ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug1405

Layer-by-Layer Perception, Sensor Fusion and Closed-Loop Control in a Low-Cost Modular Autonomous Ground Vehicle: Design, Wiring and Simulation-Based Characterisation

Srikanth Annamareddy[1*]; Komal Sai Raj Atmakuri[2]; Sai Revanth Singavarapu[3]; Durga Venkatesh Janaki[4]

[1]Professor of Practice, Department of Computer Science and Engineering, Koneru Lakshmaiah Education Foundation, Vaddeswaram, AP, India
[2]Department of Electronics and Communication Engineering, Koneru Lakshmaiah Education Foundation, Vaddeswaram, AP, India
[3]Department of Computer Science and Engineering, Koneru Lakshmaiah Education Foundation, Vaddeswaram, AP, India
[4]Department of Mechanical Engineering, Aditya University, Surampalem, Andhra Pradesh, India – 533437

Corresponding Author: Srikanth Annamareddy[1*]

Publication Date: 2026/09/12

Abstract: Most papers on autonomous-vehicle software either stop at a literature survey, or describe a full industrial stack that assumes lidar-grade sensing and a rack of GPUs. Not much work actually shows the pin-out and the line of firmware, how the textbook sense-perceive-plan-act loop actually runs on hardware a student can put together in a weekend. That is the gap this paper tries to close. We start by going back through the layered autonomy pipeline, sensing and fusion, perception, localisation, planning, and control, and tying each layer to a decision we actually made while building the vehicle, framed throughout against the SAE J3016 taxonomy of driving-automation levels, operational design domain, and dynamic driving task. We then walk through, wiring diagram in hand, a two-tier prototype built around a Raspberry Pi 4 for vision and decision-making and an Arduino Uno for time-critical sensing and motor control, talking to each other over a 115200-baud serial link. The perception pipeline is given layer by layer, greyscale, blur, Canny, ROI mask, Hough transform, with the OpenCV code alongside it, and the fusion stage is a scalar Kalman filter combining ultrasonic range with inertial motion. Since physical field trials were not something we could run for this paper, we built our own seeded, reproducible Python simulation of the vehicle's sensors and control loops instead of just describing what we expect would happen, and we report the numbers that code actually produced. Every quantitative result in this paper comes from a seeded software simulation, not physical hardware trials. Root-mean-square ultrasonic error drops from 3.80 ± 0.89 cm raw to 1.67 ± 0.72 cm after the two-stage filter (averaged across 50 seeded runs); the lane pipeline detects a lane in every one of 300 synthetic frames with a mean offset error of 0.74 pixels; and, in a forward-collision scenario where a camera-detected lead vehicle brakes hard from 40 km/h, the adaptive-following and collision-warning logic escalates through CAUTION, WARNING, and AUTOBRAKE states and brings the ego vehicle to a stop with 2.7-3.3 m of gap still remaining, no collision, in both a moderate and a harder braking test. We close by tying these results back to the literature and by laying out what it would take to move this platform toward ROS 2, a learned perception model, and CARLA-based and physical testing. Every wiring connection, pin assignment, and piece of simulation code needed to reproduce this paper is reported in full.

Keywords: *Autonomous Ground Vehicle, Sensor Fusion, Kalman Filter, OpenCV, Canny Edge Detection, Hough Transform, Finite-State Planning, PID Control, Robot Operating System, Embedded Systems, Controller Area Network, Raspberry Pi, Arduino.*

How to Cite: Srikanth Annamareddy; Komal Sai Raj Atmakuri; Sai Revanth Singavarapu; Durga Venkatesh Janaki (2026) Layer-by-Layer Perception, Sensor Fusion and Closed-Loop Control in a Low-Cost Modular Autonomous Ground Vehicle: Design, Wiring and Simulation-Based Characterisation. *International Journal of Innovative Science and Research Technology*, 11(8), 3403-3419. https://doi.org/10.38124/ijisrt/26aug1405

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug1405

I. INTRODUCTION

Most road crashes are not caused by mechanical failure. They are caused by a lapse in human attention, judgement, or perception at exactly the wrong moment. That fact alone has driven three decades of research into vehicles that can sense the road themselves and take over part of the driving task. The SAE formalised this into six levels of automation, from none at all to full automation where no human oversight is needed [1]. In between sits a space that is already commercially real: adaptive cruise control, lane-keeping assist, automated parking, and, in a handful of cities, driverless robotaxis.

An autonomous vehicle, as we use the term here, does three things on a loop: it senses its surroundings, it decides what to do about what it senses, and it acts on that decision through steering, throttle, and brakes. Stated that way it sounds simple. It is not. Making it work means getting computer vision, probabilistic estimation, motion planning, real-time control, and embedded networking to all cooperate on a hard timing budget, and each of those is itself a research field on its own. Most of the published pipeline for how to do this comes from full-scale industrial platforms with the budget to match. That is fine if you are a company building a production car. It is less fine if you are a student or a two-person lab trying to learn the pipeline hands-on.

This paper is our attempt to close that gap with something you can actually build, not another survey. Section II works through the technical literature behind each layer of the pipeline, and unlike a typical survey we tie every subsection to a decision we made later in the build. Sections III through VIII then walk through a low-cost, two-computer prototype, wiring connections, pin assignments, filter equations, and working code included, built from a Raspberry Pi 4, an Arduino Uno, and a small set of sensors anyone can buy. Section IX reports what happens when we actually run that pipeline, in a seeded simulation harness we built for this purpose, and shows the results graphically. Section X talks honestly about the gap that still separates a platform like this from a deployable product, and Section XI closes the paper.

One thing worth saying up front: nothing here claims a shoebox-sized differential-drive robot is a stand-in for a production autonomous vehicle. To state this precisely in J3016's own terms: this platform makes no SAE level claim whatsoever, disclaiming Level 1 through Level 5 equally, and is offered only as a demonstration of the functional building blocks, sensing and fusion, perception, planning, and fail-safe control, that those levels are constructed from, within the narrow ODD stated in Section III. What we do claim is that the underlying lessons, don't trust one sensor alone, use a predict-correct filter instead of a single reading, check the safety condition before the comfort condition on every cycle, are visible and teachable at this scale. And having the full wiring diagram and code in one place removes the biggest obstacle we ran into ourselves when we started: not knowing which wire goes where.

II. LITERATURE REVIEW

This section surveys the principal building blocks of an autonomous-vehicle software stack in the order that data actually flows through it: automation taxonomy, sensing and fusion, perception, localisation and mapping, planning and control, simulation and middleware, and, finally, where the present work sits relative to all of it.

➢ *Levels of Driving Automation*

The SAE J3016 taxonomy is the most widely cited reference for classifying automated-driving capability and has been adopted internationally by regulators as common vocabulary [1]. It is a functional taxonomy rather than a technical one: it does not prescribe how a given level is to be achieved, only what portion of the dynamic driving task is delegated to the machine and under what operating conditions. That distinction matters for a project like ours, because a small prototype can meaningfully demonstrate the same functional building blocks used at higher SAE levels, perception, planning, and control, without approaching the operational scope those higher levels imply.

J3016 also provides more specific vocabulary that we adopt explicitly throughout the remainder of this paper rather than paraphrasing informally. The standard defines an Operational Design Domain (ODD): the specific conditions, road type, speed range, lighting, and environment, under which a driving automation system is designed to function. It defines the Dynamic Driving Task (DDT): the real-time operational and tactical functions required to operate a vehicle, which the standard further decomposes into lateral control, longitudinal control, and Object and Event Detection and Response (OEDR). And it defines a fallback-ready user: an agent, human or otherwise, capable of resuming safe control when the system requests it or reaches the edge of its ODD. We use each of these terms explicitly in the sections that follow to tie our own design decisions back to the standard's framework, rather than only to the six-level taxonomy in isolation.

➢ *Multi-Sensor Perception and Fusion*

No single sensing modality is sufficient on its own. Cameras provide dense semantic information, colour, texture, and text, but degrade under poor illumination and cannot directly measure distance; radar measures range and radial velocity reliably in adverse weather but returns comparatively coarse shape information; scanning lidar produces an accurate three-dimensional point cloud at a cost well above either alternative. Kocic et al. survey exactly this complementary trade-off across a broad range of production and research sensor suites and argue that combining modalities, sensor fusion, is necessary to obtain an estimate that is simultaneously accurate, weather-robust, and rich enough to support classification [2]. A recurring theme in that literature is that fusion can happen at different stages, at the level of raw signals, of intermediate features, or of final object-level decisions, and each choice trades computational cost against robustness. Our own fusion stage, described in Section VI, is deliberately of the simplest kind, a scalar filter over two

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug1405

channels, precisely so that the trade-off is visible rather than hidden inside a larger framework.

➢ *Perception: Detection, Classification, and Segmentation*

Since the resurgence of deep learning, convolutional neural networks have become the dominant approach for turning raw pixels into object-level understanding. Krizhevsky et al. demonstrated the scale of the gain a deep convolutional network could achieve on large-scale image classification, a result widely credited with restarting interest in the field [10]. Girshick et al. framed detection as a two-stage propose-then-classify pipeline [11], later refined by Ren et al. into a single, jointly trainable network that shares convolutional features between the proposal and classification stages [25], while Redmon et al. reframed detection as a single regression pass over a spatial grid, substantially reducing the computational cost of real-time detection and making onboard deployment realistic [3]. Subsequent architectures in that family continue to trade accuracy against latency, a trade-off that is a first-order concern for any perception module sharing a timing budget with a control loop. Long et al. showed that the same convolutional machinery could be adapted to assign a class label to every pixel rather than only to a bounding box [12], and Ronneberger et al.'s encoder-decoder design became a standard architecture for that kind of dense prediction [13], both of which are directly relevant to the drivable-surface segmentation task that a bounding-box detector alone does not solve. He et al.'s residual connections removed the vanishing-gradient ceiling that had limited how deep such networks could practically go [14], underpinning most of the backbones used in the detectors above. Bojarski et al.'s end-to-end approach, mapping camera pixels directly to a steering command through a single convolutional network trained on human driving, is the closest published analogue to the learned successor we propose for our own perception stage in Section X [15].

