Lab Guide
Multi-vehicle self-driving

Table of Content

Lab Description

In this example, a self-driving stack will be deployed to multiple cars, and they will make driving decisions independently while avoiding obstacles. In each stack, RGBD camera feed will be captured, and YOLO segmentation model will be used to detect objects of interest. In addition, LiDAR data is used to localize the QCar to follow pre-defined paths. The pipeline on each QCar is shown in Figure 1.

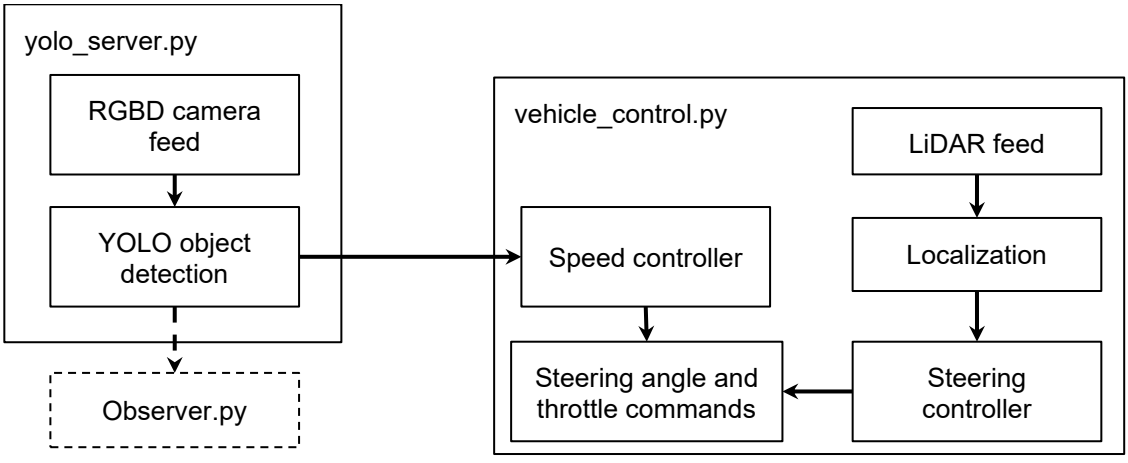

Figure 1. Component Diagram

Content Organization

This multi-vehicle self-driving example contains this document, a configuration file, 3 bat files, and 3 directories that contain additional scripts. The details of these files are provided below.

config.txt

The configuration file contains the following parameters for this research example:

1. REMOTE_PATH: The directory in the QCar, to which the python scripts are deployed to. For example, to set the remote directory to "*Documents*", configure the parameter as follows:
   `REMOTE_PATH=/home/nvidia/Documents`

2. QCAR_IPS: A list of IPs of an arbitrary number of QCar that will be used in the example. The list of IPs should be contained by square brackets and separated by comma. For example:
   `QCAR_IPS=[192.168.2.23,192.168.2.4]`

3. CALIBRATE_IP: The IP of the QCar that will be used to generate a calibration scan. For example:
   `CALIBRATE_IP=192.168.2.4`

4. PROBING_IP: The IP of the QCar to probe for its camera feed annotated with YOLO detection results. If you do not wish to probe any QCar, you can set this parameter to 0. For example:
   `PROBING_IP=192.168.2.4`

5. LOCAL_IP: The IP of the local machines, such that the probed QCar can stream back the annotated feed. For example:
   `LOCAL_IP=192.168.2.2`

6. WIDTH: The desired width of the probed camera feed. For example:
   `WIDTH=320`

7. HEIGHT: The desired height of the probed camera feed. For example:
   `HEIGHT=200`

Note: there must not be any space when configuring the parameters.

ssh_setup.bat

This bat script creates and adds SSH host keys of the QCar IPs specified in the configuration file to the local SSH **known_hosts**. This setup step only needs to be run once for new QCar used in the example.

run_all_cars.bat

Double click the **run_all_cars.bat** to run the example. All files in the **/qcar** directory will be downloaded to each QCar, and relevant scripts will be executed.

stop_all_cars.bat

Executing **stop_all_cars.bat** will stop all python scripts and QUARC RT models on all the QCars.

/reference_scan

This directory contains **calibrate.bat** which creates a reference scan of the physical environment and stores it in this directory. This reference scan will then be used to localize the QCar.

/python

This directory hosts the python files to be deployed to the QCar:

1. **yolo_server.py** – this file process the camera feeds with YOLO segmentation model, and stream obstacle data such as classification and distance to **vehicle_control.py**.

