F1Tenth Vehicle: Miniature City Autonomous Driving*

Xinhang (Owen) Ma
m.owen@wustl.edu

Sirui (Ryan) Chen
c.sirui@wustl.edu

*Supervised by Dr. Eugene Vorobeychik

Contents

This report outlines our progress and records the current state of the F1Tenth vehicle that is being designed and tested in the miniature city on the third floor of McKelvey Hall. Previously, two teams contributed to this project. George Gao was responsible for the hardware and initial functions, while Angelo Benoit and David Brodsky enhanced the algorithms and improved the image processing and lane-detection system. References to their original documentation can be found in Section 5, where we also chronicle their contributions. Our aim with this document is to consolidate past developments, and to offer a comprehensive reference for future researchers.

1 Instructions and System Components

1.1 Instructions

Here are the necessary steps to get the car up and running.

1. Install NoMachine client (https://www.nomachine.com/) on your computer. This is a free and easy remote desktop software that will allow you to gain access to the car's onboard computer, as long as you are both on the same WIFI network.

2. Currently, the car is set to connect to the wustl-encrypted-2.0 campus wide WIFI if it is available. If this is your first time connecting to the car, have it within the WIFI coverage, turn on the car by connecting it to the power bank (press the power button on the power bank if needed), wait for about a minute for the car to boot up. (Important note: the car will always be turned on immediately after it is connected to power.) Next, connect your PC to the same WIFI, and only then do you open up your PC's NoMachine client.

3. Ideally, you should see a PC icon on the main screen of your NoMachine client. In this case, simply double click on the icon. If you are unable to detect the car's NoMachine server, find yourself a monitor, mouse, and keyboard. At the Jetson Xavier NX computer located at the very center of the car, plug in the HDMI cable connected to the monitor, and plug in the connectors of the mouse and the keyboard. Then, turn on the car by providing it with power. You should now have control over the car's computer via the monitor, mouse, and keyboard. Then, to figure out what the car's current IP is, either types commands like `ifconfig` or `hostname -I` etc. in the terminal; or open the NoMachine client on the Jetson, and click "Settings", and then "Status". Once you figured out the car's current IP, you can click on "add" in NoMachine's homepage (on your own computer) to add a static connection. *Speaking from experience, the car's IP is occasionally reconfigured, so it is better to always have access to a monitor, keyboard, and mouse. Additionally, sometimes it is more convenient and efficient to work on the Jetson directly.*

4. ***Important:*** the username is **yvxaiver** and the password is **nvidia**.

5. Once you're connected to the car's computer, we can start the setup process. First, ensure the car's drive battery is fully charged. Note that the car's motors use power from the

drive battery, not the power bank on top. The power bank only powers the car's computer and its sensors. If the drive battery seems low, charge it using its designated charger. See a picture of the drive battery and the charger in Figure 1.

Figure 1: drive battery (left) and its charger (right)

6. Once the battery is charged, connect it to the ESC via a red-black wired Traxxas Connector located on the lower left belly of the car. You can place the battery into the belly of the car (it would not be very tight, but it does not fall out unless you are running terrains).

7. Ensure the Logitech F710 controller has a charged battery. The controller remains on as long as there's a battery, which can lead to battery drain. To test its power, press the vibration button on the front. If it vibrates, it has power. Next, repeatedly press the "mode" button until the green LED beside it turns off. Finally, verify that the input mode on the top of the controller is set to "X" and not "D".

8. Now that everything is ready, go to your car computer and open up 5 terminal windows. On separate terminal sessions, run the following four commands in sequence to initiate the entire image intake and processing pipeline:

    1. `ros2 run usb_cam usb_cam_node_exe --ros-args -p framerate :=60.0 -p image_width:=1280 -p image_height:=720`

    2. `ros2 run image_transport republish compressed /in/ compressed:=/image_raw/compressed raw out:=/raw_frame`

    3. `ros2 run image_processor image_processor --ros-args -p mode:=vanilla`

    The third step is for the Trajectory driving option[1], which involves starting the LaneNet lane-detection model and may require a bit of patience. Wait for windows to pop up and display the camera feeds and the processed images. Once these

[1] See more details about the two different driving options in Seciton 3.3

are visible, you can proceed with entering the next command to start the IMU nodes as well as the driver node. In the future, it is possible, and encouraged, to merge the two IMU nodes into one.