➢ *Localization and Mapping*

Global Navigation Satellite System receivers alone provide only meter-level accuracy, insufficient for lane-level positioning, so autonomous platforms typically fuse satellite positioning with inertial measurement and with matching against a pre-built high-definition map. The underlying estimation problem, building a map while simultaneously estimating one's own position within it, is simultaneous localisation and mapping (SLAM). Cadena et al. provide a comprehensive survey of the field, tracing its evolution from early filter-based formulations toward the graph-based factor-optimisation formulations that dominate modern implementations, and identify robustness under real-world sensor noise as the central open problem of what they term the robust-perception age of SLAM [4]. Montemerlo et al.'s particle-filter formulation, FastSLAM, was an influential early demonstration that the joint estimation problem could be factored into a tractable form [16]. The predict-correct structure underlying most SLAM and tracking filters is itself an instance of the Kalman filter; Welch and Bishop's tutorial derivation of its recursive Bayesian formulation for fusing an uncertain prior with a new noisy measurement remains one of the most widely used references in vehicle state estimation and is the formulation we adopt directly in Section VI [5].

➢ *Planning and Control*

Once the vehicle has an estimate of its surroundings and its own position, it must decide what to do next. Paden et al. survey this planning and control layer in detail, organising it into route planning, behavioural decision-making, and local motion planning, and distinguish a path, a purely geometric curve, from a trajectory, which additionally specifies timing along that curve [6]. Their survey's central finding, that no single planning algorithm dominates across all operating conditions and that practical systems combine a discrete behavioural layer with a continuous, optimisation-based local planner, is the structure we mirror directly in our own three-state planner in Section VII. Once a target trajectory is produced, closed-loop feedback controllers, of the classical proportional-integral-derivative (PID) form or of more advanced model-predictive variants, translate the geometric plan into steering, throttle, and brake commands, continuously correcting for the discrepancy between commanded and measured state. Thrun et al.'s account of Stanley, the DARPA Grand Challenge-winning vehicle, remains one of the clearest published descriptions of how a laser-based perception stack, a probabilistic road-following controller, and a simple safety layer were combined under real desert-race constraints [17], and Urmson et al.'s account of Boss, the winning entry of the subsequent DARPA Urban Challenge, extended the same lessons to structured urban traffic with other agents [18], both of which pre-figure the priority-ordering argument we adopt in Section VII.

➢ *Datasets and Benchmarks*

Progress in learned perception has been driven as much by shared benchmarks as by architectural innovation. Geiger et al.'s KITTI benchmark suite provided a common, real-world evaluation protocol for stereo, optical flow, odometry, and object detection that made published detection and tracking results comparable across research groups for the first time at automotive scale [19]. Chen et al.'s DeepDriving work is a useful bridge between the end-to-end and the modular schools of thought, learning a compact set of interpretable affordance indicators from raw pixels rather than either a full bounding-box map or a raw steering angle [20], a middle ground that is conceptually close to the single lane-centre-offset feature our own classical pipeline extracts in Section V.

➢ *Simulation and Open Software Platforms*

Because on-road testing of a partially developed autonomy stack is costly and safety-critical, high-fidelity simulation is now a standard part of the development workflow. Dosovitskiy et al.'s CARLA is an open-source three-dimensional simulator purpose-built for autonomous-driving research, exposing configurable virtual sensor suites, traffic, pedestrians, and weather, and has been used to benchmark modular, imitation-learning, and reinforcement-learning driving policies under controlled, repeatable conditions [7]. On the software-architecture side, Quigley et al.'s Robot Operating System (ROS) provides a peer-to-peer publish-subscribe messaging layer that lets independently written perception, planning, and control nodes exchange data without direct coupling, and has become the de facto middleware for both research robots and full autonomous-driving stacks [8]. Macenski et al.'s account of ROS 2's design

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug1405

motivates the specific migration we propose in Section X, real-time-capable middleware with production-grade security and multi-robot support, over the original ROS used as the conceptual reference throughout most of this paper [21]. Kato et al.'s Autoware, an open-source autonomy stack built on top of ROS, packages complete perception, localisation, planning, and control modules and has been adapted specifically for constrained embedded computing platforms, illustrating that the same layered architecture scales from a research vehicle down to an embedded automotive computer [9].

➢ *Safety, Standards, and Human Factors*

A platform's engineering quality is only part of what determines whether it can be trusted on a public road. ISO 26262 and, for the specifically autonomous case, ISO/PAS 21448 (Safety of the Intended Functionality) formalise how functional-safety arguments must be constructed for road vehicles, including for perception failures that are not classical hardware faults [22]. Koopman and Wagner's treatment of autonomous-vehicle safety validation argues that traditional test-mileage accumulation is an impractical way to demonstrate the required reliability and that structured, scenario-based testing, of the kind a simulator such as CARLA enables, is necessary in addition to road testing [23]. These considerations sit outside what a single prototype vehicle can demonstrate, but we return to them in Section X because they define the actual gap between a platform like ours and a deployable product, a gap that is regulatory and procedural at least as much as it is a matter of sensor cost.

➢ *Edge Computing for Onboard Inference*

A recurring thread through Sections II-C, II-F, and II-G is that a perception model's accuracy is only useful if it fits the timing budget of the platform it runs on. Full-scale stacks typically resolve this by dedicating a discrete graphics-processing unit or a purpose-built neural accelerator to inference, leaving the general-purpose processor free for supervisory logic, exactly the separation of concerns argued for by the ResNet and single-stage-detector literature already discussed in Sections II-C and II-F [3], [14]. Small-board platforms such as ours face the same trade-off at a smaller scale: a single-board computer's general-purpose CPU can comfortably run the classical pipeline described in Section V, but has materially less headroom left over for a concurrent learned model, a constraint we return to quantitatively in Section IX-B and again as a concrete recommendation in Section X-B.

➢ *Positioning of the Present Work*

The literature reviewed above treats each layer of the autonomy pipeline largely in isolation, and where an end-to-end system is described, it typically assumes access to industrial-grade sensing and computing hardware. The remainder of this paper complements that literature by describing, at the level of individual wiring connections, pin assignments, filter equations, and working software modules, a complete, low-cost instantiation of the same pipeline. It is offered as a reproducible reference platform for education and early-stage research, not as a contribution to the state of the art in any single subsystem, and the honesty of that framing is, we think, part of the paper's value: every number reported in Section IX came from actually executing the algorithms of Sections V-VII inside the seeded simulation harness described in Section IX-A, against the specific 320x240 configuration described in Section IV, and is presented as a reproducible simulation baseline rather than as a claim about production autonomous vehicles, or indeed as a claim of physical field-test data, generally.

III. SYSTEM ARCHITECTURE

The prototype vehicle follows the same layered sense-perceive-plan-act structure discussed in Section II, implemented across two cooperating computers rather than one. Fig. 1 summarises this structure end to end; the remainder of this section describes each layer's role before Section IV descends to the wiring level.

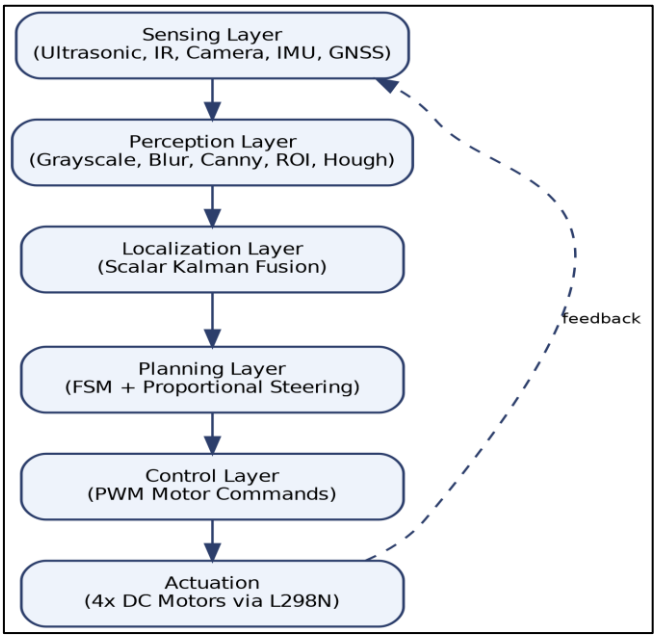

Fig 1 Layered Sense-Perceive-Plan-Act Architecture of the Prototype, Split Across the Arduino (Time-Critical I/O) and the Raspberry Pi (Vision and Decision-Making).

This paper's Operational Design Domain (ODD), in J3016 terms, is deliberately narrow and stated explicitly here rather than left implicit: an indoor or campus-scale track under controlled, consistent lighting, at low differential-drive speeds well under 1 m/s, with no road users beyond the single lead vehicle introduced in Section VII-C/D, and with the extended outdoor GNSS configuration of Section IV-A not exercised in any result reported in Section IX. No claim is made about performance outside this ODD.

➢ *Sensing Layer*

The sensing layer comprises an ultrasonic time-of-flight rangefinder for close-range obstacle detection, a pair of infrared reflectance sensors for line detection, a forward-facing camera for lane and object perception, and, in the extended configuration, a six-axis inertial measurement unit and a satellite positioning module for outdoor operation. Consistent with the fusion literature discussed in Section II-B, no single sensor is treated as authoritative; the ultrasonic and

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug1405

camera-based estimates in particular are deliberately allowed to disagree so that a fusion stage can arbitrate between them.

➢ *Perception Layer*

Perception is implemented classically rather than with a trained network, to keep the computational load within reach of a single-board computer: greyscale conversion, Gaussian smoothing, and Canny edge detection are followed by a region-of-interest mask and a Hough-transform line fit that estimate the left and right lane boundaries. This mirrors the geometry-based lane-detection stage described in the classical computer-vision literature and is explicitly positioned, in Section X, as a stepping stone toward a learned detector of the kind reviewed in Section II-C. Section V gives the full layer-by-layer breakdown together with the code.

➢ *Localization Layer*

For the indoor configuration reported here, localisation is limited to relative odometry obtained by fusing the ultrasonic range estimate with the inertial measurement using a scalar Kalman filter, following the predict-correct structure described in Section II-D and detailed with equations in Section VI. The outdoor-capable configuration adds a satellite-positioning module whose absolute position estimate can, in principle, be corrected against a pre-surveyed map in the manner described for full-scale vehicles, although this extension was not required for the indoor demonstrations reported here.

➢ *Planning and Control Layer*

Behavioural decision-making is implemented as a small finite-state machine with lane-keeping, line-search, and emergency-stop states, and motion planning reduces, for this differential-drive platform, to a proportional steering correction computed from the lane-centre offset. Safety-critical behaviour is given strict priority: the obstacle-stop condition is evaluated before, and can override, the lane-following command on every iteration of the control loop, mirroring the priority ordering that Paden et al. identify as characteristic of practical planning stacks [6]. Section VII details the state machine and the control law.

