EEC 193B Autonomous Car Senior Design Final Report

Adam Jones, Thanh Le, Vidush Vishwanath, Sarvgaya Singh, Christopher Uy, Kalani Murakami, Sam Truong, Devashish Kashikar, Gabriel Lee

I. Summary

The main purpose of this report is to summarize what the team has collectively achieved and show what each individual has contributed to the project. Our final product is a self-driving car that uses a pre-trained neural network in order to maneuver around a known track. Occasionally, if there is a stop sign, the car is able to stop itself.

An important note is that there were many tasks that our members worked on this quarter which were not needed for the current car. Because our objective at the end of the quarter was to have a car that could drive itself on a track, tasks such as Simulation, ROS communication, etc. were not directly applicable. However, such tasks are crucial for the development of future tasks, and they lay the foundation for which future project members can build upon. Because of that, this report will include every task that the team worked on to build a better car, not just the current one.

The tasks that we have accomplished are:

1) Traffic Sign and Pedestrian Detection.
2) Jetson Flashing and Installation.
3) Design and Modify the car chassis.
4) Establish UDP connection between the Jetson and Raspberry Pi.
5) Modify the motor board to install heat sinks.
6) Build a track and Collect data from it.
7) Train the data on Titan XP.

These are essential tasks that work on the current car. Furthermore, the following are tasks that we pass on to the next project members.

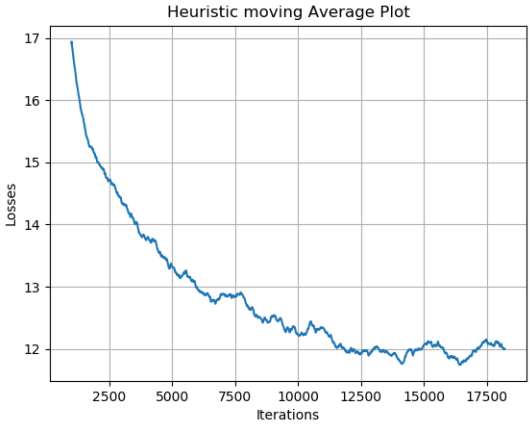
1) Simulation on Deep Drive.
2) Lane Line Detection.
3) Long-range Path Planning.
4) Short-term Path Planning and maneuvering.
5) ROS packages for communication among components.
6) SLAM with Lidar.
7) Make installation Packages for future hardware.

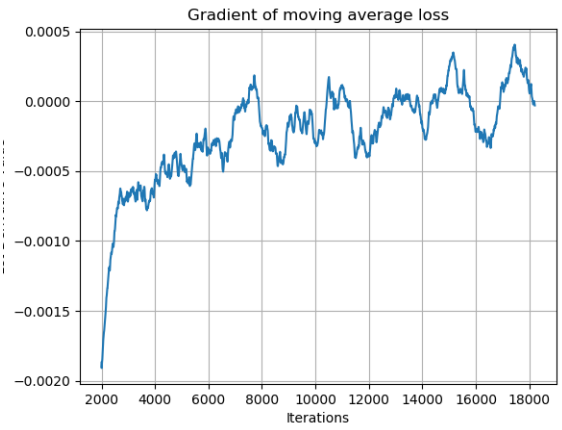
In the following sections, we will talk about each tasks listed above in more detail. We will address the final status of each tasks, the difficulties we encountered and intricacy of dealing with specific hardware bugs. All of this is in the hope that future project members do not have to repeat the same mistake, in order for them to focus on more important issues.

II. Traffic Sign and Pedestrian Detection

Sam Truong, Kalani Murakami

For traffic sign and pedestrian detection, we continued to use the Darknet framework and the TinyYOLO architecture to perform both our pedestrian and traffic sign detection. However, for this quarter, we chose not to continue with using GPU computing in the cloud, and instead, Sam used the Titan XP to train the model. This was to reduce costs for training, since the GPU computing service charged at an hourly rate. Using the Titan XP, we have more control over our model and was able to reach better convergence compare to when training on AWS. This is because not having the budget constraints allow us to have the freedom to test different hyper-parameter and fine-tuning the model. The following images are the characteristic plots through out the training.





Recalling from the previous report about training for Traffic Sign Detection and Classification, we have the loss at around 60. We were able to get the loss down to around 12 using the Titan XP. Although the loss can get even lower than

that, experiments have shown that for a very small number of classification (5-10 classes), where different classes are very distinctive, a high loss value is acceptable. We also plot the "Gradient of moving average loss". This plot helps us to know when the model is approaching convergence. If the value of this plot fluctuate around 0, then we can act accordingly. This means if the loss is still high and gradient is 0, we will decrease the learning rate or increases the batch size. It is worth mentioning that sometimes increasing the learning rate might actually help us escape convergence (as oppose to decreasing it). This is true when we are stuck in a local minimum. We need a very large step to get out of the depression. Again, without the Titan XP, testing and reverting back to last training check-point are financially expensive operations. Each time we decide to revert the training to the last saving point, it can take 3-4 hours to see the result.

The final result of this task can be confirmed by testing on multiple validation images. Our most important classes are: Pedestrian, Bike, Car and Stop sign, all of which are successfully detected with the average accuracy of 95%. We argue that for a real-time video streaming, the constraint on accuracy can be relaxed since there are more than 1 frame used as input. Even if the model fails to detect the classes in one frame, it still have a high chance of successfully detect other frames. The following images are our results:

Difficulties and Future Notice

We have modified the training files so that now it is very tractable to retrain and validate the training. The bash script "run.sh" inside the "darknet" directory takes care of all hyper-parameter specification, starting points and configuration files.

