Sensor Fusion and Vision-Based Navigation for Autonomous Vehicle Steering Control

VADALI S N B SRIVATSAV -21BMA0015
SHAM PRASATH K -21BMA0078

Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

School of Mechanical Engineering

Final Review

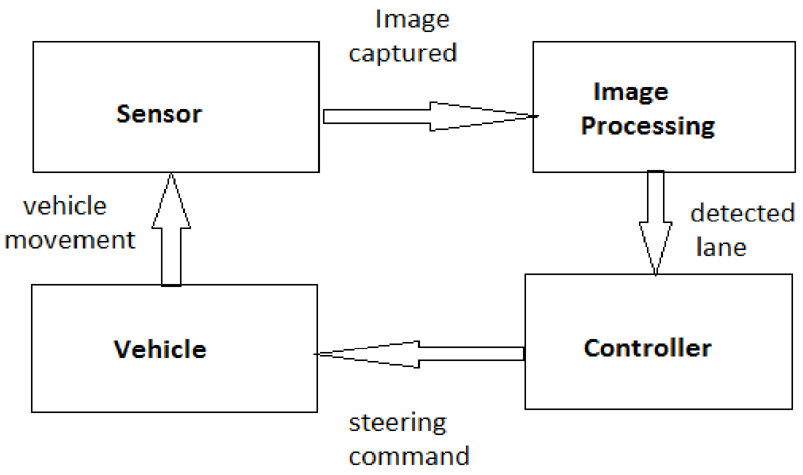
PROJECT GUIDE : PROF. DENIS ASHOK S
PROFESSOR HIGHER GRADE, SMEC

Date of Presentation: 19-3-2025

CONTENTS

Introduction

- Autonomous vehicle steering control and trajectory planning play a crucial role in achieving precision navigation and parking assistance.
- This project focuses on developing a **MATLAB Simulink model** to design and analyze **sensor-based steering control** for an autonomous system.
- By integrating **sensor fusion and vision-based navigation**, we aim to improve **path tracking, obstacle avoidance, and autonomous parking efficiency**.
- The system is designed to aim in ensuring **optimal vehicle maneuverability** through sensor-driven trajectory planning and automated steering assistance.



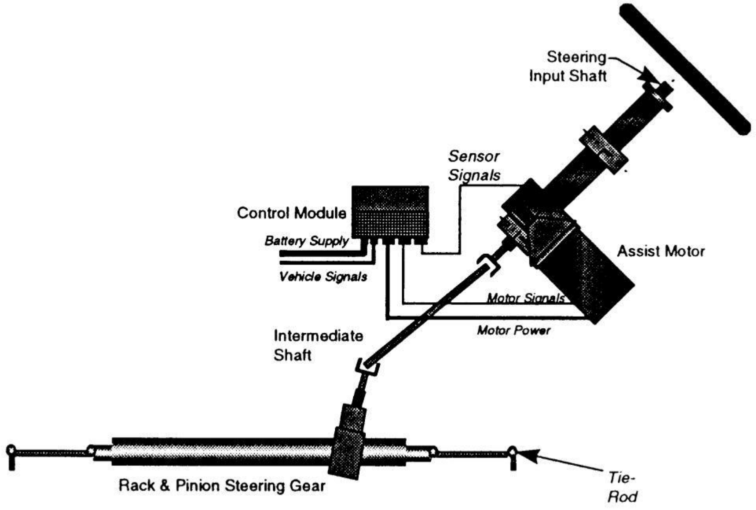


Figure 1. E•Steer™ incorporated in the steering system

Title of the paper	Journal & Year	Authors	Description	Remarks
AVP-SLAM: Semantic Visual Mapping and Localization for Autonomous Vehicles in the Parking Lot	IEEE, (2022)	Tong Qin, Tongqing Chen, Yilun Chen, and Qing Su	• Robust to Challenging Conditions – Uses semantic features like guide signs, parking lines, and speed bumps, which remain stable over time and are robust to lighting and appearance changes. • High-Accuracy Localization – Achieves centimeter-level localization using a global semantic map, surround-view cameras, and additional sensors like an IMU and wheel encoders.	The paper presents a robust and efficient semantic SLAM system tailored for autonomous valet parking, overcoming traditional SLAM limitations by using stable semantic features for long-term, high-accuracy localization in challenging parking lot environments.

Literature Review

Title of the paper	Journal & Year	Authors	Description	Remarks
Trained Trajectory based Automated Parking System using Visual SLAM on Surround View Cameras	IEEE, (2021)	Nivedita Tripathi and Senthil Yogamani	• The paper presents a novel automated parking system that utilizes (Visual SLAM) to create a persistent parking map for frequently used parking spaces (home, office, etc.). • The system captures a vehicle's trajectory using surround-view fisheye cameras, stores the path, and replays it for automated parking. Visual SLAM is leveraged to accurately relocalize the vehicle.	The proposed approach is implemented on commercial automotive systems and tested on real-world parking scenarios. The study highlights the advantages and challenges of deploying such systems in consumer vehicles.

Literature Review

Gaps in the Literature

- **High-Speed Parking Maneuvers**: The literature mainly addresses low-speed parking, missing out on EPS performance during higher-speed parking scenarios.

- **Environmental Variability**: Limited attention is given to how EPS can adapt to varying environmental conditions like slippery surfaces or poor lighting.

- **Hardware Integration**: Most studies focus on software solutions but lack integration with the physical EPS system's mechanical components for practical use.

- **Dynamic Obstacle Response**: While obstacle detection is covered, there is little emphasis on EPS tuning for quick response to moving obstacles in real-time parking.

Knowledge gained from the literature

Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

1.Path Planning: Optimizing parking path planning with smooth curves can guide the steering control system for more efficient parking, reducing sharp turns.

2.Obstacle Detection: Sensor integration and computer vision help in detecting obstacles, ensuring the steering system responds accurately to dynamic parking environments.

3.Localization Techniques: Visual SLAM and semantic mapping can improve vehicle localization in parking scenarios, helping your EPS system navigate tight spots precisely.

4.Visual-Inertial Integration: Combining visual and inertial data minimizes errors in complex parking scenarios, making the steering control more accurate and reliable.

