CEDARVILLE UNIVERSITY®

Alpha Bee

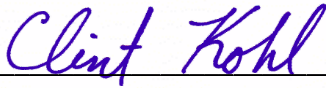
April 28, 2021

Prepared by:

Cedarville University AutoNav Senior Design Team

Team Name	Swarm	
Team Captain	Joshua Kortje	joshuakortje@cedarville.edu
Team Members	Isaiah Higgins Daniel Jacobs Trevor Loula	isaiahmhiggins@cedarville.edu danielcjacobs@cedarville.edu trevorloula@cedarville.edu
Faculty Advisor	Dr. Clint Kohl	kohlc@cedarville.edu

I hereby certify that the design and development of the vehicle Alpha Bee, described in this report is significant and equivalent to what might be awarded credit in a senior design course. This is prepared by the student team under my guidance.



Dr. Clint Kohl Ph.D.

Conduct of Design Process, Team Identification, and Team Organization

Introduction

This report describes the Cedarville University's AutoNav senior design team's work during 2020-2021 academic year in constructing an autonomous vehicle. The purpose of this vehicle is to safely and effectively navigate the IGVC (Intelligent Ground Vehicle Competition) obstacle course using line following, obstacle avoidance, and GPS waypoint navigation technologies. In this report, we describe our approach to this project, as well as our design and overall progress.

Team Organization

Our team consists of four senior computer engineering students. Joshua Kortje functions as the team's Chief Executive Officer and is responsible for facilitating communication between the team and the IGVC officials. Isaiah Higgins functions as the team's Chief Financial Officer and is responsible for tracking and reporting all financial operations. Daniel Jacobs functions as the team's Chief Technical Officer and is responsible for reporting on the robot's current physical state. Finally, Trevor Loula functions as the Chief Software Officer and is responsible for reporting on the robot's current software state. Each team member had their areas of expertise on the project, but as a small team they also all had significant overlap and a general understanding of how the entire robot functions.

Name	Year & Major	Position
Isaiah Higgins	Senior Computer Engineer	Chief Financial Officer
Daniel Jacobs	Senior Computer Engineer	Chief Technical Officer
Joshua Kortje	Senior Computer Engineer	Chief Executive Officer
Trevor Loula	Senior Computer Engineer	Chief Software Officer

Design Assumptions & Process

Since we are a new undergraduate team with no previous team's work to continue, the decision was made early on to focus on the fundamentals and work toward basic functionality. We determined these basics to be sensing the white lines and construction barrels, and to be able to navigate based on these inputs. We chose to use the ROS (Robot Operating System) as our development environment to allow for easy integration and to utilize existing libraries whenever possible.

Effective Innovations

All computations are performed on a single Nvidia Jetson Nano development board. As such, we focused heavily on keeping all our algorithms simple and lightweight. The various hardware components are all connected to the Nano as visualized in Figure 1. The Motor Controller and depth camera connect via the USB ports while all the other sensors use the GPIO pins on the side of the Jetson Nano.

Within a ROS framework, various software nodes were developed to modularize the software and keep it easy to debug and extend. Some nodes are primarily input nodes and focus on taking in readings from the environment. After those nodes take in the sensor data, that data is often sent to a processing node that will do whatever sensor processing needs to be done such as finding a line in the image or detecting if an obstacle is in the way. The processing nodes routinely send information to a Finite State Machine (FSM) which is at the heart of the decision-making process. This FSM will determine when significant changes need to the make to the operation of the robot such as to stop following a line and start

navigating by GPS waypoints. Either the sensor processing nodes or the FSM will direct the movement of the wheels depending on whether the robot is doing a transitionary maneuver or navigating via the external sensors.

