2020 NDIA GROUND VEHICLE SYSTEMS ENGINEERING AND TECHNOLOGY SYMPOSIUM
MODELING SIMULATION & SOFTWARE (MS2) TECHNICAL SESSION
AUGUST 11-13, 2020 - NOVI, MICHIGAN

MISSISSIPPI STATE UNIVERSITY AUTONOMOUS VEHICLE SIMULATION LIBRARY

Christopher R. Hudson[1], Christopher Goodin[1], Zach Miller[1], Warren Wheeler[1], Daniel W. Carruth[1]

[1]Center for Advanced Vehicular Systems, Mississippi State University, MS

ABSTRACT

Simulation is a critical step in the development of autonomous systems. This paper outlines the development and use of a dynamically linked library for the Mississippi State University Autonomous Vehicle Simulator (MAVS). The MAVS is a library of simulation tools designed to allow for real-time, high performance, ray traced simulation capabilities for off-road autonomous vehicles. It includes features such as automated off-road terrain generation, automatic data labeling for camera and LIDAR, and swappable vehicle dynamics models. Many machine learning tools today leverage Python for development. To use these tools and provide an easy to use interface, Python bindings were developed for the MAVS. The need for these bindings and their implementation is described.

Citation: C. Hudson, C. Goodin, Z. Miller, W. Wheeler, D. Carruth, "Mississippi State University Autonomous Vehicle Simulation Library", In *Proceedings of the Ground Vehicle Systems Engineering and Technology Symposium* (GVSETS), NDIA, Novi, MI, Aug. 11-13, 2020.

1. INTRODUCTION

Simulation is often regarded as critical to the development of autonomous systems in part due the ability to precisely control the environment in which the simulated vehicle is operating. This ability to control the environment is critical to many aspects of autonomy development, including safety and repeatability. Simulation can also aid in the early assessment of autonomy algorithms; if an autonomous vehicle is unable to perform a task in simulation, it is unlikely to capably perform the task in the real world. Being able to control the complexity of the environment allows for faster development of algorithms to account for a variety of different world conditions. Simulators are abundant and provide a variety of functionality with various validities. Simulations are often designed with specific purposes in mind. There are many components to a simulator that must be considered, and each of these components will have the minimum level of component fidelity necessary to satisfy the intent of the simulator. Chief among these is the type of environment the simulation is designed to operate in, since that will determine the types of simulations that can be run within the software. In general, simulators tend to focus on either on-road applications or off-road applications.

1.1. On-road Simulators

On-road simulators are often designed in-house and are used to test specific pieces of hardware for a company. There is a growing market for on-road autonomous vehicles, as on-road autonomy is seen as the next big innovation for commercial vehicles. As a result of this increased demand for self-driving capabilities, major companies have developed simulation tools focused on on-road autonomy development. It is often reported how many millions of miles Google or Tesla have driven their vehicles in simulation prior to the release of an update to their autonomous platforms [6]. These types of simulators are used to test a variety of algorithms running on a vehicle. Most autonomous vehicles operate several different types of algorithms concurrently to understand the environment they are operating in. All these algorithms must be tested in a measurable way to provide the consumer with some degree of confidence in the ability of the vehicle to follow the designated path safely and accurately.

1.2. Off-Road Simulators

Off-road simulators have not received the same amount of attention as on-road simulators, likely due to the perceived size of the market for off-road vehicle simulation. Only a fraction of a percent of the earth's surface is paved road. This fraction of a percentage however represents significant potential revenue for companies that can solve the on-road self-driving problem first. Off-road environments present a more disjoint set of needs when compared to on-road environments. The operational guidelines for off-road vary widely based on the type of vehicle being operated. Solving off-road autonomy is more difficult than on-road autonomy. This difficulty stems from a variety of different interactions that need to be modeled, including vehicle terrain interaction, soft soil modeling, sensor interaction with complex and dense environments, and complex surface materials.

