"It had to be done"
Typeset using LaTeX default style in AASTeX631

IGVC Design Report — The "g-wagon"

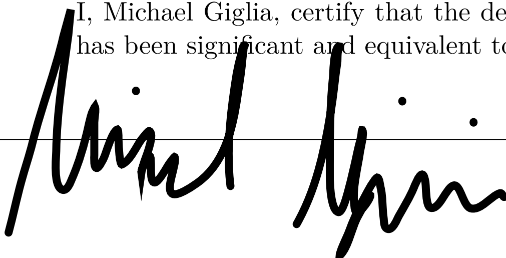
Andy Jaku <andy.jaku@cooper.edu>,[1] Anthony Kwon <anthony.kwon@cooper.edu>,[1]
Azra Rangwala <azra.rangwala@cooper.edu>,[1] Berry Xu <berry.xu@cooper.edu>,[1]
Crystal Huang <crystal.huang@cooper.edu>,[1] Daniel Park <d.park@cooper.edu>,[1]
Derrick Yu <derrick.yu@cooper.edu>,[1] Dylan Meyer-O'Connor <dylan.meyeroconnor@cooper.edu>,[1]
Fred Kim <fred.kim@cooper.edu>,[1] Giuseppe Quaratino <giuseppe.quaratino@cooper.edu>,[1]
Ilona Lameka <ilona.lameka@cooper.edu>,[1] Isaiah Rivera <isaiah.rivera@cooper.edu>,[1]
Jacob Koziej <jacob.koziej@cooper.edu>,[1,*] Jeannette Circe <circe@cooper.edu>,[1]
Kenneth Chan <kenneth.chan@cooper.edu>,[1] Lamiya Rangwala <lamiya.rangwala@cooper.edu>,[1]
Michael Giglia <michael.giglia@cooper.edu>,[1,†] Nathan Wolf-Sonkin <nathan.wolfsonkin@cooper.edu>,[1]
Noam Schuck <noam.schuck@cooper.edu>,[1] Paddy Yang <paddy.yang@cooper.edu>,[1]
Ridwan Hussain <ridwan.hussain@cooper.edu>,[1] Ridwan Hussain <ridwan.hussain@cooper.edu>,[1]
Siann Han <siann.han@cooper.edu>,[1] and Vaibhav Hariani <vaibhav.hariani@cooper.edu>[1]

[1] *The Cooper Union for the Advancement of Science and Art*

ABSTRACT

The Cooper IGVC Team is the primary research group of Cooper Union's Autonomy Lab. As an organization, we're dedicated to cultivating autonomous vehicle research at our institution. We also work to uplift the Cooper community through the mentorship of underclassmen with the development of workshops to cultivate skills generalizable to complex projects outside of IGVC. We pride ourselves on building our car full-stack. Students in Autonomy Lab have designed all core mechanical, electronic, and software systems, all of which are open source.

I, Michael Giglia, certify that the design and engineering of the vehicle by the current student team has been significant and equivalent to what might be awarded in a senior design course:



1. BACKGROUND

1.1. *Organization*

The team follows a vertically integrated structure, divided into three disciplines consisting of three to five student groups, each led by a subteam lead who facilitates weekly meetings. The subteams correspond to abstract systems on the car: Hardware (Electrical + Mechanical), Firmware, and Algorithms. In addition to weekly subteam meetings, we hold weekly team meetings to keep everyone in the loop of new developments and bi-weekly sprint meetings to establish tasks for our bi-weekly test days.

The hardware subteam designs and assembles all expansion hats for the Cooper Common Microcontroller Nodes (CCMNs), a custom `#coopermade` breakout board around the ESP32S3 microcontroller, to facilitate the needs of the Firmware subteam. In addition to PCBs, the subteam is responsible for maintaining the power distribution system of the car and designing mechanical components to retrofit our vehicle.

Our Firmware subteam writes all our safety critical actuation code and custom tooling to supplement software development. They aim to write modular firmware, allowing for the fast extension of CCMNs attached to hardware hats to specific hardware applications.

The algorithms subteam handles the perception and navigation of our car. They create & tune controllers, detect objects, perform localization based on camera and encoder data, and generate paths given all our component outputs.

1.2. *Design Assumptions & Design Process*

Our design process revolves around ensuring our systems are safe and easy to test in the cramped surroundings of the East Village in downtown Manhattan. We try to make our systems as modular as possible by breaking down every Autonomy Lab project into its various components, allowing for overlap between different projects we may take on. We host all our work in a publically available in a Git monorepo, which is licensed under copyleft licenses to ensure free access to our work to everyone. Maintaining a Git monorepo allows for easy collaboration on components, simplifies code review, and lets us accurately track changes to our work.

