AutoMice: A Testbed Framework for Self-Driving Systems

***Abstract*—Testing self-driving systems and algorithms is challenging. The widely used methods include simulation and road test. Simulation is low cost but it is difficult to include all the physical world details with a realistic setting. A road test is able to capture all the complexity of real traffic and road conditions, but it is costly and risky, i.e., imagine one line of code change may require thousands of miles of road testing.**

This paper presents AutoMice, a testbed framework that offers developers an environment to experiment with self-driving algorithms. It eases the transition from testbed validation to deployment in production by using two abstraction layers that hide the hardware details and provide unified APIs to the core system modules. The development and validation by using AutoMice follows a two-phase design process—a development phase and a deployment phase. In the development phase, the developers can implement the core self-driving system modules with given system APIs that provided by the two abstraction layers. In the deployment phase, the same core system modules can be used in the testbed vehicle system to test the functionality and performance of the modules. By ensuring the abstraction layers are compatible with a real self-driving system, the same procedure can be used in self-driving production. To demonstrate the usability of AutoMice, we implement several self-driving perception and control algorithms in the development phase and re-use the same code in the deployment phase. The system modules and algorithms we developed cover object detection, remote control, vehicle-to-infrastructure communications, 3D map construction and localization, etc. We implement AutoMice on an Android phone powered self-driving car. We believe AutoMice can be easily implemented on other platforms and ease the evaluation of self-driving systems.

I. Introduction

A self-driving vehicle is capable of sensing its environment and navigating without human input. They use various sensors to sense and understand the physical world and then use drive-by-wire control technique to navigate the vehicle from point A to point B [1], [2], [3]. In order to navigate the vehicle in the physical world, a High Definition(HD) 3D map is constructed. The HD 3D map records the static road network, the lane markers, traffic signs and traffic lights. The self-driving system uses a perception module to localize the vehicle in the 3D map and identify the moving objects, such as vehicles and pedestrians. It must also be able to predict the movement of the objects and then uses a planning and control module to control the vehicle. The planning and control module has to avoid any possible collisions and navigate the vehicle to the destination efficiently. Sometimes, a remote assistant module is also required for a remote operator to instruct the vehicle in the scenario when the self-driving system could not decide next move and there is no in-vehicle driver.

The self-driving system is very complex and there are many open and challenging problems. Among all the problems, we are interested in how to test and validate the core system modules with low-cost and extensible infrastructures. In other words, given an updated solution for a particular problem, e.g., 3D map update, how we can verify the new solution works. There are two viable solutions exist: simulation and road test. Simulation simplifies the physical interactions so that it is not able to capture all the details in the real world [4], [5]. On the other hand, the road test is too expensive in many cases, i.e., just imagine one line code change may lead to thousands of miles of testing.

In this paper, we present AutoMice, a self-driving testbed that can be used to test and validate self-driving solutions in a low-cost way but close to realistic settings. The basic idea is to use a custom small vehicle and small scale road infrastructures to validate the self-driving functionalities. One challenge in such a case is the transition from the testbed to a real production self-driving car, since we do not want the developers to write the code twice for the same functionality. To ease such transition, AutoMice provides two abstraction layers. The two layers are used to get the data from sensors and send the control commands to the custom vehicle. Essentially they provide a system abstraction which separates the data processing module from the hardware interaction modules. In such a design, the same code running on AutoMice for the data processing and decision-making modules can also be used in production.

AutoMice consists of two phases: development phase and deployment phase. In the development phase, the algorithms and system modules are developed as the normal process before shipping to production. In the deployment phase, the algorithms and system modules are shipped to the testbed environment. The code written in the development phase can be used in the production directly.

In our prototype implementation, we use an Android phone to build the self-driving system which controls a custom toy car. The self-driving system modules are implemented as a native application written with C/C++. C/C++ is widely used by the self-driving companies due to its high efficiency and portability. We define several APIs on the Android as the abstraction of the hardware, and it is expected that a self-driving system in production shares the same set of APIs. Once we finish the development of the algorithms and system modules, the code is shipped onto the Android platform without any modification.

We use several study cases of perception and control tasks to demonstrate the usability of AutoMice. One study case is about the perception and control of the self-driving systems. For example, if there is a stop sign in front of the vehicle, the running vehicle must be able to stop. Similarly, if the traffic light runs red, the vehicle must be able to stop as well. Such functionality involves both perception and control, and can be tested by using AutoMice with a smaller scale of traffic sign infrastructure. We also demonstrate that AutoMice can be used to evaluate 3D map construction, remote control and vehicle-to-vehicle communication protocols, by using both indoor controlled experiments and outdoor uncontrolled experiments.

We realize the promise of AutoMice by addressing several conceptual as well as technical challenges. Our main contribution is the design and implementation of AutoMice, a self-driving testbed to ease algorithm and system validation in a low-cost but close to a realistic environment. It uses a two-phase design to ease the transition from algorithm validation to production deployment. It separates the hardware interaction code with the data processing code so that the self-driving system developers do not have to understand the implementation of the testbed. To demonstrate the usability of AutoMice, we develop several perception, control, and communication system modules and deploy/validate them on the testbed.