This added difficulty compounds the number of components that must be accounted for in a simulation platform designed for an off-road environment. The major potential applications of off-road autonomy are military and agriculture. Each of these industries has specific and often conflicting requirements for an off-road autonomous system. Farming vehicles often operate on large, relatively flat terrains while moving in slow but precise ways down rows of crops. These types of autonomous systems do not have to focus significant efforts on being able to operate at high speeds. Military systems, however, have very different needs than farming systems. Military systems can be a variety of sizes and require a range of operational speeds based on specific circumstance. Military systems will not always be operating on flat terrain, they require the ability to operate anywhere: Hills, Forest, Swamp, Fields, Desert, etc.

1.3. Component Design

Every simulator will have a set of components available to the user. These components include sensors, vehicles, environmental objects, and controllers. These components will have a variety of fidelity levels based on the design needs of the simulator. There are two basic types of designs: those that focus on looking realistic at face value, and those that focus on a highly accurate physics model. Many simulators in use today leverage game-engines such as Unreal Engine 4 (UE4) and Unity. These game-engines provide excellent graphics capabilities, but often lack high-fidelity physics models, particularly for sensors. These properties are often mutually exclusive when considering real-time or faster than real-time simulators. Many developers choose to use game engines due to their availability and accessibility. Game engines offer a low barrier to entry, and often come with pre-defined systems

which allow developers to quickly produce simulation environments that aesthetically look amazing and run quickly. These simulations, however, often lack the in-depth physics components that can be critical to simulating component level systems such as vehicle terrain interaction, or accurate lidar sensor readings.

1.4. Functional Use

Every simulator is designed for a set of functional uses. Some simulators such as the one use by the Google smart car, are simply designed to run the machine learning algorithms that run on the vehicle's hardware, on a virtual model of the vehicle as it drives around. Other types of simulators can be leveraged to generate synthetic environments to train machine learning algorithms during their development phase.

2. Mississippi State University Autonomous Vehicle Simulator (MAVS)

The Mississippi State University Autonomous Vehicle Simulator (MAVS) was designed as an off-road simulator. MAVS was originally designed to enable high fidelity sensor simulation in unstructured environments with a wide variety of environments. MAVS enables the creation of synthetic data sets and testing environments for off-road vehicles in an accurate physics-based environment. MAVS can be leveraged to train both autonomous systems, as well as intelligent systems using high-fidelity sensor data. It leverages ray-tracing to accurately model sensors and light propagation throughout the environment [1][2]. MAVS was designed as a library of simulation tools to allow for real-time, high performance, ray traced simulation capabilities for off-road autonomous vehicles. MAVS functionality is geared towards off-road vehicles, and as a simulation library, it provides several tools for several different tasks such as data labeling, path planning, lidar segmentation, and motion planning. These tools include automated off-road terrain generation, automatic data labeling for both camera (Figure 1) and lidar data (Figure 2) and swappable vehicle dynamic models.

2.1. Automated Data Labeling

MAVS allows for automated data labeling of images and LIDAR scans generated during a simulation [4][5]. Each camera can generate a pixel-accurate segmented image based on the classes the user defines. Within the configuration, specific objects can be tagged with their classification. When each image is generated, the ray tracer produces a ray at each pixel in and checks the tag for the object it collides with to generate a corresponding pixel accurate segmentation map [4]. Segmented image and lidar data can be leveraged by machine learning algorithms in order to classify what each sensors see's within the environment. By generating both a labeled and unlabeled image, MAVS can provide users who are developing machine learning algorithms with both accurate ground truth training data and test data. This feature, when combined with the automated scene generation tools, allows for the rapid development of an extremely large set of randomized scenes, that share common ecosystem traits. These random scenes can test algorithms, with a ground truth reference, without needing to be hand labeled by a person.