Objectives

Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

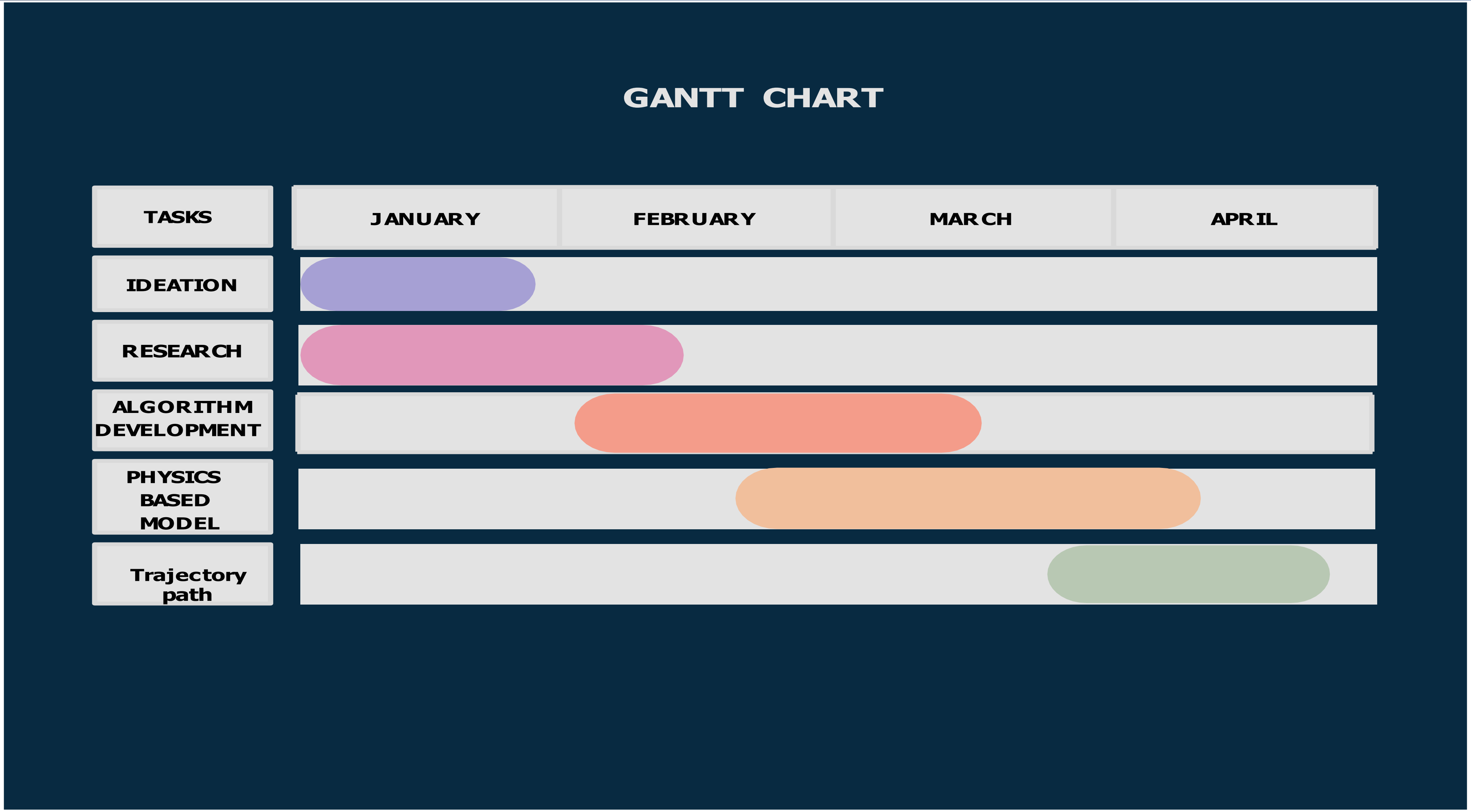
Primary Objective

- Develop a reliable and efficient steering control system using an electric power-assisted mechanism.
- Implement basic trajectory control to guide the vehicle along predefined paths.
- Use control algorithms to maintain accurate lane positioning and smooth transitions.
- Incorporate multiple sensors such as **LiDAR, Radar, and Cameras** to enhance perception and situational awareness.

Secondary Objective

- Ensure that if one system component fails, the others can take over to maintain vehicle movement and safety
- Design algorithms for autonomous parking, including trajectory planning and obstacle avoidance.
- Enhance the system to perform both **parallel and normal parking maneuvers** efficiently.

Milestones


GANTT CHART
TASKS
JANUARY
FEBRUARY
MARCH
APRIL
IDEATION
RESEARCH
ALGORITHM DEVELOPMENT
PHYSICS BASED MODEL
Trajectory path