Also, the Jetson only has about 200 CUDA core, and therefore having a detection/classification algorithm on board is costly. This is because the Jetson is already inferencing a supervised neural network in order to output car's action. To fix this issue, initially we want to stream the camera output to the Titan XP, where the detection/classification will be inferenced. However, using UDP as network prototcol, we encountered latency issue. The video was only streamed at 0.6 fps.

For future reference, we believed having a more powerful processor on board will allow the detection/classification algorithm to run in parallel with other tasks on the GPU. Alternatively, we could try to fix the UDP connection, because other applications have shown that it was possible to stream video in reasonable frame per second. Both approaches needed additional requirement. While having more powerful processor requires a better car chasais, fixing UDP protocol required a close network for edge computing.

III. Jetson Flashing and Installation

Sam Truong, Christopher Uy, Devashish Kashikar, Vidush Vishwanath

The Jetson was flashed with Jetpack 3.1 which included CUDA 8.0, cudNN 6.0, Ubuntu 16.04 LTS, TensorRT 2.1 and OpenCV 2.4.1. The OpenCV version was archived and did not support Python3, so the binaries and libraries had to be removed and rebuilt from source. The Jetson TX2 ran on an ARM64 architecture and some packages like Tensorflow had to be compiled from source to work on the platform. We recognized that ARM architecture helped reduce the power consumption on embedded device, however, there were not a lot of support for needed libraries.

Difficulties and Future Notice

There were quirks with the Jetson we found while working on it. The HDMI connection does not work with a USB hub plugged in, and sometimes suddenly shuts off. There also was an issue activating the SPI protocol on the Jetson, which will be explained in the UDP connection section later on.

Additionally, since the libraries and frameworks we installed on Jetson (CUDA, cudNN, TensorRT, OpenCV, Tensorflow) took up more than the available space on the Jetson (they took up more than 32 GB), we needed to purchase a separate SSD and a SATA cable to provide more secondary storage. This required us to move the root director and boot files to the directory. In all, those procedures were tiresome but needed.

IV. Mounting the Car

Adam Jones, Christopher Uy, Devashish Kashikar, Thanh Le, Sam Truong

The car was modified many times during the course of this project. The first edition of the car was built only with the parts that came with it. In order to all run the software to make the car "self-driving" we needed to include many things on the car. The Nvidia Jetson, Raspberry Pi, lidar, motor control board, battery pack, heat sink and camera were all the things that we had mounted on the car. We needed to build a platform on the car in order to fit all of these things.

The first model we built consisted of 1 level. We flipped the car around so that the the belly of the car did not touch the ground. We used ply wood to make the surface of the chassis flat. This allowed us to put the Lidar, 1 battery unit, and the Raspberry Pi on it. We also later mounted an adjustable camera on to the chassis. However, this configuration did not allow us to have the Jetson on board. This turned out to be a big issue. because the raspberry pi was not fast enough to run our neural nets, we had to put the jetson on board.

The second model we built consisted of two levels of plexi glass which held all the hardware we needed. After mounting all the materials on the glass, we made an attempt to run the car and realized that the glass was way to heavy. It squished the tires making the car very difficult to accelerate. The car would overheat almost immediately. The power bank itself was 2 lbs, so we had to buy an even lighter power bank that could power both the Jetson and Raspberry Pi at the same time.

The third model consisted of two levels of thinner plexi glass. The original two pieces of plexi glass were heavier than the car itself. With these two thinner pieces of plexi glass and a lighter power bank, we thought the car would be able to move without overheating. However, while the car was able to run for longer, it still overheated too early.

Our latest edition of the car consists of two levels of ply-wood to hold all of the hardware. The ply-wood was even lighter than the plexi glass, and we added heat sinks to the tops and bottom of both the motor control board and the raspberry pi. With this final reduction in weight and the heat sinks, the car is able to make 2-3 laps around the track without overheating.

Difficulties and Future Notice

As mentioned earlier, the weight of the car was a huge problem. It took a week to come up with a solution and gather the materials to implement that solution. Our car does not have very high-torque motors and can only handle up to 3.8 Amps. Our car also has no suspension, which makes it very difficult to hold any weight the motors still overheated. Even with all the reductions in weight and the heat sinks, our car can still not run continuously for very long. For the future, we will need to buy a car that can handle a lot of weight.

V. Establish UDP connection between the Jetson and Raspberry Pi

Thanh Le, Gabriel Lee

The UDP connection between the Jetson and the Pi is for device communication. Originally, we had planned to use SPI, but the SPI protocol could not be unlocked to use on the board without re-flashing the Jetson. We decided against re-flashing since all prerequisite packages, binaries, and libraries were all ready on the board. We decided to switch to a UDP connection because it was easy to establish and we were limited in our UDP connection by how fast our model can output steering angle. We had a UDP python script from last quarter stored on the Pi, but since then the Pi had been wiped several times and all our progress was lost. We can blame this issue on our poor use of git during the duration of our first quarter. Writing the UDP script again was not a huge issue; however, if we had the previous script this issue would have only taken us 30 minutes instead of 2 hours. This issue highlights the importance of having well-documented and back-up code for our project.

