FLUIDS: A First-Order Local Urban Intersection Driving Simulator

Hankun Zhao*[1], Andrew Cui*[1], Schuyler A. Cullen[3], Brian Paden[3], Michael Laskey[1], Ken Goldberg[1,2]

***Abstract*— To facilitate automation of urban driving, we present an efficient, lightweight, open-source, first-order simulator with associated graphical display and algorithmic supervisors. The goal of FLUIDS is to allow for large-scale data collection of challenging simulated traffic intersections with varying state configurations and large-scale evaluation for early stage development of planning algorithms. FLUIDS supports an image-based birds-eye state space and a lower dimensional quasi-LIDAR representation. FLUIDS additionally provides an implemented algorithmic planner as a baseline controller. We find that FLUIDS can gather data in parallel from the baseline controller at 4000 state-action pairs per minute and evaluate in parallel an imitation learned policy on the baseline at 20K evaluations per minute. We demonstrate the usefulness of FLUIDS' data collection speeds by training a velocity controller for avoiding collisions and obeying traffic laws using imitation learning on the provided baseline controller as the supervisor. We also use FLUIDS to automate an extensive sensitivity analysis of the learned model to various simulation parameters. FLUIDS 1.0 is available at `https://berkeleyautomation.github.io/Urban_Driving_Simulator/`.**

I. Introduction

A large number of end-to-end driving simulators have been designed to prototype algorithms for autonomous cars. Most simulators focus on providing complex photo-realistic "drivers-eye" perspective renderings of the physical world and modeling accurate car dynamics [25][7][31]. While these simulators are ideal for testing the perception and the control software stack of an autonomous car, they are potentially too rigid and data-intensive for early-stage prototyping new learning algorithms for the behavior of autonomous vehicles.

To allow for large-scale data collection and rapid evaluation, we present FLUIDS (First-Order Local Urban Intersection Driving Simulator). FLUIDS is a lightweight and efficient simulator for multi-agent driving and pedestrian behavior at intersections. The Python-based simulator is designed to simulate plausible driver behavior at intersections quickly. FLUIDS provide the ability to test both traditional planning algorithms and model-free learning agents in a large number of varying traffic conditions. As shown in Fig. 1, we examine intersections where a variety of cars must coordinate to achieve safe, collision-free behavior.

In Section IV we discuss how FLUIDS is designed and implemented. As a simulator built on Python, FLUIDS is designed to be configurable to new types of traffic intersections. The simulator provides an API that allows for user-designed N-way intersections or more complex scenarios such as roundabout traffic circles.

*Denotes Equal Contribution
[1]Department of Electrical Engineering and Computer Science
[2]Department of Industrial Engineering and Operations Research
[3]Samsung Strategy and Innovation Center
[1−2]The AUTOLAB at UC Berkeley; Berkeley, CA 94720, USA
jerryz123@berkeley.edu , andycui97@berkeley.edu, laskeymd@berkeley.edu, goldberg@berkeley.edu

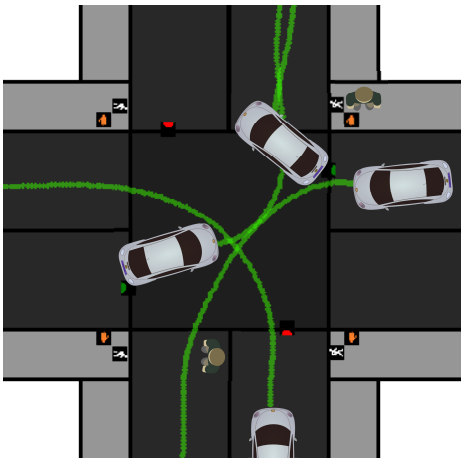

Fig. 1: An example of a driving intersection in FLUIDS showing four cars, two pedestrians, and traffic lights. FLUIDS focuses on efficiently modeling the interaction between different drivers and pedestrians. The colored lines represent trajectories generated by FLUIDS' nominal trajectory generator.

In Section V we discuss the state spaces supported by FLUIDS. FLUIDS provides access to three different types of state representations. The first is a explicity representation, which gives the agents access to the full state of the world. The second is a bird's-eye perspective, which is a high-dimensional RGB image representation generated from an overhead perspective. The third is a quasi-LIDAR state featurization, which represents the data a car could extract from LIDAR and camera sensors. For both the quasi-LIDAR representation and the birds-eye view, FLUIDS also provides a parameterized model for sensor noise.

In Section VI we discuss the control spaces in FLUIDS. FLUIDS offers control spaces at different hierarchical levels of an autonomous cars planner. The top level of control dictates the behavioral logic of the scene, which specifies the starting location and destination for each agent. Another possible control space is the Cartesian path of (x, y) points for each agent to follow to its target goal state. The third possible control space is the target velocities to set for the agent to avoid collisions and obey traffic laws. Finally, the fourth and lowest level in the hierarchy is steering angle and acceleration.

In Section VI we also discuss the planners FLUIDS provides at each level in the control space hierarchy. At the behavioral level, FLUIDS samples start and goal states uniformly. At the Cartesian path level, FLUIDS leverages RRT*. At the velocity level, FLUIDS provides a predictive planner. Finally, at the lowest level, FLUIDS provides a Proportional Integral Derivative (PID) controller to track the resulting trajectory generated from the planners mentioned above. These planners are intended to be used as a baseline for algorithms or as a way to collect preliminary training data.

Preprint submitted to 14th IEEE International Conference on Automation Science and Engineering. Received March 18, 2018.

In Section VII we describe our setup for using FLUIDS for data collection and evaluation of algorithms by performing imitation learning.

