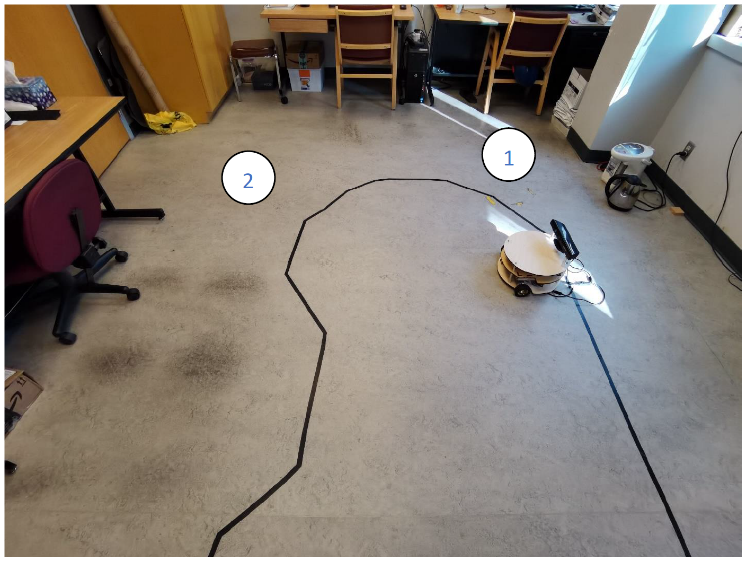
Figure 3.8: The RGB, depth, and laser scan images obtained by Kinect camera.

3.3.2 Experimental results

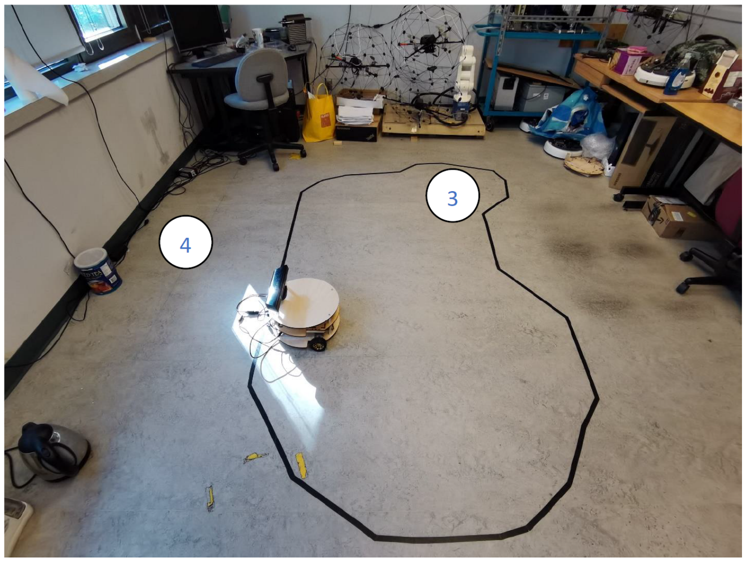
Similarly to our simulation, we first create the map of the working environment of our UGV. A series of four target waypoints is also set in the working environment as the navigation goal for UGV. The four waypoints along with the environment in which the UGV navigates are illustrated in Figure 3.9.

To create the map, we first start the communications between the UGV and the Kinect camera, MCU board and also start to publish necessary transforms. And then gmapping node is ran with the same configuration as simulation shown in Table 2.16 and the teleoperation node is launched similarly to simulation.

After controlling the UGV to move in its working environment, the map is obtained as shown in Figure 3.10.



(a) The first and second target waypoints for navigation.



(b) The third and fourth target waypoints for navigation.

Figure 3.9: Four target waypoints in the working environment.

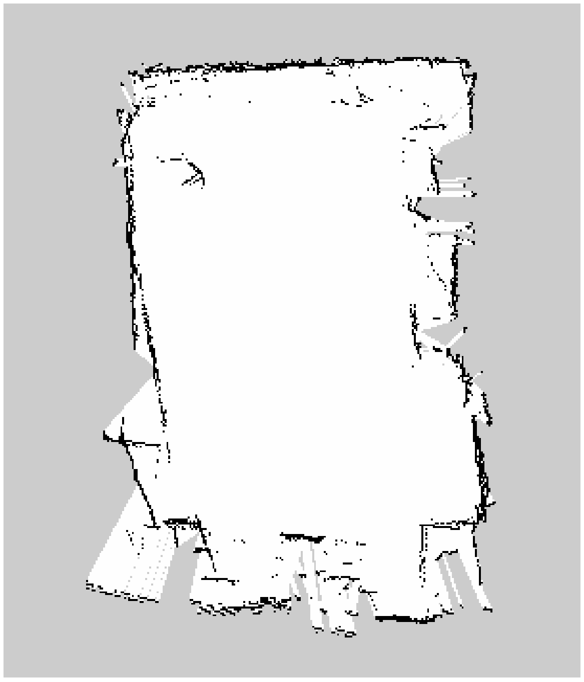


Figure 3.10: The obtained map of working environment for UGV.

