CHAPTER 1

INTRODUCTION

1.1 Background

Lane detection technology has become a pivotal component in modern vehicular safety systems and autonomous driving solutions. As vehicles become increasingly sophisticated, the demand for advanced driver assistance systems (ADAS) and fully autonomous vehicles has grown. These systems rely on accurate and reliable lane detection to function effectively, making it essential to develop and refine technologies that can identify and interpret lane markings in various driving conditions.

The field of computer vision has significantly contributed to advancements in lane detection, enabling vehicles to perceive and analyze their surroundings with greater precision. By leveraging camera sensors and image processing techniques, lane detection systems can provide real-time feedback to drivers and assist in maintaining proper lane positioning, thereby enhancing overall road safety.

1.2 Purpose and Importance of Car Lane Detection

Car lane detection is a crucial technology for enhancing vehicle safety and assisting with autonomous driving. The primary purpose of car lane detection is to identify lane markings on the road, which helps drivers stay within their lane and avoid unintended lane departures. This technology is foundational for Advanced Driver Assistance Systems (ADAS) and autonomous driving, where it provides real-time feedback to drivers and supports automated lane-keeping functions. By accurately detecting lane lines, the system can significantly reduce the risk of accidents caused by lane drifting or merging errors.

1.3 Project Outline

This project aims to develop a robust car lane detection system using computer vision techniques. The project is divided into several key components:

Camera Calibration: This step involves correcting the distortions introduced by the camera lens to ensure accurate lane detection. The calibration process uses images of a known calibration pattern to compute the distortion coefficients and camera matrix.

1. **Thresholding**: In this step, the system extracts relevant pixels from the input images by applying colour and gradient thresholds. This process helps isolate the lane markings from the rest of the image.
2. **Perspective Transformation**: This component converts the images from a front view to a top-down view (bird's-eye view), making it easier to detect lane lines and fit polynomial curves.
3. **Lane Line Detection**: The core of the project involves detecting lane lines by fitting polynomial curves to the binary images obtained from the thresholding step. This allows the system to track the lane lines and estimate their curvature.
4. **Overlay and Visualization**: Finally, the detected lane lines and additional information such as lane curvature and vehicle position are overlaid on the original image to provide visual feedback to the driver.

1.4 Scope and Uses

The scope of this project includes the development and implementation of a lane detection system that can be applied in various scenarios. The primary uses of this technology include:

- **Lane Departure Warning Systems**: Alerting drivers when they unintentionally drift out of their lane.
- **Lane Keeping Assist Systems**: Providing corrective steering inputs to keep the vehicle centred within its lane.
- **Autonomous Vehicles**: Assisting self-driving cars in understanding and navigating road lanes safely.

By addressing these use cases, the project contributes to improving vehicle safety and supporting the development of autonomous driving technologies.

1.5 Project Description

The project is implemented using Python and leverages computer vision libraries such as OpenCV and Matplotlib. The implementation involves several key classes:

- **CameraCalibration**: Handles the correction of lens distortion by computing the camera matrix and distortion coefficients from calibration images.
- **Thresholding**: Extracts relevant lane pixels from images using color and gradient-based thresholds.
- **PerspectiveTransformation**: Transforms images to a top-down view to simplify lane detection.
- **LaneLines**: Detects and fits polynomial curves to lane lines and provides feedback on lane curvature and vehicle position.
- **FindLaneLines**: Integrates all components to process images or videos, applying each step in sequence to detect and visualize lane lines.

1.6 Existing System

The existing systems for lane detection typically involve traditional computer vision techniques combined with machine learning models. These systems often rely on methods such as color thresholding, edge detection, and the Hough transform to identify lane lines. They may also utilize fixed camera setups and are generally tailored to specific types of roadways and driving conditions.

These systems have the following characteristics:

- **Limited Flexibility:** They may not adapt well to varying lighting conditions or different types of roads.
- **Hardware Dependency:** Often requires specialized camera systems and processing units.
- **Predefined Algorithms:** Typically use fixed algorithms that may not generalize well across different scenarios.