4. `ros2 run ros2_imu_package ros2_imu_node`

5. `ros2 run imu_filter_package imu_filter_node`

6. `ros2 run lanenet_driver lanenet_driver_node`

9. If you are running the car with the Controller option, replace the third step above by:

   3. `ros2 run image_processor image_to_command`

10. Finally, run the following command to start the drive stack of the vehicle:

    7. `ros2 launch f1tenth_stack bringup_launch.py`

    Now, if everything went accordingly, the drive stack should be getting commands to follow the nearest lane. However, the car is not yet moving. The car's drive stack has a deadman's switch, RB, for autonomous navigation activities. To activate lane following, hold down the RB button. If you release it, the car will stop.

1.2 Core Modules Overview

For information on the ROS nodes and topics, please refer to the documentation created by David and Angelo, which includes a detailed ROS nodes and topics graph. These elements are not covered in depth within this report because they should be quite straightforward once you're familiar with the basics of ROS and the car's architecture. It's important to note that the provided link takes you to the graph constructed in the summer of 2023. There have been updates since then, particularly regarding the nodes associated with IMU messages. These updates will be discussed later in this document. Furthermore, you can always build a realtime graph of nodes and topics by running the command: `rqt_graph`

Below we provide an overview of the important Python files on the Jetson. Note that if you don't intend to rebuild the directory, you may find the corresponding file in the `install` directory and test your codes without the need of rebuilding every time, but the path of those files are a bit more complicated.

For easier access, this repository, https://github.com/xhOwenMa/f1tenth/tree/main, contains all these important scripts. **Note:** the `imu` directory contains the source code for the two IMU nodes (driver and filter) while the other directory only contains Python scripts. Later in Section 3.4, we walk through the steps of creating a ROS2 node.

`camera_ws/src/image_processor/image_processor_node.py`

- this file handles feeding the camera frame into lanenet model, then converting outputs into path to be followed, and publishing that path.
- this is the most convenient file in which you can modify the code to enable saving the images for training the lane detection model. You can read more about how the lane detection model is trained in Section 4.

`camera_ws/src/image_processor/bridge/lanenet_bridge.py`

- arguably the most important file. The `image_processor_node.py` relies on the functionalities here to process the images.
- the `image_to_trajectory` function handles feeding the image to the LaneNet model and post-processing the outputs to publish the trajectory to be followed by the car.

`ros2_imu/src/imu_ros2_package/imu_ros2_node.py`

- this node handles retrieving the raw IMU data from the IMU device on the car.

`comfilter_ws/src/imu_filter_package/imu_filter_node.py`

- this node subscribes to the IMU data that is published by the above `imu_ros2_node`, then it filters the data into useful measurements of the car's acceleration and orientation.
- more details about these two IMU nodes are covered in Section 3.4.

`f1tenth_ws/src/f1tenth_system/lanenet_driver/lanenet_driver_node.py`

- for the Trajectory option, this node receives the trajectory to be followed and publishes the drive message (command) to the car.
- for the Controller option, this node receives the throttle and steer commands which are transformed into `ackermann_msgs` that control the car's driving.
- furthermore, this node subscribes to the IMU node and uses the IMU data to update the car with its velocity and orientation.

`f1tenth_ws/src/f1tenth_system/lanenet_driver/stanley_controller.py`

- this is important for the Trajectory option.
- receives the trajectory and calculates and outputs the velocity and steering angles based on its simulation of the vehicle dynamics.
- it is highly unlikely that you would need to modify the codes in this directory, but it could be helpful to know that important factors in driving are controlled here.

`camera_ws/src/image_processor/image_to_command.py`

- this is for the Controller option.
- handles the same function as the `image_processor_node` discussed above: preprocessing the image, feeding the image to the LaneNet model, fits the left and right lanes with third degree polynomial respectively, and uses a controller model to get the driving command with these data. More details in Section 3.3.