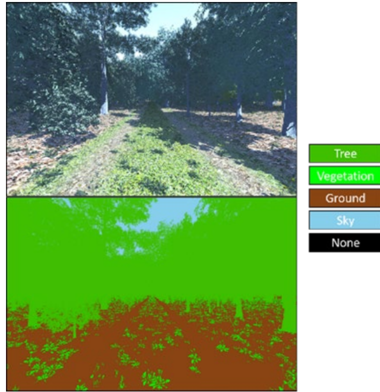


FIGURE 1. Automatically Labeled Camera Data

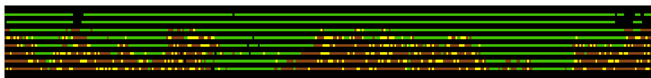
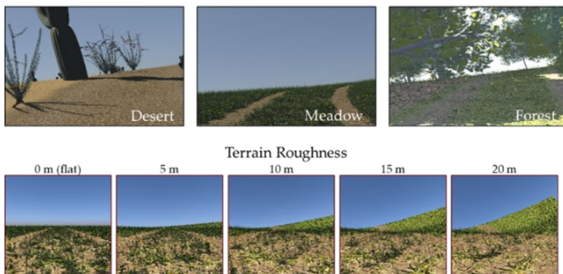


FIGURE 2. Automatically Labeled LIDAR Data

2.2. Terrain Generation

The automated off-road terrain generation allows for procedurally generated scenes based on terrain roughness, environment type (forest, desert, meadow), environment conditions (sunny, night, haze, snow, fog, dust, rain), and lighting conditions (time of day) (Figure 3). This functionality allows for any number of uniquely simulated environments for repeated testing and/or synthetic data generation.



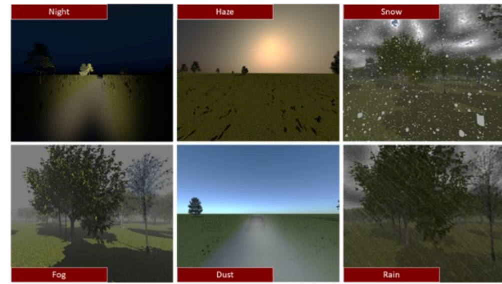


FIGURE 3. Environment properties

2.3. Simulation

MAVS allows for both desktop and High-Performance Computing (HPC) simulation. Validation efforts for MAVS are ongoing. Because autonomous vehicles consist of a complex system of sensors, mechanical components, and software, simulators like MAVS must be validated at both a component level (e.g. lidar model, vehicle model) as well as a system level (e.g. simulated obstacle avoidance test). By using a comprehensive, multi-level validation approach, aspects of MAVS such as the lidar simulation, vehicle-terrain interaction, and overall simulation capability are being validated and documented to provide confidence in the results of simulated studies conducted with MAVS.

Current validation efforts for the vehicle-terrain interaction model focus on comparison to historical assessments of wheeled vehicle mobility in soft soil. These include slope-climbing simulations on sand as well as soft-soil traversability simulations in clay. Additional validation for the lidar model has been conducted by comparing MAVS simulations to published experiments on lidar “mixed-pixel” effects, as well as measurements of the influence of rain on lidar. Ongoing validation experiments measuring the influence of rain droplets on camera lenses are also being conducted. Finally, system-level validation tests for obstacle

detection and avoidance are currently being planned.

MAVS has been used for a variety of applications to date. These applications include leader-follower simulation using MAVS and ANVEL, measuring error propagation, as a teaching aide for a graduate-level course on AI, for automated labeling for synthetic environments [7], and for training machine learning algorithms with synthetic data [4].

3. MAVS Python Interface

While MAVS provided a large range of functionality in a library form, the library was not easily leveraged by systems not written in C++. This dependence on a single language, without the ability to connect to other application can severely limit the usability of a simulation library. To resolve this outward compatibility issue, a DLL was compiled from the library, and Python bindings were written using CTypes to allow for the simulation library to be leveraged in any Python script. The deployment of a DLL with Python bindings allows for the rapid adoption and usage of MAVS for a wide variety of tasks by lowering the barrier to entry into these complex simulations. The MAVS architecture allows for the MAVS library to easily be leveraged within Python.