II. Related Work

A. Self-Driving Systems

There are many corporations and researchers are developing fully self-driving techniques, such as Cruise Automation, Waymo, Mercedes-Benz and AutoX [1], [2], [3]. Waymo uses Lidar as the primary input for object detection [1]. AutoX proposes camera-first self-driving solution to reduce the cost to build a self-driving vehicle [2]. [6] presents high level possible challenges and directions for remote control systems. [7] presents a sensory-fusion perception framework that combines Lidar point cloud and RGB images as input and predicts oriented 3D bounding boxes. [8] describes the architecture and implementation of an autonomous vehicle designed to navigate using locally perceived information in preference to potentially inaccurate or incomplete map data. The existing validation techniques include simulation and real road test, both of which have their own limitations. AutoMice provides a tradeoff for the validation of some system modules and ease the testing of self-driving algorithms.

B. Simulation

Simulation is a relatively low-cost solution (comparing with testing on real road) for self-driving system testing and validation. A self-driving simulator has to build a 3D virtual world from collected real world data or a digital model. To interact with the virtual self-driving vehicle, it also simulates the sensor input, images from cameras and point cloud data from Lidar, etc., as well as the action of the vehicle based on the calculation of the self-driving system modules. To simulate the camera input, the simulator projects the 3D model to a 2D surface that aligns with the camera lens. The projection of 3D Lidar data has the similar line of sight rule, the only difference is that the point cloud data has higher density so that an object far away can still be preserved just like an object nearby. According to these sensor inputs, the self-driving system modules calculates the reaction, i.e., steering angle and acceleration, to avoid possible collisions and keep the vehicle within the correct lane. The detailed physical interaction between the car and other objects are very challenging to simulate precisely. [4] presents a simulator to train self-driving systems by using various methods, i.e., a classic modular pipeline, an end-to-end model trained via imitation learning, and an end-to-end model trained via reinforcement learning. [5] presents real-time synthetic benchmark and evaluation system that focus on computer vision and graphics applications, so the processing of Lidar and Radar data is not included. [9] presents an open-source simulation framework for cooperative interacting automobiles, and it focuses on the interaction of traffic participants instead of individual vehicles. Simulators provide a low-cost way to train and validate self-driving algorithms. However, it takes much more efforts to develop realistic settings and physical interaction in a simulated world. Comparing with simulator, AutoMice provides a low-cost way to train and test self-driving algorithms with much more realistic settings.

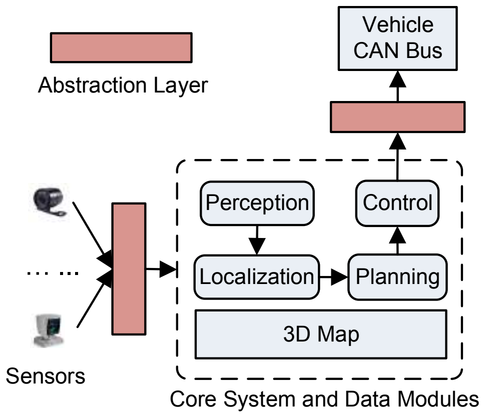

Fig. 1: The overview of AutoMice.

III. The Design of AutoMice

In this section, we present the architecture of AutoMice and its design principle. Also, we present several system modules and applications that are challenging to be tested in the wild but can be easily tested by AutoMice.

A. AutoMice Overview

A self-driving system consists of several modules that perform various tasks. The input of a self-driving system is from the sensors installed on the vehicle, e.g., Lidar, cameras, etc. The output of a self-driving system is the control command sent to the vehicle through the control module. The testing and validation of self-driving system modules relies on simulation [4] and real road test [1], [3]. As discussed in section II, both simulation and road test have their limitations, and AutoMice is designed to be a better alternative to test self-driving algorithms.

A high-level architecture of AutoMice is illustrated in Fig. 1. AutoMice is designed to encapsulate the input and output while the core system modules can be reused between the testbed and a real self-driving car. Since the self-driving system is time sensitive, it is usually implemented in C/C++. So, we assume the APIs used by the self-driving core system and data module are written with C/C++. From the perspective of developers, there is no difference to write code for AutoMice or a self-driving vehicle. AutoMice can be used in two scenarios, a physical city road model that a toy car can drive with AutoMice system installed, or a real car to test whether the system control decision is the same with a human driver control decision.