VI. Build a track and collect data from it

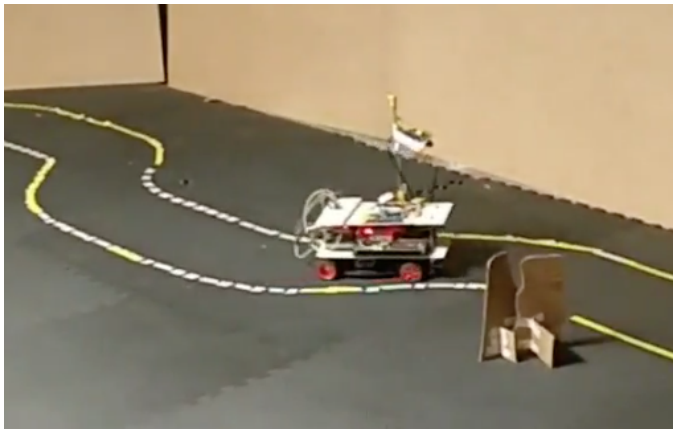
Adam Jones, Kalani Murakami, Christopher Uy, Thanh Le, Devashish Kashikar, Vidush Vishwanath

We built the track during the first quarter of the independent senior design in order to capture reliable training data for our models. The track consisted of 53 interlocking black foam floor mats covered in white and yellow tape signifying the lanes lines. Throughout the project, we knew that we had to modify the track-layout in order to create a track that was traverse-able by the RC Car.

Our initial layout of the track was too ambitious. The car could not physically make some of the turns. In the second iteration of our track layout, we smoothed out some of the sharp turns, but it was still difficult for a person driving the RC car to make some of the turns without slowing down to a crawl. We trained the RC car on this iteration of the track, but the car could only consistently make three turns on the track.

The turns required the RC car to switch from turning full right to turning full left instantly on a slight downhill. A person driving the car can easily anticipate the rapid change in steering angle, but the RC car could not. We added correction data for the two sharp turns on the track by driving the car through them again and again, but it did not change the behavior of our model. With only a week before our deadline, we decided to modify the layout of the track again to accommodate the car.

Thus, in our final layout, we designed a track with turns that were optimal for the RC Car to traverse, but still included a variety of curves and turns that our model might encounter in real world scenarios. Our layout includes both short and long curves at various curvatures to produce reasonable training data.



Difficulties and Future Notice

It was difficult to collect data from the track due to poor lighting in Hershel's garage. We had to install a ceiling light in order to light the track in an even manner. Along with poor lighting, the garage was also slanted towards the garage door. This made it even more difficult to train the car because the way we kept the car moving was by making the power constant and the car needed more power for turns and less for straight parts. We soon found out that this method will not work because the car started on a downhill and was able to make the first turn with ease. On the second turn, it was difficult for the car to turn because now, the car had to go on an uphill and would often times stop moving. It is very important that the track be on a flat surface in order to make sure that the car will continue moving in a normal manner.

VII. Training Models on Titan XP

Adam Jones, Sam Truong

The Titan Xp used for this research was donated by the NVIDIA Corporation. We mainly used the Titan XP to train 2 tasks: (1) the aforementioned YOLO network for detection/classification of traffic signs and pedestrian. (2) the supervised neural network used to output the steering angle given an image containing at least one line. Here, we would summarize both tasks training statistics and further elaborated on task (2).

Using the Titan XP was been a great choice, because working with AWS was very expensive. We acknowledged that using AWS can further scale our application, however, our objective right now did not require such a bulky and expensive resource. It took a lot of fine tuning in order to training a neural network. Spending time adjusting the hyper parameters on AWS meant we needed to spend a large portion of our budget on fining the right set of parameters.

For task (1), the total training time, including the time that we went back and fine tune the parameters, took 36 hours. We trained on 200 epochs, with the learning rate decreased from 0.001 to 0.0005 to 0.0002. Because we had the luxury of going back to the last checkpoint and retrain, there are a lot of local minimum that we escaped by changing the learning rate. The batch size also got increased from 32 to 64 as we started to approach convergence. Having larger batch size allowed the descent to be less stochastic and more general.

For task (2), we used the term "Lane Line detection", however, it was important to know that this task is separate from Adam's lane line detection, which used traditional computer vision. The original lane line detection will be used to provide the inputs to our reinforcement learning approach, which will be implemented in the future. Here, we used supervised learning in order to output an steering angle number based on the camera input. This tasked is purely supervised learning. We tried two different neural nets: Nvidia and LeNet. Click here for the original Nvidia neural network architecture Click here for the original LeNet neural network architecture

Nvidia's model worked on a real car. It took in an image as an input and output a steering angle. This alone was able to drive a real car. However, it did not work so well in Hershel's garage. The model that worked was a modified version of LeNet. LeNet is sort of the "hello world" when it comes to image classification. We modified the input and output layers to match what we needed for the self-driving car, added another 2D Convolutional layer, and applied the same image pre-processing that Nvidia did for their real self-driving car. This included cutting off the top of the image (don't need to look at the roof to know where to drive), converting to YUV color space, applying gaussian blur, and converting image shape to (66, 200, 3). The original image shape was (180, 320, 3). We kept the input image size as small as possible to make the neural net as small as possible. We think that the modified LeNet model worked so much better than Nvidia's model because modified LeNet had 8.9 million parameters, whereas Nvidia's model only had 250,000 parameters.

