SSRG International Journal of Mechanical Engineering
ISSN: 2348 – 8360 /doi:10.14445/23488360/IJME-V7I11P108

Volume 7 Issue 11, 57-62, November 2020
© 2020 Seventh Sense Research Group®

The study of tracking control for autonomous vehicle

Wen-Kung Tseng[#1], Hou-Yu Chen[*2]

Graduate Institute of Vehicle Engineering, National Changhua University of Education, Taiwan

Abstract - *The research and development of autonomous vehicles are growing immensely. With the upgrading of hardware equipment, autonomous vehicles are becoming more advanced, and their development costs are also increasing. This study's main objective is to construct an autonomous vehicle tracking control system based on Robot Operating System (ROS), integrating various ROS feature suites, ROS library for self-driving control, SLAM, path planning, and obstacle avoidance. The STM32 ARM microcontroller is used to drive the autonomous vehicle deceleration motor, and ROS is installed on the Raspberry Pi 3B+ with low-cost optical LIDAR (light detection and ranging) and Inertial Measurement Unit (IMU). These enable the autonomous vehicle to complete functions such as positioning, map construction, autonomous navigation, and arrive at the desired destination through the planned path with obstacle avoidance.*

Keywords — *autonomous vehicle, robot operating system, Raspberry Pi, LIDAR, inertial measurement unit.*

I. INTRODUCTION

With technology development, autonomous vehicles have attracted great research attention, and hardware equipment upgrades equipped autonomous vehicles with attractive functions. As a result, the cost of autonomous vehicle development is rising. Nowadays, various studies of autonomous vehicles' development have been reported [1], but few used low-cost computers for control of autonomous vehicles.

This study uses low-cost computers with ROS [2] for the control of the autonomous vehicle. Three different mapping methods, G mapping SLAM [3], Karto SLAM [4], and Hector SLAM [5], were compared, and the one with minimum power consumption and optimized mapping performance was selected. The map developed by the selected method was used for self-guided navigation, positioning, and obstacle avoidance functions of autonomous vehicles, combined with cameras. In this way, autonomous vehicles are enabled with the track-trajectory function.

In 2009, Quigley *et al.* proposed ROS, which is an open-source operating system. This study discussed how ROS is associated with the robot software framework and described some application software using ROS [2]. In 2010, Wongwirat *et al.* proposed tracking the moving robot using the Inertial Measurement Unit (IMU). The IMU provides x, y, and z coordinates. In this study, experiments were conducted to verify the IMU signals while tracking the moving robot. The results demonstrated the limitations of low-cost IMU sensors inaccuracy [6].

In 2011, Kohlbrecher *et al.* proposed a fast e-learning system that takes up grid diagrams. This system requires reduced computing resources as it combines a robust scan matching approach using a LIDAR system with a 3D attitude estimation system based on inertial sensing. The use of fast approximation of map gradients and multi-resolution meshes enables reliable positioning and mapping in various challenging environments. Additionally, its applicability in embedded handheld mapping systems was also mentioned [5].

In 2013, Foote *et al.* presented the tf: conversion library, which aims to provide a tracking coordinate system and a standard method for data transformation across the system so that individual component users can be confident that the data is in the desired coordinate system without having to know all coordinate systems in the system [7]. In 2016, Hess *et al.* proposed a method for LIDAR indoor SLAM mapping. This method achieved real-time mapping and loop closure at a 5 cm resolution [8]. In 2018, Ueda *et al.* described the development of ROS robots using Raspberry Pi. The article consists of three parts: development environment, ROS basics, and ROS applications [9].

II. THEORY OF TRACKING CONTROL FOR AUTONOMOUS VEHICLE

ROS (Robot Operating System) is different from the traditional Operating System such as Microsoft Windows, Android, Apple macOS, BXD, and Linux. It is based on the Linux operating system Ubuntu. The main function of the system is as a communication framework between the various components of the robot. Taking the robot arm as an example, to move the robot arm to a certain position, it is necessary to control its motor and its sensors to avoid obstacles. Many similar functions like this one in the ROS allow the motor control program and sensor control program to communicate with each other. An example can be shown in Fig. 1.