It can be observed that in the created map, the black line segments outline the corresponding objects and walls in practice. Thus the map is applicable to the autonomous navigation. However, there exist overlapping lines in areas near the borders of the map. This mainly results from the quality of images taken by the Kinect camera and further degradation of images when converted into the type of laser scan. Additionally, it is difficult for the Kinect camera to observe the border in the bottom. As a result, there exists extrusion of free space in the bottom of the map.

3.4 Implementation of autonomous navigation for the UGV

3.4.1 Experimental setup

As aforementioned, a UGV with autonomous navigation ability can achieve the given goal without causing collisions with objects which include existing ones in its working environment, and unexpected ones during the navigation. Therefore, to test the autonomous navigation along with the collision avoidance ability, two obstacles are put in the working environment as shown in Figure 3.11.

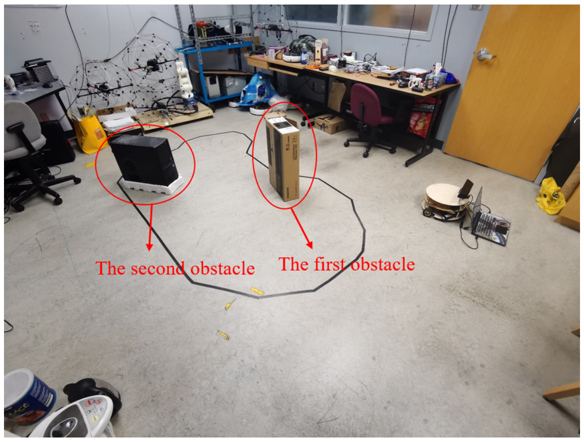


Figure 3.11: Two obstacles in the working environment.

Then we need to configure the move_base node which includes three parts: The global and local costmap, and the local planner by setting the values of parameters introduced in Tables 2.17, 2.18, 2.19, and 2.20. The configuration of these parameters are listed in Tables 3.4, 3.5, 3.6, and 3.7. In addition, the amcl node is configured as shown in Table 3.8.

Table 3.4: Adopted parameter values in common configuration for both costmaps.

Parameter	Description	Value
obstacle_range	Maximum range for robot to detect an obstacle (m)	2.5
raytrace_range	Maximum range for robot to clear out free space (m)	3.0
robot_radius	Radius of robot if setting the robot center as origin (m)	0.1778
inflation_radius	Minimum safety distance between robot and obstacles (m)	0.3

Table 3.5: Adopted parameter values in global configuration for global costmap.

Parameter	Description	Value
global_frame	Coordinate frame the costmap runs in	“map”
robot_base_frame	Coordinate frame the costmap refers for the robot base	“base_link”
update_frequency	Update frequency of the costmap (Hz)	5.0
static_map	If the initialization of the costmap is based on the map served by the map_server	true

Table 3.6: Adopted parameter values in local configuration for local costmap.

Parameter	Description	Value
global_frame	Coordinate frame the costmap runs in	“odom”
update_frequency	Update frequency of the costmap (Hz)	5.0
publish_frequency	Frequency of the costmap publishing visualization information (Hz)	2.0
static_map	If initialization of costmap is based on the map served by the map_server	false
rolling_window	If robot needs costmap to remain centered around robot	true
width	The size of the costmap (m)	3.0
height		3.0
resolution	Resolution of the costmap (m/cell)	0.05

Table 3.7: Adopted parameter values in base local planner configuration.

