Lightweight Multi Car Dynamic Simulator for Reinforcement Learning

Abhijit Majumdar, Patrick Benavidez and Mo Jamshidi
Department of Electrical and Computer Engineering
The University of Texas at San Antonio
San Antonio, TX 78249
Email: abhijit.g.majumdar@gmail.com, patrick.benavidez@utsa.edu, moj@wacong.org

***Abstract*—With improvements in reinforcement learning algorithms, and the demand to implement these algorithms on real systems, the use of a simulator as an intermediate stage is essential to save time, material and financial resources. The lack of particular features in a unified simulator for applications to autonomous cars and robotics, encouraged this research, which produced a simulator capable of simulating multiple car like objects, in either one or several arenas (environments). Being a lightweight application, multiple instances of the simulator can run at the same time, only constrained by the available computational resources.**

***Index Terms*—multi agent, simulation, reinforcement learning, multiple instances, multi environment**

I. Introduction

The exponential increase in the computational capability can be accounted for the extraordinary developments in reinforcement learning algorithms in the recent years. This is due to the fact that with the availability of such computational ability, algorithms can be rapidly tested and evaluated to make improvements and correct nuances. Application of such systems to control systems is a major area, since such algorithms can often produce much more optimal solutions to a control problem than one built manually.

Modeling of systems has had a rich history, which enables the development of a simulator feasible and resourceful. For the purpose of testing an algorithm which uses exploration to sample data to learn from, a simulator is ideally the first implementation platform. This is due to several reasons, one of which is that computers can simulate a dynamical system much faster than real-world dynamics. Running iterations in this manner expedites the testing phase though it might suffer from the problem of approximation due to the *curse of dimensionality*.

Another reason to use a simulator in testing reinforcement learning algorithms is that such algorithms demand huge sampling, which means using a physical system to perform such tests would require a lot of material and financial resources, other than time. Using a simulation to perform initial learning and then moving the learned model onto a physical system to perform fine tuning helps reduce these problems.

*This work was supported by Grant number FA8750-15-2-0116 from Air Force Research Laboratory and OSD through a contract with North Carolina Agricultural and Technical State University.

Existing simulators oriented towards reinforcement learning and machine learning in general [1]–[6], provide a either the capability to simulate high definition graphics in the environment [1], [3], which consume a substantial amount of available computational resources, or are not as flexible in reconfigurability as we intend to use it with. For example the use of a configuration file to construct a multi-agent multi-arena simulator, with minimal resource use to enable the algorithm to use the available resources to compute faster.

In this research we construct a simulator to emulate a four wheeled car model with the concept of *Ackermann* steering whose dynamics are controlled by a *Bicycle* model of a car. Our focus on cars is achieved with the developments in the field of autonomous driving and self-driving vehicles. Similar research is also beneficial to the field of robotics, especially in the field of mobile robots. When controlling complex dynamics like the non-holonomic car model, with any new addition to the structural construct of the system, the control method needs to be modified to incorporate the changes. Using reinforcement learning algorithms one can achieve controllable results with different configurations, for example sensor placements, dimensions of car and steering limits, while also performing analysis on the best configuration.

This enables the use of our research in several derived robotics fields as well. Using senor networks to build a map of the environment and localizing, is a commonly known field Simultaneous Localization and Mapping (SLAM), which can be incorporated into the simulator if needed, which also encouraged us to use distance sensors in the simulator.

The paper is organized by first describing the development of different aspects of the simulator. This includes the dynamics of the environment and the cars, the graphical user interface, the reinforcement learning interface and the reconfiguration details. Following are experimental results showing different use of the simulator, followed by conclusion and future development.

