Spring 2017

Intelligent Ground Vehicle Competition

Austin R. Tyler
The University of Akron, art42@zips.uakron.edu

Chris R. Estock
The University of Akron, cre20@zips.uakron.edu

Johnathan P. Johenning
The University of Akron, jpj17@zips.uakron.edu

Garrett W. Chonko
The University of Akron, gwc11@zips.uakron.edu

Allen C. Gilleland
The University of Akron, acg55@zips.uakron.edu

IGVC Design Report

Group 14

Project Leader: Garrett Chonko

Archivist: Christopher Estock

Sensor Specialist: Allen Gilleland

Hardware Manager: Johnathan Johenning

Software Manager: Austin Tyler

Faculty Advisor: Dr. Jay Adams

5/1/2017

Table of Contents

List of Figures

List of Tables

Abstract

The Intelligent Ground Vehicle Competition (IGVC) draws teams from various universities to compete in the annual autonomous vehicle challenge at the Oakland University campus. To compete, a vehicle must be fully autonomous and can navigate a course designated by various obstacles and painted white lines. Some design challenges are motor control, navigation, environment sensing and safety. A complex navigation system will utilize several tools including a high-precision differential GPS. The vehicle's surroundings will be mapped using a combination of Light Detection and Ranging (LiDAR) and computer-vision enabled imaging. To comply with IGVC rules, the vehicle must also follow several safety requirements such as physical and wireless emergency stop, safety lighting, and the ability to assume manual control. By fulfilling these design challenges, the design team is seeking to compete in the 2017 Intelligent Ground Vehicle Competition.

- Ground Vehicle
- Traverse course autonomously to provided destination
- Maintain speeds between 1-5 MPH
- Implement sensors and software for object detection and avoidance

Intelligent Ground Vehicle (IGV)

1. Problem Statement

1.1 Need

Each year, Oakland University hosts the Intelligent Ground Vehicle Competition (IGVC). The competition offers a multidisciplinary design experience for engineering students. The autonomous guidance and obstacle avoidance technologies employed in the competition have industry applications in military mobility, transportation systems, and manufacturing. The vehicle must autonomously navigate a closed circuit using lane following and waypoint navigation and must avoid barrel obstacles. The intelligent ground vehicle (IGV) needs to comply with all competition rules.

[ART]

1.2 Objective

The objective is to create a ground vehicle to compete in the IGVC 2017. The vehicle will be autonomous and conform to competition rules including minimum and maximum speed, mechanical and wireless emergency stop, safety lighting, and must carry a payload of approximately twenty pounds from start to finish. In order to pass qualification, the vehicle must also demonstrate the ability to follow lanes, avoid basic obstacles, and navigate to set waypoints.

[CRE]

1.3 Background

1.3.1 *Patent Search*

There are numerous patents which describe methods and systems for autonomous vehicle navigation and obstacle avoidance. These patents cover a range of techniques of various complexities. The systems in each of the patents get input from an imaging device. The input is processed and used to adjust vehicle trajectory and speed.

US Patent 7587260 describes a system for navigation in an autonomous vehicle. The system uses an event timing loop and adjusts translational and rotational velocities to avoid obstacles detected in its path. The system uses at least one perception sensor, at least one locomotor, and a system controller. The process for adjusting rotational velocity uses a proportion of the current rotational velocity less the proportion of the range to the nearest obstacle. The adjustment of translational velocity when an obstacle is within the event horizon is based on a promotion of the range to that obstacle. When no obstacle is detected within the event horizon, the translational velocity is adjusted to a factor of the maximum allowable speed [1].

US Patent 5675489 describes a system and method for estimating lateral position of a vehicle within a marked lane. The system uses a camera and a digitizer to create a perspective-free image comprised of rows and columns of pixels. The image is processed by summing the columns to create a single row called the scanline profile. The measured scanline is compared with the stored scanline profile of proper lane centering to determine the lateral offset of the vehicle from the center of the lane [2].