In Section VIII we evaluate the effectiveness of FLUIDS in three ways. First, we benchmark FLUIDS against other commonly used simulators. We then use the algorithmic supervisor and study how fast it can provide training data points, defined as a (state,action) pair for each agent, in FLUIDS for the imitation learner. We then train a policy as described in Section VII to learn a mapping from quasi-LIDAR observations to the target velocity of the car at each state and use FLUIDS to run a sensitivity analysis of the learned policy to six parameters.

Using a 3.2 GHz i7-970 CPU, we find that FLUIDS can collect data from its implemented supervisor at a rate of 5K data points/minute with four agents using ten threads of parallelization. Additionally, FLUIDS can rapidly evaluate the performance of the learned agent in parallel at a rate of 28K data points/minute. This rapid simulation enables a complex sensitivity analysis of the learned agent across 256 parameter configurations.

Summary of Contributions:

1) An open-source Python-based simulator for multi-driver and pedestrian behavioral modeling at traffic intersections.
2) Experiments demonstrating how FLUIDS can be used to collect large amounts of training data and quickly determine the sensitivity of driving algorithms to noise.

II. Related Work

We observe that existing driving simulators fall into one of two categories. End-to-end simulators provide environments which simulate both the perception and controls components of the autonomous vehicle stack, while first-order simulators offer a more focused, application-specific environment.

A. End-to-End Simulators

DeepGTAV [25] is a plugin for the open-world urban sandbox game Grand Theft Auto V and can be used to collect data for autonomous driving [24]. This module provides a TCP interface to the driving portion of the game. DeepGTAV offers a detailed and realistic "drivers-eye viewpoint" to provide an end-to-end environment for driving. However, the game engine backend prevents the user from running faster than real time, or extensively customized simulations. Furthermore, the behavior of non-player agents is not configurable.

CARLA [7] is a recent design of an open source 3D driving simulator. CARLA provides a high fidelity environment with varied weather, lighting, and pedestrian effects. A cost function drives pedestrian behavior, while a reactive finite state machine drives vehicle planning. Training agents in CARLA is challenging and data intensive.

TORCS (The Open Car Racing Simulator) [31] has been used as a sandbox for hundreds of artificial intelligence systems. TORCS is open-source, highly configurable, and has an extensive library of supporting software. However, TORCS models vehicle racing conditions from a drivers-eye viewpoint, rather than urban navigation, which may not apply to testing autonomous car planning for day to day driving.

An example of a driving simulator widely used in industrial applications is CarMaker [18]. CarMaker seeks to accurately model car dynamics for companies to test new vehicle models. These simulators also focus on interactions in a wide variety of driving situations including intersections. Unfortunately, CarMaker is not open-source, which leads to difficulty in collecting data and verifying experiments.

B. First Order Simulators

First order simulators are commonly used to develop and evaluate reinforcement and imitation learning algorithms. One example is OpenAI Gym's [5] set of simulation environments. These environments and their flexible API encourage rapid prototyping and parallel simulations. OpenAI Gym provides continuous control benchmarks like Mujoco, [29], a physics engine with a wide variety of simulated environments used to benchmark reinforcement learning algorithms [8].

In the driving domain, researchers have used lightweight, application-specific simulators. These include a computational framework for developing traffic controllers called Flow [30], a simulator to study inter-vehicle reactions in driving [26], and a simulator to study cooperative driving in highway environments [16]. OpenAI Gym also provides a driving environment implemented using Box2D that simulates a car driving around a racetrack [6].

While application-specific simulators exist there is not one that is easily re-configurable, supports multi-agent interactions, and is designed for simulating intersections. FLUIDS seeks to fill this niche by providing a lightweight, efficient, first-order, configurable driving simulator to model multi-agent behaviors at intersections for data collection and algorithm evaluation.

C. Autonomous Vehicle Planning

The controls stack of a typical autonomous vehicle can be decomposed into four components [17]. The route planning and behavioral layers select a high-level goal position for the motion planning and positional control layers to follow. In a dynamic urban environment, the behavioral layer must consider not only the static features of the environment but also the dynamic actions of other vehicles and pedestrians. For this task, a predictive planner which estimates the behavior of other agents is appropriate [32][10].

While predictive planners can generate safe behaviors in dynamic environments, running predictive planners in simulation is challenging due to the scaling of generated predictions in the number of agents. Although it is possible to solve multi-agent planning as a global optimization problem in polynomial time, such an approach struggles when run in simulation with an unpredictable trial agent [21][13]. The modular structure of an autonomous vehicle controller also allows for the development of components of the system independent from each other. FLUIDS provides a system to support both the rapid design and evaluation of these algorithms.

III. Problem Statement

Assumptions For evaluating agent behavior, FLUIDS assumes that users of the simulator have access to an agent, which can take a state of the world and return a control for the current time step following the OpenAI Gym interface.

The agent must be able to operate on either a bird's eye image representation of the world or a lower dimensional state, which specifies cars' and pedestrians' relative distances and velocities. Additionally, the tested agent must provide a control in one of three representations: a Cartesian target trajectory of (x, y) points, a target velocity, or an acceleration and steering angle. FLUIDS assumes a planar, two-dimensional world.

Objective The goal of FLUIDS is to provide a sandbox for prototyping the planning of an autonomous vehicle at traffic intersections. We aim to design a lightweight Python-based traffic intersection simulator that allows for generating a number of background agents that exhibit behavior plausible enough to test autonomous car algorithms quickly. FLUIDS aims to achieve this by accomplishing two goals:

1) Being able to collect large amounts of highly varied data from planners in the domain, to facilitate testing of learning algorithms.
2) Being able to quickly perform in parallel large-scale sensitivity analysis of a learned agents policy via exhaustive grid search over simulation parameters.