ROS is an excellent hierarchical framework for robotics, consisting of three levels: Filesystem Level, Computation Graph Level, and Communication Level [9]. Filesystem Level: ROS internal structure, file structure, and required core files are at this level. Understanding the ROS file system is the foundation for getting started with ROS. The structure of a ROS program is several folders that are distinguished by different functions. The general folder structure is:

Workspace folder (workspace) → Source File Space folder (src), Compile Space folder (build) and Development Space folder (devel); source file space folder, and further place the feature pack.

Computation Graph Level: Refers to communication between nodes, such as Fig. 2. ROS creates a network that connects all the programs. Through the interaction of these nodes, information is shared, and information is obtained from other nodes.

Communication Level: This refers to the acquisition and sharing of ROS resources, such as ROS Wiki, the ROS Knowledge Center. It includes various ROS documentation and tutorials. Github is the repository for ROS related documents and packages, and ROS Answers allows users to ask questions about ROS and its related areas. Through an independent online community, we can share and access knowledge, algorithm, and source code. The support of the open-source community enables ROS systems to grow rapidly.

Localization, Mapping, and Path Planning have three issues that need to be resolved to enable the robot to navigate autonomously. Simultaneous Localization and Mapping (SLAM) are the intersections of positioning and mapping. SLAM technology uses data obtained by sensors to calculate its closest position and movement in an unknown environment and continuously construct and update two-dimensional or three-dimensional spatial information for that environment. The robot uses SLAM technology to obtain effective spatial information and then plan motion through ROS Navigation for autonomous navigation. The following is ROS Navigation and a brief description of the three SLAMs used in this lab.

G mapping is the most widely used 2D slam method using 2D plane laser Rangefinder and mileometer (Odometry) sensor data as input sources for the algorithm to produce a two-dimensional map. The G mapping SLAM algorithm was developed based on the Rao-Blackwellized Particle Filter (RBPF). The RBPF was proposed to solve grid-based SLAM problems, and it requires odometry information and the sensor's observations(i.e., scans).

G mapping was developed by Grisetti et al. [6] to improve the performance of the RBPF-based SLAM. The main idea of the RBPF is in an unknown environment; the robot starts from the starting position. During the movement, the odometer is used to record its motion information u_{t-1}. and environmental information Z_t obtained by external sensors, estimate the robot trajectory x_t and construct an environmental map m. At the same time, use the created map and sensor information to achieve self-positioning. This joint posterior is denoted as $P(x_t, m \mid Z_t, u_{t-1})$ and can be factorized into Eq. (1) through the Rao-Blackwellization technique [9]:

$$P(x_t, m \mid Z_t, u_{t-1}) = P(m \mid x_t, Z_t) \cdot P(x_t \mid Z_t, u_{t-1}) \quad (1)$$

Using the odometry and observation information, the algorithm's estimation with the highest probability in the associated map can be obtained. G mapping allows you to build indoor maps in real-time, with less computational power and more accurate calculations required to build small scene maps. As the scene increases, the number of particles required will also increase because each particle carries a map. Therefore, the amount of memory and calculation required when building a large map will increase, so G mapping is less suitable for building a large scene map. Because there is no Loop Closure Detection, the map may be misplaced when the loop is closed. Although increasing the number of particles can close the map, it is at the expense of more calculation and memory consumption.

Karto SLAM is a graph-based SLAM approach developed by SRI International's Karto Robotics, extended for ROS by using a highly-optimized and noniterative Cholesky matrix decomposition for sparse linear systems as its solver [5]. Graph optimization is the mainstream optimization method in visual slam. The so-called graph optimization is to express the general optimization problem in the form of a graph. The graph consists of vertex and edge. In common slam problems, the robot's position is a vertex, and the relationship between the positions at different times constitutes an edge. The vertices and edges that are continuously accumulated form a graph structure and the graph is optimized. The goal is to satisfy the edges by adjusting the vertices' pose as much as possible. Among them, building diagrams from the sensor accumulates information is called the front end in slam, and the optimization process of adjusting the position to meet constraints becomes the back-end.