US Patent 8050863 describes a navigation and control system for autonomous vehicles. The system uses a 3D image capturing device to create a 3D contour image of the field in front of the vehicle. The image data and a GPS are used to detect obstacles and plan the path and speed of the vehicle. The path planning module adjusts the steering of the vehicle and the speed planning module adjusts the acceleration of the vehicle [3].

[ART]

1.3.2 *Article Search*

Michael L. Nelson outlines the importance of code reusability and object oriented programming techniques when creating a control architecture for an autonomous vehicle. Nelson suggests that all Autonomous Vehicles (AV) should have a clear and distinct goal, can handle any problems that can be reasonably anticipated, and should keep several fallback positions if an issue arises. Object oriented programming should be used to make these components of the system work together by complementing each other's operation to complete the final task. Nelson later recommends the use of a Strategic-Tactical-Execution Software Control Architecture (STESCA), which focuses the design of each software component on *what* it should accomplish rather than *how* it should be accomplished [4].

Ming Huang et al. conducted research on an intelligent vehicle based highway driving system that synthetically considers the current traffic conditions and corrects pathing for a single vehicle or set of vehicles moving in a group. Researchers created an intelligent vehicle using an ARM based platform combined with a DSP controller, power conversion module, motor drive module, ZIGBEE wireless communication module, CMOS digital camera, ultrasonic sensor, rotary encoder and external memory. The DSP controller used several algorithms outlined in the article to process data received from the CMOS digital camera to create a navigable map of the area to the front of the vehicle [5].

B. Dumitrascu et al. designed an algorithm for trajectory tracking and obstacle avoidance for autonomous vehicles. This algorithm consists of both global trajectory tracking and local trajectory tracking with obstacle avoidance. In general, the autonomous vehicle follows the global trajectory while adjusting on a short-range level with the local trajectory algorithm. Included in the overall algorithm are concepts such as a "sensitivity bubble," which is used to determine if an obstacle is blocking the global trajectory. If an obstacle is detected within the sensitivity bubble, a new path is calculated that contains the minimum density of obstacles. The vehicle follows this new route until the end goal is visible or a new obstacle is detected, then it is repeated [6].

[CRE]

1.3.3 *Other Source Search*

The Intelligent Ground Vehicle Competition has been attempted by a previous design team at the University of Akron in the 2012-2013 academic year. The team used a camera, a compass, a GPS, and LiDAR to get positioning data to navigate the course. The team outlined a method of using weighted data to determine the best path for the vehicle to take [7].

[ART]

1.4 Marketing Requirements

The marketing requirements for the vehicle are listed in Table 1. Fulfilling these requirements will ensure the vehicle can maneuver on the IGVC course and complete the objectives specified in the IGVC official rules.

Table 1. Marketing Requirements

Marketing Requirements	
1	The vehicle must comply with the IGVC rules and regulations.
2	The vehicle will be able to perform in off-road conditions
3	The vehicle must move at a steady pace.
4	The vehicle must have a tight turning radius.
5	The vehicle will be a safe and reliable vehicle
6	The vehicle will autonomously follow lanes, avoid obstacles, and navigate to waypoints.
7	The vehicle must be able to be shipped to competition site.
8	The vehicle must be able to represent University of Akron in the 2017 IGVC competition.

[GWC, ART]

1.5 Objective Tree

The objective tree in Figure 1 separates the requirements of the vehicle into three categories: mechanical body specifications, system capability specifications, and safety specifications. The specifications are dictated by the official rules of the Intelligent Ground Vehicle Competition.