Having an understanding of these files should be enough to let you catch up to what have been achieved so far. The other files play a lesser role in the overall project, and it's quite unlikely that you'll encounter issues requiring modifications to those less critical files.

2 Hardware Architecture

2.1 Drive Stack

The car's driving related hardware and electronics include the following:

- Traxxas Rally 4WD R/C chassis. This includes all base structural parts of the vehicle, the traxxas motor, a servo motor, gear box, transmissions, suspensions, the drive battery, etc.
- Electronic Speed Controller (ESC) unit. There are two available ESCs:
  - there is a bigger one left in the box, which is the FLIPSKY FSESC 6.7 PRO based upon VESC6 with Aluminum Case. This ESC has a switch, Figure 2 below shows how to use it (the figure was taken when it was installed on the car): the red circle is where to connect the drive battery, and the red rectangle is the switch. Remember to turn it on after connecting the battery and also to turn it off before disconnecting the battery. The switch will shine a blue light when turned on.

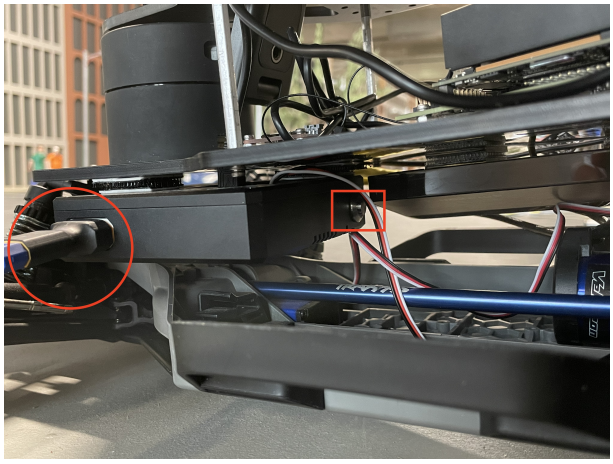

Figure 2: the alternative ESC, photo was taken when it was mounted on the car

  - the current ESC onboard is the FSESC 4.12 50A Based on VESC4.12. We've set both ESCs up in line with the F1Tenth official documentation. This guide offers a thorough overview of nearly all functionalities tied to the car's hardware components, and we have followed it rigorously to make sure the hardware is free of problems.
  - known problem: we have not perfectly configured the ESC's speed PID controller gains, this makes the ESC to not work smoothly at slow speed.

2.2 Perception & Processing Stack

The car's sensory and processing hardware include the following:

- Jetson Xavier NX. This is the brain of the car. Just before we concluded our work, we cleaned up the legacy codes and directories. Therefore, everything that is left in the `home` directory is more or less relevant to the functionalities of the car. Later in this document, we will talk about the various components of the autonomous driving pipeline in more details.

- Logitech c920 1080p Camera. This is the most important sensor on the car. In fact, currently all of the autonomous driving functionalities rely on the captured images from this camera. This can be a place of upgrade that is worth considering in the future for several reasons:

    1. We don't think the resolution provided by this camera is benefiting the lane-detection. On the contrary, we suspect that it may even be introducing unnecessary overhead into the real-time image processing pipelines as well as the data collection and annotation phase of model training. This may appear more evident as we discuss these topics later in more details.
    2. The field of view offered by this camera is relatively restricted considering the task it is doing. This limitation poses challenges for autonomous operations, especially on curved paths where the camera often misses the adjacent lane, which is essential for accurate trajectory prediction.

- YDLIDAR. Currently we are not using it.

- Power Bank. Make sure the Jetson is connected to the 20V output port.

- Amazon USB Hub.

- SparkFun IMU.

    – We have installed the Arduino IDE through which you can directly work with this device. Note that the firmware that is currently installed is linked with the driver node that receives the messages from the IMU, so it would be unlikely to ever need to do any more works with respect to this part. If you find yourself in the situation where you would need to update it, please refer to Section 3.4 for more directions.

- Logitech F710 Controller and USB Receiver.

While the performance of the car might be influenced by its hardware, the main focus of this project isn't on optimizing the electronics or hardware designs. If you encounter any issues with the hardware (as occasional failures can occur), try rebooting the system or rebuilding the `f1tenth_ws` directory; also, please feel free to reach out to us and we will do our best to help.

