FrostAV: Lane & Sign Detecting Tenth-Scale Vehicle

Lewis Collum

Contents

1 Vehicle Monitoring with a Web Server

1.1 Installing Nginx, Flask, and GUnicorn

Since we are using Arch Linux ARM on our Pi, we used Arch Linux's package manager `pacman` for Nginx and Gunicorn. We used `pip` to install Flask.

```
sudo pacman -S nginx
sudo pacman -S gunicorn
sudo pip install flask
```

1.2 Starting the Server the First time

For ease, this repository was cloned on our Raspberry Pi to the user directory (e.g. "/home/alarm"). In the `frostServer/config` directory, there is a setup script[1] which does the following:

1) links the frostServer.service to the system directory;
2) links the frostServer.nginx config file to nginx's enabled-sites directory;
3) enables both the nginx and the frostServer services;
4) starts both services.

Now, on a computer connected to the same WiFi as our Pi, we can type the IP of the Pi into the browser search bar (e.g. "192.168.0.106") and the Frost website appears. Since we enabled the server service, the server will start at boot without any user intervention.

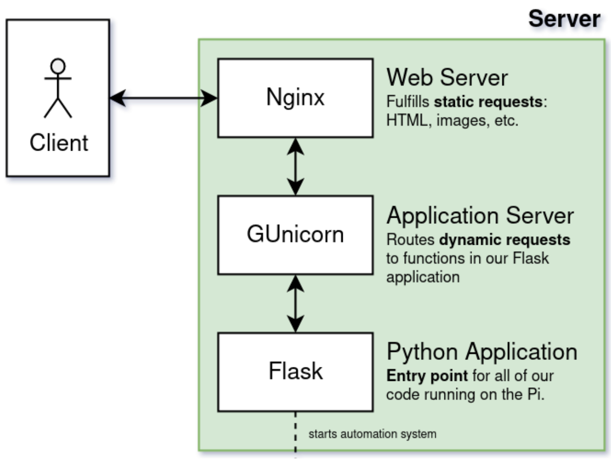
1.3 Server Overview: Flask, GUnicorn, and Nginx

We use Flask, GUnicorn, and Nginx. Let's work our way down, from the website, to our application code (in python).

Nginx acts as our web server, fulfilling requests from clients for static content from our website (e.g. HTML pages, files and images). Nginx forwards dynamic requests (e.g. all of the requests that we want Python to handle) to GUnicorn, our application server.

GUnicorn ensures that Nginx and our Flask Python application can talk to each other. Ultimately, GUnicorn routes requests (passed through Nginx) to their corresponding function in our Flask application.

Flask is a web microframework in the form of a Python library. It is used in our Python application to provide functions that can receive requests, and return a response (e.g. sensor data).


Server
Client
Nginx
Web Server
Fulfills static requests: HTML, images, etc.
GUnicorn
Application Server
Routes dynamic requests to functions in our Flask application
Flask
Python Application
Entry point for all of our code running on the Pi.
starts automation system

1.3.1 Example: Real-Time Sensor Reading

We have real-time plots on our front-end that request sensor data. Eventually, we expect to receive this sensor data from a URL of our choice (e.g. "/sensor_reading"). These requests are sent to Nginx, which are then passed to GUnicorn (since they are dynamic requests). GUnicorn sends the requests to a function in our python application that is registered for the formerly mentioned URL. The function is registered with the given URL using Flask. This function has code to get sensor data from the underlying linux system (typically, by reading a sensor file or running a shell command). Finally, the function returns a response which contains the sensor data.

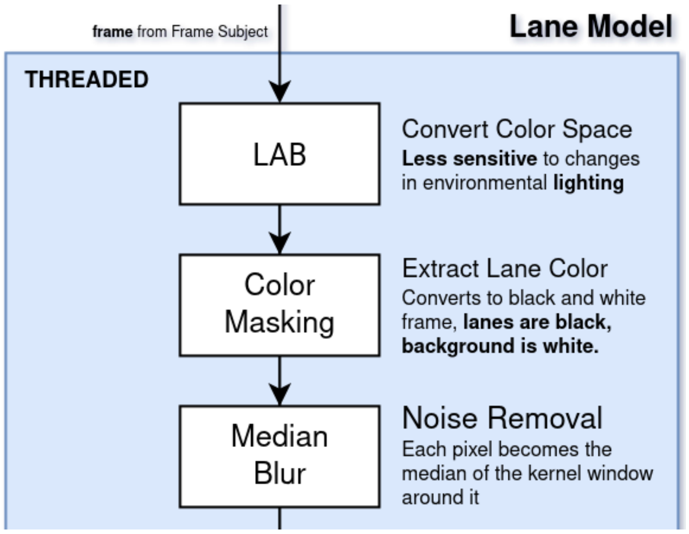
1. The setup script should be run as `sudo`.

