FEEDBACK CONTROL AND VEHICLE DYNAMICS IN AN ADAPTIVE CRUISE CONTROL SYSTEM

by

Wesley Crapo

A senior thesis submitted to the faculty of

Brigham Young University - Idaho

in partial fulfillment of the requirements for the degree of

Bachelor of Science

Department of Physics

Brigham Young University - Idaho

February 2026

BRIGHAM YOUNG UNIVERSITY - IDAHO

DEPARTMENT APPROVAL

of a senior thesis submitted by

Wesley Crapo

This thesis has been reviewed by the research advisor, research coordinator, and department chair and has been found to be satisfactory.

Date	Jon Johnson, Advisor
Date	David Oliphant, Committee Member
Date	Lance Nelson, Committee Member
Date	R. Todd Lines, Thesis Coordinator
Date	Evan Hansen, Department Chair

ABSTRACT

FEEDBACK CONTROL AND VEHICLE DYNAMICS

IN AN ADAPTIVE CRUISE CONTROL SYSTEM

Wesley Crapo

Department of Physics

Bachelor of Science

This work investigates the challenges associated with implementing an Adaptive Cruise Control (ACC) system on an older car. The objective was to develop a prototype system capable of measuring the distance to a lead vehicle using a LiDAR time-of-flight sensor and processing these measurements on a Raspberry Pi 4B to estimate relative speed. Vehicle speed would be obtained through the onboard diagnostics (OBD-II) interface. By combining measured distance from the lead-vehicle to the ego-vehicle (your vehicle), relative velocity, and ego-vehicle speed, a second-order feedback control law was designed to regulate following distance as a function of speed.

The proposed actuation method employed a relay driven by the Raspberry Pi's General Purpose Input/Output (GPIO) pins to interface with the vehicle's cruise control circuitry. By grounding the appropriate control lines, the sys-

tem could emulate button presses and command acceleration or deceleration through the factory cruise control module.

Although full closed-loop operation was not achieved, a functional sensing and decision platform was developed. The distance sensor was successfully integrated and mounted on the vehicle, and the embedded system produced real-time recommendations for cruise control adjustments.

Simulation programs were created to model vehicle-following behavior, revealing significant oscillatory responses and highlighting the importance of proper damping in second-order control systems. These results demonstrate both the feasibility of low-cost ACC prototyping and the substantial practical challenges associated with sensing reliability, integration, and stable control implementation.

ACKNOWLEDGMENTS

I want to acknowledge my research advisor, Dr. Jon Paul Johnson for his endless ability to deal with my questions and take the time out of his busy day each week to help me put this paper together. I would also like to acknowledge Dr. Lance Nelson and Dr. David Oliphant for their willingness to help me with my odd project, completely separate from their respective specialties.

Contents

List of Figures

Chapter 1

Introduction

Adaptive Cruise Control (ACC) is one of the foundational technologies enabling modern driver-assistance systems and automated driving. By automatically adjusting vehicle speed to maintain a safe following distance, ACC reduces driver workload, improves traffic flow stability [1], and enhances roadway safety. As vehicles progress toward higher levels of automation, reliable longitudinal control remains a critical prerequisite for fully autonomous operation.

Human drivers exhibit delayed reaction times, inconsistent spacing behavior, and frequent speed oscillations. These effects contribute to traffic instability, rear-end collisions, and reduced fuel efficiency. ACC systems mitigate these issues by continuously monitoring the distance and relative speed to a lead vehicle and adjusting throttle and braking accordingly. Consequently, ACC has become a standard feature in many modern vehicles and a central component of advanced driver-assistance systems (ADAS).

1.1 History of Adaptive Cruise Control

During one of my early driving experiences in Utah, I became acutely aware of the difficulty of maintaining a constant speed in moderate traffic. Frequent acceleration and deceleration of surrounding vehicles required continual manual adjustment of the cruise control setpoint over an extended period. This experience led me to consider whether a vehicle could automatically adapt its speed to that of a leading car. Subsequent investigation revealed that such functionality already existed in the form of Adaptive Cruise Control systems.

As has been observed many times, Mercedes' flagship vehicle line, the S-Class, implemented pioneering technologies including anti-lock braking, crumple zones, airbags, electronic stability control, braking assist system, active body control, and finally the "DISTRONIC" system. In 1998, they released the "DISTRONIC" system. This system, when on the highway, uses radar to sense the car ahead and adapt to that car's speed. This marked the first implementation of the concept in a production vehicle..

Later, the "DISTRONIC PLUS" system was created which allowed the car to slow to a stop, then smoothly accelerate with traffic. The system was then combined with cameras and helped with emergency braking and lane centering. After lane centering came predictive steering to avoid accidents. Finally, from predictive steering and early research platforms such as the DARPA Grand Challenge vehicles[2], came today's full self-driving vehicles. Longitudinal autonomy refers to a vehicle's ability to control its motion along the direction of travel—specifically acceleration, deceleration, and speed regulation—without driver input. Adaptive Cruise Control is a primary enabling technology for this capability and is therefore often considered the backbone of longitudinal autonomy [3, 4]. As a result, most manufacturers have adopted ACC systems across their vehicle product lines.

By 2025, most cars can be optioned with some form of Adaptive Cruise Control (ACC). After studying the evolution of Adaptive Cruise Control, I became interested in obtaining a vehicle equipped with DISTRONIC. I spent considerable time searching, but ultimately concluded that these vehicles are difficult to find and not readily accessible. This problem bothered me for years until I was in the Physics Intermediate Lab learning about Arduinos and their ability to connect to a myriad of sensors. I was very excited, but never had time to figure out how to implement it in my cruise control.

I then realized the beauty of the senior thesis: I could design my own system and have constant interaction with professors about it. It followed that I had to figure out a more academic reason to study this problem, and I discovered the way manufacturers control their Adaptive Cruise Control systems. The programs I found rely on large decision libraries that define what to do when approaching a car at specific relative velocities and distances. While effective, these systems often obscure the underlying physics[5]. I felt that a system should be able to adapt properly to circumstances using a clear, continuous control function instead. I decided to build the project and test it for myself.

To enable Adaptive Cruise Control, the system must first measure the longitudinal traffic state. The following section introduces the sensing technologies used to obtain these quantities.

1.2 Distance Sensing Technologies

An ACC system must measure the longitudinal traffic state using

$$s = \text{separation distance to the lead vehicle}$$

so

$$s(t) = x_{\text{lead}}(t) - x_{\text{ego}}(t)$$

and

$$\dot{s} = \text{relative speed.}$$

Here, *longitudinal* refers to motion along the direction of travel of the road or vehicle (forward and backward), as opposed to lateral motion such as lane changes or steering. Without reliable measurement of these two quantities, the system cannot determine whether to accelerate, coast, or decelerate. The choice of sensing technology, therefore, directly influences control performance and overall system stability.

1.2.1 Radar

Automotive radar operates at millimeter-wave frequencies (typically 77 GHz). The system emits electromagnetic waves and measures the reflected signal delay and Doppler shift.

Radar measures both distance and speed directly and maintains performance in rain, fog, and snow. Engineers therefore most commonly use radar as the primary ACC sensor[6].

1.2.2 LiDAR

LiDAR (Light Detection and Ranging) uses near-infrared laser pulses (905 nm or 1550 nm) to measure time-of-flight

$$s = \frac{ct}{2}$$

where s is the measured distance, c is the speed of light and t is the measured time-of-flight. Because optical wavelengths are short, LiDAR achieves centimeter-scale

resolution and produces detailed three-dimensional point clouds of the environment. This high spatial resolution makes LiDAR attractive for mapping and object detection tasks.

However, fog, rain, and snow scatter optical waves and degrade performance. Highly reflective or low-reflectivity surfaces also affect return signal strength. LiDAR systems typically estimate speed indirectly by tracking objects across successive frames rather than measuring it directly. As a result, while LiDAR excels at geometry, it may require additional processing to obtain stable speed information.

1.2.3 Camera-Based Sensing

Cameras operate as passive optical sensors that measure light intensity rather than emitting energy. The system must therefore infer distance using geometric and motion-based principles instead of direct time-of-flight physics. Because they cannot directly measure distance, cameras estimate it from image structure.

Camera systems therefore complement radar: cameras identify what an object is, while radar measures where it is and how fast it moves. Modern systems, like Mercedes' full self-driving system, often fuse these sensing modalities to improve reliability and robustness[7].

1.3 Control Theory for Adaptive Cruise Control

ACC functions as a closed-loop feedback system that regulates gap distance. The dynamics form a second-order system. The quantities appearing in the preceding equations are defined as follows:

s	Separation distance between the main (ego) vehicle and the lead vehicle (gap distance)
$\dot{s} = \frac{ds}{dt}$	Relative speed between the vehicles ($\dot{s} = v_{lead} - v_{ego}$)
$\frac{d\dot{s}}{dt}$	Time derivative of the relative speed
a_{lead}	Longitudinal acceleration of the lead vehicle
a_{ego}	Longitudinal acceleration commanded for the ego vehicle
u	Control command generated by the controller
k_p	Proportional gain controlling distance error response
k_d	Derivative gain providing damping against relative speed changes
s_{ref}	Desired following distance (reference gap)

We have

$$\dot{s} = \frac{ds}{dt}$$

and

$$\ddot{s} = \frac{d\dot{s}}{dt} = a_{\text{lead}} - a_{\text{ego}}.$$

A proportional-derivative control law regulates spacing,

$$u = k_p(s - s_{\text{ref}}) + k_d\dot{s}.$$

This system behaves like a mass-spring-damper system, where k_p acts as stiffness, k_d provides damping and s_{ref} defines the equilibrium spacing.. The proportional term corrects position error, while the derivative term counteracts rapid changes in relative speed. Proper tuning determines whether the system responds smoothly or oscillates[5, 8–10].

The objective is not simply to eliminate spacing error, but to do so in a way that feels stable and predictable to the driver. Overshoot, hunting, or delayed response all reduce comfort and confidence in the system.

1.4 Embedded Systems and Implementation

Engineers build ACC prototypes using embedded processors such as microcontrollers or single-board computers to interface with sensors and actuators. These systems process sensor data in real time, execute control algorithms, and ensure deterministic timing. Designers typically use automotive communication networks such as CAN bus or other high-reliability interfaces, although production automated driving systems may employ more complex proprietary architectures integrating high-bandwidth data links and centralized processing [11]. Software is commonly written in C++ or Python, depending on performance requirements.

Implementing an ACC system requires integrating several interdependent components. First, distance sensor hardware must be selected, mounted, powered, and protected from environmental effects. As demonstrated later in this thesis, mounting geometry and enclosure design significantly influence performance.

Second, raw sensor measurements must be converted into usable state information. This includes estimating relative speed from distance measurements and filtering measurement noise to avoid unstable control responses.

Third, the longitudinal control algorithm must convert spacing error into acceleration or deceleration commands. In this project, the algorithm is intentionally based on physical feedback principles rather than large decision libraries.

Fourth, actuator command generation translates desired acceleration into physical vehicle action. While this prototype focuses primarily on sensing and control computation, practical systems must consider throttle limits, braking authority, and comfort constraints.

Finally, safety constraints and operational limits must be respected. A mathematically stable controller may still be unacceptable if it produces dramatic or un-

predictable behavior[12–15].

The system must maintain stability, comfort, and safety while responding smoothly to traffic.

1.5 Original Project Objectives

This project aimed to design and evaluate a prototype Adaptive Cruise Control framework. The first objective was to design and integrate a LiDAR-based sensing module capable of measuring forward vehicle distance in realistic driving conditions. This required hardware integration, enclosure design, and embedded processing implementation.

The second objective was to implement a physics-based feedback controller grounded in classical dynamics rather than heuristic rule sets. The intent was to determine whether a transparent and continuous control law could reproduce the essential behavior of Adaptive Cruise Control.

The third objective was to evaluate system stability through simulation and experimental testing, including sensitivity to measurement noise and implementation constraints.

Finally, this work sought to compare a physically derived control approach to rule-based systems commonly used in commercial vehicles.

The following chapters describe system architecture, hardware integration, software implementation, experimental testing, and performance evaluation of the prototype Adaptive Cruise Control system.

Before describing the system design, it is necessary to establish the physical constraints that govern sensing and control performance.

Chapter 2

Physical Constraints and System Limitations

This chapter establishes the fundamental physical constraints and governing principles underlying the sensing and control system developed in this work. The system is analyzed through physical modeling, optics, signal theory, and vehicle dynamics. Rather than relying solely on observed performance, the analysis is grounded in first principles, using governing equations to explain system behavior. The limitations discussed in this chapter arise directly from the physics of light propagation, signal detection, numerical estimation, and feedback control dynamics [4, 16].

2.1 Optical Time-of-Flight Distance Measurement

The chosen distance sensor used optical time-of-flight to determine a distance. It emitted a short light pulse, received a reflection, and measured the time between emission and return. From the measured time, we determine the travel time of light, traveling at the speed of light, to travel from the laser to the target and back. The

time for light to travel to the target is therefore half the measured time, assuming our speed is much less than the speed of light.

The sensor computed the distance using

$$s = c\frac{t}{2} \tag{2.1}$$

where $c = 3.00 \times 10^8\,\mathrm{m/s}$ and t is the measured round-trip time. This equation shows that the sensor fundamentally performs a timing measurement rather than a geometric one. Therefore, any uncertainty in timing directly becomes an uncertainty in distance. This is called timing jitter and is converted into distance using

$$\sigma_s = c\frac{\sigma_t}{2} \tag{2.2}$$

If $\sigma_t = 1\,\mathrm{ns}$, then $\sigma_s \approx 0.15\,\mathrm{m}$. This matches the short-range noise I observed.

2.2 Optical Propagation Limits

The optical time-of-flight sensor used in this project is subject to several physical propagation limits that determine the maximum usable measurement distance. As light travels from the transmitter to the target and back to the receiver, multiple effects reduce the strength and reliability of the returned signal. These include beam spreading, surface reflectivity, atmospheric attenuation, and fundamental geometric power loss.

Table 2.1 summarizes the primary mechanisms that limit detection range and measurement reliability. Each phenomenon is listed along with a representative governing equation and a brief explanation of its effect on system performance.

The variables appearing in the following equations are defined as follows.

A_{spot}	Area illuminated by the laser beam at distance R
R	Distance from the sensor to the target
θ	Full beam divergence angle of the optical emitter
I_r	Reflected optical intensity returning to the sensor
I_0	Incident optical intensity at the surface
ρ	Surface reflectivity (albedo) of the target or air density (context dependent)
α	Angle between the incident beam and the surface normal
$P(R)$	Optical power remaining after propagation distance R
P_0	Transmitted optical power
β	Atmospheric attenuation coefficient
P_r	Received optical power at the detector
P_t	Transmitted optical power from the emitter
A_r	Effective receiving aperture area of the detector
P_{min}	Minimum detectable optical power of the sensor
R_{max}	Maximum theoretical detection range

Figure. 2.1 illustrates these variables.