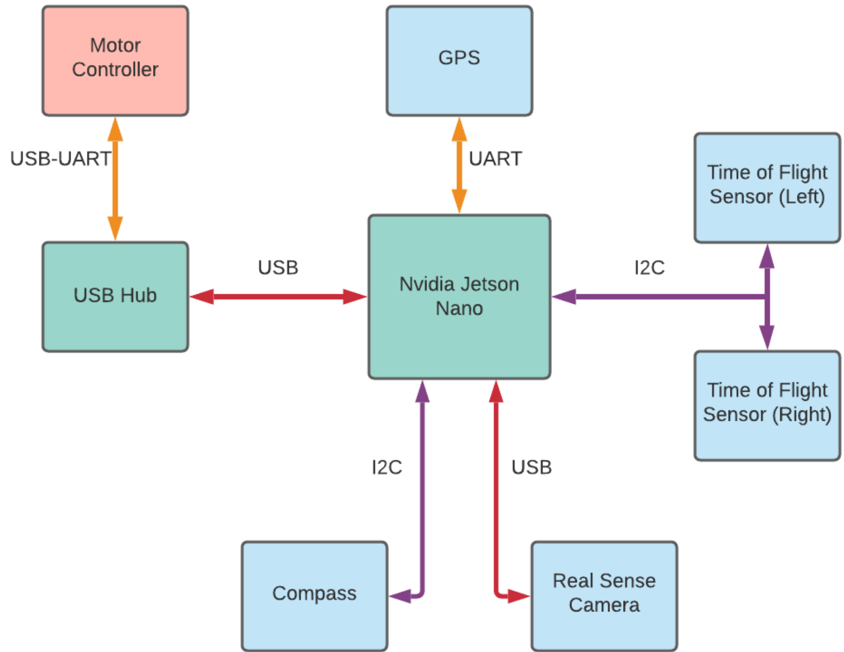


Figure 1. Connection Diagram

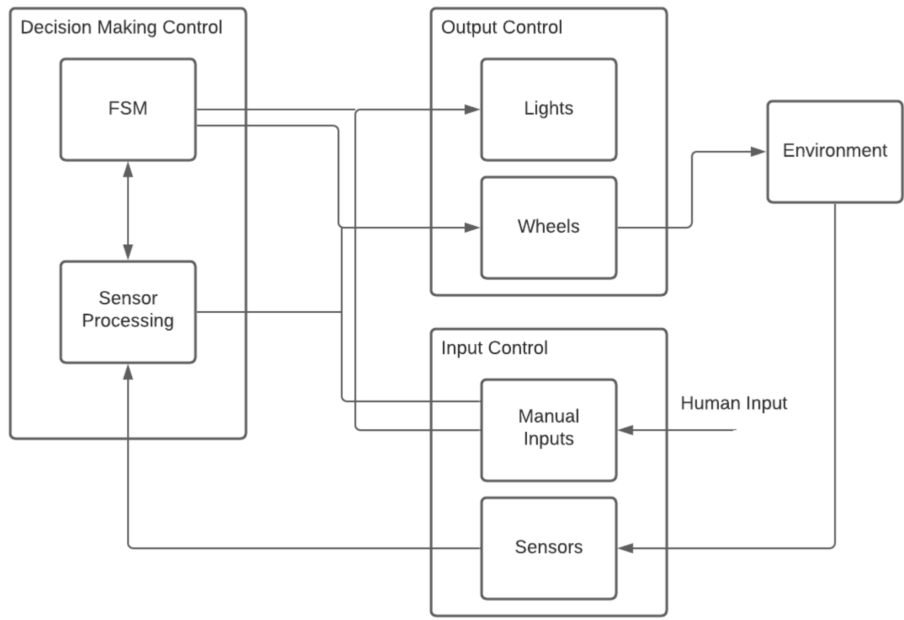


Figure 2. Control Flow Diagram

Mechanical Design

When designing the robot frame, the greatest consideration was given to the placement of sensing components, weight distribution, and ease of access for internal electronics. Only two driven wheels were used to provide the most maneuverability with a back wheel for stability. The frame was made of welded ½" square metal tubing. The major considerations in the chassis design were meeting all the requirements of the competition rules while remaining small but fully functional. The chassis was able to house all the needed sensing equipment and allowing them to be placed at ideal angles and positions. The frame is lightweight and sturdy and allows for easy access to the inside components, making removal, movement, and addition of parts quite easy. Many mounting components