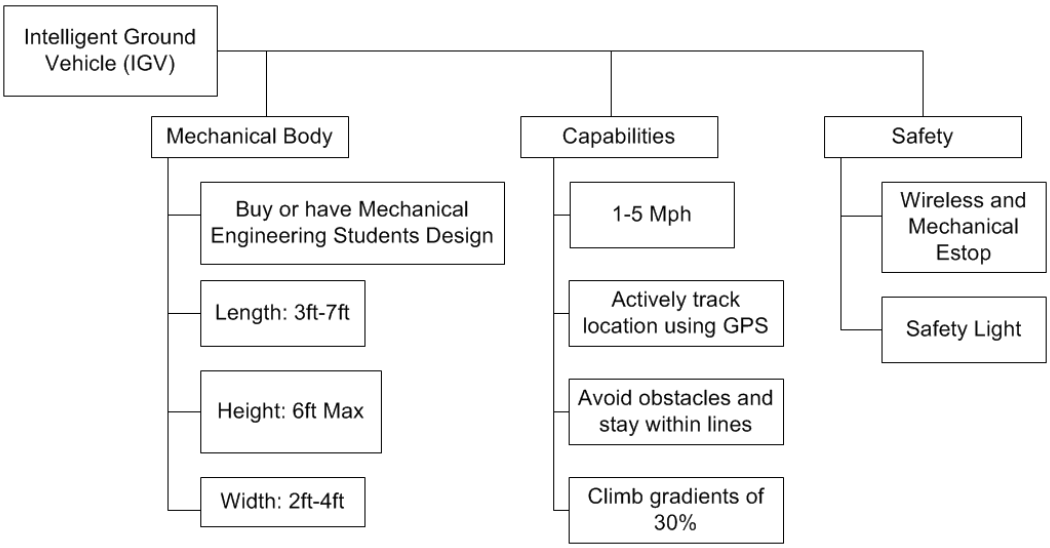


Figure 1. Objective Tree

[GWC]

2. Design Requirements Specification

The design requirements in Table 2 define quantitative, verifiable goals that the finished vehicle must meet. These requirements are chosen to meet the marketing requirements defined in Table 1.

Table 2. Design Requirements

Marketing Require-ments	Engineering Requirements	Justification
1,2	Vehicle must be able to Traverse a 30-degree incline.	This is the steepest hill grade on the course that the Intelligent Ground Vehicle will encounter.
1,5	Vehicle must have a red emergency stop button no less than one inch in diameter, and must have a hardware-based wireless emergency stop.	Safety specification that must be met to ensure the safety of spectators and Intelligent Ground Vehicle Competition officials.
1,3,5	Vehicle must have a maximum speed of 5 mph and a minimum speed of 1 mph; the maximum speed must be hardware enforced.	To ensure the systems remains under control and does not pose a danger to spectators or Intelligent Ground Vehicle Competition officials.
1,8	Vehicle must be a grounded and propelled by direct mechanical contact to the ground.	To ensure conformance with the Intelligent Ground Vehicle Competition rules.
1,7	Vehicle must not exceed six (6) feet in height except for the emergency stop antenna.	To ensure conformance with the Intelligent Ground Vehicle Competition rules.
1,4,7	Vehicle must have a minimum length of three (3) feet and a maximum length of seven (7) feet.	To allow the Intelligent Ground Vehicle to travel around obstacles in its path and to ensure conformance with the Intelligent Ground Vehicle Competition rules.
1,4,7	Vehicle must have a minimum width of two (2) feet and a maximum width of four (4) feet.	To allow the Intelligent Ground Vehicle to travel around obstacle in its path.
1,5	Vehicle must indicate states of operation, via onboard lights.	To insure the safety of spectators and other Intelligent Ground Vehicle Competition officials.
1,6	Vehicle must be able to autonomously detect and avoid obstacles, and travel to GPS waypoints while detecting and traveling within a lane defined by white lines painted on the ground.	This is the basis for the Intelligent Ground Vehicle Competition.
1,8	Vehicle must be able to carry a 20-pound payload securely for the duration of the run.	To ensure conformance with the Intelligent Ground Vehicle Competition rules.

[ART, GWC]

3. Accepted Technical Design

The technical design of the intelligent ground vehicle is detailed starting with level 0 (most broad) which encompasses all areas of design. Subsequent design levels are separated into power and motors (dubbed hardware), sensors, and software and show more detail about the modules designed for the vehicle.

