ARV

University of Michigan - Ann Arbor

Submitted:	**May 15th, 2023**
Team Captain:	**Ashwin Saxena | ashwinsa@umich.edu**
Faculty Advisors:	**Xiaoxiao Du | xiaodu@umich.edu**
	Damen Provost | provostd@umich.edu
Statement of Integrity:	**Provided Separately**

Team Roster

Name	Email
Ashwin Saxena*	ashwinsa@umich.edu
Christian Foreman*	cjforema@umich.edu
Jose Diaz*	diazjose@umich.edu
Platform	
Kohei Nishiyama*	kohein@umich.edu
Drew Boughton**	drbought@umich.edu
Felicia Sang	fsang@umich.edu
Chloe Akombi	cjakombi@umich.edu
Simba Gao	simbagao@umich.edu
Cara Blashill	blashill@umich.edu
Aidan Deacon	agdeacon@umich.edu
Ryan Bird	ryanbird@umich.edu
Vance Kreider	vkreider@umich.edu
Brooke Kelsey	bckelsey@umich.edu
Megan Dzbanski	mdzbansk@umich.edu
Embedded Systems	
Aakash Bharat*	aakashvb@umich.edu
Julian Skifstad	juliansk@umich.edu
Joshua Ning**	joshning@umich.edu
Brinda Kapani	bkapani@umich.edu
Eric Barbieri	ericbarb@umich.edu
Sujit Lakshmikanth	sujlaks@umich.edu
Yuvraj Singh	uvsingh@umich.edu
Peter Susanto	psusanto@umich.edu
Grace Strom	gstrom@umich.edu
Liam Donegan	ldomegan@umich.edu
Katherine Shih	katshih@umich.edu
Layth Abdelkarim	laythabd@umich.edu
Business	
Alan Teng*	thtalan@umich.edu
Eric Qiao	ericqiao@umich.edu
Aaryan Chandola	aaryanc@umich.edu

Name	Email
Navigation	
Gannon Smith	gansmith@umich.edu
Chris Erndteman	chrisern@umich.edu
Emily Wu	emilyywu@umich.edu
Aarya Kulshrestha	akulshre@umich.edu
Krishna Dihora*	kdihora@umich.edu
Ryan Lee	ryalee@umich.edu
John Rose	johnrose@umich.edu
Maaz Hussain	maazh@umich.edu
Akhil Nair	aknair@umich.edu
Benjamin Rossano**	brossano@umich.edu
Computer Vision	
Alex de la Iglesia	alexdela@umich.edu
Connor Pang	pangc@umich.edu
Parsanna Koirala	parsanna@umich.edu
Lohit Kamatham	lohitk@umich.edu
Sydney Belt	sydbelt@umich.edu
Liyufei Meng	liyufeim@umich.edu
Awrod Haghi-Tabrizi**	ahaghita@umich.edu
Adi Balaji	advaithb@umich.edu
David Welch*	dswelch@umich.edu
Josh Nigrelli	jnigrell@umich.edu
Daniel (Xinzhou) He	xinzhouh@umich.edu
Tom Vu	tomvu@umich.edu
Waseem Alsayed	alsayed@umich.edu
Nihal Kurki	nkurki@umich.edu
Sensors	
Jason Ning*	zyning@umich.edu
Annie Li	anranli@umich.edu
Deric Dinu Daniel	dericdd@umich.edu
Chancellor Day	dchance@umich.edu
Kari Naga	knga@umich.edu

* Lead, ** Assistant Lead

Introduction

After a year-long development process, our team is delighted to introduce University of Michigan - Ann Arbor's Autonomous Robotic Vehicle Team's (ARV) new robot – **mARVin** for the 2023 Intelligent Ground Vehicle Competition. ARV is supported by both campus and corporate sponsors. The team is supported by the University of Michigan Robotics Institute. Our corporate sponsors include Ford, Bose, Ann Arbor SPARK, Aptiv, Siemens, Northrop Grumman, and Raytheon.

Team Organization

Our team is organized into six subteams: Business, Computer Vision, Navigation, Embedded Systems, Platform, and Sensors. The Business subteam handles sponsor relations, as well as the media/marketing side of the team. The Computer Vision (CV) subteam develops the lane and pothole detection system using CV techniques. The Navigation subteam develops the path planning, control systems, and simulation systems. The Embedded Systems subteam develops the electrical system for the robot; this includes motor control, safety features, the status indicator light, and the power delivery on the robot. The Platform subteam designs and builds the robot chassis to fit within the given design requirements. The Sensors subteam configures the robot's sensors to compute odometry and Simultaneous Localization and Mapping (SLAM) to produce a map of the surrounding environment.

The leadership team consists of the Team Lead, Operations Director, Engineering Director, as well as leads for each subteam. The Team Lead, Operations Director, and Engineering Director provide the general direction and strategy for the team, and the subteam leads focus on the technical development of their respective subsystems.

Design Process and Assumptions

The design process for the robot was split up and assigned to the aforementioned subteams. Throughout the fall semester, the platform team consulted with the other subteams to develop a CAD plan for the robot, while the other subteams primarily used the time for onboarding and simulation. We also held monthly design review meetings with our mentors to discuss our design choices and implementation plan. In the winter semester, we finished building the robot platform in two months, and the software subteams consequently tested and debugged the vehicle code. Figure 1 shows how different aspects of our season contributed to the design process.

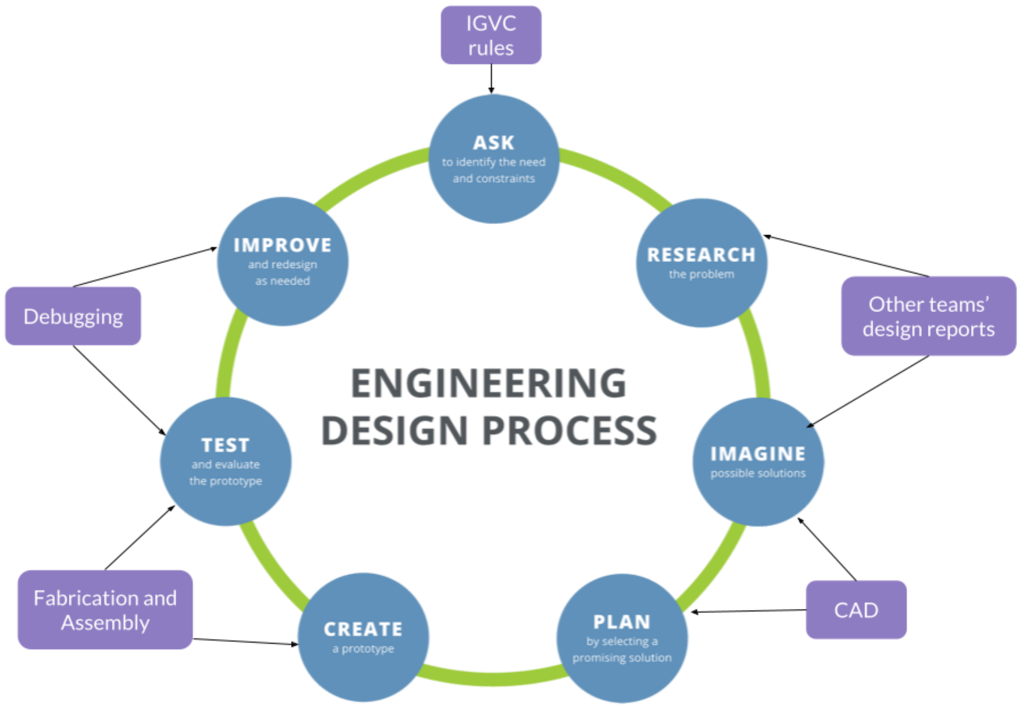

IGVC rules
ASK to identify the need and constraints
RESEARCH the problem
Other teams' design reports
IMAGINE possible solutions
CAD
PLAN by selecting a promising solution
CREATE a prototype
Fabrication and Assembly
TEST and evaluate the prototype
Debugging
IMPROVE and redesign as needed
ENGINEERING DESIGN PROCESS

Fig. 1: The ARV Design Process

There were several key assumptions and design choices that all subteams agreed upon before the chassis was designed. They are listed below:

- The design must facilitate debugging. This meant allowing the laptop to be easily accessible and have a dedicated space on the robot itself.
- Wheel slippage is negligible. Both the embedded systems and navigation subteams have error correction built into their software design.
- The design must be modular. Each subteam must be able to easily add, access, and remove components.
- The robot should withstand light rain and other debris; the robot is both splash- and water-proof.

