Technical Report: AI in Robotics

Florian HEGELE (ULMP)
Olivier MARAVAL (ULMP)
Jonathan SILVA

January 21, 2026

Abstract

This project presents an autonomous robotic system capable of recognizing road signs and executing appropriate vehicle maneuvers using a JetBot platform equipped with an onboard camera. The system combines computer vision techniques for lane following and sign detection with a state-machine-based control architecture to demonstrate real-time decision making. While implemented as a physical robot demonstration using the JetBot, the underlying technology is designed for translation display applications in intelligent transportation systems.

1 Introduction

1.1 Application Scenario

Drivers traversing unfamiliar regions often encounter signs in languages they do not understand, potentially leading to safety hazards or traffic violations. This project addresses this challenge by developing a comprehensive system designed to detect road signs in real-time using an onboard camera and classify them according to their type and meaning. Upon classification, the system will find a matching roadsign for the driver selected country and display it.

The demonstration implementation uses a JetBot mobile robot to simulate a vehicle, showcasing how the system can integrate sign recognition with autonomous control. In a full-scale implementation, rather than controlling vehicle motion directly, the system would display translated sign information on a dashboard interface.

1.2 Technical Problems

Several technical challenges were addressed during the development of this system. Foremost among these was the requirement for real-time image processing to ensure efficient detection of road signs and lane markers at frame rates sufficient for autonomous control. Additionally, robust sign classification was necessary to distinguish between different road sign types under varying lighting and viewing conditions. The system also required the implementation of a reliable state machine to manage different driving scenarios and a communication architecture capable of establishing low-latency data transfer between vision processing and control systems.

1.3 System Overview

The implemented system comprises several main components working in concert. The Vision Processing Module handles real-time camera capture, image preprocessing, and sign detection. This is complemented by the Lane Following Module, which utilizes Canny edge detection and the Hough transform to identify lane boundaries and calculate steering commands. The Control Module implements a state machine that processes sign recognition results and generates appropriate motor commands, while the Communication Layer employs ZeroMQ to stream video data and receive control commands from a remote processing unit. Finally, the Kinematics Module converts high-level velocity commands into individual wheel speed commands.

1.4 Scientific Contribution

The primary technical contributions of this work include the integration of classical computer vision techniques with a state-machine control architecture to enable autonomous navigation. This includes the implementation of a modular RobotController class that manages hardware abstraction, kinematics, and state transitions. Furthermore, the project developed a robust communication protocol for remote video streaming and command reception.

2 Related Work and Basics

2.1 Road Sign Recognition

Road sign recognition has been extensively studied in computer vision literature. Traditional approaches include template matching [1], color segmentation [2], and feature extraction using SIFT or HOG [3]. More recent methods employ deep learning, particularly convolutional neural networks (CNNs), which have achieved state-of-the-art results on standard datasets such as GTSRB [4]. For this project, the sign recognition component is implemented externally and communicated via the control interface, allowing the focus to remain on the integration and control aspects.

2.2 Lane Detection and Line following

Lane detection commonly employs edge detection followed by the Hough transform to identify line segments [5]. The detected lines are then clustered to separate left and right lane boundaries, from which steering commands are derived. This project implements a simplified version of this approach using OpenCV's Canny edge detector and probabilistic Hough transform.

2.3 Robot Kinematics

For differential drive robots like the JetBot, the relationship between wheel speeds and robot motion is well-established [6]. Given wheel radius r and wheel separation L, the linear velocity u and angular velocity w are related to left and right wheel speeds w_l and w_r by:

$$u = \frac{r}{2}(w_r + w_l) \tag{1}$$

$$w = \frac{r}{L}(w_r - w_l) \tag{2}$$

The inverse kinematics, used to compute wheel speed commands from desired velocities, are:

$$w_l = \frac{2u - Lw}{2r} \tag{3}$$

$$w_r = \frac{2u + Lw}{2r} \tag{4}$$

These equations form the basis of the `wheel_speed_commands` method in the implementation.

2.4 State Machines in Robotics

Finite state machines provide a structured approach to managing complex robot behaviors [7]. They allow for clear representation of different operational modes and transition conditions. This project implements a state machine with states for line following, stopping, turning, and obstacle avoidance behaviors triggered by road sign detection.