The level zero block diagram in Figure 2 and functional requirements in Table 3 detail the inputs and outputs for the intelligent ground vehicle. The system will operate on 24-volt power sources. The image data provides information about the environment surrounding the vehicle and is used to avoid obstacles and stay within the lane lines. The GPS data provides accurate position information for the vehicle and is used to navigate to waypoints and to map objects detected by the image sensors. The two emergency stops fulfill the safety requirements for emergency stopping. The vehicle has a status light that indicates when the vehicle is powered on and when it is operating autonomously. This fulfills additional safety requirements. The diagnostic output will provide users with indication of what the vehicle is seeing and what decisions it is making. The dual motor drives are used to drive the motors that propel and steer the vehicle.

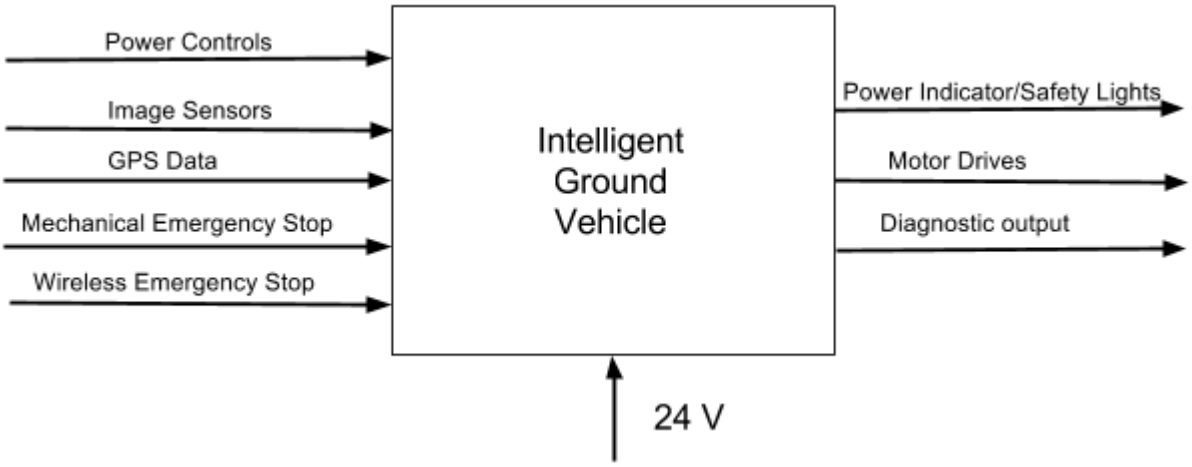


Figure 2. Level 0 Block Diagram

Table 3. Functional Requirements Level 0

Module	Intelligent Ground Vehicle
Inputs	• Power Control • Image Sensors • Global Positioning System • Mechanical Emergency Stop • Wireless Emergency Stop • 24V Battery
Outputs	• Power Indicator/Safety Lights • Motor Drives • Diagnostic Output
Functionality	• The vehicle will navigate an Intelligent Ground Vehicle Competition course by following lanes, avoiding obstacles, and using waypoint navigation.

3.1 Motor Block Diagrams and Functional Requirement Tables

The level one block diagram in Figure 3 and functional requirements in Table 4 detail the inputs and outputs for the intelligent ground vehicle. The system will operate on 24-volt portable power sources. In addition, the input power will be able to protect the motor driver from inrush current, transient voltage spikes from the motors, as well as the ability to cutoff the motor through a mechanical and wireless implementation.