2 Lane Detection

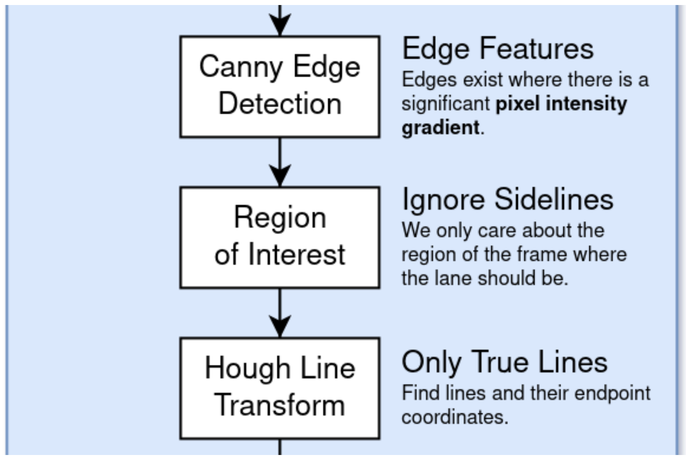
2.1 Summary

Get lane error to plug into vehicle controller.

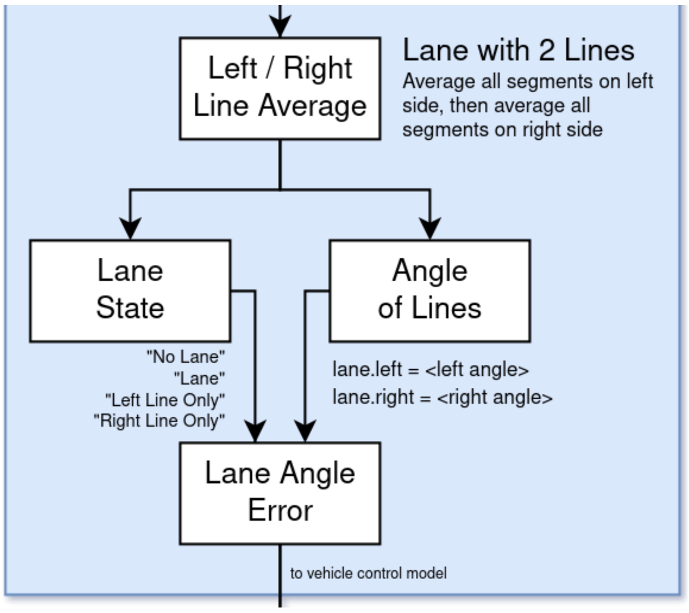
2.2 Black and White Lane Mask from Raw Frame


frame from Frame Subject
Lane Model
THREADED
LAB
Convert Color Space
Less sensitive to changes in environmental **lighting**
Color Masking
Extract Lane Color
Converts to black and white frame, **lanes are black, background is white.**
Median Blur
Noise Removal
Each pixel becomes the median of the kernel window around it

2.3 Line Segment Coordinates from Lane Mask


Canny Edge Detection
Edge Features
Edges exist where there is a significant **pixel intensity gradient**.
Region of Interest
Ignore Sidelines
We only care about the region of the frame where the lane should be.
Hough Line Transform
Only True Lines
Find lines and their endpoint coordinates.