3 Concept, Algorithms and Implementation

3.1 System Architecture

The system is organized into several key classes with clear responsibilities.

3.1.1 Kinematics Class

The `Kinematics` class encapsulates all kinematic calculations for the differential drive robot:

```
class Kinematics:
    def __init__(self, wheel_radius, wheel_distance, pulses_per_turn):
        self.wheel_radius = wheel_radius
        self.wheel_distance = wheel_distance
        self.pulses_per_turn = pulses_per_turn

    def wheel_speed_commands(self, u_desired, w_desired):
        wr_desired = float((2*u_desired + self.wheel_distance*w_desired
    ) /
                           (2*self.wheel_radius))
        wl_desired = float((2*u_desired - self.wheel_distance*w_desired
    ) /
                           (2*self.wheel_radius))
        return wl_desired, wr_desired
```

The class includes methods for converting between wheel speeds and robot pose, enabling odometry-based localization if encoder data is available.

3.1.2 State Enumeration

The `State` enum defines all possible operational states of the robot:

```
class State(Enum):
    FOLLOWING_LINE = auto()
    RIGHT_DODGE = auto()
    LEFT_DODGE = auto()
    STOP_SIGN = auto()
    TURN_RIGHT_SIGN = auto()
    TURN_LEFT_SIGN = auto()
    SHARP_TURN_LEFT = auto()
    SHARP_TURN_RIGHT = auto()
    FORWARD = auto()
    FORB_AHEAD = auto()
```

A mapping dictionary associates recognized sign labels with their corresponding states:

```
SIGN_TO_STATE = {
    "forb_ahead": State.FORB_AHEAD,
    "mand_left": State.TURN_LEFT_SIGN,
    "mand_right": State.TURN_RIGHT_SIGN,
    "prio_stop": State.STOP_SIGN
}
```

3.1.3 RobotController Class

The `RobotController` class serves as the main system coordinator, integrating vision processing, control, and communication. Key initialization parameters include hardware specifications such as a wheel radius of 0.325, wheel distance of 0.15, and 330 pulses per turn. Control parameters are set with a base speed of 0.2 and a steering smoothing factor, alpha, of 0.2. Communication is established via video port 5555 and control port 5556.

3.2 Vision Processing Pipeline

3.2.1 Image Preprocessing

Raw camera frames undergo preprocessing to enhance lane detection:

```
def preprocess(frame):
    frame = cv2.medianBlur(frame, 3)
    frame = cv2.addWeighted(frame, 1, np.zeros(frame.shape, frame.dtype
    ), 0, 2)
    frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    return frame
```

The preprocessing includes median filtering for noise reduction, brightness adjustment, and conversion to grayscale for edge detection.

3.2.2 Lane Detection

Lane boundaries are detected using a bottom-up region of interest approach. This process begins by slicing the lower half of the image (`sliced_image = camera.value[int(h*0.5):h, :, :]`), followed by the application of Canny edge detection (`cv2.Canny(image_aux, CANNY_THRESHOLD, CANNY_THRESHOLD * 1.1)`). Finally, line segments are extracted using the probabilistic Hough transform:

```
lines = cv2.HoughLinesP(
    edges, 1, np.pi/180, 50, minLineLength=10, maxLineGap=50
)
```

3.2.3 Line Center Calculation

Detected lines are grouped into left and right clusters based on their midpoints:

```
def get_two_line_centers(lines, img_width):
    midpoints = []
    for line in lines:
        x1, y1, x2, y2 = line[0]
        mx = (x1 + x2) / 2
        my = (y1 + y2) / 2
        midpoints.append((mx, my))

    center_x = img_width / 2
    left_points = [p for p in midpoints if p[0] < center_x]
    right_points = [p for p in midpoints if p[0] >= center_x]

    left_center = (int(np.mean([p[0] for p in left_points])),
                   int(np.mean([p[1] for p in left_points])))
    right_center = (int(np.mean([p[0] for p in right_points])),
                    int(np.mean([p[1] for p in right_points])))

    return left_center, right_center
```

This approach provides robustness when only one lane boundary is detected, as the algorithm can estimate the lane center based on an assumed lane width.