IV. Simulator Design

A. Terminology

Agent A dynamic object which receives a state representation from the system, and responds with an action to take.

State The collection of all simulated objects in the environment, including their position, orientation, and velocity.

Intersection A layout of sidewalks, lanes, terrain, and traffic signals which govern when and where cars and pedestrians may travel.

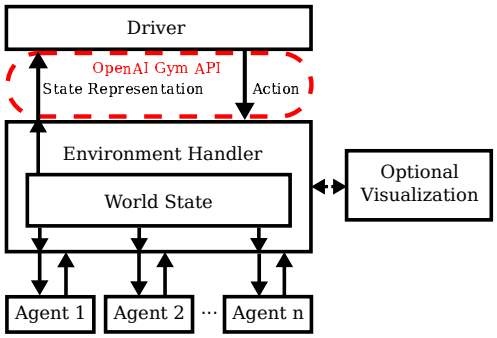

Fig. 2: FLUIDS' system architecture. The evaluated driver interfaces to the world via the OpenAI Gym interface. In the back-end, the simulator coordinates the behavior of additional background drivers and pedestrians. There is also an optional rendering feature for visualization of the simulator.

B. System Architecture

FLUIDS is built following OpenAI's Gym API to promote usability. The OpenAI Gym API provides a high-level interface governing interactions between the world environment and the experimental agent. FLUIDS conforms to this API for its background agents as well, allowing users to swap in multiple types of background agents if needed.

We implement FLUIDS with the demands of algorithm development in mind. To keep the codebase fluid and easily customizable to meet user specifications, we choose to use Python. Using Python also allows for interfacing with popular learning frameworks such as Tensorflow [1], PyTorch [19], and SciKit Learn [20].

FLUIDS can also be run using parallel frameworks such as Python's native multiprocessing package for quicker data collection and evaluation.

Fluids also provides a demonstration agent for controlling the background cars and pedestrians in a scene. These background agents are designed to test the ability of a user agent in different multi-agent interactions.

C. Agent Dynamics

The dynamic agents of the world are divided into three classes: pedestrians, cars, and signal lights.

1) Pedestrian Dynamics: A pedestrian is parameterized by the state space $\mathbf{x}_p(t) = [x, y, \theta, v]$, which corresponds to its positional coordinates, orientation and velocity. The control signal for a pedestrian at time t is specified as $\mathbf{u}_c(t) = [\psi, a]$, which refers to steering angle and acceleration.

For pedestrian dynamics FLUIDS uses a point model to reflect the omni-directional nature of pedestrian movement. The differential equations that describe the model are as follows.

$$\dot{x} = v\cos(\psi) \tag{1}$$
$$\dot{y} = v\sin(\psi) \tag{2}$$
$$\dot{v} = a \tag{3}$$
$$\dot{\theta} = \psi \tag{4}$$

2) Car Dynamics: Each car in FLUIDS is parameterized by the state space $\mathbf{x}_c(t) = [x, y, \theta, v]$, which corresponds to positional coordinates, orientation and velocity. The control signal for each car at time t is specified as $\mathbf{u}_c(t) = [\psi, a]$, which refers to steering angle and acceleration.

For cars, FLUIDS uses the kinematic bicycle model [22]. The differential equations that describe the model are as follows.

$$\dot{x} = v\cos(\theta + \beta) \tag{5}$$
$$\dot{y} = v\sin(\theta + \beta) \tag{6}$$
$$\dot{v} = a \tag{7}$$
$$\dot{\theta} = \frac{v}{l_r}\sin(\beta) \tag{8}$$
$$\beta = \tan^{-1}\left(\frac{l_r}{l_f + l_r}\tan(\psi)\right) \tag{9}$$

This model defines β as the angle between the current velocity of the center of mass with respect to the longitudinal axis of the car. l_r and l_f are the distance from the center of the mass of the vehicle to the front and rear axles. In FLUIDS, both l_r and l_f are set to be half the length of the car.

3) Traffic Lights and Crosswalk Lights: Traffic lights and crosswalk lights are set up at the end of each lane or sidewalk with which they are associated. Traffic lights appear in the standard three colors of red, yellow, and green. Crosswalk lights are either white or red. Cars are aware of traffic light states and pedestrians are aware of crosswalk light states. FLUIDS allows for both types of lights to loop through its colors, stopping for any specified time on each color. This control over the timing of lights allows for the designing

Preprint submitted to 14th IEEE International Conference on Automation Science and Engineering. Received March 18, 2018.

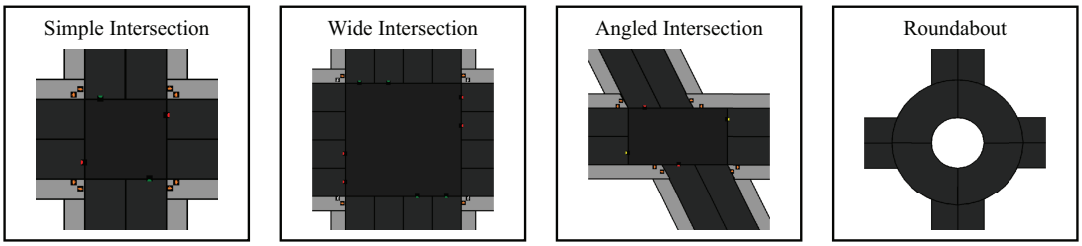

Fig. 3: A sample of the different types of traffic intersections that can be designed in FLUIDS. Intersection types include common intersection like a 4-way stop and more intricate ones such as a roundabout. Intersections can also contain sidewalks, traffic lights, and crosswalk lights.

of intersections with a wide variety of light synchronization schemes.