3 Software

3.1 Drive Stack

The drive stack serves as a translator, converting desired driving behaviors—like velocity, acceleration, and steering angle—into machine-level code. This code then communicates directly with the electronic speed controller, which in turn dictates the operation of the motors. For this project, George Gao had started with the widely-recognized open-source F1Tenth car drive stack that was developed by UPenn, and we only made minor adjustments to fine-tune the driving performance. No structural changes were made. *Essentially, this means that, if you suspect that there are problems with starting up the VESC or related components, it is an option to rebuild the whole* `f1tenth_ws` *directory*.

The original setup documentation is linked here: https://f1tenth.readthedocs.io/en/foxy_test/getting_started/firmware/drive_workspace.html#doc-drive-workspace. It is the same documentation that is mentioned above, we highly recommend that you bookmark this page so that you can easily access it at anytime.

For convenience, here is their original repository: https://github.com/f1tenth/f1tenth_system.

Use these resources to understand the workflow, but there is no need to reinstall or reconfigure anything. The downstream folder in the car is located at `~/f1tenth_ws`. We will discuss more about the specifics later after we introduce every component. For now, simply remember that command 5 from the Instructions (Section 1.1) will launch all the hardware components here.

3.2 Perception Inputs

The images are taken by the Logitech c920 USB camera. To handle the camera inputs, the current pipeline does two things: first, it decodes input USB data into usable frame; second, it compresses frame information for faster ROS intra-communication transport.

More specifically: to decode USB camera input, we used an ROS2 package called "usb_cam"; its original repository can be found here: https://github.com/ros-drivers/usb_cam/tree/ros2. Downstream in the car, it is located at `~/camera_ws/src/usb_cam`. To start this process, call the first command listed in the Instructions. There is no need to modify the codes in this directory unless you are absolutely certain what you are doing.

Then, to compress images for transport, we used a ROS2 package called "image_transport" (http://wiki.ros.org/image_transport) to republish compressed images. To start this process, call the second command listed in the Instructions.

3.3 Lane Detection & Driving

Now we have managed to get the image frames from the camera, we will need to process them to detect where the lane markings are located. After that, we can estimate the trajectory to be followed.

The lane detection model used is LaneNet (Neven et al. (2018)). Figure 3 below is the basic network architecture: Please refer to their paper for more details about how the model works.

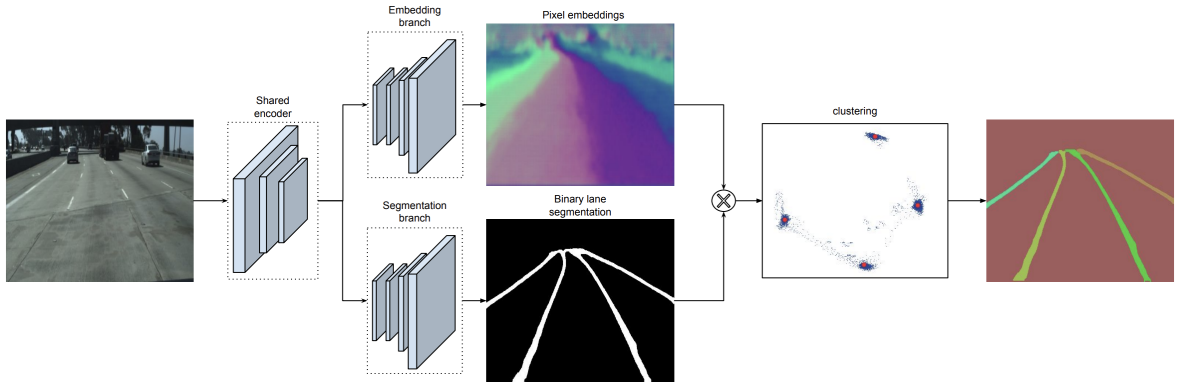

Figure 3: LaneNet architecture from Neven et al. (2018)

Later in Section 4 we discuss the details of how we trained the model as well as the directions to train your own models. For now, to see the model in action, run the third command listed in the Instructions.