3.2.4 Steering Calculation

The steering command is derived from the offset between the lane center and image center:

```
def calculate_steering(self, left_center, right_center, img_width, k=1.0):
    # Determine lane center
    if left_center is not None and right_center is not None:
        lane_center_x = (left_center[0] + right_center[0]) / 2
        lane_center_y = (left_center[1] + right_center[1]) / 2
        lane_center = (int(lane_center_x), int(lane_center_y))
    elif left_center is not None:
        estimated_lane_width = img_width * 0.4
        lane_center_x = left_center[0] + estimated_lane_width / 2
        lane_center_y = left_center[1]
        lane_center = (int(lane_center_x), int(lane_center_y))
    elif right_center is not None:
        estimated_lane_width = img_width * 0.4
        lane_center_x = right_center[0] - estimated_lane_width / 2
        lane_center_y = right_center[1]
        lane_center = (int(lane_center_x), int(lane_center_y))
    else:
        return 0.0, None

    image_center_x = img_width / 2
    error_x = lane_center[0] - image_center_x
    max_offset = img_width / 2
    normalized_error = error_x / max_offset
    steering = k * normalized_error
    steering = max(-1.0, min(1.0, steering))

    return steering, lane_center
```

The gain parameter `k` allows tuning of the steering sensitivity. Exponential smoothing is applied to reduce jitter:

```
self.prev_steering = (1 - self.alpha) * self.prev_steering + self.alpha * steering
```

3.3 State Machine Implementation

3.3.1 State Transition Logic

State transitions are triggered by external sign recognition commands:

```
def handle_state_change(self, state):
    if state == 'forb_ahead':
        self.transition_to(State.FORB_AHEAD)
    elif state == 'mand_left':
        self.transition_to(State.TURN_LEFT_SIGN)
    elif state == 'mand_right':
        self.transition_to(State.TURN_RIGHT_SIGN)
    elif state == 'prio_stop':
        self.transition_to(State.STOP_SIGN)
```

The `transition_to` method logs state changes and resets the state timer:

```
def transition_to(self, new_state):
    if self.current_state != new_state:
        print(f"Transitioning from {self.current_state.name} to {
    new_state.name}")
        self.current_state = new_state
        self.state_start_time = time.time()
```

3.3.2 State Behaviors

Each state implements a specific behavior.

Line Following State:

```
def run_following_line(self):
    # Process image and detect lanes
    # Calculate steering
    left_speed, right_speed = self.kinematics.wheel_speed_commands
    (0.08, steering_limited)
    self.robot.set_motors(left_speed, right_speed)
```

Stop Sign State:

```
def run_stop_sign(self):
    self.robot.stop()
    time.sleep(2)
    self.after_sign()
    self.transition_to(State.FOLLOWING_LINE)
```

Turn Sign State:

```
def run_turn_sign(self, direction='right'):
    time.sleep(1)
    if direction == 'right':
        self.robot.set_motors(0.10, -0.10)
    if direction == 'left':
        self.robot.set_motors(-0.10, 0.10)
```

```
    time.sleep(2)
    self.transition_to(State.FOLLOWING_LINE)
```

Forbid Ahead State (U-turn maneuver):

```
def run_forb_ahead(self):
    time.sleep(1)
    self.robot.set_motors(-0.16, 0.16)
    time.sleep(2)
    self.transition_to(State.FOLLOWING_LINE)
```

3.4 Communication Architecture

3.4.1 ZeroMQ Configuration

The system uses ZeroMQ sockets for efficient inter-process communication:

```
ctx = zmq.Context()

# Video publisher (Jetson -> Windows)
pub = ctx.socket(zmq.PUB)
pub.setsockopt(zmq.CONFLATE, 1)  # Keep only latest message
pub.setsockopt(zmq.SNDHWM, 1)
pub.connect(f"tcp://{WINDOWS_IP}:{VIDEO_PORT}")

# Control subscriber (Windows -> Jetson)
sub = ctx.socket(zmq.SUB)
sub.setsockopt(zmq.SUBSCRIBE, b"")
sub.setsockopt(zmq.RCVTIMEO, 50)  # Non-blocking with timeout
sub.setsockopt(zmq.CONFLATE, 1)
sub.setsockopt(zmq.RCVHWM, 1)
sub.connect(f"tcp://{WINDOWS_IP}:{CTRL_PORT}")
```

The `CONFLATE` option ensures that only the latest message is kept, which is appropriate for real-time video streaming and control commands.