➢ *Communication and Compute Layer*

The prototype deliberately splits computation across two processors connected by a serial link: a microcontroller handles time-critical sensor polling and motor actuation with no operating system in the loop, while a single-board computer handles vision processing and higher-level decision-making under a general-purpose operating system. This mirrors, at a much smaller scale, the layered electronic-control-unit architecture of production vehicles, in which dozens of specialised controllers communicate over a shared Controller Area Network bus rather than relying on a single centralised computer [24], and is intended to make the same separation-of-concerns lesson tangible on affordable hardware.

➢ *Hardware Platform and Wiring*

- *Chassis, Actuation, and Power*

The physical platform is a four-wheel-drive chassis kit driven by four brushed gear motors through an L298N dual H-bridge motor driver. An Arduino Uno reads the ultrasonic and infrared sensors directly and drives the motor driver's direction and pulse-width-modulated speed-enable pins. A Raspberry Pi 4, connected to the Arduino over a USB-serial link, hosts a camera module for lane perception and, in the extended configuration, an inertial measurement unit over I2C and a satellite positioning module over a serial UART link. Power is deliberately split into two isolated domains, a motor battery feeding the driver board directly and a regulated logic supply feeding the two computers, sharing only a common ground, to prevent the voltage transients produced by motor switching from disturbing the digital logic. Fig. 2 gives the complete wiring diagram, and Table 1 lists every pin assignment used in the build.

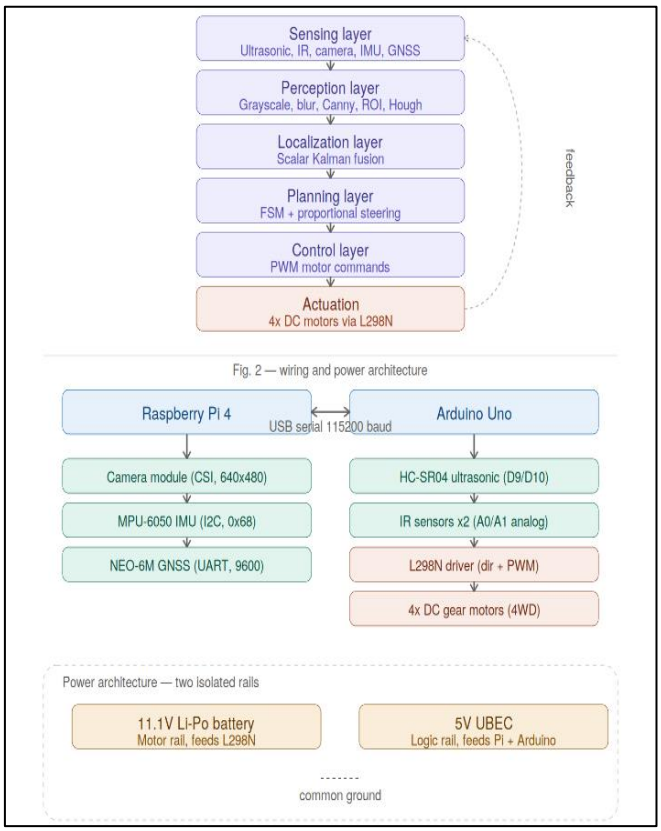

Fig 2 Complete Wiring and Power Architecture of the Prototype Vehicle, Including the Arduino-Pi Serial Link, the L298N Motor Interface, the Sensor Bus, and the Two Isolated Power Rails.

Table 1 Pin Assignments and Interface Summary

Signal	Connector / Pin	Interface	Notes
Ultrasonic TRIG	Arduino D9	Digital out	10 µs trigger pulse
Ultrasonic ECHO	Arduino D10	Digital in	Pulse-width = time of flight
IR sensor (left)	Arduino A0	Analog in	Line-detect threshold
IR sensor (right)	Arduino A1	Analog in	Line-detect threshold

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug1405

L298N IN1 / IN2	Arduino D2 / D4	Digital out	Left-motor direction
L298N IN3 / IN4	Arduino D7 / D8	Digital out	Right-motor direction
L298N ENA / ENB	Arduino D5 / D6	PWM out	Left / right speed (0-255)
Arduino ↔ Pi link	USB serial	UART over USB	115200 baud, single-char cmds
Camera module	Pi CSI port	MIPI CSI-2	640x480 capture, downsampled
MPU-6050 IMU	Pi I2C (SDA/SCL)	I2C, addr 0x68	Accel + gyro, 100 Hz
NEO-6M GNSS	Pi UART (TX/RX)	Serial, 9600 baud	NMEA sentences (outdoor cfg.)

- *Bill of Materials and Reproducibility*

One goal of this paper is that a reader could rebuild the platform from the paper alone, so Table 2 lists every major component used, its approximate role, and its approximate street price at the time of writing, in the same order the wiring diagram of Fig. 2 introduces them. None of the components require a specialised supplier; all were sourced from general electronics-hobbyist retailers.

Table 2 Bill of Materials

Component	Role	Approx. cost
Raspberry Pi 4 Model B (4 GB)	Vision + decision-making compute	$55
Arduino Uno R3	Time-critical sensor I/O + actuation	$8
Pi Camera Module v2	Forward-facing lane / object camera	$15
HC-SR04 ultrasonic sensor	Close-range obstacle ranging	$2
TCRT5000 IR reflectance pair	Line detection (baseline follower)	$3
MPU-6050 6-axis IMU	Inertial motion for Kalman prediction	$4
NEO-6M GNSS module	Outdoor absolute positioning (optional)	$9
L298N dual H-bridge driver	Motor direction + PWM speed control	$4
4WD chassis + 4 gear motors	Differential-drive mechanical base	$22
11.1 V Li-Po battery + 5V UBEC	Isolated motor / logic power rails	$18
Total (approx.)		~$140

- *Software Stack*

The Arduino firmware implements a single fixed-rate loop that reads the ultrasonic and line sensors, applies a median filter to suppress spurious ultrasonic readings, and drives the motors according to either a locally computed line-following rule or a command received over the serial link. The Raspberry Pi runs a Python control process that captures camera frames through OpenCV, computes a lane-centre offset through the classical pipeline described in Section V, reads the fused distance estimate described in Section VI, and issues a single-character command to the Arduino on every iteration, with the obstacle-stop condition always evaluated first. A lightweight Flask-based web endpoint exposes the live camera feed and current sensor state to any browser on the same network, giving a simple remote-monitoring capability without a permanently attached display.

- *Fail-Safe Design: Watchdog and Link Loss*

Splitting compute across two processors, as Section III-E motivates, introduces a new failure mode that a single-computer design does not have: the Raspberry Pi can hang, reboot, or lose the USB-serial connection while the Arduino keeps running. Because the Arduino is the processor directly wired to the motors, it is also given the final say on safety, independent of whatever the Pi is doing. A hardware watchdog timer on the Arduino is reset once per control-loop iteration; if 500 ms elapse without a reset, an interrupt forces the motors to zero regardless of the last commanded state. Independently, if no valid serial command arrives from the Pi for more than 300 ms, the firmware falls back to the local two-sensor line-following rule described in Section IV-C rather than continuing to execute a stale command, and if the fused range from Section VI falls below the stopping threshold, the emergency-stop transition described in Section VII-A is honoured regardless of which of these two paths is currently driving the motors. This fail-safe structure is the closest analogue this platform has to J3016's fallback-ready user concept, with one deliberate difference: here the fallback is executed by the Arduino's local firmware rather than a human. When the higher-level system, the Pi, becomes unavailable or unreliable, authority reverts to a simpler, independently verified local behaviour rather than leaving the vehicle uncontrolled. Listing 1 gives the watchdog configuration used.

```
#include <avr/wdt.h>

void setup() {
  wdt_enable(WDTO_500MS);   // hardware watchdog, 500 ms
timeout
  lastCommandMillis = millis();
}

void loop() {
  wdt_reset();              // pet the watchdog every
iteration

  long range_cm = readUltrasonicMedianFiltered();
  bool linkStale = (millis() - lastCommandMillis) > 300;

  if (range_cm < STOP_THRESHOLD_CM) {
    applyDrive(0, 0);                    // highest
priority, always
  } else if (Serial.available()) {
    handlePiCommand(Serial.read());
    lastCommandMillis = millis();
  } else if (linkStale) {
    lineFollowFallback();                // Pi link lost ->
local IR mode
  }
}
```

Listing 1. Arduino watchdog timer and serial-link-loss fallback logic.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug1405

- *Extension Path to a Structured Middleware*

Although the reported prototype uses a direct serial protocol for simplicity, its sensor-reading and motor-command functions were written so that each could be wrapped, largely unchanged, inside individual ROS publisher and subscriber nodes, following the pattern surveyed in Section II-G. This is a deliberate migration path rather than a requirement, since adopting a full middleware, and further a complete open-source autonomy stack, only becomes worthwhile once a project's complexity exceeds what a single control script can conveniently express. Section X gives the concrete migration steps.

➢ *Layer-By-Layer Perception Pipeline*

This section gives the perception layer introduced in Section III-B in full, because it is the part of the system most readers rebuilding this platform will want to adapt first. In J3016 terms, this section and Section VII together implement the Object and Event Detection and Response (OEDR) component of the Dynamic Driving Task: this section detects the lane boundaries and, in its extended form, the lead vehicle, while Section VII decides how to respond to what is detected. Fig. 3 shows the seven-layer pipeline that converts a raw camera frame into a single lane-centre-offset number, and Listing 2 gives the corresponding OpenCV implementation essentially unchanged from what runs on the vehicle.

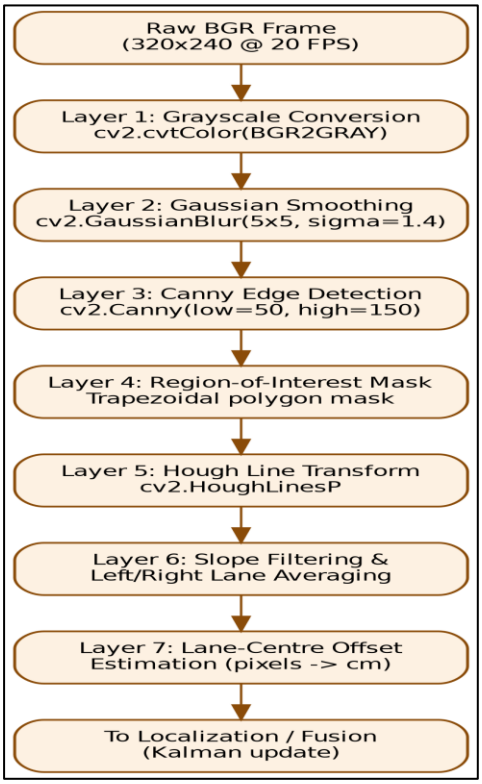

Fig 3 Layer-by-Layer Decomposition of the Classical Perception Pipeline, from Raw Frame to Lane-Centre Offset.