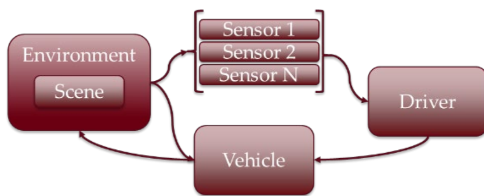


FIGURE 4. MAVS Architecture

MAVS defines four different components: the environment, sensors, vehicle, and driver. The environment component describes the scene which the vehicle and sensors will operate. This includes all objects within the scene such as trees, ground clutter, buildings, etc. A vehicle is defined to operate within the environment. This vehicle is an abstraction of the vehicle dynamics, which can leverage the built-in MAVS vehicle dynamics model, Chrono, or ANVEL. Each vehicle has any desired number of sensors. These sensors can include any of the pre-defined MAVS sensors including camera, GPS, IMU, lidar, and radar. The last component needed is the driver. The driver is any control method the user wishes to use for the vehicle. This can be a keyboard control, or any autonomy algorithms through the ROS interface.

The MAVS library is designed to work either as a stand-alone application or in conjunction with other simulators acting as a "driver" component.

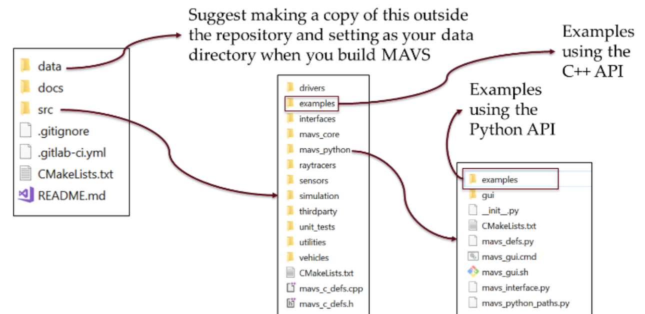


FIGURE 5. MAVS File Structure

Once MAVS is built and installed it can be imported into any Python script.

```
import sys
# Set the path to the mavs python api, mavs_interface.py
sys.path.append(r'C:/Users/cgoodin/Desktop/vm_shared/shared_repos/mavs/src/mavs_python')
# Load the mavs python modules
import mavs_interface
import mavs_python_paths
# Set the path to the mavs data folder
mavs_data_path = mavs_python_paths.mavs_data_path
```

FIGURE 6. MAVS import

Once MAVS has been loaded into a Python script, each component within the architecture needs to be defined, starting with the environment and its associated scene.

```
# Select a scene and load it
mavs_scenefile = "/scenes/cube_scene.json"
scene = mavs_interface.MavsEmbreeScene()
scene.Load(mavs_data_path+mavs_scenefile)
```

FIGURE 7. Loading a Scene

When a scene is loaded, it contains a description of the geometry and objects within that scene, including the surface information. However, one of the benefits of MAVS is the random scene generation tool. If the user desires, a scene can be randomly generated within a specific ecosystem instead.

```
random_scene = mavs.MavsRandomScene()
# terrain width and length in meters
random_scene.terrain_width = 100.0
random_scene.terrain_length = 100.0
# magnitude of low frequency roughness, in meters (rolling hills), 0=flat
random_scene.lo_mag = 2.0
# magnitude of hi-freq roughness, [meters] (bumpy terrain), 0 = smooth
random_scene.hi_mag = 0.1
# plant density should be from [0-1]. 0=no plants, 1=most plants
random_scene.plant_density = 0.05
# trail properties
random_scene.trail_width = 2.0
random_scene.track_width = 0.3
random_scene.wheelbase = 1.8
# pothole depth and diameter in meters
random_scene.pothole_depth = 0.5
random_scene.pothole_diameter = 1.0
# Tell mavs exactly where you want the pothole
random_scene.AddPotholeAt(5.0,5.0)
# Next few lines create the scene and save it to disk
scene_name = 'pothole_scene'
random_scene.basename = scene_name
random_scene.eco_file = 'american_pine_forest.json'
random_scene.path_type = 'Ridges'
random_scene.CreateScene()
# Load the scene we just created
scene = mavs.MavsEmbreeScene()
scene.Load(scene_name+'_scene.json')
```

FIGURE 8. Defining a random scene

Each ecosystem file contains a list of objects (trees, plants, and other vegetation) that would be present within the ecosystem being generated. This ensures that the scene generated have the appropriate objects for the area they represent. However, users can define their own list for ecosystem files if the provided ones do not satisfy their needs.

```
# Create a MAVS environment and add the scene to it
env = mavs_interface.MavsEnvironment()
env.SetScene(scene.scene)
# Set whatever environment properties
env.SetTime(13) # 0-23
env.SetFog(0.03) # 0-1.0
env.SetSnow(0.0) # 0-25
env.SetTurbidity(5.0) # 2-10
env.SetAlbedo(0.1) # 0-1
env.SetCloudCover(0.0) # 0-1
env.SetRainRate(0.0) # 0-25
env.SetWind( [2.5, 1.0] )
```

FIGURE 9. Defining environment properties

The variables defining the environmental conditions can drastically change the appearance of the environment during the simulation, as shown in Figure 10.