2.4 Lane Angle Error from Line Segment Coordinates



3 YOLOv3 Real-Time Sign Detection

3.1 Results

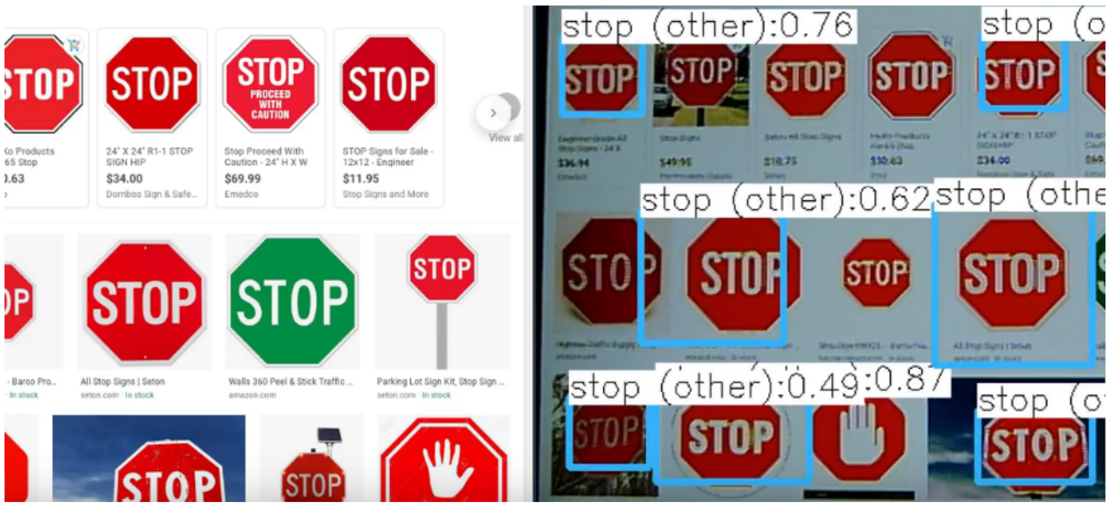
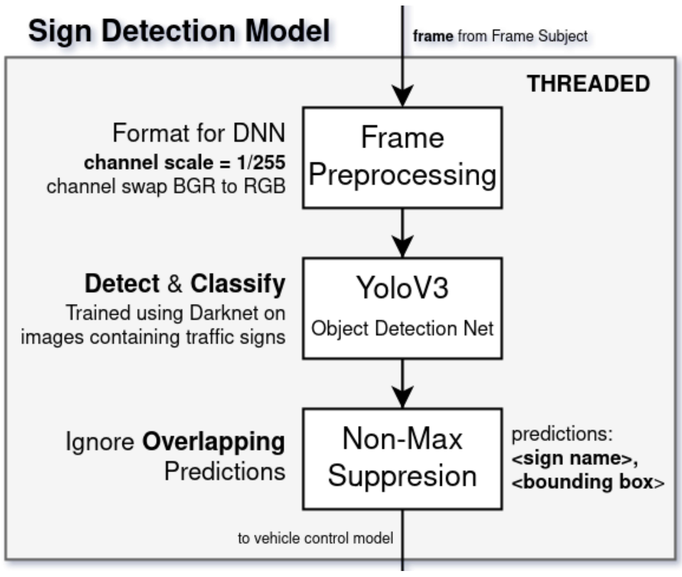


Fig. 1. Stop sign images on a monitor (left), and the processed video feed (right).

3.2 How YOLOv3 was Implemented



3.3 Easy. Detect Signs Then Classify.

No. This is how object detection[2] algorithms started (e.g. R-CNN), but they can be too slow for real-time (and embedded) object detection. Those looking for speed, use algorithms that extract classified objects from the frame in a single pass, as opposed to two passes. YOLOv3 is an algorithm that detects in a single pass.

There are multiple versions of YOLOv3. We are using YOLOv3-tiny-prn, which has the highest frames per second (FPS) compared to other commonly used algorithms (see figure 2). The sacrifice to speed is accuracy, as YOLOv3-tiny-prn borders around 35% average precision. Since we implemented sign detection on a Raspberry Pi as a proof-of-concept, this accuracy is acceptable.

4 Vehicle Control Model

4.1 Control Value Packaging on the Raspberry Pi

4.1.1 Summary

Get lane error to plug into vehicle controller.

2. Classification is not detection. Objects first need to be detected before they can be classified. But, for briefness, we say "object detection" when we really mean "object detection and classification."