D. Collision Checking

FLUIDS performs collision checking using the Shapely geometry library [23]. Shapely provides a fast, configurable interface for manipulating and designing two-dimensional polygons. Every object in FLUIDS is associated with a Shapely object that acts as its bounding volume. At every timestep, collision checking is performed and all collisions logged.

E. Designing Intersections

An intersection is specified by K lanes, where each lane, l, is represented by its width, height, position, and direction. Each intersection may also contain sidewalks with similar representations to lanes. The simulator also consists of N dynamic agents, where $N = N_p + N_c + N_l$. N_p and N_c refer to the number of pedestrian and cars respectively. N_l refers to the number of traffic lights and crosswalk lights in the scene. Lights are considered dynamic objects, as the colors which describe their state space and dictate traffic flow change over time.

FLUIDS provides the ability to design a variety of intersections. Specifically, FLUIDS supports various static objects including lane, street, and sidewalk objects. Lane objects have an associated direction to prevent cars from driving the wrong way. FLUIDS also uses street objects, on which cars can travel in any direction, where lanes meet up at an intersection. Sidewalks represent areas where only pedestrians can travel.

In addition to static objects, FLUIDS also supports dynamic objects such as cars, pedestrians, and traffic lights. Obstacles represented by impassable terrain can be arranged arbitrarily through the scene, simulating realistic conditions such as parked cars, potholes, and road construction. Cars and pedestrians are generated automatically by lanes and sidewalks respectively, but traffic lights need to be manually placed and synchronized.

V. State Spaces in FLUIDS

FLUIDS offers three options for accessing the state space of the world; explicit, bird's-eye view, and quasi-LIDAR perspective.

A. Explicit View

In the explicit view the agent has access to the true state space (i.e., x_c and x_p of all N agents in the scene) Additionally, the agent can observe the state of all traffic lights and pedestrian crosswalk lights.

This view can be seen as having access to a global oracle where all information is available to an agent. FLUIDS also implements a Gaussian noise model over this space.

B. Bird's-Eye View

In bird's-eye view, a top-down image of the intersection is available to the agent. These images use an RGB color representation of configurable resolution.

Viewpoints from above, such as those from unmanned aerial vehicles [27] or traffic cameras [14] motivate this state space. In this state representation, agents must infer from pixel data the location of other agents in the scene and the current state of traffic lights. This state representation is relevant for potentially evaluating the efficiency of recent advances in Convolutional Neural Networks for planning [15].

FLUIDS additionally provides a sensor noise model for this state representation. Specifically, FLUIDS supports adding a bounded zero-mean Gaussian noise to each pixel value, with variance defined by λ_p.

C. Quasi-LIDAR Representation

Quasi-LIDAR representation is a state space that models the world with information that mimics what an autonomous car extracts from the physical world. With recent advances in LIDAR and visual object detection, it is possible for a car to extract relative poses and class labels of other agents adjacent to it [3][9].

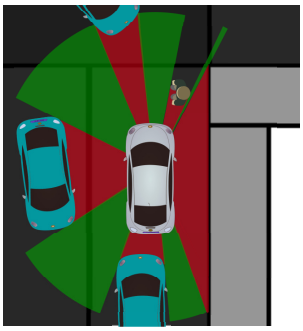

Fig. 5: Illustration of the quasi-LIDAR perspective state space. Rays are projected into the scene from the car's perspective. The red rays correspond to detected objects in the environment. The density of projected rays and noise model are both configurable.

To create the quasi-LIDAR state space m rays are projected from the agent's car at angular intervals along a 360° arc. If a ray collides with an object in the scene, it returns the following information $\{d, \text{label}, \psi_r, v_r\}$, which corresponds to the distance to, the class label of, the relative angle to, and the relative velocity to the object. Fig. 5 shows an illustration of this state space.

FLUIDS models the sensor noise of this state space, via additive element-wise zero-mean Gaussian noise to the relative distance, velocity, and angle. The parameter λ_l specifies the Gaussian noise. In addition to Gaussian noise, FLUIDS also models dropped observations. With probability ϵ a reading from a given ray is returned as empty space, to model the effect of missing data.

Preprint submitted to 14th IEEE International Conference on Automation Science and Engineering. Received March 18, 2018.

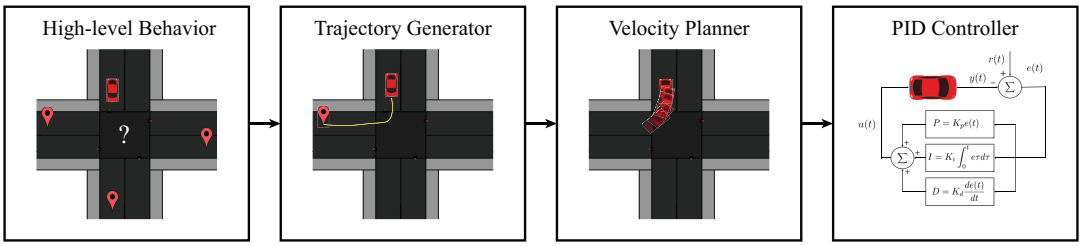

Fig. 4: The planner for the built-in driving agent. First, a high-level destination, specified as a destination lane, is randomly selected for the agent. Then a positional trajectory is generated via OMPL to take the agent to the goal position. A velocity planner is then called to ensure the agent reaches the destination without collisions or violation of traffic laws with respect to the traffic light. Finally, a PID controller is used to execute the planned path.