Vehicle Design Innovations

Two two main innovations in our robot for this year include a rack-railing concept as well as improved weatherproofing. These innovations were necessary to increase the efficiency of hardware and software development as well as improve environmental protection of the robot, specifically from rain.

Innovative Concept from Other Teams' Vehicles: Rack-Railing Concept

One innovative concept integrated into our new vehicle is the railing system. Last year, the team had difficulty debugging the vehicle's code because there was a lack of space for a laptop. Also, the difficult access to the components in the system made the identification of hardware problems worse. The railing system was designed so that during the testing process, different systems could be checked and tested individually, without having to disconnect wirings to systems other than the one of interest. Figure 2 shows the three movable shelves, one for the electrical system, one for the laptop, and one for the Jetson and sensor systems, along with the rack-railing mechanism that facilitates guided motion. Each shelf could be slid out because the wirings are long enough to not disconnect from the components. With the increased ease in accessibility, the testing process became quicker and allowed the design cycle to move much smoother.

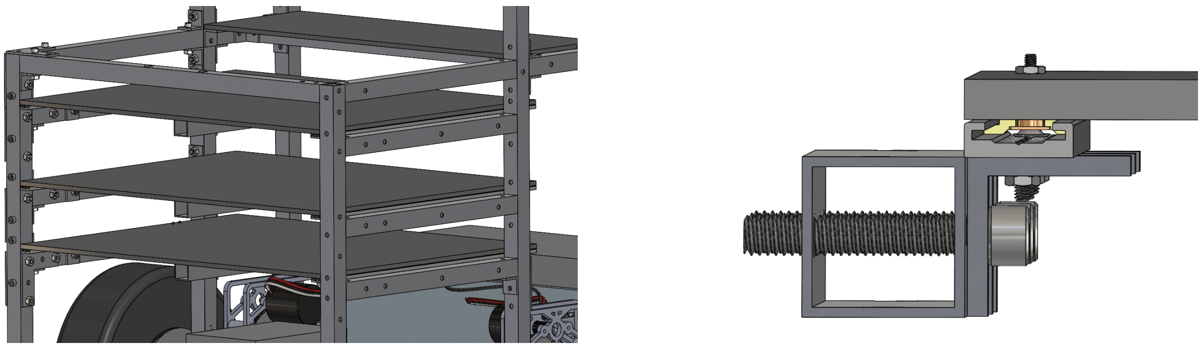


Fig. 2: Complete shelving stack and view of the rack-railing mechanism

Innovative Technology Applied to our Vehicle

To improve our robot's weatherproofing ability, we also decided to use shower curtains to make a raincoat for the robot. We will discuss this in depth in the later sections.

Description of Mechanical Design

The objective of the mechanical system is to provide the chassis, payload support, and a maneuvering method for our robot, mARVin. The team used Solidworks for designing the platform and various machines for manufacturing including a waterjet, drill press, horizontal bandsaw, and laser cutter. CAD was used to create models for different parts of the chassis and to assemble all the parts with bolt placement in consideration. Figure 3 shows the CAD model of mARVin in Solidworks.

Decision on Frame Structure, Housing, Structure Design

Designed with portability, serviceability, and longevity in mind, the chassis is constructed from 1-inch square aluminum tubing to minimize overall system weight and maintain rigidity. The dimensions are within the competition requirements at 35 inches wide by 47 inches long and well below the height limit. of six feet. The chassis is secured by custom-made mending plates and attachment brackets and secured with one common 1/4 inch thread hex bolt and nut to improve serviceability. The brackets and mending plates can be easily manufactured and the hex bolt prevents stripping. Having a common bolt and nut type makes the entire system easier to service. The robot features a two-wheel drivetrain with a third free caster wheel to provide a balance between power, stability, and maneuverability. Inside the main chassis of the robot, there are two gearboxes with motors, encoders, a battery, and shelves for the embedded system, Jetson, and a laptop. The batteries are placed in the interior of the robot, behind the drivetrain to act as a counterweight against the payload. This allows the chassis to be balanced even on a fifteen percent incline. Corner guards were 3D-printed to hold the batteries firm against the chassis. At the rear of the robot, there is a physical emergency switch with LED safety light and a power switch of the entire system, as stipulated by the rules requirement.

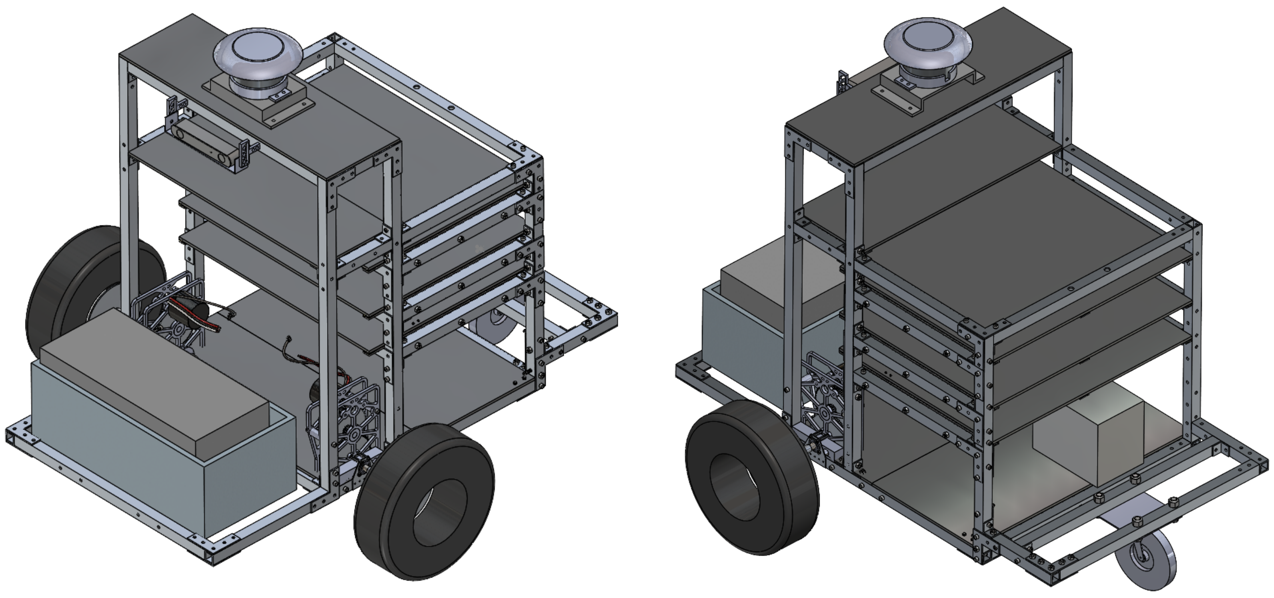


Fig. 3: Front and back isometric views of mARVin as designed in Solidworks

Weatherproofing

The white main structure is built from metal frames with plastic coverings on all six sides to protect from debris. A covering made from a sewed polyester shower curtain was placed over the vehicle to protect the primary and secondary computers from rain. Figure 4 demonstrates the functionality of the covering as it allows for ease of access to vehicle electronics without sacrificing protection when needed. The LiDAR and stereo camera attachments are waterproof, with the LiDAR having additional protection from the rain in the form of a curved 3D printed covering. The safety stop, the stereo camera, and the payload holder are placed outside the main metal frame because they would not be damaged by weather conditions.