II. Development

A. Modeling and Dynamics

In order to test reinforcement learning algorithm applications to control problems in a complex problem, an environment was constructed where the objective is for a car like object to reach a destination. The simulator was built

using *Python* to enable fast prototyping and testing. This also enables modularity and reconfigurability, such that each component of the simulator is independent in operation from one another, and hence can be imported as per the user requirement. The use of mostly built-in libraries for python and few openly available and widely used libraries was targeted to ensure compatibility and common issue resolution. The required libraries for performing computation are *math* and *random*, both of which are built-into *Python* and *numpy*.

Several features have been added to this simulator to incorporate the use with reinforcement learning application and testing as described below.

1) Arena: The arena is defined as the area where the car(agent) is able to perform actions and explore the environment. This arena can be defined in the configuration file under the tab *Arena*, where the simulator expects the user to input a series of coordinates defining a polygon. Figure 1 shows exampled of types of arena that can be defined using intelligent use of these coordinates. Some examples of arenas are already defined in the example configuration file and can be used to derive a custom environment. A class is defined *Environment*, which uses the extracted configuration details to construct the environment.

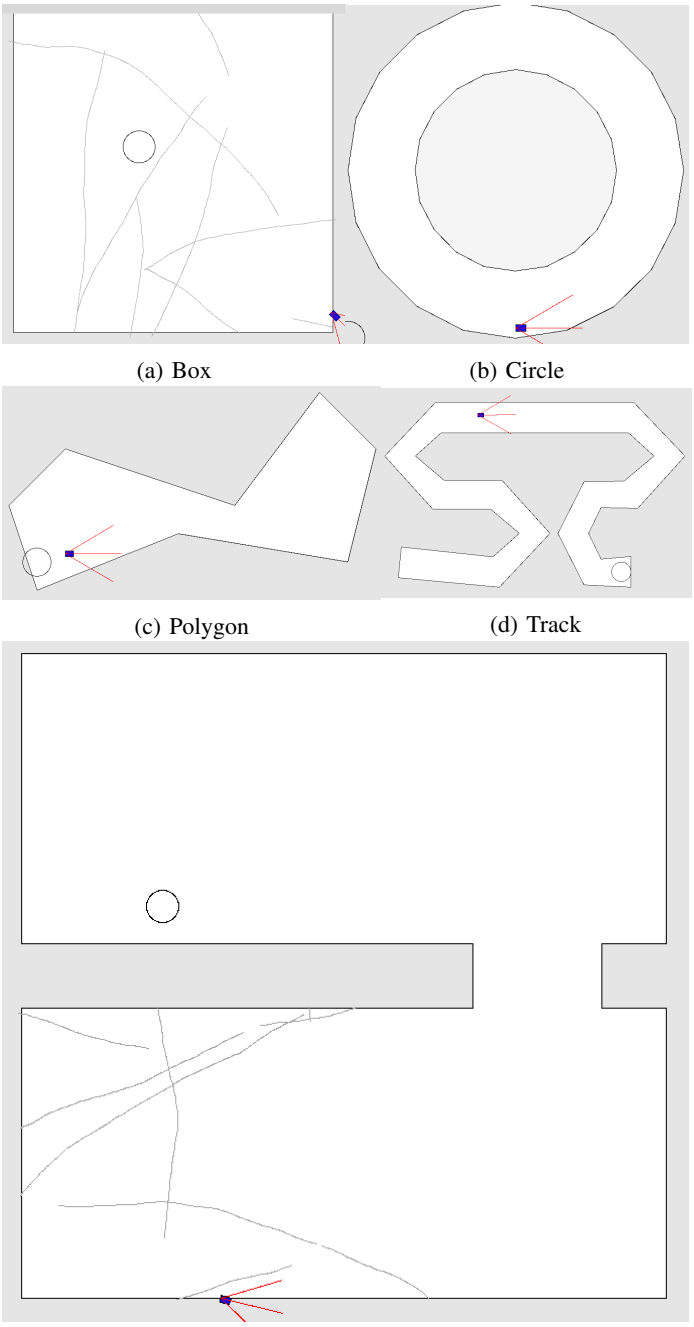
Under the same tab, we can define obstacles in the area space, of which the simulator expects the user to provide coordinates creating a polygon. Obstacles coordinates may be defined under a sub-tab group *Obstacle* under the *Arena* tab.

