Lab Guide
Lane Following

Content Description

The following document describes a lane following implementation in either python or MATLAB software environments.

MATLAB – Example 1

Lane following and obstacle detection using the QCar. The process is shown in Figure 1.

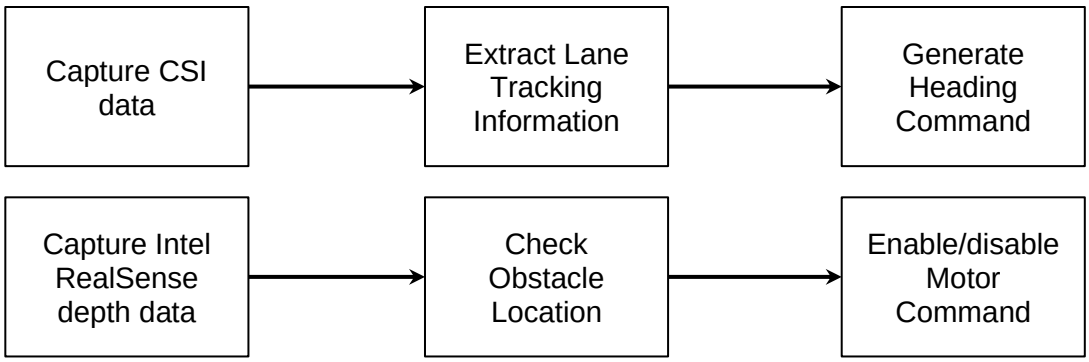


Figure 1. Component diagram

The Simulink implementation is displayed in Figure 2 below.

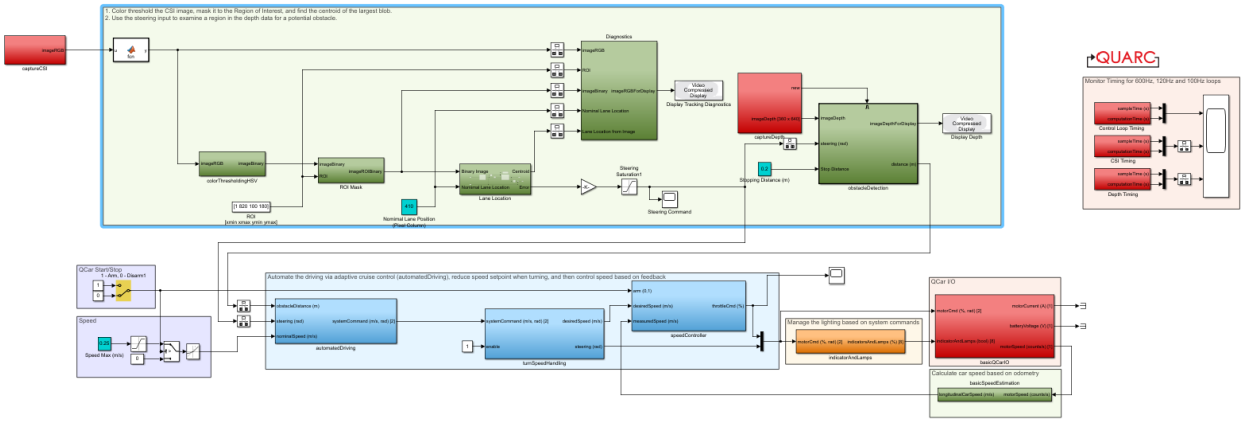


Figure 2. Simulink implementation of lane following with obstacle detection.

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

Running the example

Check User Manual – Software Simulink for details on deploying Simulink models to the QCar 2 as applications.

The following example can be run by configuring a continuous lane loop using the roadmap layouts as part of the SDCS studio. Highly saturated (vibrant) colors for the line will produce the best results.

