A Mobile Robot Experiment System with Lightweight Simulator Generator for Deep Reinforcement Learning Algorithm

Yunfei Xiang[1*], Jiantao Qiu[1,2*], Jincheng Yu[1], Jiahao Tang[1], GuangJun Ge[1], Yu Wang[1] and Huazhong Yang[1]

***Abstract*— More and more researchers are trying to use deep reinforcement learning (DRL) for mobile robot tasks due to its powerful inference capability. However, deep reinforcement learning requires a large amount of data for DRL training in the pre-experimental stage, which hinders the application of the algorithm. On the other hand, the inconsistency between the ROS data interface and DRL GYM-Like data interface leads to a high cost of migration of the algorithm verification. This paper proposes a fast simulator generation method using linear approximate kinematics model and bake-based lidar rendering methods to generate a fast approximate simulator used in the pre-experiment stage to solve the problem of data cost. At the same time, an experimental system design scheme that converts the ROS interface into a GYM-like interface is also proposed to simplify the deployment process of deep reinforcement learning. We evaluate our proposed method on collision avoidance tasks in a variety of kinematics models and lidar scenarios. Our Method achieves about 14.2 times kinematics simulation speedup and 2.56 times lidar rendering speedup. We open-sourced our simulation environment and robot system software at `https://github.com/efc-robot/MultiVehicleEnv` and `https://github.com/efc-robot/NICS_MultiRobot_Platform`**

I. Introduction

With the development of Deep Reinforcement Learning (DRL), the inference ability of artificial intelligence (AI) has been improved. For example, DRL helps AI outperform humans in many tasks, such as Goes [1], Mahjong [2], and StarCraft [3]. The success of DRL in these game tasks shows the application prospect of DRL in mobile robot tasks. Actually, the previous work has introduced DRL to mobile robots navigation in dynamic environments [4], [5]. As a data-driven method, DRL needs to collect massive data for policy training. For example, the training of a simple multi-agent navigation policy consumes 2×10^6 attempts [4]. It is impossible to conduct so many experiments in the real world, so DRL depends on simulators. In order to implement the DRL algorithm on a real robot, the first step is to model the kinematics and sensor of the robot in the simulator, then train and verify the DRL algorithm in the simulator, and finally deploy the algorithm on the real robot as shown in Fig. 1.

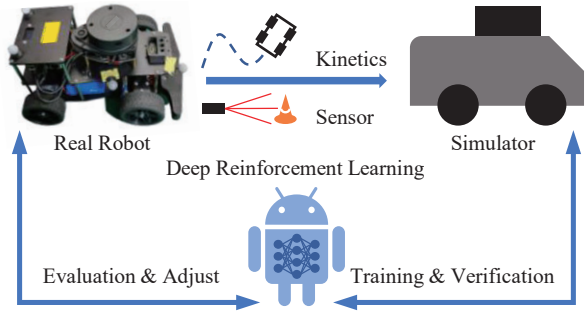


Fig. 1: Development process of applying deep reinforcement learning on mobile robot. Building a simulator with the kinematics model and sensors of the real robot as data source is usually the first step.

In order to make the DRL policy adaptive to the real environment, some simulators finely model the kinematics and sensors, such as the widely used Gazebo [6] and AirSim [7]. However, these fine-grained simulators are not fast enough for DRL training, which costs weeks for millions of attempts, reducing the experimental efficiency of researchers. For efficient data collection, DRL researchers often design fast simulators at a high level of abstraction. The data collection costs can be reduced from weeks to tens of hours. [4], [5].

However, the simulators used by current DRL developers lack physical constraints. For example, MADDPG [5] models each agent as a particle, which can change its moving direction and velocity arbitrarily, resulting in the failure of the trained policy on the real robot.

Developing simulators according to a specific task and robot with detailed physical constraints needs large effort, which becomes the bottleneck of DRL research. There are two main challenges to build the fast simulator:

From the perspective of **real-to-sim**, it is challenging to model different types of sensors and kinematics constraints, which results in the inability of the simulator for a specific task to quickly migrate to other tasks. For example, the Ackerman simulator uses two control signals (rear wheel speed and front wheel angle) for movement. The simulator needs to be re-built to control the differential vehicle, whose two control signals are the left and right wheel speed.