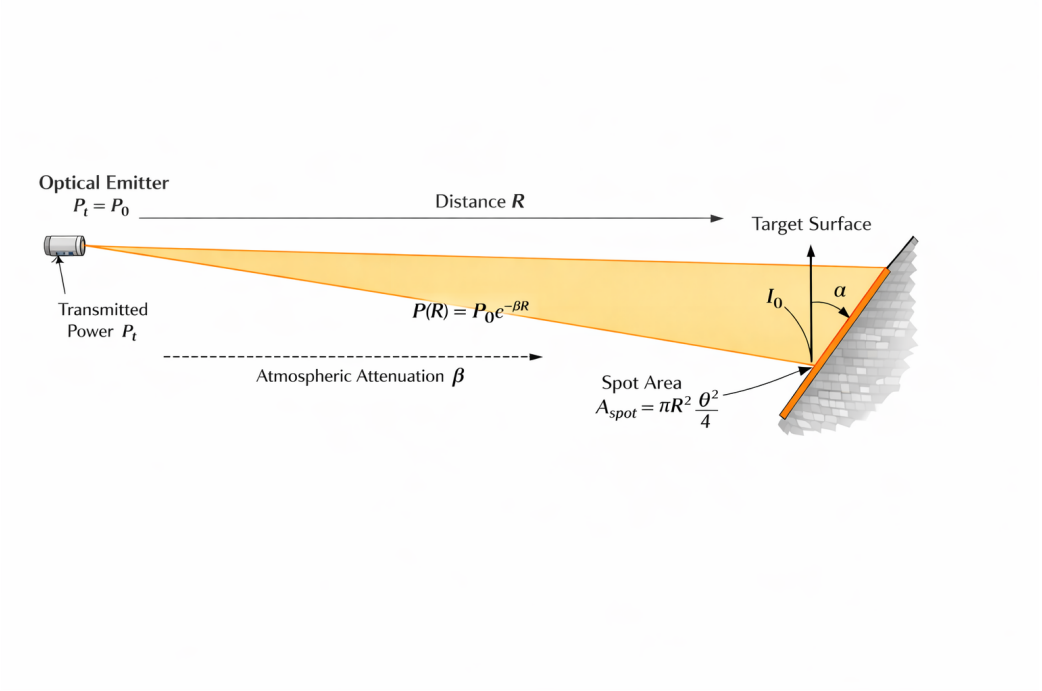


Figure 2.1 Schematic illustrating optical propagation between the sensor and a target at distance R. The diagram defines key variables including transmitted power P_t, received power P_r, beam divergence angle θ, and illuminated spot area A_{spot}. It demonstrates beam spreading, atmospheric attenuation, and reflection geometry, which collectively determine the strength of the returned signal and limit measurable range. This was created by ChatGPT from a drawing I made.

Table 2.1 Optical propagation mechanisms limiting sensor range and measurement reliability.

Phenomenon	**Governing Equation**	**Explanation and System Impact**
Beam Divergence	$A_{\text{spot}} = \pi \left(R \tan(\theta/2)\right)^2$	The beam spreads as distance increases, causing the illuminated area to grow. Power density decreases with range, reducing reflected signal strength.

Phenomenon	**Governing Equation**	**Explanation and System Impact**
Diffuse Surface Reflection	$I_r = I_0 \rho \cos(\alpha)$	Vehicle surfaces reflect diffusely according to Lambert's cosine law. At shallow angles, reflected intensity drops significantly, causing inconsistent readings.
Atmospheric Attenuation	$P(R) = P_0 e^{-\beta R}$	Air, dust, and moisture attenuate optical power exponentially with range. Even small β reduces long-distance signal strength.
Inverse Square Signal Loss	$P_r = \frac{P_t \rho A_r}{4\pi R^2}$	Return power decreases as $1/R^2$. Doubling distance reduces detected power by a factor of four. This strongly limits long-range detection.
Maximum Detection Range	$R_{\max} = \sqrt{\frac{P_t \rho A_r}{4\pi P_{\min}}}$	Detection range scales only with the square root of transmitted power. Large power increases produce modest range improvements.

These five mechanisms compound. Signal strength decreases due to spreading, divergence, reflectivity loss, atmospheric attenuation, and detection thresholds. This explains the practical range limits of the optical sensor [7].

2.3 Measurement and Signal Limitations

Even if optical propagation were perfect, the measurement system itself introduces several limitations that affect the accuracy and stability of the sensed distance. Time-of-flight sensors estimate distance by measuring the travel time of a reflected light

pulse. The precision of this measurement depends on electronic timing resolution, signal noise, filtering, and system latency. These effects introduce uncertainty in the measured distance and can propagate into the control system through speed estimation and feedback delay.

Table 2.2 summarizes the primary electronic and signal-processing limitations of the sensing system. Each phenomenon is accompanied by a representative equation and a brief description of its effect on measurement accuracy and control stability. The variables appearing in the following equations are defined as follows:

Again, this figure, Fig. 2.2, was made solely to understand some of these variables.

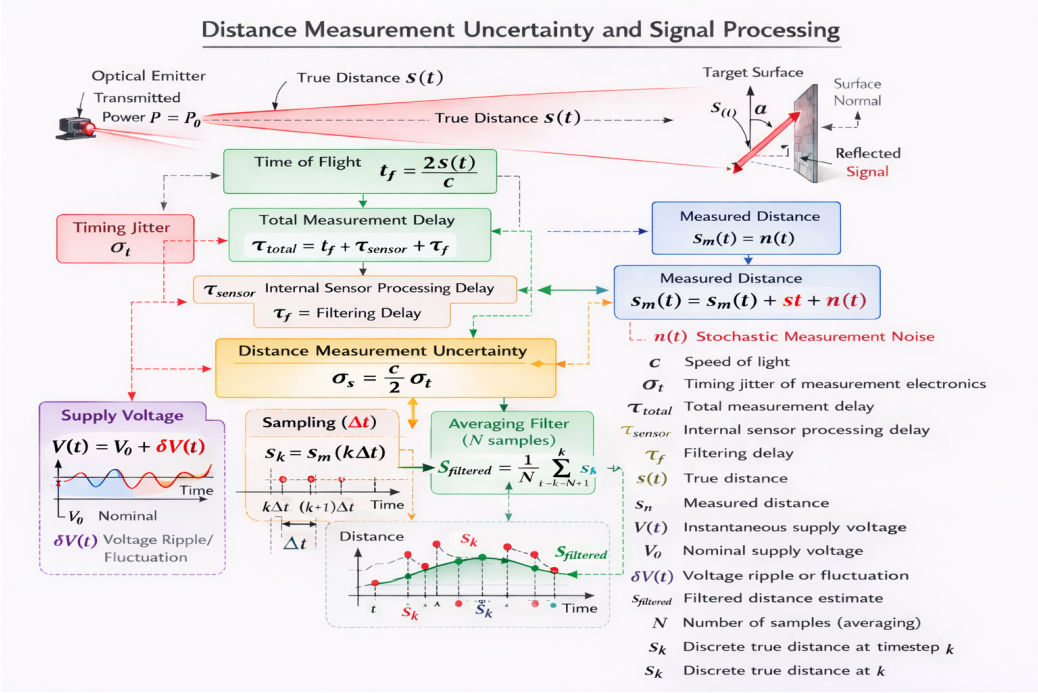


Figure 2.2 Diagram illustrating the measurement pipeline and sources of uncertainty in time-of-flight distance sensing. Contributions from timing jitter, electronic noise, signal filtering, and sampling delay are shown, along with their impact on measured distance $s_m(t)$. The figure highlights how noise and latency propagate into velocity estimation and affect control stability. This was also created with ChatGPT from a drawing I made.

Table 2.2 Electronic and signal-processing limitations affecting distance measurement accuracy.

Phenomenon	**Governing Equation**	**Explanation and System Impact**
Timing Resolution Limit	$\sigma_s = \frac{c}{2}\sigma_t$	Nanosecond timing jitter produces decimeter-scale distance uncertainty. Precision is fundamentally limited by electronics.
Sensor Delay	$\tau_{\text{total}} = \tau_{\text{sensor}} + \tau_f$	Hardware latency and filtering delay introduce phase lag. Delay reduces damping and stability margin in the control loop.
Measurement Noise	$s_m(t) = s(t) + n(t)$	Measured distance equals true distance plus stochastic noise. Noise directly contaminates speed estimation.
Power Noise	$V(t) = V_0 + \delta V(t)$	Voltage ripple perturbs timing electronics. This contributes additional jitter to measurements.
Average Filtering	$s_{\text{filtered}} = \frac{1}{N}\sum_{i=0}^{N-1} s_{k-i}$	Averaging reduces random fluctuations but slows response. Noise decreases, but delay increases.[1]

Noise reduction always trades off against response speed. The system must balance stability with responsiveness [5].

[1]When the response happens many times a second and human intervention time is far larger, this effect is less important.

2.4 Vehicle Dynamics and Control Behavior

Once distance measurements are obtained from the sensor, they must be converted into quantities that the controller can use to regulate vehicle spacing. In particular, Adaptive Cruise Control must estimate relative speed and determine how the vehicle should accelerate or decelerate to maintain a desired following distance. These actions are governed by the physical dynamics of the vehicle, including aerodynamic drag, vehicle inertia, and the structure of the feedback control system.

Table 2.3 summarizes the primary relationships connecting measured distance to vehicle motion and control response. The equations describe how distance measurements are differentiated to estimate speed, how forces act on the vehicle during motion, and how the resulting system behaves like a second-order mass–spring–damper system. These dynamics ultimately determine whether the controller responds smoothly, overshoots the desired gap, or becomes unstable. The variables used in the following equations are illustrated in Fig. 2.3 to aid interpretation of the system geometry and forces.

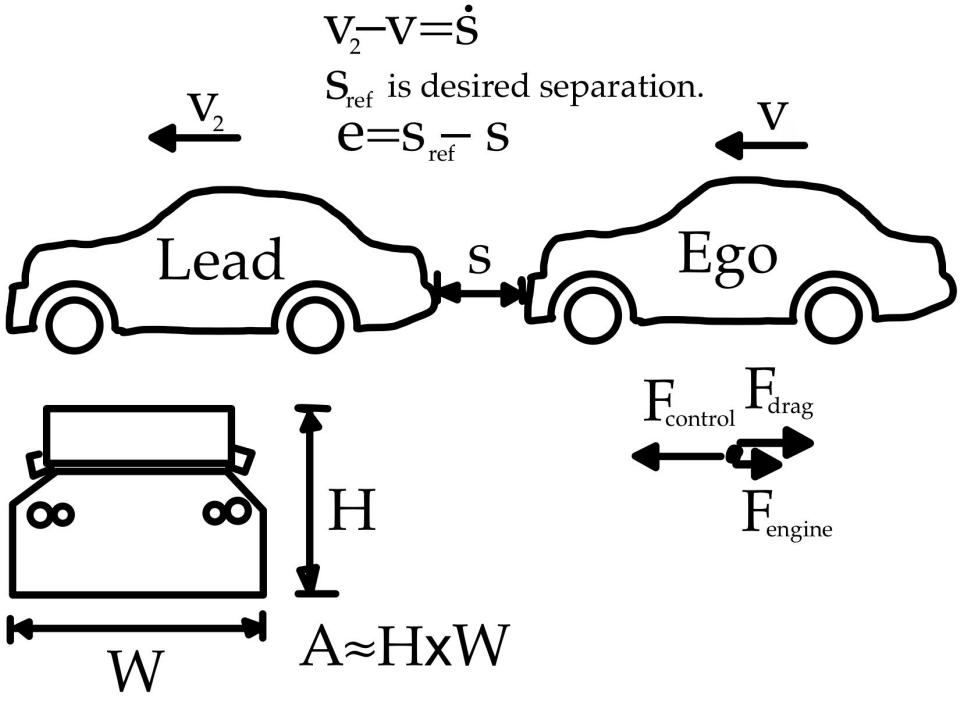


Figure 2.3 Simplified vehicle-following model showing the lead and ego vehicles, separation distance s, and relative speed $\dot{s} = v_{\text{lead}} - v_{\text{ego}}$. Forces acting on the ego vehicle include aerodynamic drag and control input. A frontal view defines the effective area A used in drag calculations. This diagram establishes the physical variables used in the longitudinal dynamics model. This diagram was courtesy of Dr. Johnson, used by permission.

The variables are defined as follows:

$\dot{s}$	Relative speed between vehicles
v	Vehicle speed
m	Vehicle mass
F_{drag}	Aerodynamic drag force
F_{control}	Control force generated by throttle or braking
F_{engine}	Engine-related resistive force
C_d	Aerodynamic drag coefficient
A	Vehicle frontal area
e	Gap error ($s - s_{\text{ref}}$)
c	Damping coefficient (equivalent dynamic system)
k	Stiffness coefficient (equivalent dynamic system)

Table 2.3 Vehicle dynamics relationships governing longitudinal motion and gap regulation.

Phenomenon	**Governing Equation**	**Explanation and System Impact**
Speed Estimation	$\dot{s} = \frac{s_k - s_{k-1}}{\Delta t}$	Finite difference differentiation amplifies measurement noise. Small distance noise produces large speed fluctuations.
Aerodynamic Drag	$F_{\text{drag}} = \frac{1}{2}\rho C_d A v^2$	Drag increases quadratically with speed and provides passive damping in longitudinal motion.

Phenomenon	**Governing Equation**	**Explanation and System Impact**
Vehicle Dynamics	$m\dot{v} = F_{\text{control}} - F_{\text{drag}} - F_{\text{engine}}$	Acceleration depends on net force divided by mass. Vehicle mass limits achievable dynamic response.
Gap Control Dynamics	$m\ddot{e} + c\dot{e} + ke = 0$	Distance regulation behaves like a mass-spring-damper system. Oscillation, overshoot, and settling time follow classical second-order behavior.

From these equations, we can more fully understand how each phenomenon restricts measurement accuracy, transient response, and closed loop stability, providing the physical basis for subsequent controller design and tuning.

2.5 Radar Comparison

Given the limitations identified in optical sensing, it is instructive to compare these constraints with radar-based systems. An additional reason for the widespread preference of radar over LiDAR in automotive Adaptive Cruise Control is that radar performance follows fundamentally different propagation physics, governed by the Radar Range Equation, rather than optical attenuation and surface reflectivity effects. Specifically,

$$R_{\max} = \left(\frac{P_t G^2 \lambda^2 \sigma}{(4\pi)^3 S_{\min}} \right)^{1/4}. \tag{2.3}$$

Radar range scales with the fourth root of transmitted power, antenna gain, wavelength, and target radar cross section, and inversely with receiver sensitivity. The variables are defined as follows:

- $R_{\max}$ — Maximum detectable range
- P_t — Transmitted power
- G — Antenna gain (assumed identical for transmit and receive)
- λ — Radar wavelength
- σ — Target radar cross section
- $S_{\min}$ — Minimum detectable signal at the receiver

This makes radar more robust for long-range automotive sensing [6].

2.6 System-Level Interpretation

These considerations establish the fundamental physical constraints that govern the performance of the sensing and control system developed in this work. The selected sensor, while subject to optimistic manufacturer specifications, operates within well-defined limits imposed by optical propagation, detector sensitivity, timing resolution, and signal noise. The analysis presented in this chapter provides a physics-based framework for understanding these limitations and for interpreting system behavior in later sections. By grounding the design in first principles, this work ensures that observed performance and system responses can be directly explained through the governing equations, rather than treated as empirical or ad hoc outcomes.

Chapter 3

System Design and Methodology

Having established the physical constraints governing sensing and control performance, this chapter describes the design and implementation of a system intended to operate within these limits. In the following sections, I will define exactly what I built and why I designed it that way. The goal is to make the final setup reproducible in its overall layout. I will finally discuss early simulation attempts [5, 16].

3.1 System Layout

This section describes the physical architecture and signal flow of the experimental Adaptive Cruise Control (ACC) sensing platform [5].