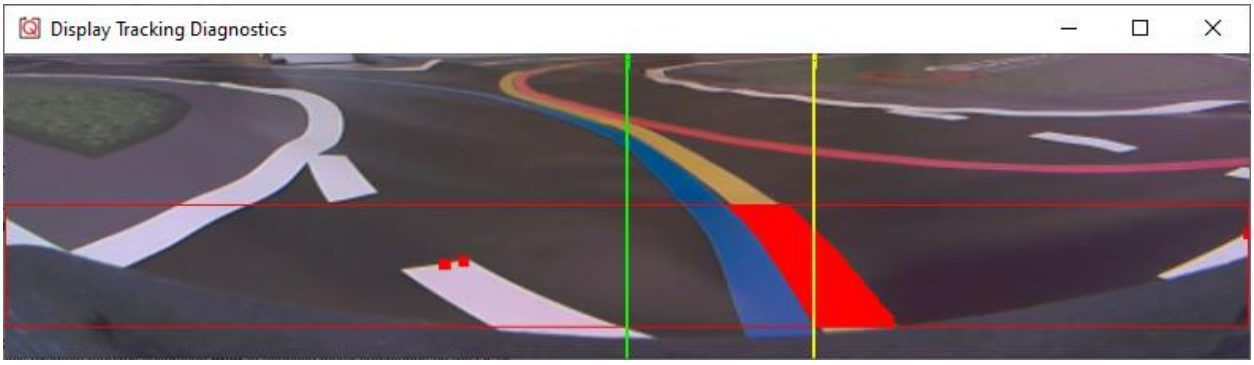


Figure 3. Processed lane extraction Image.

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 **Concept Review** 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 clean black and white image.

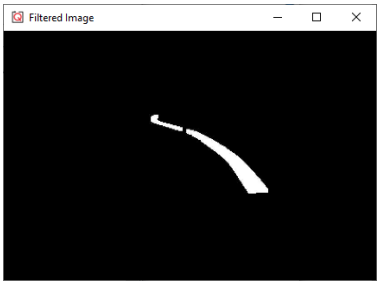
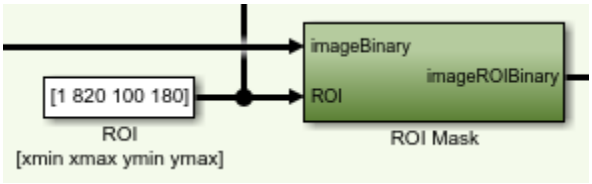


Figure 5. HSV thresholding combined with filtering.

2. Region of Interest

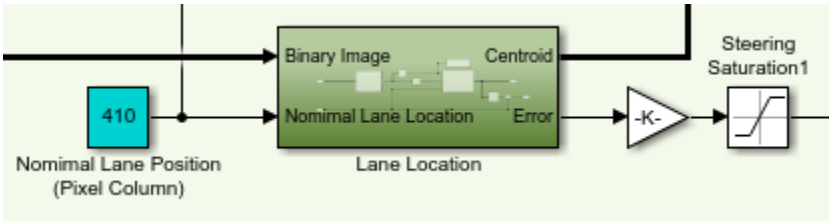


The image is further filtered by applying a logical AND of a rectangular mask. This is done with constants in this example, but more advanced examples could move the window to better target where the lane is expected to be located or move further up the horizon proportional to the speed of the vehicle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

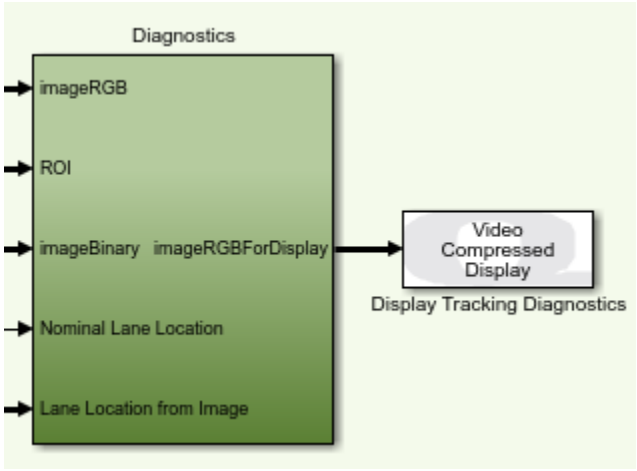
3. Lane Location and Steering



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**) 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.