From the perspective of **sim-to-real**, the widely used interfaces differ between DRL field and robotics field. The DRL

*Equal contribution
[1]Department of Electronic Engineering, Tsinghua University, Beijing, China. yu-wang@tsinghua.edu.cn
[2]Currently working at Shanghai AI Laboratory, Shanghai, China. This work was completed in Tsinghua University.
This research was supported by National Key Research and Development Program of China (No. 2019YFF0301500), National Natural Science Foundation of China (No. U19B2019, M-0248), Tsinghua-Meituan Joint Institute for Digital Life, Tsinghua EE Independent Research Project, Beijing National Research Center for Information Science and Technology (BNRist), and Beijing Innovation Center for Future Chips.

researchers use GYM-like [8] simulators, which provide synchronization sensor and control interfaces for the DRL policy. However, real robots provide asynchronous interfaces such as ROS [9]. Thus, additional costs are required to deploy the DRL policy (if it works) on real world robots.

To address the above challenges about real-to-sim modeling and sim-to-real deployment, we propose a method to generate fast simulators from a high-precision simulator or actual environment and design a framework to connect the GYM-like interface of DRL to the ROS interface on real robots. The contributions are as follow:

- We propose fast kinematics modeling method and fast Lidar sensor rendering method, providing a large amount of data at low costs for DRL training.
- We provide unified sim-to-real interfaces and their paradigm, supporting the collaborative development of DRL policy and real robot systems.
- We build and open-source the generator of the fast simulator from high-precision simulator with unified interfaces.

The remaining of this paper is organized as follows: Section II introduces typical research methods of DRL and robotics. Section III gives a brief review of related work. The real-to-sim simulator generation method is detailed in Section IV and evaluated in Section V. The sim-to-real interface and system are described in Section VI. Section VII concludes this paper.

II. Background

A. Reinforcement Learning

Reinforcement learning models the system as a Markov decision process (MDP): $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{O}, r \rangle$. The state space $\mathcal{S}$ is the set containing all possible states of the system. The action space $\mathcal{A}$ is the set containing all possible actions that one agent can take. When the system is in the state $s_t \in \mathcal{S}$ and takes one action $a_t \in \mathcal{A}$, the next state s_{t+1} obeys conditional distribution $\mathcal{P}(s_{t+1}|s_t, a_t)$ which is the transition probability function. The state transition probability function describes the transition characteristics of the system state when the control signal is given, which is similar to the kinematics function in the field of robotics.

In each time step, the agent takes action a_t which obeys $a_t \sim \pi(a_t|o_t)$ and then obtains the next observation o_{t+1} and the reward $r_t = r(s_t, a_t)$, where o_t is the observation $o_t = \mathcal{O}(s_t)$ and π is the policy function.

Reinforcement learning obtains transition data (o_t, a_t, r_t, o_{t+1}) from interaction with the environment and uses methods such as Q-learning [10], policy gradient [11], [12], [13] to improve the policy. Recent advances integrate DRL and deep learning and thus make full use of the development of computing power. However, the DRL algorithm has the problem of low data efficiency [14], and there are a large number of hyperparameters in the algorithm [15], which need to be adjusted through many experiments. The deployment of DRL algorithms requires the training process to be repeated many times, which brings huge data calculation overhead and hinders the application of DRL in real world tasks.

B. Mobile Robot

There are many types of mobile robots, including wheeled robots (Ackerman wheel, differential wheel, Mecanum wheel, etc.), underwater robots, drones, etc. A mobile robot generally includes a motion controller, some sensors, a central control and calculation module, a power supply module, and a wireless communication module.

The main body of the movable robot moving on the 2D plane is regarded as a rigid body, and the position of the robot is usually represented by the 3-degree-of-freedom(3-DOF) transformation matrix of the rigid body relative to the earth coordinate system $T(t) = [R(t), p(t); 0, 1]$ [16]. The coordinates of the components in the robot relative to the main body determine the internal state of the robot $S(t) \in \mathbb{R}^I$ and I-DOF are introduced. The global state of the robot is determined by the generalized coordinates and generalized velocities corresponding to these 3+I-DOF.

By applying a control signal $C(t) \in \mathbb{R}^J$ on the motion controller, the internal components of the robot change the internal state and interact with the environment, which will change the global state of the robot. The kinematics function of the robot can be expressed as $[\dot{T}(t), \dot{S}(t)] = f(S(t), C(t))$. The simulator computes the increment of each dimension of the state according to the kinematics function and transforms the increment into the earth coordinate system for accumulation to obtain the global state of the mobile robot.

In the real world, mobile robots need to perceive their own state and the environment through various sensors, and make decisions based on the input data. Different kinds of sensors have their own data format and sampling frequency. These sensors collect data asynchronously and publish them to other components for subsequent processing.

III. Related Work

As described in Section II, DRL is used to solve the step-by-step MDP problem. GYM[8], developed by OpenAI Inc, is a widely used DRL platform that provides a simulator, different tasks, and synchronous data interfaces for MDP. For multi-agent DRL, OpenAI open-sources Multi-agent Particle Environments (MPE) [5] provides a simulator with GYM-like synchronous interfaces and a set of tasks such as multi-robot collision avoidance and pursue & escape.