Due to Cooper's small size, it's easy for us to keep tabs on one another to ensure things get done, but outside of meetings, we also communicate using a Cooper-hosted instance of Matrix. We also believe that maintaining good documentation is essential to avoid amassing technical debt, so we keep a mix of public-facing documentation in our monorepo and institution-specific documentation that does not make sense to share outside Cooper.

* Team Captain
† Faculty Advisor

2. SYSTEM ARCHITECTURE

We like to think of our system as an hourglass. We aim to separate our hardware components as much as possible from the algorithms we write, where our car's firmware is the narrow bridge between these two worlds.

Major hardware components include:

- Our new Brake-By-Cable system
- Water-proof trunk for power distribution systems
- DC-DC converters
- Cooper Common Microcontroller Nodes
- A CAN Bus for node communication
- CCMN hats for our brakes, throttle, and encoders
- An ODrive for controlling our EPAS

Safety devices include:

- Hardware E-STOPs around the car and inside the cabin
- ATC and glass tube fuses
- Circuit breakers
- Undervoltage protection

Major software modules include:

- Shared firmware on all CCMNs
- OpenCAN for serialization/deserialization of CAN messages at cyclic rates
- Velocity and pressure controllers
- ROStouCAN for interfacing between nodes on our CAN Bus and our ROS nodes
- Lane detection algorithm
- Stanley controller for trajectory following
- A fine-tuned YOLO model for object detection
- State machines for state and event transitions based on the output of our object detection and lane detection

Even though we try to maintain our hourglass shape, it does not detract from the close collaboration between our subteams. For example, our Algorithms team needed more accurate braking on our vehicle this year, so they met with the hardware subteam to make this possible. It was then the responsibility of the hardware and firmware teams to work together to ensure our new brake system performed better without substantially changing how the system worked from the Algorithm perspective.

81 ## 3. INNOVATIONS

82 This year, we decided to redesign our braking system and move to a vision-based approach to path generation and
83 object detection.

84 Our previous braking system, based around a Hydrastar trailer brake system, was too powerful to allow consistent
85 braking at low speeds. Our new braking system, dubbed Brake-By-Cable, is our solution to add fine control over the
86 brakes. We installed a DC motor that rotates a pulley so that a cable pulls on the brake pedal. We utilize a pressure
87 and current sensor as feedback for our control loop to allow for accurate pressure control as requested by our vehicle
88 controller.

89 To determine what lane our car is in and to calculate our heading and cross-track errors, we've decided to use two
90 cameras placed on each A pillar of our vehicle. Since we can assume that our car operates on a flat plane, we measure
91 points along lane lines to create a trapezoidal unwrap of each video feed, allowing us to use perspective transforms to
92 make an image where distance is a linear function of pixels.

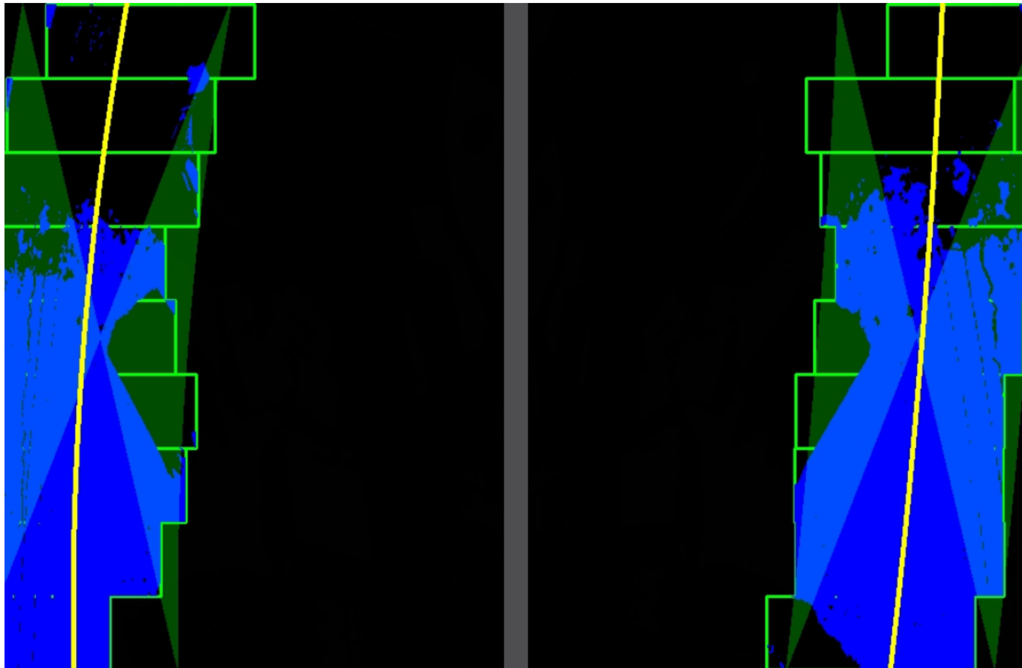


Figure 1. Fitted lane lines.