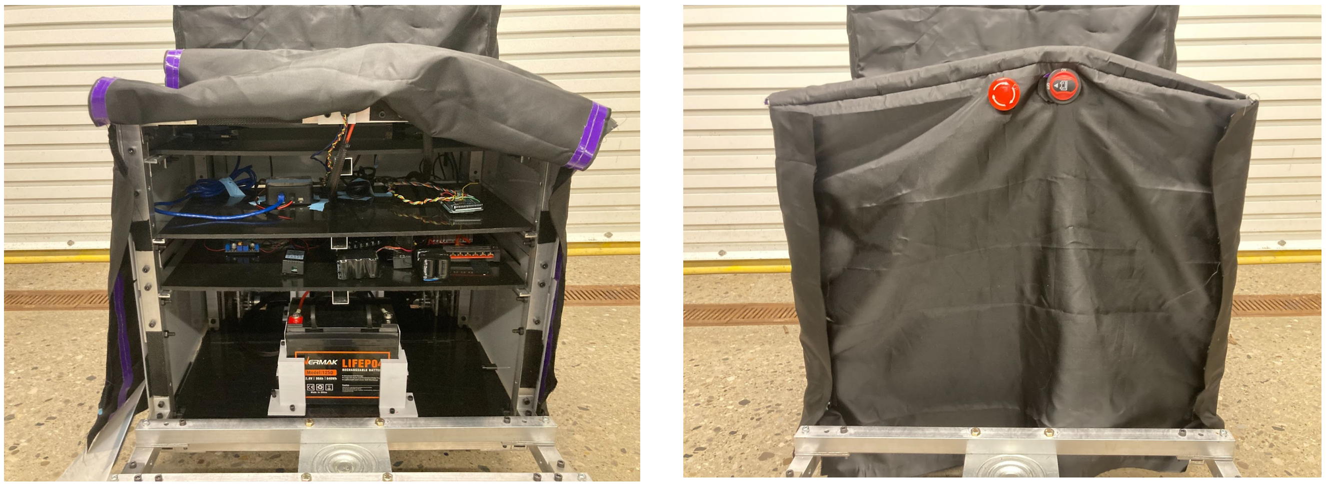


Fig. 4: A demonstration of the raincoat protecting the electronics stack while not limiting function

Suspension

mARVin lacks suspension in its structure as it would create unnecessary weight on the vehicle. We determined that suspension would not be needed as the robot would be driven on asphalt, which is relatively flat and smooth. Although there would be slopes and inclines on the course, the shock would not be too large on the robot as the change in slope is not immediate.

Electronic and Power Design

The electronic and power design was implemented by the Embedded Systems subteam. The team updated the vehicle with O-Drive motor controllers with onboard PID tuning and new DC brushless motors. Fig. 7 shows a diagram of the electrical and power system.

Power Distribution System

The power distribution system consists of a single nominal 12V 50 A-hr $LiFePO_4$ battery, two power rails, several step-down buck converters, and decoupling capacitors. The battery is connected to a main power rail that roughly maintains a 13V difference. This then feeds into a 12V high-power buck converter that is connected to a secondary 12V rail. This rail then feeds into a 9V buck converter that powers the Arduino Mega. Another 5V buck converter exists to step down the signal from the remote E-Stop before being sent to the GPIO pin on the Arduino Mega. Almost all other components that are not powered by their respective computers or microcontrollers are powered by the 12V rail. The exception is the ODrive and the motors, which are powered directly from the 13V rail. To ensure power integrity, twelve 4700 µF capacitors are connected in parallel between power and ground. This mitigates changes in the rail's voltage upon sudden current draw changes from the various components.

Electronics Suite Description

We use a wide range of electronics on the vehicle, including computers, GPSs, and motors. These are described further below.

NVIDIA Jetson Orin

The NVIDIA Jetson provides a high-speed discrete GPU suitable for real-time image processing and pairs particularly well with the ZED camera, as Stereolabs maintains an SDK specifically for the Jetson. The Jetson provides exceptional performance considering its power consumption, form factor, and price. In addition, the included development board has integrated HDMI, Ethernet, USB, USB-C and WiFi to speed up development.

Razer Blade 15

The Razer Blade 15 has an Intel i7 CPU paired with Nvidia RTX 3060 mobile GPU. This combination of hardware enabled the ability to perform point cloud processing, localization & mapping, path planning, and computer vision tasks simultaneously. The laptop form factor is convenient to work with and allows for hours of testing without the need to be powered as shown in figure 5.

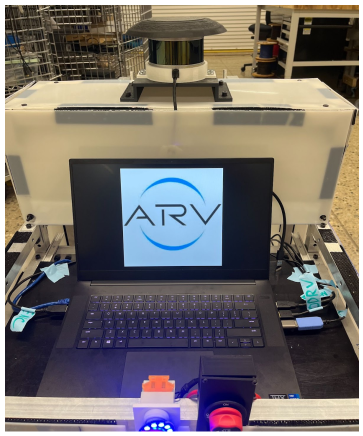


Fig. 5: Laptop on the top shelf

Velodyne VLP-16

The Velodyne VLP-16 outputs a 360 degree 3D point cloud with a refresh rate of 10 Hz. The Velodyne has significantly increased range, accuracy, and weatherproofing over the RP-LiDAR we used for the previous years and has become an integral part of our sensor systems.

Adafruit BNO055 Absolute Orientation Sensor
The BNO055 IMU provides absolute orientation, angular velocity, acceleration, magnetic field strength, and temperature data. This sensor was chosen because of its high refresh rate, low noise data, and the well documented libraries and packages.

Stereolabs ZED 2i Camera
The primary draw of the ZED camera (shown in figure 6) is its low cost and high depth-sensing range. Compared to other RGBD solutions, the ZED camera offers much higher depth cloud resolution through software processing of the stereo images. The Stereolabs development team has provided a rich SDK with ROS integration included, speeding up deployment cycles by reducing hardware and embedded development time.

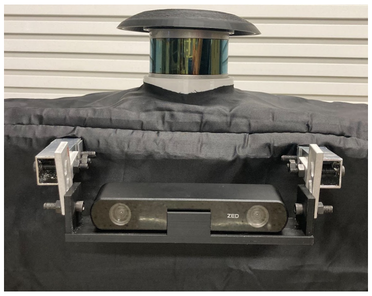


Fig. 6: Mounted ZED camera

Garmin GPS 18x USB
The Garmin 18x GPS offers < 3 meters of accuracy with a 95% typical through a developer friendly USB interface. The 18x is designed for automotive applications and as such comes weatherproofed, a significant factor in our decision to keep using the same model.

ODrive Motor Controller
The ODrive motor controller offers tight integration with velocity commands, having built-in PID position and velocity control. In addition, a wide variety of customization and diagnostic options provide a significant quality-of-life boost while interfacing with hardware. Examples include monitoring system voltage and current usage for each of the motors and specifying performance characteristics to match user-specified operating parameters. ODrives are also able to add velocity and current limits via software, creating another level of safety for the vehicle.