Parameter	Description	Value
controller_frequency	Frequency of sending control signal to the base controller (Hz)	7
acc_lim_x	The x acceleration limit of the robot (m/s^2)	0.1
acc_lim_y	The y acceleration limit of the robot (m/s^2)	0.0
acc_lim_theta	Rotational acceleration limit of the robot (rad/s^2)	0.1
max_vel_x	Maximum forward velocity allowed for the base (m/s)	0.2
min_vel_x	Minimum forward velocity allowed for the base (m/s)	0.02
max_vel_theta	Maximum rotational velocity allowed for the base (rad/s)	0.2
min_in_place_vel_theta	Minimum rotational velocity allowed for the base when performing in-place rotations (rad/s)	0.3
escape_vel	Speed used for driving during escapes (m/s)	-0.15
holonomic_robot	If the robot is holonomic	false

Table 3.8: Parameter values in amcl configuration.

Parameter	Description	Value
use_map_topic	When set to be true, amcl subscribes to map topic instead of receiving map from server	false
odom_model_type	Type of model for odometry system	"diff"
odom_alpha3	Expected noise in the estimate of odometry's translation based on the translational component of the robot motion	0.8
gui_publish_rate	Maximum rate of publishing information on visualization (Hz), -1.0 represents disablement	10.0
laser_max_range	Maximum range for laser scans (m)	8.0
min_particles	Number of minimum particles allowed	500
max_particles	Number of maximum particles allowed	5000
kld_err	Maximum error between true and estimated distribution	0.05
laser_z_hit	Mixture parameter for z_hit, z_short, z_max, and z_rand part of model	0.5
laser_z_short		0.05
laser_z_max		0.05
laser_z_rand		0.5
odom_frame_id	Coordinate frame for odometry system	"odom"
update_min_d	Translational distance required for filter to perform an update (m)	0.2
update_min_a	Rotational angle required for filter to perform an update (rad)	0.5
resample_interval	Number of updates for filter before re-sampling	1

3.4.2 Experimental results

After we launch the communication with UGV, start the move_base and amcl nodes, and send the navigation goal to UGV, the autonomous navigation is performed. Figures 3.12, 3.13, and 3.14 show that the UGV reaches positions 1, 2 and 3, respectively.

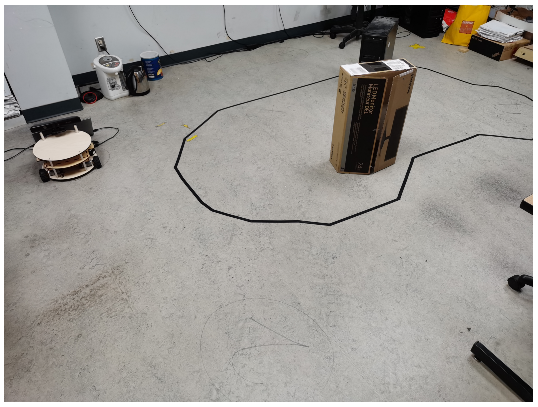


Figure 3.12: The UGV reaches the first target position.

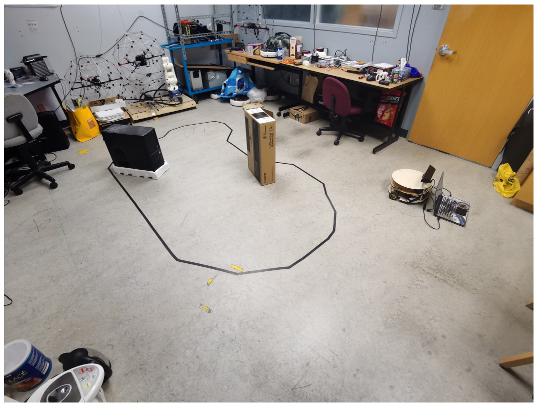


Figure 3.13: The UGV reaches the second target position.

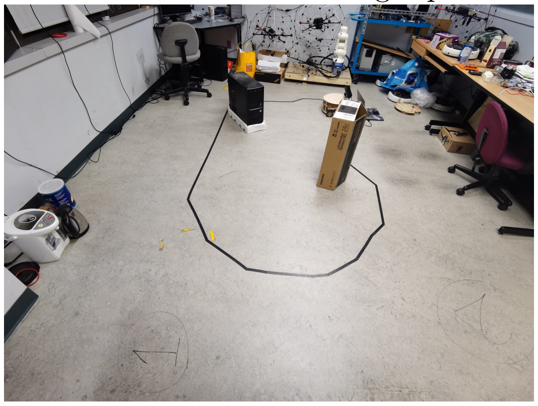


Figure 3.14: The UGV reaches the third target position.