With 100 iterations, it took the Titan XP about 30 minutes to train the neural network. Using other GPU's from AWS, it would have taken between 45 - 60 minutes. The result was the car being able to self driven based solely on the camera input. This was a good use of the Titan XP since it allowed us to try different models with different hyper-parameters in a very short time, delivering us a working model.

Difficulties and Future Notice

Because the Titan XP requires its own computer (our workstation), it would not be mounted on the car. Unless we got a much bigger car, it would be impossible to have wired communication between the workstation and the Jetson. We ended up using a flash drive to transfer the models from the workstation and the Jetson because it was the fastest solution. As we mentioned, UDP protocol was not fast enough to transfer frames in real time for now. This defeated the purpose of having a very powerful piece of hardware, because we have a bottleneck when transferring data. To solve this, we needed to figure it out how to speed up UDP communication.

VIII. Simulation on DeepGTAV

Kalani Murakami, Thanh Le, Christopher Uy, Gabriel Lee

1) Introduction: Deep Drive is a software that allows the training of self-driving car models using Grand Theft Auto V as its environment. Official development of Deep Drive has stopped due to take down notices from Grand Theft Auto V publisher Take-Two Interactive so we needed to use forked repositories. We decided to use a fork of the repository called VPilot because it was the best documented out of all the forked DeepGTAV networks.

DeepGTAV works as a plugin for the game when you run it. The files inside the repository go into the game directory of your copy of GTAV. The plugin was created in C++ using Visual Studios. The plugin works off a already created Saved File which runs as you are driving the car. The plugin comes with different configurations that you can adjust as you drive the car. You are able to configure these parameters using JSON messages you send to the game using a client. There are 4 different messages you can send. The 'Start' signal which is required to run the simulator from the start. You cannot run any other command unless this is started first. The 'Config' signal is then used to override any settings used at the current time such as the settings of the 'Start' signal. The 'Command' signal is used to give manual directions to the car at any given moment allowing you to configure the throttle, brake, and steering angle. The car must be in manual mode for this to happen. The last signal is the 'Stop' signal. This stops the car from receiving messages from the Client. Each signal comes with two different parameters that you must configure besides the 'Stop Signal'. The first parameter is the 'Scenario'. The 'Scenario' allows you to configure the location in GTAV, time, and others such as auto mode or manual mode. The 'dataset' parameter allows you to configure rate, frames and what you want the car to do itself.

2) Installation: To get DeepGTAV to work we need to revert the version of GTAV that was installed in our team member's machine to an older version. After some research we were able to install the correct version. The first repository that we were testing our simulations with seemed promising, because we were able to use the in-game AI to collect data consistently, however, the function included in the repository to train the actual car itself did not work. The model which was included in the project also did not work. After struggling with the issues for a week, we decided that our best bet was to look at different implementations.

The program runs on Windows 10 and was a challenge for us since it was a different development environment than Unix/Linux. The DeepGTAV project is compiled in a visual studio project. It was difficult for us to get the simulation to work properly because there were issues in the repository that we forked. The first issue we encounter was that we weren't able to run any of the functions included in the repository due to the fact that we didn't have the correct version of OpenCV installed in the system. Although we picked the most well documented version of DeepGTAV available, it didn't mention anything about which version of OpenCV was used for the project.

To understand what the simulator is doing we had to modify the Visual Studios project. We first had to understand Visual Studios. We ran into errors using Visual Studios including version control of visual studios. We went from Visual 2013 - 2017 only to learn that it could only be done in 2017 due to the Windows Game SDK that is only available in the 2017 version. Then from there we imported the project using the .sln files from the main GTAV Github. We then had to import all of the libraries used in the GTAV project from files that build from other libraries. Due to the development environment of our nature, using multiple drives, we ran into issues there.The OpenCV directory where all the libraries are being contained wasn't found. From there, we first tried to manually link them together only to find that we weren't suppose to use OpenCV 3.4, but OpenCV 2.3.4. The reason we struggled with that was because there was no indication that we could find on which one to use. From there we learned we didn't have to use CMAKE to build OpenCV, but store it in the main C:Drive. The main C drive is where Visual Studios finds libraries for some reason, even though in the build settings of the project, we linked it to the D:drive to build from. We don't know why it works the way it does, but it does. We are now able to successfully use Visual Studios for the project and modify the C++ code.

3) VPilot: VPilot is a scripts and tools using the Python libraries to communicate with DeepGTAV. This was also created by 'aitorzip'. This communicates with DeepGTAV over a TCP connection. With this, we can easily write Python implementations of this so we can also incorporate our previous knowledge of using machine learning libraries in Python. VPilot also comes with four important functions. Messages.py and Client.py are the main functions that allow us to communicate with DeepGTAV. They take care of the JSON messages you send and starting up the TCP client. The python scripts that build off these are dataset.py and drive.py. Inside dataset.py, the car is set into autopilot mode which uses the functions inside of GTAV and drives around. The game doesn't use the in-game AI to drive. The car is actually using scripts that were made inside the original plugin of DeepGTAV and uses the files 'Lanerewarder.cpp' to stay in the lane lines consistently. In drive.py, the car takes in a dummy model and uses this model to drive the car. The purpose of this skeleton code is to modify ourselves and place it where the dummy model is created.