B. Design Principle

1) Abstraction: There are two abstraction layers on the input and output sides. On the input side, the abstraction layer emulates the production environments. In our implementation, we used an Android phone as the self-driving platform and the self-driving system is implemented as a native application, which we will discuss more in section IV. The camera and sensor data of Android phone are captured by an Android app written in Java. We then use Java Native Interface (JNI) to convert the data into the format that can be used by the core system modules. The JNI is a programming framework that enables Java code running in a Java Virtual Machine (JVM) to call native applications and libraries written in other languages such as C, C++, and assembly. The image captured from Android phone is converted into Mat format used in OpenCV. The sensor data collected from the Android phone and the speed data collected from the vehicle hall effect sensor are abstracted into a Trace data format, which consists of a timestamp field and corresponding values.

On the output side, the abstraction layer converts the steering angle and acceleration into the command format used by the testbed car. No matter how complex the underlying core system modules are, the output of a self-driving system is very simple, i.e., the steering wheel angle and the vehicular speed. However, the exact command of such two parameters could vary from platform to platform. The abstraction layer is used to provide a compatible data format and linear translation between the control module and the vehicle CAN bus.

2) Two-Phase Design: The development and deployment on AutoMice is divided into two phases. In the development phase, the developers focus on the development of the core system and data modules. The core system modules are implemented in C/C++ and can be implemented by using standard compiler and development environment, i.e., C++ on Eclipse. Such core system and data modules can be used in either the testbed or a self-driving production. In the deployment phase, the developers only have to ship the code into the testbed and observe the reaction of the vehicle testbed. For example, to test if the car can stop at red traffic light, we use a toy infrastructure to turn on a red light to see whether the car can successfully stop.

3) APIs: AutoMice provides three set of APIs to interact with sensors and vehicles. The APIs are shown in Table. I. The first set of APIs is used for perceptions. It reads the data from the image and/or Lidar to understand the surrounding environments. In our implementation, we use only one front facing camera, but we generalize our APIs to handle all possible directions as well as other sensor types, such as Lidar. The second set of APIs is used to control the vehicle. To be compatible with real cars, we use three interfaces to control the acceleration, brake and steering. The third set of APIs is used to process the data from other vehicles or infrastructures.

C. Core System and Data Module Testing

We present how we can use AutoMice to make the testing of self-driving car easier in terms of several system modules.

1) Object Detection: Object detection can be used at the 3D map construction to label traffic signs and lights as well as the system running time to identify moving vehicles. Given the known location of traffic lights, the self-driving car can identify them with much higher accuracy and efficiency. After the 3D map is built, the self-driving car captures real-time Lidar and camera data to understand the semantics of surrounding objects. The localization module takes the real-time data to localize the vehicle within the 3D map (and the real road network). The perception module is also responsible for identifying moving objects, such as vehicles, pedestrians, etc. The planning module consists of a long-term planning logic and a short-term planning logic. The long-term planning logic calculates the route to the destination, which is similar to existing navigation applications such as Google Map. The short-term planning logic takes traffic condition into consideration and navigates the vehicle around obstacles to follow traffic rules and avoid possible collisions. The control module then sends the steering wheel angle and acceleration/brake commands to the CAN bus to control the vehicle. AutoMice provides a low-cost way to test perception and link it with corresponding actions a vehicle would take.

2) Remote Control: Like many other computer systems, we have to define the system behaviors of self-driving cars to deal with various challenging situations, i.e., road construction, bad weathers, etc. If we failed to define the corresponding system behaviors, the reaction of a self-driving car is then undefined and possible danger could happen. In order to handle such corner cases, remote assistance has been proposed recently [6], [10] to augment self-driving systems. It raises another set of questions about how much a remote operator should be involved with the decision making of the self-driving cars. One design option is the operator has a complete control of the vehicle. Another design option is the self-driving car still drives itself and the human operator only provides trajectory suggestions. There could be another design option that the self-driving system provides a minimum collision avoidance functionalities, while the remote human operator controls the car under such constraints. There are also open questions about what kind of data should be transmitted according to the three different design options [6].

Testing various design options by using real self-driving cars is difficult. Firstly, the LTE network performance in the wild is uncontrollable and it is difficult to test the remote control system under various network conditions without guaranteed quality of service. Secondly, testing in the wild is labor-intensive and requires many human hours and road miles. AutoMice provides a low-cost and easy way to evaluate remote control algorithms under controllable wireless network connections before testing them in the wild.

TABLE I: APIs

Signature
// local data processing
void processLocalImage(const Mat& img, const Point& direction);
void processLocalLidar(const vector<vector<bool>>& points, const Point& direction);
// vehicle controls
void steering(double ste);
void acceleration(double acce);
void brake(double brk);
// remote data processing for V2V and V2I communications
void processRemoteImage(const Point& location, const Mat& img, const Point& direction);
void processRemoteLidar(const Point& location, const vector<vector<Point>>& points, const Point& direction);
void vehicleMovement(const Point& location, const Point& next);

3) Vehicle-to-Vehicle (V2V) and Vehicle-to-Infrastructure (V2I) Communication: Even though we can put a lot of different sensors in a car, the abilities of the sensors to observe the outside world are limited in some scenarios because the sensors are installed in the car. For example, the front view camera of the car will not be able to capture a pedestrian when she/he is behind a tree, or around the street corner. Enabling V2V and V2I communication can help to overcome such limitations of in-car sensors. A car can query sensor data from other nearby vehicles or from the infrastructure to get an exhaustive understanding of the environment, and thereafter, to make reasonable driving decisions. Swarun et al. built CarSpeak, a content-centric communication system for self-driving vehicles, enabling them to query and access data captured by sensors in other cars in a manner similar to how they access information from their local sensors [11].