To integrate LaneNet into the autonomous driving pipeline, we currently have two separate systems:

3.3.1 Trajectory

This relies on processing the image frame to detect the left and right lanes from which the trajectory is estimated. For this purpose: we have developed a ROS2 node named `image_processor`. It's important to note that `image_processor` node heavily utilizes methods from the `lanenet_bridge.py` script for the actual image processing and lane detection tasks. The whole process works as follows:

Finding the lane points from the LaneNet model's outputs:

1. the model outputs `instance_segment` and `binary_segment` are used for these post-processing steps.
2. apply a mask over the instance image to filter out the blue background. Then, perform a bitwise 'or' operation between the mask and the binary image.

3. Clean the mask to remove noise: first uses opencv's morphology function to fill to reduce the small area; then performs a connected components analysis and eliminate small components.

4. check for straight lanes: identify continuous horizontal lines of white pixels; then output the median of these lines as a lane point.

5. check for curved lanes: identify continuous vertical lines of white pixels; then output the median of these lines as a lane point.

6. for all lane points, we ignore single points and points above $y = 520$ pixels.

Finding and Sorting the Lanes:

1. Warp each lane point identified in the previous steps.

2. Sort the lane points by their y-coordinate, so that the processing starts from the topmost (smallest y-values) points in the image and proceeds downwards (increasing y-values). This is based on the observation that the lanes that are closer to the camera (bottom of the image) are wider and more spread out than the lanes further from the camera (top of the image). Processing from top to bottom can help in sequentially identifying and tracking lanes as they widen. Hence, starting from the top can ensure that the initial point sets the direction or foundation for a lane, and subsequent points either belong to this lane or form new lanes based on the proximity or separation parameters.

3. to organize the points into lanes, we base our algorithm on the following conditions (the parameters may be changed to adjust the performance):

    - Set a lane separation parameter of 50 pixels because consecutive lane points should have similar x-values.
    - set a maximum y gap parameter of 25 pixels to allow for small gaps in lanes due to mispredictions.
    - Define new lanes starting below y=100 as valid lanes.
    - For lanes starting above y=100, set a new lane x parameter of 25 pixels to determine the start of a new lane, that is, the start of this kind of new lane must be within 25 pixels of either the left or right edge of the image.

4. For each point, add it to an existing lane, start a new lane, or ignore it, based on the conditions above.

5. Discard lanes that contain too few points (this parameter is set to 10).

Fitting lanes & Finding the left and right lane:

1. Represent each lane as a polynomial in terms of y.

2. Determine whether each lane follows a linear or cubic polynomial.

3. Choose the linear model if the error of the linear fit is less than 8 times the error of the cubic fit. This factor (8) is hardcoded and can be adjusted experimentally.

4. To identify the left and right lane: go through each lane, focusing on the point at the bottom (closest to the car). If this point's x-coordinate is greater than the car's x position, classify this lane as the right lane. The lane immediately before it should be identified as the left lane. This approach works because we've already sorted the lanes from left to right in the previous step.

After we have the left and right lanes, the centerline trajectory is estimated via Algorithm 1: Line 2: the points are evenly spaced from the bottom of the image frame to the end of L_l or L_r depending on which one is smaller.

Algorithm 1 Trajectory Estimation

Input: L_l, L_r

Output T

1: get the polynomial fit of L_l and L_r from *Poly_Lanes*
2: generate 20 evenly-spaced points along L_l and L_r
3: compute the x-coordinate of the midpoint for each generated pair of points
4: compute the y-coordinate of the midpoint for each generate pair of points
5: publish the midpoints as the trajectory to be followed

Now that we have the trajectory to which the vehicle should follow, the final part of the pipeline deals with how to automate the driving given the trajectory. To summarize, the trajectory is published in the form of vectors of points, George Gao had defined a ROS2 message for this type, for reference: https://github.com/ggao22/points_vector.

However, there is a subtle difficulty: the car needs continuous commands for autonomous driving, however, although `image_processor` is working on lane detection and trajectory estimation all the time, there is an irreducible processing time that takes about 0.2-0.3 seconds. This means that there is a 0.2-0.3 second gap between trajectory updates, and therefore there are no drive commands during this processing time.