93 For our lane detection algorithm, we use a sliding window approach. We divide the video feed from the pillar cameras
94 into N windows, where we threshold all white pixels within each window. We then use the thresholded video feed to
95 generate a polynomial model of the lane line. This approach allows us to remove outliers and mitigate noise in our
96 input signals. Using these two polynomials, we can extract the cross-track error (the distance from the center of the
97 lane) by comparing the distance between points on the polynomial and the front axle. We can also extract the heading
98 error (the angle relative to the lane lines) by using a linear approximation of our generated polynomials at the front
99 axle and determining the deviation from the vertical axis of the video feed.

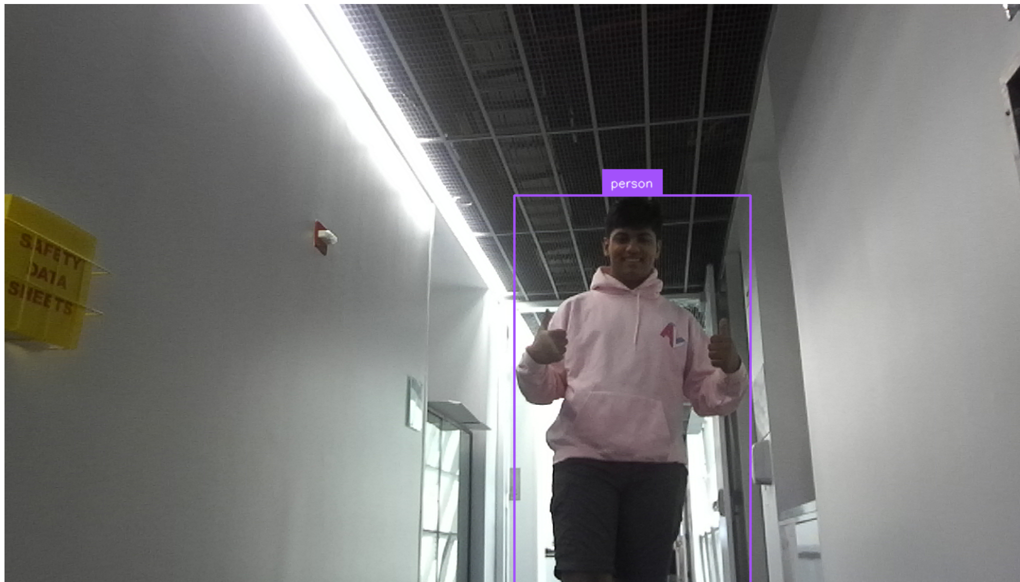


Figure 2. Object detection demo.

We've also gone away with using a LIDAR sensor for object detection and instead decided to use a deep learning model with a stereovision camera. We're using Ultralytics's YOLOv8 model for its state-of-the-art performance in real-time applications. We fine-tuned the model by freezing the first ten layers and training on a small dataset of common road obstructions and traffic signs.

104 # 4. DESIGN

105 ## 4.1. *Mechanical*

106 Our mechanical design work revolved around retrofitting our vehicle so that it's possible for it to drive itself. We
107 added a gas-spring emergency brake, a brake-by-cable system, spring-loaded rear encoders, and a water-proof trunk to
108 protect our power distribution system. In addition to this, we designed various mounts for sensors, microcontrollers,
109 and our car computer. We lean towards aluminum and steel for rigidity and machinability when designing mechanical
110 components to add to the car. We use 3D-printed components made from PETG filament for anything not load-bearing,
111 and when possible, we drill mounting holes into our car's frame to mount components.

112 ### 4.1.1. *Emergency Brake (Parking Brake)*

113 We mounted a gas spring rated for 20 lbs of force to our vehicle's parking brake and we hold it in the armed position
114 during operation using an electromagnet. On each end of the spring, we bolted aluminum extrusions between 3D-
115 printed spaces that are free to rotate. We machined brackets from 1/4" aluminum sheet metal to attach the spring to
116 the aluminum extrusions and hold any 3D-printed pieces to the car.

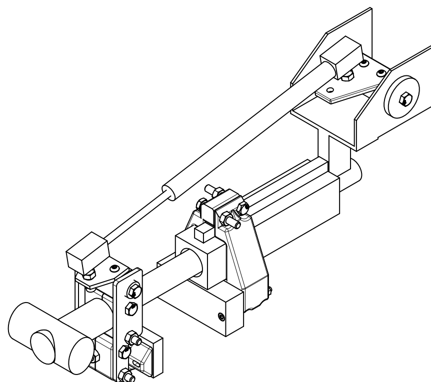


Figure 3. CAD drawing of the parking brake assembly in the open position.