Neo Brushless Motors
The Neo motors were chosen to drive the vehicle due to their onboard hall effect encoders and their torque at the desired RPM which gives around 5 mph with the gear ratios. It was necessary to change to brushless motors since the ODrive motor controllers are only compatible with brushless motors. This gave the opportunity to upgrade the motors and include the encoders into the motors instead of having to attach them ourselves. This reduced complexity and a point of failure of the vehicle.

Phidgets Optical Rotary Encoder ISC3004
The Phidgets Encoder is mounted on the gear box. With the calculated gear ratio, and the 360 CPR, 80 kHz data, we are able to interpolate the position and velocity of the robot. These encoders also add physical support to the wheel axles themselves.

Arduino Mega
A simple hardware/software layer was required to interface between our ROS layer and the serial interface of the ODrive motor controllers. Using an Arduino Mega allows us to process ROS messages on a lower-level device, allowing the maximum abstraction of the drivetrain to the ROS stack. The ODrive's manufacturer also provides and maintains an Arduino library to interface with the velocity controls of many motor controllers connected over serial, speeding up development and reducing testing time.

The robot currently uses two Arduino Megas: one to receive velocity commands, transmit them to the ODrive, and set the light, and another one to solely read the Phidget encoders and transmit them for sensor fusion and odometry. The extra Mega was introduced to reduce the workload of a single microcontroller and better fine-tune data publishing rates.

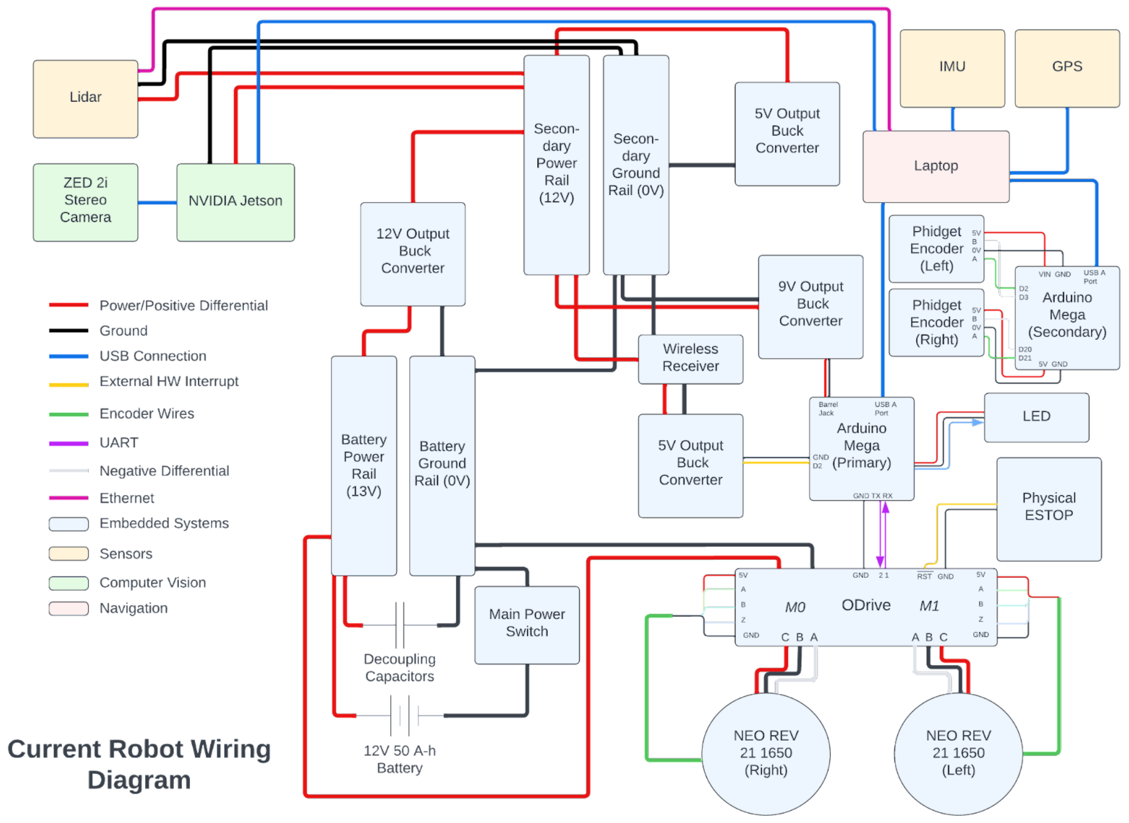


Fig. 7: Electronics and Power diagram

Safety Devices and Integration

Being able to operate our robot safely is a key part of the competition. When enabling the robot, main power from the batteries is enabled by flipping a circuit breaker mounted on the outside of the bot, easily seen and accessible by anyone. When the robot is turned on, power is supplied to a status light, showing its current state. In an effort to make the robot more modular, the Platform subteam designed 3-D printed mounts for the batteries that can be easily attached and removed.

To ensure that no safety issues arise during a run, a physical E-Stop, remote E-Stop, and speed limiters are integrated into the robot using the Arduino microcontroller and ODrive. The physical E-Stop is a large red button connected directly to the ODrive, which upon being pressed, will immediately interrupt the ODrive and cut power to both motors. The remote E-Stop has a range of 250 feet and is operated by a small remote. Pressing the “A” button will send a signal to a GPIO pin on the Arduino, interrupting the processor and forcing a 0 mph command to be transmitted via software and override any other velocity commands. Pressing the button again will allow the robot to resume where it left off. The ODrives also have a 5 mph limit and 30A current draw limit per motor set as part of its configuration. Upon exceeding this threshold, the ODrive will immediately halt the motor in violation.

Software Strategy and Mapping Techniques

All of the robot's software is powered by the Robot Operating System (ROS) running on a base Ubuntu 20.04 installation. In line with our modular design philosophy, ROS was selected as the robot's operating system due to its extensive modularity, community support, and power features. ROS is a distributed networking and communications library allowing multiple devices to work together. A ROS computation graph is divided into discrete nodes that can publish and subscribe messages. Nodes communicate with each other over TCP, allowing them to connect to nodes on other computers through our Ethernet switch. This system facilitates the communication between different processes and enables the team to work on independent tasks; each software subteam can develop nodes entirely separately from the others.

The goal of the robot is to navigate through a series of waypoints while avoiding obstacles identified with data from the onboard sensors. Figure 8 shows the connections between the sensors and navigation subteams. The sensors team creates an occupancy grid using a variety of different tools, which is then passed to the navigation subteam to path plan to the next GPS waypoint. This process is explained in more detail below.