To address this problem, we need to rely on a simulator so that given a trajectory, the simulator can create its own simulation of the vehicle dynamics and control the vehicle to follow the trajectory. By doing this, the simulator will continuously (every 50 milliseconds) output the drive command as long as there is more trajectory to follow in its simulation.

3.3.2 Controller

This process is arguably easier but it has a black-box nature. Downstream in the car, the ROS2 node `image_to_controller` is the main script that governs this whole process: It

begins by preprocessing the image frame, then employs the LaneNet model to extract the `binary_segmentation` map. From this map, the left and right lane polynomials are derived. These are then fed into a controller model, which generates the driving commands. These commands are subsequently relayed to the `lanenet_driver_node` for actual driving.

Specifically, once we acquired the `binary_segmentation` map from the LaneNet model. The post-processing steps to identify the left and right lanes include:

1. enhancing the binary road segmentation result to improve visibility and display this enhanced image for visual inspection.
2. identifying lane pixels by searching for non-zero brightness values in the segmentation map. These bright pixels are assumed to represent the lanes on the road.
3. determining whether each lane pixel belongs to the left or right lane based on its horizontal position compared to the assumed car position in the center of the image.
4. For pixels meeting specific criteria (e.g., being on the left and below a certain vertical position), they are added to the respective lane masks. The reason for a vertical cut is based on the observation that irrelevant lanes start higher up in the image frame while the immediate left and right lanes start from the bottom of the image.
5. then we fit the left and right lane to a third-degree polynomial respectively. To achieve this, we created the `camera_geometry.py` script. This script calculates a grid that facilitates fitting the polynomial directly onto the binary segmentation map.

Once we've determined the left and right lane polynomials, their coefficients, along with the car's current and target speeds, are inputted into a controller model. This model, developed by Baiting Luo and Ayan Mikhopadhyay and trained on the CARLA simulator, outputs the necessary steering and throttle commands to control the car's movement. To make communication between the ROS nodes simpler, we have declared our custom ROS message: https://github.com/xhOwenMa/f1tenth/tree/main/core_modules/controller/drive_msg, which consists of the steering and throttle commands from the controller model. Finally, to ensure these drive commands are issued consistently, we employ the similar looping technique, executing them every 50 milliseconds.

3.4 IMU

Besides the ability to process the images, the other important component of the car is its knowledge of its speed and heading, for which we have developed a ROS2 node tailored to the onboard IMU (Inertial Measurement Unit) device.

First, here is the hookup guide from SparkFun regarding the specific IMU device we have: https://learn.sparkfun.com/tutorials/mpu-9250-hookup-guide/all. *Note: do* ***not***

use the library provided in this guide, it is not wrote correctly, we have set up the correct library and firmware already. This guide offers some insights into how you can work with the device, and, in case you want to update the firmware, provides instructions on how to update the library and firmware on the IMU device.

For reference, this section is based on the `imu` directory in our GitHub repository: https://github.com/xhOwenMa/f1tenth/tree/main/core_modules/imu.

3.4.1 IMU Driver Node

To retrieve the data from the IMU, the driver node (in `ros2_imu`) first subscribes to the `Imu.msg`, which is a `sensor_msgs` in ROS2. This subscription allows us to gather the raw data from the device. The code here is based on this repository: https://github.com/ENSTABretagneRobotics/razor_imu_9dof/tree/indigo-devel, which is written for ROS1. Hence we suggest that you learn the functions of the driver node directly from our code. Here we also walk you though how to build a ROS2 node (this tutorial: https://docs.ros.org/en/foxy/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Py-Publisher-And-Subscriber.html provides an additional example):

1. to keep things organized, create a workspace directory and the source directory by `mkdir -p ~/[your-ws-name]/src`, and navigate into the `/src` directory. You should always put your package into the source folder. Run the command:

   ```
   ros2 pkg create --build-type ament_python [your-package-name]
   ```

   to create the package with ament python build type and your-package-name.

2. Navigate into `your-ws-name/src/your-package-name/your-package-name`. Write your source code here (or paste desired Python script in here), and make sure that `__init__.py` is also here.