The simulator uses vector math and coordinate geometry to compute the dynamic interactions of the cars(agents) with the environment and hence does not depend on any external physics engine. It is capable of detecting collision and interaction whenever the *compute_interaction* method is called by the user, which provides interactions of the agents(provided as an argument) with the environment. The edges of the arena as well as the obstacles can serve as an intractable object. This method also computes the sensor interactions providing the distance of the closest obstacle to the agent. Since the destination for each agent is known by this class, it can also compute the agent physical state, as weather it has collided, reached its destination or has run out of time. The agent is assigned the state of collided which is considered a terminal state when it hits either the boundaries of the environment or any of the obstacles.

The environment also defines methods to randomize and change the destinations for each agent. This method makes sure that the destination is not inside an obstacle or outside the defined arena, thus having a plausible solution.

The environment is also configurable to run multiple arenas with assignment of different agents to each arena. This is useful to simulate Multi Agent Reinforcement Learning (MARL), with the flexibility to test the resulting learned policy in a different environment.

2) Car Object: The agents used in the environment are car like objects which are governed using a bicycle model of a rear wheel drive car. This was selected in contrast to using a differential drive model to encourage the application of such an algorithm to be used in real-world self-driving applications. The car dimensions of width and length are configurable. The



(a) Box (b) Circle

(c) Polygon (d) Track

(e) Corridor

Fig. 1: Examples of different configurable arenas

car state at any time s_t is defined by the position and the orientation of the car, and its dynamics are governed by the following equations,

$$s_t = [x_t, y_t, \omega_t] \tag{1}$$

$$\dot{x_t} = v_t \cos(\omega_t) \tag{2}$$

$$\dot{y_t} = v_t \sin(\omega_t) \tag{3}$$

$$\dot{\omega_t} = \frac{v_t}{L} \tan(\psi_t) \tag{4}$$

Where,

$x_t, y_t \rightarrow$ car position(center of rear axle) in environment frame

$\omega_t \rightarrow$ orientation of the car in the environment frame

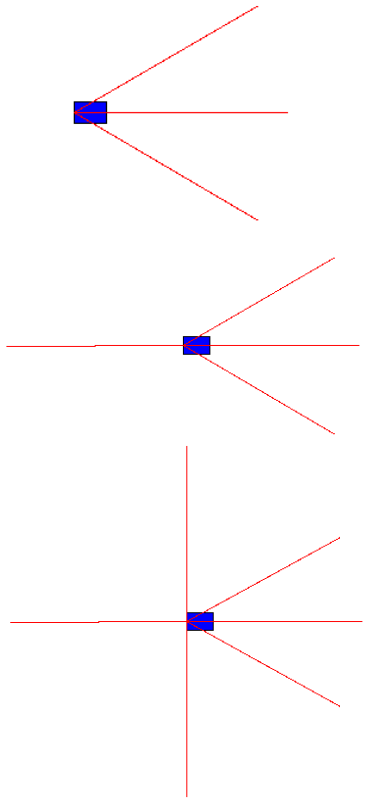


Fig. 2: Sensor placement configuration examples

$v_t \rightarrow$ the velocity of the car at time t
$\psi_t \rightarrow$ the steering angle of the car at time t
$L \rightarrow$ the length between the front and rear axle of the car
It should be noted that the position in the state of the car is defined at the center of the rear axle of the car, and all interactions with the car are in correspondence to this point. This means the collision model for the car is a point model.

Each car is equipped with sensing devices, which behave like laser range finders measuring distance to the closest obstacle in the path. The angle and range of the sensors are configurable, and any number of sensors can be attached to a car. Examples of different configurations possible are shown in Figure 2. Control of the car is performed by manipulating the velocity and steering angle of the car with the provided methods. These methods limit the maximum velocities and the maximum steering angles that the car can achieve using pre-configured values.