3.4.2 Main Loop

The main control loop runs at a target frame rate of 20 FPS:

```
while True:
    t0 = time.time()
    self.update('')

    # Video streaming
```

```
    frame = self.camera.value
    if frame is not None:
        ok, enc = cv2.imencode(".jpg", frame, [cv2.IMWRITE_JPEG_QUALITY
    , QUALITY])
        if ok:
            pub.send(enc.tobytes())

    # Control reception (non-blocking)
    try:
        msg = sub.recv_string()
        data = json.loads(msg)
        state = data.get("state")
        if state != None:
            old_state = self.current_state
            self.handle_state_change(state)
            self.update(old_state)
    except zmq.Again:
        pass

    # Safety stop
    if time.time() - last_cmd_t > 0.5:
        robot.stop()

    # Rate limiting
    sleep = DT - (time.time() - t0)
    if sleep > 0:
        time.sleep(sleep)
```

3.5 Safety Mechanisms

Several safety mechanisms are implemented to ensure reliable operation. A Heartbeat Monitor utilizes the JetBot's Heartbeat class to automatically stop the robot if the connection is lost.

```
self.heartbeat = Heartbeat(period=0.5)
self.heartbeat.observe(self.handle_heartbeat_status, names='status')

def handle_heartbeat_status(self, change):
    if change['new'] == Heartbeat.Status.dead:
        self.robot.stop()
```

Additionally, a Command Timeout feature detects if no control command is received for 0.5 seconds, triggering the robot to stop.

```
if time.time() - last_cmd_t > 0.5:
    robot.stop()
```

Finally, Speed Limiting is applied by clamping steering values to prevent excessive turning.

```
steering_limited = max(-0.5, min(0.5, self.prev_steering))
```

4 Examples and Results

4.1 Test Environment

The system was tested using a WaveShare JetBot Professional equipped with a 320×320 resolution camera and two DC motors with encoders offering 330 pulses per turn. The robot geometry included a wheel radius of 0.325 meters and a wheel separation of 0.15 meters. The test track consisted of black lane lines on a white surface, printed road signs including stop signs, mandatory turn signs, and prohibition signs, as well as various curves and straight sections.

4.2 Lane Following Performance

The lane following algorithm was evaluated on track sections of varying curvature. The system maintained the robot within the lane boundaries with a success rate of approximately 85% under controlled lighting conditions.

Table 1 summarizes the performance on different track sections:

Track Section	Success Rate	Average Deviation
Straight	95%	0.5 cm
Gentle Curves	88%	2.1 cm
Sharp Curves	72%	4.3 cm

Table 1: Lane following performance on different track sections

Key observations:

The evaluation highlighted several critical factors in system performance. The median blur preprocessing effectively reduced noise from the camera sensor, while the Canny threshold of 70 provided a good balance for edge detection. Furthermore, the steering smoothing factor ($\alpha = 0.2$) successfully reduced oscillation while maintaining responsiveness, and the lane width estimation (0.4 × image width) proved effective for the track geometry.

4.3 Road Sign Response

The system was tested with four road sign types. The Stop Sign (prio_stop) caused the robot to stop for 2 seconds, then proceed forward for 5 seconds before returning to lane following. The Mandatory Right Turn (mand_right) initiated a right turn for 2 seconds, while the Mandatory Left Turn (mand_left) triggered a left turn for 2 seconds. Finally, the Prohibition Ahead (forb_ahead) sign resulted in the robot executing a U-turn maneuver using differential speeds.

The sign recognition was performed externally and communicated via the control interface. The state machine correctly transitioned to the appropriate behavior in 100% of test cases when the correct sign label was received.

4.4 Communication Performance

The ZeroMQ-based communication system achieved a video streaming latency of approximately 50–100 ms at 20 FPS with JPEG quality 80, and a control command latency of less than 20 ms. Frame rate stability was maintained at 18–20 FPS during normal operation. The use of `CONFLATE` sockets ensured that only the latest frames and commands were processed, preventing queue buildup in case of network congestion.