The current work using DRL on robotics also adopts GYM-like interfaces. Chen et al [17] use DRL to solve the challenging multi-robot collision avoidance task with static obstacles. Semnani et al [4] further expand DRL to multi-robot collision avoidance in dense and dynamic environments. Both Chen and Semnani model the robot as a particle and give the moving direction and velocity as the outputs of policy to control the robot, which conflicts with real robot physical constraints. For example, the robot can move laterally in the simulator, but the real world Ackerman robot cannot laterally move.

In recent work, researchers attempt to provide DRL simulators with real world physical constraints. Amazon Inc develops DeepRacer [18], which provides a simulator for the robot to compete with a predefined Ackerman vehicle. DeepRacer also provides a real car and corresponding interfaces to facilitate the migration of the algorithm trained in the simulation to the actual system. However, DeepRacer is only suitable for racing, not for other tasks. Robotarium [19] designs a multi-agent simulator (for developing & validation) together with a remote accessible test field (for deployment). However, robots in Robotarium are not equipped with sensors, so Robotarium only supports simple tasks. Similarly, OffWorld-Gym [20] also offers a remotely accessible test field where the robots are equipped with a camera. OffWorld-Gym also models the environment and exposes a simulation environment based on Gazebo. Due to the insufficient efficiency of Gazebo, DRL data requirements of DRL cannot be met. Robopheus [21] uses data-driven methods instead of manual models for calibration, and thus it can quickly build scenes in Gazebo simulator for a given actual environment. Although Robopheus uses a fast calibration method and ROS interface to support multiple tasks in simulation and real world, the slow simulation speed and asynchronization interface makes it unsuitable for DRL.

To sum up, the present simulation environments including actual physical constraints are based on the slow Gazebo emulator [18], [21], or provide fast simulators for a few particular tasks [19], [18]. The above environments cannot meet the data requirements of DRL, nor can they validate DRL algorithms in different tasks.

We provide a simulator generation framework, which can customize a fast simulator for different tasks. The generated simulator not only has fast simulation speed, but also has conversion from the DRL GYM-like interface to the real robot ROS interface.

IV. Low-cost Simulator Approximation Method

A. Linear kinematics Approximate Method by Taylor Expansion

In order to reduce a large amount of computation overhead caused by complex kinematics simulation, it is necessary to build the robot kinematics model and simplify the calculation as much as possible. Since the kinematics models of robots with different structures are different, we use linear functions to build the kinematics model, and only select the variables with the largest correlation coefficients in the simulator. This method reduces the amount of calculation required for kinematics simulation and does not require detailed knowledge of the internal structure of the robot.

We construct the linear approximate kinematics function of the mobile robot based on $[\dot{T}(t), \dot{S}(t)] = f(S(t), C(t))$. We merge S and C into X, and T and S into $Q \in \mathbb{R}$, and study the relationship between X and $\dot{Q}$ in the i-th DOF $\dot{q}_i$. For $\dot{q}_i = f_i(X)$, we take the second-order Taylor expansion at $X = 0$ to get

$$\dot{q}_i(X) = f_i(0) + \nabla f_i(0)^T X + \frac{1}{2} X^T H_i(0) X + \cdots \quad (1)$$

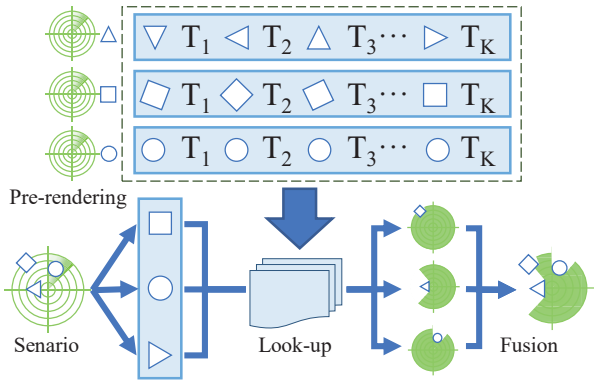


Fig. 2: Bake-based Lidar rendering method. Generate pre-rendering table and fusion the lookup result when rendering.

The above formula can be regarded as an approximate linear model of $\dot{q}_i$ about the constant term, the first order term, the quadratic term, and the cross term of X. The kinematics model using linear approximation can be expressed as $\dot{Q} = A\mathcal{X}$, where $\mathcal{X} = vec([1, X] \otimes [1, X^T])$.