Each car property including its dimensions, limits on velocity and steering angles, sensor range and sensor orientations are configurable by defining them under the tab *Cars*. Each car can be defined by defining a sub-tab group with the index of the car. The configuration file uses the initial state of the car, the dimensions and the limits. Sensors can be defined using the third level of sub-group tabing *Sensors*, where can be used to define as many sensors as needed with the range and angle of the respective sensor. A class defining a car object provides access to all its methods, an object of which can be initialized by providing details of the car which is read from the configuration file. An example of a car object in the environment is shown in Figure 3.

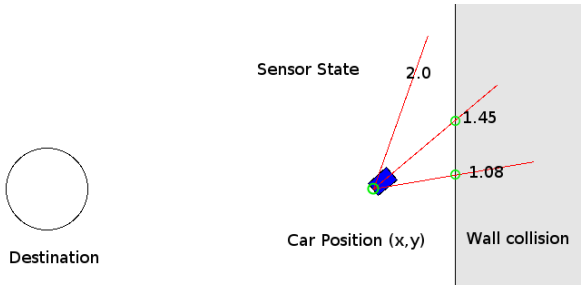


Fig. 3: Car interactions with environment using sensors

The class also defines an update method which is used to compute one iteration of the using the velocity and steering angle to update the state of the car. This takes in a time δt to compute the dynamic update which is also defined in the configuration file.

A method is also defined to encourage using Partially Observable states to solve Partially Observable Markov Decision Processes (POMDPs). This computes the encoded sine and cosine of the difference in the angle between the orientation of the vector pointing towards the destination, and the orientation ω of the car. These encoded angles indicate the direction of the destination from the car. It also computes the euclidean distance of the destination from the car. These states are then concatenated with the inverted sensor readings to indicate an observed obstacle. Hence a set of such states is returned to compute the best method to reach the destination.

B. Graphical User Interface(GUI)

The GUI is constructed using a *Python* built-in library *Tkinter*, which avoid the need to install additional libraries. It is constructed using a modular structure such that it can be disabled and reinitialized as desired. Being independent form other modules in the simulator, it draws the environment and agents based on the information provided during its update. A class is defined *GUI*, which uses configurations from the configuration parser to draw the arena, obstacles, agents and sensors. The window resolution used for the GUI is configurable and hence can adapt to different size of displays.

The GUI initializes a window which contain three main components: the canvas, the debugging area and the graph view. The canvas is used to draw the arena with obstacles along with the cars,and their destinations. The debugging area is equipped with two buttons to switch between learning and testing an environment, which is useful and found to be often necessary while testing reinforcement learning algorithms. It also displays any debugging information that needs to be updated at regular intervals. Finally the graph section consists of an area to view a graph and buttons to switch between different plots.

As described the GUI window provides a graph section for the viewing incremental performance indexes. The number of graphs and the plotted contents are configurable by the user, such that they can provide any data to be plotted on the graphs during an update of the GUI. It should be noted that only one graph can be plotted at a time.

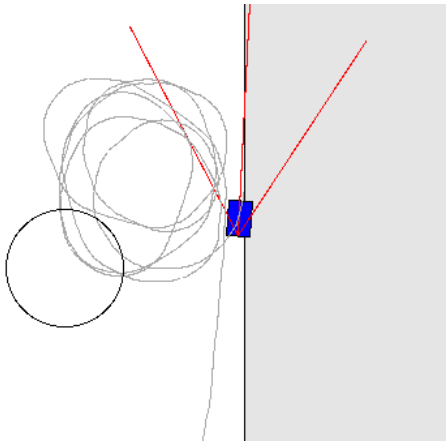


Fig. 4: Incorrect reward function causing agent to spin in circles, detected by the trace lines, even though updates are performed in the background

The GUI also provides a feature to trace the steps taken by the agent in the environment. This is developed with the use case in mind, where in reinforcement learning algorithm testing, the GUI is updated only every n^{th} episode. During this time, the user may not be aware of the trajectories taken by the agent, which might be performing poorly or learning an undesirable behavior. Figure 4 shows an example case of such a failure detection. This feature can be enabled by the user as a configurable parameter.