3. After you finish coding your package, navigate one level back to `your-ws-name/src/your-package-name` directory. Here you will find three files that are created for you:

   - `package.xml`: add the dependencies corresponding to your node's import statements. For example, our IMU driver node imports the `sensor_msgs`, so we need to add

     ```
     <exec_depend>sensor_msgs</exec_depend>
     ```

   - `setup.py`: add the entry point by adding

     ```
     'your-node-name = your-package-name.your-python-script-
        name:main',
     ```

     in the console_scripts brackets of the entry_points field. For example, our IMU driver has the entry point

```
'ros2_imu_node = ros2_imu_package.ros2_imu_node:main',
```

- `setup.cfg`: this file should be populated automatically, its functionality is telling setuptools to put your executables in `lib`, because `ros2 run` will look for them there.

4. Now you are ready to build your package. First navigate into the root of your workspace and check for missing dependencies by running

   ```
   rosdep install -i --from-path src --rosdistro foxy -y
   ```

   Then, still in the root of the workspace, build your package:

   ```
   colcon build --packages-select your-package-name
   ```

   Note: you can choose to build your package with the additional suffix `--symlink-install`, which will allow your future modifications to the source code to automatically apply.

5. Now open a new terminal, navigate to your workspace, and source the setup file:

   ```
   source install/setup.bash
   ```

6. Now you can run your node by

   ```
   ros2 run your-package-name your-node-name
   ```

Going back to our IMU driver node, some additional notes regarding its current state:

- This is an important tutorial that covers all aspects that are crucial for the correct functionalities of the IMU device: https://github.com/Razor-AHRS/razor-9dof-ahrs/wiki/Tutorial#sensor-calibration.
- Currently, the IMU device is not calibrated, and you should follow the above tutorial to calibrate it.
- You can also launch the driver node with hyperparameters. This launch file is similar to the one in the ROS1 driver. Remember to create the yaml file referenced there.

3.4.2 IMU Filter Node

After the driver node gets the raw data from the IMU device, we apply a filter to transform the data into useful measurements. Our code for the filter node is based on the complementary_filter that is included in this repository: https://github.com/CCNYRoboticsLab/imu_tools/tree/noetic/imu_complementary_filter. For our purpose, we simply adopted the logic to convert the x-direction acceleration into speed. In the future, you can modify this filter node to make use of all the other measurements provided by the IMU device. For example, a very useful measurement is the horizontal movement of the car: with this information, we can give

the car a real-time command to adjust its steering to make sure that it stays in the middle of the lane.

After the raw data is filtered, the node publishes this refined information. The `lanenet_driver_node` then subscribes to this data, utilizing it to update the car's position and velocity. This process ensures that the car has precise and up-to-date information regarding its movement.

3.5 Driving Evaluation

To evaluate the car's driving performance, we utilized the detected left and right lanes as a basis for objectively measuring the car's position relative to the center of the lane. This assessment operates under the assumption that the car begins its driving positioned in the middle of the lane. Algorithm 2 provides an overview: Line 3: *cwc* is a temporary variable name which stands for 'car with respect to center'. Line $7-11$: the intuition here is very simple, essentially, *cwc* represents the difference between the car's distance to the right lane and its distance to the left lane. An increasing *cwc* value indicates that the car is moving closer to the left lane, and conversely, a decreasing value suggests a shift towards the right lane. In the future, this method can be used to continuously monitor and adjust the car's autonomous driving system to fine-tune it to maintain optimal lane following.

Algorithm 2 Driving Evaluation

Input: car_x_pos, L_l, L_r, $image\_index$

1: get the most bottom point p_l from L_l
2: get the most bottom point p_r from L_r
3: compute $cwc = (p_r[x] - car\_x\_pos) - (car\_x\_pos - p_l[x])$
4: **if** $image\_index$ is 1 **then**
5: store cwc to the attribute $self.car\_wrt\_center$
6: **else**
7: compare cwc to $self.car\_wrt\_center$
8: **if** $cwc > self.car\_wrt\_center$ **then**
9: the car is steering to the left
10: **else**
11: the car is steering to the right
12: **end if**
13: **end if**

4 Lane Detection Model

We adopted the model that was trained by David Brodsky and Angelo Benoit and retrained it further on additional images. For reference: their repository (link) was modified based on this repository. The mentioned functions in this section are all based on their repository. Additionally, all the training data can be found here: `https://drive.google.com/drive/folders/1pTW1RPH19Qc8EysiyhTDlCBtD6bV9aTh?usp=drive_link`, which contains approximately 1000 images. In Section 4.1, we describe how to collect and annotate images; in Section 4.2, we provide instructions to how to run the above linked repository to train your model.