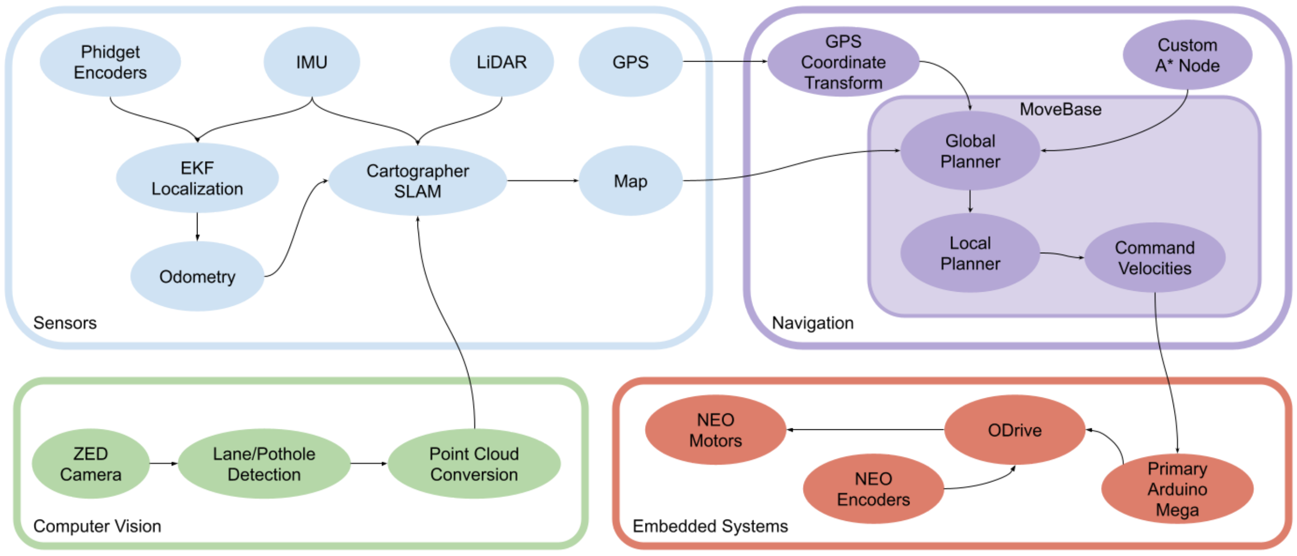


Fig. 8: Software architecture diagram of mARVin

Obstacle Detection and Avoidance

We use the Velodyne VLP-16 for identifying obstacles above the ground level. It has a 100-meter range, and 360° field of view, which is perfect for detecting cones and other roadblocks. We use the ZED camera for detecting ground level obstacles such as lanes and potholes. We take the raw camera feed and the depth image as the input. First, we run our lane and pothole detection algorithm, which consists of a white color thresholding, Gaussian blur, and other image manipulation techniques. After running our lane and pothole detection, we output the depth values of the lanes and potholes in a point cloud format.

Finally, the point clouds from the LiDAR and the Depth camera are fed into Google Cartographer SLAM. Using the occupancy grid that Cartographer outputs, the vehicle navigates around obstacles in real time. The navigation stack is consistently sent updated maps, and the global planner and local planner work together to create a path to the next waypoint while making sure not to move too close to any obstacles detected. Our custom A* node is explained in more detail in the Software Strategy and Path Planning section, but allows us to quickly change our pathing to avoid obstacles that we may discover or encounter while moving.

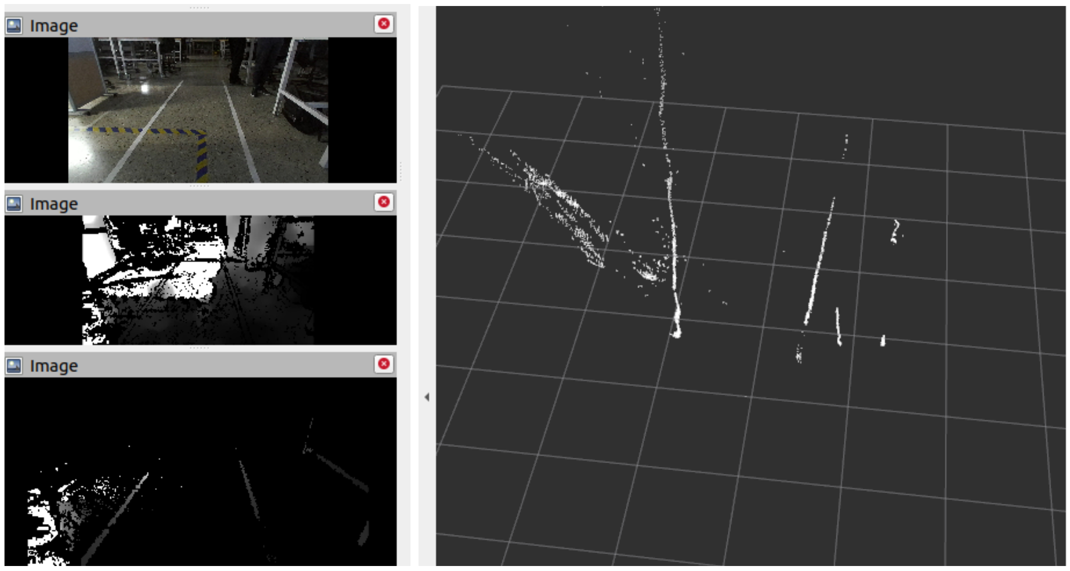


Fig. 9: Different steps of the lane detection algorithm

Map Generation

We utilize a sophisticated pose-graph Simultaneous Localization and Mapping (SLAM) solution called Google Cartographer. Cartographer offers a robust and highly configurable solution that permits us high confidence in the quality of generated maps, especially in noisy environments.

Cartographer integrates into ROS and provides an occupancy grid containing the obstacles identified in the point cloud data from the LiDAR and camera. At the same time, we utilize data from encoders and the IMU in an Extended Kalman filter, as well as LiDAR point cloud matching to estimate the location of the robot in the map.

Software Strategy and Path Planning

Sensor fusion between the IMU and wheel encoders is accomplished through an Unscented Kalman Filter, which is more forgiving than an Extended Kalman Filter when it comes to calibrating the sensor odometry. The GPS was chosen to be left out of odometric sensor fusion due to its non-continuous nature, which testing revealed significantly reduced the accuracy of pose estimates.

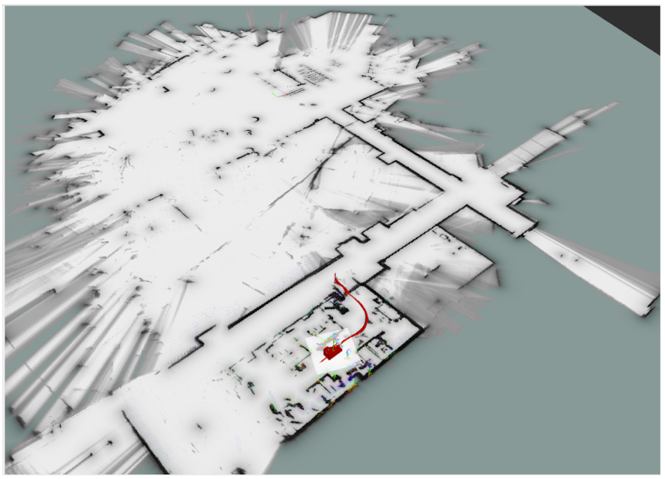


Fig. 10: Picture of the Map

The final costmap from lane/pothole detection and SLAM is continuously provided to our global planner, which uses a custom A* node. Our A* node implementation takes advantage of the efficiencies found in D*Lite, leading to our algorithm only recalculating a path only when there are new obstacles that would directly interfere with the current path of the robot. We created a MoveBase plugin that uses our custom A* code to replace the base global planner provided in MoveBase by default, which uses repeated A*.

Goal Selection and Path Generation

The GPS waypoints are transformed into the robot's world frame to simplify path planning. Given a global costmap, a local target is found to move the robot toward the nearest GPS waypoint. To calculate this local target, the goal was to find a straight line to the GPS waypoint and use the intersection point between that line and the border of the cost map as a target position. In the end, we will receive a cost map from move_base, perform our search algorithm to find a path, and send a set of nodes back to move_base for the command velocities to be performed.