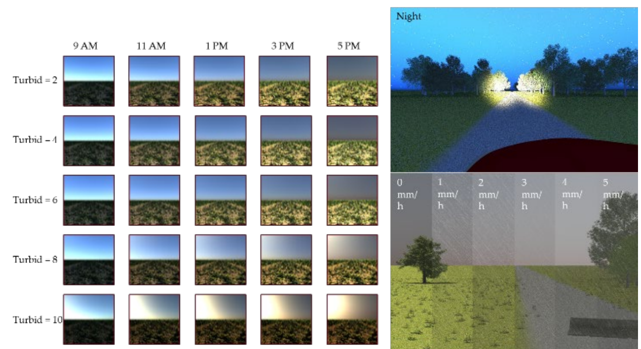


FIGURE 10. Environment property examples

Once the environment and scene have been defined and loaded, the sensors to be used within the simulation need to be defined. All sensors have certain functions such as Update(), SetPose(), SetOffset(), and Display(). Specific sensors will have additional functionality that are sensor-specific and defined within the API documentation (https://cgoodin.gitlab.io/msu-autonomous-vehicle-simulator/). Each sensor type will have a unique constructor.

```
# Create a MAVS Lidar
# options are 'VLP-16', 'HDL-32E', 'HDL-64E', 'M8',
# 'OS1', 'OS2', 'RS32', and 'LMS-291'
lidar = mavs_interface.MavsLidar('HDL-32E')

# Create a MAVS camera
cam = mavs_interface.MavsCamera()
# Initialize it with specified parameters
#cam.Initialize(1620,1080,0.006784,0.0054272,0.004)
# or choose an existing model from
# 'XCD-V60', 'Flea',' HD1080', 'MachineVision', or 'LowRes'
cam.Model('MachineVision')

# Create a MAVS radar and set properties
radar = mavs_interface.MavsRadar()

# Create a MAVS RTK sensor
rtk = mavs_interface.MavsRtk()
```

FIGURE 11. Defining a sensor

Properties for each sensor can be modified once the sensor is defined.

```
# Set camera properties
cam.RenderShadows(True)
# Increasing anti-aliasing will slow down simulation
# but give nicer images
cam.SetAntiAliasingFactor(2)
# This should be called for the camera to know about
# environmental factor like rain, etc.
cam.SetEnvironmentProperties(env.obj)
# If raining, render drops splattered on the lens
cam.SetDropsOnLens(True)
# Set the gamma (0-1.0) and gain (0-2.0) of the camera
cam.SetGammaAndGain(0.65,1.0)

# Set radar properties
radar.SetMaxRange(150.0) # max range in meters
radar.SetFieldOfView(15.0) # HFOV in degrees

# Set RTK properties
rtk.SetError(1.4) # avg error, meters
rtk.SetDropoutRate(1.0) # num dropouts/hour
rtk.SetWarmupTime(300.0) # Warmup time in seconds
```

FIGURE 12. Sensor properties

Each sensor has a defined offset. This offset is the relative position and orientation to the vehicles center of gravity and only needs to be set once at the beginning of the simulation. The arguments are an x-y-z position in the vehicle frame, followed by a quaternion for orientation.

```
# Set the sensor offsets
cam.SetOffset([1.0, 0.0, 2.0],[1.0,0.0,0.0,0.0])
lidar.SetOffset([1.0, 0.0, 2.0],[1.0,0.0,0.0,0.0])
radar.SetOffset([1.0, 0.0, 2.0],[1.0,0.0,0.0,0.0])
rtk.SetOffset([1.0, 0.0, 2.0],[1.0,0.0,0.0,0.0])
```