4) Data Collection: In the C++ implementation,the files are already generated in a file called dataset.txt. This file contains 8 different parameters. They are all found in the Github of the main project, but the mains one we want to focus on are steering, braking, and throttle as we did in the Santos Net implementation. The dataset.txt file is a space separated formatted file with the first number being the indication of which frame to associate the line of data this is from. This is important in training the model as we associate the line number with the frame at that time. The data collection can be configured using the config.ini file located in the Github. The config.ini file is a file that is read in when

the simulator starts.The config.ini file allows you to change various parameters such as the capturing frame rates, weather, and environment type. More can be found in the Github. You are also able to specify the path of where the data and frames are stored. The frames and dataset.txt file are stored in the D:drive. This picture below shows an example of the simulation environement in DeepGTAV.

5) Training:

a) Santos Net: This is a implementation is a sample implementation that uses the skeleton code provided in VPilot and builds on top of this. This is able to drive around and take the dataset into a .pz file which is a video file and train based off that. On further inspection of the code, we seem to cannot use this code and modify it for our purposes. They model they train only trains the steering of this car.

b) Lemon Net: The neural net was help created using the Github called LemonAniLabs. The model uses a CNN and LSTM to do temporal inferences on the network. The model takes in multiple dataset.txt files and the frames in a numpy array. Then it goes through the layers and exports the results in a .hdf which is the file needed to run the script drive.py. Drive.py uses the model.py functions to drive around the car in a reinforcement learning environment. Drive.py uses a server to communicate with the simulator of DeepGTAV. After that it chooses the predicted actions given the current frame

6) Conclusion: In the end, we managed to get the simulation to run and work, and had roughly 60GB of training data ready, however the training script ran into an error and would only pick up a black box. The GTAV window could not be found by the program to do training on. We hotfixed this issue by making the window full-screen, and just capturing the whole screen instead of a specific window, while throwing out the first hundreds frames of data which capture irrelevant data. After applying the hotfix, we were able to produce a model from our training data.

Difficulties and Future Notice

The future of the simulation team is collect more data and train a robust model from that data. After creating a model we apply that model to the RC Car by mapping all the outputs of the model to controls on the car. Our RC car has different steering and acceleration characteristics than the Car in the simulation. Changing how the simulation car behaves would be easier because all it's characteristics are controlled with software variables. We would also have to optimize the RC Car controls so that it would align with how the model thinks the car would behave. The simulation will also be a platform for testing our traffic sign, pedestrian, and lane line detecting algorithms. By simulating a wider variety of environments and conditions we could optimize our Reinforcement Learning algorithms. After successfully produce a policy in simulation, we could try to transfer it to the real car model. Anyone who wishes to take over the project must be well versed in Visual Studios. Even though members from our team were well versed in C++, understand how Visual Studios handles packages and dependencies took a significant amount of time. Our team spend a significant amount of time not being able to run the functions from Lemon Net because we did not understand that Visual Studios required its own version of OpenCV in order to compile the program. Even after figuring this out, we encounter numerous dependency and compilation errors with the code. To replicate our current simulation on another machine will require a significant time to create/fix any environmental dependency. We have documented all the steps necessary to achieve in a document. Future considerations include creating a stream lined process to train, extract, transform, and load data. Another consideration in the future is to add the lane line detection in the simulator itself without relying on the auto reward function in the Visual Studios Project.

IX. Lane Line Detection

Adam Jones, Vidush Vishwanath, Sarvagya Singh, Devashish Kashikar, Christopher Uy

The lane line detection algorithm is the same as it was last quarter, but this quarter our focus was on making it faster. We initially planned to convert the Python code to C++ and use the OpenCV CUDA wrapper to parallelize functions and increase performance. The preprocessing functions such as calculating magnitude and angle were successfully converted to C++ code because the python code was using the CV2

library. However, once numpy arrays were being used there was no direct OpenCV C++ function that mapped to them, so brand new code was written. Towards the end, it became too time costly to rewrite function in C++ because some of the functions we wanted to use like logical-and and logical-or only existed within the OpenCV CUDA C++ wrapper. At the time of writing the code, we did not have access to an NVIDIA Jetson nor the Titan XP.

With this information in conjunction with analysis using cProfiling and GraphViz time analysis libraries, it was revealed that translation of program into C++ was not going to make much difference. These scripts analyzed the lane-line detection python scripts in run time and created a profile of run-time of each function invoked by the script. This revealed that most time was consumed by openCV functions which are actually implemented in C++ with a Python wrapper around them. Hence, translation of the lane-line detection would have been unfruitful. This can also be seen in figure 1 which highlights the bottlenecks in our lane-line detection script.

However, we found a python wrapper called NumbaJit which does the same functionality for numpy. By adding this wrapper and compressing the original video resolution of the camera to 320x180, the Titan XP was able to run at 99fps. Ultimately, it is important to note that this code was implemented on a CPU. Therefore, the majority of OpenCV calculations were done in Serial. It was noted that decreasing the image size should significantly decrease the amount of time taken per image and therefore increase the throughput of the algorithm. We therefore created a simple OpenCV program to half all the frames extracted from a video file. Running these frames through the lane line detection program indicated a massive speedup in throughput.

X. Long-range Path Planning

Sarvagya Singh

The main focus of long-range path planning is to help the autonomous car maneuver from point A to B in the actual world. In order to do so, the car needs an extensive amount of data that can help it follow an efficient path from the origin to the destination while continuously guiding it. We used the Google Maps API to help use resolve this issue.