VI. Hierarchical Control Spaces

FLUIDS supports the ability to test algorithms at different levels of the planning hierarchy that exist in autonomous cars. The implemented planner consists of four components: a high-level behavior component, a nominal trajectory generator, a velocity planner, and a PID controller. These four components encompass what is commonly referred to as the high-level behavioral layer, motion planner, and local feedback control components of the autonomous car control stack [17].

For each level of the hierarchy, FLUIDS provides an algorithmic supervisor that generates plausible behavior. Additionally, users can operate on one level of the hierarchy and use the planner for the others. Users are also able to implement their own controllers at each level of the hierarchy if they wish.

A. Behavioral Logic

Control Space Each agent in the driving simulator needs to be assigned a target lane for it go towards. Given K lanes, the control space for the agent is the selection of a target lane (i.e. $\mathcal{U} = \{l_0, ..., l_K\}$).

Implemented Supervisor The implemented planner operates by sampling a start lane uniformly and target lane with the following probability distribution

$$p(l_t|l_g) = \begin{cases} \frac{1}{K-1} & \text{if } l_t \neq l_g \\ 0 & \text{if } l_t = l_g \end{cases},$$

which corresponds to assigning uniform probabilities to the other $K-1$ lanes in the scene and zero to the start lane.

B. Nominal Trajectory Generator

Control Space Once any car has an intended goal state, the next level in the hierarchy is to generate a path in Cartesian (x, y) space towards some designated goal state given an initial state representation of the world. Formally, the agent's control space is a set of 2D-positional points (i.e. $U \in \{\mathbb{R} \times \mathbb{R}\}^T$).

Implemented Supervisor The implemented planner uses Open Motion Planning Library (OMPL) [28] to generate the nominal trajectory. We formulate the problem as geometric planning around static restricted areas such as impassable terrain or, in the case of driving, sidewalks. Avoiding collisions with other dynamic objects is handled by the velocity planner and is not considered in this step. We use the rapidly expanding random tree algorithm (RRT*) [12], which plans the x,y positioning of the car. RRT* can be used as a search algorithm over geometric space, and has been used extensively in motion planning for autonomous vehicles [17] [2]. The generated trajectory is interpolated using cubic splines, then smoothed using a Gaussian filter. FLUIDS additionally provides a geometric planner for pedestrians to guide them to the cross the streets.

C. Velocity Planner

Control Space Similar to [4], we have an agent that plans for the velocity of the trajectory after generation of the nominal trajectory. The agent on this level is given a current state representation of the world and must select the velocity from the bounded range $\mathcal{U} \in [0, v_{max}]$, where v_{max} corresponds to the maximum value.

Implemented Supervisor The implemented velocity planner performs a forward projection for each agent for T time-steps in the environment. Then the velocity is incrementally lowered until the path is collision-free.

If no collision-free path is found, the planner will greedily instruct the agent to stop. If, however, a collision-free path is found the agent is instructed to go at v_{max}, which is set to correspond to city driving speeds.

The forward projection algorithm is advantageous because it can simulate the effect of each velocity signal and prevent collisions from occurring. FLUIDS also use this velocity planner to control pedestrian velocities to avoid collision.

D. Control Input

Control Space Given a trajectory of positional (x, y) points and target velocities, we now need to generate controls that can accurately track the trajectory. In this low-level of the hierarchy, the input is steering angle and acceleration, which an agent needs to determine given some current state representation of the world.

Implemented Supervisor FLUIDS uses two PID controllers. One controls the steering angle, ψ, to track the direction to the next positional point and the other controls the acceleration a to the next target velocity. We empirically observe the PID controllers can track over 99% of 200 test trajectories generated by the planners from higher in the heirarchy.

VII. Benchmarking FLUIDS with Imitation Learning

To evaluate FLUIDS as a platform for developing autonomous vehicle agents at multiple levels, we explore learning a velocity planner from the FLUIDS predictive velocity planner. We collect training data from rollouts of the supervisor on cars interacting in a four-way intersection. We study how quickly the data can be collected, how fast the learned policy can be evaluated, and how sensitive the learned policy is to noise.

Preprint submitted to 14th IEEE International Conference on Automation Science and Engineering. Received March 18, 2018.

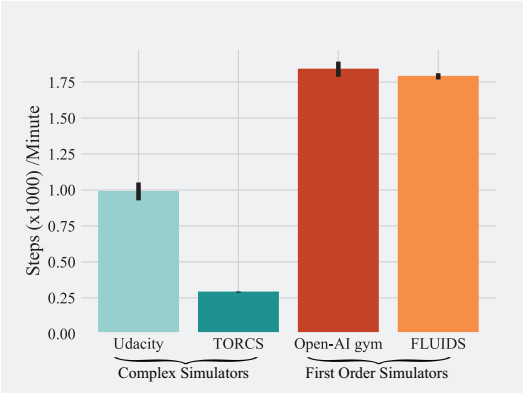

Fig. 6: Comparison of FLUIDS to other common driving simulators with rendering on. FLUIDS is simulating a four-way intersection with an idling car. FLUIDS demonstrates comparable steps per second to other lightweight simulators such as Open-AI gym and significant improvement over graphically heavy simulators such as TORCS and Udacity.

Fig. 7: The performance of the FLUIDS built-in supervisor. The FLUIDS supervisor performs reliably, with over a 90% success rate for up to 7 agents.