2. **vehicle_control.py** – this file localizes the QCar using lidar data and receive obstacle information from **yolo_server.py**. After combining localization and obstacle data, steering and throttle commands are sent to the QCar motor.

3. **utils.py** – this file hosts helpful python classes used in this example, including streaming server and client, and the drive logic to process the YOLO data.

/qcar

This directory hosts all the local python scripts, which facilitates establishing SSH connection to the QCars, transferring the files to the QCars and receiving live feeds from the QCar:

1. **observer.py** – this will be run automatically when a valid probing IP is provided and receives the annotated camera feed from the probed QCar.

2. **calibrate.py** – this file transfer **vehicle_control.py** to the specified QCar (*CALIBRATE_IP*) and run it in calibration mode. After calibration, the reference scan is transferred to the local machine and distributed to all the QCars used in this example.

3. **start.py** – this file transfers all python scripts in **/qcar** to the QCars and execute **yolo_server.py** and **vehicle_control.py** on the QCars along with appropriate arguments.

4. **stop.py** – this file sends termination signals to all python processes on the QCars.

Run the Example

1. Ensure **config.txt** has the correct parameters.

2. Run **ssh_setup.bat**.

3. Calibrate the QCars by placing the calibrating QCar (*CALIBRATE_IP*) at **node 11**, as shown in Figure 2, and run **calibrate.bat**. If the physical environment changes, the calibration step should be done again.

4. Execute **run_all_cars.bat**. If an appropriate *PROBING_IP* is provided, a pop-up window will show the live annotated camera feed.

   Note: The first time you run this script, ensure the QCars are connected to the internet. This is necessary to download the pretrained YOLO model, which will then be converted into a TensorRT engine for faster inference. The whole process can take up to **20 minutes.**

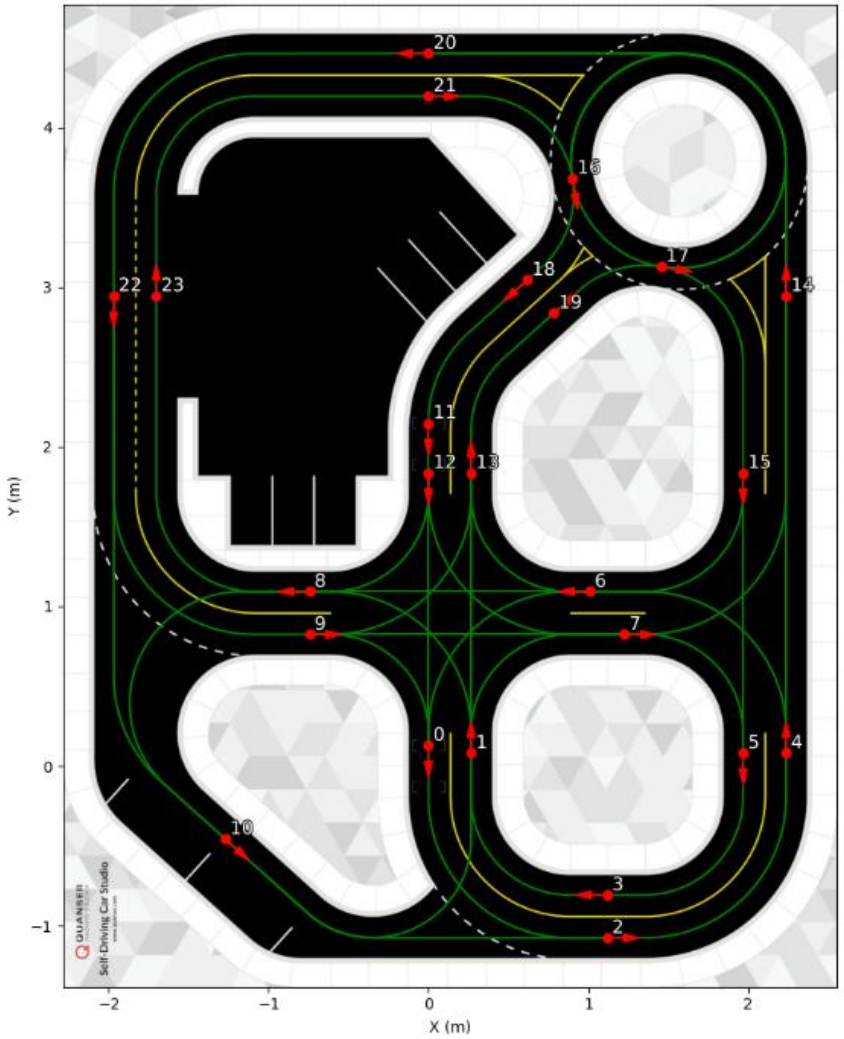

Figure 2. Pre-defined Nodes of the SDCS Mat.

