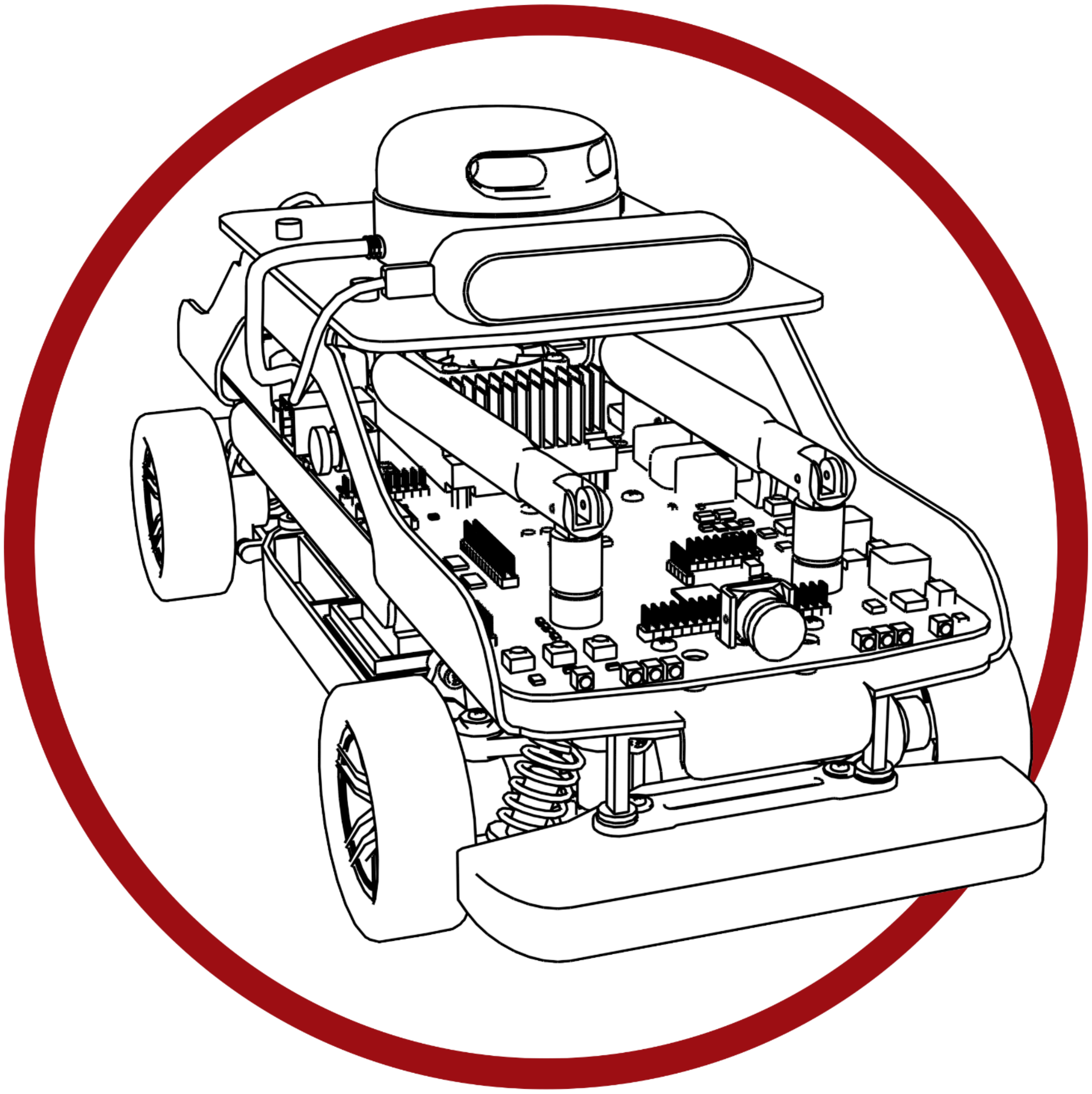
Self-Driving Car Research Studio



Autonomous Driving Car Example 1 - Simulink

V 1.0 (April 2021)

Table of Contents

I. System Description

In this example, we look at the application of autonomous lane following and obstacle detection using the QCar. The process is shown in Figure 1.