While effective for basic lane detection, these systems may struggle with more complex environments or varying conditions. The proposed system aims to address these limitations by incorporating advanced image processing techniques and dynamic calibration methods to improve accuracy and adaptability.

1.7 Proposed System

The proposed system aims to deliver a comprehensive solution for detecting and analyzing lane lines. It includes the following features:

- **Real-time Lane Detection**: Processes live camera feed or video input to detect lane lines and provide immediate feedback.
- **Calibration and Correction**: Corrects camera distortions to ensure accurate lane detection.
- **Thresholding and Transformation**: Applies color and gradient thresholds to extract lane markings and transforms images to a top-down view.
- **Polynomial Fitting**: Detects lane lines by fitting polynomial curves and calculates their curvature.
- **Visual Feedback**: Overlays lane lines and additional information on the original image, providing clear feedback to the driver.

This system is designed to enhance vehicle safety by improving lane-keeping capabilities and supporting autonomous driving functions.

CHAPTER 2

SYSTEM REQUIREMENTS

2.1 Software Requirements

To implement and operate the lane detection system efficiently, several software tools and libraries are utilised. Each component serves a specific purpose in the development and execution of the system:

Operating System: Windows 10 / Linux / macOS

Language Tool: Python 3.x

Compiler/IDE: Visual Studio Code / PyCharm / Jupyter Notebook

Libraries: OpenCV, NumPy, Matplotlib, MoviePy, SciPy, scikit-image

Additional Tools: PIL (Pillow) for image processing

2.2 Hardware Requirements

The hardware requirements ensure that the lane detection system operates smoothly and efficiently. The key components include:

Processor: Intel i5 or equivalent (or faster) processor

Memory: 4 GB RAM (minimum)

Storage: At least 200 MB of free disk space for the application files and dependencies

Graphics: Integrated or dedicated graphics with support for OpenGL or equivalent for image processing

Display: 1280x720 resolution or higher

Input Devices: Keyboard and mouse for user interaction; optionally, a camera for live video processing

CHAPTER 3

SYSTEM TOOLS

In developing the lane detection system, several essential tools and libraries are employed. This chapter provides an overview of these tools, focusing on OpenCV technology and additional libraries that are integral to the system's functionality.

3.1 History

1. Origins and Early Development:

- **1999**: OpenCV was created by Intel to advance real-time computer vision applications. It was initially developed by Gary Bradski and the Intel Open Source Computer Vision Library team.
- **2000**: The first official release of OpenCV (version 0.9) was made. The library aimed to provide a comprehensive set of tools for real-time image processing and computer vision.

2. Growth and Open Source Expansion:

- **2006**: OpenCV was released under the BSD license, which made it more accessible for both academic and commercial purposes. This move helped in expanding its user base and contributions from the open-source community.
- **2008**: Version 1.0 was released with significant improvements, including enhanced performance and additional functionalities. The project also transitioned to a more community-driven approach.

3. Advancements and Modernization:

- **2012**: OpenCV 2.4.x was released, introducing many new features, improvements in performance, and better support for modern hardware.
- **2015**: OpenCV 3.0 was released, marking a significant update with modularization, improved performance, and new functionalities. It introduced features like deep learning support and integration with TensorFlow and Caffe.

4. Recent Developments:

- **2018**: OpenCV 4.0 was released with major enhancements, including improved performance, better support for deep learning frameworks, and new features in computer vision and machine learning.
- **2021**: OpenCV continued to evolve with incremental updates, improving its capabilities in areas such as object detection, image segmentation, and real-time processing.

5. Current Status:

- OpenCV remains a leading library in the computer vision field, widely used in academia, industry, and hobbyist projects. Its active community and continuous updates ensure it stays relevant with the latest advancements in computer vision and machine learning.

OpenCV's development has been driven by a combination of corporate support, academic research, and open-source contributions, making it a robust and versatile tool for a wide range of computer vision applications.

3.2 OpenCV Technology

OpenCV (Open Source Computer Vision Library) is a cornerstone of the lane detection system. It is a widely-used, open-source library designed to facilitate real-time computer vision and image processing tasks. OpenCV provides a comprehensive suite of functions for handling various image and video processing needs.