VIII. Experiments

Our experiments seek to answer three questions.

1) How fast is the FLUIDS simulator?
2) How much data can FLUIDS collect from driving algorithms?
3) How well can FLUIDS provide an extensive in-depth analysis of driving algorithms?

We perform all timing experiments on a 6-core 12-thread 3.2GHz i7-970 CPU. Parallel timing experiments were collected using Python's multiprocessing module.

A. Benchmarking Simulation Speed

To understand the speed at which FLUIDS enables simulation of the environment, we benchmark its steps per second of three different simulators Udacity, TORCS and OpenAI against FLUIDS. The recorded speeds [11], measured in thousands of steps per minute, with rendering enabled, are reported in Fig. 6. In each of these experiments, the simplest scenario is loaded to get the absolute fastest speeds each simulator can run. In FLUIDS, we load a four-way intersection with traffic lights and one idling car.

We observe that FLUIDS allow for faster simulation than Udacity and TORCS. We also note that FLUIDS is on par with OpenAI's lightweight racetrack simulation, showing that the simulator portion contributes little additional overhead when running experiments. The additional timing experiments in Table I shows FLUIDS performing at nearly 70 times its maximum speed with rendering. This result demonstrates the lightweight framework for the simulator in FLUIDS that allows for experiments to be run quickly without the overhead of visual rendering.

FLUIDS Speed	mean	stdev
with rendering	1.79	.0214
without rendering	131.00	2.24

TABLE I: Steps (x1000) per minute for FLUIDS with and without rendering.

B. Collecting of Data for Learning

For the rest of the experiments in this paper we disable real-time visual rendering for data collection and policy evaluation.

We now examine how fast FLUIDS can collect data points for learning. In this experiment, we use the implemented supervisor as the demonstrator and collect 800 trajectories. We sample new demonstrations from an initial state distribution that ranges from $2-10$ cars.

For the learner's policy representation, we use SciKit-Learn's decision tree implementation [20]. Additionally, the state space of the world used for imitation learning experiments is the quasi-LIDAR representation. We choose the quasi-LIDAR representation for its lower dimensional featurization.

We collect 800 trajectories of data from the implemented supervisor. From each trajectory, we extract data points as state-action pairs for every vehicle in the trajectory. In Fig. 8, we evaluate how many data points we can obtain per minute with the algorithmic supervisor. For 2 cars and parallelization across 10 cores, we can collect over 8000 data points per minute. Adding cars decreases the collection rate as the velocity planner takes a longer time to plan each time step. Despite this, we still collect over 2000 data points per minute for a dense 7 car scene. These results also show the importance of the parallelized data collection, which consistently shows speedups of over four times that of single-threaded data collection.

Additionally, we roll out the learned policy and collect data from it to study the potential data collection limits of FLUIDS with a fast agent using learned behavior. The results, shown in Fig. 8, indicate that data collection speeds are improved by a factor of four again over data collection speeds on the supervisor. Run in parallel across 10 cores, and with 2 agents, FLUIDS manages over 24000 data points per minute. The performance dropoff due to overhead in the parallel experiments as the number of agents increases is not nearly as severe, with FLUIDS maintaining over 15000 data points per minute for up to 7 cars in a scene.

We also report the success rate of the supervisor versus the number of cars in the current scene. Fig. 7 shows that the supervisor agent can handle scenes with up to and including seven cars with over 80% success rate. A run of the simulator is designated successful if all the cars make it through the intersection without gridlock, colliding with other cars, and breaking traffic laws. However, with more cars, there is a steep dropoff showing the supervisor struggles to coordinate on challenging scenes with many multi-agent interactions.

C. Evaluation of Policy

Given the learned policy, we use FLUIDS to quickly evaluate how sensitive it is to observation noise and simulator

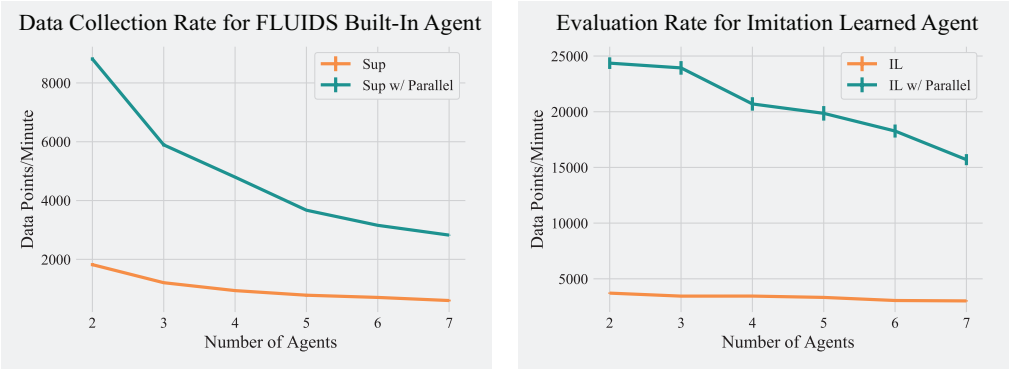

Fig. 8: The data collection rate for the FLUIDS built-in agent (left) is compared to the number of agents simulated. This data is used to train an imitation learned agent using the built-in agent as the supervisor. The evaluation rate of the imitation learned agent (right) is also compared to the number of agents simulated. We note the expected dropoff with the increased number of supervisor agents as well as the 5x speedup with parallelization. We also note a less