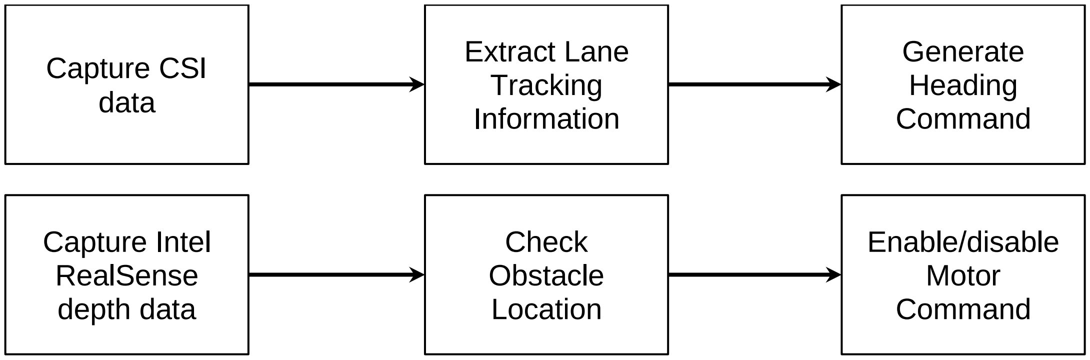

Capture CSI data
Extract Lane Tracking Information
Generate Heading Command
Capture Intel RealSense depth data
Check Obstacle Location
Enable/disable Motor Command

Figure 1. Component diagram

In addition, a timing module will be monitoring the entire application's performance. The Simulink implementation is displayed in Figure 2 below.

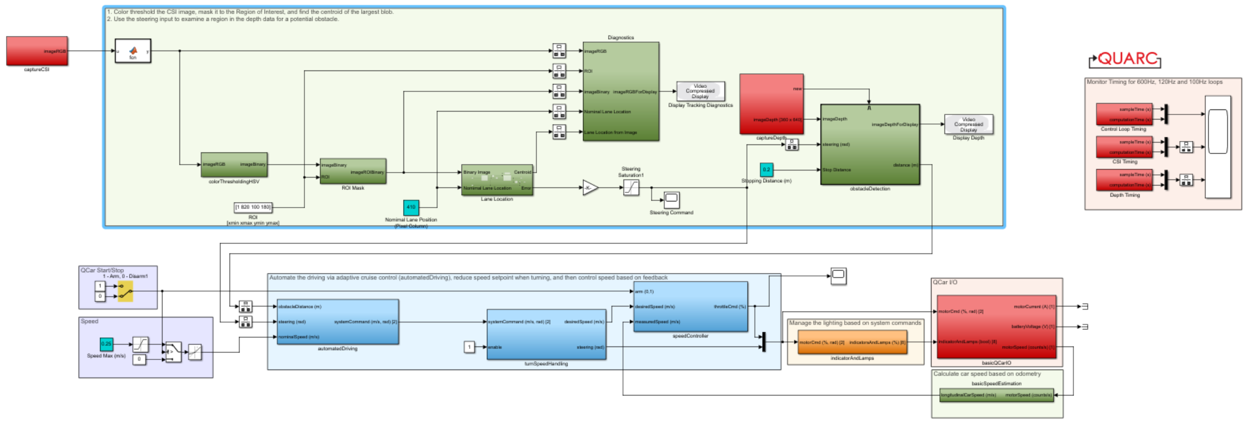


Figure 2. Simulink implementation of lane following with obstacle detection.

It should be emphasized that this is **NOT** a performant example. Please see the discussion throughout for tips on creating a more optimized system.

II. Running the example

Check the user guide **IV - Software - Simulink** for details on deploying Simulink models to the QCar as applications.