We collect data from $D_t = (t, Q_t, X_t, \Delta Q_t)$ in the high-precision simulator, and input it into the above linear model to get the model parameters by fitting. However, in the above model, there are some pairs of independent variables and dependent variables that are not statistically correlated. The direct use of all parameters in the model coefficient matrix will cause the introduction of systematic deviations, and additional calculations will also be added. Therefore, it is necessary to prune the independent variables that have no remarkable correlation with the dependent variable when the linear fit is obtained.

We divide each independent variable by its own standard deviation, so that the coefficients corresponding to each independent variable can be compared with each other. After that, we sort all the absolute value coefficients in descending order to get $|\beta_1| \geq |\beta_2|, \cdots \geq |\beta_N|$, select the biggest-k values so that

$$\sum_{1}^{k} |\beta_i| < \alpha \sum_{1}^{N} |\beta_i| < \sum_{1}^{k+1} |\beta_i| \quad (2)$$

Among them, $0 \leq \alpha \leq 1$ represents the filtering threshold. The larger the α, the more terms will be retained. By adjusting the size of α, we can trade-off between accuracy and calculation speed.

B. Bake-based Lidar Rendering Method

In order to avoid a large amount of computational overhead caused by real-time rendering of lidar, we borrowed the "baking" method in the 3D game engine. This method includes: 1. Modeling the perception equation of lidar. 2. Pre-rendering and store the lidar perception results of a single object in some possible situations. 3. Reading the stored results and merging them into the final result during the simulation. The bake-based Lidar rendering method is shown in the Fig. 2.

1) Lidar Perception Model: In our system, obstacles are modeled as rigid cylinders, and all surfaces have good diffuse reflection characteristics for the laser. The attributes of an obstacle include: transformation related to the earth T_O , surface information P_O. A lidar is modeled as emitting lasers in multiple directions simultaneously from the center point, and measuring the distance from the edge of the obstacle that each laser hits to the center point of the lidar. The attributes of lidar include: transformation related to the earth T_L, laser ranging range $[R_{min}, R_{max}]$, laser angle range $[\alpha_{min}, \alpha_{max}]$, and lidar resolution N_{lidar}, we collectively refer to the attributes other than T_L as P_L.

Considering a scene with one lidar L and M different obstacles $[O_1, \cdots O_m, \cdots O_M]$, the ranging result is an N_{lidar}-dim vector R_{lidar}.The lidar perception model $R_L = \mathcal{L}(L, O_1, \cdots O_M)$ can be obtained by the following formula:

$$\mathcal{L}(L, O_m) = \mathcal{L}(T_L, T_{O_m}, P_L, P_{O_m}) = \mathcal{L}(T_{LO_m}, P_L, P_{O_m}) \tag{3}$$

$$\mathcal{L}(L, O_1, \cdots O_M) = \min(\mathcal{L}(L, O_1), \cdots \mathcal{L}(L, O_M)) \tag{4}$$

Where T_L and T_{O_m} are the transformation matrix of the lidar L and the obstacle O_m related to the earth and T_{LO_m} is relative transformation matrix between L and O_m. Formula (3) indicates that when the parameters of the lidar and the obstacle remain unchanged, the perception result is only determined by the relative position of the two. Equation (4) indicates that the laser is always reflected after encountering the first obstacle, so the perception result of multiple obstacles is consistent with the minimum distance of each obstacle.

2) Pre-rendering Lookup Table: In order to build a lookup table for pre-rendering, we fix the lidar at the origin of the coordinate system, list all obstacles $[O_1, \cdots O_M]$ with different boundaries, formulate a coordinate list $\mathcal{T} = [T_1 \cdots T_K]$ for each obstacle, and render $\mathcal{L}_{k,m} = \mathcal{L}(T_k, P_L, P_{O_m})$ one by one by using the high-precision lidar rendering function. When rendering, the position of the obstacle is limited to the main axis of the lidar, only the relative distance between the obstacle and the lidar and its own rotation angle are scanned, and the rendering results of different azimuth angles are obtained directly through rotation. The angle error of the rendering result is smaller than $2\pi/(2N_{lidar})$ according to the rotational symmetry with the relative azimuth angle of lidar.

3) Rendering by Lookup and Fusion: After constructing the rendering result table $\tilde{\mathcal{L}}(k, m) = \mathcal{L}_{k,m}$ and the corresponding index $\mathcal{T}$, each time the observation function is called in the DRL algorithm training, the measurement result of the lidar can be obtained by the way of "Lookup and Fusion". For each lidar L, for the obstacle O_m within its detection range, we calculate the relative position T_{LO_m}, get the closest option in the $\mathcal{T}$ according to $\arg\min_k ||T_k^{-1} T_{LO_m}||$, and look up the table to get $\mathcal{L}_{k,m}$ as the rendering result of a single obstacle. After obtaining the rendering result $\mathcal{L}_{k,1} \cdots \mathcal{L}_{k,M}$ of all obstacles in the detection range, the approximate result of the current lidar can be obtained by taking the minimum value of each dimension measurement result.