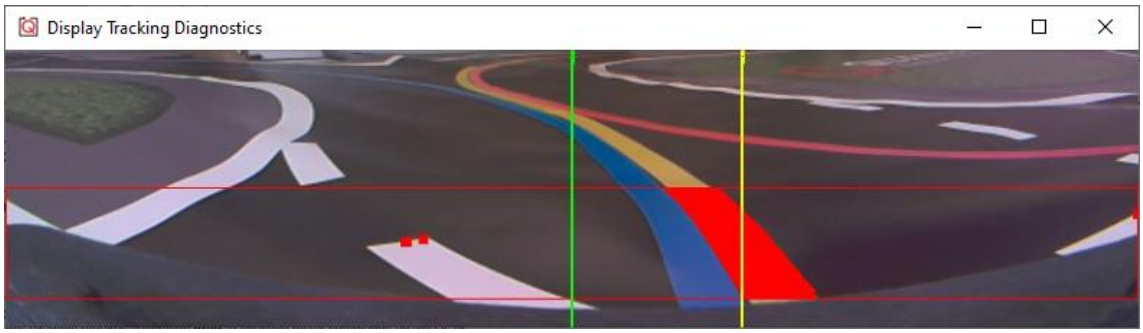
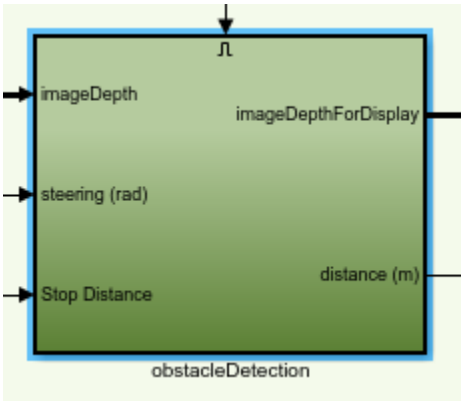


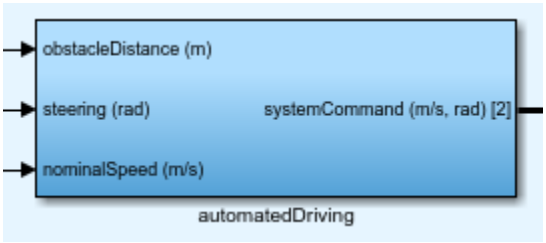
Figure 6. Tracking diagnostics display. Useful information, but computationally expensive.

5. obstacleDetection



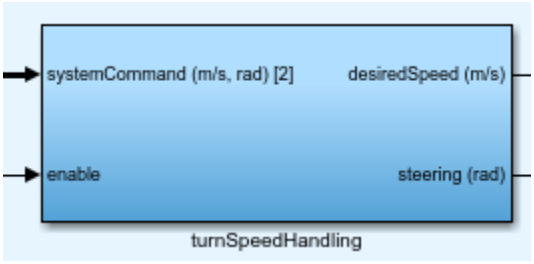
The obstacle detection function uses an input depth image and the desired steering angle to extract a region of interest and sequence of points for the border of this region of interest. The depth information for the selected region is passed to the **calculateDistance** function. We find the region 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 the average depth which we calculated is >= the stopDistance.

6. automatedDriving



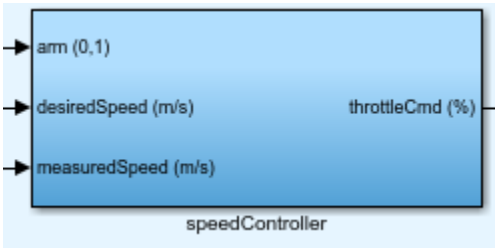
This controller subsystem will adjust the **systemCommand** desired speed based on a commanded **nominalSpeed(m/s)** and **obstacleDistance(m).** Using a fixed stop_distance and nominal_tracking_distance the linear speed command is modulated such that the QCar slows down until the **stop_distance** is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

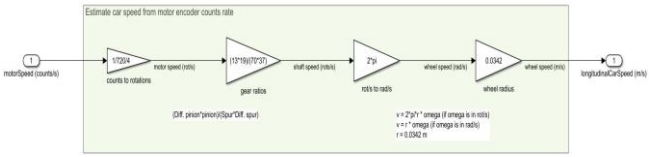


A **feedforward PI** controller is used to generate the desired **throttleCmd(%)** signal sent to the ESC of the QCar. The measuredSpeed of the QCar is compared to the desiredSpeed where the error term is converted from m/s to % via a proportional gain and m to % via an integral gain. To avoid integrator 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