Hector SLAM combines a 2D SLAM system based on robust scan matching and 3D navigation technique using an inertial sensing system [6]. Hector SLAM mainly uses the Gaussian Newton method to solve the problem of scan-matching. The algorithm seeks to find the optimum alignment of the laser scan's endpoints with the constructed map by finding the rigid transformation ξ=$(px, py, \psi)^T$ that minimizes:

$$\xi^* = \arg\underset{\xi}{min} \sum_{i=1}^{n} \left[1 - M\left(S_i(\xi)\right)\right]^2 \quad (2)$$

Where the function returns the map value at $S_i(\xi)$ which is the world coordinates of $M(S_i(\xi))$ the scan point. To minimize Eq. (2), given a starting estimate of ξ and the step transformation Δξ, the value of Eq. (3) should first be minimized [9].

$$\sum_{i=1}^{n}[1 - M(S_i(\xi + \Delta\xi))]^2 \to 0 \qquad (3)$$

Therefore, applying first-order Taylor expansion to $M(S_i(\xi + \Delta\xi))$ and setting the partial derivative concerning $\Delta\xi$ to zero yields the Gauss-Newton equation for the minimization problem [9]:

$$\Delta\xi = \quad H^{-1}\sum_{i=1}^{n}\left[\nabla M(S_i(\xi))\frac{\partial S_i(\xi)}{\partial \xi}\right]^T \ [1 - M(S_i(\xi))] \qquad (4)$$

Where:

$$H = \left[\nabla M(S_i(\xi))\frac{\partial S_i(\xi)}{\partial \xi}\right]^T \left[\nabla M(S_i(\xi))\frac{\partial S_i(\xi)}{\partial \xi}\right] \qquad (5)$$

Hector SLAM requires a higher radar update frequency. The ideal speed for the self-driving vehicle speed control during the mapping process is relatively low. Because it has no loops (loop close), no mileage is required, so aerial drones and ground trolleys can be used in the map of uneven areas. Laser beam raster is optimized to estimate the representation of laser points in the map and the probability of occupying the mesh; the state estimation in navigation is added to the Inertial Measurement System (IMU). Navigation Stack is a set of related programs that allows robots or automated vehicles to move steadily in space. It is an important tool for using ROS. In general use, we assume that you have already built a 2D map in the environment with SLAM, then when you start navigation, you can use the built map as a static map to the Global planner. When we have started navigation, we need to give the robot a goal, which can be given via Rviz or send a message tomove_base_simple/goal this topic.

When a goal is sent, the Global planner first creates a track that does not encounter obstacles in the static map, the Global Path. However, with this Global path, the robot can only navigate a static environment, because as long as the scene is different from a static map, the Global path may hit an obstacle. To solve this problem, the approach here is to use the Global path as the approximate route and then add a local planner for dynamic obstacle avoidance. This local planner must always be aware of any new obstacles, so the local planner needs sensor data as input.

In Navigation Stack, the algorithm used by default is the DWA (Dynamic-Window Approach) proposed by S.Thrun [9]. This method is a velocity space search method that considers robot dynamics. It is mainly divided into three stages. The first stage eliminates the speed that the robot cannot reach. In the second stage, the speed that could not be stopped before hitting an obstacle was eliminated. In the third stage, DWA evaluates an admissible velocity set by maximizing its objective function shown in Eq. (6). DWA will predict the results of each speed pair candidate for the final heading angle, the minimum distance to obstacles, and linear speed value, and choose the best speed pair by maximizing the objective function:

$$G(V,w) = \sigma[\alpha \text{ heading}(V,w) + \beta \text{ dist}(V,w) + \gamma \text{ velocity}(V,w)] \qquad (6)$$

The heading function $\text{heading}(V,w)$ represents the approximate value of the angle to the target, and its value will increase when the heading of the robot approaches the target position. The purpose of the distance function $\text{dist}(V,w)$ is to promote safe navigation. It calculates the minimum distance from the trajectory obtained from the speed pair to the obstacle. The velocity function $\text{velocity}(V,w)$ calculates the linear velocity values in the velocity set. Coefficients α ⋅ β and γ are weights of these functions, and σ is a smoothing operator. Maximizing this objective function will cause the safety trajectory to reach the objective as soon as possible.