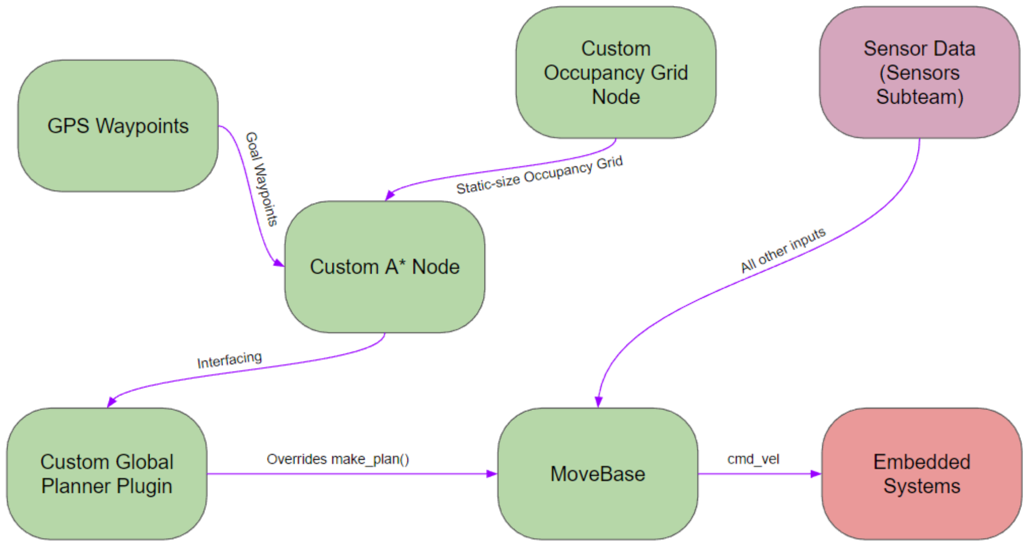


Fig. 11: Software architecture of the navigation stack

Software Vehicle Failure Modes

If the vehicle becomes stuck or is unable to find any possible paths to the goal, the vehicle will enter a recovery behavior state. The vehicle will start by slowly rotating in place to re-localize itself on the created map. Once the vehicle has remapped the surrounding area, and finds a path to the provided goal, it will resume normal navigation behaviors. In the extreme case that the robot is fully stuck in place, it will increase the power provided to the motors to forcibly remove itself from an obstacle.

In case of SLAM scan matching algorithm failure, the newest odometry information is used to estimate the current pose of the robot. SLAM nodes are updated using forward projection according to the optimal solution for the pose graph.

Hardware Vehicle Failure Points

Mechanically, the robot could potentially fail from the velcros and the plastic frames getting loose. Furthermore, the bolt attachments could potentially get loose. Electrically, the robot could violate user-specified thresholds (speed, current, etc.), tripping errors on the ODrive and motors. This in turn could potentially shut off the motors.

Failure Prevention Strategy

The general troubleshooting process for hardware and software failures is as follows:

1. Check that the status lights are lit and indicate nominal operation.
2. Check that connector cables are securely attached.
3. Verify that software nodes are running and messages are being transmitted.
4. Run ROS troubleshooting like `roswtf`, `rqt_graph`, and `view_frames` to verify that the node and message graphs are properly set up.

Mechanical

To prevent the aforementioned mechanical failure points, the robot has been designed with an aluminum frame, deferring most of the potential stress on the velcros to the subsystems themselves. In case this is not sufficient, the team will keep a surplus of extra velcros to replace any loose velcro connections. The bolt attachments have been designed so that there are no shear forces acting on the bolt during robot operation, mitigating this failure point.

Electrical

To prevent the aforementioned electrical failure point, the primary Arduino Mega will periodically monitor the error flags the ODrive sets for each of the motors. Upon detecting any error, the Mega will temporarily halt the robot, ignore all navigation commands, reset the ODrive error flags, and recalibrate the motors that are in violation before resuming normal navigation.

Software

On the software side, there are many safeguards put in place to prevent unwanted behavior of the vehicle. First, the robot will not map and move to locations that are in completely unknown space, outside the range of the global costmap. This prevents the robot from moving too far away from the emergency stop range during testing. Additionally, real time SLAM and path planning allows for dynamic obstacle avoidance, so the vehicle should avoid any spontaneously appearing objects on its path.

The sensors team utilizes sensor fusion concepts to minimize the effect of a sensor failure. The data from the IMU and the encoders are combined to form odometry data. When either sensor fails during an operation, the robot position can still be estimated using the other sensor, though with less precision. We have also added redundancy and modularity in our sensor systems. The encoder readings can also be provided by the Odrive motor controller. Thus, in a case of critical encoder failure, it is possible to read data directly from the Odrive by simply subscribing to another topic in ROS.

Testing

Navigation

To test the navigation systems, we took advantage of buildings on campus. We successfully planned paths through hallways and large rooms that contain many obstacles such as tables, chairs, and pedestrians. Waypoints were added during testing by directly adding a 2D navigation goal through RViz on the onboard laptop. To test GPS functionality, we collected multiple rosbags of GPS data while moving through the city of Ann Arbor. This GPS data was then tested with the GPS node code separately. The main difficulty we faced during testing of the navigation stack was that the sensors stack would often detect the people setting up the vehicle as obstacles directly behind itself.

Lane Detection

White tape was placed on a parking lot in the shape of lane lines, and the robot was pushed through the course to observe what the point cloud output was. By comparing this to the real world we were able to determine whether or not the computer vision algorithm was working. When testing, we found that the depth map we received from the ZED camera did not always have the depth values on the lane lines as shown in figure 12 below.

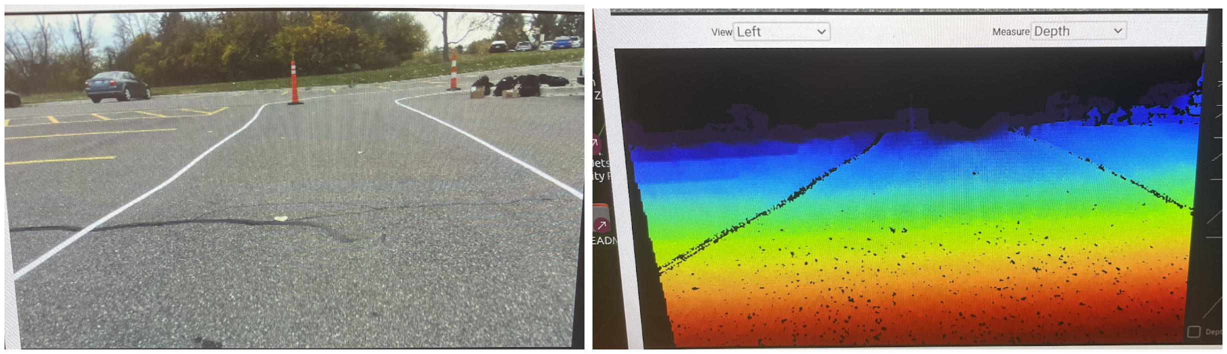


Fig. 12: Absence of depth values on lane lines

The left image is the raw camera image, and the right image is the depth map generated by the ZED camera. Since the white lanes are so uniform, the depth map has trouble generating depth values on those lane lines which greatly disrupts our lane detection pipeline. Our solution is to dilate the depth map, which allows us to find the maximum value nearby the lane lines, which is close enough to the correct depth that the output will still be correct.

Sensors

Testing the sensor system involves testing individual sensors separately and integration testing with a combination of different sensors. To test the encoders, we have pushed the robot on the ground, ensuring no wheel slippage. We then verified the various distances pushed with the number of rotations recorded by the encoder multiplied by the corresponding gear ratio and wheel radius. To test the IMU sensor, we were able to visualize the data collected by the data in RViz and ROS. We have observed an accurate gyro but a significant drift in the acceleration data. To combat inaccurate readings, we wrote a custom python calibration script that offsets the acceleration readings to achieve better results. To test the GPS sensor, we walked different movement configurations in the parking lot, then plotted the collected coordinates. The figure 13 below shows the received coordinates for walking in a straight line along the parking lane.