Methodology

Modelling of control system for steering

↓

Creating a Trajectory path planning using algorithms

↓

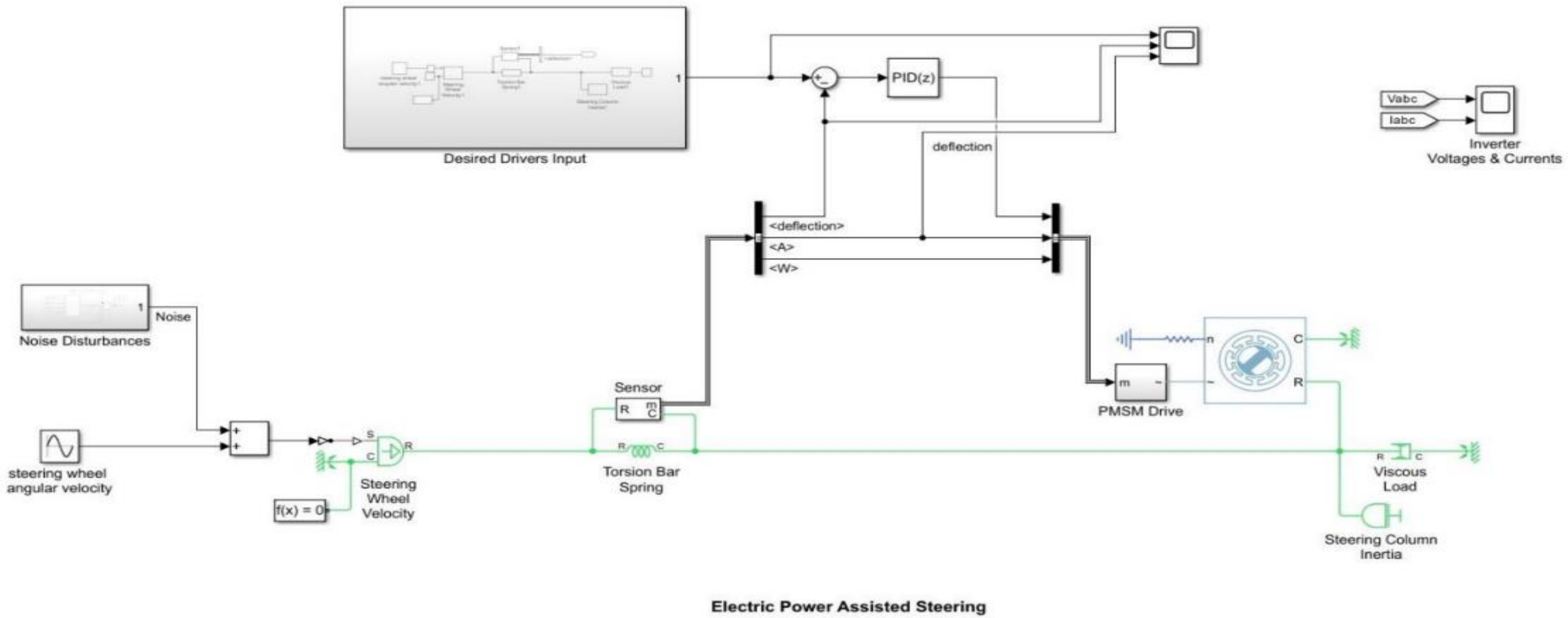
Camera development

↓

Trajectory path

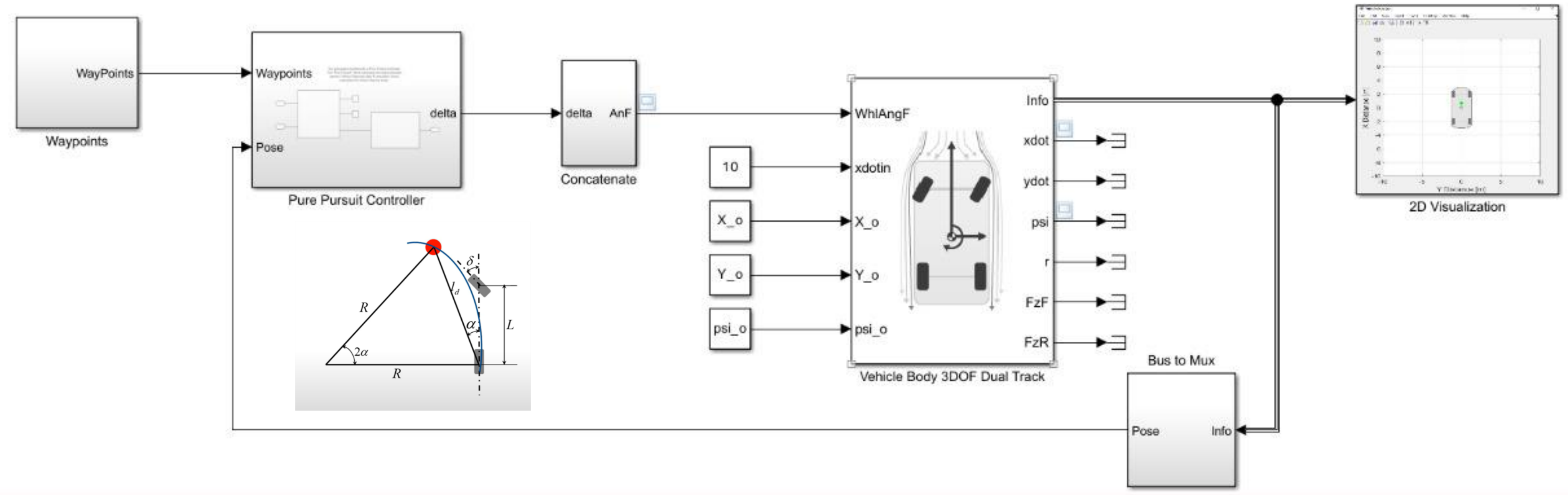
Work carried out so far

Steering control model



Work carried out so far

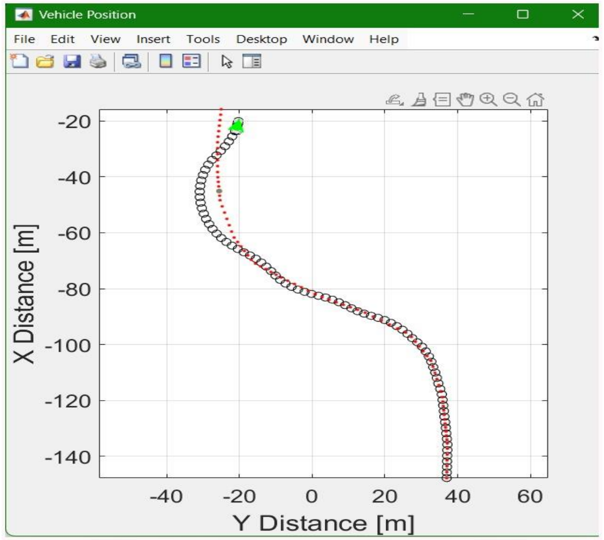
Trajectory and path prediction using pursuit control model