The main idea of this algorithm is that the robot generates a lot of random speed options, then simulates the robot to follow those speeds for a short period of time, and then scores each speed option's position. The basis for scoring includes distance to the goal, distance to obstacles, etc. The algorithm selects the speed option with the highest score as the local plan at this time, and it will become a command and be sent to /cmd_vel to let the robot move. After waiting for the next position, the algorithm randomly generates multiple speed options until the global plan is completed, as shown in Fig. 3.

III. EXPERIMENT AND RESULTS

This section describes the system tests and experimental results associated with autonomous vehicles. First, an overview of the experiment environment has been presented. A comparison of the three SLAM mapping effects and system energy consumption have been evaluated. Then the self-driving navigation and obstacle avoidance has been verified. Finally, self-driving line-tracking performance has been presented. The experiment has also been performed in the laboratory. Due to many chairs and personal belongings in the seating area, the part close to the seating area should be enclosed with cardboard boxes to reduce the interference during the experiment.

This experiment uses three different SLAM algorithms, G mapping, Karto, and Hector, for verification. We use s-tui during the mapping process to monitor the CPU Utilization and compare the mapping results to select a SLAM algorithm that is suitable for Raspberry Pi with low energy consumption and high accuracy. The mapping and CPU utilization results are shown in Fig. 4(a)~(f), in the order of G mapping, Karto, Hector. After comparing the resulting map, it can be seen that the Karto mapping effect is similar to the actual map. No skew generated. CPU utilization is maintained at 30~ 40% level, and the system consumes less energy. G mapping effect is suboptimal, the end of the map is slightly skewed, footing and other map information scanning are less complete, CPU utilization fluctuates between 40 and 60%, and the system has high energy consumption. Hector mapping effect is poor; the end of the map is seriously skewed. The table footing and other map information are not completed. CPU utilization is stable at 60%, and the system energy consumption is high.

Therefore this experiment uses the Karto diagram results from the previous paragraph for navigation and obstacle avoidance. The experiment environment is divided into three phases. The first part is the navigation. Given coordinate points 0 and 1 on Rviz, the navigation node plans the path. After reaching coordinate 0, it moves to coordinate 1, as shown in Fig. 5(a)~(e). When there is a fixed barrier between the self-drive and the specified coordinate point, as shown in Fig. 6, navigation systems automatically map the path to avoid obstacles. It also selected various paths to reach the target, as shown in Fig. 7(a)~(e).

IV. CONCLUSIONS

This study's main objective is to construct a tracking control system for autonomous vehicles, with functions of SLAM mapping, path planning, obstacle avoidance, and trajectory tracking. Meanwhile, low cost, low power consumption, and utilization of the Robot Operating System, which facilitates the development of various programs, could be achieved. The Raspberry Pi was employed as the drive computer, and low-cost optical radar was employed for obstruction detection. Combined with inertial sensor signals, accurate positioning of the vehicle was achieved.

G mapping SLAM, Karto SLAM, and Hector SLAM were compared with each other in terms of accuracy and power consumption. The results demonstrated that Karto SLAM has high accuracy and low power consumption. Hence, it is an ideal candidate for the development of Raspberry Pi autonomous vehicles. It also operated accurately on navigation, obstacle avoidance, and trajectory tracking. However, the combination of face recognition and control of autonomous vehicles is beyond the proposed system's computing power. Meanwhile, scanning using a single optical radar is limited by issues such as those obstacles in different planes that cannot be detected. Future studies may fuse scanning results by two radars on different planes to obtain improved obstructions scanning capability.

REFERENCES

[1] C. Chatzikomis, A. Sorniotti, P. Gruber, M. Zanchetta, D. Willans, and B. Balcombe, -Comparison of Path Tracking and Torque-Vectoring Controllers for Autonomous Electric Vehicles, IEEE Transactions on Intelligent Vehicles, (2018) 559-570.

[2] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. B. Foote, J. Leibs, -ROS: An open-source robot operating system, Proc. ICRA Open-Source Softw. The workshop, 2009.

[3] G. Grisetti, C. Stachniss, and W. Burgard, -Improved Techniques for Grid Mapping With Rao-Blackwellized Particle Filters, IEEE Transactions on Robotics, (2009) 34-46.

[4] R. Vincent, B. Limketkai and M. Eriksen, -Comparison of indoor robot localization techniques in the absence of gps, Detection, and Sensing of Mines Explosive Objects Obscured Targets XV, 2010.