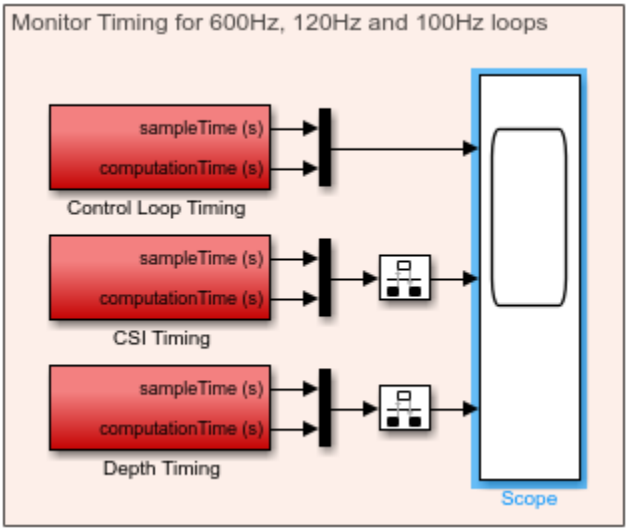
The logic inside this subsystem enables the LEDs on the QCar to function as a direction indicator. For the amber LEDs located at the front and the back of the QCar, these function as steering indicators. The **steering** is either **greater** than **0.3** for a **left** direction or **less** than **0.3** 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.

MATLAB – Example 2

In this example, we look at the application of autonomous lane following and obstacle detection using the QCar. The process is shown in Figure 7.

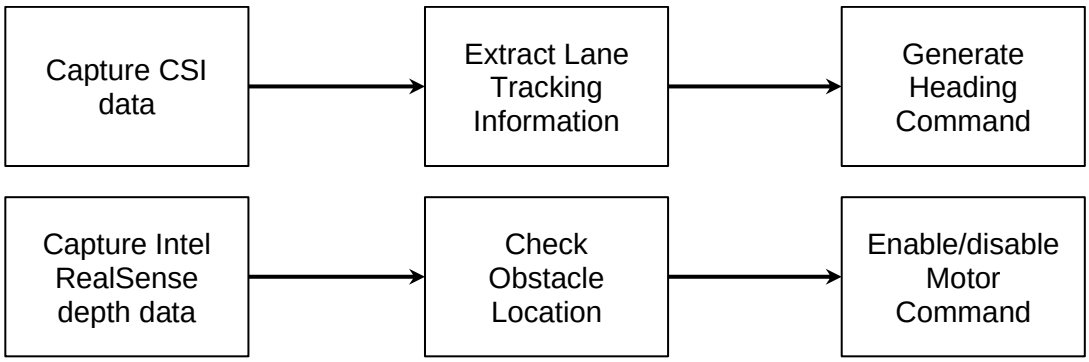


Figure 7. Component diagram

In addition, a timing module will be monitoring the entire application's performance. The Simulink implementation is displayed in Figure 8 below.

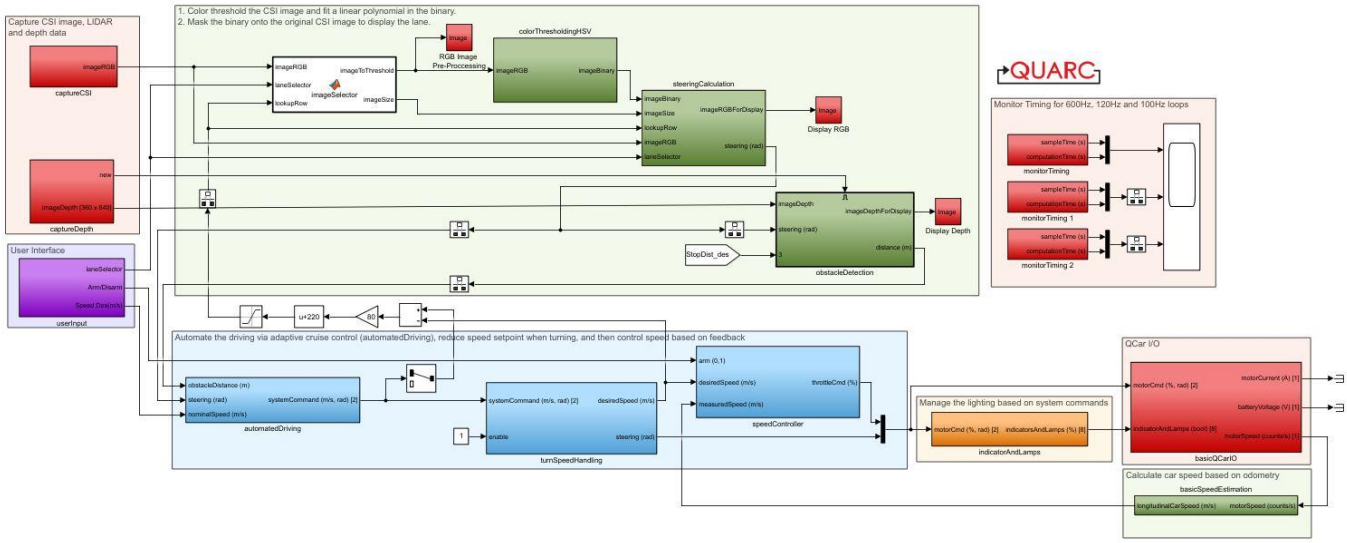


Figure 8. Simulink implementation of lane following with obstacle detection.