One can simply provide origin, destination and the mode of transport to the Google Maps API, and it returns a detailed data about the path in JSON format. The JSON file is returned in the form of a web page, so we have to use a web-scraper to copy the data on our local machine. Once, we have the data we created a wrapper to access the data, so it can be used to for the safe maneuver of the car.

Each path is divided into multiple legs where the legs are divided by turns, merge and freeway entry and exits. The user can use the wrapper to get detailed data about each leg. The user can get data about the current leg by using the command print current step. This gives a detailed description of set of closely located GPS coordinates that car must follow to complete the current leg. It also gives detail about the traffic condition about that leg, data about the transition to the next leg, duration of the leg, distance of the leg, end location and start location. All these can be essential for safe and efficient maneuvering of the car. Once the current leg has been completed, the user can simply use the next step command to go to the next leg.

Below is an example showing the details provided by the API about one of the legs of the path from Davis to San Jose. The polyline feature is actually encoded set of GPS coordinates that constitute the path, and is helpful in making sure that the autonomous car is following the supposed path.


```
{'distance': {'text': '318 ft', 'value': 97},
 'duration': {'text': '1 min', 'value': 43},
 'end_location': {'lat': 38.5419547, 'lng': -121.7406881},
 'html_instructions': 'Turn <b>right</b> onto <b>1st St</b>',
 'maneuver': 'turn-right',
 'polyline': {'points': 'mwfjFng`fVPhBTnB'},
 'start_location': {'lat': 38.542154, 'lng': -121.7396041},
 'travel_mode': 'DRIVING'}
```


XI. Short-term Path Planning and maneuvering using SLAM

Sam Truong, Sarvagya Singh

Simultaneous Localization and Mapping referred to a computation problem that refers to constructing a map of an unknown territory, while continuously keeping track of the mapper with respect to the feature present in the surrounding. It was useful in case of the autonomous car as it could be used to map its current traffic environment and helped the car made better decisions.

For example, in case of pedestrian detection, SLAM could help us point out the exact physical location of the pedestrian and ensure its safety. SLAM would also be helpful in ensuring the safety of the autonomous car when it was merging into the freeway or changing lanes as it would make sure that the no obstruction was presented in the blind spots of the car. It would also be helpful in scenarios involving parallel parking.

For this project, we used the hector SLAM library and RPLidar in the ROS environment to create a SLAM map of the local environment, which could be clearly noticed in the graphic below. The graphic displayed a rough map of Sam's room and the blue spot represents the location of the lidar with respect to Sam's room (Figure 2). The project was implemented such the the mapping and the localization data can be published to a ROS topic, and could be used by our AI models to make decisions on a finer granularity.

Difficulties and Future Notice

There are several ROS launch files that needed to be modified before SLAM works. This was because the default status of the package assumed simulated time. That meant the input was taken from a pre-recorded file and not in realtime. Also, the odometry of the Lidar was also needed to be specified. All changes were saved into a new ROS package.

Initially, we wanted to put Lidar ROS packages onto the Raspberry Pi. Since it was decided that the Jetson would be on board the car, it simplified most of our installation procedure. ROS was a very particular domain