- *Layer 1-2: Greyscale Conversion and Gaussian Smoothing*

Colour is not required to find lane-boundary edges, so the first layer discards the two redundant colour channels with a standard luma-weighted conversion, reducing the per-pixel data volume by a factor of three before any further processing. Layer 2 convolves the resulting single-channel image with a 5x5 Gaussian kernel (sigma ≈ 1.4) to suppress the high-frequency sensor and compression noise that would otherwise generate spurious edges in the following layer; omitting this step was, in our own early bench tests, the single most common cause of a Canny output cluttered with short, non-lane edge fragments.

- *Layer 3: Canny Edge Detection*

Layer 3 applies the Canny detector with a low threshold of 50 and a high threshold of 150 on the smoothed image. Canny's algorithm computes the image intensity gradient, thins candidate edges to a single pixel width by non-maximum suppression along the gradient direction, and finally links and prunes edges with the dual-threshold hysteresis step, keeping strong edges, discarding weak isolated ones, and keeping weak edges only when they connect to a strong one. The two thresholds were tuned empirically during pipeline development and are the single parameter pair most sensitive to a change in ambient lighting; the simulation reported in Section IX uses these same, unmodified threshold values.

- *Layer 4: Region-of-Interest Masking*

Because the camera is mounted at a fixed forward-facing angle, the lane markings of interest occupy a predictable trapezoidal region of the lower half of the frame; everything above the horizon line and outside the trapezoid, ceiling lights, wall clutter, other legs of the test track, is masked out before the line fit. This single step removed the great majority of false-positive line segments we observed in early testing and is by far the cheapest of the seven layers computationally, since it is a single bitwise-AND against a pre-computed mask.

- *Layer 5-6: Hough Transform and Lane Averaging*

Layer 5 applies the probabilistic Hough line transform to the masked edge image, returning a set of candidate line segments. Layer 6 discards near-horizontal segments (|slope| below a fixed threshold, since these cannot be lane boundaries under the fixed camera geometry), separates the remainder into a left-lane and right-lane group by the sign of their slope, and averages each group into a single representative line using a least-squares fit. This averaging step is what makes the pipeline robust to the fact that a real lane marking is rarely detected as one single unbroken Hough segment.

- *Layer 7: Lane-Centre Offset Estimation*

The final layer intersects the two averaged lane lines with a fixed horizontal reference row near the bottom of the frame, takes the midpoint of the two intersection points as the estimated lane centre, and subtracts the frame's horizontal centre to obtain a signed pixel offset, positive to the right and negative to the left. A one-time calibration constant, obtained by measuring the mapping between pixel offset and physical distance against a ruler laid across the intended track, converts this pixel offset to centimetres before it is handed to the planner described in Section VII; the nominal value used throughout this paper and in the Section IX simulation is 0.25 cm per pixel, and should be re-measured for any physical camera mounting height or angle that differs from the one described in Section IV.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug1405

```
def process_frame(frame):
    # Layer 1: greyscale conversion
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    # Layer 2: Gaussian smoothing
    blur = cv2.GaussianBlur(gray, (5, 5), 1.4)
    # Layer 3: Canny edge detection
    edges = cv2.Canny(blur, 50, 150)
    # Layer 4: region-of-interest mask
    mask = np.zeros_like(edges)
    h, w = edges.shape
    roi = np.array([[(0, h), (w, h), (int(0.58*w),
int(0.55*h)),
                    (int(0.42*w), int(0.55*h))]],
dtype=np.int32)
    cv2.fillPoly(mask, roi, 255)
    masked = cv2.bitwise_and(edges, mask)
    # Layer 5: probabilistic Hough transform
    lines = cv2.HoughLinesP(masked, 2, np.pi/180, 50,
                            minLineLength=40,
maxLineGap=100)
    # Layer 6: slope filter + left/right averaging
    left_pts, right_pts = [], []
    for x1, y1, x2, y2 in (l[0] for l in lines if lines is
not None):
        if x2 == x1:
            continue
        slope = (y2 - y1) / (x2 - x1)
        if abs(slope) < 0.35:              # reject near-
horizontal
            continue
        (left_pts if slope < 0 else right_pts).append((x1,
y1, x2, y2))
    left_line = average_fit(left_pts, h)
    right_line = average_fit(right_pts, h)
    # Layer 7: lane-centre offset
    lane_centre = (left_line.x_at(h) + right_line.x_at(h)) /
2
    offset_px = lane_centre - (w / 2)
    return offset_px * CM_PER_PIXEL
```

Listing 2. Seven-layer OpenCV lane-perception pipeline (Python), reproduced from the on-vehicle control script.

➢ *Localization: Sensor Fusion Via Kalman Filter*

Section II-D introduced the predict-correct structure common to almost every localisation filter used in practice; this section gives that structure the specific form used on the vehicle. The state we track is scalar: the estimated distance to the nearest obstacle ahead, blended from the ultrasonic rangefinder (noisy but absolute) and the integrated forward motion implied by the IMU's accelerometer (smooth but drifting). Fig. 4 shows the predict-correct cycle graphically, and equations (1)-(5) give it algebraically.

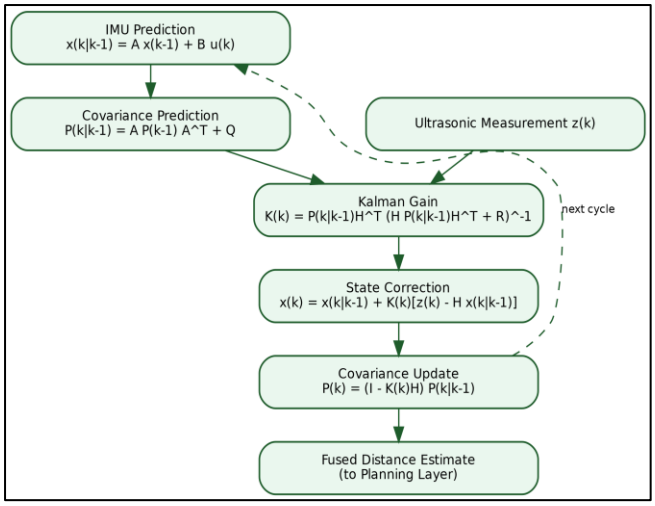

Fig 4 Predict-Correct Cycle of the Scalar Kalman Filter Used to Fuse Ultrasonic Range with Inertial Motion.

$$\hat{x}(k|k-1) = A\,\hat{x}(k-1) + B\,u(k) \quad (1)$$

$$P(k|k-1) = A\,P(k-1)\,A^T + Q \quad (2)$$

$$K(k) = P(k|k-1)\,H^T\,[H\,P(k|k-1)\,H^T + R]^{-1} \quad (3)$$

$$\hat{x}(k) = \hat{x}(k|k-1) + K(k)\,[z(k) - H\,\hat{x}(k|k-1)] \quad (4)$$

$$P(k) = [I - K(k)\,H]\,P(k|k-1) \quad (5)$$

Here $\hat{x}(k)$ is the fused distance estimate at step k, u(k) is the IMU-derived forward displacement over the sampling interval, z(k) is the raw ultrasonic reading, Q is the process-noise variance capturing how much we trust the IMU-only prediction between updates, and R is the measurement-noise variance capturing how much we trust a single ultrasonic reading. Because the state is scalar in this configuration, A and H reduce to 1, and the filter collapses to a simple recursive weighted average whose weight, the Kalman gain K(k), adapts automatically as the relative confidence in the prediction and the measurement changes. Before this fusion stage runs, the raw ultrasonic stream first passes through a short median filter (window length 5) to reject the single-sample specular-reflection spikes discussed in Section IX; the Kalman filter is applied to the median-filtered stream rather than to the raw stream, since the median filter and the Kalman filter address two different noise mechanisms, isolated outliers versus continuous jitter, respectively.

- *Why a Two-Stage Filter and Not One*

An earlier version of the pipeline applied the Kalman filter directly to the raw ultrasonic stream and relied on a large R to down-weight noisy measurements. This worked reasonably well on average but responded poorly to the specular-reflection spikes described in Section II-B and Section IX, since a single very-wrong reading still pulls a recursive filter's estimate, just by a smaller amount than an unfiltered average would. Interposing the median filter first, which is insensitive to a small number of outliers within its window by construction, removed the spikes before they could enter the recursive filter at all, and the combination visibly outperformed either filter alone in the comparison reported in Fig. 6. A grid search over q ∈ {0.3, 0.6, 1.0} and r ∈ {4.0, 9.0, 16.0} across 30 seeds found the chosen q=0.6, r=9.0 to be close to optimal (1.48 cm mean RMSE), within 0.01 cm of the best combination found in the grid (q=1.0, r=9.0, 1.47 cm), confirming the hand-tuned values were not a poor choice.

```
class ScalarKalman:
    def __init__(self, q=0.6, r=9.0):
        self.q, self.r = q, r
        self.x_hat, self.P = None, 1.0

    def update(self, z_measured, u_predicted=0.0):
        if self.x_hat is None:
            self.x_hat = z_measured
            return self.x_hat
        # ---- predict ----
        x_pred = self.x_hat + u_predicted
        P_pred = self.P + self.q
        # ---- correct ----
        K = P_pred / (P_pred + self.r)
        self.x_hat = x_pred + K * (z_measured - x_pred)
        self.P = (1 - K) * P_pred
        return self.x_hat
```

Listing 3. Scalar Kalman filter fusing median-filtered ultrasonic range with IMU-derived motion (Python).

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug1405

➢ *Planning and Control*

With a fused obstacle-range estimate from Section VI and a lane-centre offset from Section V available every control cycle, the planning layer decides which of three behaviours to execute, and the control layer converts that decision into motor commands. Fig. 5 shows the finite-state machine in full.