Running the Example

1. Check User Manual – Software Simulink for details on deploying Simulink models to the QCar as applications.

   The following example can be run by configuring a continuous lane loop using the roadmap layouts as part of the SDCS studio. Highly saturated (vibrant) colors for the line will produce the best results.

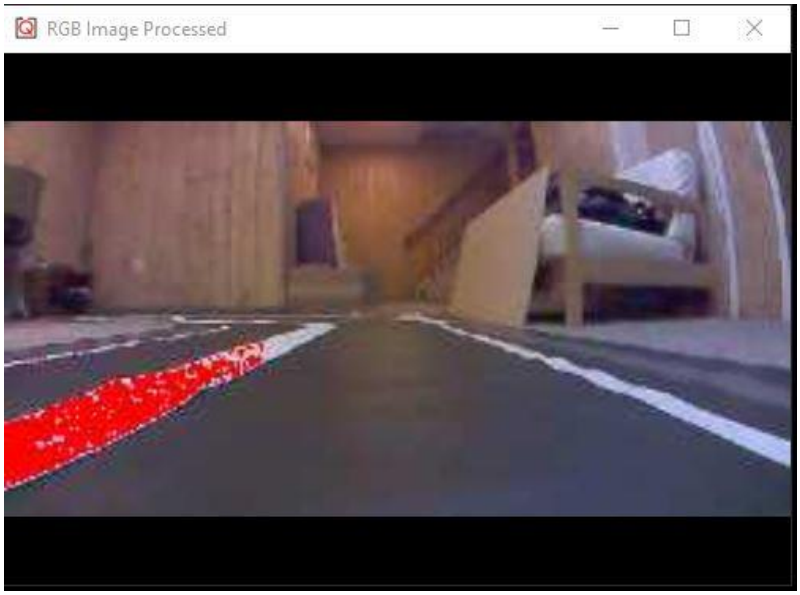


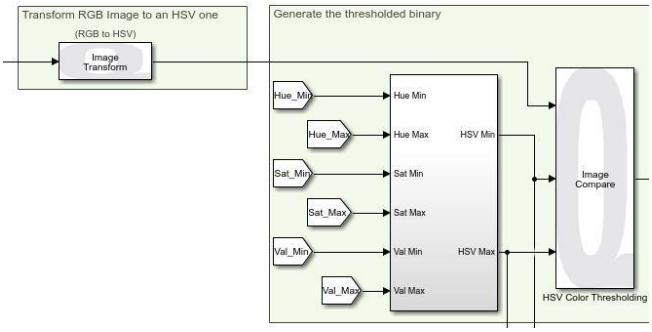
Figure 9. Processed lane extraction Image.

Details

1. imageSelector

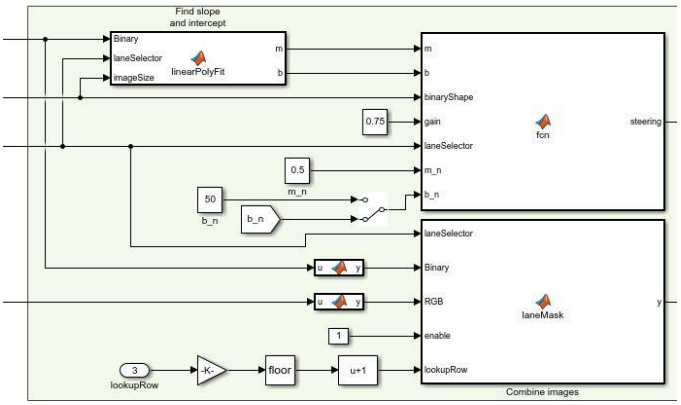
Based on the **laneSelector** option the following function will select the portion of the image used for the color thresholding algorithm. A laneSelector value of **1** will select the region of the image where the **right lane** is most likely to be present. A laneSelector value of **0** will select the region of the image where the **left lane** is most likely to be present.