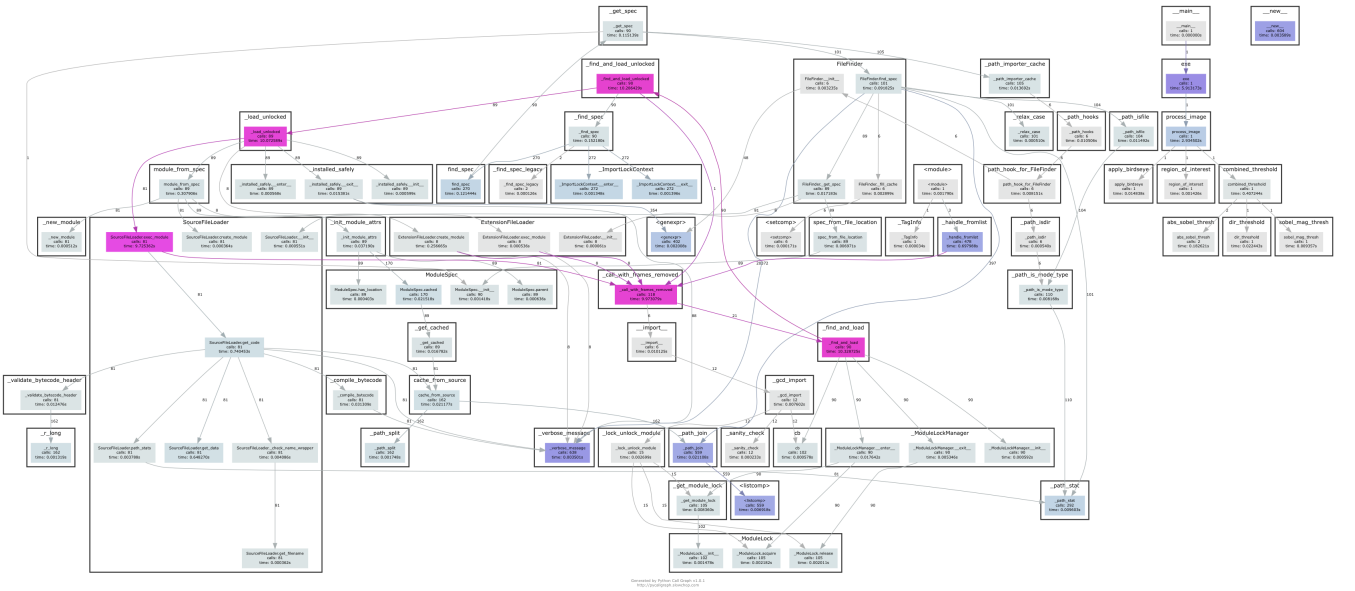


Figure 1: Lane Line Detection C++ Analysis

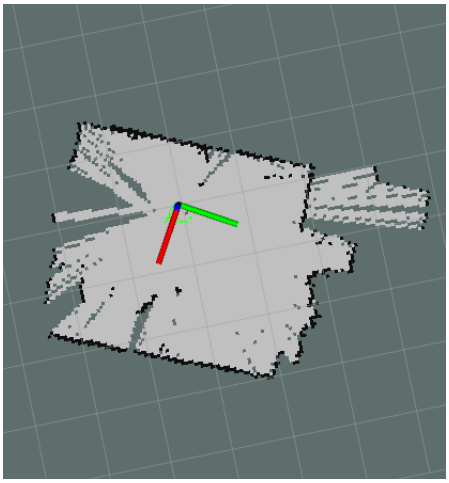


Figure 2: SLAM mapped Sam's room

that only worked well on Ubuntu machine. The Raspberry Pi OS (Raspian) required all of its ROS packages to be built with CMAKE. From our experience, that was a very tiresome task since built fail rate is high. Moving ROS packages away from the Pi was a good thing. We recommended only let the Pi handle motor control later on. All communications between nodes, including SLAM nodes, should be processed from Jetson or other Ubuntu machines.

XII. Short-term path planning using heuristic A* with potential function

Sam Truong

This Short-term path planning is different from SLAM because it offers more fine tuning over how the path is made. While SLAM offers how to make the car follow a desired path, heuristic A* can defined what a desired path look like. This task helped us when it came to studying maneuverability. It would be helpful later on when we need a reward function that evaluate the car's path during Reinforcement Learning. To further illustrate the difference and role between SLAM and A*, we can look at the following scenario.

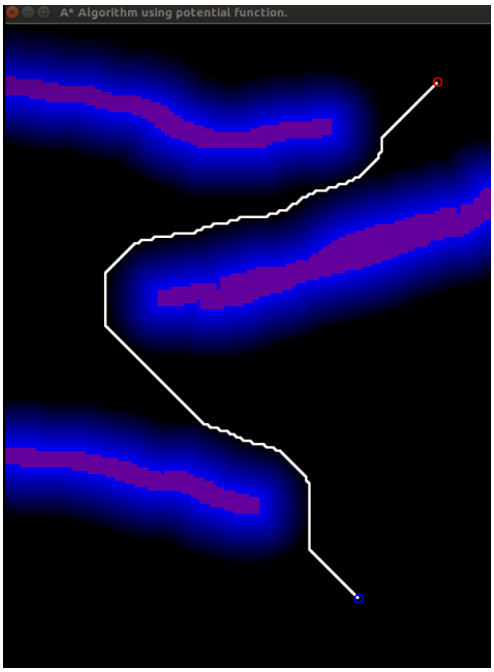
Given a situation where a car needs to move to the right lane. SLAM will give us a full description of the surrounding obstacles. With that information, A* (and later Reinforcement Learning) can sketch out an ideal path for movement. The car would then incrementally follow that path. SLAM will tell the car if it was going of the ideal course and readjust the car. Here, A* assumed the role of a reinforcement learning policy. It was easier to study and develop A*, but later on the the role of sketching the path would fall back to a refined RL policy, once that would be trained with Simulation.

A* algorithm was an extension of Dijkstra's shortest path algorithm. However, it did not search every possible next node in order to find the optimal path. This reduced the amount of computation, which mean sketching a path in real time is possible. Not only that, we added a potential function that associated different weights values to different point in space. Points that are closer to the obstacles would receive higher weights. This effectively made the optimal path to be the one that is furthest away from every obstacles. The following image describes the optimal A* path with the potential shows in blur. The weight of the space increases as it is closer to the obstacles in purple.

Difficulties and Future Notice

We recommended if A* is used to test out the maneuverability of the car in the future, the algorithm should be

parallelized. This would ensure the algorithm to not be a bottleneck. The search space of the algorithm could grow very large if the search space is a test range. We knew this was a parallelizable operation, since the next nodes can be processed in parallel.



XIII. ROS Packages For Communication Among Components

Sam Truong, Thanh Le, Adam Jones, Gabriel Lee

This quarter, we anticipated a need for ROS in order to provide a unified communication layer among our system components. Our idea was to calculate the steering angle using the lane line detection algorithm and publish the steering angle over a ROS topic. This way, the Raspberry Pi could subscribe to the topic and send the appropriate steering angle to the motor control board. Although we did not incorporate ROS as part of our final self-driving RC car, we were able successfully publish individual topics and subscribe to topics on separate machines, by publishing values from Sam's desktop and subscribing to those topics via laptop.