The system is composed of a Raspberry Pi 4B, a distance sensor, cables, and some other miscellaneous parts. The system is designed to measure the motion of a lead vehicle and process this information in real time to support future closed-loop cruise control logic [5]. The sensing chain begins with a forward-facing distance measurement unit and ends with a Raspberry Pi 4b responsible for data processing and the eventual control decision [7].

3.1.1 Lead Vehicle Sensing

Recall that the vehicle ahead of ours will be called the lead, our vehicle will be called the ego vehicle. The separation between the lead vehicle and the ego vehicle is measured using a "PTF-800" distance sensor (JRT Meter Technology Co.). This device provides direct line-of-sight distance measurements to objects ahead of the host vehicle. The sensor operates at a sampling rate of 50 Hz, and has an estimated range of 400 m. However, empirical testing and comparison with similar low-cost time-of-flight devices indicate that these manufacturer-stated performance figures are likely optimistic and do not reflect typical achievable range or measurement reliability [7].

Despite these limitations, inclusion of the manufacturer's data sheet remains important for completeness and reference. It provides a baseline specification against which experimentally observed performance can be compared.

Table 1-1 PTF-800
performance index

Model	PTF-800
Range (Range)	3-400@70%reflectance(1)
Maximum single measurement time Frequency	~1s 50Hz
Absolute Accuracy	+/-0.5~1m
Blind Area	3m
Resolution	0.1m
Light	905nm laser PTF-400 Class II laser
Working voltage (Voltage)	Typical value DC +3.3V, Working range (+2.5V~+3.5V)
Working current (Current)	100mA
Power consumption (Power)	330mW@3.3V
Operating Temperature	0~50℃
Communication Interface (Communication interface)	UART, default baud rate 115200bps
Serial level	TTL 3.3V
Volume	43*35*21mm
Weight (Weight)	~1 9 g

Description:
(1) The reflectivity of ordinary white wall/white paper is ~70%.

Figure 3.1 Manufacturer-provided specifications for the PTF-800 laser distance sensor, including range, accuracy, sampling rate, and power requirements. These values serve as nominal performance benchmarks; however, experimental observations indicate that real-world performance is significantly reduced, with reliable distance measuring of under 80 m instead of the claimed 400 m.

The sensor is housed inside a sealed metal enclosure originally designed for outdoor security cameras. This enclosure provides mechanical protection, environmental shielding and a flat optical window allowing unobstructed sensor operation.

The enclosure is rigidly mounted to the hood of a 2001 Buick Park Avenue Ultra.

3.1.2 Signal and Power Routing

A 10-ft USB cable provides both communication and power between the sensor and the processing unit. The cable routing was selected to minimize thermal exposure and mechanical interference as follows:

1. Exits rear of the sensor enclosure
2. Routed along the rear edge of the hood
3. Passed into the engine bay through an existing hood gap
4. Routed through the firewall via an existing wiring pass-through near the steering column
5. Directed upward beneath the dashboard
6. Plugs in at the embedded processing system (Raspberry Pi)

I selected this routing to avoid structural modifications, minimize thermal exposure, and reduce the likelihood of mechanical interference. As shown in Fig. 3.2, the sensor is mounted on the vehicle.

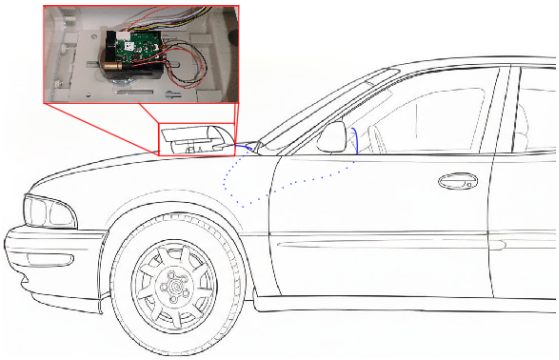


Figure 3.2 Side-view diagram of the vehicle showing the mounting location of the forward-facing distance sensor and routing of the USB cable. The diagram illustrates the signal path from the sensor through the engine bay and firewall to the embedded processing unit, highlighting practical integration constraints. This photograph was converted to a drawing using Google's integrated image processing.

3.1.3 Embedded Processing System

I am using a Raspberry Pi 4B to process sensor data, selected for its adequate computational performance for real-time estimation, General Purpose Input/Output (GPIO) availability for future hardware interfacing, low power consumption and its ease of rapid prototyping.

I mounted the Raspberry Pi on the dashboard using a windshield tablet mount to maintain visibility and mechanical stability during vehicle motion. A 7-inch Raspberry Pi display in a protective case provides system monitoring and interface capability.

Peripheral input devices (keyboard and mouse) are connected via USB during development and testing.

Power is supplied through a vehicle USB port, allowing the system to operate only when the vehicle's electrical system is active, as shown in Fig. 3.3.

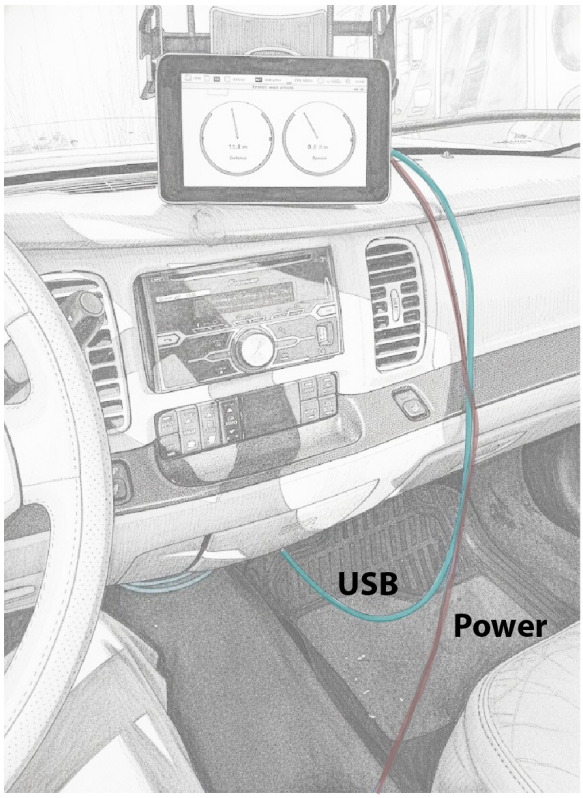


Figure 3.3 Interior view of the vehicle showing the mounted Raspberry Pi display and cable routing. The system provides real-time visualization of measured distance and estimated variables. The placement ensures visibility while maintaining stable operation during vehicle motion. This photograph was converted to a drawing using Google's integrated image processing.

3.1.4 Data Processing and Estimation Role

The Raspberry Pi performs the following real-time functions:

1. Distance acquisition from the PTF-800 at 10 Hz
2. Numerical differentiation of distance to estimate relative speed [5]
3. Future expansion:
    - OBDII (On-Board Diagnostic) port fusion for accurate current speed acquisition [16]

- Decision logic for cruise control response [5]
- GPIO-based actuation interface via relay to vehicle cruise control circuitry

At the current stage, my system is a measurement and estimation platform, not a direct vehicle control system [5].

3.2 Software Selection

I selected the Raspberry Pi due to its low cost, accessibility, and suitability for quick changes. Its Linux-based operating system allows for many well-supported libraries, straightforward file organization, and the convenience of hardware interfacing through onboard GPIO pins. These features make the Pi excellent for iterative experimentation and real-time system monitoring [11].

While a microcontroller like an Arduino would be more appropriate for my finalized embedded control product, it was not chosen for the experimental phase. Microcontrollers typically require longer development cycles for code modification, recompilation, and reflashing. In contrast, the Raspberry Pi allows live program modification, rapid debugging, and easy visualization of system variables during operation. This capability is required for validating control logic and observing real-time calculations.

The system software is written in Python. This language was selected for its readability, widespread adoption, and widespread support in a scientific computing ecosystem. This choice was created specifically to be easily understood, reproduced, and extended by other interested parties through the use of Github.

Control law computation was intentionally kept simple to prioritize transparency over optimization. The objective at this stage is not to develop production-grade control software, but rather to clearly demonstrate how sensor inputs are transformed

into control-relevant state estimates [5]. To support this, the codebase is heavily commented and structured for clarity, making the estimation and control processes straightforward to follow.

Overall, the hardware and software choices reflect a design philosophy centered on accessibility, observability, and experimental reproducibility, rather than deployment-level efficiency or automotive-grade integration.

3.3 Hardware Selection

This section explains the hardware selections made for this system, discusses alternative options considered, and identifies components that may be improved in future iterations.

3.3.1 Vehicle Platform

I selected this vehicle for both practical and personal reasons. At the beginning of the project, the car was not operational and required a complete engine and transmission replacement.

Restoring the vehicle provided me a strong and timely motivation to pair a mechanical rehabilitation effort with an engineering research objective.

Before its mechanical failure, the vehicle had proven itself as an exceptionally comfortable and stable long-distance driver, featuring extremely soft air suspension and seats, driver-assistance amenities such as a head-up display, and efficient fuel economy. These characteristics make it representative of the type of vehicle in which Adaptive Cruise Control (ACC) systems began [5].

From a technical standpoint, the vehicle offers several advantages as an experimental platform:

- It is structurally safe and suitable for extended highway operation
- It is not my primary vehicle, allowing controlled experimentation
- It provides OBDII access to vehicle speed data and to vehicle cruise control circuitry [16]
- Its interior space allows convenient mounting of prototype hardware

Rebuilding the vehicle therefore, served a dual purpose: it restored my favorite road trip vehicle while simultaneously creating a realistic and safe environment for integrating and evaluating sensing and control components of an Adaptive Cruise Control system [5]. The reassembly process is shown in Fig. 3.4.

Figure 3.4 Sequence of images showing the mechanical restoration of the test vehicle, including engine and transmission removal, installation, and final assembly. This process enabled the use of a controlled and instrumented platform for experimental evaluation of the adaptive cruise control system.

3.3.2 Sensor Selection Trade Study

I evaluated multiple sensing approaches including Texas Instruments radar, factory automotive radar modules, LiDAR systems, ultrasonic sensors, and laser rangefinders [6]. The Texas Instruments radar offered excellent detection range but was prohibitively expensive. Production automotive radar modules also required integration with proprietary CAN (Controller Area Network) vehicle communication systems, which introduced unnecessary complexity for this project. Professional LiDAR systems offered strong performance but remained expensive and complex. Ultrasonic sensors were inexpensive but limited to short-range applications unsuitable for highway scenarios.

I selected the PTF-800 laser rangefinder as a compromise between cost, range, and integration simplicity. It provides sufficient range for medium-distance detection while maintaining low system complexity and cost. Figure 3.5 shows the trade relationship between effective sensing range and cost that motivated this decision.

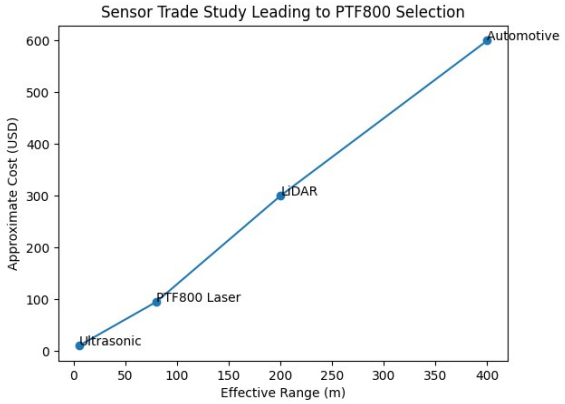


Figure 3.5 Sensor trade study comparing effective detection range versus cost for multiple sensing technologies, including radar, LiDAR, ultrasonic sensors, and laser rangefinders. The selected PTF-800 represents a compromise between cost, range, and integration complexity.

3.4 Early Simulation Model

This section is the logical progression after hardware and software development. Here, I begin to see why the second-order k_d and k_p components matter and how they relate to control theory. This model serves as the bridge between hardware, software, and vehicle dynamics [16].

3.4.1 Modeling Assumptions

In this work I only model one-dimensional motion. For the purposes of this project, there is no side-to-side motion considered. It would be unsafe to allow a single forward-facing distance sensor to control steering, so only longitudinal motion is included. Since this is not an aircraft, I assume motion occurs along one straight line. Therefore, the system tracks a single continuous separation distance between vehicles, denoted as s.

From this distance, the change in distance over time gives the relative speed:

$$v_{\text{rel}} = \frac{ds}{dt} = \dot{s} \tag{3.1}$$

The early program considers three situations: the ego vehicle moving slower than the lead vehicle, both vehicles moving at the same speed, and the ego vehicle moving faster than the lead vehicle.

The ego vehicle is modeled as a point mass. This simplifies the physics by removing rotational and suspension effects, allowing the focus to remain on longitudinal forces. This assumption appears in the program, though not always explicitly.

3.4.2 Governing Equations

The governing equations for the ego vehicle are

$$\dot{x}_e = v_e \tag{3.2}$$

$$m\dot{v}_e = F_{\text{control}} - F_{\text{drag}} - F_{\text{engine}} \tag{3.3}$$

With no control input (no cruise adjustment), speed remains constant. Only when a speed adjustment is commanded does the vehicle accelerate or decelerate [16].

The early program asks for the ego vehicle's initial speed, the lead vehicle's speed, and the initial separation distance s_0. If the ego vehicle is traveling faster than the lead vehicle, deceleration forces are applied until the speeds match. If the lead vehicle is faster, acceleration is applied until velocities match. If the speeds are equal, no change occurs.

Aerodynamic drag is included and modeled as

$$F_{\text{drag}} = \frac{1}{2}\rho C_d A v^2 \tag{3.4}$$

where ρ is air density, C_d is the drag coefficient and A is frontal area [16].

A constant engine braking force is also included. This is a simplification, since real engine braking varies with gear and RPM, but it provides a reasonable approximation for early modeling. A future version of the program will include variable engine braking.

This early program performs reasonably well and is included in Appendix A. Figure 3.6 shows the result for a lead vehicle traveling at 45 mph while the ego vehicle begins at 55 mph.

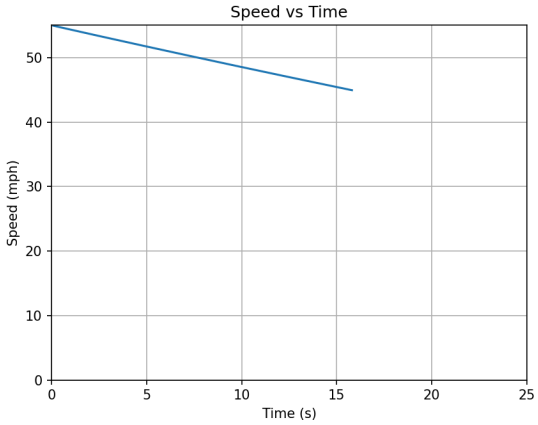


Figure 3.6 Simulation of ego vehicle deceleration from 55 mph to match a lead vehicle traveling at 45 mph using a speed-matching control strategy. The controller disengages once speeds match, resulting in an uncontrolled following distance. This demonstrates the limitation of speed-only control.

Chapter 4

Simulation and Controller Development with Results

This chapter examines the dynamic behavior encountered in both simulation and real-world implementations of Adaptive Cruise Control (ACC) distance regulation [5, 16]. In the previous chapter, a very simple original simulation program was examined. The program slowed the car to the speed of the vehicle ahead, then effectively stopped controlling the system. To continue the development of the simulation, the problem was formulated as a second-order dynamic control system, where overall stability depends primarily on the proper selection of damping parameters [5, 8].