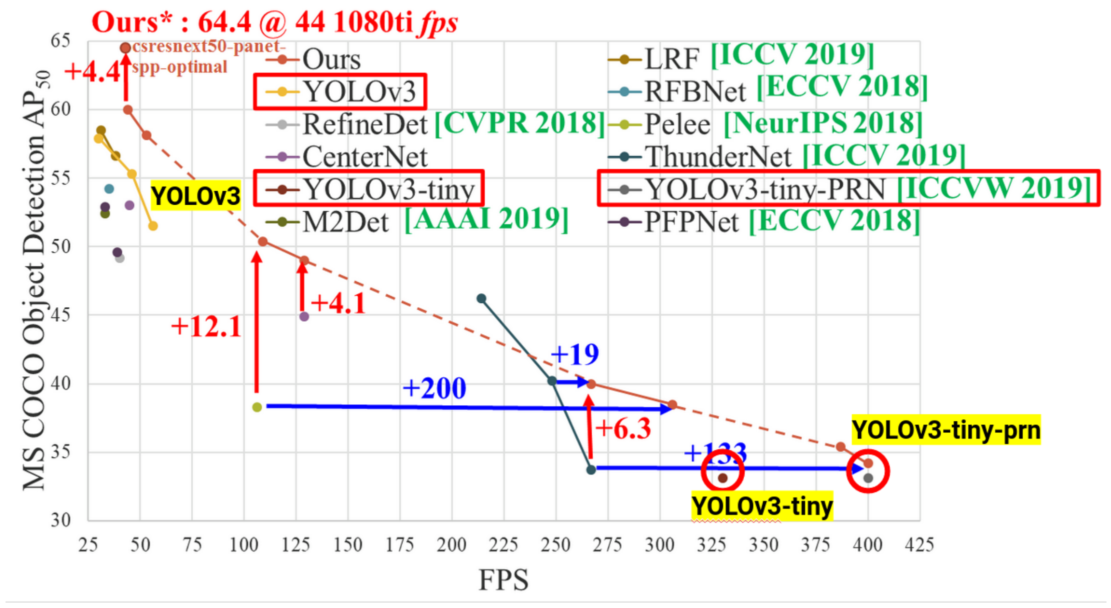
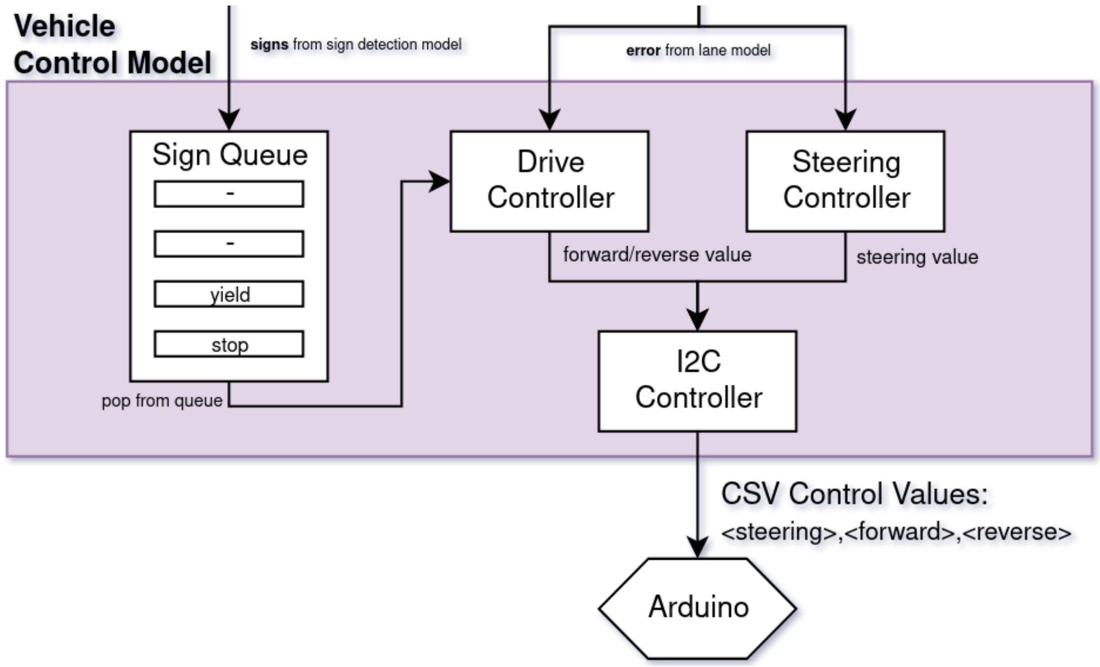


Fig. 2. YOLOv3-tiny-prn has the highest FPS, with an acceptable 35% average precision.



4.2 Vehicle Interface Controller (Arduino)

4.2.1 Overview

We are using an Arduino Uno to take in i2c steering and drive values to control the vehicle's steering servo and electronic speed controller (ESC). We refer to the Arduino as the Vehicle Interface Controller (VIC).

The VIC is programmed in C++ (not using Arduino's packages). This gives us control over the implementation details of utilities like strings, pwm, usart, and i2c. Note that this decision was made for our own academic curiousity.

4.2.2 Application Makefiles

The VIC uses GNU Makefiles. We have modularized the makefile process, such that when starting a new application file (i.e. a source file which contains `main`) the makefile to build the application only needs to contain

```
TARGET = <name of your application>
ROOT = ../..
include $(ROOT)/mk/all.mk
```

where `ROOT` is the directory that contains the `mk` directory.

4.2.3 Making a New Application

We have included a bash script in `app` called `createNewApp`. To make a new application, we can go to the app directory and run

```
  ./createNewApp <name of application>
```

This script ensures there are no existing applications and if that is true:

1) will make a directory with the same name as our application;
2) will make a `cpp` source file with the name as our application;
3) will create a Makefile, filled with everything necessary to build our application.

4.2.4 Flashing the Arduino

To flash the Arduino, we go to the directory of an application (in the `app` directory), and then we run `make flash`. The build files are exported to the `app/build` directory.

4.2.5 Serial Interface

To communicate with the Arduino, via Serial, we use `picocom`, a linux terminal utility. As part of our modular makefile system, we can open `picocom` by going to the directory of an application (in the `app` directory), and running `make com`.

5 Simple Framework for Nodal Frame Processing