- **Core Features**:
    - **Image Processing**: OpenCV includes fundamental operations such as image resizing, color space conversions, filtering, and geometric transformations.
    - **Feature Detection**: It provides algorithms for detecting and describing features in images, such as edges, corners, and contours.
    - **Object Detection**: Tools for detecting objects and patterns in images, including lane markings, are supported through various algorithms and machine learning models.

- **Camera Calibration**: OpenCV offers functionalities for calibrating cameras to correct lens distortion and improve the accuracy of measurements.

- **Applications**:
  - **Perspective Transformation**: OpenCV's functions are used to transform images from a front view to a top-down view, which is crucial for lane detection.
  - **Thresholding**: It provides methods for converting images to binary format based on intensity thresholds, aiding in the extraction of lane lines from complex backgrounds.
  - **Drawing and Visualization**: OpenCV allows for drawing lines, shapes, and text on images to visualize detected lanes and other relevant features.

OpenCV's robust set of tools and efficient performance make it ideal for real-time image processing and computer vision tasks in the lane detection system.

3.3 OpenCV Functions Used

The following OpenCV functions are pivotal in implementing the lane detection pipeline:

cv2.getPerspectiveTransform(src, dst):

- **Purpose**: Computes the perspective transformation matrix for warping images between different views.
- **Usage**: Transforms images from the front view to the top-down view and vice versa.

cv2.warpPerspective(img, M, dsize, flags=cv2.INTER_LINEAR):

- **Purpose**: Applies a perspective transformation to an image using the computed matrix.
- **Usage**: Converts images to and from top-down views based on the transformation matrix.

cv2.cvtColor(img, code):

- **Purpose**: Converts an image from one color space to another.
- **Usage**: Changes images between RGB, HLS, and HSV color spaces to facilitate thresholding and feature extraction.

cv2.normalize(src, dst, alpha, beta, norm_type, dtype):

- **Purpose**: Scales and normalizes image data.
- **Usage**: Ensures that images are in a consistent range and format for further processing.

cv2.line(img, pt1, pt2, color, thickness):

- **Purpose**: Draw a line on an image.
- **Usage**: Visualizes detected lane lines by drawing them on the image.

cv2.putText(img, text, org, fontFace, fontScale, color, thickness):

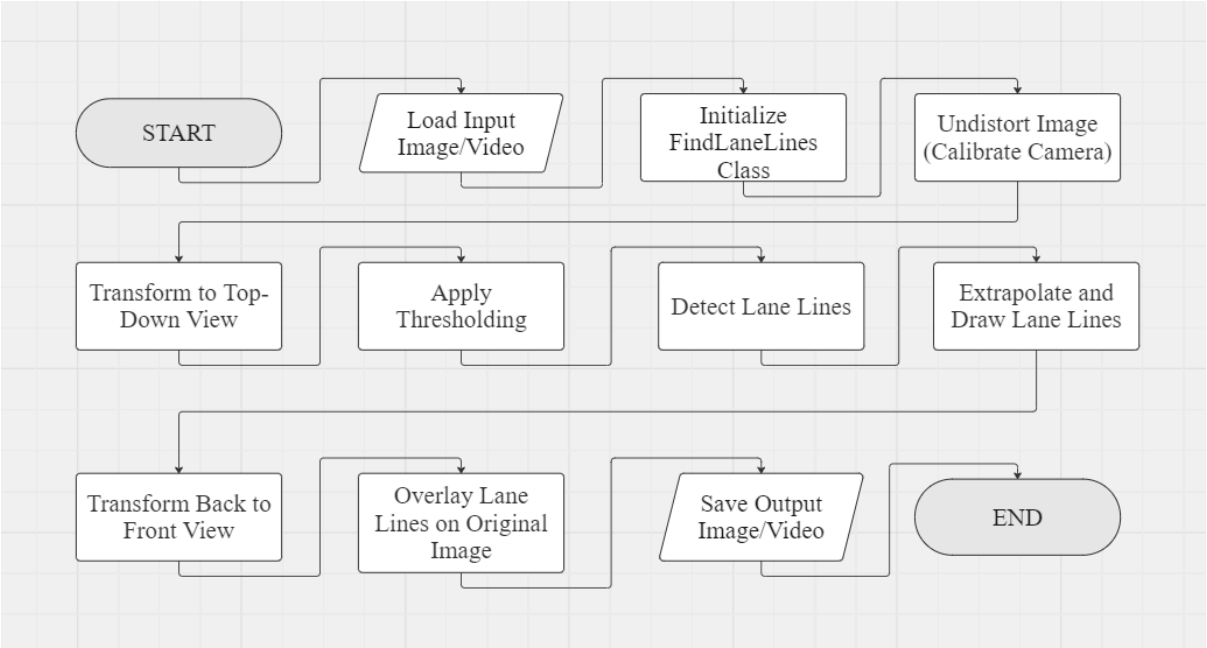
- **Purpose**: Adds text to an image.
- **Usage**: Annotates images with information such as lane curvature and vehicle position.