Figures 3.15, 3.16 and 3.17 show the process of UGV moving from position 3 to position 4 while avoiding the collisions with the two obstacles. And from Figure 3.18 we can see that UGV successfully reaches the fourth target waypoint position 4.

In conclusion, the autonomous navigation using our two-wheeled UGV without collisions with objects is successfully implemented.

Video result of the experiment can be referred to the link given in the Appendix.

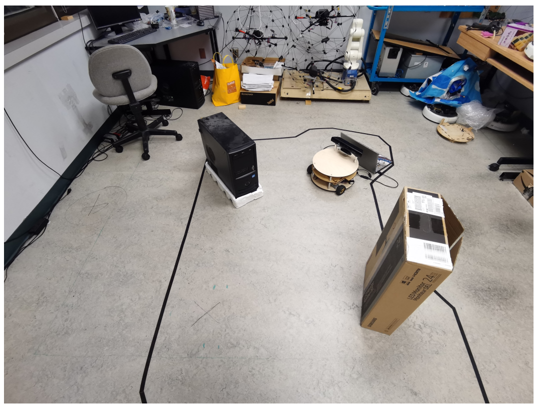


Figure 3.15: The UGV starts to move from the third position.

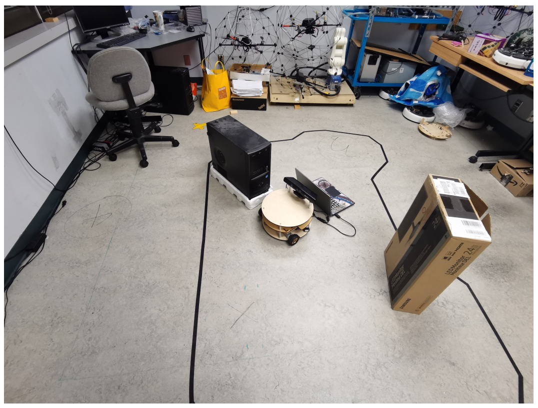


Figure 3.16: The UGV stops to plan the local path.

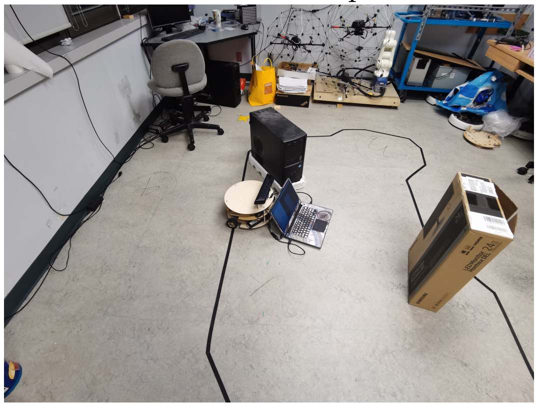


Figure 3.17: The UGV moves to avoid the obstacles.

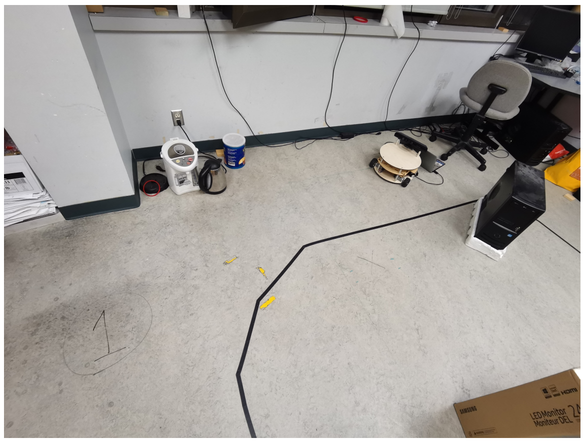


Figure 3.18: The UGV reaches the fourth target position.

3.5 Conclusion

In this chapter, we present the experimental studies on the autonomous navigation for a two-wheeled UGV. The hardware devices on the UGV are first introduced. For the software part, differently from the simulation, the classical PID-based speed control for the DC motors loaded with wheels is applied. In addition, transforms configured to publish odometry information for our UGV are introduced. By following similar setups to the simulation, the SLAM and autonomous navigation using our UGV are both successfully implemented.

Chapter 4

Conclusions and Future Work

4.1 Conclusions

This thesis presents the realization of map-based autonomous navigation for a two-wheeled UGV from both the simulation and experimental perspectives. The implementation is based on the framework of navigation stack under the ROS environment.