The following example can be run by configuring a continuous dual lane loop using the roadmap layouts as part of the SDRS which are an optional add-on of the SDRS ground station peripherals. Or even a loop of colored tape on a contrasting floor. Highly saturated (vibrant) colors for the line will produce the best results.

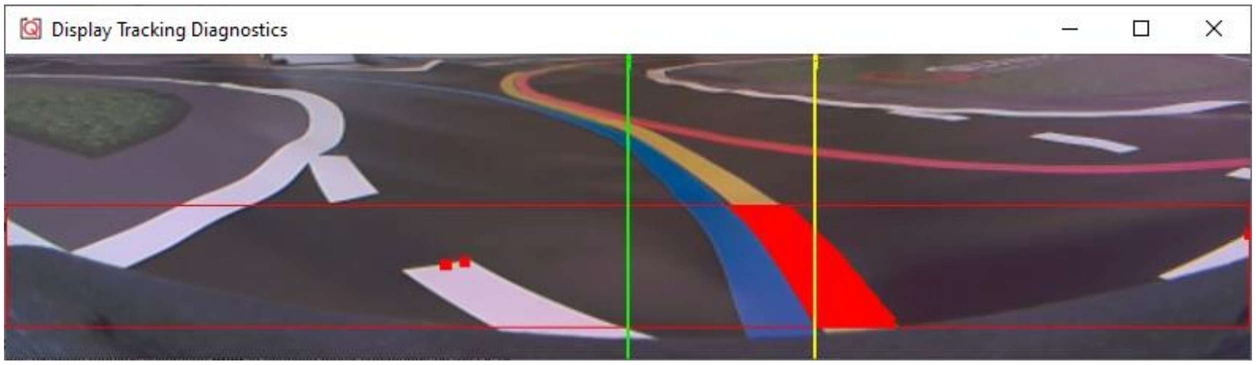
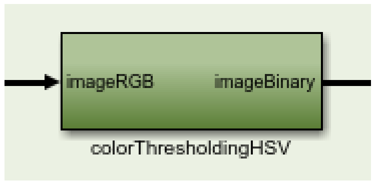


Figure 3. Processed lane extraction Image.

III. Details

1. colorThresholdingHSV



Color thresholding is done in two components. Component one converts the **imageRGB** input from the RGB to the HSV image plane. For in-depth information for the HSV image plane you can look at the **Image Color Spaces** document in the **3. Supporting Documentation directory**. Prior to identifying the regions of the image where a specific HSV values are present a subsystem generates the **HSVMin** and **HSVMax** values used to set the range for the specific color we want to select. Using the **ImageCompare** block we can generate a binary image which contains the portions of the image for which the selected color is valid.

To aid the tuning process (at the expense of performance), this example separates the HSV into separate planes so you can see the direct effect of each change to the separate HSV Min/Max values. The combined image is the logical and of the three color planes.

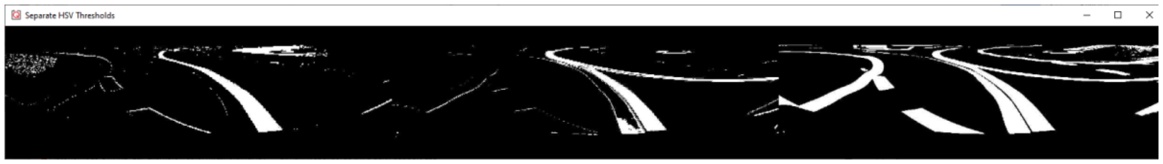


Figure 4. Separate Hue (color), Saturation (vibrance), and Value (brightness).

Following the HSV thresholding are separate Minimum and Maximum filters used to remove small specs of noise and fill holes respectively. The final image should be a relatively clean black and white image.

