Lane-Keeping Assist System: Documentation

Zrinszki Ármin (WTHTDS)

November 3, 2025

Abstract

This report is for the ADAS LKA assignment for the Vehicles and Sensors course. This project implements a real-time lane keeping assist pipeline based on a classical computer vision baseline. The system is built in Python using OpenCV. The system processes a forward-facing, centered video feed to identify the track boundaries.

Contents

1 Introduction

1.1 Project Objective

The objective of this project was to develop a complete computer vision pipeline capable of processing a single video feed to determine the lines and their certainty. It should create a .csv file for each frame of the video, and also annotate the video feed itself. The system should emit a warning when the lines are not sufficiently found, thus prompting the driver to take over the driving.

2 System Architecture: The Pipeline

The system runs this entire pipeline on every frame of the input video.

1. **Image Preprocessing**: The raw video frame is converted into a binary image where white pixels represent a possible line
2. **Perspective Transformation**: The binary image is then warped into a bird's-eye view, so that the traepzoid lines from the driver's perspective become parallel.
3. **Lane Pixel Detection**: The warped image is analyzed to find the (x, y) coordinates of all lane pixels. A histogram finds the base of the lanes, and a sliding window algorithm follows them up the frame.
4. **Polynomial Fitting**: A 2nd-order polynomial (a parabola) is fit to the detected pixels for both the left and right lanes. These fits are subjected to "sanity checks" to discard any misdetections.
5. **Temporal Smoothing**: The raw polynomial fits are passed to a `Line` class, which applies an Exponential Moving Average (EMA). This smooths the detection over time, preventing "jitter" and allowing the system to work through brief detection failures.
6. **Metrics Calculation**: Using the smoothed polynomials, real-world metrics are calculated, including the radius of curvature (in meters) and the vehicle's lateral offset from the lane center (in meters).
7. **Overlay and Visualization**: The detected lane area is drawn onto the original, unwarped frame, along with the calculated metrics and system status warnings, providing a final visual output to the driver.

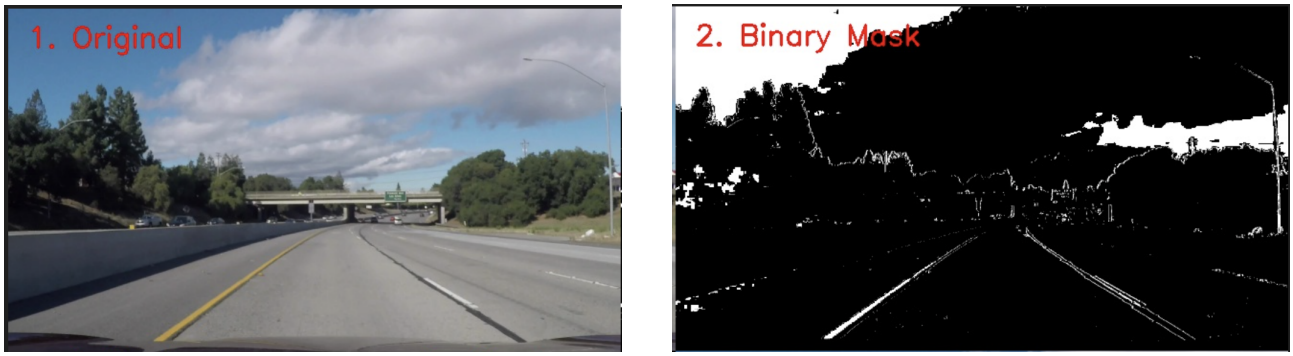
3 Detailed Module Implementation

3.1 Preprocessing (`preprocess.py`)

The goal of preprocessing is to create a binary mask that isolates lane pixels from the rest of the image. The implementation combines color and gradient information.

- **Color (HLS Space)**: The image is converted to the HLS (Hue, Lightness, Saturation) color space. The S-channel is excellent for isolating saturated colors (like yellow lines) under varying light, while the L-channel is effective for identifying white lines based on brightness.
- **Gradient (Sobel Operator)**: The Sobel operator is applied in the x-direction to the L-channel. This highlights strong vertical edges, which are presumably lines.

The final mask combines these: we take any pixel that is identified by the S-channel (color) **OR** by *both* the L-channel (brightness) and the Sobel-X (gradient) masks.



(a) Original (b) Masked

Figure 1: Comparison of the original and masked images.

3.2 Perspective Warp (`warp.py`)

To analyze the lanes, we must transform the image into a top-down "bird's-eye" view. This is achieved with a perspective transform.