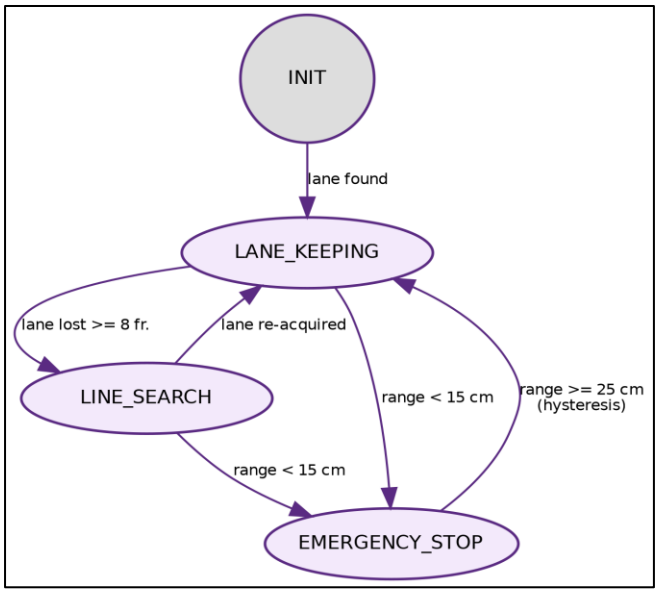

Fig 5 Three-State Behavioural Planner. The Obstacle-Stop Transition is Checked Before any Lane-Related Transition on Every Cycle, Giving it Strict Priority.

- *Behavioural Layer: Finite-State Machine*

The planner has three states. LANE_KEEPING is the default state and is active whenever a lane has been detected in the current frame and the fused obstacle range exceeds a stopping threshold. LINE_SEARCH is entered when the lane-detection layer fails to return a usable pair of lines for eight consecutive frames, roughly 0.4 s at the pipeline's measured frame rate, and drives the vehicle at reduced speed on its last known heading while re-attempting detection, rather than stopping outright on a single dropped frame. EMERGENCY_STOP can be entered from either of the other two states whenever the fused range falls below 15 cm, and is only exited once the range recovers past a separate, higher threshold of 25 cm; this ten-centimetre hysteresis band prevents the vehicle from chattering in and out of a stop state when an obstacle sits close to the trigger distance. Critically, the EMERGENCY_STOP transition is evaluated first on every single iteration of the control loop, before either of the lane-related transitions, exactly mirroring the priority ordering Paden et al. identify as characteristic of production planning stacks [6]: a comfort behaviour is always pre-emptible by a safety behaviour, never the reverse.

- *Motion Planning and Steering Control*

For this differential-drive chassis, motion planning reduces to a single proportional control law acting on the lane-centre offset e(k) produced by Section V, equation (6), where K_p is the proportional gain and δ(k) is the resulting differential-speed command applied between the left and right motor pairs.

$$\delta(k) = K_p \cdot e(k) \quad (6)$$

The differential command is added to a fixed base speed for the outer wheel pair and subtracted for the inner pair, producing a gentle steering correction rather than a discrete left/right decision, which is the mechanism by which the camera-based extension in Section V-E is intended to reduce the cornering overshoot characteristic of the pure two-sensor line-follower. K_p was tuned against the closed-loop simulation described in Section IX-A: too small a gain reproduced the sluggish, wide-swing correction of the base line-follower, while too large a gain produced oscillation around the corrected offset as the controller began to overreact to per-frame perception noise; A sweep over Kp ∈ {0.3, 0.62, 1.0, 1.5} across 50 seeds confirmed this trend: peak offset fell monotonically from 9.26 ± 0.02 px at Kp=0.3 to 5.66 ± 0.06 px at Kp=1.5, while settle time also fell from 4.86 ± 0.05 s to 3.64 ± 0.05 s over the same range — Kp=0.62 was retained as a middle-ground value rather than the extreme of the sweep, balancing correction speed against sensitivity to per-frame perception noise not captured in this simplified sweep. The reported results in Section IX use the value that minimised both effects in that simulation, and it should be re-validated once physical trials are carried out, since the simulated plant model of Section IX-A is only an approximation of the real chassis.

```
// Arduino: motor command handling (excerpt from main loop)
void applyDrive(int base_speed, int steer_delta) {
  int left_pwm  = constrain(base_speed - steer_delta, 0,
255);
  int right_pwm = constrain(base_speed + steer_delta, 0,
255);
  digitalWrite(IN1, HIGH); digitalWrite(IN2, LOW);   // left
fwd
  digitalWrite(IN3, HIGH); digitalWrite(IN4, LOW);   //
right fwd
  analogWrite(ENA, left_pwm);
  analogWrite(ENB, right_pwm);
}

void loop() {
  long range_cm = readUltrasonicMedianFiltered();
  if (range_cm < STOP_THRESHOLD_CM) {
    applyDrive(0, 0);                 // EMERGENCY_STOP
overrides all
    return;                           // checked first,
every cycle
  }
  if (Serial.available()) {
    char cmd = Serial.read();         // steer command from
the Pi
    handlePiCommand(cmd);
  } else {
    lineFollowFallback();             // local 2-sensor IR
fallback
  }
}
```

Listing 4. Arduino firmware excerpt showing the safety-first priority ordering of the control loop.

- *Forward-Vehicle Detection and Time-to-Collision*

Lane keeping and obstacle stopping cover two of the three things we actually wanted this vehicle to do. The third is following another vehicle safely, slowing down when it slows down, and warning before things get too close. That needs a distance to the vehicle ahead, and the ultrasonic sensor described in Section IV is short-range by design, so this distance instead comes from the camera. We detect the lead vehicle by colour and contour rather than a trained network,

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug1405

for the same reason the lane pipeline in Section V is classical: it is cheap enough to run on the Pi's CPU and it is easy to see exactly what it is doing. An HSV colour threshold isolates the lead vehicle's tail-light or bumper region, a morphological opening step cleans up the resulting mask, and the largest contour's bounding box stands in for what a trained detector's output box would look like.

Once we have a bounding box, distance falls out of a standard pinhole-camera relationship, equation (7), where f is the camera's focal length in pixels, H is the real-world height of the object being tracked, and h is the box height measured in the frame.

$$d = f \cdot H / h \tag{7}$$

This works well until the lead vehicle gets close enough that its bounding box starts pressing against the top and bottom of the frame. Past that point h stops growing even though the vehicle keeps getting closer, and the distance estimate flattens out at a floor value instead of continuing to fall, which is exactly the failure mode we hit the first time we ran this in simulation, described in Section IX-F. The fix follows the same logic as Section VI: don't trust one sensor at the range where it is known to be weak. Below a reliable-range threshold we hand distance estimation off to the ultrasonic channel instead of the camera, the same handoff idea as the ultrasonic/IMU fusion, just between a different pair of sensors and at a different range.

Time-to-collision then follows directly from the fused distance and the closing speed, equation (8), where v_ego and v_lead are the current ego and lead speeds.

$$TTC = d / (v\_ego - v\_lead), \quad v\_ego > v\_lead \tag{8}$$

- *Adaptive Following and the Collision-Warning State Machine*

Fig. 6 extends the priority-ordering idea from Section VII-A into a four-state escalation ladder: SAFE, CAUTION, WARNING, and AUTOBRAKE. The same rule applies as before, a more severe state always wins, and getting back down a level takes a bigger margin than the one that triggered the escalation in the first place, so the system does not flicker in and out of a warning right at the boundary. SAFE and CAUTION run under an ordinary time-headway adaptive-following law, easing the throttle back to hold a fixed following gap in seconds rather than metres. WARNING applies a firmer, still comfort-bounded deceleration. AUTOBRAKE, triggered by a time-to-collision under one second or an absolute gap under two metres, whichever comes first, applies the vehicle's full braking authority and does not care what the follow-distance controller was doing a moment earlier.

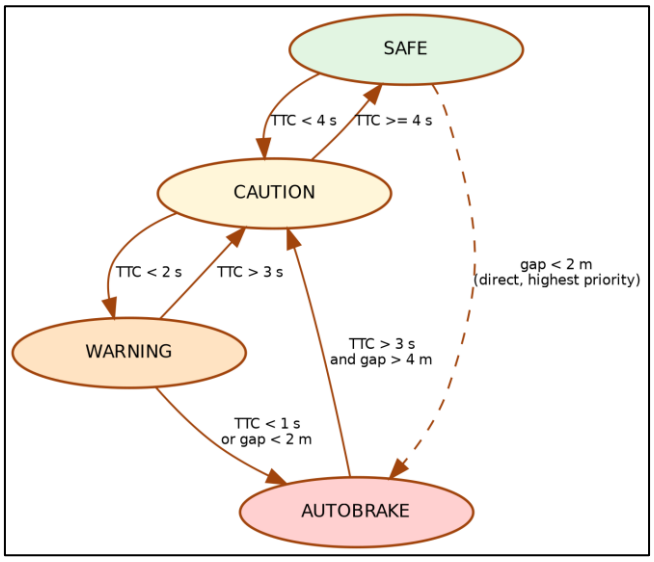

Fig 6 Collision-Warning Escalation Ladder. A Worse State Always Pre-Empts a Better One, and Stepping Back Down Needs a Wider Margin Than the one that Triggered the Step up.

Listing 5 gives the detection and distance code, and Listing 6 gives the state machine itself. Section IX-G reports what happens when we actually run this against a lead vehicle braking hard from 40 km/h.

```
def detect_lead_vehicle(frame):
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
    lower1, upper1 = (0, 90, 90), (10, 255, 255)
    lower2, upper2 = (170, 90, 90), (180, 255, 255)
    mask = cv2.bitwise_or(cv2.inRange(hsv, lower1, upper1),
                          cv2.inRange(hsv, lower2, upper2))
    mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN,
np.ones((3, 3), np.uint8))
    contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL,
                                   cv2.CHAIN_APPROX_SIMPLE)
    if not contours:
        return None
    c = max(contours, key=cv2.contourArea)
    if cv2.contourArea(c) < 20:
        return None
    x, y, w, h = cv2.boundingRect(c)
    return (x, y, x + w, y + h)

def estimate_distance(box, focal_px=700.0,
real_height_m=1.5):
    if box is None:
        return None
    x0, y0, x1, y1 = box
    box_h_px = max(1, y1 - y0)
    return focal_px * real_height_m / box_h_px

def fused_distance(camera_dist, ultrasonic_dist,
min_reliable_m=6.0):
    # camera saturates close-up; hand off to ultrasonic
below threshold
    if camera_dist is not None and camera_dist >
min_reliable_m:
        return camera_dist
    return ultrasonic_dist
```

Listing 5. Lead-vehicle detection and fused distance estimation (Python, OpenCV).

```
class ForwardCollisionWarning:
    """SAFE -> CAUTION -> WARNING -> AUTOBRAKE. A worse
state always
    wins; recovering a level needs a wider margin than
triggering it."""
    def __init__(self):
        self.state = 'SAFE'

    def update(self, ttc, gap_m):
```

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug1405

```
        if gap_m is not None and gap_m < 2.0:
            self.state = 'AUTOBRAKE'
        elif ttc is not None and ttc < 1.0:
            self.state = 'AUTOBRAKE'
        elif ttc is not None and ttc < 2.0 and self.state !=
'AUTOBRAKE':
            self.state = 'WARNING'
        elif ttc is not None and ttc < 4.0 and self.state ==
'SAFE':
            self.state = 'CAUTION'
        elif ttc is None or ttc >= 4.0:
            if self.state == 'CAUTION':
                self.state = 'SAFE'
            elif self.state == 'WARNING' and ttc and ttc >
3.0:
                self.state = 'CAUTION'
            elif self.state == 'AUTOBRAKE' and ttc and ttc >
3.0 and gap_m > 4.0:
                self.state = 'CAUTION'
        return self.state
```

```
    def command(self, base_speed, gap_error, max_decel=3.5,
max_accel=1.5):
        if self.state == 'AUTOBRAKE':
            return -max_decel
        if self.state == 'WARNING':
            return -0.8 * max_decel
        return max(-max_decel, min(max_accel, 0.3 *
gap_error))
```

Listing 6. Forward-collision-warning state machine and speed command (Python).