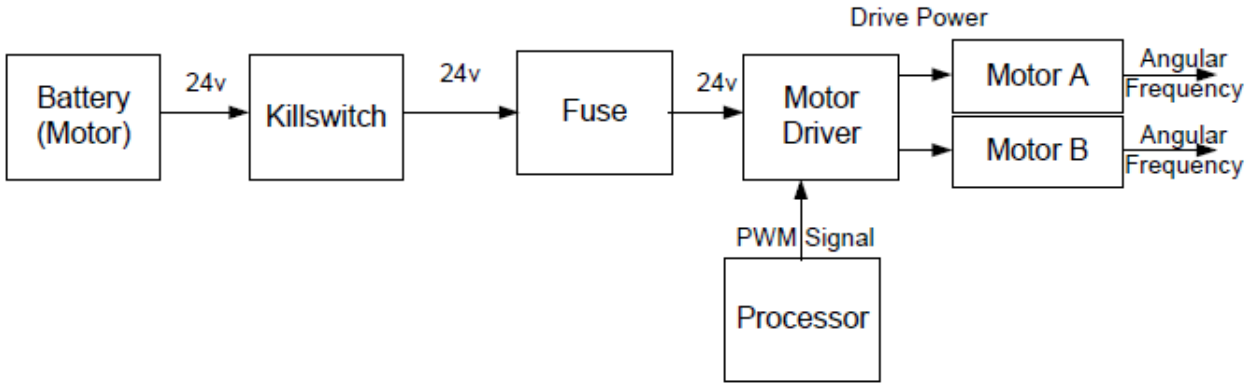


Figure 3. Block Diagram Level 1 (Motors)

[JPJ]

Table 4. Functional Requirements Level 1 (Motors)

Module	Intelligent Ground Vehicle: Battery
Inputs	• None
Outputs	• 24V
Functionality	• Provides a source of energy that will later be applied to the motors.

Module	Intelligent Ground Vehicle: Kill Switch
Inputs	• Battery Voltage: 24V
Outputs	• Controlled 24V
Functionality	• For safety; to manually stop the vehicle when needed.

Module	Intelligent Ground Vehicle: Fuse
Inputs	• 24V from Kill Switch.
Outputs	• Protected 24V that will not exceed the rated value.
Functionality	• To protect the attached devices from a potentially damaging amount of current.

Module	Intelligent Ground Vehicle: Motor Drive
Inputs	• Control signal from processor. • Power.
Outputs	• Motor A UART signal. • Motor B UART signal. • Speed sensor output.
Functionality	• The vehicle will input transfer power from the battery to the prospective motor via the motor driver with which the computer communicates.

Module	Motors (A/B)
Inputs	• Controlled Power Delivery from Motor Driver
Outputs	• Torque to the wheels.
Functionality	• To move the vehicle to the designated location at the requested speed.

[GWC, JPJ]

Figure 4 Shows the Level 2 Motor block diagram and Table 5 shows the functional requirements table for the motor level 2 block diagram.