[5] S. Kohlbrecher, J. Meyer, O. Von Stryk, U. Klingauf, -A Flexible and Scalable SLAM System with Full 3D Motion Estimation, Safety Security and Rescue Robotics (SSRR), (2011) 155-160.

[6] O. Wongwirat and C. Chaiyarat, -A Position Tracking Experiment of Mobile Robot with Inertial Measurement Unit (IMU), International Conference on Control, Automation and Systems, pp. 27-30, 2010.

[7] T. Foote, -tf: The transform library, Technologies for Practical Robot Applications (TePRA), (2013) 1-6.

[8] W. Hess, D. Kohler, H. Rapp, and D. Andor, -Real-time loop closure in 2D LIDAR SLAM, Robotics, and Automation (ICRA), (2016) 1271-1278.

[9] Lentin Joseph, -ROS Robotics Projects, Packt Publishing Ltd, 2018.

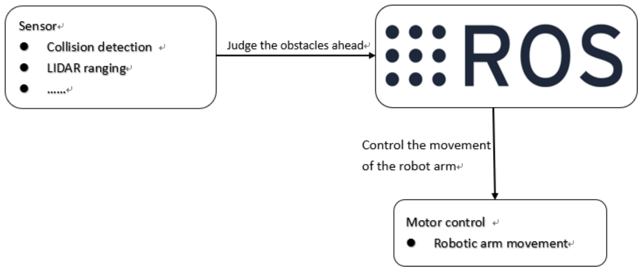


Fig. 1 ROS control process

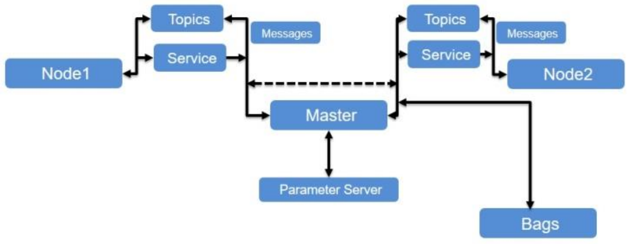


Fig. 2 ROS Computational Level

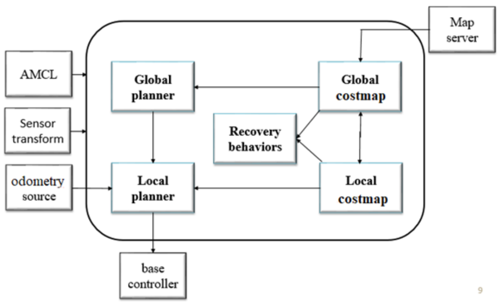
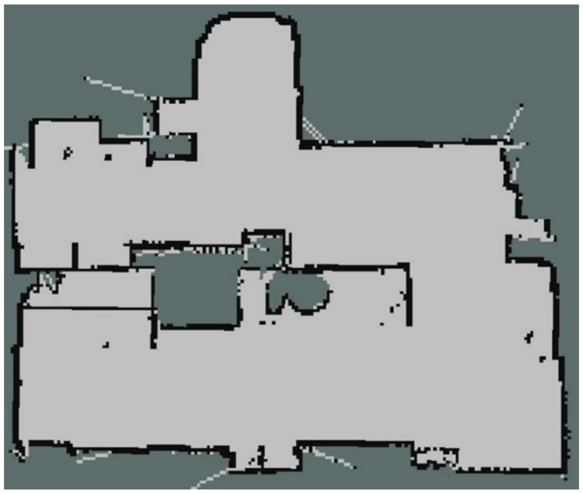
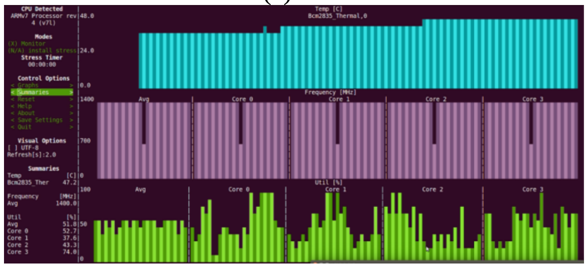


Fig. 3 Navigation stack.