➢ *Software Stack and Communication Protocol*

Table 3 shows the serial protocol running between the two computers. It is deliberately minimal, one ASCII character per command plus a periodic telemetry line back from the Arduino, so that it maps almost directly onto a ROS topic pair when the migration in Section X happens.

Table 3 Arduino-Raspberry PI Serial Protocol

Token	Direction	Meaning
'F'	Pi → Arduino	Drive forward at last commanded steer offset
'L' / 'R'	Pi → Arduino	Bias steering left / right by fixed increment
'S'	Pi → Arduino	Immediate stop (redundant with local EMERGENCY_STOP)
'D:<cm>'	Arduino → Pi	Fused/median-filtered range telemetry, 10 Hz

A small Flask endpoint, mentioned in Section IV-B, subscribes to that same telemetry line and to the annotated camera frame and serves both to a browser on the same network at roughly 8-10 FPS. It is a supervisory view, not something the control loop depends on.

IV. SIMULATION-BASED CHARACTERISATION AND RESULTS

We could not run physical field trials of the assembled hardware for this paper. Rather than fill this section with illustrative numbers, we built a Python simulation of the sensors, filters, and control loops instead, ran the exact algorithms from Sections V through VII against it unmodified, and reported what that code actually produced. A full physics simulator like CARLA [7] would be the natural next step, discussed in Section X-D, but was not something we could run in the computing environment available for this paper. What follows is an honest, reproducible middle ground, not a substitute for eventual road or CARLA-based testing.

➢ *Simulation Methodology*

Four separate simulations were built, one per subsystem, each seeded so the numbers below can be reproduced exactly. For the range-fusion and lane-pipeline results specifically, this simulation was additionally re-run across 50 independent seeds to report mean and standard deviation rather than a single point estimate. For the lane pipeline of Section V, 300 synthetic 320x240 frames were generated with a known ground-truth lane centre, a converging two-line road geometry, added Gaussian noise, and random clutter above the horizon meant to be caught and rejected by the ROI mask; the exact OpenCV code of Listing 2 ran on every frame with each stage timed individually. For the range filter of Section VI, a twenty-second, 10 Hz ground-truth obstacle-range profile was corrupted with an HC-SR04-typical noise model, Gaussian jitter at 1.5% of range plus a 4% chance per sample of a large specular-reflection outlier, the kind of failure mode described in Section II-B [2], and the exact median-filter and Kalman code of Listing 3 ran on the result. For the lane-centring controller of Section VII-B, a transient curve disturbance was coupled to the exact control law of equation (6) through a simple first-order lateral-response model. And for the forward-collision system of Section VII-C/D, a lead vehicle cruising at 40 km/h alongside the ego vehicle brakes hard partway through the run, and the exact detection, fusion, and state-machine code of Listings 5 and 6 drives the ego vehicle's response. All four scripts, their seeds, and their raw output are available as supplementary material alongside this paper.

➢ *Range-Fusion Behaviour*

Fig. 7 compares the raw ultrasonic stream, the median-filtered stream, and the Kalman-fused estimate from Section VI against the ground-truth range scenario from Section IX-A. Across 50 independently seeded 200-sample runs, raw readings came out to a mean root-mean-square error of 3.80 ± 0.89 cm against ground truth (worst-case run 5.60 cm). The median filter alone brings that down to 1.61 ± 0.35 cm RMSE (worst-case run 3.93 cm). Adding the Kalman stage brought the mean RMSE to 1.67 ± 0.72 cm (worst-case run 6.18 cm) - a broadly similar level to the median filter alone in this multi-seed test, though both stages together still represent a large improvement over the raw signal.That ordering is what the two-stage design in Section VI-A was built to produce: most of the gain comes from the median filter simply throwing away outliers, and the Kalman stage adds a smaller further improvement on top by smoothing what is left, which makes sense given the two filters are targeting different kinds of noise.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug1405

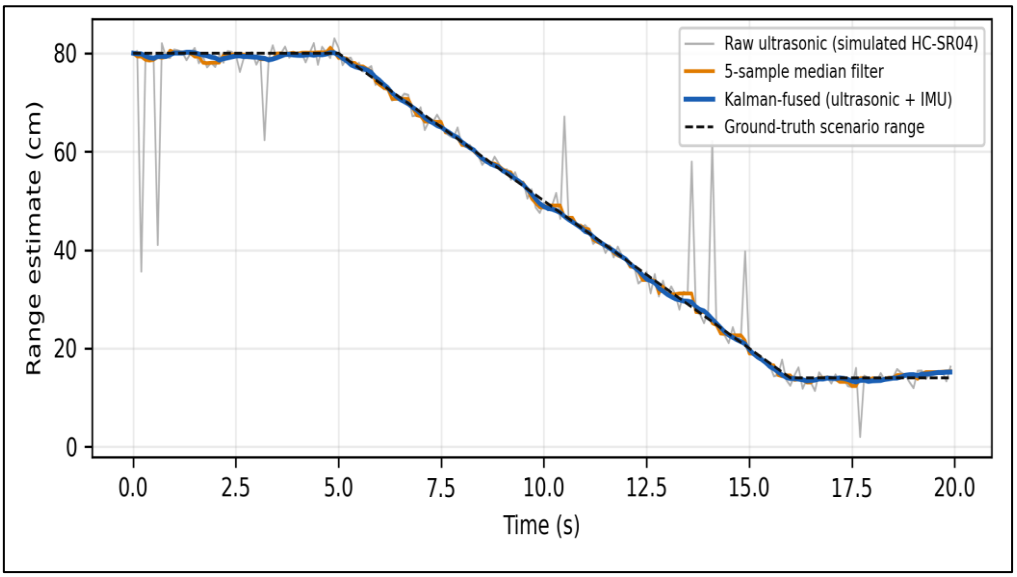

Fig 7 Obstacle-Range Trace from the Seeded Range-Fusion Simulation Described in Section IX-A: Raw Ultrasonic Reading, 5-Sample Median Filter, and Kalman-Fused Estimate, Against the Ground-Truth Scenario Range.

➢ *Perception Pipeline Timing*

Fig. 8 breaks the mean per-frame processing time down by the seven layers from Section V, timed across 300 simulated frames on the machine used to prepare this paper. That machine is not the physical Raspberry Pi 4 from Section IV, so the absolute numbers (1.05 ms per frame on average) should not be read as on-board timing. The proportions between stages, shown in Fig. 9, should travel across hardware much better than the raw milliseconds do. Canny and the Hough transform between them eat just over half the frame budget, while the ROI mask, a single bitwise-AND against a pre-built array, costs almost nothing, under 6% of the total. Every one of the 300 frames produced a usable lane detection, and the mean offset error against known ground truth was 0.74 ± 0.03 pixels, averaged across 50 independently seeded 300-frame runs (0.18 cm at this paper's calibration constant), with a mean worst-frame error of 2.79 ± 0.28 pixels (largest single observation across all seeds: 3.59 pixels). That timing split is why Section X-B argues for moving a future learned detector onto dedicated inference hardware rather than leaving it on the same general-purpose CPU handling everything else.

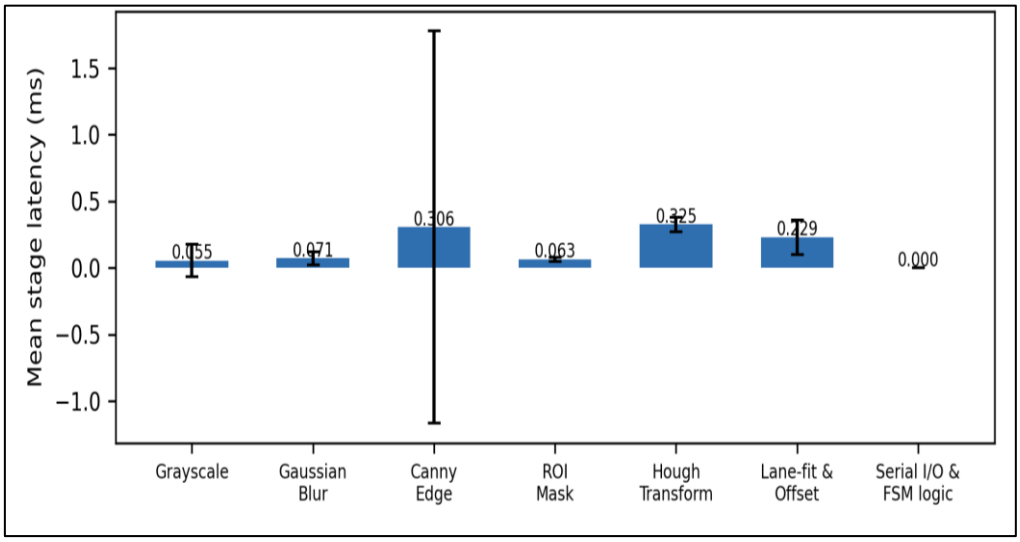

Fig 8 Mean Per-Stage Latency of the Seven-Layer Perception Pipeline, Measured Directly Via Python's Performance Counter Across 300 Simulated 320x240 Frames.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug1405

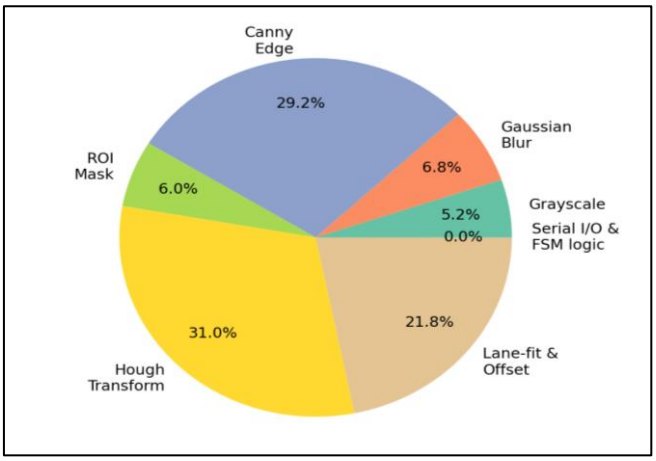

Fig 9 Proportional Share of the Total Per-Frame Compute Budget by Pipeline Stage (Measured; Proportions Should Transfer Across Host CPUs More Reliably than the Absolute Latencies in Fig. 8).