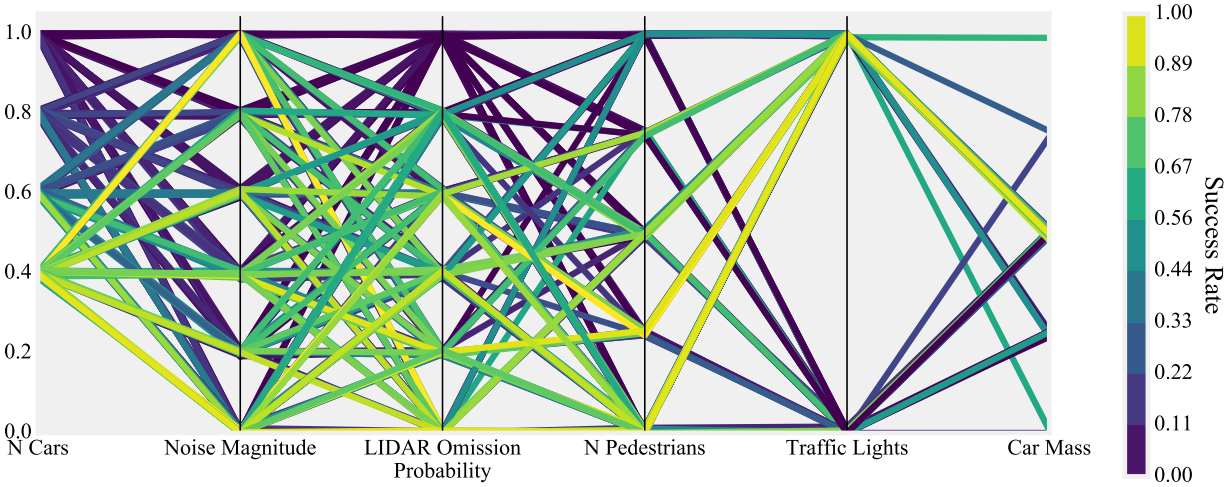

Fig. 9: Sensitivity analysis of the imitation learner to state complexity and observation noise, performed over 256 configurations by evaluating the learned policy over half a million data points. The simulator facilitates a rapid parallel search over these configurations. For visualization we normalize all parameters between 0 and 1. We sweep betwwen 2 and 5 cars, 0 and 4 pedestrians, and vary the noise and dynamics parameters. We report success rate as the percentage of evaluations in which the learned policy successfully guides all vehicles to their goals. The analysis reports that our imitation learned agent is tolerant to noisy observations, but fails when run in an environment with a large number of cars or pedestrians. We also observe model mismatch when we alter the car mass and remove traffic lights.

parameters. In this experiment, we show how FLUIDS can effectively conduct a sensitivity analysis.

We evaluate the sensitivity of the learned velocity controller to parameters in the simulator. We perform a grid search over six parameters of the simulator:

- The number of cars in the scene, in the range $[2, 5]$.
- The number of pedestrians in the scene, in the range $[0, 4]$.
- The variance, λ_l of quasi-LIDAR noise, in the range $[0, 1]$.
- The probability of omitting a sensor reading from the quasi-LIDAR state, in the range $[0, 1]$.
- The presence of traffic lights, $[0, 1]$.
- The mass of the cars, in the range $[0, 200]$.

Fig. 9 reports the generated sensitivity analysis. FLUIDS evaluates the learned policy across 256 configurations, running in total over 5000 trajectories. The analysis reports that the performance of the learned policy is very sensitive to perturbations in the number of cars in the scene, with more cars providing a challenge and leading to a lower success rate. The sensitivity analysis also allows us to visualize the effect of quasi-LIDAR omissions, with our imitation learned agent being robust to increased omission probability until all the quasi-LIDAR observations are omitted, after which the success rate drops drastically. Additionally, the sensitivity analysis indicates the importance of traffic lights, which coordinate traffic through the intersection and prevent gridlock. Finally, the sensitivity analysis shows the importance of car mass, with slightly different car masses providing a model mismatch and leading to lower success rates.

All of these conclusions illustrate the effectiveness of the sensitivity analysis that FLUIDS is able to provide, which will allow researchers to quickly and better understand their autonomous driving algorithms.

IX. Discussion and Future Work

FLUIDS is a fast first-order simulator of multi-agent driving behaviors. The interface is lightweight, highly configurable, suitable for developing and evaluating autonomous agents in urban environments.

FLUIDS can also be extended to cover a more diverse array of driving situations beyond intersections. Examples include

navigating a dense parking lot or successfully merging onto a highway lane. The variety of agents can also be expanded to include bicyclists, emergency services, and cars with trailers. Adding a more diverse set of signals such as yield signs, railroad crossings signs, and turn only signals will also allow for more diverse intersections.

There are also a few major components that will make FLUIDS more accessible to researchers. Creating an interface to programmatically generate, load, or save new intersections will allow researchers to conduct extensive studies across a wide variety of intersections, hone in on particularly tricky configurations for an in-depth study, and even share driving scenarios with others. Creating a ROS interface or another API will allow other researchers with well-established software stacks to easily integrate with FLUIDS.

X. Acknowledgments

This research was performed at the AUTOLAB at UC Berkeley in affiliation with the Berkeley AI Research (BAIR) Lab, the Real-Time Intelligent Secure Execution (RISE) Lab, and the CITRIS "People and Robots" (CPAR) Initiative and with UC Berkeley's Center for Automation and Learning for Medical Robotics (Cal-MR). The authors were supported in part by donations from Samsung, Siemens, Google, Honda, Intel, Comcast, Cisco, Autodesk, Amazon Robotics, Toyota Research Institute, ABB, Knapp, and Loccioni.