TABLE I: RMSE and FPS of the linear kinematics model for three different kinematics model

Kinematics	RMSE			FPS(s^{-1})
	$\Delta x(m)$	$\Delta y(m)$	$\Delta\theta(rad)$	
Ackerman	3.08×10^{-9}	9.71×10^{-6}	9.89×10^{-5}	9.01×10^{4}
Differential	2.56×10^{-7}	5.11×10^{-9}	1.56×10^{-16}	5.72×10^{4}
Mecanum	1.23×10^{-17}	9.41×10^{-18}	3.75×10^{-17}	4.73×10^{4}

V. Experiments and Results

A. Error Analysis of Linear kinematics Approximate Method

In order to test the kinematics error of our proposed method, we extract kinematics for three typical wheeled robots (Ackerman, Differential, and Mecanum) models. The three kinematics models and the meaning of the variables are shown in Fig. 3.

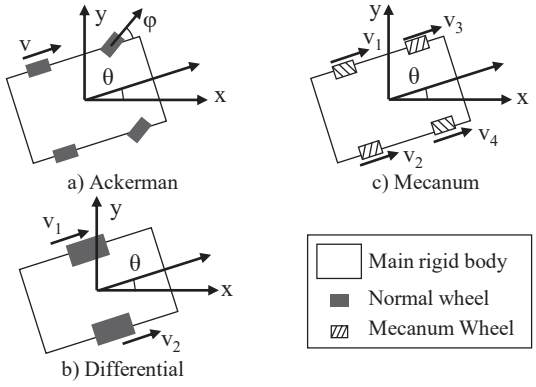


Fig. 3: Three kinds of robot with different kinematics

We use the kinematics models from the textbook [16] to do the simulation with simulation time step $\Delta t = 0.001s$ to generate data as data source ground truth. We use the proposed linear kinematics model to approximate the kinematics models. We report the RMSE between our linear model and the original model in Table I.

We also run the gazebo simulator with one Ackerman robot on an empty plane and set "$real\_time\_update\_rate$" as 0 to do the simulation as fast as possible. The running frame rate of gazebo is 6.358×10^3 FPS, and our method achieves a speedup of about 14.2 times.

B. Error Analysis of Bake-based Lidar Rendering

In this section, we will analyze the errors caused by the baking-based lidar rendering method. In order to measure our proposed baking-based lidar rendering method, we design four scenes containing different types of obstacles.

In the scenes, we fix the lidar at the center of the field and randomly place the positions of all obstacles to obtain the lidar perception results of different rendering methods. We use a rendering method based on computational geometry scanning as the baseline, and also use this method as a high-precision simulator for pre-rendering and obtaining a lookup table. We then use the bake-base method to obtain the lidar perception results in the same scene. The following table

TABLE II: Distance RMSE and Area RMSE of bake-based method for difference sampling scenes

Sence	Scan-mode	Bake-based		
	FPS(s^{-1})	FPS(s^{-1})	RMSE (m)	RMSE (m^2)
Circle	5.73×10^{1}	3.30×10^{2}	8.47×10^{-2}	5.63×10^{-3}
Square	3.64×10^{0}	3.32×10^{2}	8.88×10^{-2}	6.71×10^{-3}
Strip	4.16×10^{0}	3.24×10^{2}	1.05×10^{-1}	9.86×10^{-3}
Mixture	5.70×10^{0}	3.39×10^{2}	9.06×10^{-2}	7.20×10^{-3}
Average	5.65×10^{0}	3.31×10^{2}	9.23×10^{-2}	7.35×10^{-3}

TABLE III: Area RMSE of bake-based method for different sampling density cases

Table Size	100×240	50×240	100×120
Shape	$RMSE_A(m^2)$		
Circle	5.63×10^{-3}	1.76×10^{-2}	5.58×10^{-3}
Square	6.71×10^{-3}	2.34×10^{-2}	6.99×10^{-3}
Strip	9.86×10^{-3}	3.46×10^{-2}	1.15×10^{-2}
Mixture	7.20×10^{-3}	2.65×10^{-2}	8.65×10^{-3}
Average	7.35×10^{-3}	2.55×10^{-2}	8.18×10^{-3}

records the rendering time of the two rendering methods in different scenes, as well as the RMSE of the distance measurement and the area measurement.