FIGURE 13. Sensor offsets

Once the sensors are defined, a vehicle model can be selected. A vehicle can be created by loading a pre-defined vehicle input file. The initial position and orientation of the vehicle needs to be set within the script.

```
#Create a MAVS vehicle
veh = mavs_interface.MavsRp3d()
veh_file = 'forester_2017_rp3d.json'
veh.SetInitialPosition(5.0,5.0,1.0) # in global ENU
veh.SetInitialHeading(3.14159) # in radians
veh.Load(mavs_data_path+'/vehicles/rp3d_vehicles/' + veh_file)
```

FIGURE 14. Loading a vehicle

Once a vehicle is selected, a driver needs to be defined. A user can leverage keyboard teleoperation, built in waypoint following, or external controllers. If the user wants to drive using a keyboard, they must define a camera sensor for driving the vehicle. An example of this is:

```
# Create a window for driving the vehicle with the W-A-S-D keys
# window must be highlighted to input driving commands
drive_cam = mavs_interface.MavsCamera()
drive_cam.Initialize(256,256,0.0035,0.0035,0.0035)
drive_cam.SetOffset([-10.0,0.0,3.0],[1.0,0.0,0.0,0.0])
drive_cam.SetGammaAndGain(0.6,1.0)
drive_cam.RenderShadows(False)
```

FIGURE 15. Keyboard Control Camera

A vehicle driven with an autonomy algorithm does not need a drive camera, and is defined as follows:

```
waypoints = mavs_interface.MavsWaypoints()
waypoints_file = 'square.vprp'
waypoints.Load(mavs_data_path+'/waypoints/'+waypoints_file)
controller = mavs_interface.MavsVehicleController()
controller.SetDesiredPath(waypoints.GetWaypoints2D())
controller.SetDesiredSpeed(5.0) # m/s
controller.SetSteeringScale(0.7)
controller.SetWheelbase(2.6) # meters
controller.SetMaxSteerAngle(0.615) # radians
```

FIGURE 16. Waypoint Controller Example

Once all components are defined MAVS needs a simulation loop. At each time step, each of the components must be updated (environment, sensors, vehicles, driver, animations). Sensors that run at a slower frequency may not need to update at each step.

```
dt = 1.0/30.0
n = 0
while (True):
    # Get the driving command
    if (control_mode=='human'):
        dc = drive_cam.GetDrivingCommand()
    else:
        controller.SetCurrentState(veh.GetPosition()[0],veh.GetPosition()[1],
                                   veh.GetSpeed(),veh.GetHeading())
        dc = controller.GetDrivingCommand(dt)

    # Update the vehicle
    veh.Update(env, dc.throttle, dc.steering, dc.braking, dt)

    #update the environment
    position = veh.GetPosition()
    orientation = veh.GetOrientation()
    env.SetActorPosition(0,position,orientation)
    env.AdvanceTime(dt)

    # Update the sensors
    cam.SetPose(position,orientation)
    cam.Update(env,dt)
    cam.Display()
    radar.SetPose(position,orientation)
    radar.Update(env,dt)
    rtk.SetPose(position,orientation)
    rtk.Update(env,dt)
    if n%3==0:
        lidar.SetPose(position,orientation)
        lidar.Update(env,dt)
        lidar.Display()
    drive_cam.SetPose(position,orientation)
    drive_cam.Update(env,dt)
    drive_cam.Display()
    n = n+1
```

FIGURE 17. Simulation Loop

Alternatively, MAVS-Python implements a simulation class that handles the simulation management and updates automatically. In a simulation file, all parameters are pre-defined within a single input file.

```
# Simulation input file
sim_file = 'sims/sensor_sims/halo_city_sim.json'
# Create a MAVS simulation and load input file.
simulation = mavs_interface.MavsSimulation()
simulation.Load(mavs_python_paths.mavs_data_path+'/'+sim_file)

while True:
    simulation.Update(1/30.0,update_actor=True)
```

FIGURE 18. Simulation class

This can be useful for sharing pre-defined simulations which have been saved to a sim file for re-simulation.