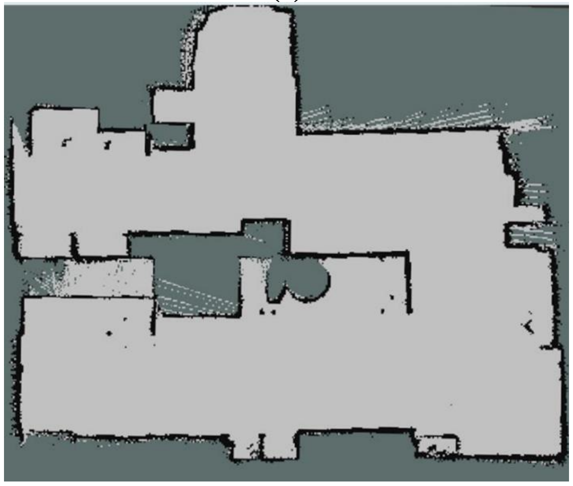


(a)

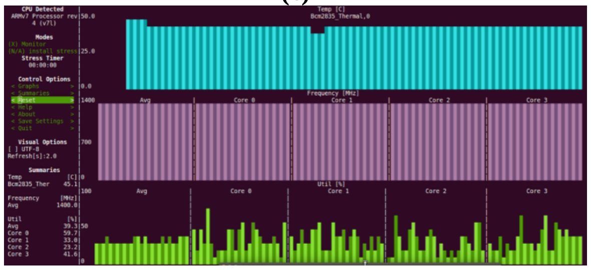


```
    CPU Detected
 ARMv7 Processor rev
      4 (v7l)

       Modes
(X) Monitor
(N/A) install stress
   Stress Timer
     00:00:00

  Control Options
 < Graphs         >
 < Summaries      >
 < Reset          >
 < Help           >
 < About          >
 < Save Settings  >
 < Quit           >

  Visual Options
[ ] UTF-8
Refresh[s]:2.0

     Summaries
Temp             [C]
Bcm2835_Ther    47.2

Frequency      [MHz]
Avg           1400.0

Util             [%]
Avg             51.8
Core 0          52.7
Core 1          37.6
Core 2          43.3
Core 3          74.0
```

(b)

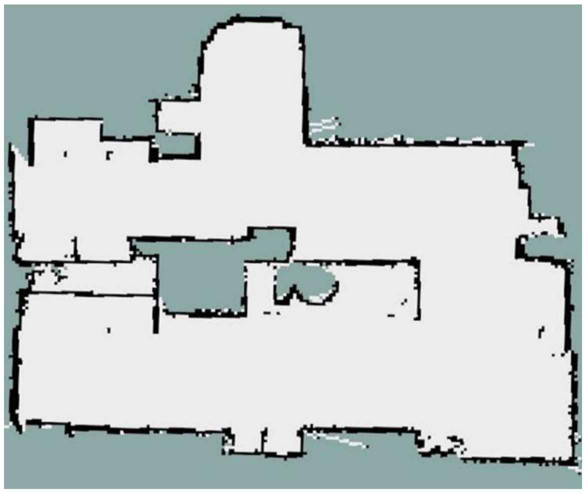


(c)

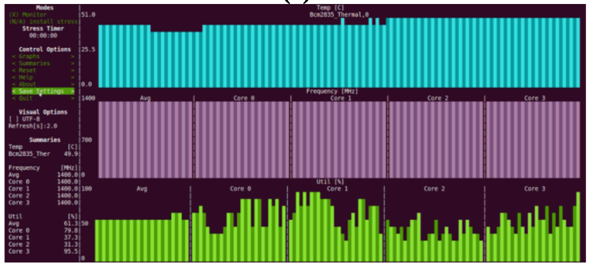


```
    CPU Detected
 ARMv7 Processor rev
      4 (v7l)

       Modes
(X) Monitor
(N/A) install stress
   Stress Timer
     00:00:00

  Control Options
 < Graphs         >
 < Summaries      >
 < Reset          >
 < Help           >
 < About          >
 < Save Settings  >
 < Quit           >

  Visual Options
[ ] UTF-8
Refresh[s]:2.0

     Summaries
Temp             [C]
Bcm2835_Ther    45.1

Frequency      [MHz]
Avg           1400.0

Util             [%]
Avg             39.3
Core 0          59.7
Core 1          33.0
Core 2          23.2
Core 3          41.6
```