These functions enable the lane detection system to perform essential image processing tasks, from transforming perspectives to visualising results.

CHAPTER 4

SYSTEM DESIGN

4.1 Flow Chart

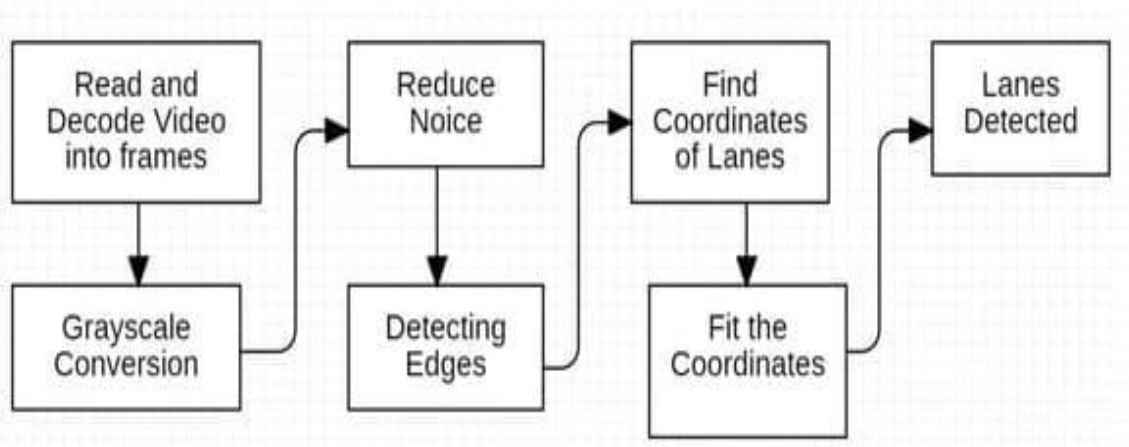


This flowchart outlines the key steps in your lane detection system pipeline:

1. **Load Input Image/Video**: The input image or video file is loaded for processing.
2. **Initialize FindLaneLines Class**: An instance of the FindLaneLines class is created to manage the overall process.
3. **Undistort Image**: The image is corrected for camera distortion using the CameraCalibration class.
4. **Transform to Top-Down View**: The front view image is transformed into a top-down (bird's-eye) view using the PerspectiveTransformation class.
5. **Apply Thresholding**: Relevant pixels are extracted from the transformed image using the Thresholding class.
6. **Detect Lane Lines**: Lane lines are detected in the thresholded image using the LaneLines class.
7. **Extrapolate and Draw Lane Lines**: Detected lane lines are extrapolated to fit the full image height and drawn on the image using the LaneLines class.

8. **Transform Back to Front View**: The top-down view image with detected lane lines is transformed back to the front view using the PerspectiveTransformation class.
9. **Overlay Lane Lines on Original Image**: The detected lane lines are overlaid on the original input image.
10. **Save Output Image/Video**: The processed image or video with lane lines is saved to the specified output path.