2. colorThresholdingHSV



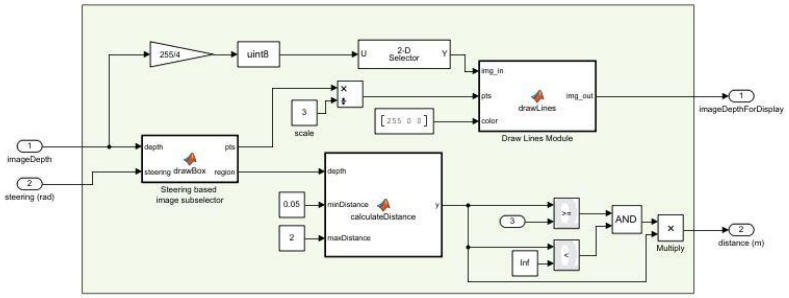
Color thresholding is done in two components. Component one converts the **imageRGB** input from the RGB to the HSV image plane. For in-depth information for the HSV image plane you can look at the **Image Color Spaces** document in the **Concept Review** directory. Prior to identifying the regions of 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.

3. steeringCalculation



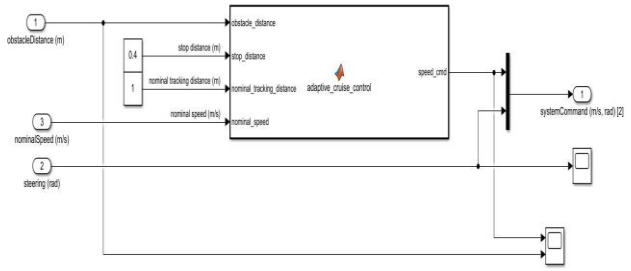
The first step for calculating the steering angle is to approximate two parameters which define the lane being tracked. The **linearPolyFit** function analyses the lane properties based on the **laneSelector** input. Using a linear approximation a **slope [m]** and **y-intercept [b]** is passed onto a second MATLAB function. We **compare** the **nominal_x** and the **desired_x** components of the slopes to identify how much our steering angle needs to be adjusted. The last MATLAB function **laneMask** combines the RGB image with the binary image from the **colorThersholdingHSV** to show the regular RGB image with red pixels over the lane which is currently being tracked.

4. obstacleDetection



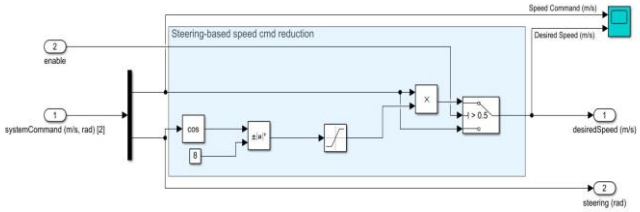
The **drawBox** function uses an input depth image of size 640x480 and the desired steering angle to extract a region of interest and sequence of points for the border of this region of interest. The depth information for the selected region is passed to the **calculateDistance** function. We find the 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 the average depth which we calculated is >= the stopDistance. The **drawLines** function uses the pixels from the **drawBox** function to define the lines that draw a red box for visualizing the region in the image of where the depth information is being computed.

5. 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**.

6. turnSpeedHandling



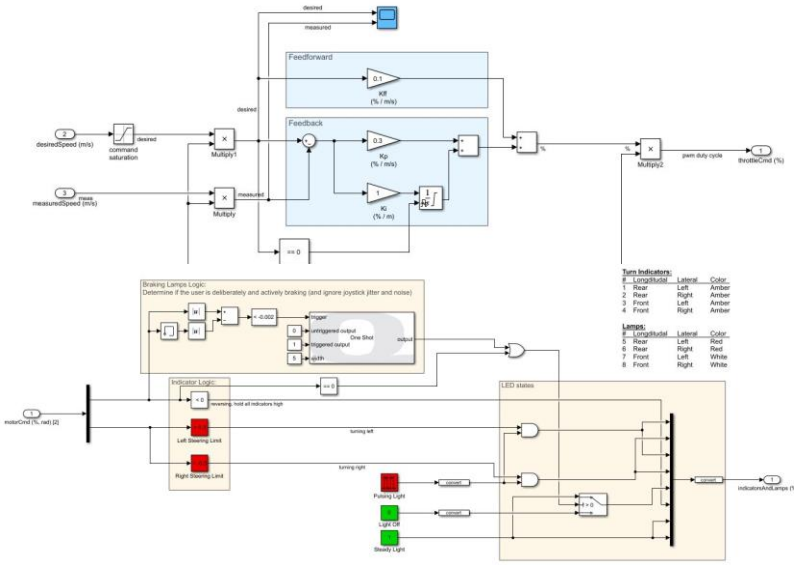
An enable constant is used to configure what the **desiredSpeed(m/s)** should be. We can pass the linear velocity command directly or evaluate the cosine of the steering to the power of 8. This secondary method will slow down the QCar closer to a turn and speed up during straight sections of road.