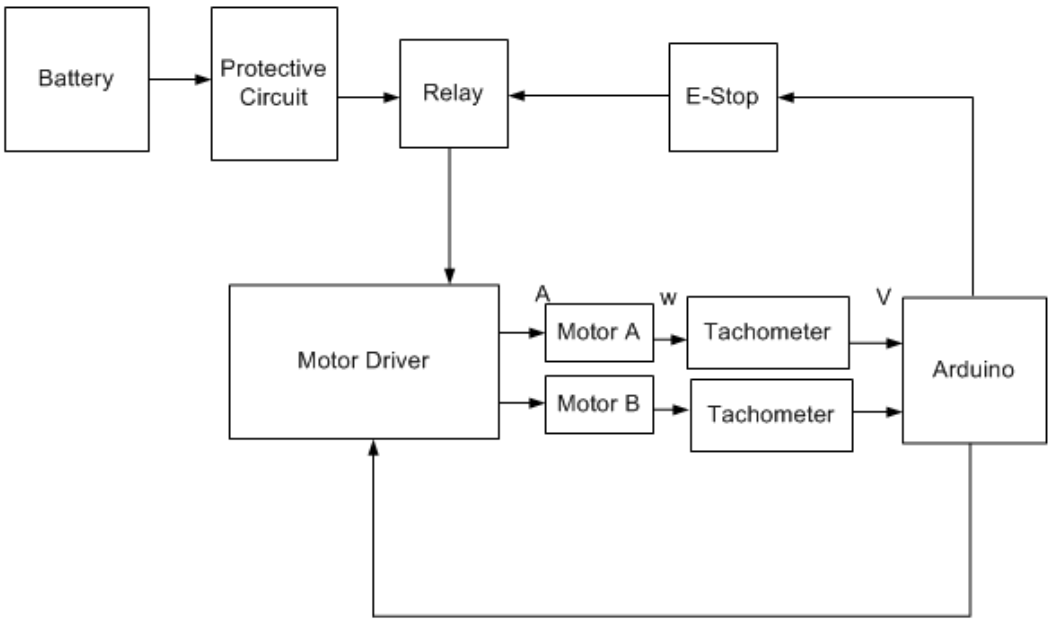


Figure 4. Block Diagram Level 2 (Motors)

[JPJ, GWC]

Table 5. Functional Requirements Level 2 (Motors)

Module	Intelligent Ground Vehicle: Battery
Inputs	• None
Outputs	• 12V
Functionality	• Provides a source of energy that will later be applied to the motors.

Module	Intelligent Ground Vehicle: Protective Circuit
Inputs	• Battery Voltage: 12V
Outputs	• Controlled 12V
Functionality	• Protect the devices and wires from overheating and transient voltage spikes from the naturally inductive motors.

Module	Intelligent Ground Vehicle: Relay
Inputs	• Controlled 12V for load • Relay control signal
Outputs	• Controlled 12V
Functionality	• For safety; to stop the vehicle when there is an electrical or mechanical failure.

Module	Intelligent Ground Vehicle: Kill Switch
Inputs	• Digitally High Signal (pulled to ground) from microcontroller
Outputs	• When Closed: signal from microcontroller which is high/low depending on whether the wireless receiver is communicating • When open: leaves the signal at the relay to be a low voltage which breaks continuity between load side input and output.
Functionality	• To provide a mechanical based emergency stop when the vehicle malfunctions • To coincide with the IGVC rules

Module	Intelligent Ground Vehicle: Motor Driver
Inputs	• Communications with the processor. • Power that has been conditioned for the motor driver.
Outputs	• Motor A UART signal. • Motor B UART signal. • Speed sensor output.
Functionality	• The vehicle will input transfer power prospective motor via the motor driver with which the computer communicates.

Module	Motors (A/B)
Inputs	• Controlled Power Delivery from Motor Driver
Outputs	• Torque to the wheels.
Functionality	• To move the vehicle to the designated location at the requested speed.

[JPJ]

3.2 Sensor Block Diagrams and Functional Requirement Tables

The sensor network of the intelligent ground vehicle are the tools needed to achieve successful autonomous travel in-route to the vehicles provided destination. The sensors listed below will ensure that the three goals of travel – course mapping, object detection and speed monitoring, are realized through integration in software. The block one diagram for the sensor

network is provided in Figure 5 with the functional requirements for each block detailed in Table 6.