It can be seen from Table II that our method can achieve a speed increase of 2 orders of magnitude compared with the method of computational geometry. The average RMSE of the distance is $9.23 \times 10^{-2}(m)$ in all scenes, which is 3.08% relative to the maximum measured distance $3(m)$, and the RMSE of the area is $7.35 \times 10^{-3}(m^2)$, which is 0.25% relative to the maximum detection area. Table II also shows that the shape of strip has a larger error than circle and square, due to its relatively large size and large aspect ratio. We also run gazebo with the same scenes to record the frequency of the lidar. We set the update frequency as 1000 so the gazebo will update lidar data as fast as possible. The average frequency of gazebo lidar is about 370Hz with 290% CPU Utilization. Our method runs with 100% CPU Utilization and have about 2.59 times speedup.

In order to further investigate the relationship between the density of the table and the error, we set three different rules for the table. One is 100 equal divisions in the distance and 240 equal divisions in the angle, the other two are reduced by half in distance and angle respectively. And we measure the area RMSE relative to the scan-mode method in four different scenes.

As can be seen from Table III, after reducing the distance sampling density, the error increases by 3.47 times on average. The decrease of the sampling density of the angle has a relatively small effect on the error. This means that in the case of the same table building time budget and storage budget, denser distance sampling and fewer angle sampling should be used.

In order to further verify the effect of the lidar of our algorithm in a more intuitive way, we draw the lidar point cloud from different data sources. The robot moves follow a fixed path in a scene with 4 obstacles and collect lidar data points from 3 different environments. The lidar point cloud is transformed into the earth coordinate system through the pose of the robot for drawing and superimposed to obtain the lidar point cloud. We draw the cloud point by the data from scan-mode, bake-base, and gazebo environment, and the results are plotted in Fig. 4. It can be seen from the plot that the point cloud of the three methods have very few differences. We also run SLAM algorithm [22] by the bake-based method data and plot the cloud point in the 4 d), the result show that our lidar data can support SLAM algorithm.

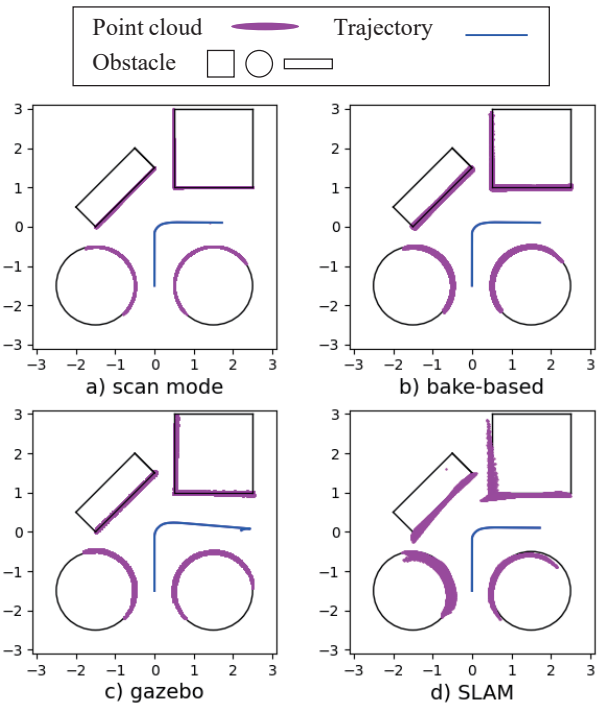


Fig. 4: Lidar point cloud result plot from a) scan-mode, b) bake-based, c)gazebo and d) SLAM point cloud.

VI. DRL-Robot Experiment System

To apply the DRL algorithm in the robot system, it is necessary to transfer the trained policy function to the robot system. The data obtained by the sensor is used as input, and the output of the policy function is used as the control quantity of the robot. To reduce the additional maintenance cost caused by frequent modification of the DRL task description in the algorithm migration process, we borrow the task description software specifications commonly used by MARL researchers and reuse the MARL task description in the actual robot system. At the same time, a unified data interface is provided for the strategies obtained by MARL training, which allows the DRL task description to be modified to the robot system at almost no cost.

A. System Design Guidelines

We analyzed the structure of the "DRL algorithm + simulator" system during the training process and divided it into the DRL front end and the Simulator. The DRL front end contains the policy function, which provides data observation call-back and external output actions for the policy. The simulator part contains kinematics simulation and sensor perception modules. The observation callback function of