4.2 Architectural Design



The architectural design outlines the overall structure of the lane detection system, including the interaction between its various components. This design ensures that each part of the system functions cohesively to achieve the desired results. The main components and their interactions are as follows:

- **Input Module:** Captures images from the camera or video feed. This module handles image acquisition and preprocessing.
- **Preprocessing Module:** Performs image calibration and perspective transformations to standardize the input for further processing.
- **Feature Extraction Module:** Extracts relevant features such as lane lines using techniques like color thresholding and edge detection.
- **Lane Detection Module:** Applies algorithms such as Hough Transform and polynomial fitting to detect and extrapolate lane lines.
- **Visualization Module:** Draws the detected lane lines on the image, and overlays additional information such as curvature and direction.

- **Output Module:** Saves or displays the processed image or video with the detected lane lines and associated information.

Each module is designed to handle a specific aspect of the lane detection process, ensuring modularity and clarity in the system's design. The architectural design facilitates easy updates and maintenance by isolating changes to individual components without affecting the entire system.

CHAPTER 5

IMPLEMENTATION

This chapter outlines the practical aspects of implementing the lane detection system. It covers the overall pipeline of the lane detection system, the classes and methods used, and how they are integrated to achieve the final output.

5.1 Overview of the Pipeline

The lane detection system is structured as a series of processing steps that convert raw image data into actionable lane line information. The pipeline consists of:

1. **Camera Calibration**: This step corrects for lens distortion and ensures that the images accurately represent the real-world geometry. Calibration involves capturing multiple images of a calibration pattern (such as a checkerboard) and using them to compute distortion coefficients.
2. **Perspective Transformation**: The system transforms the front view image into a top-down (bird's-eye) view to simplify lane line detection. This transformation involves defining source and destination points on the image and using them to compute a perspective transformation matrix.
3. **Thresholding**: Relevant pixels are extracted from the transformed image using color and gradient thresholding techniques. This step is crucial for isolating lane lines from other road features and background noise.
4. **Lane Line Detection**: The system identifies lane lines by detecting edges and using the Hough Transform to find lines in the binary image. Detected lines are further processed to determine their positions and curvature.
5. **Lane Line Extrapolation and Visualization**: The detected lane lines are extrapolated to fit the full image height and drawn on the image. Additional visual cues are provided to indicate lane curvature and vehicle positioning.
6. **Output Generation**: The processed image or video is generated, with lane lines overlaid and additional information such as curvature and lane direction displayed.

5.2 Source Code

5.2.1 Camera Calibration Class

```
import numpy as np
import cv2
import glob
import matplotlib.image as mpimg
import matplotlib.pyplot as plt

class CameraCalibration():
    def __init__(self, image_dir, nx, ny, debug=False):
        fnames = glob.glob("{}/*".format(image_dir))
        objpoints = []
        imgpoints = []

        objp = np.zeros((nx*ny, 3), np.float32)
        objp[:,:2] = np.mgrid[0:nx, 0:ny].T.reshape(-1, 2)

        # Go through all chessboard images
        for f in fnames:
            img = mpimg.imread(f)

            gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
            ret, corners = cv2.findChessboardCorners(img, (nx, ny))
            if ret:
                imgpoints.append(corners)
                objpoints.append(objp)

        shape = (img.shape[1], img.shape[0])
        ret, self.mtx, self.dist, _, _ = cv2.calibrateCamera(objpoints, imgpoints, shape, None)

        if not ret:
            raise Exception("Unable to calibrate camera")
```

```
    def undistort(self, img):
        gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
        return cv2.undistort(img, self.mtx, self.dist, None, self.mtx)
```

5.2.2 Perspective Transformation Class

```
import cv2
import numpy as np

class PerspectiveTransformation:
    def __init__(self):
        self.src = np.float32([(550, 460),     # top-left
                               (150, 720),     # bottom-left
                               (1200, 720),    # bottom-right
                               (770, 460)])    # top-right
        self.dst = np.float32([(100, 0),
                               (100, 720),
                               (1100, 720),
                               (1100, 0)])
        self.M = cv2.getPerspectiveTransform(self.src, self.dst)
        self.M_inv = cv2.getPerspectiveTransform(self.dst, self.src)

    def forward(self, img, img_size=(1280, 720), flags=cv2.INTER_LINEAR):
        return cv2.warpPerspective(img, self.M, img_size, flags=flags)

    def backward(self, img, img_size=(1280, 720), flags=cv2.INTER_LINEAR):
        return cv2.warpPerspective(img, self.M_inv, img_size, flags=flags)
```