4.1 Choosing a Mathematical Model

The original simulation model exhibits several limitations. It can generate incorrect responses when the sensor detects an overpass or roadside barrier during a turn, and it also fails to distinguish between a lead vehicle traveling at 45 mph in a 70 mph zone versus one traveling at 65 mph in the same zone. In both cases, the controller responds

identically, resulting in significantly different final following gaps. As discussed previously, a simplified longitudinal vehicle-following model was therefore adopted [4,9,10]. Using this model, the following sections analyze the resulting controller behavior.

4.1.1 State Variables

The system state vector is defined as

$$x = \begin{bmatrix} s \\ \\ v_e \end{bmatrix} \tag{4.1}$$

where

s = separation distance (gap) between the ego vehicle and the lead vehicle

v_e = ego vehicle speed

The control objective is to regulate vehicle separation distance rather than relative speed. The desired spacing is denoted as s_{ref}. The spacing error is therefore

$$e = s - s_{ref} \tag{4.2}$$

The gap dynamics follow directly from kinematics such that

$$\dot{s} = v_{lead} - v_e. \tag{4.3}$$

The ego vehicle longitudinal dynamics are approximated by

$$m\dot{v}_e = F_{control} - F_{drag} - F_{engine} \tag{4.4}$$

where $F_{control}$ represents the commanded braking force, and the remaining terms account for resistive effects [16]. Note that this is a metered step away from reality, counting the control as having impact on the brakes will not happen in the scope of

this project but will allow a treatment any physics student is well accustomed to in the subsequent sections.

4.1.2 Why the System is Second Order

Regulating following distance inherently depends on both position and speed. The control input acts as a force, which directly alters vehicle speed. Speed, in turn, determines position through time integration. Because the control input affects the derivative of speed, and distance itself is the integral of relative speed, the system contains two dynamic states [5, 16]. As a result, the system exhibits classic second-order behavior, including oscillation, overshoot, and settling characteristics that depend strongly on the amount of damping present [8].The second simulation demonstrates this behavior clearly, as shown in Fig. 4.1.

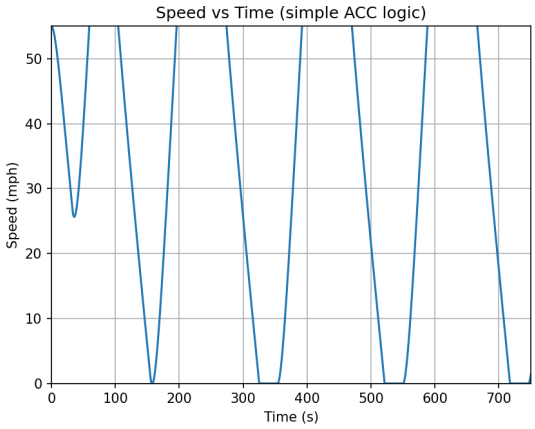


Figure 4.1 Simulation showing distance-based control of the ego vehicle when approaching a slower lead vehicle. The system attempts to regulate separation distance rather than match speed, resulting in second-order dynamic behavior including transient response and settling.

Thus, ACC gap control behaves analogously to a mass–spring–damper system [5].

4.2 Damping and Stability Optimization

Stability was hard to achieve with this controller. The central problem of ACC control is selecting a damping that prevents oscillation while maintaining responsiveness [5,8]. Most manufacturers use a proportional control [5].

4.2.1 Proportional Control Behavior

A simple proportional (P) control law was first tested:

$$u = k_p(s - s_{ref}). \tag{4.5}$$

This control law generates a commanded longitudinal acceleration, analogous to a spring-like restoring force proportional to gap error. However, because the vehicle has inertia, the system still exhibits similar behavior, namely gap oscillations, overshoot beyond desired distance and hunting around the reference gap, as shown in Fig. 4.2.

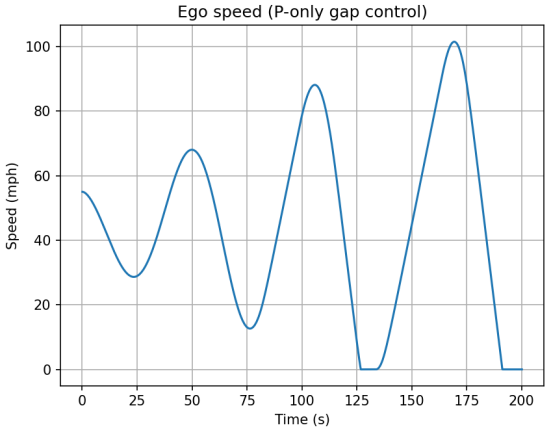


Figure 4.2 Response of the system using proportional-only control. The vehicle exhibits oscillations and overshoot around the desired following distance due to the absence of damping, illustrating the limitations of a purely stiffness-based control law.

A proportional-only controller introduces a restoring effect but lacks any damp-

ing action, so the system cannot remove excess kinetic energy and therefore shows oscillatory behavior [5, 8].

4.2.2 Adding the Derivative Term

To introduce damping, a derivative term was added, making it a Proportional-Derivative control (PD):

$$a = k_p(s - s_{ref}) - k_d\dot{s} \tag{4.6}$$

where k_p = position gain ("stiffness") and k_d = derivative gain ("damping").

The derivative term opposes relative speed, acting like a viscous damper that absorbs motion energy. This suppresses oscillations and stabilizes convergence to the desired gap [5, 16].

New issues were encountered at this stage, as shown in 4.3.

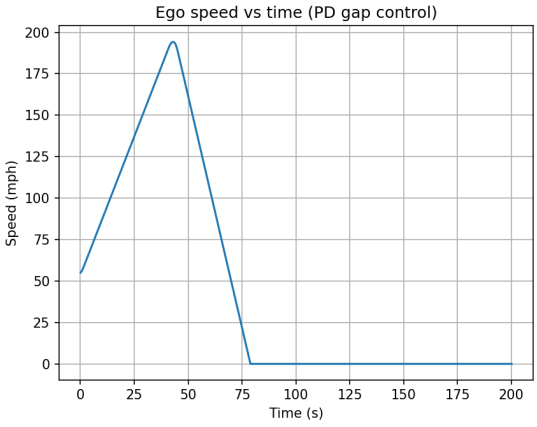


Figure 4.3 Simulation result from an incorrectly implemented proportional-derivative control law. A sign error in the derivative term leads to unstable behavior, demonstrating the sensitivity of feedback systems to implementation details.

It was simply a sign problem, in the code, but I thought it was funny. I changed one sign and found 4.4.

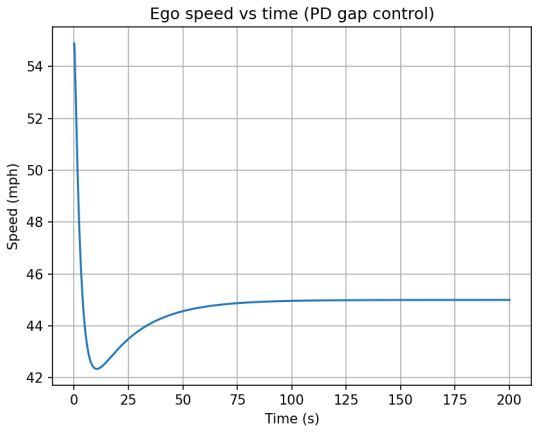


Figure 4.4 Stable system response using a correctly implemented proportional-derivative control law. The derivative term introduces damping, reducing oscillations and enabling convergence toward the desired separation distance.

4.2.3 Critical Damping

We see that this system reacts fairly well but is not perfect. The goal is to achieve critical damping. Critical damping occurs when the system returns to equilibrium as quickly as possible without oscillating. For a standard second-order system, the dynamics can be written as

$$m\ddot{x} + c\dot{x} + kx = 0. \tag{4.7}$$

This equation represents the behavior of an equivalent mass–spring–damper system, where

m Effective mass of the system

c Damping coefficient

k Stiffness constant of the spring

The stiffness constant k describes the physical spring stiffness and should not be

confused with the proportional controller gain k_p or the k_d used in the ACC control law.

Critical damping occurs when the damping coefficient equals

$$c_{\text{crit}} = 2\sqrt{km}. \tag{4.8}$$

In the ACC system, k_p plays the role of stiffness and vehicle inertia provides mass. Aerodynamic drag introduces additional natural damping [16]. Therefore, the derivative gain must supply only the remaining damping needed and I learned that

$$k_{d,crit} = 2\sqrt{k_p m} - c_{drag}. \tag{4.9}$$

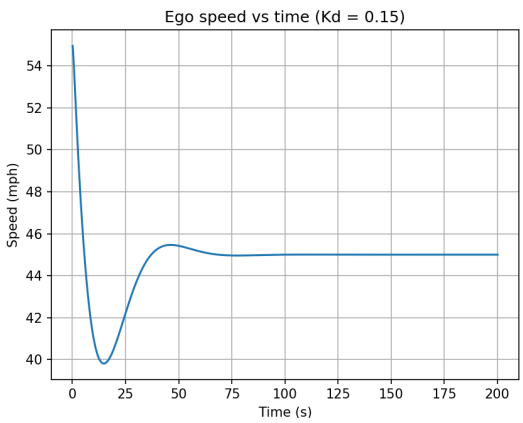
Here, c_{drag} represents the effective viscous damping introduced by aerodynamic drag, obtained by linearizing the quadratic drag force about a nominal operating speed. Physically, the derivative gain must counteract vehicle inertia but should not exceed the natural damping already provided by aerodynamic drag. Excessive damping slows system response, while insufficient damping causes oscillations [5, 8].

4.3 Early Simulation Results

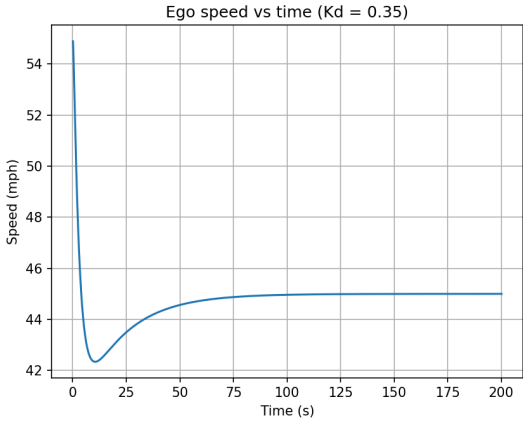
Some simulations were attempted using varying values of k_d while keeping k_p constant.

Case	Behavior
Low k_d	Oscillatory response, large overshoot
Near critical k_d	Fast settling, minimal overshoot
High k_d	Sluggish response, slow convergence

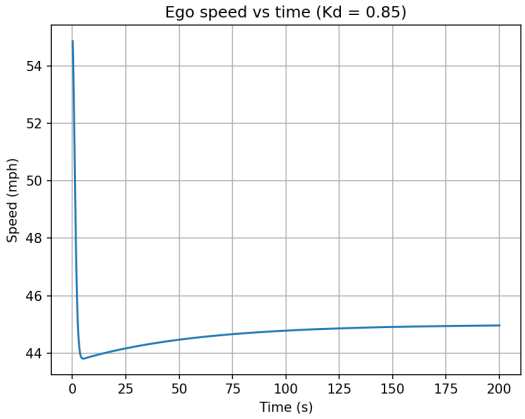
The resulting system responses are shown in Fig. 4.5.



(a) Low k_d: oscillatory response



(b) Near-critical k_d: fast settling



(c) High k_d: sluggish convergence

Figure 4.5 Comparison of system responses for varying derivative gain k_d. Low damping results in oscillatory behavior, near-critical damping produces fast settling with minimal overshoot, and high damping leads to slow convergence. These results confirm classical second-order system dynamics.

These results confirm classical second-order dynamics. The best performance occurred near the analytically predicted critical damping value [5, 8].

4.4 Failure Points and Real-World Limits

In this section, I outline the primary limitations of the simulation and the factors that prevent direct equivalence with physical vehicle behavior.

4.4.1 Sensor Performance and Measurement Uncertainty

The distance sensor presented a major practical limitation. Its long-range performance was inconsistent; near 80 meters, the output frequently alternated between a valid measurement and an error state. The signal also contained intermittent spikes that produced substantial measurement noise.

Accurate representation of this noise in simulation would require a stochastic model (e.g., Monte Carlo analysis) based on a known noise distribution. However, the sensor's noise magnitude, frequency characteristics, and distance dependence were not sufficiently characterized to construct a defensible model. For this reason, sensor noise was neglected, and the simulation assumed idealized measurements.

4.4.2 Control Sensitivity to Measurement Noise

During development, measurement noise caused excessive oscillatory control action. This was most severe in the derivative term, which amplifies high-frequency signal components by design. Without mitigation, this resulted in rapidly varying and unstable acceleration commands [5].

To reduce this effect, signal smoothing was implemented to attenuate high-frequency noise before differentiation, improving closed-loop stability. This highlights a key real-world constraint: derivative-based control is inherently sensitive to sensor quality [5, 7].

4.4.3 Model–Reality Divergence

The simulation assumes perfect distance measurement and negligible latency. In real systems, sensing delays, dropout events, and filtering introduce phase lag in the feedback loop, reducing stability margins and altering system dynamics [5, 8].

Additionally, the model assumes symmetric vehicle actuation, meaning the vehicle can accelerate and decelerate with equal authority. In reality, vehicles have unequal limits for acceleration and braking due to powertrain output and braking capability. In this project the limitation is even more pronounced because the intended implementation relies only on simple cruise control actuation rather than direct throttle or brake control. These nonlinearities further increase the difference between simulated behavior and real-world vehicle response. [16].

Consequently, a controller that appears stable in simulation may not perform equivalently when implemented on a physical vehicle [12].

Chapter 3 Takeaway

The main lesson from this chapter is that regulating following distance in Adaptive Cruise Control is fundamentally a problem of managing vehicle dynamics rather than simply matching the speed of the car ahead. Because the control input affects acceleration, acceleration changes speed, and speed changes position, the system naturally behaves like a two-stage dynamic process. As a result, the vehicle shows familiar behaviors such as overshoot, oscillation, and gradual settling.

This conclusion emerged through the development of the simulation. The first model slowed the vehicle until it matched the speed of the car ahead. While this seemed reasonable, it did not truly regulate distance and produced inconsistent following gaps depending on the speed of the lead vehicle. This limitation motivated the transition to a model that directly controls the spacing between vehicles.

Once distance regulation was introduced, the dynamic behavior became clear. A simple proportional control was first tested, which attempted to correct the distance error. However, simulations showed that this approach caused the vehicle to overshoot

the desired spacing and oscillate around it. The inertia of the vehicle allowed it to continue moving even after the error had been corrected.

Adding a derivative term introduced damping into the system. This resisted rapid changes in the distance between vehicles and reduced the oscillatory behavior. By testing different gain values, the simulations showed that the best performance occurred when the system was near critical damping. In this condition the vehicle returned to the desired spacing quickly without repeated oscillations.

This result is important because it shows that the stability of Adaptive Cruise Control depends primarily on selecting the correct balance between responsiveness and damping. Too little damping causes repeated acceleration and braking, while too much damping makes the system react too slowly.

The simulations also revealed practical limits that would appear in a real vehicle. Sensor noise, measurement dropouts, and delays can strongly affect controller stability, especially because derivative control amplifies rapid measurement changes. These effects highlight why real systems require filtering and careful tuning.