Code Details

vehicle_control.py

1. Understanding Node Sequence

   When the script is executed, between 6 to 10 random nodes are chosen to compose the node sequence which auto-generates the path that the QCar will follow for the duration of this example. Valid path segments are in green, as shown in Figure 2. The initial location of the QCars can be anywhere on the green line, they will automatically follow a path leading to the start node before repeating the node sequence.

2. YOLO-based Obstacle Avoidance

```
yolo.read()
vGain = yoloDrive.check_yolo(yolo.stopSign,
                             yolo.trafficlight,
                             yolo.cars,
                             yolo.yieldSign,
                             yolo.person)
```

   After receiving the YOLO segmentation results from **yolo_server.py**, the helper class **yoloDrive()** processes the results and outputs a velocity gain, which will be used to dynamically adjust the desired speed for the speed controller.

3. Speed Controller

```
v = qcar.motorTach
u = speedController.update(v, v_ref*vGain, dt)
```

   A PI speed controller is used to maintain the speed of the QCar at desired level. The measure speed, *v*, is calculated based on the tach sensor output, the desired speed, *v_ref*, is defined in the beginning of this script, and loop time, *dt*, is calculated based on the difference between current timestamp and previous timestamp. Given these variables, the throttle command, *u*, is calculated.

4. Steering Controller

```
x = ekf.x_hat[0,0]
y = ekf.x_hat[1,0]
th = ekf.x_hat[2,0]
p = ( np.array([x, y]) + np.array([np.cos(th), np.sin(th)]) * 0.2)
delta = steeringController.update(p, th, v)
```

   An Extended Kalman filter was implemented to fuse measured speed, measured acceleration, and LiDAR-based localization data to yield accurate location, *p*, and orientation, *th*. A Stanley lateral controller is implemented to keep the vehicle on the desired path. Given the estimated pose, *p & th*, as well as measured speed, *v*, the steering angle command, *delta*, is computed.

yolo_server.py

1. Initialization

```
myYolo = YOLOv8(modelPath = 'path/to/model',
                imageHeight= imageHeight,
                imageWidth = imageWidth)
```

To initialize the YOLOv8 model, an optional path to a pretrained model can be provided. Otherwise, the default yolov8s-seg model trained by Quanser will be loaded, which can reliably detect other QCars. In addition, height and width of the input video stream should be provided, as the model will be optimized for that specific input size. After initialization, an **ultralytics.YOLO()** object is created and stored under the **myYolo.net** attribute.

When the script is being executed for the first time, the default model will be downloaded and then converted to a TensorRT engine for optimized inference time. The entire process can take up to 20 minutes.

2. Pre-processing and Prediction

```
rgbProcessed = myYolo.pre_process(QCarImg.rgb)
predecion = myYolo.predict(inputImg = rgbProcessed,
                           classes = [0,2,9,11,33],
                           confidence = 0.4,
                           half = True,
                           verbose = False
                           )
```

Before prediction takes place, the input image needs to be pre-processed into dimensions specified during initialization (if it is not so already).

The processed RGB image should be used as input for the **predict()** method. The argument **classes** specify the class IDs of the objects that should be included in the prediction. In addition, **confidence** defines the minimum confidence the detected objects, and **half** controls whether the model should predict using half floating-point precision, which can improve inference speed with minimal impact on accuracy.

The class IDs of the objects used for this example are [0,2,9,11,33] which represents person, car, traffic light, stop sign and yield sign respectively.

3. Post-processing and render

```
processedResults=myYolo.post_processing(alignedDepth = QCarImg.depth,
                                        clippingDistance = 5)
# annotatedImg=myYolo.render()
annotatedImg=myYolo.post_process_render(showFPS = True)
```

The **post_processing()** method takes an optional depth image as input to calculate distance of detected objects. This is done by applying the segmentation mask of each detected on the aligned depth image and computing the average depth value in these regions.

The **post_process_render()** method can only be used after **post_processing()** is called. Annotated images from **post_process_render()** also include distances to detected objects and traffic light status and is sent to the observer.