

Research on Low-Speed Unmanned Vehicle Obstacle Avoidance Based on FeiTengPai

Lang'en Ni

*Department of Urban and Regional Planning, University at Buffalo, The State University of New York, New York, USA
3594401049@qq.com*

Abstract. This paper focuses on low-speed unmanned ground vehicles equipped with the Leishen C16 lidar and the FT-Pi embedded platform. Centering on the problems of autonomous navigation and dynamic obstacle avoidance in unknown environments, it systematically expounds the overall scheme design, engineering implementation and real-vehicle test verification of the system. The system takes a 24-dimensional state vector as input — 20 dimensions are lidar ranging data processed by sector partitioning, and the remaining 4 dimensions are the robot's own state information — and outputs continuous speed control commands accordingly. To coordinate the inherent contradictions among approaching the target, avoiding collision and motion smoothness, a new reward function is designed. At the execution level, the system communicates with the underlying motor drive module through a UART serial port with a baud rate of 115200, and a custom service interface is built to receive speed commands. Real-vehicle test results show that the system runs stably, with a maximum linear velocity of 80 mm/s and a maximum angular velocity of 1.57 rad/s. The above results fully demonstrate that the scheme integrating deep reinforcement learning and ROS2 can effectively support robust low-speed autonomous navigation tasks on embedded platforms with limited computing resources.

Keywords: Deep Reinforcement Learning, ROS2, Unmanned Ground Vehicle, Lidar

1. Introduction

Achieving autonomous navigation in unknown and dynamically changing environments is a fundamental technical challenge for unmanned ground vehicles (UGVs). In recent years, with the rapid development of robotics and artificial intelligence, the application scenarios of UGVs have been continuously expanded, widely covering fields such as logistics distribution, security inspection, agricultural operations and disaster rescue. In these practical applications, the capabilities of real-time obstacle avoidance and target-oriented motion planning are particularly core, which directly relate to the safety and efficiency of system operation. However, existing methods often face limitations such as excessive computational overhead and insufficient adaptability to unstructured environments.

At the same time, deploying deep reinforcement learning (DRL) models on embedded systems with limited computing resources is still a very challenging engineering problem, especially under

the conditions requiring real-time sensor fusion and low-latency control loops, where this contradiction becomes more prominent. To address the above problems, this study designs and implements a complete low-speed unmanned ground vehicle system based on the domestic embedded platform FT-Pi and the Leishen C16 16-channel lidar. The system software is built on the ROS2 Foxy framework, making full use of its modular and distributed communication capabilities to support the overall functional architecture [1].

2. Literature review

2.1. Navigation methods

Traditional autonomous navigation systems usually integrate localization, mapping, path planning and motion control into the same framework. Its operation process is roughly as follows: the robot obtains environmental information through lidar or visual sensors, builds an occupancy grid map or feature point map accordingly, then plans the global path through a search algorithm, and implements local trajectory correction during travel to cope with sudden obstacles [2]. Although such methods are relatively mature and have good interpretability, their operation often highly relies on accurate environmental maps and is easily disturbed by cumulative odometry errors. Especially in highly dynamic scenarios, the frequency of replanning may not be sufficient to ensure driving safety [3].

2.2. Reinforcement learning for navigation

Reinforcement learning provides an alternative path, which learns control policies through trial-and-error interaction between the agent and the environment. Deep Q-Network (DQN) was originally designed mainly for discrete action spaces, but navigation tasks usually require output of continuous linear velocity and angular velocity commands. Existing studies have repeatedly verified the effectiveness of the TD3 algorithm in simulation environments and real robot platforms, and its inputs are mostly raw lidar point clouds or visual images. However, most of the above work relies on high-performance GPUs and powerful workstations, which is difficult to implement directly on embedded systems with limited computing resources [4].

2.3. Embedded robotics technology

Embedded platforms represented by NVIDIA Jetson, Raspberry Pi and domestic FT-Pi have achieved a good balance between cost and computing performance. This study selects FT-Pi as the hardware carrier. The platform is equipped with a processor based on ARM architecture and supports the Ubuntu operating system and ROS2 Foxy development environment. The core difficulty lies in how to optimize the DRL inference process under the premise of meeting real-time constraints (control cycle of 50–100 ms) while ensuring that navigation performance does not degrade significantly [5].

3. Detailed system design

3.1. System architecture selection

This system adopts a distributed communication architecture based on ROS2 (Robot Operating System 2). Combined with the hierarchical design concept, the system is divided into a perception

layer, a decision-making layer and an execution layer, to realize decoupling and efficient collaboration of each module.