Overall, this chapter demonstrates that Adaptive Cruise Control distance regulation behaves like a damped mechanical system. Understanding and controlling this dynamic behavior is therefore the key requirement for designing a stable and comfortable ACC controller.[5, 8].

Chapter 5

Experimental Implementation and Results

This chapter presents the experimental implementation of the proposed sensing and estimation system, along with the practical challenges encountered during integration. Emphasis is placed on identifying the physical and system-level factors that limited performance, and relating these observations to the constraints established in Chapter 2.

5.1 Initial Setup Struggles

The experimental platform initially began in a difficult state. Connecting an older computer monitor without an HDMI port proved unexpectedly complicated, primarily because the required signal conversion only functioned in one direction and the necessary adapter was not available locally. After some effort, however, the system was successfully connected to a monitor 5.1.

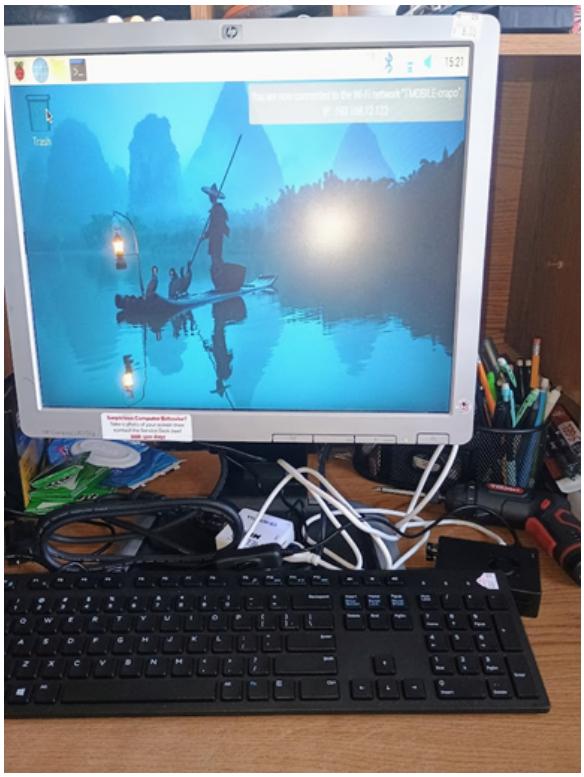


Figure 5.1 Initial system setup showing the Raspberry Pi connected to an external monitor for development and debugging. This configuration enabled visualization of system output during early testing.

After connecting to the monitor, I purchased a mouse and keyboard and began setting up the system, including installing the Raspberry Pi OS onto the SD card, which turned out to be one of the most impressive aspects of the process. Specifically, I did not initially realize that installing a complete operating system onto an SD card could be so efficient. When using a minimal OS configuration, the Raspberry Pi Imager allows the system to be programmed in just a few minutes. At several points in this project, I maintained a secondary SD card that I could repeatedly wipe and reinstall with OS settings in under five minutes. The official Raspberry Pi Imager remembers common configuration settings, making the process extremely low stress, especially when experimenting.

Once the Raspberry Pi OS was installed, more advanced work began. Working

within the Linux command line environment proved to be exceptionally efficient. The command line keeps operations clean and commands straightforward. Code execution, software installation, and system configuration can all be handled rapidly. Compared to Windows PowerShell, the Linux terminal environment was significantly more effective for this project.

After gaining familiarity with the Linux-based OS, I began modifying boot-up behavior. I started with a simple program that launched a weather application at boot, then progressed to more ambitious tasks—one of the primary reasons for using a Raspberry Pi: the GPIO (General Purpose Input/Output) pins. I connected an LCD display from an Arduino kit and programmed it to show CPU temperature, including automatic display on boot 5.2.

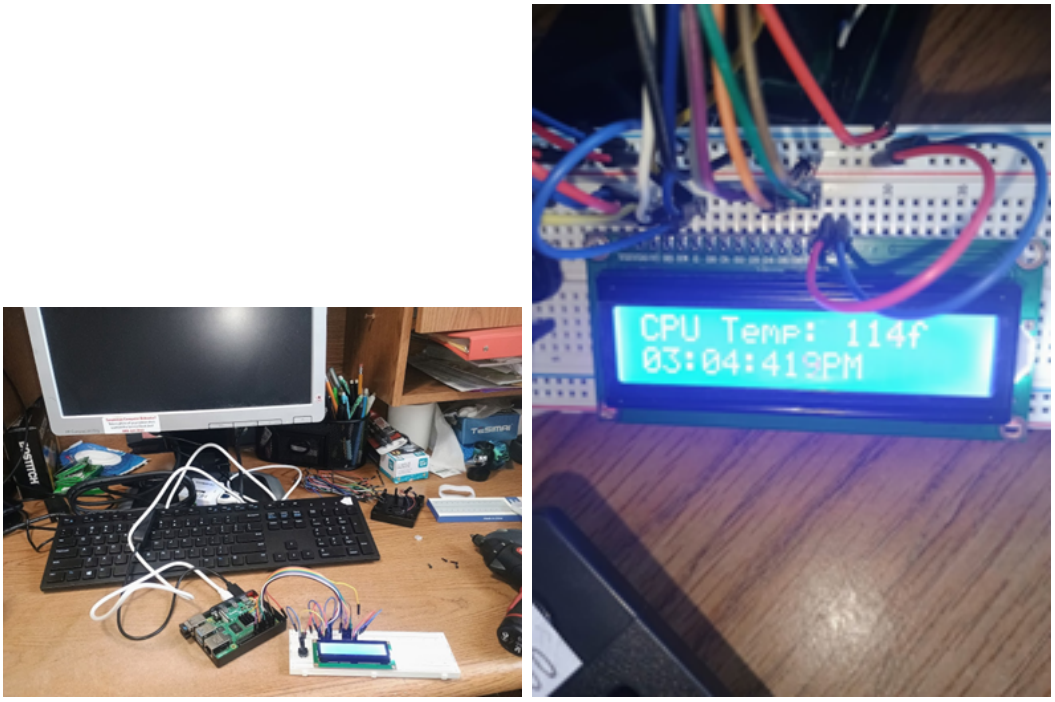


Figure 5.2 LCD display output showing system status information, including processor temperature. This interface was used to verify correct system initialization and operation.

While waiting several weeks for the JRT distance sensor to arrive from China, I

worked on additional Raspberry Pi projects. These included a button interface on the LCD breadboard. This interface cycled through weather forecasts, precipitation predictions 5.3, date, and time in hundredths of seconds.

Figure 5.3 LCD display showing real-time data output. This interface demonstrates the system's ability to present processed information during operation.

I also implemented a safe shutdown button and an LED indicator to confirm button presses.

5.2 Sensor Integration Issues

When the sensor finally arrived, it came without instructions. I spent approximately four hours attempting to connect it to my laptop. The sensor was visible to the system, but no data was received. I shifted focus to the Raspberry Pi and tested

all available documentation, again without success. It became clear that this sensor used a new control protocol not documented in common forums.

I contacted JRT directly. The message can be found in Appendix C.

After considerable troubleshooting, I resolved the issue. The manufacturer provided a translated manual, and either the translation or my interpretation was initially incorrect. Eventually, the sensor began returning real distance values in continuous measurement mode, as shown in Fig. 5.4.

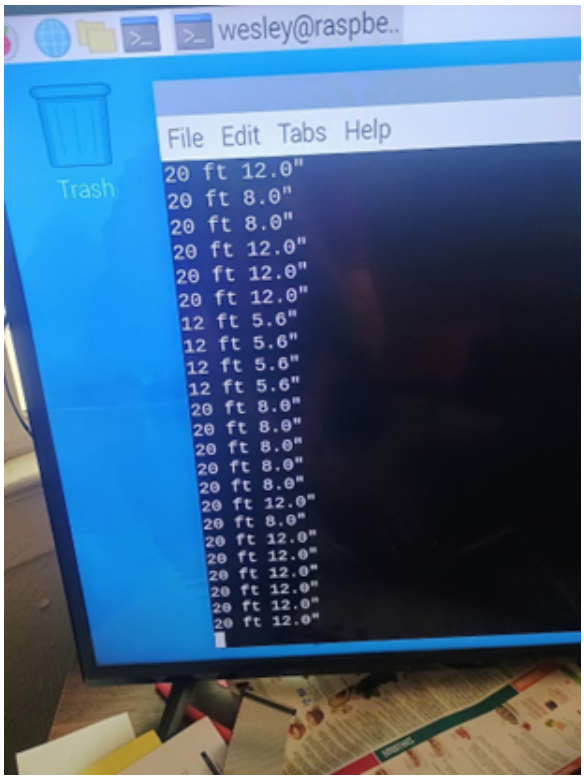


Figure 5.4 Sensor output showing measured distance values. This data stream was used to evaluate sensor performance and reliability.

I then integrated distance readings into the LCD display and began preparing for vehicle installation, as shown in Fig. 5.5.

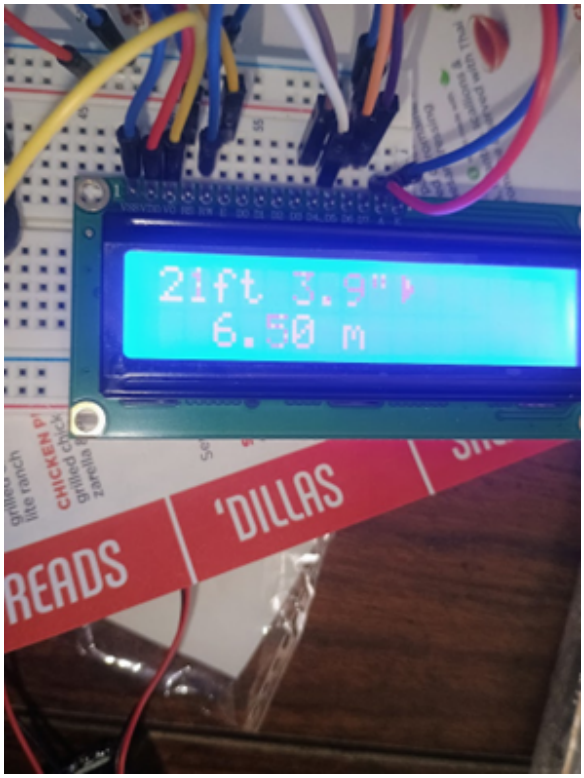


Figure 5.5 LCD displaying measured distance in real time. This visualization was used to verify correct data acquisition and communication between the sensor and processing system.

I wrote a program to display distance readings at boot and implemented a shutdown button.

Field testing revealed a serious limitation: the sensor only functioned reliably to approximately 26 meters in daylight, then returned zero beyond that. Below, I show some sensor testing 5.6.

Figure 5.6 Outdoor testing configuration of the distance sensor. This setup illustrates real-world operating conditions, including environmental exposure, which influence measurement reliability.

Manufacturer specifications claim a daylight range of 400 meters, which was not observed. I tested outdoors with an extension cord powering it 5.7 and measured about 27 meters maximum range.

Figure 5.7 Sensor testing configuration using an external power source. This setup was used to isolate and evaluate sensor behavior independent of vehicle integration.

I implemented measurement averaging to smooth noise and investigated timeout settings but saw no improvement. Night testing confirmed sunlight was not the limiting factor.

Further testing revealed the sensor capped at approximately 25.6 meters using example code. I unintentionally overwrote working code during debugging and had to rewrite. In doing so I found an error; I was not reading a byte. Testing in the vehicle now showed consistent performance to 30 meters. I then discovered that this may be due to the curvature of the windshield as operation to 80 meters was now possible when manually holding the sensor steady outside the sunroof. Ultimately, this sensor proved unreliable for long-range outdoor use but not as bad as initial testing indicated.

5.3 Display Disaster

My next major challenge involved connecting a screen directly to the Pi instead of using the simple LCD. I ordered a low-cost screen from eBay.

I struggled extensively with this display. Its configuration made login impossible, and progress made after multiple days of effort was effectively lost. This is what I got to display on the screen 5.8.

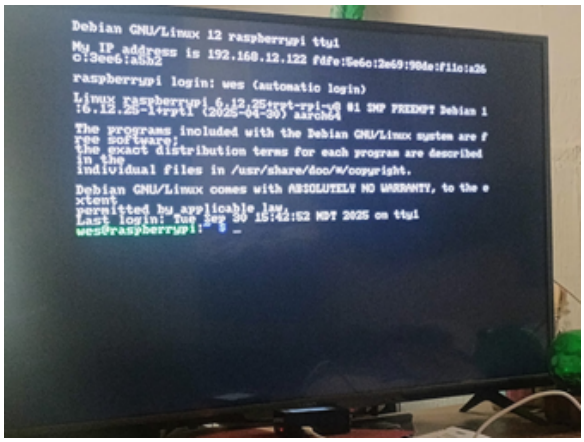


Figure 5.8 Display output when running manufacturer-provided software. The system exhibits abnormal behavior, highlighting compatibility and integration challenges.

The following sections describe a complex troubleshooting process encountered during display integration. Multiple framebuffer driver implementations and display controller configurations were tested, along with variations in GPIO assignments, SPI communication parameters, and driver initialization routines. Although several configurations successfully compiled and initialized, none initially produced visible display output, indicating deeper incompatibilities within the display interface stack.

I confirmed /dev/fb0 existed, reviewed pinouts, and attempted touchscreen overlays. Despite many variations, no image appeared, though backlight activation confirmed power.

Eventually, SPI communication produced "TV snow" style noise, confirming partial communication but incorrect initialization. Numerous OS reinstalls and boot configuration changes followed.

A breakthrough occurred when disconnecting the HDMI cable caused the display to begin functioning. This next figure, Fig. 5.9, is what could now be seen on the screen.

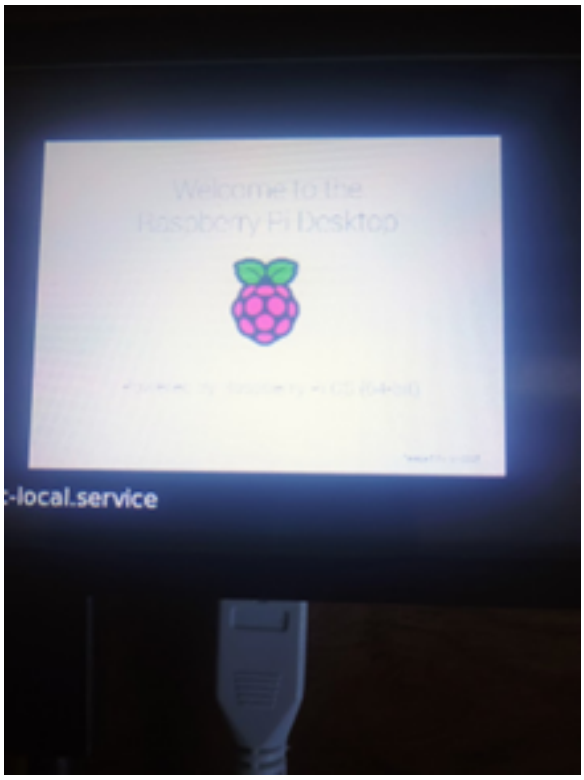


Figure 5.9 Display freezing during operation. This behavior indicates software or hardware instability encountered during system integration.

I later determined the display initialization scripts were conflicting with HDMI output.

Ultimately, I abandoned the low-cost display and purchased the official Raspberry Pi 7-inch screen. Installation required only power and a display cable. It worked immediately and reliably, as shown in Fig. 5.10.