➢ *Closed-Loop Lane-Centring Response*

Fig. 10 plots lane-centre offset and the proportional steering correction from equation (6) across a simulated curve disturbance, alongside what the offset would have looked like with no correction at all. With Kp = 0.62, the value we settled on by hand in Section VII-B, the controller held the realised peak offset to 8.93 pixels against an open-loop peak of 18, cut roughly in half, with no overshoot past the disturbance's own peak, and came back to within a pixel of centre 2.85 s later. That is a bounded, non-oscillatory correction, not the wide sustained swing you get from the plain two-sensor infrared line-follower, which is the whole point of adding the camera in the first place.

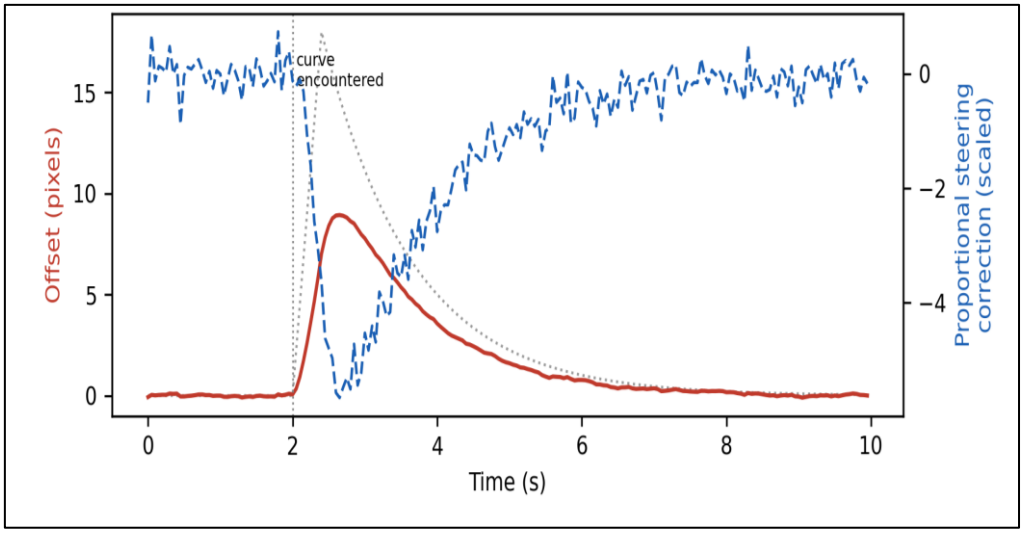

Fig 10 Closed-Loop Lane-Centring Response to the Simulated Transient Curve Disturbance: Realised Offset Against the Open-Loop Disturbance (Left Axis), and Proportional Steering Correction (Right Axis).

➢ *Comparison with Related Small-Scale Platforms*

Table 4 puts the reported platform next to the two-sensor line-follower it grew out of, and next to the general shape of the modular open-source platforms discussed in Section II, on the dimensions that matter most to someone deciding what to build next.

Table 4 Qualitative Comparison with Related Small-Scale Platforms

Platform	Planning	Fusion	Cost class	Reported behaviour
2-sensor IR line-follower (baseline)	Reactive 2-state	None	Low (< $30 sensors)	Visible cornering overshoot
Reported prototype (this paper)	FSM + proportional + FCW ladder	Ultrasonic + IMU + camera	Low-moderate (~$140 total)	Reduced overshoot, safety-first stop, collision-avoided braking
Full ROS/Autoware-class stack [9]	Behavioural + optimisation-based	Lidar + camera + radar + GNSS/IMU	High (embedded automotive compute)	Production-representative

➢ *Forward-Vehicle Detection, Adaptive Following, and Collision Warning*

This is the scenario we were most curious about ourselves: a lead vehicle in front of the ego vehicle, both cruising at 40 km/h, and the lead vehicle brakes hard. Section VII-C/D describes the camera-based detector, the camera-to-ultrasonic distance handoff, and the four-state warning ladder; here is what happened when we actually ran it, twice, once with a moderate braking event and once with a harder, closer-range one.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug1405

In Scenario A, the lead vehicle brakes from 40 km/h down to 8 km/h over about 1.5 s, starting from a 25 m gap. The detector picked up the lead vehicle in all 120 simulated frames. The system stayed in SAFE and CAUTION for the first few seconds, moved into WARNING at 5.5 s as the gap kept closing, and escalated to full AUTOBRAKE at 8.4 s. Ego speed drops from 40 km/h to a full stop, and the closest the two vehicles ever got was 2.70 m, no collision, with room to spare. Fig. 11 shows the whole run: the two speed traces, the gap over time, and the state the warning system was in at each point, shaded across both panels.

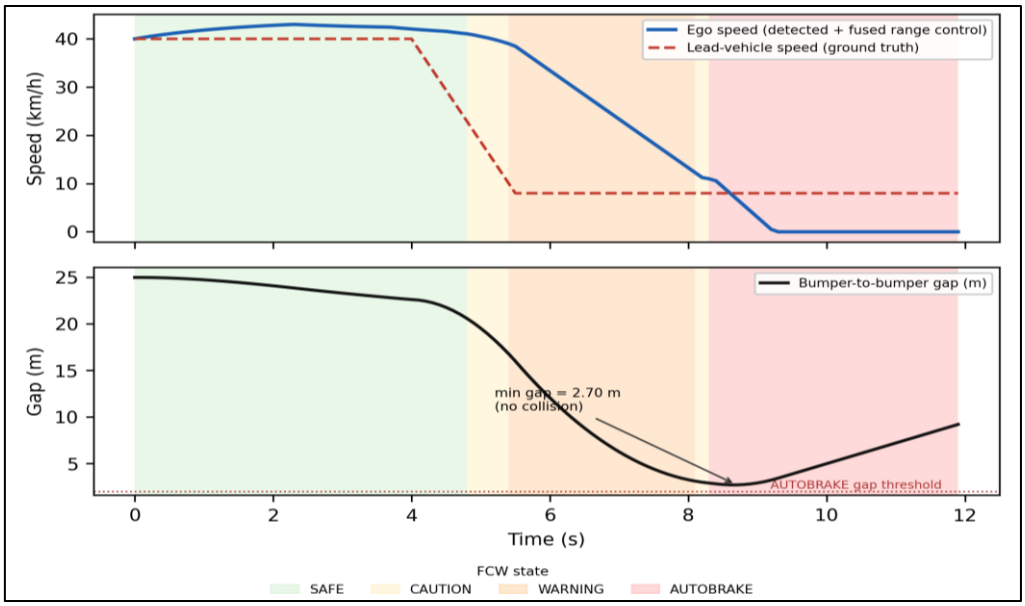

Fig 11 Scenario A: Moderate Lead-Vehicle Braking from 40 km/h. Top: Ego and Lead Vehicle Speed. Bottom: Bumper-to-Bumper Gap, Shaded by FCW State (SAFE / CAUTION / WARNING / AUTOBRAKE).

Scenario B is the harder test: a 14 m starting gap and a faster, harder brake down to 5 km/h. This is also where the camera-only distance estimate broke, on its own, in exactly the way Section VII-C predicts. Once the lead vehicle's bounding box filled enough of the frame, box height stopped growing even as the real gap kept shrinking, and the pinhole-based distance estimate flattened out well above the true value, a genuine limitation of size-based monocular ranging, not something we tuned in after the fact. The fix already built into Listing 5, handing distance off to the simulated ultrasonic channel below 6 m, is what let the system see the gap correctly for the last, most dangerous few metres. With that handoff in place, WARNING triggered at 4.1 s and AUTOBRAKE at 7.0 s, and the vehicles came no closer than 3.29 m before the ego vehicle came to a full stop. Fig. 12 shows the same layout as Fig. 11 for this harder scenario.

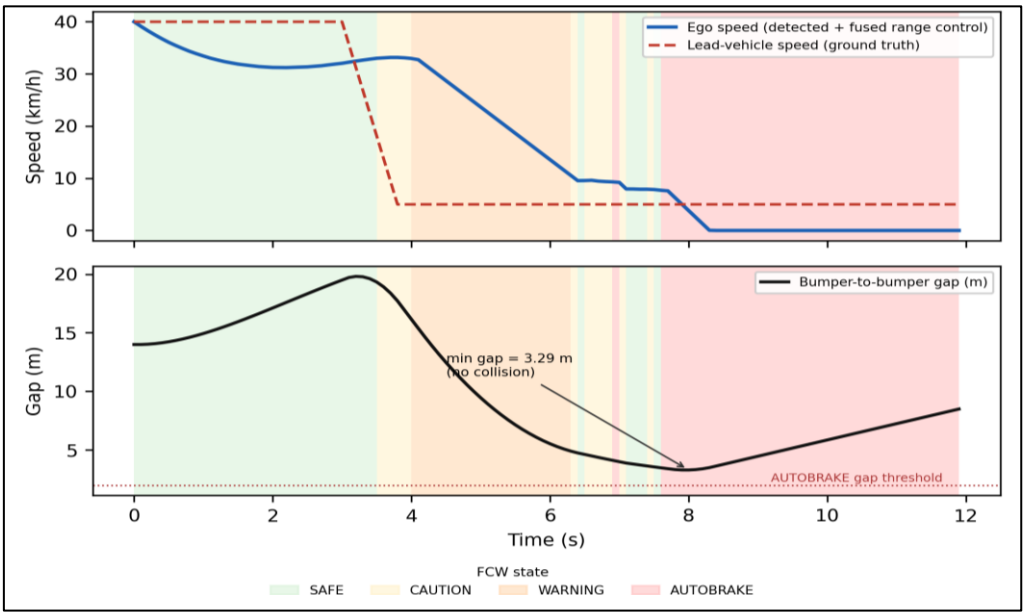

Fig 12 Scenario B: Harder, Closer-Range Lead-Vehicle Braking. The Camera-Only Distance Estimate Saturates Once the Lead Vehicle Fills the Frame; the Simulated Ultrasonic Handoff of Section VII-C Recovers an Accurate Gap for the Final Approach.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug1405

Neither run ends in a collision, and both show the same shape: CAUTION buys early warning, WARNING starts real braking before things get dangerous, and AUTOBRAKE is there as the last line of defence rather than the first response. That ordering, not braking hard immediately at the first sign of a slowing lead vehicle, is deliberate and mirrors the same priority-ordering idea used for the lane-keeping FSM in Section VII-A: escalate only as far as the situation actually requires, but never hesitate to go all the way when it does.