The sensor integration testing started with simulation. We have implemented a version of Extended Kalman Filter (EKF) for robot state estimation, and compared the estimated odometry data with the exact odometry vectors that's provided in simulation. We also deployed all the sensors to the robot and published the collected data to Google Cartographer SLAM. We tuned the configuration file with an iterative approach and was able to make the map building efficient and accurate.

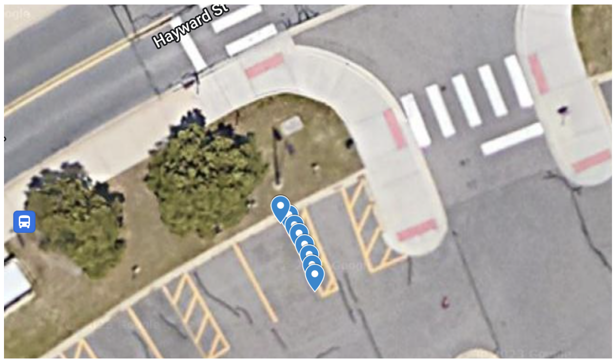


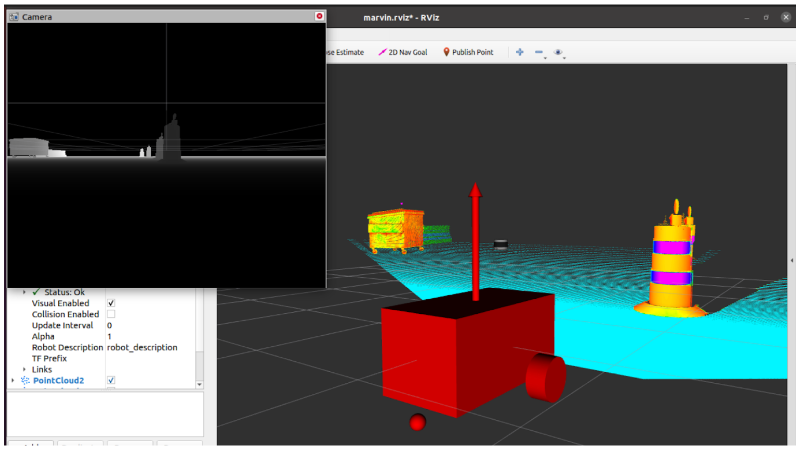
Fig. 13: GPS coordinates from walking in a straight line

Vehicle Safety Design Concepts

In addition to both E-stops and software limits, both failure prevention strategies and testing implement several safety design concepts. For the software and mechanical prevention strategies, a soft bumper, made using a pool noodle, was included in the robot to minimize any damage from potential collisions. For the electrical prevention strategy, the robot is halted to prevent any undefined behavior. For testing, one person is assigned to solely operate the remote E-stop, ensuring that someone can immediately stop the robot when necessary.

Simulations in Virtual Environment

The robot and its sensors are simulated in Gazebo, and the various types of data visualized in RViz. These software were chosen because of their well documented integration with ROS. We have built the robot simulation model from scratch, with simulated sensors such as the IMU, depth camera, LiDAR. The Navigation Stack and Google Cartographer SLAM subscribes to the sensor data, generating the map the the planned path in the process. The environment map was based on the Auto-Nav course illustrated in the competition rules. We have made some custom models such as ramps and lane lines using Blender and imported them into Gazebo.



✓ Status: Ok
Visual Enabled ✓
Collision Enabled
Update Interval 0
Alpha 1
Robot Description robot_description
TF Prefix
Links
PointCloud2 ✓

Fig. 14: The simulated camera and LiDAR visualized with RViz

Theoretical Concepts in Simulations

We were able to simulate and debug the SLAM and autonomous navigation inside simulation. To start with, we modeled the robot and its sensors using the URDF format and Gazebo sensor plugins. We wrote custom launch files to spawn and tele-op the robot into the competition world that we built last year. We were able to set up

Google Cartographer inside simulation, subscribing to the LiDAR and IMU nodes. We tuned the LUA configuration files to improve the map generation speed and quality.

One important theoretical concept that we implemented in simulation was the ground filter. The VLP-16's 3D scan will reach the ground and make that into an obstacle. The ground filter can then be used to delete points below a desired height to achieve clean maps for navigation.

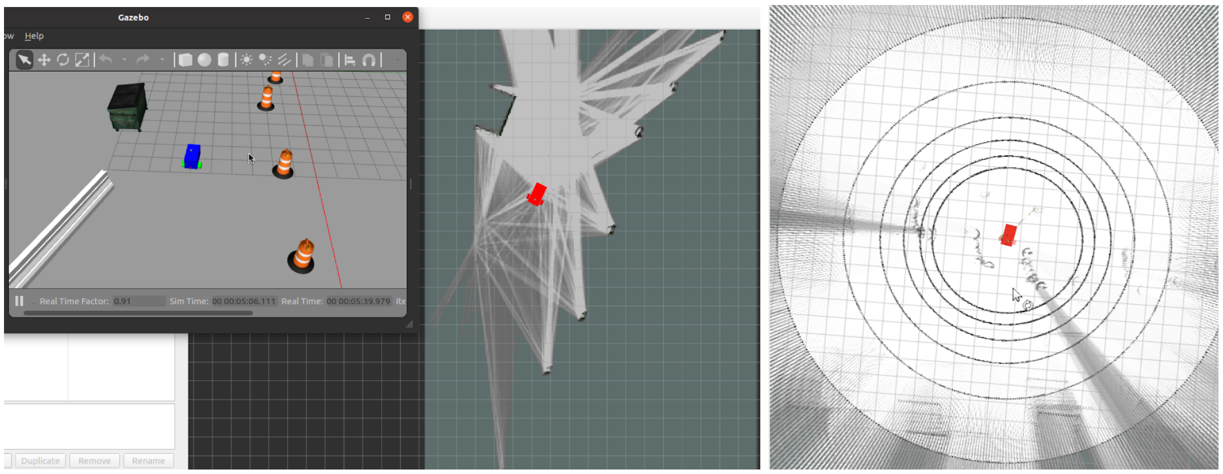


Fig. 15: SLAM algorithm with/without the Ground Filter

Performance Testing To Date

Component Testing

Individual components on the vehicle such as the motor controller, wheel encoders, LiDAR, GPS, IMU, Depth Camera, wireless and physical e-stops have been tested. We utilized a tele-op controller to test the motor control, and RViz to visualize the sensor data collected.