117 ### 4.1.2. *Brake-by-Cable (BBC)*

118 We designed the Brake-By-Cable (BBC) system for fine control over pressure in the master brake cylinder to better
119 replicate a human driver's control over the brake pedal. We designed the system based on the constraint that it had to
120 apply 100 lbs of force to displace the brake pedal by two inches in less than a second. We designed the entire assembly
121 to fit in the small region between the master cylinder, differential, and steering arms.
122 We chose a cable rated for 150 lbs to add a factor of safety. To reduce our motor's torque requirements, we designed
123 a 3:1 lever and machined it from 1/4" steel. We connect the shorter side of the lever to the brake pedal and the
124 longer side to the DC motor. We machined a three-inch steel pulley to be within the diameter recommended by the
125 manufacturer of our cable. We also machined an aluminum shaft coupler to connect the top cable to our DC motor.
126 For the DC motor, we chose it such that the operating point of the system would be well below 20% of the motor's
127 stall torque. Additionally, we added an extension spring to the bottom of the lever to ensure that the cable stays in
128 tension when the motor releases the brake pedal.
129 We machined smaller pulleys from aluminum and used copper sleeves for the cable terminations on the lever and
130 brake pedal. Each termination freely rotates using bronze bushing and shoulder bolts fastened to the steel components.
131 The large backplate covering the entire range of motion of the lever is to ensure there are no pinch points in the system.
132 Each side of the lever also has an M6 shoulder bolt and a limit switch so that our firmware can detect the end of the
133 range of motion.

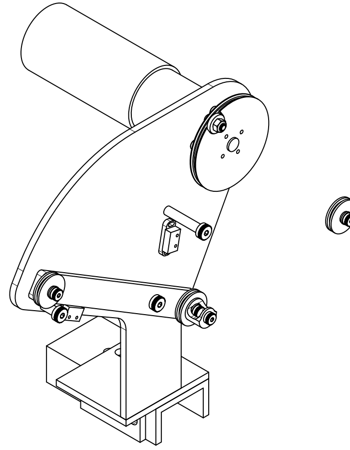


Figure 4. CAD drawing of the BBC.

134 4.1.3. *Rear Encoder Mounts*

135 Critical for calculating odometry and velocity for the vehicle, this year, we designed the rear encoders to have more
136 precision and less slip than the previous design. We designed and 3D printed a small wheel (1.6 inches diameter) that
137 directly attaches to the encoder shaft. This wheel touches the inside of the car's wheel so that we can measure the
138 tire's rotation. This geometry allows us to have over 30,000 counts per tire rotation. To ensure that the encoder wheel
139 does not slip and maintains contact with the tire at all times, we designed a spring-loaded bracket that connects with
140 the previous encoder mount's bracket. The rotation point is an M8 shoulder bolt with thrust bearings on each side of
141 the 1/8th-inch aluminum bracket to ensure low-friction rotation.

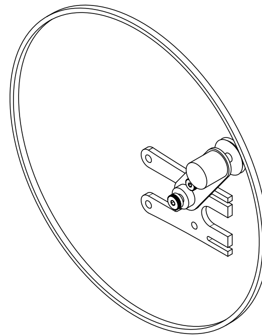


Figure 5. CAD drawing of the rear encoders.

142 4.1.4. *Weather Proofing*

143 To protect the power distribution system from weather, we designed and built a trunk that mounts to the back of
144 the vehicle above the car battery pack. We made the trunk's frame from 20mm aluminum extrusion, held together
145 with brackets made from a laser-cut 1/4" aluminum sheet. We laser cut the trunk walls from 1/8" cast acrylic and
146 fastened to the aluminum extrusions. For ease of access, we mounted the circuit breakers and under-voltage protection
147 boards to the acrylic on top of the trunk. We bolted the extrusion pieces to the chassis to ensure a secure mount and
148 used silicone caulk on the inside edges to ensure the enclosure is weatherproof.