The reason why we felt we needed ROS was that our system setup at the beginning of the quarter differed from our system setup at the end of the quarter. Namely, at the beginning of the quarter, we envisioned the following pipeline:

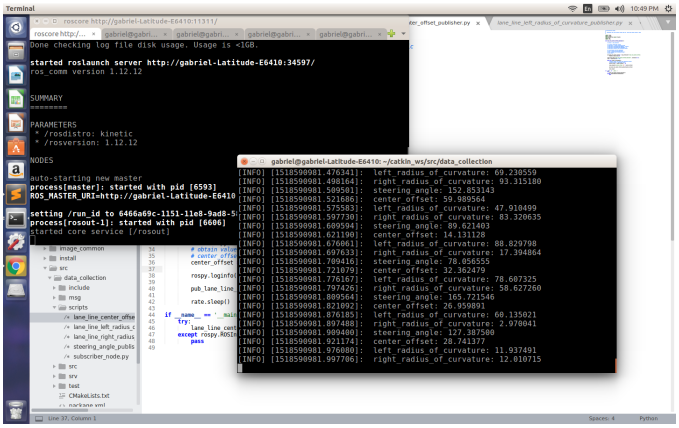
1) Produce image frame(s) from fish-eye camera connected to **Raspberry Pi**
2) Store frames on Raspberry Pi and publish frame(s) over ROS over a "/camera" topic
3) Subscribe to "/camera" topic on a machine with GPU processing capabilities (e.g. Sam's desktop or Jetson TX2)
4) Process frame(s) on machine, calculate steering angle based on lane line detection algorithm
5) Publish steering angle over ROS as a "/steering_angle" topic
6) Subscribe to a "/steering_angle" topic on Raspberry Pi to receive steering angle value
7) Send steering angle to motor board to turn the servo motor accordingly
8) Repeat process

In contrast, our current pipeline is a lot simpler:

1) Produce image frame(s) from fish-eye camera connected to **Jetson TX2**
2) Process frame(s) on Jetson TX2, calculate steering angle based on lane line detection algorithm
3) Send steering angle to motor board over a communication protocol such as UART, SPI (not used) or UDP (currently used) to turn the servo motor accordingly
4) Repeat process

Clearly, our current work-flow obviates the need for ROS, opting for a more ad-hoc, specialized approach to communication over a restrictive, bulky framework such as ROS.

Although we chose to not use ROS for communication in our final product, our team was able to publish outputs from the lane line detection algorithm as well as the steering angle over ROS, as seen in the following image:



Our published topics included:

- left radius of curvature (for the left lane line)
- right radius of curvature (for the right lane line)
- center offset (for the alignment of the detection coloring)
- steering angle (for the Raspberry Pi motor board)

These values, in addition to the actual video frame, are the outputs of the lane line detection algorithm. The image shows each topic being published by multiple publisher nodes and being subscribed to by a single subscriber node.

Apart from publishing values, we were also able to send/receive messages on multiple machines by connecting machines on the same network. Specifically, we were able to communicate between Sam's desktop and a laptop by setting up the master node on Sam's desktop and changing the laptop's ROS_MASTER_URI parameter to be the same as the master node's address/port. With our published topics, subscriber node, and ability to communicate across machines, we were ready to connect the lane line detection algorithm to the topics and begin porting our code to the Raspberry Pi.

Although we were able to group these topics into a single Data Collection package (consisting of a separate publisher node and a single subscriber node), our development code only worked on our laptop, and when we attempted to install ROS on the Raspberry Pi, we ran into Raspbian compatibility issues (ROS has mainly been tested for Jessie, not Stretch), particularly with the image transport libraries (image-common).

In particular, when we installed and compiled the image-common packages (needed for image transport among ROS nodes), we ran into segmentation fault errors while attempting to execute example code. We looked into the appropriate issues on GitHub (https://github.com/ros-perception/image_common/issues/49), but could not resolve the error. We tried to modify the OpenCV version in our CMakeLists.txt file as recommended, but this merely resulted in compilation errors.

To deal with this, Thanh and Adam re-installed Raspbian Jessie and Raspbian Stretch multiple times on the Raspberry Pi, ultimately setting with Raspbian Stretch, in order to get the image-common code working. We ended up going with Raspbian Stretch because it was easier to install dependencies and libraries on. Eventually, we realized that we didn't need ROS (we potentially didn't even need the Raspberry Pi), and halted all ROS-related development. As mentioned in the previous sections, we decided to transport frames from the Jetson to the Raspberry Pi using UDP instead of SPI or UART, due to its favorable speed.

Difficulties and Future Notice

The main issues we ran into with incorporating ROS in our project involved installation and compatibility errors on our Raspberry Pi. As mentioned above, we had great difficulty in getting our image-common packages to work properly with our example executables, which we needed in order to process the frames (according to our old pipeline).

Although we did not use ROS this quarter, in the future we can add it as a alternative/backup method of sending data among components, assuming that all components have access to the same network. Furthermore, ROS is a more scalable, modular method of communicating that would allow us to perform analytics for the system as a whole. This would be immensely helpful for optimizing the car and analyzing its performance, because we would simply have to subscribe to all the system topics instead of manually checking each communication interface. We would be able to aggregate topic data in a single location (e.g. a GUI) and quickly add new topics for combined analysis. As the complexity of our car increases, so will the usefulness of having a modular, unified communication system among all components.

We also had to re-flash the Raspberry Pi at least 12 times. It took over a week just to get all the dependencies and libraries that we needed on the Raspberry Pi working properly.

Acknowledgement: The Titan XP used in this research was donated by the NVIDIA Corporation. We also acknowledge the REU support from NSF CCMI-1301496 grant.