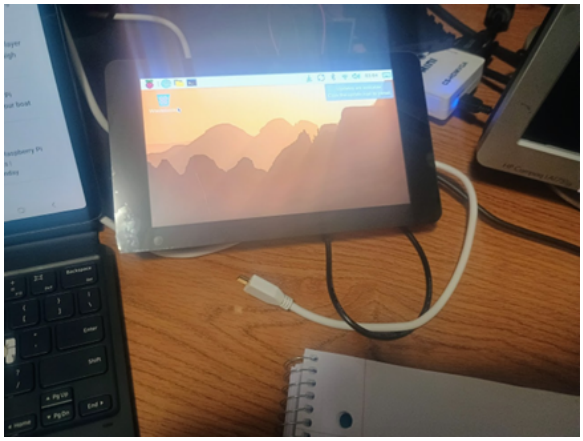


Figure 5.10 Official 7-inch Raspberry Pi display used for system monitoring. This display replaced earlier configurations and provided improved reliability.

5.4 Final Setup Setbacks

Distance data was finally displayed on-screen as seen in Fig. 5.11.

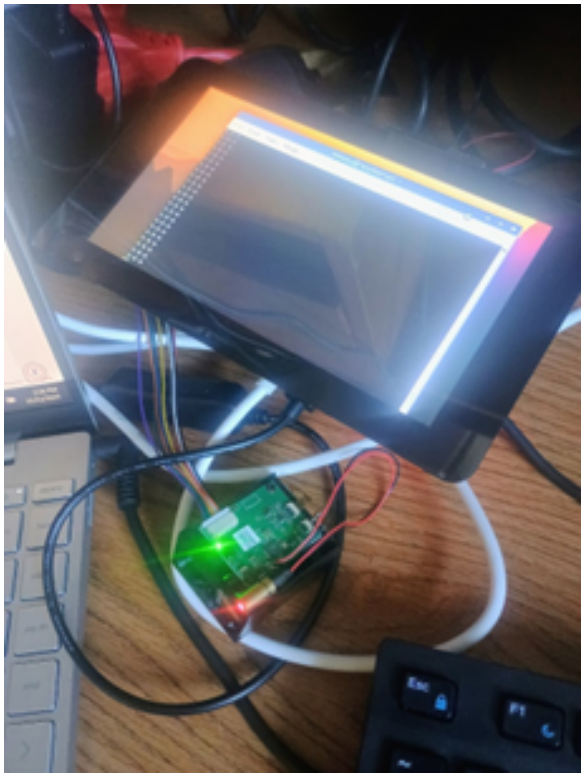


Figure 5.11 Display showing processed distance data during operation. This confirms successful integration of sensing and visualization components.

Testing in the vehicle showed stable readings to 30 m and occasional readings to 80 m only under ideal manual positioning. I began searching for mounting solutions, ultimately ordering commercial tablet mounts and a protective case.

Alternator voltage stability was improved, and the required cables were routed through the firewall by drilling openings in an existing rubber grommet. I installed the USB as seen in Fig. 5.12.

Figure 5.12 USB cable installation within the vehicle. Proper routing was necessary to ensure reliable communication and minimize interference.

With the electrical and data connections established, attention was turned to environmental protection of the forward-mounted sensor. The initial attempt, Fig. 5.13, at a waterproof enclosure proved inadequate in real-world conditions.

Figure 5.13 Sensor enclosure after environmental failure. Exposure to moisture compromised system functionality, demonstrating the importance of robust environmental protection.

After the first waterproof enclosure failed, a security camera enclosure was ordered. I started developing the user interface and added shutdown controls. I then created a fairly advanced program that shows the distance and relative speed of the following with arrows to aid in using the cruise control, Fig. 5.14.

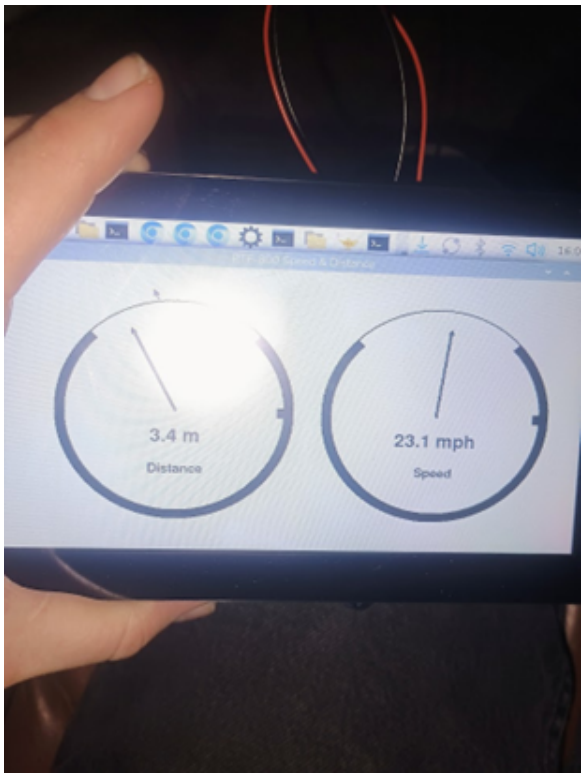


Figure 5.14 Display showing computed speed and distance values. This output represents the real-time estimation capabilities of the system.

The system was mounted and aligned. This process is documented in Fig. 5.15.

Figure 5.15 Final mounted position of the distance sensor on the vehicle. This configuration was used during experimental testing.

Unfortunately, the last bit of tragedy struck while mounting the sensor. I dropped the sensor in its enclosure on the windshield, cracking it. Afterward, the sensor failed to communicate. I tested all saved programs and SD cards; none worked. I have included a picture, Fig. 5.16, of the results the sensor communicates.

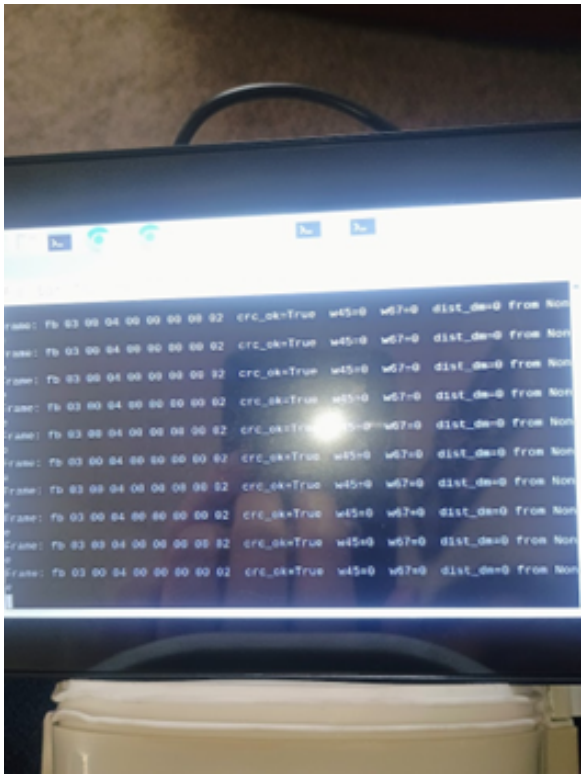


Figure 5.16 Final sensor response following damage, showing loss of valid measurement output. This represents the failure mode of the sensing system under real-world conditions.

I decided that the sensor was confirmed damaged, and I began replacement discussions with JRT, the sensor manufacturer and eventually bought another sensor. It began its ponderous travel across the world.

Overall, the experimental implementation confirms that system performance is strongly limited by sensing reliability, measurement noise, and integration constraints. These limitations are not incidental, but arise directly from the physical and signal-processing constraints established in Chapter 2. While the system successfully demonstrated real-time distance measurement and estimation, consistent closed-loop operation was prevented by hardware fragility and measurement instability. These results highlight the importance of robust sensing and signal conditioning in practical Adaptive Cruise Control implementations.

Chapter 6

Conclusion

This work investigated the feasibility of implementing an Adaptive Cruise Control system using a physics-based approach grounded in fundamental vehicle dynamics, sensing principles, and feedback control. Rather than relying on heuristic decision tables, the system was designed and analyzed using continuous governing equations to describe both sensing and control behavior.

6.1 Revisiting the Original Motivation

This project began long before it became a senior thesis. It began when I was fifteen years old, driving from Provo, Utah to Blackfoot, Idaho and repeatedly pressing the cruise control buttons in frustration. At that time, the problem appeared simple: if a driver can see that traffic ahead is slowing down, why can the vehicle not respond automatically?

Years later, that frustration evolved into a formal question: can Adaptive Cruise Control be implemented using a transparent, physics-based control law rather than relying on large rule-based decision libraries? What began as a personal annoyance

matured into an investigation of sensing physics, feedback dynamics, embedded systems, and real-world implementation limits [5, 16].

6.2 What Was Accomplished

Although the final system was not fully completed due to hardware failure, key technical milestones were achieved.

On the hardware side, a complete embedded sensing platform was designed and integrated into a 2001 Buick Park Avenue Ultra. This included:

- Rebuilding the vehicle's engine and transmission to create a viable test platform
- Selecting and configuring a Raspberry Pi 4B for real-time processing
- Designing power delivery from the vehicle electrical system
- Routing signal and power lines safely through the firewall
- Mounting and housing a forward-facing distance sensor
- Designing a user interface to display distance and relative speed

Each of these tasks required problem solving across mechanical, electrical, and software domains. Mounting geometry, enclosure design, power stability, and vibration all influenced system behavior in ways not visible in simulation.

On the software side, two major systems were developed. The first was a sensor acquisition and estimation framework capable of measuring distance, calculating relative speed, and managing startup and shutdown procedures safely. The second was a predictive control framework implementing a proportional-derivative style feedback law. This controller was tuned through simulation to reduce oscillation, eliminate hunting behavior, and predict unsafe closing speeds.

The physical sensor was damaged before final integration testing. However, a functional sensing and estimation platform was successfully developed, capable of measuring separation distance and estimating relative speed in real time. In addition, simulation models were constructed to analyze vehicle-following behavior and to investigate the stability characteristics of proportional and proportional-derivative control laws[5].

6.3 Technical Lessons Learned

The most significant lesson learned is the difference between theoretical design and physical implementation.

In simulation, Adaptive Cruise Control behaves as a clean second-order system. Stability is governed by proportional gain, derivative gain, and damping ratio. Oscillations can be eliminated by tuning parameters appropriately. The mathematics is structured and predictable.

In reality, sensor noise, communication delays, mechanical alignment, electrical ripple, and component limitations all influence performance. A control law that is stable in simulation may behave differently when measurement uncertainty and delay are introduced. Hardware failures can halt progress entirely. Low-cost components do not behave like idealized models [6, 7].

I gained extensive experience in debugging, embedded programming, serial communication protocols, signal filtering, and real-time system behavior. I also learned the importance of validating assumptions early and respecting development timelines. The engineering effort required to implement even a simplified Adaptive Cruise Control system is far greater than it appears from the driver's seat [3].

6.4 Reflection on the Physics-Based Approach

One of the central motivations of this project was the belief that Adaptive Cruise Control should be expressible as a clean physical system rather than a collection of heuristic rules. Through simulation and control analysis, this belief proved justified.

The gap regulation problem naturally forms a second-order dynamic system. It behaves like a mass–spring–damper model. Proportional control introduces restoring behavior, and derivative control introduces damping. Stability depends on proper gain selection. These relationships are not arbitrary; they arise directly from Newtonian mechanics [8, 16].

Although commercial systems may incorporate extensive rule sets for safety and redundancy, the underlying behavior can be captured by relatively simple feedback principles. This reinforces the idea that physical intuition and control theory provide powerful tools for system design [5].

6.5 Physics Insight

A key result of this work is that Adaptive Cruise Control behavior can be directly understood through classical second-order system dynamics. The interaction between vehicle inertia, aerodynamic drag, and control input produces behavior analogous to a mass–spring–damper system. This framework explains observed phenomena such as oscillation, overshoot, and the necessity of proper damping for stable operation. Importantly, these behaviors emerge naturally from the governing equations and do not require heuristic interpretation.

6.6 Future Work

Future work should focus on improving sensor robustness, incorporating more reliable sensing modalities such as radar, and implementing real-time filtering techniques to mitigate measurement noise. Integration with vehicle actuation systems capable of direct throttle and braking control would also significantly improve system performance and allow full closed-loop validation.

The immediate next step is integration of a replacement distance sensor and final testing of the combined sensing and control system in the vehicle. With reliable hardware, the control code and sensor framework can be merged and evaluated under controlled driving conditions.

Beyond basic functionality, future improvements could include:

- Sensor fusion with vehicle OBDII speed data
- More advanced filtering to reduce speed estimation noise
- Direct cruise control actuation rather than advisory output
- Expanded modeling of braking authority and actuator limits
- Robustness testing under varying environmental conditions

With the foundation already established, these improvements are incremental rather than conceptual leaps [7].

6.7 Final Thoughts

This project transformed a teenage frustration into a rigorous study of sensing, control, and embedded systems. I began by believing that Adaptive Cruise Control was

simply a clever idea that had already been implemented by industry. I now understand that it is a carefully engineered interaction between physics, computation, and hardware [2, 16].

Perhaps most importantly, this experience reshaped my understanding of applied physics. I now appreciate why automotive systems require years of validation and why development costs are high. The difference between a functioning prototype and a production-ready system is enormous [12].

Even with setbacks, this project succeeded in its most important objective: it replaced curiosity with understanding. It demonstrated that Adaptive Cruise Control is not magic, nor is it merely a collection of rules. It is a dynamic physical system governed by measurable quantities and controllable parameters.

And it all started with repeatedly pressing the cruise control buttons on a highway in Utah.

Bibliography

[1] A. Corli and H. Fan, "String stability in traffic flows," Applied Mathematics and Computation **443,** 127775 (2023).

[2] S. Thrun *et al.*, "Stanley: The Robot that Won the DARPA Grand Challenge," Journal of Field Robotics **23,** 661–692 (2006), available at https://robots.stanford.edu/papers/thrun.stanley05.pdf.

[3] S. Shladover, "Impacts of Cooperative Adaptive Cruise Control on Traffic Flow," Transportation Research Record (2012).

[4] D. Helbing, "Traffic and Related Self-Driven Many-Particle Systems," Reviews of Modern Physics (2001).

[5] R. Rajamani, "Adaptive Cruise Control: A Tutorial and Survey," IEEE Control Systems Magazine **21,** 44–52 (2001).

[6] S. Hasirlioglu *et al.*, "Automotive Radar Sensor Models," IEEE Sensors Journal (2020).

[7] R. C. Luo and C.-C. Chang, "Multisensor Fusion and Integration for Intelligent Vehicles," IEEE Transactions on Industrial Informatics **15,** 5104–5116 (2019).

[8] D. Swaroop and J. Hedrick, "String Stability of Interconnected Systems," IEEE Transactions on Automatic Control **41,** 349–357 (1996).

[9] P. Gipps, "A Behavioural Car-Following Model," Transportation Research Part B **15,** 105–111 (1981).

[10] M. Treiber, A. Hennecke, and D. Helbing, "Congested Traffic States in Empirical Observations and Microscopic Simulations," Physical Review E **62,** 1805–1824 (2000).

[11] Unknown, "Automated Driving System Perception and Control Methods," arXiv preprint arXiv:1910.06070 (2019), available at https://arxiv.org/pdf/1910.06070.

[12] International Organization for Standardization, "ISO 15622: Adaptive Cruise Control Systems," Technical Report No. ISO 15622:2018, ISO, (2018) , international standard describing ACC system requirements.