4.2. *Electrical*

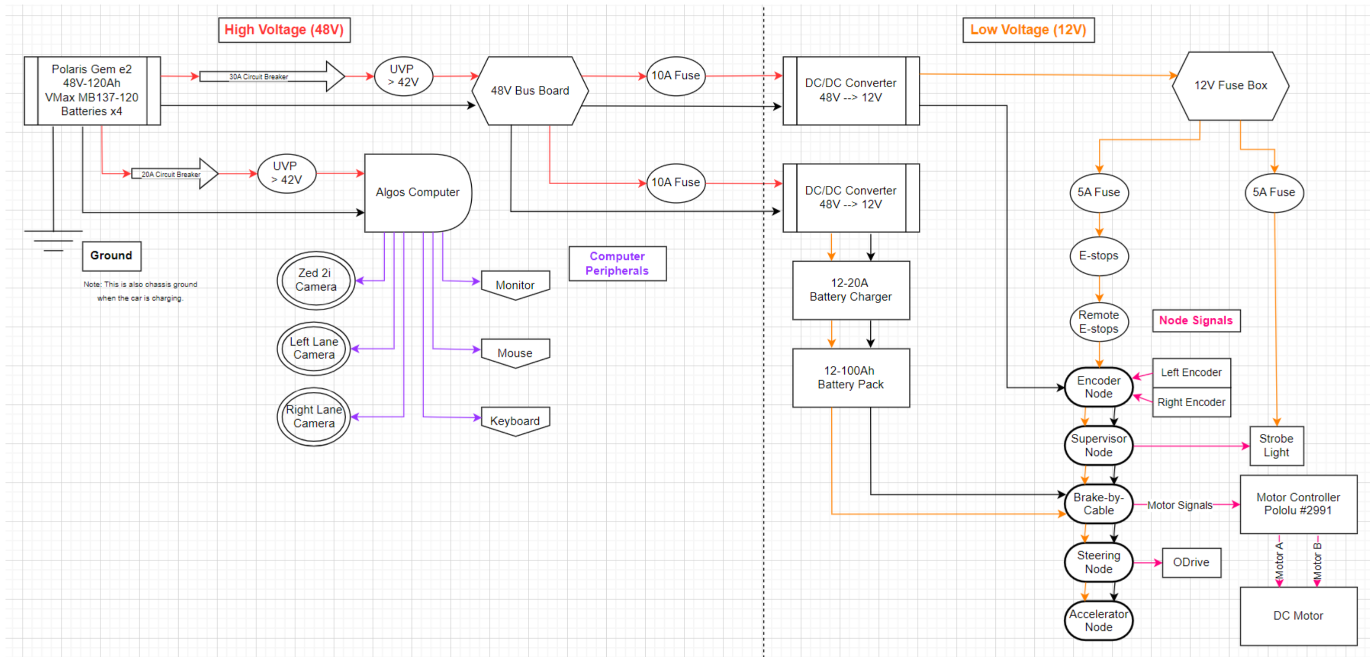


Figure 6. Electric system diagram.

We distribute power from the 48V DC pack through circuit breakers, fuses, under-voltage protection, and DC-to-DC converters to various sensors, CCMNs, and our car computer. We daisy-chain all CCMNs with Cat6A cables that supply 12V DC power and our car's CAN Bus. The boards have voltage regulators, buck converters, and e-fuses for safe operation and reverse polarity protection.

4.2.1. *Power Distribution System*

Four 12V DC, 120Ah AGM batteries connected in series power the car and the electronic stack above. This setup yields us 5.8kWh, which lets the car run for approximately six hours without needing to charge.

The DC motor controller (a Pololu G2 High-Power Motor Controller) draws a lot of in-rush current when activated to drive the DC motor. To prevent our DC motor controller from drawing too much current from the DC-to-DC converters (causing their overcurrent protection to trip), we've connected them to a battery charger, which charges a 12V battery connected to our DC motor controller. We've also added a fast-burn 7A fuse to the system to prevent damage if the motor stalls for an extended period of time.

4.2.2. *Electronics Suite*

We connect a CCMN to an actuator, defining its node identity by its expansion hat (if present). Each node interfaces with its actuator, listening to control inputs over CAN and broadcasting senor values of CAN. Webcams and our ZED2i are connected directly to our car computer.

- **Accelerator:** sends two voltage signals directly to the throttle through relays to allow manual overrides.
- **BBC:** interfaces with a DC motor via a motor controller to perform pressure control and filters the pressure sensor before feeding it to the built-in ADC of the ESP32S3.
- **Encoders:** gets encoder ticks through level shifters.
- **Protoboard:** a generic expansion board for prototyping new expansion boards.

4.2.3. *Safety Devices*

Safety devices fall into two categories: mechanical/electrical devices and software.

We use mechanical switches for e-stop buttons around the car to cut power to our low-voltage system, effectively disabling all autonomous aspects of our vehicle. Additionally, we've installed circuit breakers around the car so that all systems have to be explicitly enabled by a user, with the plus of adding over-current protection.

On the electrical side, we have ATC fuses in line with each device. We've also added under-voltage protection to our 48V battery pack to ensure its voltage does not fall below 40V and cause permanent damage. Finally, each CCMN has a Schottky diode e-fuse on its input to allow boards to survive accidental reverse input.

On the software side, we have a supervisor CCMN that ensures we never latch in our vehicle's autonomous mode. This node inspects CAN messages and, if all checks pass, can authorize other nodes on the CAN network for autonomous control. This authorization has a TTL on the scale of tens of milliseconds, meaning if any check fails, unauthorized components of the system will exit autonomous control.