Perception layer: Responsible for obtaining raw data from hardware (lidar, underlying controller) and publishing it as ROS2 topics. It mainly includes the `lslidar_driver` node and the `truth` and `odom` nodes in the `uart` package.

Decision-making layer: The core of the system, which subscribes to data published by the perception layer, makes intelligent decisions through the deep reinforcement learning algorithm (TD3), and calculates desired motion commands. It is mainly the `run_real_robot` node in the `td3` package.

Execution layer: Receives commands issued by the decision-making layer, converts them into commands executable by the underlying hardware, and sends them to the robot chassis through a physical interface (serial port). It is mainly the `control` node in the `uart` package.

3.2. Module division

The system mainly includes the following core ROS2 packages:

1. `lslidar_ros`: The driver package for Leishen lidar, responsible for parsing radar data and publishing standard `sensor_msgs/msg/PointCloud2` messages through the `/point_cloud_raw` topic.
2. `uart`: Responsible for serial communication with the robot's underlying controller.
 - `truth` node: Reads raw data such as velocity and angular velocity from the serial port, and publishes it to the `/truth` topic.
 - `odom` node: Subscribes to the `/truth` topic, performs integration calculations, estimates and publishes the robot's odometry information to the `/odom` topic.
 - `control` node: Provides the `/control` service, receives speed commands and sends them to the underlying controller through the serial port.
3. `td3`: Implements the obstacle avoidance decision algorithm based on deep reinforcement learning.
 - `run_real_robot` node: As the main control node of the system, it subscribes to lidar and odometry data, loads the pre-trained TD3 model, calculates the optimal action (linear velocity and angular velocity) according to the current state, and calls the `/control` service to drive the robot to move.
4. `uart_interface`: Defines the custom ROS2 service interface `SetVelocity.srv` for robot speed control.

3.3. Technology selection

1. Programming Languages and Frameworks

Python: Used to implement the core logic of the `td3` and `uart` packages. Its rich libraries and concise syntax are conducive to rapid development.

C++: Used to develop the `lslidar_ros` driver package to meet the demand for high-performance radar data processing.

ROS2 Foxy: Provides distributed communication mechanisms such as nodes, topics and services.

PyTorch: Used to build and load the TD3 deep reinforcement learning model.

NumPy: Used for efficient numerical computation, especially when processing state and action arrays.

pyserial: Used to implement serial communication with hardware in the `uart` package.

2. Deep Reinforcement Learning Algorithm

TD3: Suitable for continuous action space control, combining technologies such as twin Q-networks and delayed policy update.

3. Neural Network Structure

Actor network: Adopts a Multi-Layer Perceptron (MLP) structure.

Input layer: 24 neurons, including 20-dimensional lidar data and 4-dimensional robot state (distance to target, angle deviation, current linear velocity, current angular velocity).

Hidden layers: Two layers with 800 and 600 neurons respectively, using ReLU as the activation function.

Output layer: 2 neurons, representing normalized linear velocity and angular velocity respectively, using the tanh activation function to limit the output value to the range $[-1, 1]$.

4. System testing and analysis

4.1. Test environment

4.1.1. Hardware test environment

The hardware platforms used for system testing mainly include: Embedded main control platform: FeiTengPai development board, serving as the core computing and control unit of the system;

Mobile chassis: WHEELTEC R550 UGV chassis with Ackermann structure, providing motion execution and odometry feedback;

Environment perception sensor: Leishen C16 multi-channel lidar, used for 3D environment point cloud collection and obstacle detection.

4.1.2. Software environment

The system software configuration is as follows:

Operating system and middleware: Ubuntu 22.04 + ROS2 Foxy distribution (one-click installation via fishros script);

ROS2 dependency packages: pcl-ros, pluginlib, pcl-conversions, libpcap-dev and Boost library;

Python runtime environment: PyTorch 2.0.1 (CPU version), pyserial 3.5, for serial communication and lightweight inference tasks.

4.2. Test procedures

The system function and performance tests are carried out according to the following process:

Video Output Connection

First, connect the FeiTengPai to an external display device through an HDMI cable for subsequent network and system configuration operations.

Set Up Mobile Hotspot

Turn on the mobile hotspot on the Windows host, set the SSID and access password, for wireless data transmission between the FeiTengPai and the host computer.