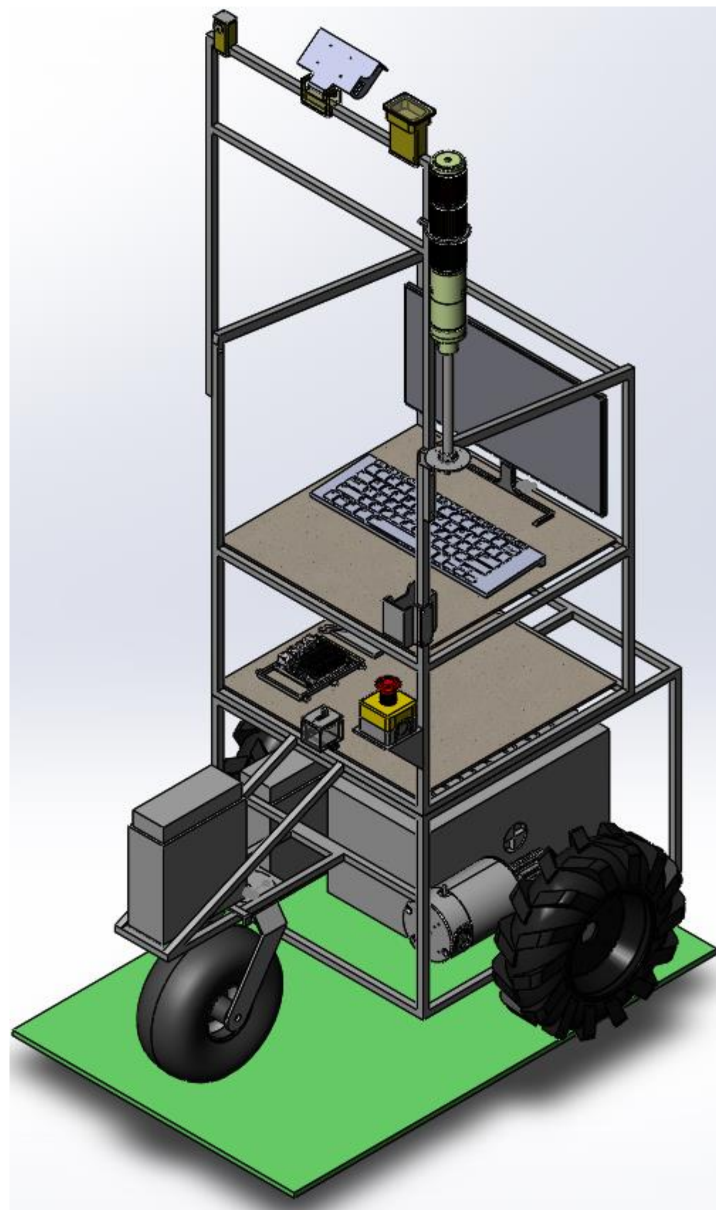


Figure 3. Robot Chassis

The height of the camera mounting hardware was designed to be as tall as possible within the rules in order to give a clear vantage point for the RealSense Depth Camera which was being used for detecting lines and objects. This also allows our GPS unit and digital compass to be placed high and away from the main chassis, minimizing interference in their measurements. The back wheel gives stability to the weight distribution and moves the center of gravity towards the back of the robot, helping to prevent tipping from inclines.

The camera, GPS, compass, and monitor are all secured to the robot via 3D printed plastic. These custom housing and mounting hardware pieces were designed in Solid Works and printed on our department printers. The movement of the robot over rough terrain resulted in the breaking or cracking of some of these clamps, so additional clamps and stronger supports were added to stabilize these parts.

The payload is designed to fit on the bottom shelf of the robot near the motor controllers, keeping much of the weight low to prevent rocking. The batteries were also placed near the ground to keep a low center of gravity. During initial testing, we found that the robot would tip during sudden stops, especially on steeper inclines. As a result, one of the batteries was re-located from its initial design location to over the back wheel which effectively shifted the center of gravity further behind the drive wheel axis. The pneumatic tires provide a reasonable but firm suspension for the robot.