In **Chapter 2**, the adopted framework of navigation stack is first introduced by detailing the major components of navigation stack. Then the process of building a URDF-based model of the two-wheeled UGV is described in steps. The simulation is conducted under rviz and gazebo environments. The simulation results show that the autonomous navigation for the UGV with the obstacle avoidance ability is performed. Therefore, the effectiveness of the adopted framework of navigation stack can be validated.

In **Chapter 3**, the experimental studies on achieving autonomous navigation for our two-wheeled UGV are described. Firstly the hardware peripherals used in our UGVs are introduced. Then the experimental results on the implementation of SLAM and autonomous navigation are presented. The implementation of SLAM for our UGV is realized in advance to the navigation to obtain the map of the environment where the UGV operates. This results from that for a map-based autonomous navigation, the map is needed by a map server when the navigation starts. Finally, the experimental results demonstrate the implementation of autonomous navigation for a practical two-wheeled UGV.

4.2 Future work

In this thesis, to achieve autonomous navigation for our two-wheeled UGV, we adopt the open source framework of navigation stack provided by the ROS community. Though it has been proved to be a feasible approach, there still exist shortages in following the framework of navigation stack. Therefore, further research work can focus on improving the current framework or designing new ones. Additionally, the applications with utilizing autonomous navigation can be further completed. These lead to two possible research directions as listed below.

- The first issue is the efficiency of the navigation stack framework. Specifically, the path planning algorithm adopted in the move_base package is inefficient to some extent. The generation of a feasible local path for the UGV is often time-consuming. This results from the low efficiency of solving the optimization problem involved in the path planning algorithm. Further work will focus on adopting new optimization algorithms with higher efficiency.
- Our current work can be further developed to be incorporated into applications such as the intelligent car parking management system. Such applications require the use of hardware devices with higher practicability. For instance, battery with larger capacity should be used to be sufficient for long operating time of the UGV. And other smart devices as the master computer instead of

the laptop should be used to enhance the portability of the UGV. Also the successful rate of performing autonomous navigation should be improved. Currently, the situation which our UGV collides with obstacles still occurs due to the limitations of Kinect camera and the compatibility problem with the depthimage_to_laserscan package. To improve the operation stability, visual sensors with higher precision should replace the current camera in use.

Appendix A

Supplementary Materials

Here are the links to two videos which show the simulation of autonomous navigation for our two-wheeled UGV: `https://www.youtube.com/watch?v=YGI9IJ3Z5xE`, `https://www.youtube.com/watch?v=JPM6fnoJiWk`. The first video shows the navigation without any unexpected obstacles and the second shows the navigation with unexpected obstacles.

Here is the link to a video which shows the experimental results: `https://www.youtube.com/watch?v=7OyUPlrRGLc&t=4s`.

Table A.1: Measured linear velocity of wheels in backward direction for corresponding duty cycle of PWM signal.

Value Written to analogWrite Function	Linear Velocity of Left Wheel (m/s)	Linear Velocity of Right Wheel (m/s)
dc_1 = 10	vl_1 = -0.01514696516096592	vr_1 = -0.01514696516096592
dc_2 = 20	vl_2 = -0.03912965953350067	vr_2 = -0.037867408245801926
dc_3 = 30	vl_3 = -0.06311235576868057	vr_3 = -0.06185010448098183
dc_4 = 40	vl_4 = -0.08835729211568832	vr_4 = -0.08457054942846298
dc_5 = 50	vl_5 = -0.11107774078845978	vr_5 = -0.10855324566364288
dc_6 = 60	vl_6 = -0.13127368688583374	vr_6 = -0.13127368688583374
dc_7 = 70	vl_7 = -0.16156762838363647	vr_7 = -0.15525639057159424
dc_8 = 80	vl_8 = -0.18555031716823578	vr_8 = -0.1754523366689682
dc_9 = 90	vl_9 = -0.21079525351524353	vr_9 = -0.2019595354795456
dc_{10} = 100	vl_{10} = -0.23477794229984283	vr_{10} = -0.2259422093629837
dc_{11} = 110	vl_{11} = -0.2638096213340759	vr_{11} = -0.24866266548633575
dc_{12} = 120	vl_{12} = -0.286530077457428	vr_{12} = -0.2638096213340759
dc_{13} = 130	vl_{13} = -0.3117750287055969	vr_{13} = -0.29662805795669556
dc_{14} = 140	vl_{14} = -0.3370199501514435	vr_{14} = -0.31808626651763916
dc_{15} = 150	vl_{15} = -0.3622649013996124	vr_{15} = -0.3408066928386688
dc_{16} = 160	vl_{16} = -0.3862476050853729	vr_{16} = -0.3647893965244293
dc_{17} = 170	vl_{17} = -0.41275477409362793	vr_{17} = -0.38750985264778137
dc_{18} = 180	vl_{18} = -0.4367374777793884	vr_{18} = -0.41275477409362793
dc_{19} = 190	vl_{19} = -0.46072015166282654	vr_{19} = -0.43168848752975464
dc_{20} = 200	vl_{20} = -0.4859651029109955	vr_{20} = -0.45314669609069824
dc_{21} = 210	vl_{21} = -0.5099478363990784	vr_{21} = -0.4859651029109955
dc_{22} = 220	vl_{22} = -0.5276192426681519	vr_{22} = -0.5074233412742615
dc_{23} = 230	vl_{23} = -0.5528641939163208	vr_{23} = -0.5187835693359375
dc_{24} = 240	vl_{24} = -0.5680111646652222	vr_{24} = -0.5314059853553772