The refresh rate of the GUI update is limited to the human perception to conceivable information, which implies having a standard delay before the next frame. For this reason the GUI refresh function being called at each iteration slows down the learning process of the reinforcement learning algorithm. Yet, one might need to draw traces, while the GUI is not being refreshed. For this purpose, different methods are provided to update the cars, the traces, the graphs, debug information and refresh the GUI. As a result the user may decide to use these methods at their own discretion, to expedite the learning process while maintaining a intuitive perception.

This update rate however is configurable as a scaling factor δk to the dynamic update time δt, to indicate how realistically time should propagate in the simulation. An example of the environment developed is shown in Figure 5.

C. Reinforcement Learning Interface

The class defined here are replaceable and may be used as is or extended to custom classes as per user requirements. Since we provide Deep Q-Network (DQN) [7] as an example reinforcement learning algorithm implementation, this module defines a *ReplayMemory* class which is initialized with the length of the buffer, the length of the state, the terminal states and minimum buffer length. As a result different configuration of a car state can be used to train the neural network, for example, the number of sensors may be varied. Also training of the neural network may be started only when a part of the buffer is filled before which the replay memory samples *None* values.

Experience samples can be added using the *add* method which expects a tuple of state, action performed in the state, reward obtained , the new state, and if the state was terminal. If the buffer is full, it overwrites the previous experiences, replacing the oldest values first.

An example DQN implementation is provided using the class *DQN*. Values of all associated parameters like ϵ for $\epsilon - greedy$ policy, discount factor γ, learning rate α, and other parameters are configurable under the tab *Reinforcement Parameters*. The Deep Neural Network (DNN) used in the DQN algorithm is also configurable using the configuration file, with specifications of number of neurons in each hidden layer and the activation function for each under the tab *Neural Network*. The action space is configurable using the tab *Actions* while the terminal states are configurable along with their respective rewards under the *Reinforcement Parameters* tab in the configuration file. We use *Keras* with *Tensorflow* backend for performing neural network operations.

It was made sure that each run of the simulator does not consume entire Graphical Processing Unit (GPU) resources, to enable running multiple instances of the simulator simultaneously. This feature is appended with logging debugging data and saving checkpoints at regular intervals to a different folder for each iteration, which avoided overwriting previous values. Different folders are created inside the specified log directory to store data for each run, the log directory and saving interval in which is a configurable parameter.

An agent terminates if it reaches the destination, collides into an obstacle or runs out of time to complete its task. Such states are called terminal states, and an episode is defined as a run from the initiation of a car object in random position until it reaches either of the terminal states. The time-out time is configurable and maybe changed based on the complexity of the environment and the approximate range of time estimated for an agent to reach the destination in the given environment.

At each epoch (end of an episode) of an agent, the total reward collected by the agent, the terminal state, the mean reward and the epoch number is logged for each agent. This log file along with the learned neural network weights and model is saved at regular intervals configured by the user. After each epoch, the state of the agent and the destination may be randomized, each individually configurable by the user.

D. Reconfigurability

A configuration file can be passed as an argument to the simulator to load all the aforementioned parameters. If no configuration file is provided, the simulator expects a *config.ini* file inside the directory where the simulator is run. As stated earlier, each run is logged into a separate folder inside the directory specified by the user. The configuration file used for the run is also copied into the folder, as a reference to the process details.