(d)



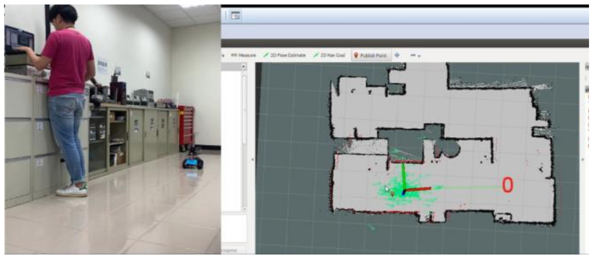
(e)



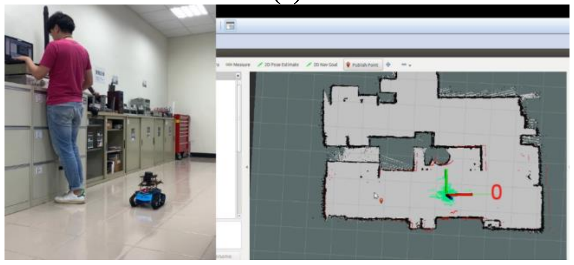
```
       Modes
(X) Monitor
(N/A) install stress
   Stress Timer
     00:00:00

  Control Options
 < Graphs         >
 < Summaries      >
 < Reset          >
 < Help           >
 < About          >
 < Save Settings  >
 < Quit           >

  Visual Options
[ ] UTF-8
Refresh[s]:2.0

     Summaries
Temp             [C]
Bcm2835_Ther    49.9

Frequency      [MHz]
Avg           1400.0
Core 0        1400.0
Core 1        1400.0
Core 2        1400.0
Core 3        1400.0

Util             [%]
Avg             61.3
Core 0          79.8
Core 1          37.3
Core 2          31.3
Core 3          95.5
```

(f)

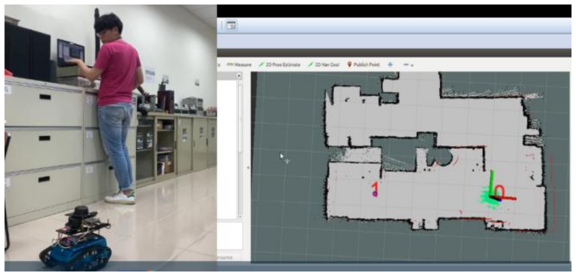
Fig. 4 Results of the mapping and CPU utilization (a) G mapping SLAM results (b) G mapping SLAM CPU utilization (c) Karto SLAM results (d) Karto SLAM CPU utilization (e) Hector SLAM results (f) Hector SLAM CPU utilization.

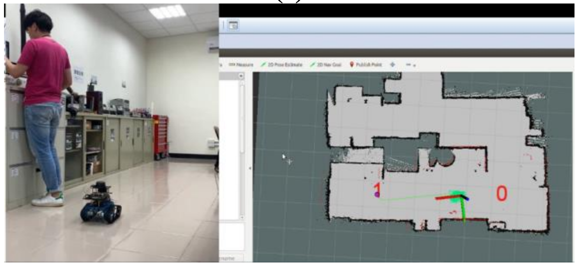


(a)



(b)


(c)


(d)

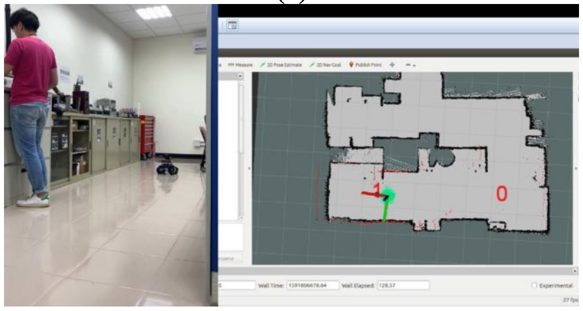

(e)

Fig. 5 Navigation schematic diagram.