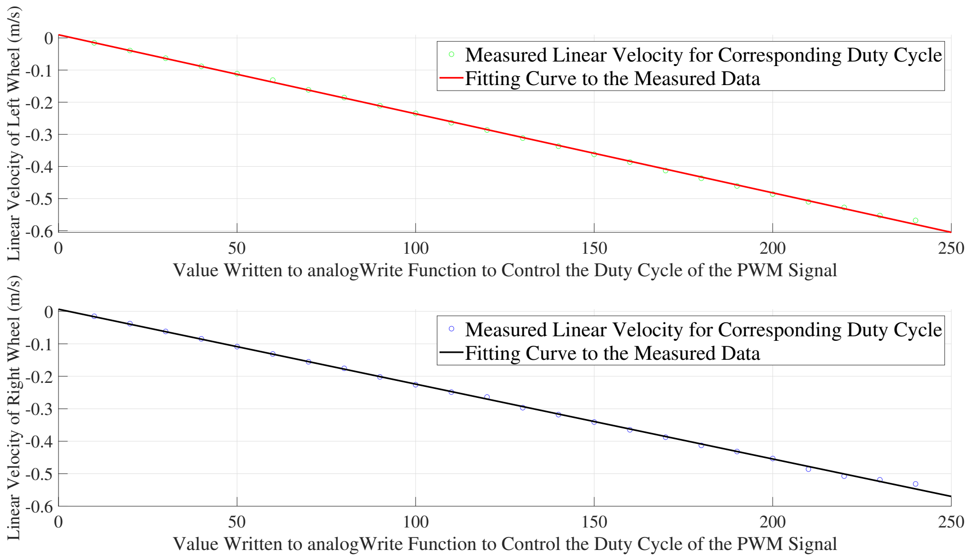


Figure A.1: The relationship between the duty cycle of PWM signal and the operation speed of wheels in backward direction.

Bibliography

[1] Arduino mega 2560 datasheet. [Online]. Available: `https://www.robotshop.com/media/files/pdf/arduinomega2560datasheet.pdf`.

[2] Unmanned ground vehicle - Wikipedia. [Online]. Available: `https://en.wikipedia.org/wiki/Unmanned_ground_vehicle`.

[3] D. W. Gage, "UGV history 101: A brief history of Unmanned Ground Vehicle (UGV) development efforts," tech. rep., Naval Command Control and Ocean Surveillance Center RDT and E Div San Diego CA, 1995.

[4] H. Li, C. Xu, Q. Xiao, and X. Xu, "Visual navigation of an autonomous robot using white line recognition," in *Proceedings of the IEEE International Conference on Robotics and Automation*, vol. 3, pp. 3923–3928, 2003.

[5] H. Surmann, A. Nüchter, and J. Hertzberg, "An autonomous mobile robot with a 3D laser range finder for 3D exploration and digitalization of indoor environments," *Robotics and Autonomous Systems*, vol. 45, no. 3-4, pp. 181–198, 2003.

[6] J. Jia, W. Chen, and Y. Xi, "Design and implementation of an open autonomous mobile robot system," in *Proceedings of the IEEE International Conference on Robotics and Automation*, vol. 2, pp. 1726–1731, 2004.

[7] Dingo Indoor Mobile Robot - Clearpath Robotics. [Online]. Available: `https://clearpathrobotics.com/dingo-indoor-mobile-robot/`.