III. Evaluation

In order to test the simulator, the example DQN class provided was used to train on several arenas, where the agent and the destination were randomized at each epoch. The car is

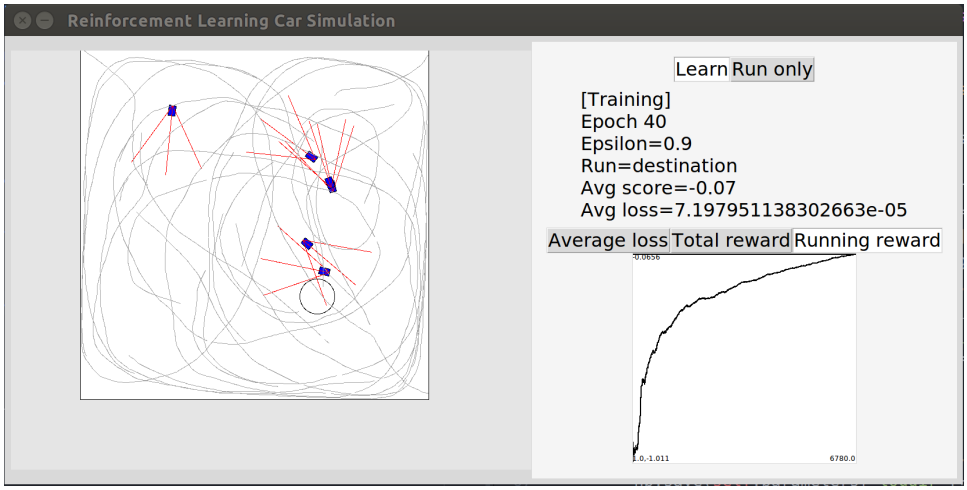


Fig. 5: Example of the environment when resuming training from a previously saved checkpoint

equipped with 3 sensors each of which has a range of 2 units, offset at angles of 30^o from each other. Simple obstacles are introduced in the area, and modification to the shape of the area is performed.

Figures 6-8 show different types of arenas which can be defined and how the algorithm uses the sensors in each case. It is interesting to note the turn that the car makes around the corners, since using a bicycle model, it needs to make a larger turn to execute a turn without collision. It should also be noted that since the states are not absolute, there are fail cases, where the agent observes a point of no return (since there are no reverse actions), and learns to crash into the edge to restart the environment with a better scope of expected rewards. Figure 8 shows how multi agent multi arena simulation can be performed where several agents interact with the environment at the same time, with either same or different type of arenas.

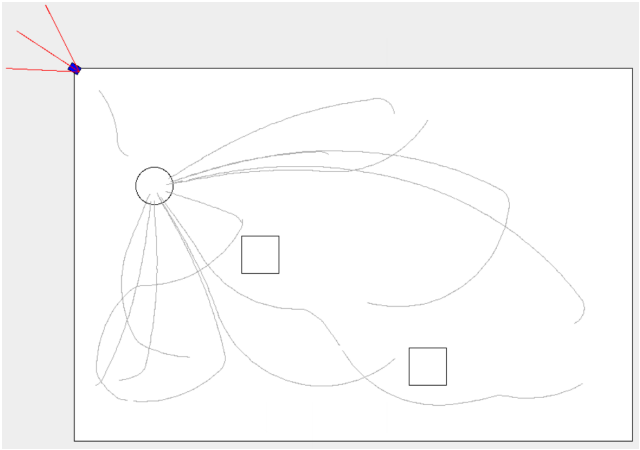


Fig. 6: Evaluation of the simulator learn objectives in different environments: Learning to avoid obstacles to reach destination

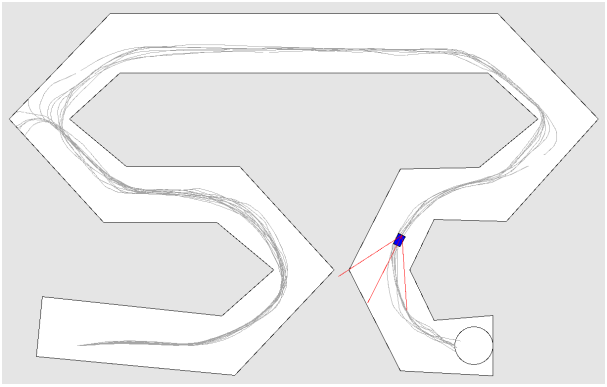


Fig. 7: Evaluation of the simulator learn objectives in different environments: Learning to reach target following track and cutting corners