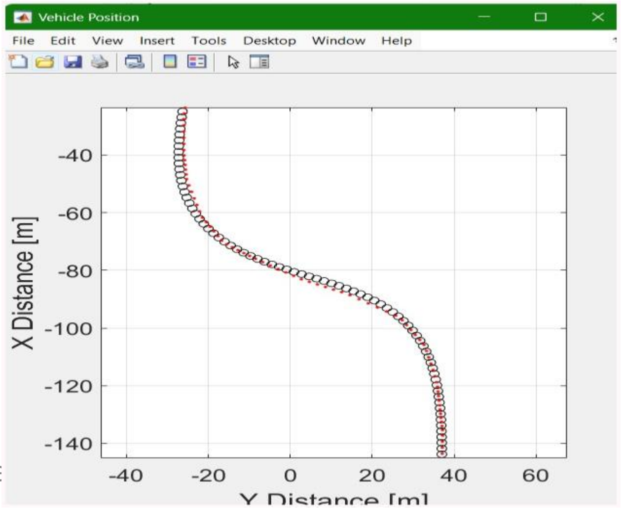
Work carried out so far

CODE:

```
%% add Image to the path
addpath(genpath('Images'));

%% load the scene data file generated from Driving Scenario Designer
load('curveLowVel.mat');

%% define reference points
refPose = data.ActorSpecifications.Waypoints;
xRef = refPose(:,1);
yRef = -refPose(:,2);

%% define reference time for plotting
Ts = 50; % simulation time
s = size(xRef);
tRef = (linspace(0,Ts,s(1)))'; % this time variable is used in the "2D '

%% define parameters used in the models
L = 3; % length
ld =20; % lookahead distance
X_o = refPose(1,1); % initial vehicle position
Y_o = -refPose(1,2); % initial vehicle position
psi_o = 0; % initial yaw angle
```



ld==1
>> avg path prediction
>>high oscillation



ld==20
>> Good path prediction
>> low oscillation

Work carried out so far

Real-Time Object Distance Detection in Dynamic Environments Using OpenCV:

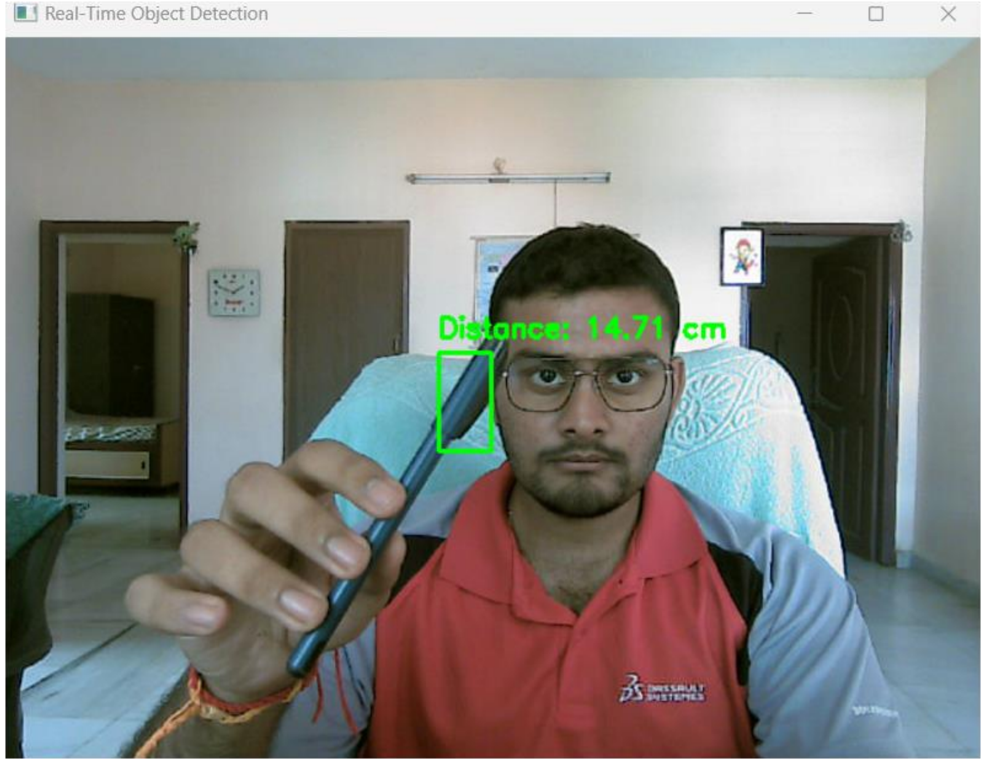
```python
import cv2
import numpy as np

# === Parameters ===
KNOWN_WIDTH_OBJECT = 1.0  # Width of the smallest object (e.g., pin) in m
FOCAL_LENGTH = 500        # Adjust this value after calibration
MAX_DISTANCE = 30.0       # Maximum distance to detect objects in m

# === Distance Calculation Function ===
def calculate_distance(known_width, focal_length, per_width):  1 usage
    """
    Calculate the distance from the object to the camera.
    """
    return (known_width * focal_length) / per_width

# === Camera Initialization ===
cap = cv2.VideoCapture(0)  # Use the default camera

if not cap.isOpened():
    print("Camera not detected.")
    exit()

# === Background Subtractor ===
back_sub = cv2.createBackgroundSubtractorMOG2(history=100, varThreshold=50, detectShadows=False)

print("Press 'q' to quit the application.")
```

```python
# === Object Detection and Distance Calculation ===
closest_distance = MAX_DISTANCE + 1
closest_contour = None

for contour in contours:
    # Filter small contours
    if cv2.contourArea(contour) > 100:
        x, y, w, h = cv2.boundingRect(contour)

        # Calculate distance
        distance = calculate_distance(KNOWN_WIDTH_OBJECT, FOCAL_LENGTH, w)

        # Find the closest object within the distance range
        if distance < closest_distance and distance < MAX_DISTANCE:
            closest_distance = distance
            closest_contour = contour

# === Display the closest object and its distance ===
if closest_contour is not None:
    x, y, w, h = cv2.boundingRect(closest_contour)
    cv2.rectangle(frame, (x, y), (x+w, y+h), (0, 255, 0), 2)
    cv2.putText(frame, text: f"Distance: {closest_distance:.2f} m", org: (x, y-10),
                cv2.FONT_HERSHEY_SIMPLEX, fontScale: 0.6, color: (0, 255, 0), thickness: 2)
    print(f"Object detected at: {closest_distance:.2f} m")
```