To weatherproof the robot, a thin window heat shrink wrap was used to cover the outside of the robot. This wrapping is resistant to light rain and keeps the electronic components dry. On the side of this covering, holes were cut for the Time-of-Flight sensors to allow these sensors to function properly. On the front of the robot where the payload is to reside exists a covering that will swing open and allow the insertion of the payload. The back also has a door that can be opened to view and adjust the electronics inside and work with the mouse and keyboard. This maintains easy access to the electronic components on the inside of the robot.

Electronic & Power Design

The robot electronics can be broken down into the power supply, the core processing unit (Jetson Nano), peripherals, the sensing units (inputs), the output units (motors and lights), and the Emergency Stop System. The power supply consists of two 12V batteries. The full 24 Volts is applied to a select few components while the rest of the components require 12 volts or less. The Jetson Nano and its peripherals lie at the center of the robot and receive power from a 12 to 5V Buck Converter. All the sensing equipment is linked to the Jetson Nano either through the USB ports or the GPIO pins. Since the buck converter is over 85% efficient one can roughly consider the components receiving +5V as being supplied with half the voltage but twice the current. The motor controller, motors, monitor, and lights get their power directly from the batteries. The emergency stop system has a mechanical and remote-control mechanism that both cut off power to the motor controller directly.

The power requirements of the various components are listed in the table below.

Part	Quantity	Amps/Unit Rating (Max)	Total Amps Drawn from Batteries
Motors	2	8 A	16 A
MDDS30 Motor Controller	1	0.2 A	0.2 A
Monitor	1	1.1 A	1.1 A
Misc Peripherals	1	1.0 A	1.0 A
Items below this line are powered through the buck converter so they only draw half the current from the batteries.			
Jetson Nano	1	4.0 A	2.0 A
VL53L0X Time of Flight	2	0.02 A	0.02 A
QMC5883L Compass	1	0.02 A	0.01 A
ZED F9P GPS	1	0.1 A	0.05 A
Depth Camera	1	0.4 A	0.2 A
Total			**20.58 A**

Table 1. Power Consumption by Component

The robot uses two batteries rated for 20 Amp-hours. Adding up all of the power consumption from these components amounts to 20.58 Amps. Our maximum runtime under this load is calculated as follows:

$$\frac{20\ Amp * hours}{20.58\ Amps} = 0.972\ hours = 58.3\ minutes$$

This is more than enough time to run the AutoNav course multiple times on a single battery charge. It is worth noting that this is the worst-case scenario. The main power draw is the motors, and they are rated for 8 A under maximum load. It would be very rare for the motors to actually draw this kind of current from the batteries and they certainly would not draw that much current for an extended period of time. Our expected battery life should be much longer than 1 hour. The team has been able, to test and run the robot for up to 4 hours without the batteries even starting to die. In all the testing done over the year, the team has never actually seen the batteries run low or experienced a power outage.

List of key electronic components

Nvidia Jetson Nano

The Nvidia Jetson Nano serves as the primary controller for the robot. It is connected to the RealSense Camera over USB 3.0, the time-of-flight sensors and compass using I2C, the motor controller using a USB to UART cable, the GPS using UART over the GPIO pins, and the signal tower using the GPIO pins directly.

Motor Controller

The motor controller is an MDDS30 Cytron SmartDriveDuo-30. It was particularly well suited for our application since it accepts commands by several different means including RC (Radio Controlled) Mode

and a Serial UART connection. The team can easily manually drive the robot into position and test things in RC mode and then when the robot is switched to autonomous mode the controller uses the Simplified Serial Mode.

Intel RealSense D435 Depth Camera

The Intel RealSense D435 Depth camera is used to capture color images for line detection, line-following, and pothole detection as well as stereo depth arrays for obstacle detection.

Compass

The compass is an orientation sensor used to aid our transitions between states – particularly the exiting of obstacle avoidance.

Global Positioning System

Our team used the ZED F9P GPS to do our GPS navigation. Our software utilized the GEOPY library to determine the distance that we were from a waypoint. We also utilized complex numbers and phase angles to determine our orientation utilizing the Python "cmath" library.