5.2.3 Thresholding Class

```
import cv2
import numpy as np
```

```
def threshold_rel(img, lo, hi):
    vmin = np.min(img)
    vmax = np.max(img)

    vlo = vmin + (vmax - vmin) * lo
    vhi = vmin + (vmax - vmin) * hi
    return np.uint8((img >= vlo) & (img <= vhi)) * 255

def threshold_abs(img, lo, hi):
    return np.uint8((img >= lo) & (img <= hi)) * 255

class Thresholding:

    def __init__(self):
        pass

    def forward(self, img):
        hls = cv2.cvtColor(img, cv2.COLOR_RGB2HLS)
        hsv = cv2.cvtColor(img, cv2.COLOR_RGB2HSV)
        h_channel = hls[:,:,0]
        l_channel = hls[:,:,1]
        s_channel = hls[:,:,2]
        v_channel = hsv[:,:,2]

        right_lane = threshold_rel(l_channel, 0.8, 1.0)
        right_lane[:,:750] = 0

        left_lane = threshold_abs(h_channel, 20, 30)
        left_lane &= threshold_rel(v_channel, 0.7, 1.0)
        left_lane[:,550:] = 0

        img2 = left_lane | right_lane
        return img2
```

5.2.3 Main Class

```
import numpy as np
import matplotlib.image as mpimg
import cv2
from docopt import docopt
from IPython.display import HTML, Video
from moviepy.editor import VideoFileClip
from CameraCalibration import CameraCalibration
from Thresholding import *
from PerspectiveTransformation import *
from LaneLines import *

class FindLaneLines:
    def __init__(self):
        self.calibration = CameraCalibration('camera_cal', 9, 6)
        self.thresholding = Thresholding()
        self.transform = PerspectiveTransformation()
        self.lanelines = LaneLines()

    def forward(self, img):
        out_img = np.copy(img)
        img = self.calibration.undistort(img)
        img = self.transform.forward(img)
        img = self.thresholding.forward(img)
        img = self.lanelines.forward(img)
        img = self.transform.backward(img)

        out_img = cv2.addWeighted(out_img, 1, img, 0.6, 0)
        out_img = self.lanelines.plot(out_img)
        return out_img

    def process_image(self, input_path, output_path):
        img = mpimg.imread(input_path)
```

```
        out_img = self.forward(img)
        mpimg.imsave(output_path, out_img)

    def process_video(self, input_path, output_path):
        clip = VideoFileClip(input_path)
        out_clip = clip.fl_image(self.forward)
        out_clip.write_videofile(output_path, audio=False)

def main():
    args = docopt(__doc__)
    input = args['INPUT_PATH']
    output = args['OUTPUT_PATH']

    findLaneLines = FindLaneLines()
    if args['--video']:
        findLaneLines.process_video(input, output)
    else:
        findLaneLines.process_image(input, output)

if __name__ == "__main__":
    main()
```