4.3. *Software*

Our software falls into two categories: Firmware and Algorithms. Firmware creates the software interface between the hardware of the car, while Algorithms generates inputs that we feed into our Drive-By-Wire stack.

As an example, let's follow how a path generated by our Stanley controller interacts with the car. We publish this path as a ROS topic to another ROS node that interfaces with our CAN Bus. We then translate this CAN message on our velocity controller CCMN using a PID controller to a brake and throttle percentage. The throttle and BBC nodes then act on these inputs only if authorized by our supervisor.

4.3.1. *Extracting Items from the Current Scene*

We handle object detection through a combination of a real-time deep-learning model and an optical character recognition engine. We chose Tesseract's OCR engine for reading road signs and only run it on signs detected by our instance of the YOLOv8 model. We also use point-cloud data from our ZED2i camera to compute the distance to detected objects. Once an obstacle is detected, its location relative to the vehicle gets published as a ROS topic.

We use two web cameras mounted on the A-pillars of our car for lane detection. We use OpenCV to remove lens distortions and calculate a linear mapping between pixels and physical distances. From here, we threshold the image to detect white pixels and apply sliding windows to find lane-line positions.

To detect which lane the car sits in, we compare the number of windows with less than a determined number of white pixels from our sliding windows. By nature, dashed lines have more empty windows, meaning we can determine legal road maneuvers without a need for additional localization.

We achieve "sensor fusion" using the `ekf_localization_node` from the `robot_localization` ROS package. This node utilizes an extended Kalman filter to merge IMU and encoder data for better estimates of our car's position.

4.3.2. *World Frame Representation*

We use odometry data from the `ekf_localization_node` to obtain the position and orientation information of the vehicle in the world frame. The ZED2i's point cloud data measures the vertical and horizontal displacement of objects relative to the vehicle's location. We then add the object's displacement relative to the vehicle's location from its starting point to determine the object's location in the world frame. We then feed this information into a state machine for decision-making.

4.3.3. *Vehicle Operating Modes*

In self-driving mode, the vehicle operates in one of two states: lane keeping or land changing. A land change generates paths to avoid road obstructions by creating a clothoid path. We generate clothoid paths based on the car's current position, orientation, endpoint, and final orientation. Our vehicle switches operating modes based on the conditions present.

4.3.4. *Trajectory Generation*

In lane-change mode, we operate purely on generated clothoid paths. A lane change gets triggered by a detected object. Once detected, the clothoid path gets generated and followed to avoid a collision. Additionally, we generate trajectories to merge into different lanes or to cross intersections.