Time of Flight Sensors

There are two VL53L0X Time of Flight (ToF) sensors mounted on the robot, one on each side. These sensors are used to determine if there is an obstacle on the side of the robot and when detected are used to follow around the obstacle.

Safety Devices

The robot has both wired and wireless emergency stops. The wired emergency stop consists of a large red mushroom pushbutton controlling the flow of power to the motor controller. Activating the e-stop cuts off the power to the motor controller and immediately brings the robot to a halt. The wireless emergency stop is triggered by a nob on the robot's wireless controller. Upon receiving the signal from the wireless receiver, a servo motor manually flips a switch which also cuts off power to the motor controller. Both E-Stop mechanisms work regardless of whether the robot is in RC mode or Autonomous mode.

The red light on top of the robot functions as a safety light. This light indicates whether the robot is set to the autonomous mode or the manual mode. When the robot is being manually driven the red light is solid and when the robot is set to move autonomously the red light will flash.

In addition to the Emergency Stop mechanisms and the safety light, the software of the robot requires that each piece of software be successfully initialized and functioning for the robot to operate autonomously. If any software node shuts down or does not initialize properly, all other nodes will also be shut down to prevent unpredictable behavior from some nodes not being operational.

Software Strategy and Mapping Techniques

We utilized the Robot Operating System (ROS) framework for connecting the various components of our robot. Each of our input and output devices is initialized as a ROS node. We also have the main node which is set up as a finite state machine (FSM) and functions as the primary controller for the robot. The overall architecture of the main robot controller is shown in Figure 4. We did not implement any mapping algorithm; rather, our robot navigates the course solely using the sensors and logical state transitions.

Obstacle detection and avoidance are accomplished by utilizing the Intel RealSense Depth camera for detecting obstacles immediately in front of the robot, and then the dual time of flight sensors for navigating around detected obstacles. Proper exit conditions are then determined by either detecting a suitable line to resume following or determine that the robot is angled properly when GPS following.

Our obstacle detection node is continually running and outputs when there is an obstacle in the path of the robot as well as when the path is clear. When the main robot controller receives a message that an obstacle is present, it transitions to an obstacle avoidance state respectively to the current mode that the robot is in (line following or GPS following). It then reads the distance information from the time-of-flight sensors to drive around the obstacle(s). While navigating around obstacles in line following mode, the robot performs line detection which utilizes a Probabilistic Hough Transform to determine if a suitable line is present to follow. If a suitable line is detected, the robot transitions back to the line following mode. While navigating around obstacles in GPS mode, the robot continually checks for an acceptable angle between GPS position readings to trigger a return to GPS following.

Our robot takes a strategy of using a Finite State Machine (FSM) to navigate the obstacle course. Rather than planning and mapping a path through the obstacle course, the robot will handle obstacles in the course as it encounters them. This strategy of focusing on what is directly in front of the robot helps to simplify the software and avoid complex computations.