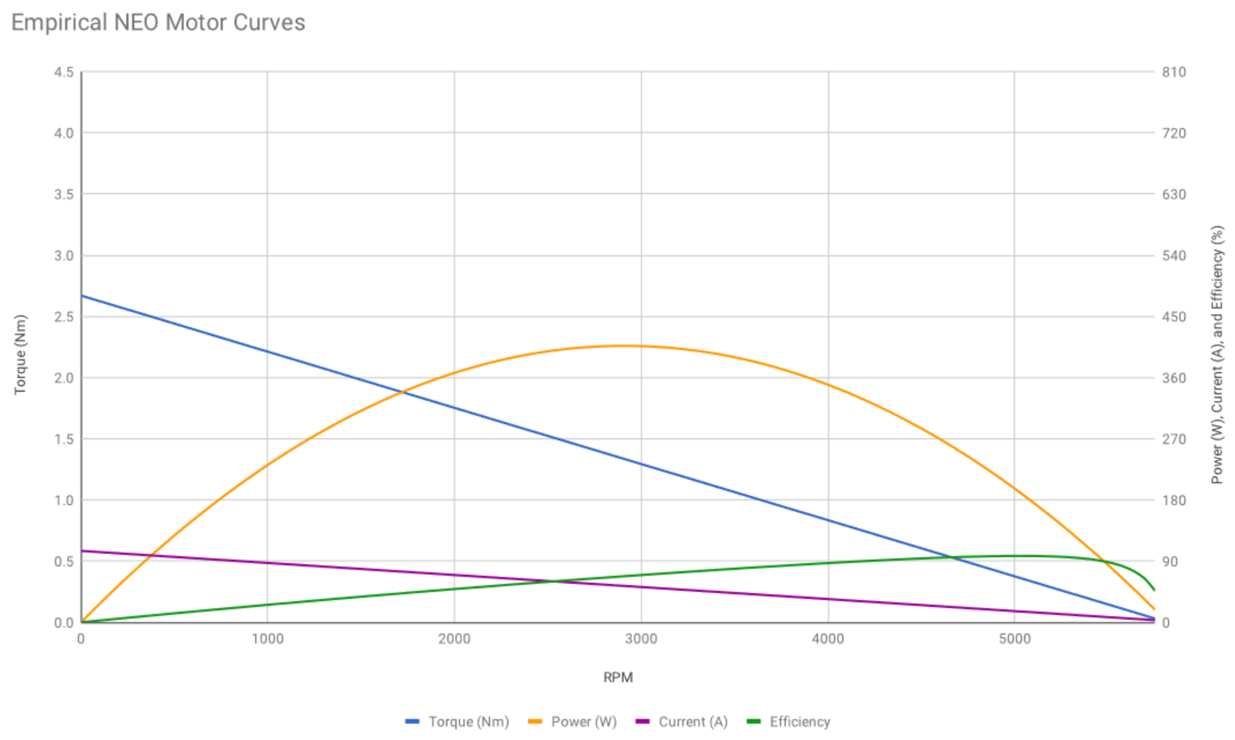
Integration Testing

Combining the separate components into a combined vehicle was a challenging task. While many of our systems had problems at first, we were able to solve these during our testing procedure. While we were able to fully combine the embedded, sensors, and navigation stacks correctly, we still have trouble with the fusion of the computer vision and lidar point clouds. Additionally, the computer vision system that detects lane lines and provides them to Google Cartographer as a depth map is not fully functional.

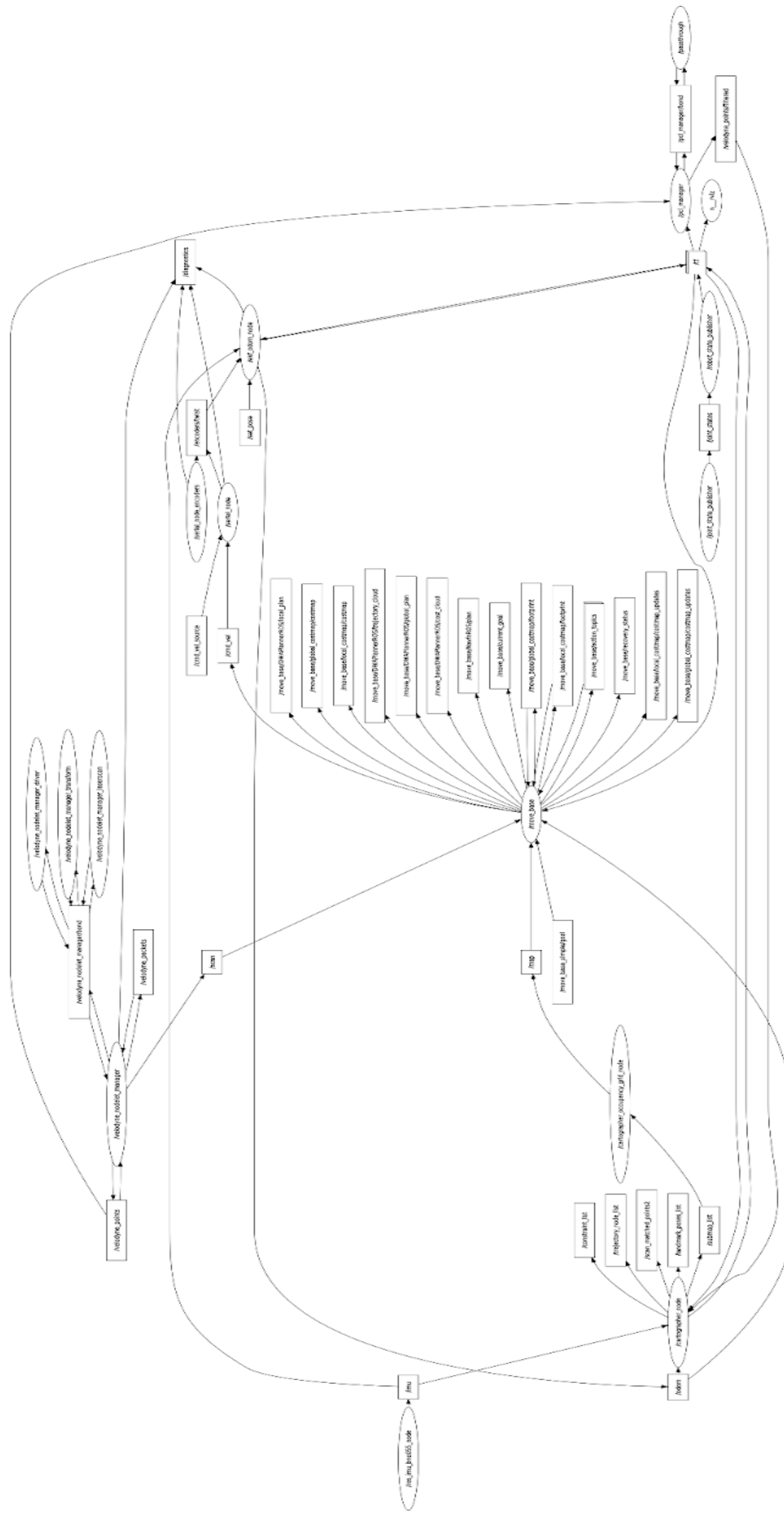
Initial Performance Assessments

Metric	Test Result
Max Speed	3.4 m/s
Acceleration	0.4 m/s^2
Ramp Climbing	20 degrees
Laptop Battery Life	1 hour
Robot Battery Life	50 hours standby 1 hour running with motors

Appendix A: Torque and Efficiency Curves for the Neo Brushless Motors



Appendix B: ROS Graph for the Software Stack


/passthrough
/pcl_manager/bond
/velodyne_points/filtered
/pcl_manager
/rviz
/tf
/diagnostics
/wheel_odom_node
/robot_state_publisher
/joint_states
/joint_state_publisher
/wheel_pose
/encoders/twist
/serial_node_encoders
/serial_node
/cmd_vel_source
/cmd_vel
/move_base/DWAPlannerROS/local_plan
/move_base/global_costmap/costmap
/move_base/local_costmap/costmap
/move_base/DWAPlannerROS/trajectory_cloud
/move_base/DWAPlannerROS/global_plan
/move_base/DWAPlannerROS/cost_cloud
/move_base/NavfnROS/plan
/move_base/current_goal
/move_base/global_costmap/footprint
/move_base/local_costmap/footprint
/move_base/parameter_updates
/move_base/recovery_status
/move_base/local_costmap/costmap_updates
/move_base/global_costmap/costmap_updates
/velodyne_nodelet_manager_driver
/velodyne_nodelet_manager_transform
/velodyne_nodelet_manager_laserscan
/velodyne_nodelet_manager/bond
/velodyne_packets
/scan
/map
/move_base_simple/goal
/move_base
/cartographer_occupancy_grid_node
/constraint_list
/trajectory_node_list
/scan_matched_points2
/landmark_poses_list
/submap_list
/velodyne_nodelet_manager
/velodyne_points
/imu
/cartographer_node
/odom
/ros_imu_bno055_node