Below is the environment we used for training:

- Albumentations 1.3.1
- Numpy 1.23.5
- OpenCV 4.7.0.72
- Torch 2.0.1
- Torchmetrics 0.11.4
- Torchvision 0.15.2

4.1 Data Collection

1. To collect the images, it is best to modify the `image_processor` or the `image_to_command` script. Both scripts interact with the original image frame, and you can leverage the functions provided by opencv to easily save them.

2. If the data is used for transfer training, make sure the camera position and angle is the same as it was in the old data.

3. Resize the images to 512×256. This should be done before annotation.

4. After collecting the images, we used the VGG Image Annotator to annotate the images. The online version is sufficient for our needs and is simpler than the downloaded application.

5. Practical tips for image annotation:

   (1) Use the `resize_images.py` function that is included in the data_creation folder in the repository or use any other methods to resize images to 512×256. This must be done before you annotate the images.

   (2) Use the 'polyline' option for tracing lanes, pressing Enter to finalize each lane.

   (3) Utilize the 'select all' option for any modifications, as selecting individual lanes can be challenging.

   (4) For curves, draw polylines as accurately as possible to prevent the model from breaking a single lane into multiple segments.

(5) Export annotations in CSV format. Ensure you do not rename the images after this, as the CSV file correlates each line with an image.

4.2 Training

To train the model after images and annotations are ready, follow these steps:

1. In the data creation folder, use `polyline_annotator.py` to create instance images.
2. Use `relabel_images.py` to add the number of lanes to the instance images.
3. Then, use `create_binary_images.py` to create the binary images from the instance images.
4. Finally, use `split_data.py` to split the binary images into training and testing set.
5. If you are not sure about the resulted data structure, please refer to the structure in data/-training_data_example/. The two text file there are also needed, which should be created in step 4 above. Make sure the path inside them are correct.

For training implementation details:

- Given the limited size of our dataset, we used the Albumentations library to augment the dataset. Here is a link to its documentation: Albumentations documentation. You can find the augmentation steps we are using in the `train.py` script. All parameters and step choices can be modified for better performance.
- in `model/utils/cli_helper.py`, explains all possible arguments for the training script. Default values are provided, and the script takes care of training dataset paths and save paths. Essentially, this means that you can run `train.py` directly, but remember that hyperparameters are crucial to the model's performance.
- Finally, All necessary libraries are listed in the README file of the repository. Once your environment is set up, you can begin training the model!
- If possible with enough memories on gpu please change batch size from 16 to 32.

5 Archive

This section is dedicated to acknowledging the significant contributions made by previous groups who have worked on this project.

- George Gao: documentation. George Gao constructed the hardware foundation and introduced various algorithmic ideas into this project.
- David Brodsky and Angelo Benoit: documentation. They refined and enhanced the software algorithms originally introduced by George Gao.

Our recommendation for engaging with these documentations from the previous groups is a two-step approach: First, you should familiarize yourself with the overall pipeline, drawing upon the insights from this document and your hands-on exploration of the car's system. This initial step will provide you with a foundational understanding of the project's workflow and its current state. Once you have a solid grasp of the process and its components, we suggest investigating into the earlier documentations, paying particular attention to the sections where the previous teams highlighted potential areas for improvement. The reason for this approach is that while this document offers the most current and comprehensive information, understanding the context and evolution of the project can help you identify opportunities for future enhancements.

References

Neven, D., De Brabandere, B., Georgoulis, S., Proesmans, M., and Van Gool, L. (2018). Towards end-to-end lane detection: an instance segmentation approach. In *2018 IEEE intelligent vehicles symposium (IV)*, pages 286–291. IEEE.