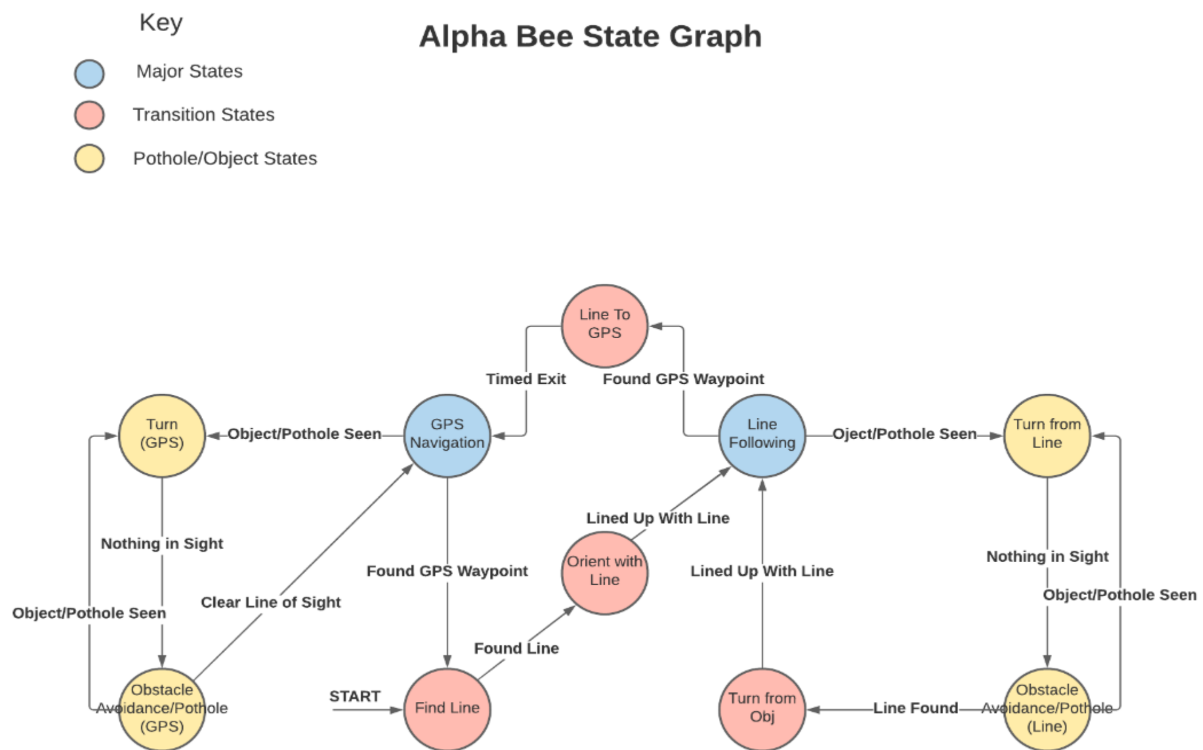


Figure 4. Alpha Bee State Graph

The FSM has four major states or modes of operation where it is moving based on information from a particular sensor. In the Line Following mode, the robot uses the camera to find the line and follow it until some disturbance from the surrounding takes precedence such as an obstacle. In the GPS Navigation mode waypoints are followed based on heading until an obstacle is found in the way or the waypoint is reached. There are two Obstacle Avoidance modes: one for when line following and one for when navigating by GPS. Each of these will follow along the side of the obstacle until the robot can return to the previous state of following a line or GPS waypoints.

In between the major states are several transition states where the robot performs specific maneuvers based on surrounding sensor data to move smoothly into the next main state. For example, when an obstacle is seen, the robot turns at a set speed and direction until the obstacle is no longer in front of it and then begins to follow the side of the obstacle. Likewise, when following an obstacle, if the line is seen the robot will turn until it is properly oriented with the line and then start following the line again.

Failure Modes, Points, and Resolutions

The main failure in the software that could arise would be if the state machine gets out of sync with the course. If a state is entered on accident (due to an invalid sensor measurement or inadequate logical processing of the sensor data), the flow of the run could be disrupted, resulting in undefined behavior. For example, if the robot thought it saw an obstacle when there was no obstacle in front of it, that would cause the robot to turn away from the line it was following. We attempted to minimize the effect of these types of errors by keeping all state transitions based on environmental stimuli and by requiring multiple transition flags before changing states. The other event that is most likely to cause a software fault is if one of the cues to leave a state is missing causing the robot to stay in the same state rather than leave when it should. We minimize this kind of failure by carefully setting the sensor read rates and keeping the speed of the robot at a level where we should always be able to detect a noteworthy change in the environment. One last kind of failure that could come happen is for one of the ROS nodes to shut down unexpectedly. This is handled by the software because ROS can detect if a node shuts down and attempt to restart the node (which usually takes about a tenth of a second). This keeps the robot from having to finish the run without an important piece of software such as the node that detects objects.

The failure points in the hardware are primarily concerned with communication between the sensors and the Jetson Nano. In particular, the I2C busses for the Time-of-Flight sensors and the Compass. Our team found that because of the long length of the communication busses, they were prone to fail due to electromagnetic interference from the motor controller. This risk has been alleviated in a couple of ways. First, pull-up resistors were installed on the lines on the side of the sensors to hold the lines high. We found that having pull-up resistors on both sides of the I2C line helped support the bus from interference. Also, we shielded the wires with aluminum foil to further prevent interference. Lastly, in the software, we detect errors on the I2C bus and restart the connection whenever it goes down.