[13] National Highway Traffic Safety Administration, "Advanced Driver Assistance Systems Research Report," Technical report, NHTSA, (2023) , nHTSA research summary on ADAS including ACC.

[14] D. de Waard *et al.*, "Driver Trust in Adaptive Cruise Control," Transportation Research Part F (2014).

[15] A. Eriksson and N. Stanton, "Effects of Cognitive Load on Takeover Performance in Automated Driving," Human Factors **59,** 689–705 (2017).

[16] R. Rajamani, *Vehicle Dynamics and Control* (Springer, 2011).

[17] Z. Li, P. Zhang, S. Liang, H. Zhou, C. Ma, H. Yao, and Q. Li, "Benchmarking Tesla's Traffic Light and Stop Sign Control: Field Dataset and Behavior Insights," arXiv preprint arXiv:2512.11802 (2025), available at https://arxiv.org/pdf/2512.11802.

[18] National Highway Traffic Safety Administration, "Adaptive Vehicle Control and Forward Collision Avoidance Systems," In *Proceedings of the 27th International Technical Conference on the Enhanced Safety of Vehicles (ESV)*, (2019), available at https://www-nrd.nhtsa.dot.gov/pdf/ESV/Proceedings/27/Session%209%20Written.pdf.

[19] National Highway Traffic Safety Administration, "Evaluating Adaptive Cruise Control and Interface Requirements for ADS," Technical Report No. DOT HS 812 172, NHTSA, (2018) , report. Available at https://www.nhtsa.gov/sites/nhtsa.gov/files/812172-evaluadaptvcruisecontrlintrfcrequiremtnads.pdf.

[20] Ford Motor Company, "BlueCruise: Ford's Hands-Free Driving Technology,", Web page, 2025, available at https://www.ford.com/technology/bluecruise/.

[21] National Highway Traffic Safety Administration, "INOA-EA25001-10004: NHTSA Investigation Documentation," Technical Report No. INOA-EA25001-10004, NHTSA, (2025) , investigation documentation. Available at https://static.nhtsa.gov/odi/inv/2025/INOA-EA25001-10004.pdf.

[22] Unknown, Master's thesis, The Ohio State University, 2021, master's thesis. Available at https://etd.ohiolink.edu/acprod/odb_etd/ws/send_file/send?accession=osu1626889287593463&disposition=inline.

[23] S. Kim, Ph.D. thesis, University of California, 2012, ph.D. dissertation. Available at https://escholarship.org/content/qt0v5399z9/qt0v5399z9_noSplash_2f5891530bb4827ca159534aa8c9479c.pdf.

[24] X. Li *et al.*, "Field Evaluation of ACC Systems," IEEE ITS (2018).

Appendix A

The Final Raspberry Pi Code

```
#!/usr/bin/env python3
"""
PTF-800 GUI speedometer + distance gauge + cruise-control helper arrows
- Assumes you are going 75 mph
- Uses a 3-second following distance as target
- Tracks how many DOWN clicks were made (each = 1 mph)
- When the car ahead leaves your lane, it tells you to click UP the same
    number of times
- Shows: "Click DOWN X times" or "Click UP X times"
"""

import argparse
import math
import os
import queue
import sys
import threading
import time
from collections import deque
```

```
from statistics import median

import serial
import tkinter as tk

# ============================================================
# PTF-800 protocol parsing
# ============================================================

def crc8_sum(frame_bytes):
    return sum(frame_bytes) & 0xFF

def build_start_cmd(continuous=True, brd_id=0xFF):
    mea_type = 0x0001
    mea_times = 0x0000
    b = [
        0xFA, 0x01, brd_id, 0x04,
        mea_type & 0xFF, (mea_type >> 8) & 0xFF,
        mea_times & 0xFF, (mea_times >> 8) & 0xFF
    ]
    b.append(crc8_sum(b))
    return bytes(b)

def build_stop_cmd(brd_id=0xFF):
    mea_type = 0x0000
    mea_times = 0x0000
    b = [
        0xFA, 0x01, brd_id, 0x04,
        mea_type & 0xFF, (mea_type >> 8) & 0xFF,
        mea_times & 0xFF, (mea_times >> 8) & 0xFF
    ]
```

```
    b.append(crc8_sum(b))
    return bytes(b)

def try_parse_measurement(buf):
    start = None
    for i in range(len(buf) - 1):
        if buf[i] == 0xFB and buf[i + 1] == 0x03:
            start = i
            break
    if start is None:
        return (max(0, len(buf) - 16), None)
    if len(buf) - start < 9:
        return (start, None)

    frame = buf[start:start + 9]
    if frame[3] != 0x04:
        return (start + 2, None)
    if crc8_sum(frame[:-1]) != frame[-1]:
        return (start + 2, None)

    data_valid = frame[4] | (frame[5] << 8)
    distance_dm = frame[6] | (frame[7] << 8)

    result = {
        "valid": (data_valid == 1),
        "distance_dm": distance_dm
    }
    return (start + 9, result)

# ============================================================
# Filtering helpers
```

```
# ============================================================

class SmoothValue:
    def __init__(self, alpha=0.35, n_med=5, max_delta_per_s=float("inf")
        ):
        self.alpha = alpha
        self.window = deque(maxlen=n_med)
        self.state = None
        self.max_delta_per_s = max_delta_per_s
        self.last_time = None

    def update(self, value, now=None):
        if value is None or (isinstance(value, float) and math.isnan(
            value)):
            return self.state
        if now is None:
            now = time.time()

        self.window.append(value)
        med = median(self.window)

        if self.state is None:
            self.state = med
            self.last_time = now
            return self.state

        dt = max(1e-3, now - (self.last_time or now))
        max_delta = self.max_delta_per_s * dt if math.isfinite(self.
            max_delta_per_s) else float("inf")
        target = max(self.state - max_delta, min(self.state + max_delta,
            med))
```

```
        self.state = self.alpha * target + (1 - self.alpha) * self.state
        self.last_time = now
        return self.state

# ============================================================
# Serial reader thread
# ============================================================

class PTFReader(threading.Thread):
    def __init__(self, port, baud, out_q):
        super().__init__(daemon=True)
        self.port = port
        self.baud = baud
        self.out_q = out_q
        self._stop = threading.Event()
        self.ser = None

    def run(self):
        try:
            self.ser = serial.Serial(
                port=self.port,
                baudrate=self.baud,
                bytesize=serial.EIGHTBITS,
                parity=serial.PARITY_NONE,
                stopbits=serial.STOPBITS_ONE,
                timeout=0.05
            )
        except Exception as e:
            print("Serial open error:", e, file=sys.stderr)
            return
```

```
self.ser.reset_input_buffer()
self.ser.reset_output_buffer()

try:
    self.ser.write(build_start_cmd(True, 0xFF))
    self.ser.flush()
except Exception:
    pass

buf = bytearray()
while not self._stop.is_set():
    try:
        data = self.ser.read(64)
        if data:
            buf.extend(data)
            while True:
                consumed, result = try_parse_measurement(buf)
                if consumed:
                    del buf[:consumed]
                if result is None:
                    break
                if result["valid"]:
                    meters = result["distance_dm"] / 10.0
                    self.out_q.put(("dist", meters, time.time())
                        )
        time.sleep(0.003)
    except Exception as e:
        print("Serial read error:", e, file=sys.stderr)
        time.sleep(0.25)
```

```
        try:
            self.ser.write(build_stop_cmd(0xFF))
            self.ser.flush()
        except Exception:
            pass
        try:
            self.ser.close()
        except Exception:
            pass

    def stop(self):
        self._stop.set()

# ============================================================
# Round gauge widget
# ============================================================

class RoundGauge:
    def __init__(self, canvas, x, y, r, label, unit,
                 min_val, max_val, start_deg=135, end_deg=45):
        self.c = canvas
        self.x, self.y, self.r = x, y, r
        self.label = label
        self.unit = unit
        self.min_val = min_val
        self.max_val = max_val
        self.start_deg = start_deg
        self.end_deg = end_deg
        self.text_id = None
        self.needle_id = None
        self._draw_static()
```

```
    def _draw_static(self):
        self.c.create_oval(self.x - self.r, self.y - self.r,
                           self.x + self.r, self.y + self.r, width=2)

        sweep = (360 - (self.start_deg - self.end_deg)) % 360
        self.c.create_arc(self.x - self.r + 6, self.y - self.r + 6,
                          self.x + self.r - 6, self.y + self.r - 6,
                          start=self.start_deg,
                          extent=sweep,
                          style="arc", width=14)

        for frac in [0, 0.25, 0.5, 0.75, 1]:
            ang = math.radians(self.start_deg - frac * sweep)
            x0 = self.x + (self.r - 22) * math.cos(ang)
            y0 = self.y - (self.r - 22) * math.sin(ang)
            x1 = self.x + (self.r - 6) * math.cos(ang)
            y1 = self.y - (self.r - 6) * math.sin(ang)
            self.c.create_line(x0, y0, x1, y1, width=3)

        self.c.create_text(self.x, self.y + self.r * 0.55,
                           text=self.label, font=("Helvetica", 14, "bold
                               "))

        self.text_id = self.c.create_text(self.x, self.y + self.r *
            0.25,
                                          text="    ", font=("Helvetica",
                                              20, "bold"))

        self.needle_id = self.c.create_line(self.x, self.y,
                                            self.x, self.y - (self.r -
```

```
                                                    26),
                                                width=4, arrow=tk.LAST)

    def set_value(self, v):
        txt = f"{v:.1f}°{self.unit}" if v is not None and math.isfinite(
            v) else "   "
        self.c.itemconfigure(self.text_id, text=txt)

        if v is None or not math.isfinite(v):
            v = self.min_val

        sweep = (360 - (self.start_deg - self.end_deg)) % 360
        frac = (v - self.min_val) / (self.max_val - self.min_val)
        ang = math.radians(self.start_deg - frac * sweep)

        x1 = self.x + (self.r - 26) * math.cos(ang)
        y1 = self.y - (self.r - 26) * math.sin(ang)
        self.c.coords(self.needle_id, self.x, self.y, x1, y1)

# ============================================================
# Main GUI App
# ============================================================

class App:
    def __init__(self, args):
        self.args = args
        self.root = tk.Tk()
        self.root.title("PTF-800 Speed & Distance")

        self.canvas = tk.Canvas(self.root, width=800, height=450)
        self.canvas.grid(row=0, column=0, columnspan=3, padx=10, pady
```

```
    =10)

# Gauge: Distance
self.g_dist = RoundGauge(self.canvas, 220, 200, 160,
                         "Distance", "m",
                         min_val=0.0, max_val=args.max_dist)

# Gauge: Speed (relative closure speed)
speed_unit = "mph" if args.mph else "km/h"
if args.mph and args.max_speed < 80:
    args.max_speed = 80

self.g_speed = RoundGauge(self.canvas, 580, 200, 160,
                          "Speed", speed_unit,
                          min_val=0.0, max_val=args.max_speed)

# Arrows between gauges
mid_x = (220 + 580) / 2.0
self.up_arrow_id = self.canvas.create_text(
    mid_x, 120, text="    ",
    font=("Helvetica", 40, "bold"),
    fill="gray"
)
self.down_arrow_id = self.canvas.create_text(
    mid_x, 240, text="    ",
    font=("Helvetica", 40, "bold"),
    fill="gray"
)

# Text box showing how many times to click up/down
self.click_text_id = self.canvas.create_text(
```

```
        mid_x, 320,
        text="", font=("Helvetica", 20, "bold"),
        fill="white"
    )

    # === Cruise logic variables ===
    self.follow_time = 3.0  # seconds
    assumed_speed_mph = 75.0
    assumed_speed_mps = assumed_speed_mph / 2.23693629
    self.target_distance_m = self.follow_time * assumed_speed_mps  #
        ~100.6 m
    self.dist_tolerance = 5.0  # good zone  5m

    # Counting clicks
    self.clicks_down = 0        # how many mph down you tapped
    self.clicks_up_remaining = 0
    self.recovering = False     # whether we are recovering (clicking
        up)

    # Buttons
    self.btn_exit = tk.Button(self.root, text="Exit", font=("
        Helvetica", 14, "bold"),
                              width=10, command=self.on_exit)
    self.btn_exit.grid(row=1, column=0, pady=10)

    self.btn_shutdown = tk.Button(self.root, text="Shutdown", font=(
        "Helvetica", 14, "bold"),
                                  width=12, command=self.on_shutdown
                                      )
    self.btn_shutdown.grid(row=1, column=1, pady=10)
```

```
        self.btn_clear = tk.Button(self.root, text="Clear spikes", font
            =("Helvetica", 12),
                                   command=self.clear_filters)
        self.btn_clear.grid(row=1, column=2, pady=10)

        # Serial thread
        self.q = queue.Queue()
        self.reader = PTFReader(args.port, args.baud, self.q)
        self.reader.start()

        # Filters
        self.dist_filter = SmoothValue(alpha=0.35, n_med=5,
            max_delta_per_s=args.max_dist_slew)
        self.speed_filter = SmoothValue(alpha=0.35, n_med=5,
            max_delta_per_s=args.max_speed_slew)

        self.last_dist = None
        self.last_t = None

        self.root.after(20, self.poll_queue)
        self.root.protocol("WM_DELETE_WINDOW", self.on_exit)

    # ------------------------------------------------------------------

    def clear_filters(self):
        self.dist_filter = SmoothValue(alpha=0.35, n_med=5,
            max_delta_per_s=self.args.max_dist_slew)
        self.speed_filter = SmoothValue(alpha=0.35, n_med=5,
            max_delta_per_s=self.args.max_speed_slew)
        self.last_dist = None
        self.last_t = None
```

```
        self.recovering = False
        self.clicks_down = 0
        self.clicks_up_remaining = 0
        self.canvas.itemconfigure(self.up_arrow_id, fill="gray")
        self.canvas.itemconfigure(self.down_arrow_id, fill="gray")
        self.canvas.itemconfigure(self.click_text_id, text="")

    # ------------------------------------------------------------

    def on_shutdown(self):
        self.on_exit(clean_only=True)
        os.system("sudo shutdown -h now")
        self.root.after(1500, self.root.destroy)

    # ------------------------------------------------------------

    def on_exit(self, clean_only=False):
        try:
            if self.reader.is_alive():
                self.reader.stop()
                self.reader.join(timeout=1)
        except Exception:
            pass
        if not clean_only:
            self.root.destroy()

    # ------------------------------------------------------------

    def poll_queue(self):
        got = False
        while True:
```

```
            try:
                tag, val, tstamp = self.q.get_nowait()
                got = True
                if tag == "dist":
                    self.update_values(val, tstamp)
            except queue.Empty:
                break

        if not got:
            self.update_values(None, time.time())

        self.root.after(50, self.poll_queue)

    # ------------------------------------------------------------------

    def update_arrows(self, dist_clamped):
        """
--------Logic:
--------- If too close: red DOWN arrow, count a down click
--------- If suddenly far: begin recovery (up clicks)
--------- During recovery: green UP arrow until all clicks restored
--------- Else: no arrows
--------"""

        if dist_clamped is None or not math.isfinite(dist_clamped):
            self.canvas.itemconfigure(self.up_arrow_id, fill="gray")
            self.canvas.itemconfigure(self.down_arrow_id, fill="gray")
            self.canvas.itemconfigure(self.click_text_id, text="")
            return

        error = dist_clamped - self.target_distance_m
```

```
# Too close      need to tap cruise DOWN
if error < -self.dist_tolerance and not self.recovering:
    self.clicks_down += 1
    self.canvas.itemconfigure(self.down_arrow_id, fill="red")
    self.canvas.itemconfigure(self.up_arrow_id, fill="gray")
    self.canvas.itemconfigure(self.click_text_id,
                              text=f"Click DOWN {self.
                                  clicks_down} times")
    return

# Car likely left your lane      sudden big gap
if error > self.dist_tolerance * 4 and self.clicks_down > 0 and
    not self.recovering:
    # Begin recovery phase
    self.recovering = True
    self.clicks_up_remaining = self.clicks_down

# Recovery mode      click UP same # of times we clicked down
if self.recovering:
    if self.clicks_up_remaining > 0:
        self.canvas.itemconfigure(self.up_arrow_id, fill="green"
            )
        self.canvas.itemconfigure(self.down_arrow_id, fill="gray
            ")
        self.canvas.itemconfigure(self.click_text_id,
                                  text=f"Click UP {self.
                                      clicks_up_remaining} times
                                      ")
        self.clicks_up_remaining -= 1
        return
```

```
            else:
                # Finished recovery
                self.recovering = False
                self.clicks_down = 0
                self.canvas.itemconfigure(self.up_arrow_id, fill="gray")
                self.canvas.itemconfigure(self.down_arrow_id, fill="gray
                    ")
                self.canvas.itemconfigure(self.click_text_id, text="")
                return

        # Good zone
        self.canvas.itemconfigure(self.up_arrow_id, fill="gray")
        self.canvas.itemconfigure(self.down_arrow_id, fill="gray")
        if self.clicks_down > 0:
            self.canvas.itemconfigure(self.click_text_id,
                                      text=f"Click DOWN {self.
                                          clicks_down} times")
        else:
            self.canvas.itemconfigure(self.click_text_id, text="")

# ----------------------------------------------------------------------

def update_values(self, distance_m, tstamp):
    dist_sm = self.dist_filter.update(distance_m, now=tstamp) if
        distance_m is not None else self.dist_filter.state

    # Compute relative velocity
    v_ms_raw = None
    if distance_m is not None:
        if self.last_dist is not None and self.last_t is not None:
            dt = max(1e-3, tstamp - self.last_t)
```

```
                v_ms_raw = (distance_m - self.last_dist) / dt
            self.last_dist = distance_m
            self.last_t = tstamp

        # Convert for display
        if v_ms_raw is not None:
            if self.args.mph:
                scale = 2.23693629
            else:
                scale = 3.6
            v_unit = abs(v_ms_raw * scale)
        else:
            v_unit = None

        v_smooth = self.speed_filter.update(v_unit, now=tstamp) if
            v_unit is not None else self.speed_filter.state

        dist_clamped = max(0.0, min(self.args.max_dist, dist_sm)) if
            dist_sm is not None else None
        v_clamped = max(0.0, min(self.args.max_speed, v_smooth)) if
            v_smooth is not None else None

        self.g_dist.set_value(dist_clamped)
        self.g_speed.set_value(v_clamped)

        self.update_arrows(dist_clamped)

# ============================================================
# Args
# ============================================================
```

```
def parse_args():
    ap = argparse.ArgumentParser(description="PTF-800 GUI + cruise click
        counter")
    ap.add_argument("--port", default="/dev/ttyUSB0")
    ap.add_argument("--baud", type=int, default=115200)

    units = ap.add_mutually_exclusive_group()
    units.add_argument("--mph", action="store_true")
    units.add_argument("--kph", action="store_true")

    ap.add_argument("--max-dist", type=float, default=150.0)
    ap.add_argument("--max-speed", type=float, default=80.0)
    ap.add_argument("--max-dist-slew", type=float, default=30.0)
    ap.add_argument("--max-speed-slew", type=float, default=60.0)

    args = ap.parse_args()
    if not args.kph:
        args.mph = True
    return args

# ============================================================

def main():
    args = parse_args()
    app = App(args)
    try:
        app.root.mainloop()
    except KeyboardInterrupt:
        app.on_exit()

# ============================================================
```