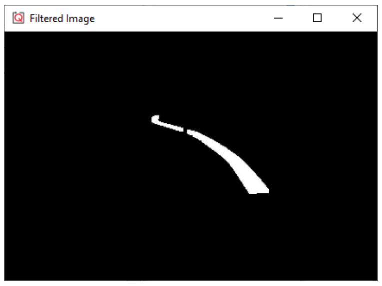
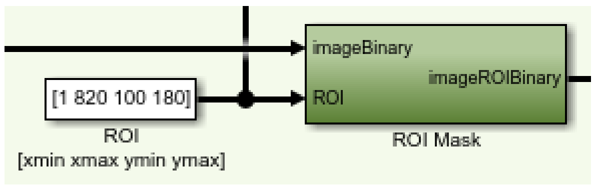


Figure 5. HSV thresholding combined with filtering.

2. Region of Interest

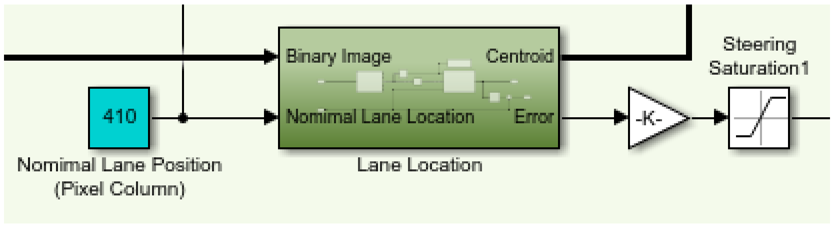


The image is further filtered by applying a logical AND of a rectangular mask. This is done with constants in this example, but more advanced examples could move the window to better target where the lane is expected to be located or move further up the horizon proportional to the speed of the vehicle to better support speeds exceeding 1 m/s.

Ideally the ROI should extract the region from the original image for processing and apply the HSV thresholding and all subsequent steps to a smaller sub-image (as is done in **Autonomous Driving Car Example 2**). In this example the entire lower-half of the CSI image is passed through the entire change to maximize flexibility and give you visibility into all the elements in the various processing steps, but this wastes substantial computational resources on areas that do not need to be processed.

Even more advanced approaches would use variable size images rather than fixed sizes allowing the ROI to be dynamic in both location and size, but this requires that all processing steps implement variable size support in their processing.

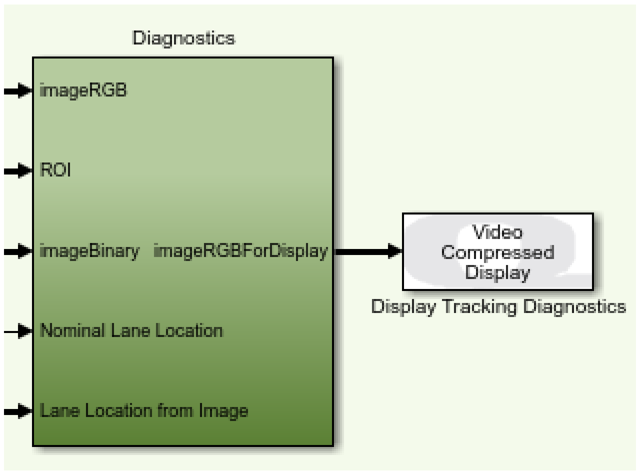
3. Lane Location and Steering



The lane location subsystem uses an Image Find Objects block which searches for blobs of a minimum size and then sorts them by size. The subsequent Matlab function block gets the centroid of the largest blob and the difference of that x pixel location from the nominal lane position is the steering error. A gain is applied to the signal outside the subsystem which is used for the steering angle.

A more advanced approach is to use a **linearPolyFit** function to get a lane trajectory (as done in **Autonomous Driving Car Example 2**) so as to better predict the heading of the lane rather than just its immediate location.

4. Diagnostics



The diagnostics section shows the camera view with the detected, masked area overlaid in red. The red rectangle indicates the ROI. The green line is the nominal lane position, and the yellow line is the current blob centroid.

The diagnostics block combining images and adding overlays is a significant draw on the computational resources. To reduce the impact, these been put in a separate sample time from the rest of the image processing. Ideally this, and any extra displays or scopes should be commented out to save the resources for additional operations.