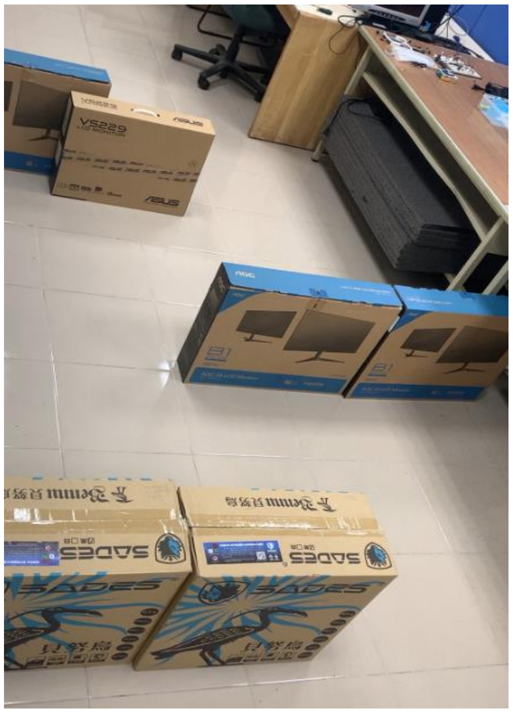
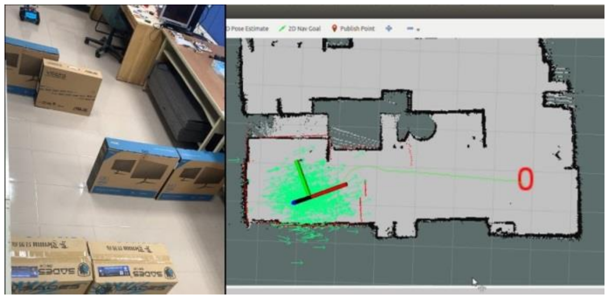
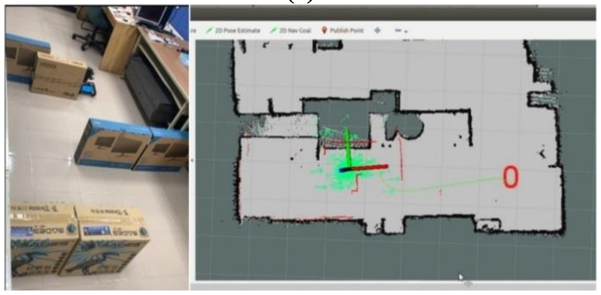
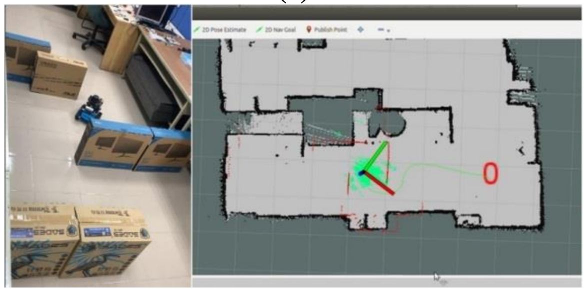
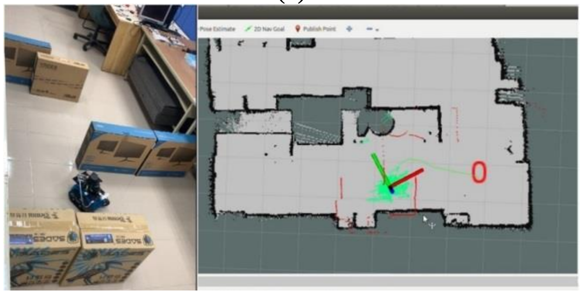


Fig. 6 Obstructions with cartons in the lab area.


(a)


(b)


(c)


(d)

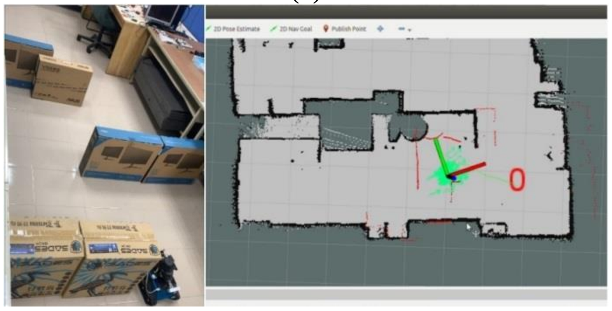

(e)

Fig. 7 Mapping the path to avoid obstacles.