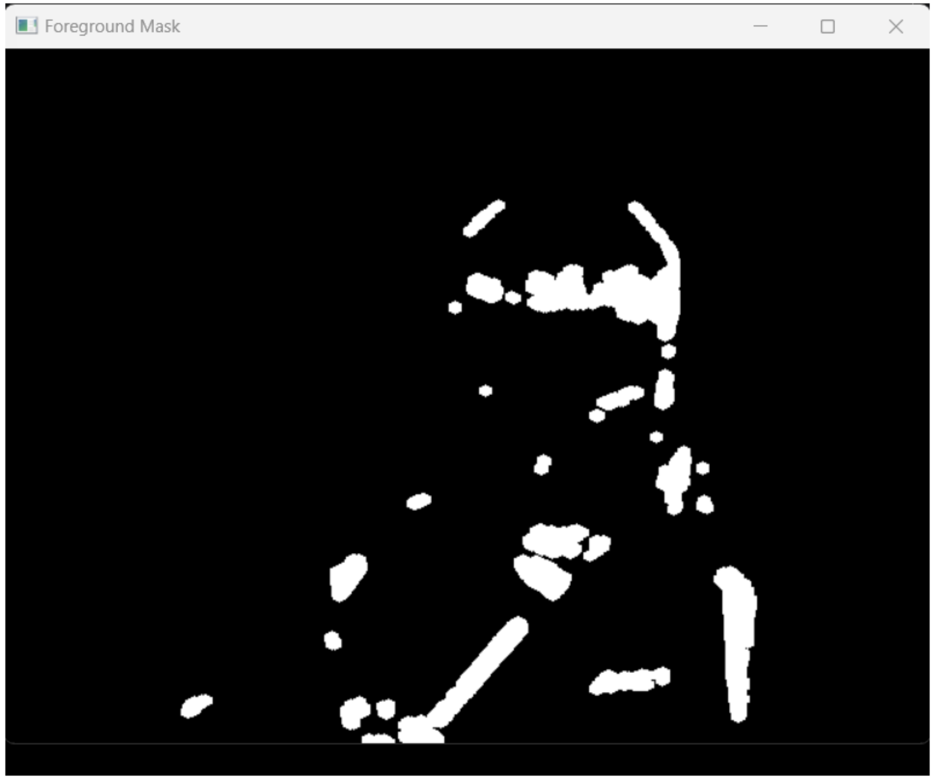
Work carried out so far

Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

OpenCV camera vision



OpenCV Fg Mask



Work carried out so far

Vision based braking and steering system detection

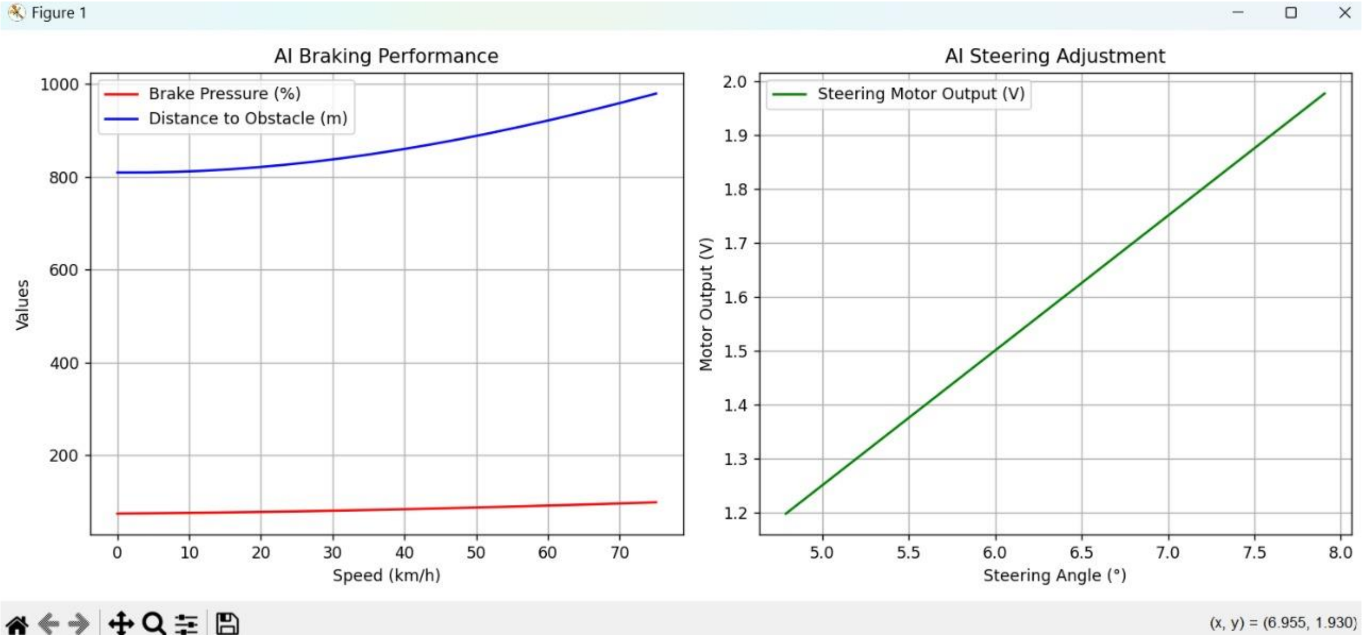
```
class CameraModule:
    def __init__(self, focal_length=700, real_object_width=1.8):
        """Camera module to estimate distance from an obstacle."""
        self.focal_length = focal_length
        self.real_object_width = real_object_width
        self.cap = cv2.VideoCapture(0)

    def get_distance_from_camera(self):
        """Detects motion and estimates real-world distance."""
        ret, frame = self.cap.read()

 def monitor_distance(self, duration=10):
     """Monitors the distance for a given duration to determine if braking is needed."""
     start_time = time.time()
     below_threshold_time = 0

     while time.time() - start_time < duration:
         distance = self.get_distance_from_camera()
         if distance:
             print(f"📏 Camera Detected Distance: {distance}m")
             if distance < 500:
                 below_threshold_time += 1

         time.sleep(1)

     return below_threshold_time >= duration  # Apply brakes if true

 def release_camera(self):
     """Releases the camera resource."""
     self.cap.release()
     cv2.destroyAllWindows()
```

```
Detected Distance: 6.49m
Detected Distance: 10.0m
Detected Distance: 12.48m
Detected Distance: 52.5m
Detected Distance: 315.0m
Detected Distance: 60.0m
Detected Distance: 7.24m
Detected Distance: 28.0m
Detected Distance: 8.75m
Detected Distance: 252.0m
✅ Safe distance maintained: NO BRAKING REQUIRED.
```