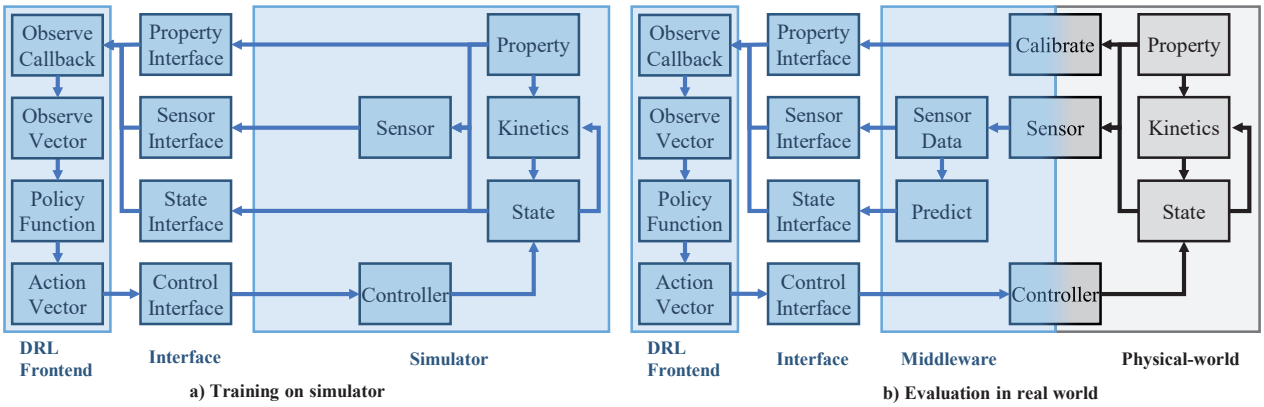
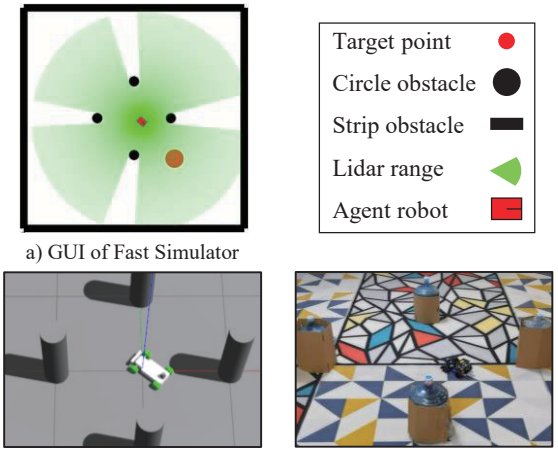


Fig. 5: Training and Evaluation of Robot System for DRL. The state and property of the robot can be directly accessed by the DRL algorithm in the simulator. In the real robot system, a middleware subscribes to the corresponding topics and provides data through a unified data interface. The DRL algorithm can access the latest data through the interface.

DRL can directly read the sensor data, property, or the state of the robot from the simulator to obtain observation results to generate an observation vector.

But in the "DRL algorithm + middleware + robot" system, the true value of the robot state and attributes cannot be directly obtained due to the kinematics process of the real robot system running in the real physical world. It is necessary to calibrate the system in advance to obtain the system properties. Sensors are used to collect sensor data during the mission and sensor data are used to predict the current system state. The DRL-Robot system framework is shown in Fig. 5.

B. Development and Deployment



a) GUI of Fast Simulator

b) GUI of gazebo

c) Real robot environment

Fig. 6: Navigation and obstacle avoidance task in a) Fast Simulator, b) gazebo and c) real world.

We build a "Fast Simulator" with linear kinematics model and bake-based lidar rending module, then run a task of navigating based on the relative coordinates of the target point and avoiding obstacles through lidar on it. We use the Fast Simulator as a data source to train the DRL algorithm and evaluate the policy function on Fast Simulator simulator, gazebo environment, and real robot environment tasks, as shown in Fig. 6.

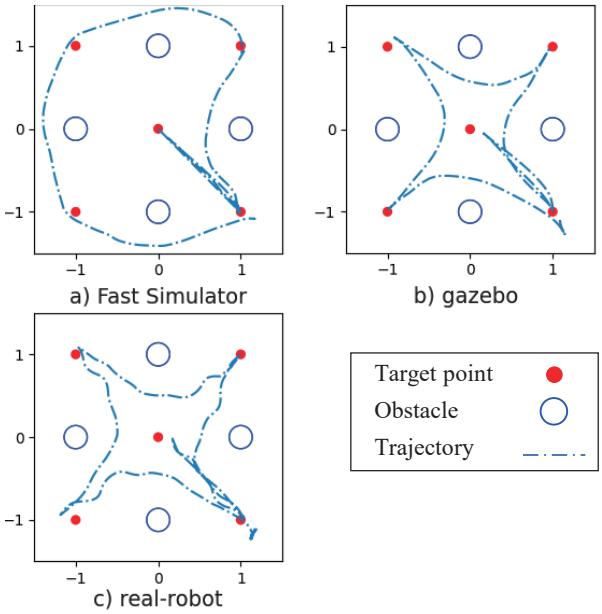


Fig. 7: Robot trajectory for navigation and obstacle avoidance task in different environments.