7. 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

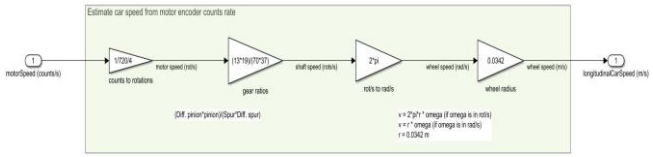
feedforward gain the controller command is no longer centered about zero but the desired setpoint defined by the feedforward gain.

8. indicatorAndLamps



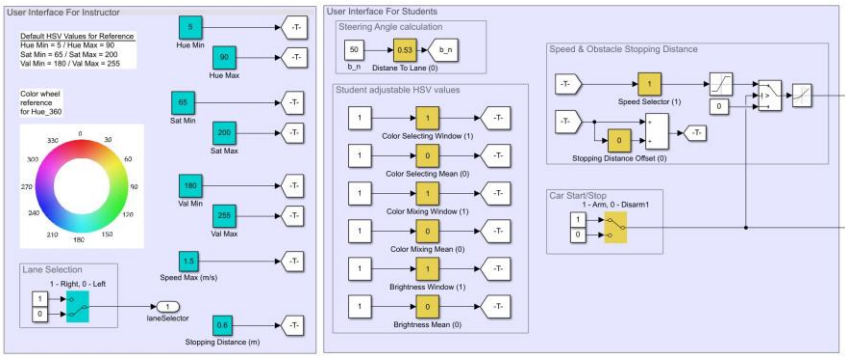
The logic inside this subsystem enables the LEDs on the QCar to function as a direction indicator. For the amber LEDs located at the front and the back of the QCar, these function as steering indicators. The **steering** is either **greater** than **0.3** for a **left** direction or **less** than **0.3** 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.

9. basicSpeedEstimation



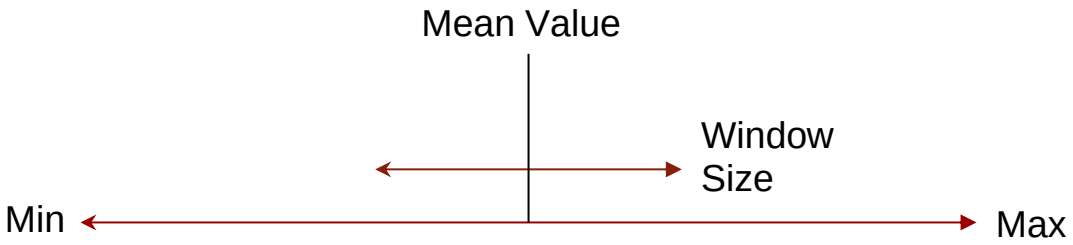
The **motor encoder** on the QCar can give us **counts/s** which is passed through four scaling terms. The **first scaling term converts** from **counts/s to rotations/s**, a **second scaling term** passes the motor rotations through a gear ratio which **gives the wheel shaft rotational speed.** The 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).

10. userInputs



This subsystem is divided into two sections. The group with the label QCar Parameter Definition allows us to define the max/min values used for the HSV thresholding, the lane which we want to follow, the stopping distance to an obstacle and the maximum speed of the QCar. For modifying the speed of the QCar we have a slider gain called **Speed Selector** which amplifies the desired speed of the QCar between