Evaluating such wireless communication protocols is challenging in real road environments, especially when moving objects exist. Those moving objects generate randomness in the evaluation which is hard to reproduce. Considering that the system behavior is not as expected, we may not able to recreate the same running environment to figure out what happened. Therefore, testing in controlled environments is very important. AutoMice can simulate V2V and V2I communications in a controllable environments. Imagine we can set up multiple self-driving cars to drive from the randomly selected start and end points and observe how they react by using various communication protocols. Such setup provides low-cost and repeatable testing environments and can improve the evaluation efficiency for new communication protocols.

4) 3D Map Construction: One important input required by self-driving system is a 3D map [12], [13]. Such a map includes all the static details of the road, i.e., the road, lane markers, position of the traffic light, etc. The 3D map can be built by using Lidar and Cameras. The output of Lidar is a 3D point cloud and each point represents a surface point that reflected to the Lidar. The advantage of Lidar is the depth information it provides, i.e., even the object is far away from the Lidar, the cloud points of the object can still be projected. The camera is not able to provide such rich depth information, but it provides colorful pixels for various objects. The fusion of Lidar and camera can provide better accuracy to identify objects and predict the corresponding behaviors. Manual labeling is required sometimes for the traffic signs, traffic lights, and corresponding semantic meanings. It is still an open problem to update the 3D map online. For example, when the road is under construction and one lane is closed, how to update the 3D map and broadcast this update so that all the self-driving systems can be synchronized.

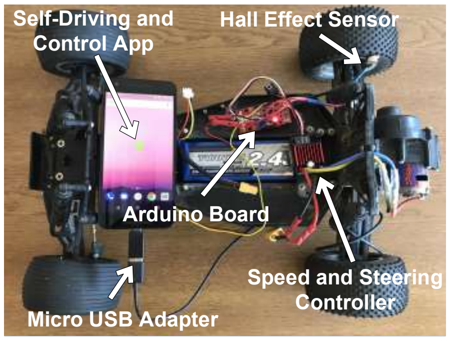

Fig. 2: The self-driving testbed.

Evaluating a 3D map construction engine is very difficult. Firstly, we have to manually measure the ground truth data, the width of the road, the length of the lane makers, the height of the traffic light, etc., to build a ground truth 3D map. After we construct the 3D map by using the sensor inputs, we then compare the constructed 3D map with the ground truth. Evaluating 3D map construction algorithm by using AutoMice would be easier since the ground truth street and buildings are pre-measured.

IV. Implementation

AutoMice consists of an Android-powered self-driving vehicle testbed and a server for the self-driving system development and remote control. The customized self-driving vehicle is illustrated in Fig. 5. We implement a Native Android Application as the self-driving system software. A native application is a mix of Java and C/C++ code. The core self-driving system modules are written in C/C++, while the abstraction layers are implemented in Java and Java Native Interface (JNI). The application is developed in a way that the self-driving developers can focus on core algorithm development and ship to Android without any modification.

We choose Android as the self-driving testbed platform for two reasons. First, it has a rich set of sensors such as Camera and IMU sensors. It is also able to connect with third-party hardware such as an Arduino board to get extra inputs. Second, it relies on battery and we can easily mount it on the real car for outdoor testing. We believe AutoMice can also be implemented on other platforms with a similar design.

A. Self-driving Vehicle Testbed

In the Android-powered self-driving system, the app transfers the control message to the Arduino board through a Micro USB cable. A Hall Effect Sensor is installed to track vehicular speed by recording the number of wheel rotations per second. The Arduino board bridges the Android app and the hardware, i.e., the steering controller and hall effect sensor. The abstraction layer of the app sends the captured images and sensor data into the self-driving system modules through Java Native Interface (JNI). There are totally 2700+ lines of Java code for the Android app implementation as the abstraction layers. The abstraction layer also includes a communication module that can communicate with other vehicles or infrastructure with UDP socket. The communication module is implemented as a Android service. The abstraction layer uses a serial communication library to communicate with the Arduino board. The Arduino board sends the analog signal to control the car and reads the speed from the Hall sensor. The camera data is collected from the Android built-in camera library and interface. We also collect the IMU sensor data from an Android service. The minimum Android SDK version is 21, which is the minimum requirement of OpenCV3.2 SDK for the Android platform. The Android phone we use to build the prototype is Nexus 5x. It has a reversed camera sensor and we have to rotate the camera by 180 degree programmatically.