Within the simulation update loop, data can be accessed by the user through sensor specific function calls.

```
# Get [x,y,z] registered points in world frame
points = lidar.GetPoints()
# Get [x,y,z,I,label] points in the sensor frame
raw_points = lidar.GetUnRegisteredPointsXYZIL()

# Get the RTK output
measured_position = rtk.GetPosition()
measured_orientation = rtk.GetOrientation()

# Get the radar output, list of
# [x,y] locations in the radar frame
targets = radar.GetTargets()
```

FIGURE 19. Sensor data access

This data can be saved to disk. The resolution of the data can cause a reduction of less-than-real-time execution depending on the complexity of the simulation environment. However, there will be no loss of data quality.

```
cam.SaveCameraImage((str(n)+'_image.bmp'))
if n%3==0:
    lidar.SaveColorizedPointCloud((str(n)+'_cloud.txt'))
    # save top-down rendering of point cloud
    lidar.SaveLidarImage((str(n)+'lidar.bmp'))
```

FIGURE 20. Saving data to disk

One of the tools defined within MAVS is the automated labeling of camera and LIDAR data.

Labels for each mesh are defined within *mavs/data/scenes/meshes/labels.json.* New label definitions can be created and added to the scene file. Labeling can be done at the material or object level. Labeling is disabled by default since it takes additional time. It must be enabled before the main simulation loop.

```
# Uncomment this if you are saving labeled data
scene.TurnOnLabeling()
```

FIGURE 21. Enabling labeling

Once it is enabled, within the simulation loop you can save out the annotated data.

```
# Save labeled data
cam.SaveAnnotation(env,('annotated_'+str(n)))
lidar.AnnotateFrame(env)
lidar.AnalyzeCloud('annotated_lidar',n,False)
```

FIGURE 22. Saving labeled data to disk

5. Conclusion

The MAVS-Python integration and usage as described in this paper provides for an easy-to-use Python integration for MAVS. This MAVS-Python interface enables novice users to interact and use MAVS to produce high-fidelity sensor data. That data can be annotated if desired. Additionally, the MAVS-Python interface provides an easy-to-use update loop that allows for the user to provide any type of input they desire to control the vehicle within the environment, beyond the pre-defined control methods. Finally, this Python integration allows for MAVS to be integrated into a variety of machine learning packages often written for use within Python. MAVS is free and open source for non-commercial use (http://www.cavs.msstate.edu/capabilities/mavs.php).

6. REFERENCES

[1] C. Goodin, M. Doude, C. Hudson, and D. Carruth, “Enabling off-road autonomous navigation-simulation of lidar in dense vegetation”, Electronics, 7(9):154, 2018.

[2] C. Goodin, D. Carruth, M. Doude, and C. Hudson, “Predicting the influence of rain on lidar in adas”, Electronics, 8(1):89, 2019.

[3] M. Cole, C. Lucas, K. Kulkarni, D. Carruth C. Hudson, P. Jayakumar “Are M&S tools ready for assessing off-road mobility of autonomous vehicles?”, Proceedings of the Ground Vehicle Systems Engineering and Technology Symposium (GVSETS), NDIA, Novi, MI.

[4] C. Goodin, S. Sharma, M. Doude, D Carruth, L. Dabbiru, C. Hudson, "Training of Neural Networks with Automated Labeling of Simulated Sensor Data," SAE Technical Paper 2019-01-0120, 2019.

[5] C. Hudson, C. Goodin, M. Doude, D. Carruth, “Analysis of Dual LIDAR Placement for Off-Road Autonomy using MAVS”, Proceedings of the World Symposium on Digital Intelligence for Systems and Machines (DISA 2018), pp. 137-142. Kosice, Slovakia.

[6] Waymo, “Waymo Safety Report. On the Road to Fully-Safe Driving”, 2018

[7] Dabbiru, L., Goodin, C., Scherrer, N., & Carruth, D. (2020). *LiDAR Data Segmentation in Off-Road Environment Using Convolutional Neural Networks (CNN)* (No. 2020-01-0696). SAE Technical Paper.