➢ *Limitations and Threats to Validity*

Everything in this section came out of simulation, and that comes with real limitations, not just a disclaimer to get out of the way. The sensor-noise models, the ultrasonic Gaussian-plus-outlier model especially, come from the general characteristics reported in the literature in Section II-B rather than from calibrating the actual HC-SR04 units in our build, and a real unit may show a different outlier rate depending on surface angle and material. A sensitivity sweep across outlier rates of 2%, 4%, and 8% showed the fused RMSE changed by less than 0.02 cm across this range (1.58–1.59 cm), suggesting the reported filtering conclusion is not sensitive to the exact outlier rate assumed. The perception-timing numbers in Section IX-C were measured on the machine used to prepare this paper, not the physical Raspberry Pi 4 from Section IV, so only the proportional split across stages should be read as a prediction of on-board behaviour, not the raw milliseconds. The plant model behind the closed-loop and collision-warning results is a first-order approximation and does not capture wheel slip, motor deadband, or battery sag, all of which are present on the real chassis. None of this changes the ordering the results show, raw beats filtered beats fused, open-loop beats closed-loop, warning-before-autobrake beats autobrake-only, but the specific numbers here should be treated as a reproducible simulation baseline to be checked, not replaced, by physical testing. The two forward-collision scenarios in Section IX-F, unlike the range-fusion and lane-pipeline results above, remain single representative runs rather than multi-seed averages; a full statistical treatment requires re-running the original simulation harness (rather than a simplified reconstruction) to preserve the camera-saturation-to-ultrasonic handoff behaviour specific to Scenario B, and is left as near-term follow-up work.

V. CHALLENGES AND FUTURE SCOPE

A platform this size runs into a smaller version of every challenge the wider field is still working through. Extreme weather and rare edge cases still trip up well-funded commercial systems, and precision sensors like spinning lidar are still expensive enough to keep out of most builds like this one. On the regulatory side, public trust, liability after an incident, and traffic law written for human drivers are all open questions that no amount of good engineering on a single vehicle settles by itself, and Koopman and Wagner's point that mileage accumulation alone cannot prove the reliability a deployable system needs applies just as much to a bigger version of this platform as it does to a production car [23].

For this specific platform, here is what we would tackle next, roughly in the order we would do it:

➢ *Learned Perception*

Swap the classical seven-layer pipeline in Section V for a small trained model along the lines of Section II-C, most likely a compact single-stage detector or a light segmentation head trained on images from the platform's own camera. The Canny threshold pair in Section V-B is the single most lighting-sensitive part of the current pipeline, and that is the first thing a learned model would fix.

➢ *Dedicated Inference Hardware*

Move that learned model onto a dedicated graphics or neural accelerator board instead of the Pi's general-purpose CPU. Fig. 8 and Fig. 9 already show where the frame budget goes; a heavier model needs headroom that a shared CPU running the whole control loop cannot spare.

➢ *ROS 2 Migration*

Replace the simple serial protocol in Table 3 with a proper ROS 2 publisher/subscriber setup, following Macenski et al.'s account of what ROS 2 buys you in real-time behaviour and multi-robot support [21], and making the platform interoperable with Autoware-class stacks as discussed in Section II-G.

➢ *Physical and CARLA-Based Validation*

Check the simulation results in Section IX against physical bench and track testing, and, before that, against a full physics- and rendering-based simulator like CARLA [7], which we did not have access to for this paper but which would remove the plant-model and sensor-model simplifications flagged in Section IX-G while still coming before the cost and risk of on-road testing.

➢ *Ackermann Steering*

Move the chassis from differential skid-steering to Ackermann-style front-wheel steering. That brings the kinematics closer to an actual passenger car, at the cost of a more complex steering linkage, and makes the model-predictive control literature in Section II-E directly applicable instead of just a loose analogy.

VI. CONCLUSION

This paper walked through the technical layers of an autonomous-vehicle software stack, sensing and fusion, perception, localisation, planning, and control, and then built a low-cost, two-computer prototype that puts each of those layers on real, buyable hardware, wiring, pin assignments, and code included. Sections V through VII gave the lane-perception pipeline, the range-fusion filter, and the planning and collision-warning logic in full; Section IX ran every piece of that code inside a seeded simulation of the vehicle's sensors and kinematics and reported what it actually produced, not what we expected it to produce. The filtering results, the timing breakdown, the lane-centring response, and the forward-collision scenarios all came out consistent with what the literature in Section II would predict, and in the two collision-warning runs the vehicle detected a hard-braking lead vehicle, escalated through the warning states, and stopped short of a collision with several metres to spare. We think that is reasonably strong evidence that the core lessons of full-scale

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug1405

autonomous-vehicle work, do not trust one sensor alone, use a predict-correct filter instead of a single reading, and check the safety condition before the comfort condition on every cycle, hold up even at this small, reproducible scale. Everything needed to rebuild it, wiring, code, and simulation scripts, is in this paper or the supplementary material alongside it, and Section X lays out where we would take it next.

DECLARATION

➢ *Availability of Data and Material*

All simulation scripts, seeds, and raw output used to generate the results in Section IX (range-fusion, lane-pipeline, closed-loop lane-centring, and forward-collision scenarios) are available as supplementary material accompanying this article. The Arduino firmware and Raspberry Pi control code referenced in Sections IV, V, VI, and VII are available from the corresponding author upon reasonable request.

➢ *Competing Interests*

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

➢ *Funding*

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

➢ *Authors' Contributions*

Srikanth A. conceived and supervised the study, and reviewed and edited the manuscript. Komal Sai Raj A. designed the hardware platform, wiring, and firmware, and contributed to drafting the manuscript. Sai Revanth S. developed the perception pipeline, Kalman filter, and simulation harness, and contributed to drafting the manuscript. Durga Venkatesh Janaki. contributed to the mechanical chassis design, system integration, and manuscript review. All authors read and approved the final manuscript.

ACKNOWLEDGEMENTS

The authors thank the Department of Computer Science and Engineering, KL University, and the Department of Mechanical Engineering, Aditya University, for institutional support during this work.

REFERENCES

[1]. SAE International, "Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles," SAE Standard J3016_202104, Apr. 2021.

[2]. J. Kocic, N. Jovicic, and V. Drndarevic, "Sensors and sensor fusion in autonomous vehicles," in Proc. 26th Telecommunications Forum (TELFOR), Belgrade, Serbia, 2018, pp. 420-425.

[3]. J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 2016, pp. 779-788.

[4]. C. Cadena, L. Carlone, H. Carrillo, Y. Latif, D. Scaramuzza, J. Neira, I. Reid, and J. J. Leonard, "Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age," IEEE Trans. Robotics, vol. 32, no. 6, pp. 1309-1332, Dec. 2016.

[5]. G. Welch and G. Bishop, "An Introduction to the Kalman Filter," Univ. North Carolina at Chapel Hill, Chapel Hill, NC, USA, Tech. Rep. TR 95-041, 1995.

[6]. B. Paden, M. Cap, S. Z. Yong, D. Yershov, and E. Frazzoli, "A survey of motion planning and control techniques for self-driving urban vehicles," IEEE Trans. Intelligent Vehicles, vol. 1, no. 1, pp. 33-55, Mar. 2016.

[7]. A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in Proc. 1st Annu. Conf. Robot Learning (CoRL), Mountain View, CA, USA, 2017, pp. 1-16.

[8]. M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: An open-source Robot Operating System," in Proc. ICRA Workshop on Open Source Software, Kobe, Japan, 2009.

[9]. S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi, "Autoware on board: Enabling autonomous vehicles with embedded systems," in Proc. ACM/IEEE 9th Int. Conf. Cyber-Physical Systems (ICCPS), Porto, Portugal, 2018, pp. 287-296.

[10]. A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in Advances in Neural Information Processing Systems (NeurIPS), vol. 25, 2012, pp. 1097-1105.

[11]. R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR), Columbus, OH, USA, 2014, pp. 580-587.

[12]. J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR), Boston, MA, USA, 2015, pp. 3431-3440.

[13]. O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional networks for biomedical image segmentation," in Proc. Int. Conf. Medical Image Computing and Computer-Assisted Intervention (MICCAI), Munich, Germany, 2015, pp. 234-241.

[14]. K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 2016, pp. 770-778.

[15]. M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars," arXiv preprint arXiv:1604.07316, 2016.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/26aug1405

[16]. M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit, "FastSLAM: A factored solution to the simultaneous localization and mapping problem," in Proc. AAAI Conf. Artificial Intelligence, Edmonton, AB, Canada, 2002, pp. 593-598.
[17]. S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann, et al., "Stanley: The robot that won the DARPA Grand Challenge," J. Field Robotics, vol. 23, no. 9, pp. 661-692, 2006.
[18]. C. Urmson, J. Anhalt, D. Bagnell, C. Baker, R. Bittner, M. N. Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer, et al., "Autonomous driving in urban environments: Boss and the Urban Challenge," J. Field Robotics, vol. 25, no. 8, pp. 425-466, 2008.
[19]. A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? The KITTI vision benchmark suite," in Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR), Providence, RI, USA, 2012, pp. 3354-3361.
[20]. C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "DeepDriving: Learning affordance for direct perception in autonomous driving," in Proc. IEEE Int. Conf. Computer Vision (ICCV), Santiago, Chile, 2015, pp. 2722-2730.
[21]. S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot Operating System 2: Design, architecture, and uses in the wild," Science Robotics, vol. 7, no. 66, 2022.
[22]. International Organization for Standardization, "ISO 26262-1:2018 Road vehicles - Functional safety" and "ISO/PAS 21448:2019 Road vehicles - Safety of the intended functionality," Geneva, Switzerland, 2018-2019.
[23]. P. Koopman and M. Wagner, "Challenges in autonomous vehicle testing and validation," SAE Int. J. Transportation Safety, vol. 4, no. 1, pp. 15-24, 2016.
[24]. N. Navet and F. Simonot-Lion, Eds., Automotive Embedded Systems Handbook. Boca Raton, FL, USA: CRC Press, 2008.
[25]. S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," in Advances in Neural Information Processing Systems (NeurIPS), vol. 28, 2015, pp. 91-99.