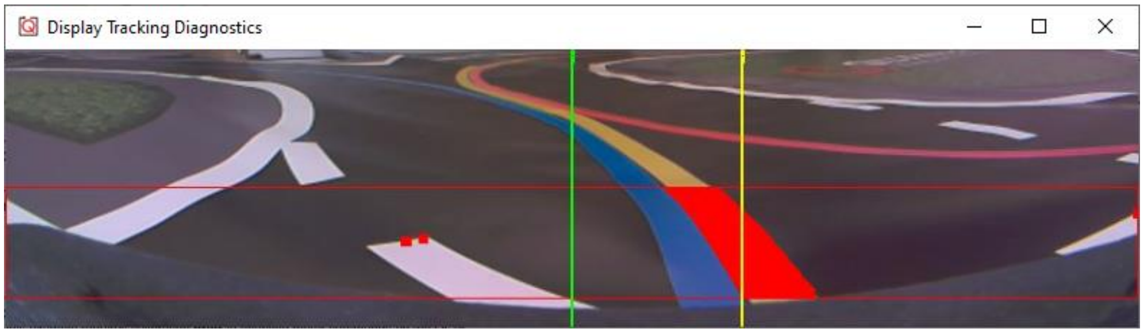
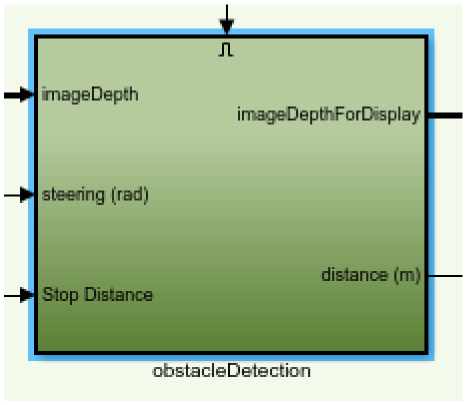


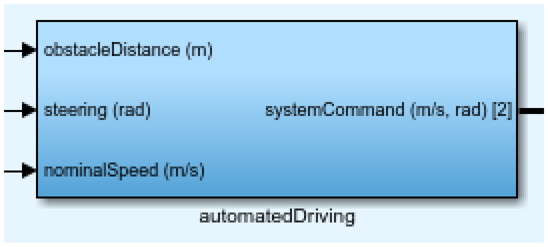
Figure 6. Tracking diagnostics display. Useful information, but computationally expensive.

5. obstacleDetection

The obstacle detection function uses an input depth image and the desired steering angle to extract a region of interest and sequence of points for the border of this region of interest. The depth information for the selected region is passed to the **calculateDistance** function. We find the region of points in the following interval **0.05m < depth < 2m** and calculate what the average depth is in the selected region. The input **stopDistance** lets us compare whether or not the average depth which we calculated is >= the stopDistance.

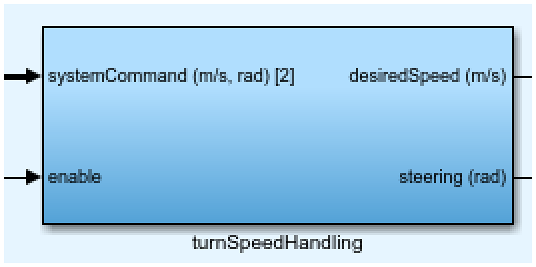


6. automatedDriving



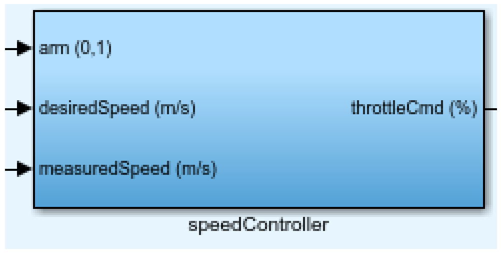
This controller subsystem will adjust the **systemCommand** desired speed based on a commanded **nominalDistance(m/s)** and **obstacleDistance(m/s).** Using a fixed stop_distance and nominal_tracking_distance the linear speed command is modulated such that the QCar slows down until the **stop_distance** is greater than **obstacle_distance**.