B. Development and Remote Control Server

The self-driving system development server is implemented in C++. It is used for self-driving system development as well as the remote control. It is built with GStreamer and OpenCV for various streaming and image process modules. As the remote control server, it uses one UDP thread to receive the video stream from the self-driving vehicle. The UDP thread sends the compressed video data to a GStreamer pipeline [14] running in another thread. The GStreamer pipeline decompresses, displays and records the video. There is another thread receives control messages from a customized controller to control the vehicle by the human operator. Our implementation of various algorithms and control server consists of 4500+ lines of C/C++ code. The remote server is implemented on Ubuntu 16.04. The GStreamer version we use is 1.8.3.

V. Evaluation

To illustrate the usability of AutoMice, we implement several self-driving system modules and demonstrate the usability of AutoMice in testing these modules.

A. Object Identification

One of the methods for a self-driving vehicle to perceive the environment is object detection, which takes in an image at a time and outputs the coordinates of a specific class of objects found within the image. Object tracking consists of continuous object detection and trajectory estimation. In object detection, it is desired that the time required to process one image be as short as possible so that more images can be processed within some period of time. Another metric of object detection is accuracy.

Since we use an Android phone as our self-driving system testbed, we can also mount it on a real car to test the accuracy of the algorithms. We demonstrate its capability to identify

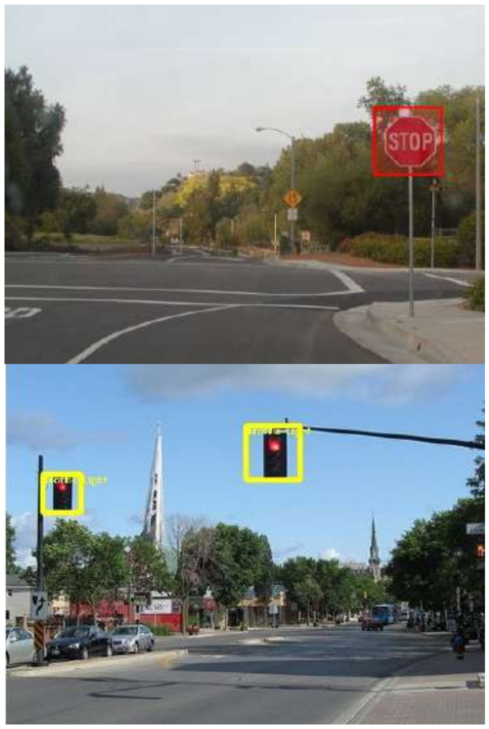
Fig. 3: Static traffic light/sign identification for 3D map construction and perception.

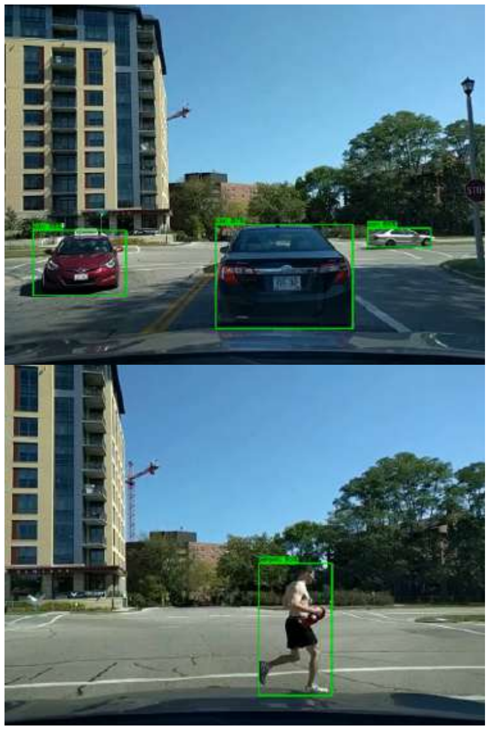
Fig. 4: Moving object identification for perception.

static and moving objects in Fig. 3 and Fig. 4, respectively. Static objects are mainly used to label the semantics of the constructed 3D map. For example, we can label the stop signs and traffic lights for each road segment at the 3D map construction time, the self-driving car can then search particular spots for possible traffic instructions at the driving time. Meanwhile, a self-driving car should also be able to identify moving objects such as cars and pedestrians. Two examples are shown in Fig. 4. By tracking the movement of such objects, the self-driving car can make correct decisions to avoid possible collisions and accidents.

To illustrate that our testbed is compatible with different object detection approaches, we implement three approaches, Mean Square Error (MSE), Cascade Classification, and SSD with MobileNet [15], to detect the objects of interest. We use stop sign detection as an example to illustrate how these three approaches work.

MSE uses a single cropped stop sign as the data model, and compare it with all the submatrix of a given image that possibly contains stop signs. To iterate over all possible submatrix of the given image, it starts the search process from upper left to bottom right and compares the pixel difference one by one. If the overall difference is smaller than a threshold, then a stop sign is detected. The detection accuracy of this approach is very sensitive to the detection threshold, i.e., a red car could be detected as a stop sign due to the color similarity.