Work carried out so far

VIT® Vellore Institute of Technology (Deemed to be University under section 3 of UGC Act, 1956)

```
🚗 Speed: 49.4 km/h | 🔵 Distance: 55.6 m
Brake Pressure: 36.6% | Fuel Flow Reduction: 29.3%
Brake Temperature: 170.0°C | Engine Temp: 97.0°C
➡️ Steering Angle: 2.0° | ⚙️ Steering Motor Output: 0.5V

🚗 Speed: 47.6 km/h | 🔵 Distance: 42.4 m
Brake Pressure: 35.3% | Fuel Flow Reduction: 28.2%
Brake Temperature: 180.0°C | Engine Temp: 97.5°C
➡️ Steering Angle: 1.3° | ⚙️ Steering Motor Output: 0.3V

🚗 Speed: 45.9 km/h | 🔵 Distance: 29.6 m
Brake Pressure: 34.0% | Fuel Flow Reduction: 27.2%
Brake Temperature: 190.0°C | Engine Temp: 98.0°C
➡️ Steering Angle: 0.7° | ⚙️ Steering Motor Output: 0.2V

🚗 Speed: 44.3 km/h | 🔵 Distance: 17.3 m
Brake Pressure: 32.7% | Fuel Flow Reduction: 26.2%
Brake Temperature: 200.0°C | Engine Temp: 98.5°C
➡️ Steering Angle: 0.1° | ⚙️ Steering Motor Output: 0.0V

🚗 Speed: 42.7 km/h | 🔵 Distance: 5.4 m
Brake Pressure: 31.5% | Fuel Flow Reduction: 25.2%
Brake Temperature: 210.0°C | Engine Temp: 99.0°C
➡️ Steering Angle: 0.0° | ⚙️ Steering Motor Output: 0.0V

🚗 Speed: 41.2 km/h | 🔵 Distance: 0.0 m
Brake Pressure: 30.4% | Fuel Flow Reduction: 24.3%
Brake Temperature: 220.0°C | Engine Temp: 99.5°C
➡️ Steering Angle: 0.0° | ⚙️ Steering Motor Output: 0.0V

✅ Car Stopped Safely.

📋 Final AI Safety Report:
Brake Fade: 9.5%
Final Brake Temperature: 220.0°C
Final Engine Temperature: 99.5°C
Brake Fluid Level: 100.0%
Oil Level: 100.0%
```

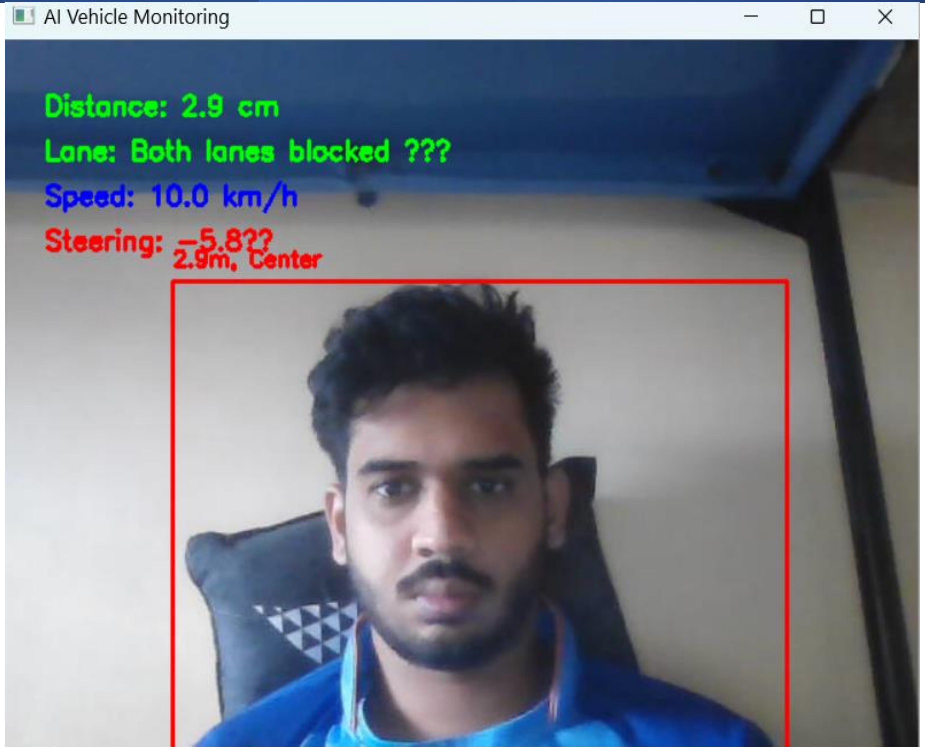


Work carried out so far

CODE:

```
# === Parameters ===
STOP_DISTANCE = 30.0  # Minimum distance before stopping (cm)
KNOWN_WIDTH_OBJECT = 1.0  # Width of the smallest object in cm
FOCAL_LENGTH = 500        # Adjust this value after calibration
MAX_DISTANCE = 100.0      # Maximum distance to detect objects in cm
SAFE_DISTANCE = 50.0      # Safe following distance in cm
MAX_SPEED = 80.0  # Maximum speed (km/h)
MIN_SPEED = 10.0  # Minimum speed to keep moving
```

```
# === Background Subtractor ===
back_sub = cv2.createBackgroundSubtractorMOG2(history=100, varThreshold=50, detectShadows=False)

def estimate_distance(bbox_width, focal_length=700, real_width=1.8):
    """Estimate distance based on bounding box width."""
    return (real_width * focal_length) / (bbox_width + 1e-6)

def determine_lane(x_center, frame_width):
    """Determine which lane the object is in."""
    lane_width = frame_width // 3
    if x_center < lane_width:
        return "Left"
    elif x_center < 2 * lane_width:
        return "Center"
    else:
        return "Right"

def get_safety_lane(current_lane):
    """Determine the safest lane to move to."""
    if current_lane == "Left":
        return "Move to Center or Right"
    elif current_lane == "Center":
        return "Move to Left or Right"
    else:
        return "Move to Center or Left"
```