7. turnSpeedHandling



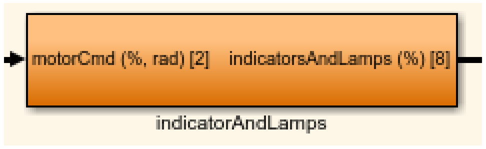
An enable constant is used to configure what the **desiredSpeed(m/s)** should be. We can pass the linear velocity command directly or evaluate the cosine of the steering to the power of 8. This secondary method will slow down the QCar closer to a turn and speed up during straight sections of road.

8. speedController



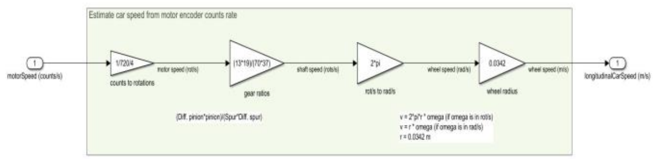
A **feedforward PI** controller is used to generate the desired **throttleCmd(%)** signal sent to the ESC of the QCar. The measuredSpeed of the QCar is compared to the desiredSpeed where the error term is converted from m/s to % via a proportional gain and m to % via an integral gain. To avoid integrator windup due to error accumulation over time the integral is reset using the arm signal which is also in charge of enabling the motor command. Lastly the error term is adjusted by a feed forward gain which converts the desired speed from m/s to %. By using a feedforward gain the controller command is no longer centered about zero but the desired setpoint defined by the feedforward gain.

9. indicatorAndLamps



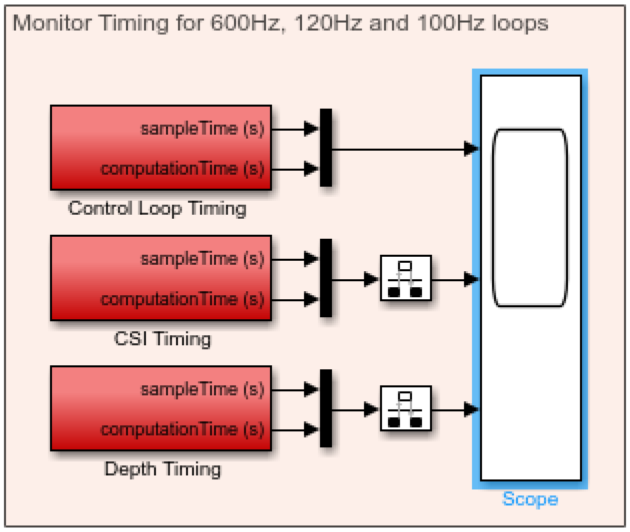
The logic inside this subsystem enables the LEDs on the QCar to act as a direction indicator. For the amber LEDs located at the front and the back of the QCar, these act as steering indicators. The **steering** is either **greater** than **0.3** for a **left** direction or **less** than **0.3** for a **right** steering indication. The rear left and right lamps are set to red when the QCar has a negative linear velocity while they are off during regular operation.

10. basicSpeedEstimation



The **motor encoder** on the QCar can give us **counts/s** which is passed through four scaling terms. The **first scaling term converts** from **counts/s to rotations/s**, a **second scaling term** passes the motor rotations through a gear ratio which **gives the wheel shaft rotational speed.** The third scaling term converts the shaft speed from rotation/s to rad/s and lastly the angular speed is multiplied by the wheel radius to get an estimate of the **logitudinalCarSpeed** (m/s).

11. Timing



If you choose to build on this example, monitor the timing scope as you make changes. Each graph shows the sample time for the respective timing rates and the computation time. If the computation time exceeds the defined sample time, then the same time will also increase. This can result in a sample loop running less than the expected rate and causing gaps in data when merging data through the rate transition blocks. If this occurs, you should either create a multi-step process to pipeline calculations or reduce the sample rate. In this example, the CSI cameras are set to run at 120Hz, but due to the less-optimal image processing implemented, the image processing loop was reduced to 60Hz. See **Autonomous Driving Car Example 2** for an example of the CSI running at the full rate.