Haar Cascades [16], [17] is a popular low-cost object detection method. It uses extracted features to train a Cascade model and then use that model to detect whether an image contains stop signs.

Another popular approach is to use deep learning techniques. In our evaluation, we use Single Shot MultiBox Detector (SSD) with MobileNet to detect objects. It is a pre-trained convolutional neural network model based on TensorFlow [18]. A convolutional neural network (CNN) consists of a number of layers for different purposes. A convolution layer finds the degree to which each area of an image matches a pattern. We input an image to a convolution layer as a matrix. A filter, which is a smaller matrix representing a pattern, is slid through the image matrix, taking the dot product of the pattern matrix and the block of the image matrix covered. The dot products form the output matrix. A max-pooling layer shrinks a matrix by, dividing a matrix into blocks and form a new matrix with the maximum value of each block. After going through a stack of these layers, the output matrices are flattened into one vector, which is a fully connected layer. Each entry of the vector has a weight for a certain pattern. By comparing the weighted sum of the vector with different weights for different patterns, the most likely pattern is chosen. The combination of layers is significant to the performance of a CNN model.

Single Shot MultiBox Detector (SSD) is a deep convolutional neural network for object detection featuring competitive accuracy and time performance in its kind [19]. Combined with the MobileNet feature extractor [20], built by Huang et al., the network is said to achieve highest speed [15], which fits our purpose well, given the limited computing resources on a self-driving vehicle.

TABLE II: Object Detection

Metric	MSE	Haar Cascade	Deep Learning
Training Set	small	medium	large
Running Time	1-2s	150ms	80ms
Accuracy	70%	90%	95%

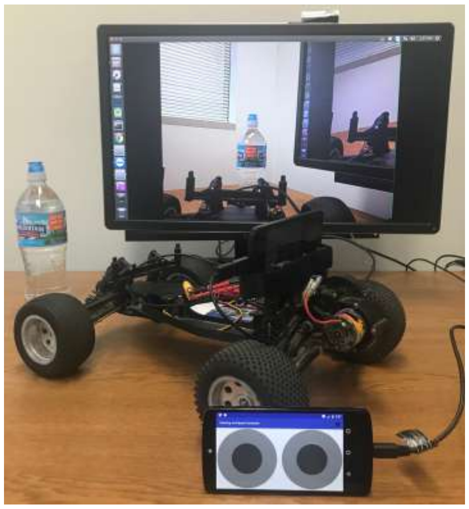
Fig. 5: Streaming the front view of the self-driving vehicle testbed to the server, so that the operator can use the controller to control the car remotely.

We test these approaches on traffic signs and cars, which are common objects in the environment around a self-driving vehicle. We use part of the LISA Traffic Sign Dataset [21] for the stop sign detection and frames of a self-recorded video for car detection.

The benchmark of various detection method is shown in Table II. MSE requires small training set but the running time is much higher than feature-based and deep learning detectors. Deep learning model provides the most accurate result but it depends on the large training set. Those approaches can be implemented in the development phase and the code can be easily migrated into AutoMice and a real self-driving car to benchmark and evaluate their performance.

1) Remote Control: To illustrate how AutoMice can be used to evaluate remote control system for self-driving cars, we implement a live streaming module on both the self-driving car and the remote server.

To reduce the size of the video, we use the standard video coding algorithm (H.264) to compress video frames. Each frame can be encoded into either I-frame or P-frame. I-frame refers to the intra-coded frame, where the frame is only compressed with intra-frame prediction. Each I-frame can be recovered into the original image without dependencies on other frames. P-frame refers to the predicted frame, which means the frame uses inter-frame prediction. A single P-frame is not able to be recovered into the original image. According to our benchmark, the video compression ratio is around 5%. We use the MediaCodec library of Android platform to encode the video and use GStreamer on the remote control server to decode the video.

A demonstration is shown in Fig. 5. In this demonstration,

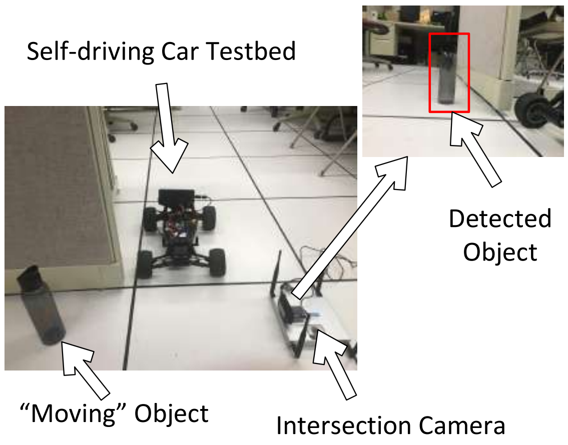

Fig. 6: Self-driving car and intersection camera communication.

the front view camera of the car is streaming to the remote server. A human operator can see the remote view and control the car remotely by using the controller. Given this setup, we can evaluate the remote control system in several aspects. Firstly, we can evaluate remote control experience by varying the network latency to see how the control experience varies. Also, we can test how the live streaming protocol reacts when we change the wireless network bandwidth and loss rate. It gives us more flexibility than system evaluation under LTE network in the wild.