Work carried out so far

Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

```
lane_status = "Left lane clear ✅" if closest_distance > SAFE_DISTANCE else "Maintaining Safe Distance 🟢" if closest_distance > STOP_DISTANCE else "Both lanes blocked
steering_angle = steering_pid(-1 if "Left" in lane_status else 1)
speed_adjustment = speed_pid(closest_distance)
current_speed = max(MIN_SPEED, min(MAX_SPEED, MAX_SPEED - speed_adjustment))
brake_pressure = max(0, min(100, (SAFE_DISTANCE - closest_distance) / 5))
fuel_reduction = min(100, brake_pressure)
engine_temp = min(110, 90 + (brake_pressure / 2))
brake_temp = min(100, 30 + (brake_pressure / 1.5))

print(f"\n📏 Distance to Obstacle: {closest_distance:.1f} cm")
print(f" Lane Status: {lane_status}")
print(f" Steering Angle: {steering_angle:.1f}°")
print(f" Speed: {current_speed:.1f} km/h | 🟢 Distance: {closest_distance} cm")
print(f" Brake Pressure: {brake_pressure:.1f}% | ⛽ Fuel Flow Reduction: {fuel_reduction:.1f}%")
print(f" Brake Temperature: {brake_temp:.1f}°C | 🔧 Engine Temp: {engine_temp:.1f}°C")
```

OUTPUT:

```
📏 Distance to Obstacle: 2.7 cm
 Lane Status: Both lanes blocked ❌
 Steering Angle: -15.6°
 Speed: 10.0 km/h | 🔵 Distance: 2.721382283539131 cm
 Brake Pressure: 9.5% | ⛽ Fuel Flow Reduction: 9.5%
 Brake Temperature: 36.3°C | 🔧 Engine Temp: 94.7°C

0: 480x640 1 person, 68.9ms
Speed: 2.0ms preprocess, 68.9ms inference, 1.0ms postprocess per image at shape (1, 3, 480, 640)

📏 Distance to Obstacle: 2.7 cm
 Lane Status: Both lanes blocked ❌
 Steering Angle: -15.6°
 Speed: 10.0 km/h | 🔵 Distance: 2.7038626551419256 cm
 Brake Pressure: 9.5% | ⛽ Fuel Flow Reduction: 9.5%
 Brake Temperature: 36.3°C | 🔧 Engine Temp: 94.7°C
```

Work carried out so far

MID-Lane warning

```
import cv2
import numpy as np
from collections import deque

# Initialize HOG detector
hog = cv2.HOGDescriptor()
hog.setSVMDetector(cv2.HOGDescriptor_getDefaultPeopleDetector())

FOCAL_LENGTH = 700
REAL_WIDTH_PERSON = 40  # in cm

cap = cv2.VideoCapture(0)

# Smoothing buffers (deque for moving average)
distance_buffer = deque(maxlen=5)   # Adjust window size for smoother effect
error_buffer = deque(maxlen=5)

while True:
    ret, frame = cap.read()
    if not ret:
        print("Camera feed error.")
        break

    (h, w) = frame.shape[:2]
    center_x = w // 2

    boxes, weights = hog.detectMultiScale(frame, winStride=(8, 8))

    for (x, y, bw, bh), weight in zip(boxes, weights):
        if weight > 0.5:
            object_center_x = x + bw // 2
            error = object_center_x - center_x

            # Distance estimation
            distance = (REAL_WIDTH_PERSON * FOCAL_LENGTH) / bw

            # Store values for smoothing
            distance_buffer.append(distance)
            error_buffer.append(error)
```

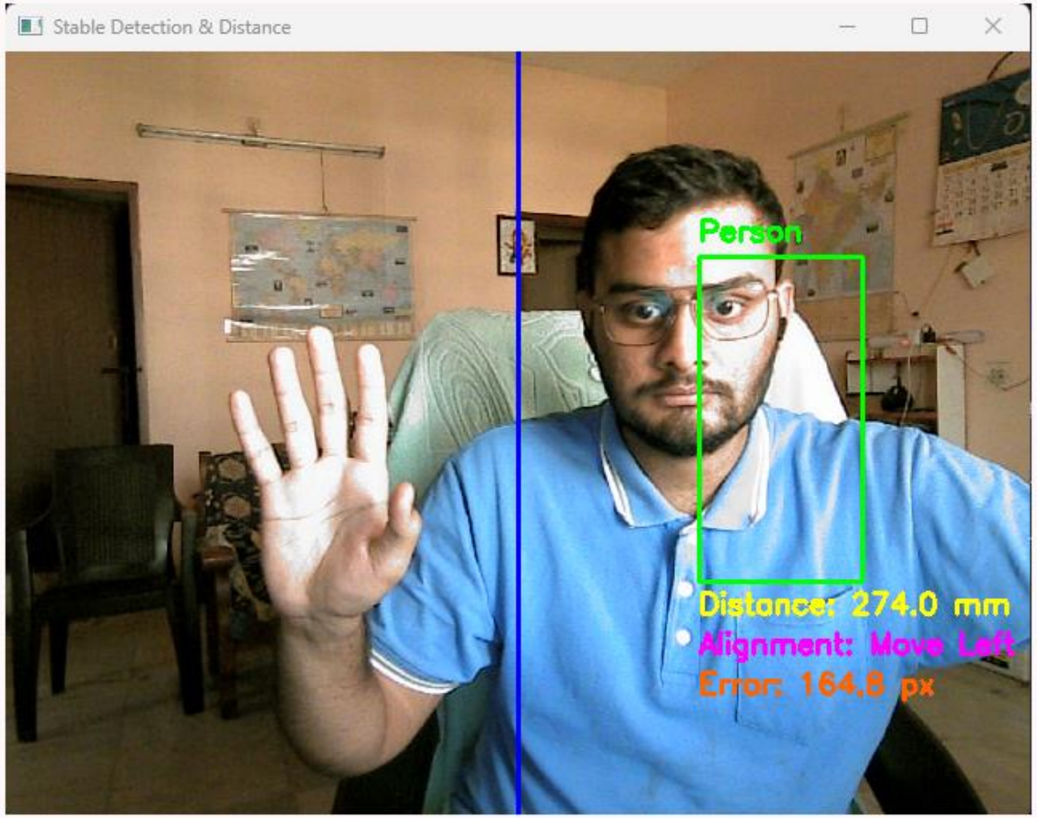
```
    if not ret:
        print("Camera feed error.")
        break

    (h, w) = frame.shape[:2]
    center_x = w // 2

    boxes, weights = hog.detectMultiScale(frame, winStride=(8, 8))

    for (x, y, bw, bh), weight in zip(boxes, weights):
        if weight > 0.5:
            object_center_x = x + bw // 2
            error = object_center_x - center_x

            # Distance estimation
            distance = (REAL_WIDTH_PERSON * FOCAL_LENGTH) / bw

            # Store values for smoothing
            distance_buffer.append(distance)
            error_buffer.append(error)

            # Smooth values using moving average
            smooth_distance = round(np.mean(distance_buffer), 2)
            smooth_error = round(np.mean(error_buffer), 2)

            # Adjustment suggestion logic
            if abs(smooth_error) < 20:
                adjustment = "Centered"
            elif smooth_error < 0:
                adjustment = "Move Right"
            else:
                adjustment = "Move Left"

            # PRINT to console
            print(f"\n[Detection]")
            print(f"  Smoothed Distance: {smooth_distance} mm")
            print(f"  Smoothed Center Error: {smooth_error} px")
            print(f"  Suggestion: {adjustment}")
```