Other failures could come from loose connections in the wiring. These were mitigated by good solder joints and covering the solder joints with electrical tape and heat shrink wrap. The connections to the Jetson GPIO pins have been carefully checked to ensure a secure connection. Most wiring such as the USB and UART lines have been very stable and not prone to errors.

To prevent failures as much as possible both during development and at the competition, the robot was designed with a very modular approach in mind. This helps in debugging and allows for replacing various components or software nodes should something fail. This was accomplished in part by keeping all sensors distinct from one another. Any single hardware component can easily be replaced with a substitute. In using ROS for the software management of our system, each distinct software component is given its own node. All the nodes communicate by predefined communication channels and protocols. Not only does this allow each node to be tested individually, but it also makes it easy to debug and find failures.

The robot was tested incrementally throughout the entire design process. This was accomplished both using simulations with ROS bags (recorded camera and sensor data that can be played to simulate how the robot would respond) and through real-time tests and small demo runs. The first tests done on a piece of software were usually with ROS bags. For both software and hardware, most testing was then done in our lab with conditions that approximated as closely as possible the expected conditions on the course. Once this phase was completed, the robot was often taken outside to a makeshift obstacle course and allowed to run on the course to test various functionalities. The process was followed during the entire design process to facilitate the testing of new code and its integration with the existing system. Since we have been doing real-world testing since October, we have found and eliminated many bugs.

The robot was designed to be as safe as possible for all bystanders. The primary means of these safety considerations were in the form of the Emergency Stop Mechanisms, which shut off power to the motor controller at the user's discretion. Additionally, the robot is also made to move at a slow enough speed that it will not pose a significant safety hazard to anyone in the nearby vicinity. Different speeds were even used for the various navigational modes to ensure that the robot was never traveling so fast that it lost control.

Employed Simulations

For much of our original vision algorithm development, we recorded ROS bag files of our robot manually driving through an obstacle course. We then played back the bags and ran our algorithms on them to test their performance. Due to the nature of our approach, we were unable to run complete simulations of overall robot performance in a virtual environment. We did however extensively test our robot outside on an obstacle course that we created.

Performance Testing

Subsystems and individual components of the robot were tested by utilizing the ROS framework to run a single node at a time. In this way, a script could be written in each ROS node to run a basic test of a hardware component. For the vision nodes, ROS bag files were used to provide camera-input without the other subsystems such as the wheels running. All the sensing nodes were also able to be tested by driving the robot manually and allowing the robot to process real sensor input in real-time. This allowed realistic conditions to be used in evaluating the performance of various algorithms and techniques. Our team found that driving the robot and allowing the robot to process data in real-time from actual data was much more helpful than simulations with the ROS bag files. The reason for this was that sometimes the incoming frame rate would be different in the simulation, or the lighting conditions or camera angle

had changed. Driving the robot around manually allowed for very realistic conditions to be tested and thus we were able to verify the accuracy of our algorithms.

The full robot was tested on a small outdoor obstacle course modeled after the real course on which the robot was designed to compete. Various conditions were assessed and tested to ensure consistency in the robot's response to obstacles, lighting, and ground gradient. To aid with testing, the tower lights were used to indicate which state the robot was in throughout the testing process, giving the team an accurate, up-to-date knowledge of which state the robot was currently in and why it acted the way it did.

Initial Performance Assessments

Alpha Bee is currently working very well. It runs with an average speed of about 1.8 miles per hour carrying a full load and can navigate consistently by following a line and following GPS waypoints. In both line-following and GPS-navigation mode, Alpha Bee can gracefully handle obstacles in its way by navigating around them. Obstacles are detected at a dynamic distance, allowing just-in-time detection and avoidance. The GPS has an observed accuracy of 1 meter allowing adequate detection of the course waypoints. The Time-of-Flight sensors allow objects on the side of the robot to be detected up to 1.5 meters away which allows the robot's obstacle following to be about half a meter away on average. Overall, Alpha Bee is ready to compete in the competition in June.