We thank our colleagues who provided helpful feedback and suggestions, in particular, Daniel Seita, and Sanjay Krishnan.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[2] A. Arab, K. Yu, J. Yi, and D. Song, "Motion planning for aggressive autonomous vehicle maneuvers," in *Automation Science and Engineering (CASE), 2016 IEEE International Conference on*. IEEE, 2016, pp. 221–226.

[3] R. Bergholz, K. Timm, and H. Weisser, "Autonomous vehicle arrangement and method for controlling an autonomous vehicle," Nov. 21 2000, uS Patent 6,151,539.

[4] A. Best, S. Narang, L. Pasqualin, D. Barber, and D. Manocha, "Autonovi: Autonomous vehicle planning with dynamic maneuvers and traffic constraints," *arXiv preprint arXiv:1703.08561*, 2017.

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

[6] E. Catto, "Box2d: A 2d physics engine for games," 2011.

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

[8] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, "Benchmarking deep reinforcement learning for continuous control," in *International Conference on Machine Learning*, 2016, pp. 1329–1338.

[9] Q. Fan, Y. Yi, L. Hao, F. Mengyin, and W. Shunting, "Semantic motion segmentation for urban dynamic scene understanding," in *Automation Science and Engineering (CASE), 2016 IEEE International Conference on*. IEEE, 2016, pp. 497–502.

[10] A. F. Foka and P. E. Trahanias, "Predictive autonomous robot navigation," in *Intelligent Robots and Systems (IROS), 2002. IEEE/RSJ International Conference on*, vol. 1. IEEE, 2002, pp. 490–495.

[11] W. Hsieh, "First order driving simulator," Master's thesis, EECS Department, University of California, Berkeley, May 2017. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-102.html

[12] S. Karaman and E. Frazzoli, "Incremental sampling-based algorithms for optimal motion planning," *Robotics Science and Systems (RSS) VI*, vol. 104, p. 2, 2010.

[13] M. M. Khorshid, R. C. Holte, and N. R. Sturtevant, "A polynomial-time algorithm for non-optimal multi-agent pathfinding," in *Fourth Annual Symposium on Combinatorial Search*, 2011.

[14] D. Koller, J. Weber, T. Huang, J. Malik, G. Ogasawara, B. Rao, and S. Russell, "Towards robust automatic traffic scene analysis in real-time," in *Pattern Recognition, 1994. Vol. 1-Conference A: Computer Vision & Image Processing., Proceedings of the 12th IAPR International Conference on*, vol. 1. IEEE, 1994, pp. 126–131.

[15] 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.

[16] S. Noh, K. An, and W. Han, "Situation assessment and behavior decision for vehicle/driver cooperative driving in highway environments," in *Automation Science and Engineering (CASE), 2015 IEEE International Conference on*. IEEE, 2015, pp. 626–633.

[17] B. Paden, M. p, S. Z. Yong, D. Yershov, and E. Frazzoli, "A survey of motion planning and control techniques for self-driving urban vehicles," *IEEE Transactions on Intelligent Vehicles*, vol. 1, no. 1, pp. 33–55, March 2016.

[18] H. Palm, J. Holzmann, S.-A. Schneider, and H.-M. Koegeler, "The future of car design systems engineering based optimisation," *ATZ worldwide*, vol. 115, no. 6, pp. 42–47, 2013.

[19] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[20] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[21] J. Peng and S. Akella, "Coordinating multiple robots with kinodynamic constraints along specified paths," *The International Journal of Robotics Research (IJRR)*, vol. 24, no. 4, pp. 295–310, 2005.

[22] P. Polack, F. Altché, B. d'Andréa Novel, and A. de La Fortelle, "The kinematic bicycle model: A consistent model for planning feasible trajectories for autonomous vehicles?" in *Intelligent Vehicles Symposium (IV), 2017 IEEE*. IEEE, 2017, pp. 812–818.

[23] S. Popinet *et al.*, "Shapely," https://pypi.python.org/pypi/Shapely, 2014–2018.

[24] S. R. Richter, V. Vineet, S. Roth, and V. Koltun, "Playing for data: Ground truth from computer games," in *European Conference on Computer Vision (ECCV)*. Springer, 2016, pp. 102–118.

[25] A. Ruano, "DeepGTAV," https://github.com/aitorzip/DeepGTAV, 2016.

[26] D. Sadigh, S. Sastry, S. A. Seshia, and A. D. Dragan, "Planning for autonomous cars that leverage effects on human actions." in *Robotics: Science and Systems*, 2016.

[27] G. Salvo, L. Caruso, A. Scordo, G. Guido, and A. Vitale, "Traffic data acquirement by unmanned aerial vehicle," *European Journal of Remote Sensing*, vol. 50, no. 1, pp. 343–351, 2017.

[28] I. A. Şucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82, December 2012, http://ompl.kavrakilab.org.

[29] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*. IEEE, 2012, pp. 5026–5033.

[30] C. Wu, A. Kreidieh, K. Parvate, E. Vinitsky, and A. M. Bayen, "Flow: Architecture and benchmarking for reinforcement learning in traffic control," *arXiv preprint arXiv:1710.05465*, 2017.

[31] B. Wymann, E. Espié, C. Guionneau, C. Dimitrakakis, R. Coulom, and A. Sumner, "Torcs, the open racing car simulator," *Software available at http://torcs. sourceforge. net*, vol. 4, 2000.

[32] J. Zhu, D. I. Ferguson, and D. A. Dolgov, "System and method for predicting behaviors of detected objects," Feb. 25 2014, uS Patent 8,660,734.