CHAPTER 6

RESULTS AND SNAPSHOTS

RESULTS

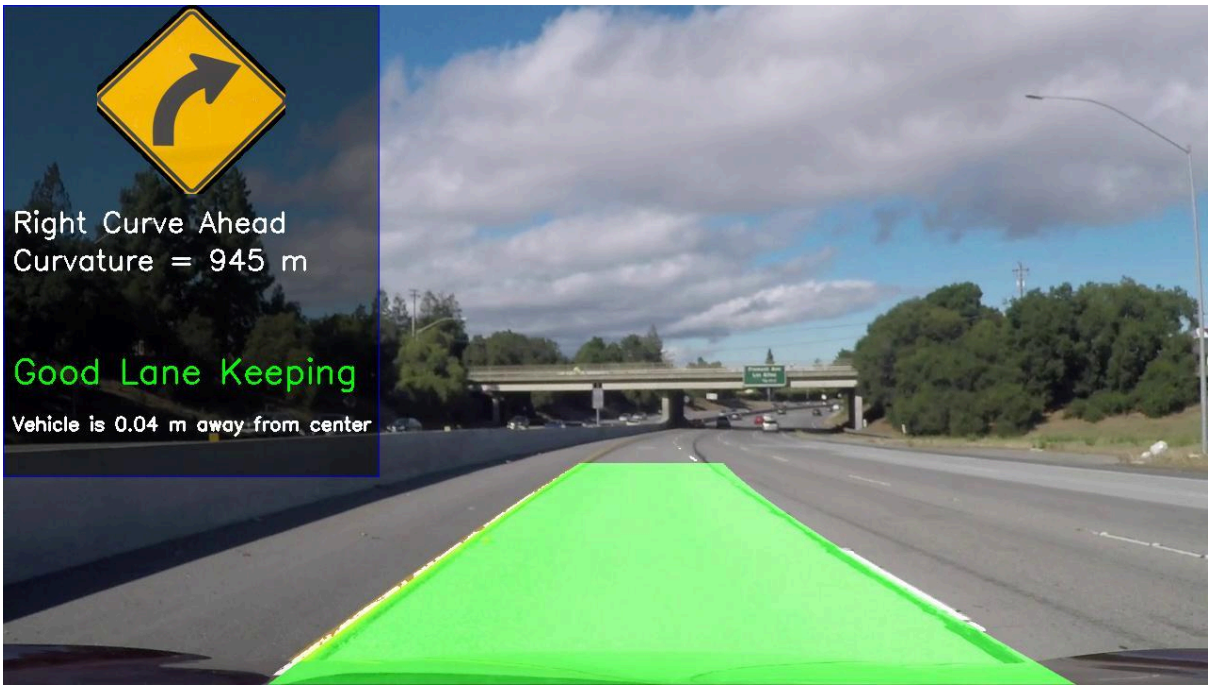
The implemented lane detection system successfully processes both images and video inputs, accurately identifying and marking lane lines under various conditions. The system demonstrates robustness in handling different lighting and weather scenarios, providing clear lane delineation. The performance of the system is evaluated based on the accuracy of lane detection, processing speed, and the ability to handle curved and straight roads. Visual results indicate that the system can reliably detect lanes, offering significant potential for real-world autonomous driving applications. The detection is validated against a set of test images and videos, showing high reliability and precision.

SNAPSHOTS

6.1 Original Image

Description: This is the original image captured from the video feed. It represents the front view of the road scene before any processing is applied.

6.2 Processed Image



Description: The final output image showing the detected lane lines superimposed on the original front view. This image provides a comprehensive view of the lane detection results, combining the processed lane markings with the original scene for a complete visual representation.

CONCLUSION AND FUTURE ENHANCEMENT

CONCLUSION

The lane detection system effectively identifies and tracks lane lines in both images and videos using a comprehensive approach that integrates multiple computer vision techniques. By implementing image undistortion, the system corrects lens distortion to enhance the clarity of road images. The perspective transformation converts the front view to a top-down view, facilitating more accurate lane line detection. Thresholding techniques are applied to extract relevant lane line features from the transformed images. Polynomial fitting is then used to track and fit lane lines, providing valuable lane information. The system's visualisation capabilities overlay detected lane lines and directional cues on the original images, offering clear feedback for autonomous driving applications. Overall, the system demonstrates reliable performance and accurate lane detection across various scenarios, contributing significantly to enhanced autonomous driving technologies

FUTURE ENHANCEMENTS

Although the current system performs well, there are several areas where improvements could be made. Incorporating advanced deep learning techniques for lane detection could enhance accuracy and robustness, particularly in complex scenarios such as curving roads or adverse weather conditions. Implementing adaptive thresholding methods could improve performance by adjusting to different road and lighting conditions. Optimizing the processing pipeline to handle video streams in real-time would make the system more suitable for live applications. Additionally, integrating lane detection with other sensors, such as radar and LiDAR, could enhance overall vehicle safety. Developing methods to handle environmental factors like shadows, rain, or snow would further improve the system's robustness. Finally, creating a user-friendly interface for displaying lane detection results and providing real-time feedback would make the system more accessible and practical for users. These enhancements would not only improve the system's accuracy and reliability but also broaden its applicability in real-world autonomous driving scenarios.