- **Calibration**: The user manually selects 4 source points on the first frame, defining a trapezoid that outlines the lane.
- **Destination**: The destination points have a hardcoded offset from the sides, making the system only work well with the example video's camera settings.
- **Matrices**: The function `cv2.getPerspectiveTransform` is used to compute both the transform matrix (M) and its inverse ($Minv$), which is needed later for visualization.

3.3 Lane Fitting (`lane_fit.py`)

This module is the heart of the detection algorithm.

- **Histogram**: A histogram of the bottom half of the warped image is computed. The two peaks (on the left and right side) of this histogram correspond to the starting x-positions of the left and right lane lines.
- **Sliding Windows**: Starting from the histogram peaks, 9 windows are stacked vertically up the image. The windows are re-centered at each step based on the pixels found inside them.
- **Sanity Checks**: Before a polynomial fit is accepted, it is checked for plausibility.
  1. *Parallelism*: The 'A' coefficients of the two polynomials must be similar (i.e., `abs(left_fit[0] - right_fit[0]) < 0.001`).
  2. *Distance*: The horizontal distance between the lines at the bottom of the frame must be close to the `warped_lane_width_px` calculated in `run_pipeline.py`.

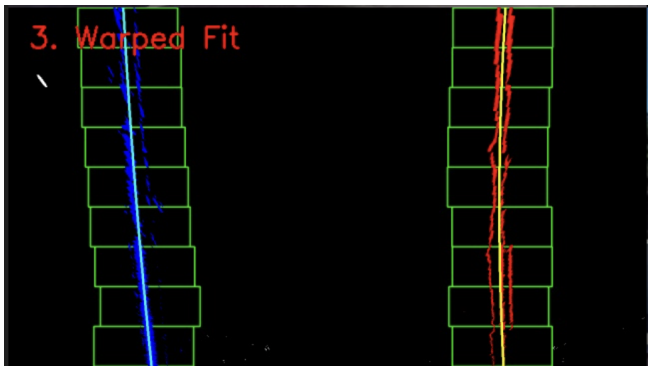


Figure 2: Warped line fitted image

3.4 Temporal Smoothing (`temporal.py`)

Raw polynomial fits can "jitter" from frame to frame. The `Line` class solves this by implementing an Exponential Moving Average (EMA).

If no fit is received (i.e., it fails the sanity check), the class increments `frames_since_detected`. If this exceeds `max_frames_lost`, the line is marked as not detected, triggering a "DRIVER TAKE OVER" warning.

3.5 Metrics Calculation (`metrics.py`)

To be useful, the system must provide real-world measurements.

- **Conversion**: I define conversion factors `xm_per_pix` (meters per pixel in x) and `ym_per_pix` (meters per pixel in y). `xm_per_pix` is calculated dynamically using the 3.7m standard lane width and our `warped_lane_width_px`.
- **Lateral Offset**: The vehicle's center (image center) is compared to the lane's center (midpoint of the two lane fits at the bottom of the frame). This pixel difference is converted to meters using `xm_per_pix`.

4 Key Challenges and Solutions

The development process was iterative. Several key challenges required significant refactoring to achieve the final robust system.

4.1 Initial Region of Interest

Problem: The initial version of `preprocess.py` contained a hardcoded triangular mask. This was intended to be a "region of interest" (ROI) mask, but it was redundant and destructive. It was applied before the perspective warp, effectively deleting valid pixel data from the upper portion of the user-selected trapezoid.

Solution: The triangular mask was removed entirely. The true ROI is already defined by the four source points the user selects during calibration in `warp.py`. Removing the redundant mask immediately provided the pipeline with the correct data, as seen in the debug images.

4.2 Sensitivity to Noise

After fixing the ROI, the pipeline successfully identified pixels, but the `lane_fit` module was highly unstable. The sliding windows were "pulled away" from the faint left lane line by a larger, noisier cluster of pixels from the road edge.

4.3 Hardcoded Dimensions

Problem: The system only works well with the camera charasteristics of the example video, as it's dimesions and offsets are hardcoded.

4.4 Poor quality or misleading road markings

A perfectly smooth road rarely exists, often there are patches on the road, or different kind of materials, or other lines that do not mark the lanes, for example those long black repair lines.

Figure 3: Enter Caption

5 Future Work

5.1 Future Work

While functional, the system has several clear paths for enhancement:

- Trying out a pretrained deep learning model or fine-tuning one.
- Making the dimensions dynamic, so the implementation can be more generic
- Better filterization for line finding. Too much noise or misleading lane markings may confuse the pipeline.