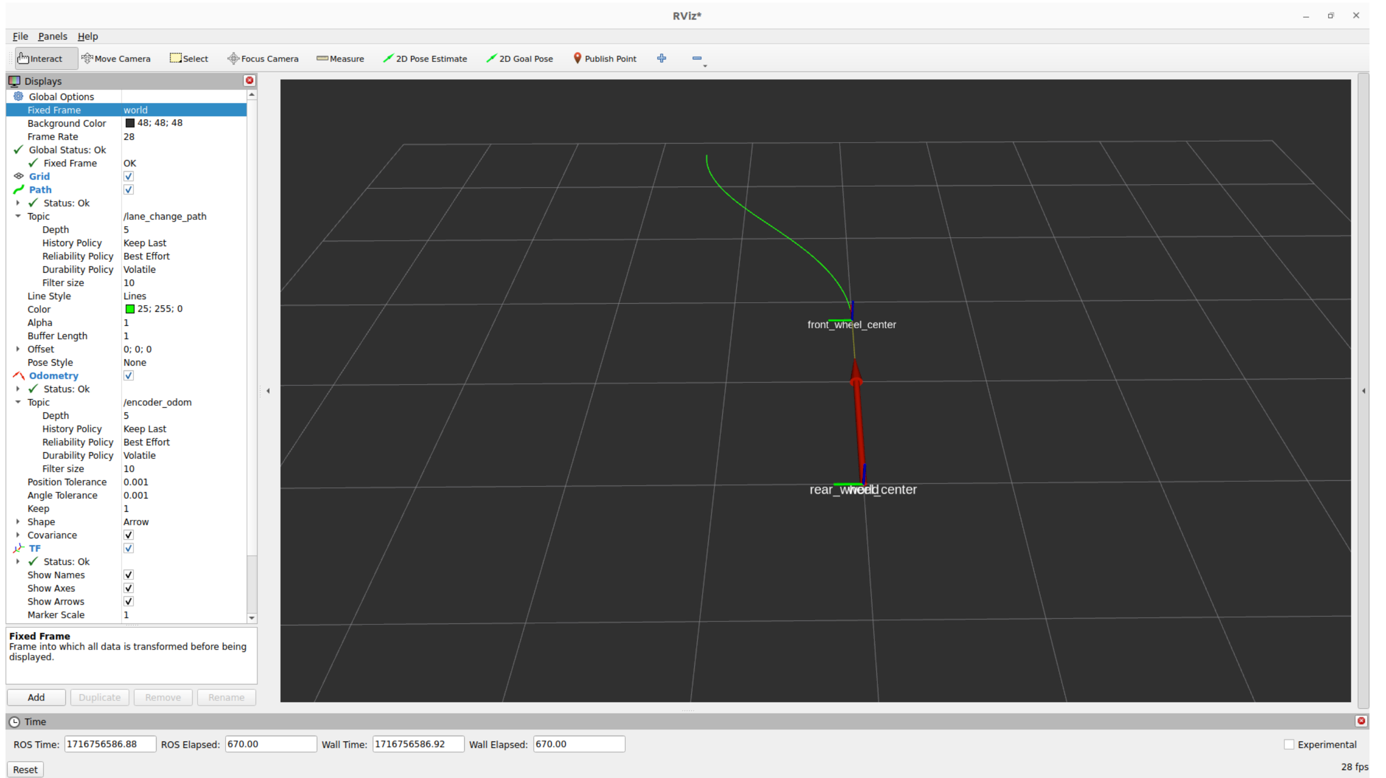


Figure 7. We use Rviz2 to model vehicle dynamics at the control level: here, we demonstrate a generated path and the car position calculated using odometry from our rear encoders.

4.3.5. *Trajectory Following Controls*

After researching trajectory-based controllers, and testing with smaller steering robots, we determined that using a simplified bicycle model and the Stanley geometric controller would suit our vehicle's needs for path following. The controller calculates a steering angle based on two errors: the heading error, the yaw compared to the desired heading of the path, and the cross-track error, the perpendicular distance from the center of an axle to the path). We calculate the steering angle using Equation 1. Where θ_e is the heading error, t_e is the cross-track error, v is the velocity, and k is the Stanley gain.

$$\delta = \theta_e + \tan^{-1}\left(\frac{kt_e}{v}\right) \quad (1)$$

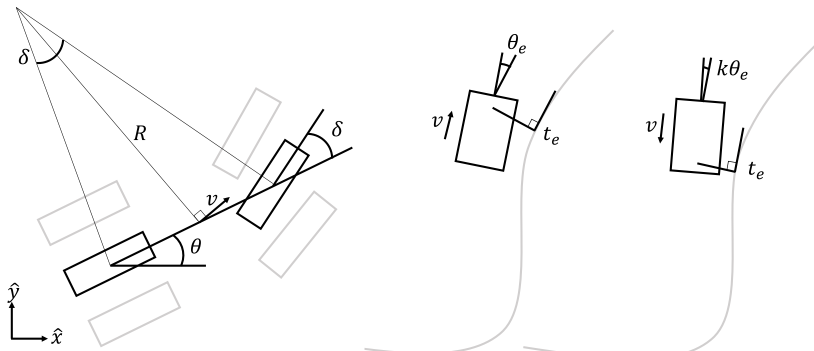


Figure 8. Kinematic bicycle model of a steering vehicle and the Stanley controller variables.

We have two methods of calculating the cross-track and heading errors for the Stanley controller. For following generated paths, such as a lane change, we use the encoders mounted to the rear tires to track the vehicle's current position and orientation. We calculate the Stanley errors based on the car's location and the nearest point of the path.

When following lane lines, when we have lines on both sides of the vehicle, we use the web cameras mounted to the top of the vehicle to detect the distance and angle of the lane lines relative to the car.

4.3.6. *Velocity Control*

On the Firmware side, once we receive a velocity command, the control node determines if the car needs to accelerate or decelerate based on the car's current velocity. We feed the error between the desired and actual velocities into a PID controller (the outer loop). If the car needs to accelerate, we use a mapping between the desired acceleration and throttle percentage and send the appropriate command to the throttle node. If the car needs to decelerate, we use another mapping between deceleration and brake pressure percentage and send that command to the brake node.

On the BBC node, we implemented another PID loop (the inner loop), which takes the error between the actual brake percentage (read by the pressure transducer) and the desired brake percentage to calculate the speed and direction of the BBC motor.

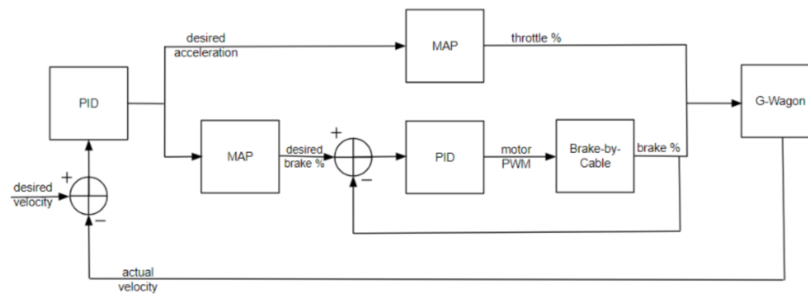


Figure 9. Block diagram of our velocity controller.