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

Appendix B

The Simulation Code

This Appendix will include all simulation code.

B.1 Slow to a Speed

This is the very basic, slow until you reach the same speed as the car ahead, code.

```
import matplotlib.pyplot as plt

desired_gap = 100.0 # meters
lead_speed = 45.0 * 0.44704
initial_follow_speed = 55.0 * 0.44704
initial_gap = 75.0 # meters

accel = 0.8 # m/s^2

m = 4000 * 0.45359237 # lb to kg
t_end = 25.0 # run time
dt = 0.1
Cd = 0.2 # drag coefficient
A = 1.9 # frontal area
```

```
rho = 1.225  # air density
engine_brake_force = 400 # N Google estimation...
v0 = initial_follow_speed
# Simulation
v = v0
t = 0
history = []
if v < lead_speed:
    while v < lead_speed and t < t_end:
        # Acceleration (positive for acceleration)
        a = accel
        # Update speed
        v += a * dt
        t += dt
        history.append((t, v))
        #print(f"Time: {t:.1f}s, Speed: {v*2.23694:.2f} mph, Accel: {a
            :.2f} m/s  ")

elif v > lead_speed:
    while v > lead_speed and t < t_end:
        # Aerodynamic drag force
        F_drag = 0.5 * rho * Cd * A * v**2
        # Total decelerating force
        F_total = F_drag + engine_brake_force
        # Acceleration (negative for deceleration)
        a = -F_total / m
        # Update speed
        v += a * dt
        t += dt
        history.append((t, v))
```

```
        #print(f"Time: {t:.1f}s, Speed: {v*2.23694:.2f} mph, Decel: {a
            :.2f} m/s  ")
else:
    print("No speed change needed.")
# Convert to plotting arrays
times = [pt[0] for pt in history]
speeds_mph = [pt[1] * 2.23694 for pt in history]

plt.plot(times, speeds_mph)
plt.xlabel("Time (s)")
plt.ylabel("Speed (mph)")
plt.title("Speed vs Time")
plt.grid(True)
plt.xlim(0, t_end)
plt.ylim(0, initial_follow_speed * 2.23694)
plt.show()

print(f"Car + accelerated in {t:.1f} seconds.")
```

That was the simplest of the programs.

B.2 Slow to a Distance

Now, onto more exciting stuff!

```
import matplotlib.pyplot as plt

cruise_up_pressed = True  # simulate cruise control UP button

desired_gap = 100.0  # meters
lead_speed = 45.0 * 0.44704
initial_follow_speed = 55.0 * 0.44704
```

```
initial_gap = 75.0  # meters

accel = 0.8  # m/s^2

m = 4000 * 0.45359237  # lb to kg
t_end = 750.0  # run time
dt = 0.1

# Simple first-order smoothing on commanded acceleration (prioritize
    smoothness over optimization)
tau = 7.5                  # seconds (bigger = more damped/smoother)
alpha = dt / tau           # smoothing factor
a_applied = 0.0            # filtered acceleration state

# Used only when we intentionally command braking (kept simple on
    purpose)
Cd = 0.2  # drag coefficient
A = 1.9   # frontal area
rho = 1.225  # air density
engine_brake_force = 800  # N (rough estimate)

# Simulation state
v = initial_follow_speed
t = 0.0
x_follow = 0.0
x_lead = initial_gap
gap_deadband = 2.0  # meters

history = []

while t < t_end:
```

```
# Update positions and gap (always tracked, even in the simple model
    )
x_lead += lead_speed * dt
x_follow += v * dt
gap = x_lead - x_follow

# Default: no input -> constant speed (simplicity over realism)
a_cmd = 0.0

# If we're too close, command a simple braking acceleration based on
    drag + engine braking
if gap < desired_gap - gap_deadband:
    F_drag = 0.5 * rho * Cd * A * v**2
    a_cmd = -(F_drag + engine_brake_force) / m

# If we're far enough and "cruise up" is pressed, command a fixed
    acceleration
elif cruise_up_pressed and gap > desired_gap + gap_deadband:
    a_cmd = accel

# Smooth acceleration changes to avoid unrealistic step changes
a_applied += alpha * (a_cmd - a_applied)

# Apply smoothed acceleration and advance time
v += a_applied * dt
if v < 0:
    v = 0.0

t += dt
history.append((t, v, gap))
```

```
# Convert to plotting arrays
times = [pt[0] for pt in history]
speeds_mph = [pt[1] * 2.23694 for pt in history]
gaps = [pt[2] for pt in history]

plt.plot(times, speeds_mph)
plt.xlabel("Time (s)")
plt.ylabel("Speed (mph)")
plt.title("Speed vs Time (simple ACC logic)")
plt.grid(True)
plt.xlim(0, t_end)
plt.ylim(0, initial_follow_speed * 2.23694)
plt.show()

print(f"Simulation ran for {t:.1f} seconds.")
```

That gives wacky results.

B.3 Slow to a Distance Using the Proportional Control Law

First attempt at smoothing the graph.

```
import matplotlib.pyplot as plt

# ------------------------------
# Scenario (yours)
# ------------------------------
desired_gap = 100.0                 # m
lead_speed = 45.0 * 0.44704         # m/s
initial_follow_speed = 55.0 * 0.44704
```

```
initial_gap = 75.0                    # m

t_end = 200.0
dt = 0.1

# ----------------------------------
# Simple P-only ACC on gap error
#   a_cmd = Kp * (gap - desired_gap)
#
# This is a classic 2nd-order loop because:
#   state 1: gap (position-like)
#   state 2: ego speed (speed-like)
#
# The actuator lag (tau_a) makes it realistically underdamped
# without adding D-term.
# ----------------------------------
Kp = 0.015            # (m/s^2) per meter  -> tune this for more/less
    oscillation
a_max = 1.5           # m/s^2 accel limit
b_max = 2.5           # m/s^2 brake limit (positive number; we clamp to -
    b_max)

# 1st-order actuator (acceleration) lag: a_applied follows a_cmd
tau_a = 2.0           # seconds (bigger = more sluggish; too big can
    destabilize)
alpha = dt / tau_a
a_applied = 0.0

# ----------------------------------
# Simulation state
# ----------------------------------
```

```
v = initial_follow_speed
x_follow = 0.0
x_lead = initial_gap
t = 0.0

history = []

while t < t_end:
    # advance positions
    x_lead += lead_speed * dt
    x_follow += v * dt
    gap = x_lead - x_follow

    # proportional control on gap error
    e = gap - desired_gap
    a_cmd = Kp * e

    # clamp to simple accel/brake capabilities
    if a_cmd > a_max:
        a_cmd = a_max
    elif a_cmd < -b_max:
        a_cmd = -b_max

    # actuator smoothing / lag (this is what makes it nicely underdamped
        vs "bang-bang")
    a_applied += alpha * (a_cmd - a_applied)

    # apply acceleration to speed
    v += a_applied * dt
    if v < 0:
        v = 0.0
```

```
    t += dt
    history.append((t, v, gap, a_cmd, a_applied))

# ---------------------------------
# Plot
# ---------------------------------
times = [h[0] for h in history]
speeds_mph = [h[1] * 2.23694 for h in history]
gaps = [h[2] for h in history]

plt.figure()
plt.plot(times, speeds_mph)
plt.xlabel("Time (s)")
plt.ylabel("Speed (mph)")
plt.title("Ego speed (P-only gap control)")
plt.grid(True)
plt.show()

plt.figure()
plt.plot(times, gaps)
plt.axhline(desired_gap, linestyle="--")
plt.xlabel("Time (s)")
plt.ylabel("Gap (m)")
plt.title("Gap response (should be underdamped: overshoot + decay)")
plt.grid(True)
plt.show()

print(f"Final speed: {speeds_mph[-1]:.2f} mph")
print(f"Final gap: {gaps[-1]:.2f} m")
```

That still gives bad results.

B.4 Slow to a Distance Using Proportional-Derivative (PD) Control

```
import matplotlib.pyplot as plt

# ----------------------------------
# Scenario
# ----------------------------------
desired_gap = 100.0                    # m
lead_speed = 45.0 * 0.44704            # m/s
initial_follow_speed = 55.0 * 0.44704
initial_gap = 75.0                     # m

t_end = 200.0
dt = 0.1

# ----------------------------------
# PD gains (tune these)
# ----------------------------------
Kp = 0.015      # (m/s^2)/m   "stiffness"
Kd = 0.35       # 1/s         "damping" (since d_dot is m/s)

# Actuator / comfort limits
a_max = 1.5     # m/s^2
b_max = 2.5     # m/s^2 (max braking magnitude)

# Optional actuator lag (keep small; PD already provides damping)
tau_a = 0.4
alpha = dt / tau_a
a_applied = 0.0
```

```
# ----------------------------
# Simulation state
# ----------------------------
v = initial_follow_speed
x_follow = 0.0
x_lead = initial_gap
t = 0.0

history = []

while t < t_end:
    # advance positions
    x_lead += lead_speed * dt
    x_follow += v * dt
    gap = x_lead - x_follow

    # gap rate: d_dot = v_lead - v_ego
    d_dot = lead_speed - v

    # PD control law
    e = gap - desired_gap
    a_cmd = Kp * e + Kd * d_dot

    # clamp acceleration/braking
    if a_cmd > a_max:
        a_cmd = a_max
    elif a_cmd < -b_max:
        a_cmd = -b_max

    # smooth actuator (optional)
    a_applied += alpha * (a_cmd - a_applied)
```

```
    # apply to ego speed
    v += a_applied * dt
    if v < 0:
        v = 0.0

    t += dt
    history.append((t, v, gap, e, d_dot, a_cmd, a_applied))

# ------------------------------
# Plot
# ------------------------------
times = [h[0] for h in history]
speeds_mph = [h[1] * 2.23694 for h in history]
gaps = [h[2] for h in history]

plt.figure()
plt.plot(times, speeds_mph)
plt.xlabel("Time (s)")
plt.ylabel("Speed (mph)")
plt.title("Ego speed vs time (PD gap control)")
plt.grid(True)
plt.show()

plt.figure()
plt.plot(times, gaps)
plt.axhline(desired_gap, linestyle="--")
plt.xlabel("Time (s)")
plt.ylabel("Gap (m)")
plt.title("Gap vs time (PD gap control)")
plt.grid(True)
```

```
plt.show()


print(f"Final speed: {speeds_mph[-1]:.2f} mph")
print(f"Final gap: {gaps[-1]:.2f} m")
```

Appendix C

Message to JRT

Hey Margaret, I never received the email with the test files and sample code you mentioned before. Here is what I have been able to do so far. I've connected the sensor to a Raspberry Pi 4B and confirmed the following: The sensor powers on: the green LED lights up and the red laser is constantly on. The USB adapter is detected as /dev/ttyUSB0 using the CH341 driver. I've tested both 115200 and 9600 baud using PySerial and Minicom. I've sent the 0x55 sync byte and the full one-shot measure command (0xAA 00 00 20 00 01 00 00 21) — no response is ever received. I also tried the jrt_laser_distance_sensor Python library with both one-shot and continuous mode — the script hangs waiting for data. I attempted to pull up a suspected enable pad on the board to 3.3 V (leftmost of two gold pads near the 10-pin connector) — no change. The laser never blinks and the sensor never responds to serial commands. I've verified power (3.3 V), ground, TX/RX wiring (via your USB adapter), and correct port access. Everything appears functional except for the UART logic itself. Can you confirm whether this sensor is configured for UART (TTL), or if it may

require additional hardware initialization or firmware activation? Thank you!