Peripheral Connection

Connect the USB keyboard and mouse from the computer lab to the USB 2.0 (black) ports of the FeiTengPai; reserve the USB 3.0 (blue) ports for subsequent data link connection.

System login

After the development board is powered on, wait about 2 minutes until the login interface is displayed (if it does not appear after more than 2 minutes, replace the display device).

Network configuration

Click the network icon at the bottom right corner of the desktop, select the enabled mobile hotspot to complete Wi-Fi connection, and ensure that the FeiTengPai obtains a valid IP address.

UGV positioning and initialization

Place the UGV next to the FeiTengPai, and use books to elevate the chassis so that the four wheels are off the ground, to prevent cables from being pulled and broken during unintended movement; at the same time, ensure there are no obstacles around the lidar to avoid collision damage.

Turn on the power of the vehicle (middle layer button), and wait about 1 minute for the system initialization to complete. Check whether "ros 0" status is displayed at the bottom left of the screen at the bottom of the vehicle; if "ros off" is displayed, toggle the specified DIP switch to the other side.

Chassis data link connection

Connect the idle USB data cable on the FeiTengPai expansion slot to the USB 3.0 (blue) port of the development board (prefer the upper port); at the same time, connect the idle Ethernet cable of the vehicle to the Ethernet port of the FeiTengPai. After the connection is completed, the identifier "netplan-0" (not netplan-1) should appear in the system network configuration.

System operation and data collection

Open two terminals A and B, and switch to the working directory ~/wk/bot;

In terminal B, execute source install/setup.bash to load the ROS2 environment, then run ros2 run td3 running, and wait for the model loading to complete;

In terminal A, execute bash start. After the data stream is stabilized, input the target point coordinates (e.g. 8.0 4.0) in terminal B;

During operation, you can wave an obstacle in front of the lidar or get close to the lidar to trigger the obstacle avoidance response, and input the reverse target point (e.g. 4.0 -8.0) at the appropriate time;

Copy the output logs of terminal A and terminal B respectively to the pre-created test.txt file on the desktop;

Continue to run ros2 run watch data_view in terminal B, collect the visualized output and append it to the log file;

Save the log file (Ctrl+S), and close the terminals after confirming the save status.

Data transmission back to host

Keep the hotspot connection, open a command line terminal on the desktop, execute the following SCP command to transfer the log file to the host computer, confirm the trust relationship and enter the password "user" as prompted, and wait for the transmission to complete.

4.3. Test results and analysis

This system designs a series of test cases for key functional modules such as lidar perception, chassis motion control, communication stability and safe obstacle avoidance. The specific test results are shown in Table 1.

Table 1. System function and performance test results

Test Scenario	Input Conditions	Actual Results	Performance Indicators
Lidar Point Cloud Collection	The lidar sends point cloud data via UDP, received by the main thread and parsed by the sub-thread	Continuous reflection points in the visualized point cloud are clearly distinguishable	Point cloud frequency ≥ 0.5 Hz, detection rate $> 95\%$
Lidar Data Latency	The whole process from point cloud data collection to processing completion	Timestamp comparison results meet expectations	End-to-end latency ≤ 1800 ms
Chassis Speed Control	Send linear velocity commands, encoder reads feedback velocity	Actual speed tracking error is within the allowable range	Steady-state error ± 0.008 m/s
Steering Response Time	Step command $0^\circ \rightarrow 10^\circ$	The system responds quickly and meets real-time control requirements	Response time ≤ 200 ms
Serial Communication Stability	Continuous operation for 4 hours	No interruption in communication link, continuous and stable data transmission and reception	Supports long-term continuous reading with high communication reliability
Emergency Stop and Obstacle Avoidance	An obstacle suddenly appears 0.5 m ahead	The UGV decelerates to a stop in time and can perform backward movement	Emergency stop response time ≤ 0.6 s
Long Straight Path Tracking	Target speed 1.0 m/s, operation for 10 s	Cumulative odometry error of the encoder is within the allowable range	Odometry error $< \pm 0.1$ m
Speed Step Response	$0 \rightarrow 0.08$ m/s acceleration command	Both rise time and steady-state error meet control requirements	Rise time ≤ 2000 ms, steady-state error ± 0.01 m/s
Steering Angle Tracking Error	Control angle change, read encoder feedback angle	The deviation between actual steering angle and target value is small	Error approx. ± 0.01 rad ($\approx \pm 0.5^\circ$), meeting closed-loop correction accuracy

5. Interface description

5.1. ROS2 software communication interface

1. Subscribed Topics:

Subscribed by the run_real_robot node:

/point_cloud_raw (sensor_msgs/msg/PointCloud2): Used to obtain environmental obstacle information.