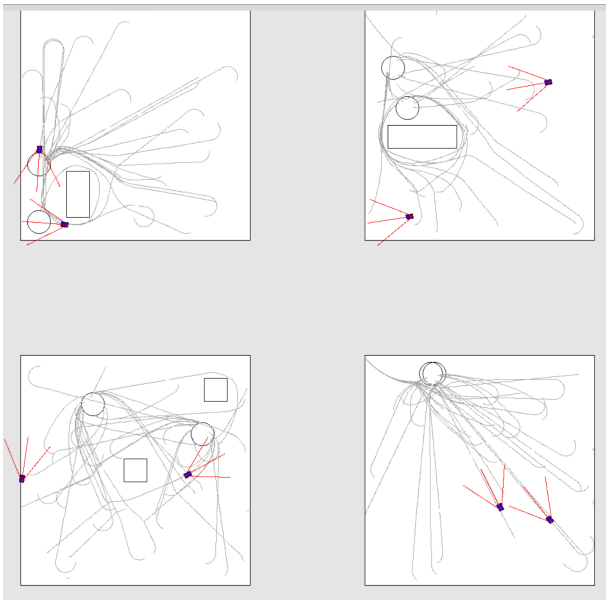


Fig. 8: Evaluation of the simulator learn objectives in different environments: Multi agent multi arena simulation

IV. Conclusion

This research produces a lightweight simulator, which can run multiple instances of itself in parallel to expedite the testing process in reinforcement learning. The simulator is oriented towards application of reinforcement learning algorithms to autonomous cars and robotics application with sensor integration. Experiments show how DQN algorithm can be implemented in the simulator to solve for the control of a non-holonomic system, using partially observable states. The simulator is made modular and reconfigurable to enable prototype testing. The source code is available from the authors repositories [8] available on-line.

V. Future developments

Though the simulator can be used for testing applications, several future modifications are planned. The use of mouse click to configure a track or an arena with obstacles is intended to be incorporated into the simulator. The update to the graphs is slow and needs to be fixed in further development. The current collision model for the car is a point, which is impractical and later release will have different collision models for the algorithms to learn the physical dimensions as well. Finally the integration with Robot Operating System (ROS) [9] is intended to enable prototyping applications in robotics.

References

[1] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," in *Field and Service Robotics*, 2017. [Online]. Available: https://arxiv.org/abs/1705.05065

[2] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *CoRR*, vol. abs/1606.01540, 2016. [Online]. Available: http://arxiv.org/abs/1606.01540

[3] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 1–16.

[4] B. P. Gerkey, R. T. Vaughan, and A. Howard, "The player/stage project: Tools for multi-robot and distributed sensor systems," in *In Proceedings of the 11th International Conference on Advanced Robotics*, 2003, pp. 317–323.

[5] A. Sallab, M. Abdou, E. Perot, and S. Yogamani, "Deep reinforcement learning framework for autonomous driving," vol. 2017, pp. 70–76, 01 2017.

[6] M. Cutler, T. J. Walsh, and J. P. How, "Real-world reinforcement learning via multifidelity simulators," *IEEE Transactions on Robotics*, vol. 31, no. 3, pp. 655–671, June 2015.

[7] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015. [Online]. Available: http://dx.doi.org/10.1038/nature14236

[8] A. Majumdar. Reinforcement learning car simulator. [Online]. Available: https://github.com/abhijitmajumdar/Reinforcement_Learning_Car_Simulator.git

[9] ROS.org. Ros technical overview. [Online]. Available: http://wiki.ros.org/ROS/Technical Overview