[8] Turtlebot. [Online]. Available: `https://www.turtlebot.com/`.

[9] M. H. Hebert, C. E. Thorpe, and A. Stentz, *Intelligent unmanned ground vehicles: autonomous navigation research at Carnegie Mellon.* Springer Science & Business Media, 2012.

[10] M. Yagimli and H. S. Varol, "Mine detecting GPS-based unmanned ground vehicle," in *Proceedings of the 4th International Conference on Recent Advances in Space Technologies*, pp. 303–306, 2009.

[11] P. A. Raj and M. Srivani, "Internet of robotic things based autonomous fire fighting mobile robot," in *Proceedings of the IEEE International Conference on Computational Intelligence and Computing Research*, pp. 1–4, 2018.

[12] E. Kayacan, H. Ramon, and W. Saeys, "Robust trajectory tracking error model-based predictive control for unmanned ground vehicles," *IEEE/ASME Transactions on Mechatronics*, vol. 21, no. 2, pp. 806–814, 2015.

[13] M. H. Ko, B.-S. Ryuh, K. C. Kim, A. Suprem, and N. P. Mahalik, "Autonomous greenhouse mobile robot driving strategies from system integration perspective: Review and application," *IEEE/ASME Transactions on Mechatronics*, vol. 20, no. 4, pp. 1705–1716, 2014.

[14] Y.-S. Yao and R. Chellapa, "Selective stabilization of images acquired by unmanned ground vehicles," *IEEE Transactions on Robotics and Automation*, vol. 13, no. 5, pp. 693–708, 1997.

[15] H. Lim, Y. Kang, J. Kim, and C. Kim, “Formation control of leader following unmanned ground vehicles using nonlinear model predictive control,” in *Proceedings of the IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, pp. 945–950, 2009.

[16] S. Çaşka and A. Gayretlı, “An algorithm for collaborative patrolling systems with unmanned air vehicles and unmanned ground vehicles,” in *Proceedings of the 7th International Conference on Recent Advances in Space Technologies*, pp. 659–663, 2015.

[17] H. Yu, K. Meier, M. Argyle, and R. W. Beard, “Cooperative path planning for target tracking in urban environments using unmanned air and ground vehicles,” *IEEE/ASME Transactions on Mechatronics*, vol. 20, no. 2, pp. 541–552, 2014.

[18] J. Chen, X. Zhang, B. Xin, and H. Fang, “Coordination between unmanned aerial and ground vehicles: A taxonomy and optimization perspective,” *IEEE Transactions on Cybernetics*, vol. 46, no. 4, pp. 959–972, 2015.

[19] R. Siegwart and I. R. Nourbakhsh, *Introduction to autonomous mobile robots.* Cambridge, Massachusetts: MIT Press, 2004.

[20] T. Luettel, M. Himmelsbach, and H.-J. Wuensche, “Autonomous ground vehicles—concepts and a path to the future,” *Proceedings of the IEEE*, vol. 100, no. Special Centennial Issue, pp. 1831–1839, 2012.

[21] S. Cooper and H. Durrant-Whyte, “A kalman filter model for gps navigation of land vehicles,” in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, vol. 1, pp. 157–163, 1994.

[22] J. Alberts, D. Edwards, T. Soule, M. Anderson, and M. O’Rourke, “Autonomous navigation of an unmanned ground vehicle in unstructured forest terrain,” in

Proceedings of the ECSIS Symposium on Learning and Adaptive Behaviors for Robotic Systems, pp. 103–108, 2008.

[23] A. Martins, G. Amaral, A. Dias, C. Almeida, J. Almeida, and E. Silva, “Tigre—An autonomous ground robot for outdoor exploration,” in *Proceedings of the 13th International Conference on Autonomous Robot Systems*, pp. 1–6, 2013.

[24] H. Moon, J. Lee, J. Kim, and D. Lee, “Development of unmanned ground vehicles available of urban drive,” in *Proceedings of the IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, pp. 786–790, 2009.

[25] C. Garcia-Saura, “Self-calibration of a differential wheeled robot using only a gyroscope and a distance sensor,” *arXiv:1509.02154 [cs.RO]*, 2015.

[26] A. Chilian, H. Hirschmüller, and M. Görner, “Multisensor data fusion for robust pose estimation of a six-legged walking robot,” in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2497–2504, 2011.