0% to **100%** of the maximum speed defined in section 1 of this subsystem. To control the stopping distance, we added an offset term called **Stopping Distance Offset**. By default, the minimum stopping distance is defined in section one of this subsystem. By **default,** the minimum stopping distance is set to be **0.6(m)**, the stopping distance offset adds an additional percentage of the minimum stopping distance. If the offset varies from **0%** which means the QCar stops at **0.6(m)** from an obstacle to **100%** offset which stops the QCar at **1.2(m)** away from the obstacle. To control how closely the QCar tracks the desired lane can be modified using the **Distance To Lane** slider constant which amplifies the desired lane slope from 0% to following the line directly and 100% which will set the QCar close to the center of the lane. Lastly, we have the sliders for the HSV parameters. They work using the following properties:



Every HSV parameter has a mean value and window size. The fine-tuning aspect of this model works as follows: The window size lets you decide how much of the interval **Max-Min** you want to use. A window size of 100% uses the complete range of values between Max and Min. The mean value allows you to modulate where in the color line the average value for your window will be.

Python

The camera image data is first capture, then we use image processing and analysis to output a steering command. The gamepad controller outputs a car speed and will overtake the steering if needed. The process is shown in Figure 10.

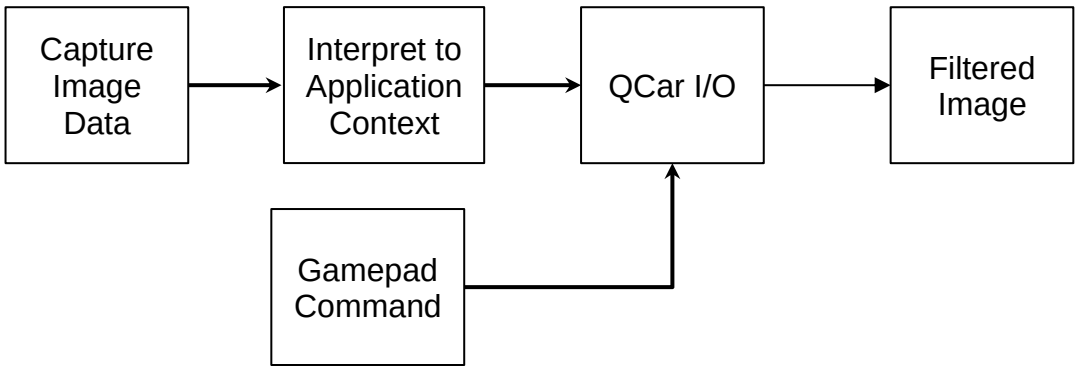


Figure 10. Component diagram

Running the example

1. First, open probe.py and change the variable *ipHost* in the "Additional parameters" section to the IP of the **local machine** before deploying it to QCar2. For details on accessing QCar2 remotely, check User Manual – Software Python.

2. Place your QCar on the right side of the yellow lane. First, run probe.py on QCar2 via PuTTY and then observer.py on the local machine. A window will show the detected yellow lane highlighted in red, as shown below. If there is no lane or the lane is grainy, adjust the **HSV** upper and lower bounds. Press **X** to enable the automatic steering. Press **RT** to provide throttle. If the QCar failed to follow the lane, release **X** to have manual control to the QCar. Use the **left stick** to manually steer.

Note: If the manual steering does not appear to work, please ensure the **mode** light on the gamepad is **off.**

Details

1. Image Processing

   We leverage the functionality of **OpenCV** in this application. After the image is cropped to let it focus on the lower half of the image frame, we use **cv2.cvtColor** to convert the image format. Please visit **OpenCV** official website to check out more functionalities. After the image is transferred to HSV format, **binary_thresholding** takes the HSV image and thresholds it based on the defined boundaries. This is the line where we can change to threshold difference colors.

```
# Convert to HSV and then threshold it for yellow
hsvBuf = cv2.cvtColor(cropped_rgb, cv2.COLOR_BGR2HSV)
```

```
binary = binary_thresholding(hsvBuf, lower_bounds=np.array([10, 50, 100]),
upper_bounds=np.array([45, 255, 255]))
```

2. Performance considerations

Raw_steering angle is controlled by the **slope** and **intercept** taking from **find_slope_intercept_from_binary**. Imagine a straight line crossing the first quadrant of an axis. The **slope** is the gradient which controls the turning angle. When the road is bending to the right, the **slope** decreases and vice versa. The **intercept** is the intercept value with the vertical axis which decides the distance that QCar tries to keep away from the yellow lane. The smaller the number, the closer it will get to the yellow lane. This is the line where we can adjust the **slope** and **intercept**.

```
# steering from slope and intercept
rawSteering = 1.5*(slope - 0.3419) + (1/150)*(intercept+5)
```