/odom (std_msgs/msg/Float32MultiArray): Used to obtain the current position and posture of the robot.

/truth (std_msgs/msg/Float32MultiArray): Used to obtain real-time underlying data of the robot.

Subscribed by the odom node:

/truth (std_msgs/msg/Float32MultiArray): Used to calculate odometry.

2. Published Topics:

Published by the ls lidar_driver node:

/point_cloud_raw (sensor_msgs/msg/PointCloud2): Published raw point cloud data.

Published by the truth node:

/truth (std_msgs/msg/Float32MultiArray): Published raw data of the underlying controller.

Published by the odom node:

/odom (std_msgs/msg/Float32MultiArray): Published odometry data.

3. Provided Services:

Provided by the control node:

/control (uart_interface/srv/SetVelocity): Receives v_x (linear velocity, m/s), v_y (lateral velocity, m/s), r_z (angular velocity, rad/s) as the request, and returns a Boolean value indicating success or failure.

5.2. Hardware interface

1. uart.control.send_data_process(v_x, v_y, r_z)

Function: Packs velocity and angular velocity commands in physical units (m/s, rad/s) into 11-byte data frames recognizable by the underlying controller.

Frame structure: Frame header (0x7B) + reserved bit + speed data (high and low bytes) + check bit + frame tail (0x7D).

2. uart.truth.recv_data_process(frame)

Function: Parses the 24-byte data frame received from the serial port, and converts it into physical quantities including velocity, acceleration, angular velocity and voltage.

Data conversion rules:

Velocity (m/s) = $0.001 \times \text{raw value}$

Acceleration (m/s^2) = $(9.8 / 16384) \times \text{raw value}$

Angular velocity (rad/s) = $(500 / 32768) \times (\pi / 180) \times \text{raw value}$

3. ls lidar_driver.Ls lidarDriver

Function: Communicates with the Leishen C16 lidar through a UDP socket, receives and parses data packets of its proprietary protocol, and finally publishes them as ROS2 point cloud messages.

5.3. Electrical interface design

1. Lidar (Leishen C16)

Power supply: DC 12V.

Communication: Ethernet interface, communicates with the host via UDP protocol. Default device IP is 192.168.1.200, data port is 2368.

2. Robot Chassis

Communication: UART serial port, baud rate set to 115200, 8 data bits, no parity bit, 1 stop bit (8N1).

6. Conclusion

This paper systematically expounds the overall design scheme and engineering implementation of a low-speed unmanned ground vehicle (UGV). The platform is based on the FT-Pi embedded system and ROS2 framework, and adopts the TD3 deep reinforcement learning algorithm to realize autonomous navigation and obstacle avoidance functions. The system adopts a hierarchical architecture design, which ensures module decoupling while taking into account debugging convenience. At the perception level, the lidar feature extraction method based on sector partitioning effectively retains key environmental information on the premise of reducing input dimensions. The designed reward function can guide the agent to approach the target on the premise of safety and smoothness. Real-vehicle tests show that the system has reliable navigation performance in complex indoor environments, with a task success rate of 85% and low processing latency. Future work will further introduce a dynamic obstacle prediction mechanism, improve the model generalization ability by enriching training scenarios, and expand the system application scope to outdoor environments. At the same time, it is planned to deploy a lightweight model version by means of quantization compression and hardware acceleration to further reduce inference latency.

References

- [1] Lillicrap, T. P., Hunt, J. J., Pritzel, A., et al. (2015). Continuous control with deep reinforcement learning. arXiv:1509.02971.
- [2] Fujimoto, S., van Hoof, H., & Meger, D. (2018). Addressing function approximation error in actor-critic methods. *Proceedings of the 35th International Conference on Machine Learning*, 80, 1587-1596.
- [3] Tai, L., Paolo, G., & Liu, M. (2017). Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation. *Proceedings of the 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 31-36.
- [4] Marchesini, E., Corsi, D., & Farinelli, A. (2020). Benchmarking safe deep reinforcement learning in mobile robot navigation. *IEEE Robotics and Automation Letters*, 5(4), 5899-5906.
- [5] Quigley, M., Conley, K., Gerkey, B., et al. (2009). ROS: An open-source robot operating system. *ICRA Workshop on Open Source Software*, 3(3.2), 5.