In our experiments, the algorithm trained on the Fast Simulator can avoid obstacles and reach all target points in all tasks. The trajectory diagram of the robot in different tasks is shown in Fig. 7. Moreover, the policy function uses a consistent interface when testing the DRL algorithm in three different environments.

VII. Conclusion and Further Work

In this paper, a fast simulator generation method is proposed to obtain a lightweight simulator for training, and a mobile robot experimental system with a consistent interface is used to help verify the DRL algorithm to solve the problem. We propose a method using a linear kinematics model and a baking-based lidar simulation method for simulating arbitrary dynamics of mobile robots and fast lidar simulations. In addition, an experimental system is designed that can transform the ROS interface for robots into the GYM-like DRL interface required by DRL. We have verified

through experiments that the generated simulator can be used to quickly train and verify the DRL algorithm, run the trained DRL algorithm on Fast-Simulator, Gazebo, and real robot tasks with no code modification.

We believe that the current deep reinforcement learning algorithms in the field of robotics still face problems such as high data costs, long experimental cycles, and development difficulties caused by tight coupling. Nowadays mobile robots can move in more complex 3-D scenes and have more complex sensors such as cameras. It is necessary to build a lightweight simulator generation method for these more complex environments and accelerate the application of DRL algorithms in the robotics field.

REFERENCES

[1] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[2] J. Li, S. Koyamada, Q. Ye, G. Liu, C. Wang, R. Yang, L. Zhao, T. Qin, T.-Y. Liu, and H.-W. Hon, "Suphx: Mastering mahjong with deep reinforcement learning," *arXiv preprint arXiv:2003.13590*, 2020.

[3] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, *et al.*, "Grandmaster level in starcraft ii using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, 2019.

[4] S. H. Semnani, H. Liu, M. Everett, A. de Ruiter, and J. P. How, "Multi-agent motion planning for dense and dynamic environments via deep reinforcement learning," *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 3221–3226, 2020.

[5] R. Lowe, Y. Wu, A. Tamar, J. Harb, P. Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," *arXiv preprint arXiv:1706.02275*, 2017.

[6] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, vol. 3. IEEE, 2004, pp. 2149–2154.

[7] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," in *Field and service robotics*. Springer, 2018, pp. 621–635.

[8] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[9] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, *et al.*, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[11] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *International conference on machine learning*. PMLR, 2018, pp. 1861–1870.

[12] C. Yu, A. Velu, E. Vinitsky, Y. Wang, A. Bayen, and Y. Wu, "The surprising effectiveness of mappo in cooperative, multi-agent games," *arXiv preprint arXiv:2103.01955*, 2021.

[13] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[14] M. Schwarzer, N. Rajkumar, M. Noukhovitch, A. Anand, L. Charlin, D. Hjelm, P. Bachman, and A. Courville, "Pretraining representations for data-efficient reinforcement learning," *arXiv preprint arXiv:2106.04799*, 2021.

[15] J. Wang, J. Xu, and X. Wang, "Combination of hyperband and bayesian optimization for hyperparameter optimization in deep learning," *arXiv preprint arXiv:1801.01596*, 2018.

[16] A. Mueller, "Modern robotics: Mechanics, planning, and control [bookshelf]," *IEEE Control Systems Magazine*, vol. 39, no. 6, pp. 100–102, 2019.

[17] Y. F. Chen, M. Liu, M. Everett, and J. P. How, "Decentralized non-communicating multiagent collision avoidance with deep reinforcement learning," in *2017 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2017, pp. 285–292.

[18] B. Balaji, S. Mallya, S. Genc, S. Gupta, L. Dirac, V. Khare, G. Roy, T. Sun, Y. Tao, B. Townsend, E. Calleja, S. Muralidhara, and D. Karuppasamy, "Deepracer: Educational autonomous racing platform for experimentation with sim2real reinforcement learning," 2019.

[19] D. Pickem, P. Glotfelter, L. Wang, M. Mote, A. Ames, E. Feron, and M. Egerstedt, "The robotarium: A remotely accessible swarm robotics research testbed," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 2017, pp. 1699–1706.

[20] A. Kumar, T. Buckley, J. B. Lanier, Q. Wang, A. Kavelaars, and I. Kuzovkin, "Offworld gym: open-access physical robotics environment for real-world reinforcement learning benchmark and research," 2020.

[21] X. Ding, H. Wang, H. Li, H. Jiang, and J. He, "Robopheus: A virtual-physical interactive mobile robotic testbed," 2021.

[22] J. Xu, "Open-source 2d-slam," https://github.com/jan-xu/2d-slam.