Work carried out so far

User Feedback & Output

•Console output: shows smoothed distance, center error, and movement suggestions.

•Overlay: draws bounding boxes and text over the video feed for user-friendly feedback.



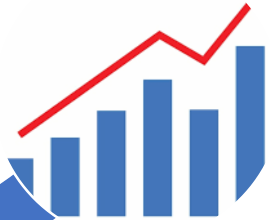
Application of this models

Camera-Centric design

Leverages computer vision to interpret the environment using affordable camera systems.

Enhance image processing

Sophisticated algorithms improve camera accuracy, making it more reliable for mid-lane detection and obstacle avoidance.



Scalable & Sellable

Enables production of more intelligent, more affordable autonomous vehicles for the mass market.

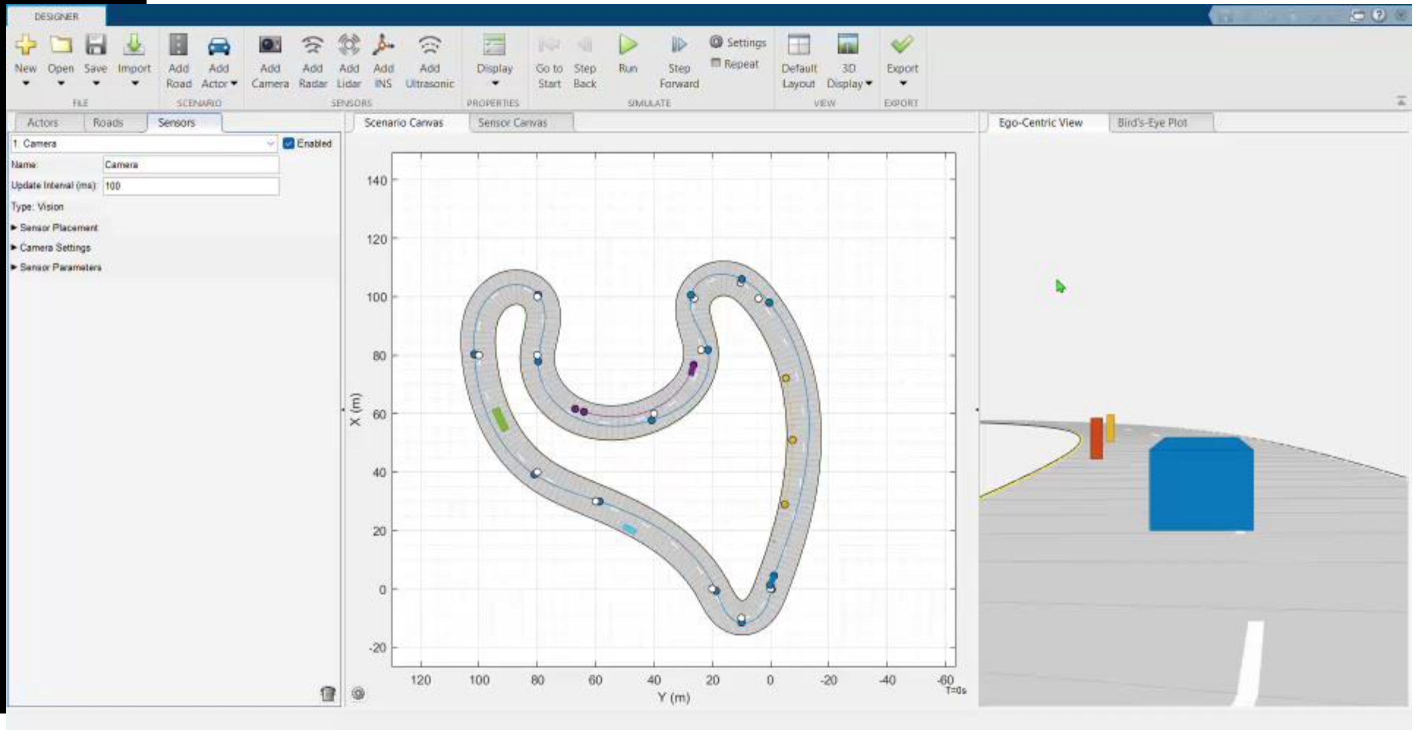
Application of this models

Lidar
$75,000

Radar
$25,000

Camera
$10,000

Final simulation



References

1) Tripathi, N., & Yogamani, S. (2020). Trained Trajectory based Automated Parking System using Visual SLAM on Surround View Cameras. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.2001.02161

2) Qin, T., Chen, T., Chen, Y., & Su, Q. (2020). AVP-SLAM: Semantic visual mapping and localization for autonomous vehicles in the parking lot. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.2007.01813

3) Li, C., Jiang, H., Ma, S., Jiang, S., & Li, Y. (2020). Automatic Parking Path Planning and tracking control research for intelligent vehicles. *Applied Sciences*, *10*(24), 9100. https://doi.org/10.3390/app10249100

4) Chen, C., Wu, B., Xuan, L., Chen, J., Wang, T., & Qian, L. (2020). A trajectory planning method for autonomous valet parking via solving an optimal control problem. *Sensors*, *20*(22), 6435. https://doi.org/10.3390/s20226435

Acknowledgements

We want to acknowledge Prof. Denis Ashok S for his invaluable support and feedback for the Final review of our project. His guidance is crucial as we continue to develop our work.