2) V2V and V2I Communication: To illustrate that AutoMice can be used for the evaluation of V2V and V2I communication, we set up a test case in our lab. As shown in Fig. 6, the self-driving car is driving top down in an intersection and there is an intersection camera to cover the road that is hidden by the "building". We use a bottle to represent the object "coming" from that direction. When there is an object detected in that direction, the self-driving car should stop and wait. If there is no object detected by the intersection camera, the self-driving car can safely drive through. This setup can be used to evaluate V2V communication if we replace the object with another self-driving car.

In our implementation, we use a smart Wi-Fi router running Ubuntu to act as the infrastructure. The camera is connected with the smart router and the router fetches the images and performs object identification in one thread. There is another thread running on the router to broadcast possible detect objects. The UDP service running on the self-driving car is listening broadcasts from surrounding vehicles and infrastructures. Our simple setup can be extended into the complex road and traffic models, with which developers are able to evaluate the performance of various the V2V and V2I communication protocols.

B. 3D Map Construction and Localization

The construction of a 3D map of the real world requires a process of 3D reconstruction, which creates a solid three-dimensional model from multiple two-dimensional images. It uses Simultaneous Localization and Mapping (SLAM) techniques to restore the 3D scenario by using multiple 2D images.

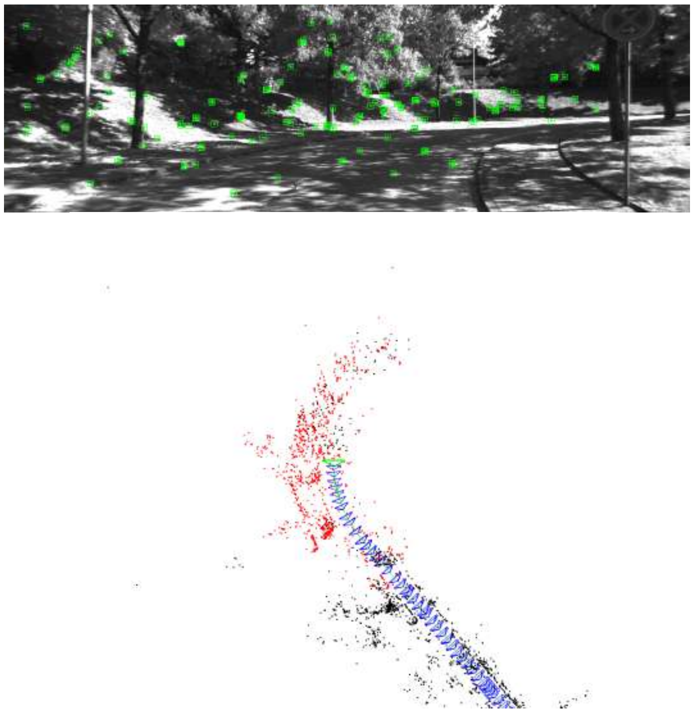

Fig. 7: 3D Map Construction and Localization.

With two or more continuous images, we can create a point cloud or a depth map that outlines the real world environment. By further conversion and surface reconstruction, a solid 3D model can be created.

One of the state-of-the-art approaches of 3D mapping using a monocular vision (as opposed to the RGB-D camera, which provides depth information, and the stereo camera that uses two separate image sensors) is proposed by ORB-SLAM [22]. It uses a feature-based method, to choose only the points that match a vocabulary as feature points [23] to be tracked. By tracking the features in the intersections of the images, it is possible to find the relative 3D positions of those points. This process is called triangulation. By applying such a process to continuous image frames, the tracked points form a point cloud, which depicts the outline of the real world environment. Meanwhile, an image alignment-based tracking is proposed by Engel et al. [24] to generates a semi-dense depth map.

It is known that a semi-dense depth map can be either densified or converted into a point cloud [25]. A large number of algorithms have been proposed to reconstruct 3D surfaces out of a point cloud [26]. However, performing localization itself does not need a solid 3D model of the environment [24], [22], therefore surface reconstruction can be omitted.

In our evaluation, we use continuous 2D image sensing to reconstruct the road as the first step of 3D map construction. One of the challenges is real-time localization by using cameras, because GPS is not able to provide highly accurate results. SLAM uses a dictionary of features to extract all the features in the image and compare the feature shifts of continuous images to conduct localization and map construction. We use the dataset from [27]. As shown in Fig. 7, the top figure is the real-time feature extraction from a 2D image, the bottom figure is the route that is under construction. The red points are the features identified in the current image, while the black points are the features extracted in the past. Running SLAM in a real road is costly and difficult to verify the accuracy. In an indoor environment, we can set up the infrastructure and compare the results with the setup easily, while the code running in either indoor or outdoor environments are the same.