5. SYSTEM ANALYSIS

While testing things individually, it's easy to keep track of changes, but once integrated into our system, it's easy to lose track of changes and decisions. One example of this was during the development of the BBC PCB. We broke the board into four main blocks: motor controller logic, parking brake logic, pressure sensor data, and limit switches for the lever's range of motion. However, when testing the entire board, we found we would flip the polarity of certain pins, such as the input to the motor, or flip which connectors went to the minimum limit switch and the maximum limit switch.

The best way to prevent such mistakes is to create a more robust, generalized system rather than case-by-case. For example, we can ensure every wire has a proper visible label rather than memorizing its location. Additionally, we can write out a simple checklist of what to check before testing something. By creating a more robust system, it is easy to prevent many of the smaller, simpler mistakes.

5.1. *Critical Hardware Failure*

One hardware failure that would prevent competition success is if an ESP32S3 were to fail. The ESP32S3, a low-power microcontroller, limits the amount of current it can supply. We maintain an external 12V battery pack on the car because even though a CCMN can supply 12V, it's limited to tens of milliamps. We handle this by using isolated circuits to turn on/off the circuit i.e. using NMOS or PMOS transistors due to their high input resistance as a switch or using a relay to keep the two systems completely isolated.

5.2. *Software Development*

We store all code in a publically hosted monorepo on GitHub. We use Git as it allows us to effectively work in parallel, while the monorepo makes integration between various components straightforward. We utilize GitHub features such as issues and pull requests to keep track of problems with our code and encourage code review.

5.3. *Testing*

Before we test a particular algorithm or state machine on the car, and because our team is limited in physical space to which we can test the car, we test our ideas on smaller platforms, some of which we developed within Autonomy Lab. Carrie is our designed in-house mini-car robot platform for testing trajectory-following controllers. We tested and tuned the Stanley geometric controller for following clothoid paths, such as a complex path generated for parallel parking. We found the Stanley controller to be an effective method of calculating a steering angle for following a path. For testing with simplified dynamics, we used a differential-drive TurtleBot platform. To initially test and tune the OpenCV parameters for lane detection, we placed cameras on the Turtlebot and added a simple controller to follow the lane lines.

5.4. *Physical Testing to Date*

We tested the BBC system by cycling the motor between the two limit switches. The initial DC motor did not have enough torque and would stall before reaching the maximum displacement of the brake pedal. We sourced a more powerful DC motor with a higher stall torque, but this larger motor can pull up to 25 Amps. To ensure the motor does not pull too much current we installed a 7A fast-blow fuse on the battery supplying power to the motor.

When measuring the pressure sensor output from the car, the original data for the pressure sensor was from 0.7V to 2.0V. While this range of values is valid for the ADC readings to the ESP32S3 (within 0.15V to 2.45V), there wouldn't be enough precision to accurately calculate the pressure percentage. By using a circuit that combines a non-inverting OpAmp and a subtractor op amp, we simulated a circuit that linearly remaps the pressure sensor range from 0.7V to 2.0V into 0.236V to 2.302V. In real-life however, we saw that the actual voltage range was 0.153V to 2.102V. This clearly demonstrates an example of when an ideal simulation is different from a real-world implementation. The reason for this difference comes from non-ideal components, device variations, and parasitic capacitance and inductance throughout our circuit in the op amps and resistors.

We tuned the inner PID loop for BBC using a pressure transducer and recording the system's response time. We found that a high proportional and small integral gain provided a fast response time with no overshoot. When releasing the brake pedal before installing the spring, the top cable would become loose and often fall off the pulley, causing the cable to become plastically deformed. After installing the spring to keep the lever and top cable in tension with the motor and pulley, we found a great improvement in the response time and reliability of the system.

288 For braking, once the inner loop's response was well tuned and we could set and hold a desired braking percentage
289 within 1 second, we characterized the relationship between brake percent and deceleration. First, we throttled the car
290 to 10 mph, then set small increments of the desired brake percent to measure deceleration using the encoders.
291 We ran tests using small increments in throttle percent to characterize the acceleration-to-throttle percent relation-
292 ship. We measured the car's acceleration using the encoders mounted on the rear wheels.
293 To test the odometry for tracking position, we used Rviz2 to keep track of the car's orientation and position relative
294 to where it started in the world frame. Since we could not test the throttle indoors, we pushed the car while in neutral
295 with the maximum steering angle to travel in a circle to ensure the odometry returned to where we started.