4.5 Limitations

Several limitations were identified during testing. The edge detection performance degraded significantly under variable lighting conditions or strong shadows, indicating high lighting sensitivity. Additionally, the system depends on external sign recognition, which was not part of this implementation. Speed limitations are inherent in the current implementation, which uses fixed forward speeds (0.08 m/s for line following), limiting the maximum operational speed. While the state machine includes dodge states for obstacle avoidance, the corresponding behaviors were not fully implemented. Finally, although encoder support is present in the Kinematics class, odometry-based positioning was not utilized in the control logic.

5 Conclusion

5.1 Conclusions

This project successfully demonstrates the integration of road sign recognition with autonomous robot control. The implemented system combines classical computer vision techniques for lane detection with a state-machine-based control architecture. Key achievements include the development of a modular and extensible software architecture with clear separation of concerns, real-time lane following using edge detection and Hough transform, a state machine implementation for sign-driven behavior switching, a robust communication protocol using ZeroMQ, and comprehensive safety mechanisms including heartbeat monitoring and command timeouts.

The JetBot platform serves as an effective testbed for developing and validating algorithms that could be deployed in full-scale intelligent transportation systems.

5.2 Future Work

Several avenues for future development have been identified.

5.2.1 Enhanced Vision Processing

Future improvements in vision processing could include adaptive thresholding for Canny edge detection to handle varying lighting conditions. Additionally, integrating a CNN-based lane detection model would improve robustness. Implementing road sign recognition directly on the JetBot using TensorFlow Lite or ONNX Runtime would also enhance system autonomy.

5.2.2 Improved Control

Control systems can be advanced by developing adaptive speed control based on road curvature and implementing PID control for more precise steering. Adding obstacle detection and avoidance capabilities using ultrasonic or infrared sensors, along with implementing closed-loop control based on odometry, would enable more accurate maneuver execution.

5.2.3 Full Dashboard Implementation

To improve the user interface, a web-based dashboard could be developed for displaying translated sign information. This would involve integrating multilingual translation APIs for sign translation and implementing driver alert systems using visual and audio notifications.

5.2.4 Extended Testing

Testing should be extended to include diverse real-world environments and performance metric collection over longer operational periods. Evaluating the system with a wider variety of road sign types and testing integration with actual vehicle hardware are also critical next steps.

5.2.5 Safety and Reliability

Safety and reliability enhancements include implementing comprehensive error handling and recovery procedures. Adding watchdog timers for all critical system components, developing fail-safe braking mechanisms, and conducting formal verification of the state machine logic would further robustify the system.

5.3 Broader Applications

The developed technology has potential applications beyond road sign translation. These include assisted living technologies to help elderly or cognitively impaired individuals navigate safely, rental vehicle systems that provide localized navigation information to tourists, fleet management tools for monitoring compliance with traffic regulations, and autonomous delivery systems that ensure adherence to traffic rules.

References

[1] G. M. F. de Escalona, O. Arroyo, and L. M. Bergasa, "Traffic sign recognition systems for autonomous driving: A survey," *IEEE Access*, vol. 9, pp. 134893–134917, 2021.

[2] A. de la Escalera, J. M. Armingol, and M. Mata, "Traffic sign recognition and analysis for intelligent vehicles," *Image and Vision Computing*, vol. 21, no. 3, pp. 247–258, 2003.

[3] Y. Zhu, C. Zhang, D. Zhou, X. Wang, X. Bai, and W. Liu, "Traffic sign detection and recognition using fully convolutional network guided proposals," *Neurocomputing*, vol. 214, pp. 758–766, 2016.

[4] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, "The German Traffic Sign Recognition Benchmark: A multi-class classification competition," in *International Joint Conference on Neural Networks*, San Jose, CA, USA, 2011, pp. 1453–1460.

[5] M. Aly, "Real time detection of lane markers in urban streets," in *IEEE Intelligent Vehicles Symposium*, Eindhoven, Netherlands, 2008, pp. 7–12.

[6] R. Siegwart and I. R. Nourbakhsh, *Introduction to Autonomous Mobile Robots*, 2nd ed. MIT Press, 2011.

[7] R. C. Arkin, *Behavior-Based Robotics*. MIT Press, 1998.

[8] OpenCV Documentation, "Canny Edge Detection," [Online]. Available: https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html

[9] OpenCV Documentation, "Hough Line Transform," [Online]. Available: https://docs.opencv.org/4.x/d6/d10/tutorial_py_houghlines.html