VI. Conclusions

Self-driving systems are aiming for a safer traffic environment. However, the validation of self-driving algorithms is very challenging. Two methods are widely used in the community: simulation (low cost but difficult to simulate all the detailed physical interactions) and road test (reliable but risky). To remedy these issues, we present AutoMice, a testbed infrastructure that offers developers an environment to experiment with self-driving algorithms. AutoMice is designed to ease the transition from testbed validation to deployment in production by using two abstraction layers on both the input and output of a self-driving system. It uses a two-phase design and consists of a development phase and a deployment phase. In the development phase, the developers can focus on the core self-driving system modules. In the deployment phase, the developers can simply ship the code to the testbed and observe the reaction of the car. By ensuring the abstraction layers are compatible with a real self-driving system, the same core system code can be used in self-driving production. To demonstrate the usability of AutoMice, we implement several self-driving perception and control algorithms (traffic light detection etc.) in the development phase that can be reused in the deployment phase.

References

[1] Waymo, "On the road to fully self-driving, waymo safety report," 2016.

[2] AutoX, "Camera-first ai brings self-driving cars out of the lab and into the real world," https://www.autox.ai/, 2016.

[3] C. Automation, https://getcruise.com/.

[4] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "Carla: An open urban driving simulator," in *Conference on Robot Learning*, 2017.

[5] M. Muller, V. Casser, J. Lahoud, N. Smith, and B. Ghanem, "Sim4cv: A photo-realistic simulator for computer vision applications," *International Journal of Computer Vision*, pp. 1–18, 2018.

[6] L. Kang, W. Zhao, B. Qi, and S. Banerjee, "Augmenting self-driving with remote control: Challenges and directions," in *HotMobile*. ACM, 2018.

[7] X. Chen, H. Ma, J. Wan, B. Li, and T. Xia, "Multi-view 3d object detection network for autonomous driving," in *IEEE CVPR*, 2017.

[8] J. Leonard, J. How, S. Teller, M. Berger, S. Campbell, G. Fiore, L. Fletcher, E. Frazzoli, A. Huang, S. Karaman *et al.*, "A perception-driven autonomous urban vehicle," *Journal of Field Robotics*, 2008.

[9] M. Naumann, F. Poggenhans, M. Lauer, and C. Stiller, "Coincarsim: An open-source simulation framework for cooperatively interacting automobiles."

[10] T. for Autonomous Vehicles, "https://phantom.auto/."

[11] S. Kumar, L. Shi, N. Ahmed, S. Gil, D. Katabi, and D. Rus, "Carspeak: a content-centric network for autonomous driving," in *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. ACM, 2012, pp. 259–270.

[12] R. W. Wolcott and R. M. Eustice, "Visual localization within lidar maps for automated urban driving," in *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*. IEEE, 2014, pp. 176–183.

[13] H. G. Seif and X. Hu, "Autonomous driving in the icityhd maps as a key challenge of the automotive industry," *Engineering*, 2016.

[14] GStreamer, https://gstreamer.freedesktop.org/.

[15] J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara, A. Fathi, I. Fischer, Z. Wojna, Y. Song, S. Guadarrama, and K. Murphy, "Speed/accuracy trade-offs for modern convolutional object detectors," in *IEEE CVPR*, 2017.

[16] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[17] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, vol. 1. IEEE, 2001, pp. I–I.

[18] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[19] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *European Conference on Computer Vision*, 2017.

[20] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," 2017.

[21] A. Mgelmose, M. M. Trivedi, and T. B. Moeslund, "Vision based traffic sign detection and analysis for intelligent driver assistance systems: Perspectives and survey," in *IEEE Transactions on Intelligent Transportation Systems*, 2012.

[22] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardos, "Orb-slam: a versatile and accurate monocular slam system," *IEEE Transactions on Robotics*, vol. 31, no. 5, pp. 1147–1163, 2015.

[23] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "Orb: An efficient alternative to sift or surf," in *Computer Vision (ICCV), 2011 IEEE international conference on*. IEEE, 2011, pp. 2564–2571.

[24] J. Engel, J. Sturm, and D. Cremers, "Semi-dense visual odometry for a monocular camera," in *Proceedings of the IEEE international conference on computer vision*, 2013, pp. 1449–1456.

[25] D. Baricevic, T. Höllerer, and M. Turk, "Densification of semi-dense reconstructions for novel view generation of live scenes," in *Applications of Computer Vision (WACV), 2017 IEEE Winter Conference on*. IEEE, 2017, pp. 842–851.

[26] M. Berger, A. Tagliasacchi, L. M. Seversky, P. Alliez, G. Guennebaud, J. A. Levine, A. Sharf, and C. T. Silva, "A survey of surface reconstruction from point clouds," in *Computer Graphics Forum*, vol. 36, no. 1. Wiley Online Library, 2017, pp. 301–329.

[27] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.