[27] E. Royer, J. Bom, M. Dhome, B. Thuilot, M. Lhuillier, and F. Marmoiton, “Outdoor autonomous navigation using monocular vision,” in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1253–1258, 2005.

[28] D. Nistér, O. Naroditsky, and J. Bergen, “Visual odometry for ground vehicle applications,” *Journal of Field Robotics*, vol. 23, no. 1, pp. 3–20, 2006.

[29] D. Silver, B. Sofman, N. Vandapel, J. A. Bagnell, and A. Stentz, “Experimental analysis of overhead data processing to support long range navigation,” in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2443–2450, 2006.

[30] G. Oriolo, G. Ulivi, and M. Vendittelli, “Real-time map building and navigation for autonomous robots in unknown environments,” *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 28, no. 3, pp. 316–333, 1998.

[31] H. Lategahn, A. Geiger, and B. Kitt, “Visual SLAM for autonomous ground vehicles,” in *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 1732–1737, 2011.

[32] N. Sariff and N. Buniyamin, “An overview of autonomous mobile robot path planning algorithms,” in *Proceedings of the 4th Student Conference on Research and Development*, pp. 183–188, 2006.

[33] T. W. Manikas, K. Ashenayi, and R. L. Wainwright, “Genetic algorithms for autonomous robot navigation,” *IEEE Instrumentation & Measurement Magazine*, vol. 10, no. 6, pp. 26–31, 2007.

[34] A. Kaplan, P. Uhing, N. Kingry, and R. Dai Adam, “Integrated path planning and power management for solar-powered unmanned ground vehicles,” in *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 982–987, 2015.

[35] A. F. Foka and P. E. Trahanias, “Predictive autonomous robot navigation,” in *Proceedings of the IEEE/RSJ international conference on intelligent robots and systems*, vol. 1, pp. 490–495, 2002.

[36] ROS robot. [Online]. Available: `https://robots.ros.org/`.

[37] A. Marquez, “Implementation of an autonomous small-scale car with indoor positioning using UWB and IMU,” Master’s thesis, University of Windsor, 2017.

[38] ROS/Tutorials/CreatingPackage - ROS Wiki. [Online]. Available: http://wiki.ros.org/Packages.

[39] ROS/Tutorials/UnderstandingNodes - ROS Wiki. [Online]. Available: http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes.

[40] ROS/Tutorials/UnderstandingTopics - ROS Wiki. [Online]. Available: http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics.

[41] ROS/Tutorials/UnderstandingServicesParams - ROS Wiki. [Online]. Available: http://wiki.ros.org/ROS/Tutorials/UnderstandingServicesParams.

[42] X. Dai, H. Zhang, and Y. Shi, "Autonomous navigation for wheeled mobile robots - a survey," in *Proceedings of the Second International Conference on Innovative Computing, Information and Control*, pp. 551–551, Sept. 2007.

[43] Tf - ROS Wiki. [Online]. Available: http://wiki.ros.org/tf.

[44] Gmapping - ROS Wiki. [Online]. Available: http://wiki.ros.org/gmapping.

[45] Move_base - ROS Wiki. [Online]. Available: http://wiki.ros.org/move_base.

[46] Amcl - ROS Wiki. [Online]. Available: http://wiki.ros.org/amcl.

[47] Moment of Inertia–Cylinder – from Eric Weisstein's World of Physics. [Online]. Available: http://scienceworld.wolfram.com/physics/MomentofInertiaCylinder.html.

[48] P. Fankhauser, M. Bloesch, D. Rodriguez, R. Kaestner, M. Hutter, and R. Siegwart, "Kinect v2 for mobile robot navigation: Evaluation and modeling," in *Proceedings of the International Conference on Advanced Robotics*, pp. 388–394, 2015.

[49] D. P. Atherton, “Almost six decades in control engineering,” *IEEE Control Systems Magazine*, vol. 34, pp. 103–110, Dec. 2014.

[50] F. Haugen, “Comparing PI tuning methods in a real benchmark temperature control system,” 2010.

[51] T. Hägglund and K. J. Åström, “Revisiting the ziegler-nichols tuning rules for pi control,” *Asian Journal of Control*, vol. 4, no. 4, pp. 364–380, 2002.

[52] Dead reckoning - Wikipedia. [Online]. Available: `https://en.wikipedia.org/wiki/Dead_reckoning`.

[53] E. Olson, “A primer on odometry and motor control,” *Electronic Group Discuss*, 2004.