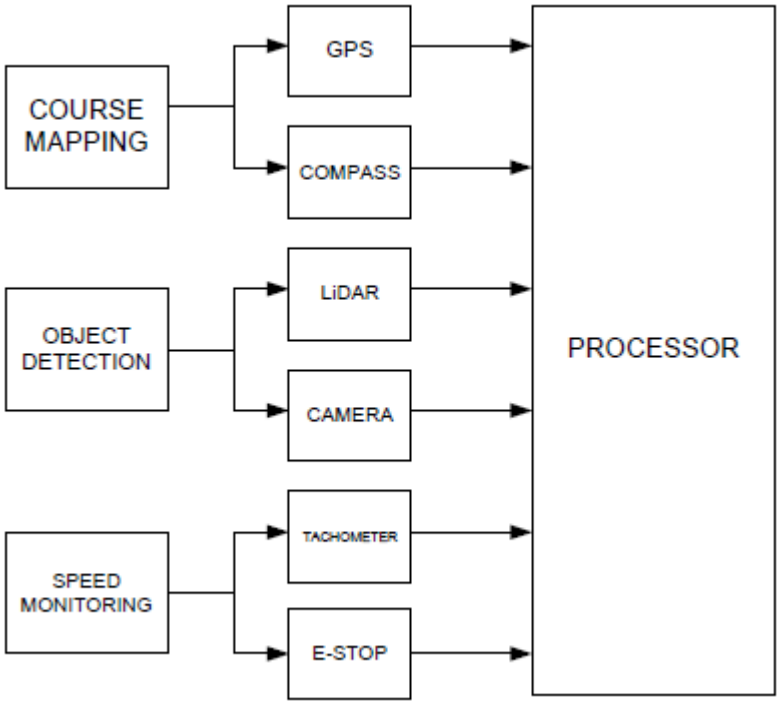


Figure 5. Block Diagram Level 1 (Sensors)

Table 6. Functional Requirements Level 1 (Sensors)

Module	Course Mapping
Inputs	• None
Outputs	• Geographical course information
Functionality	• The course mapping block describes the course environment and location in the world.

Module	GPS
Inputs	• Course Mapping
Outputs	• GPS coordinates
Functionality	• The GPS system connects to access satellites to provide real time location information to the vehicle for processing.

Module	Compass
Inputs	• Course Mapping
Outputs	• Direction of vehicle
Functionality	• The compass complements the GPS by providing the vehicles facing direction. The compass will increase precision when the vehicle is stationary or at the slower speeds required by competition rules.

Module	Object Detection
Inputs	• None
Outputs	• Position of objects along course
Functionality	• The object detection block recognizes objects or obstructions along the course.

Module	LiDAR
Inputs	• Object Detection
Outputs	• Distance of an object in reference to the vehicle.
Functionality	• The LiDAR sends a scanning pulse laser, like an infrared, and measures the return time of the pulse to determine the distance of an object to the LiDAR. The LiDAR will act as a bubble around the vehicle to notify it of any obstruction and to avoid accordingly.

Module	Camera
Inputs	• Object Detection
Outputs	• Digital Image
Functionality	• The Cameras will be mounted to the top of the vehicle pointed down towards the course. An image will be captured and processed in real time to realize the course boundaries, lane detection, potholes and flags.

Module	Speed Monitoring
Inputs	• None
Outputs	• Speed of vehicle
Functionality	• The speed monitoring block controls and monitors the speed of the vehicle.

Module	Tachometer
Inputs	• Speed Monitoring
Outputs	• Rotational Speed of wheels
Functionality	• The tachometer observes the rotational speed of the wheels on the vehicle to maintain a speed between 1-5 MPH required by the competition.

Module	E-Stop
Inputs	• Speed Monitoring
Outputs	• Emergency motor shutdown
Functionality	• The E-Stop shuts down power to the motor in case of an emergency stop.

Module	Processor
Inputs	• GPS • Compass • LiDAR • Camera • Tachometer • E-Stop
Outputs	• Motor Controller
Functionality	• The processor accepts information from the GPS, Compass, LiDAR, Camera, tachometer and E-Stop for processing and filtering. The information is sent to the motor controller for traversing the course successfully.

The Sensor block diagram is later broken down to show power supplied and to incorporate the motors for recognizing the speed of the vehicle. A step down is applied to after the battery to ensure proper power is supplied to each sensor accordingly. An antenna is shown connected to the GPS to properly communicate with GPS/GNSS satellites wirelessly. Speed is detected from each motor individually and read by the tachometer. That analog signal is then sent through an analog to digital converter. The digital speed is then sent to the processor to maintain appropriate speed throughout the course. The following is represented in Figure 6 and discussed further in Table 7.

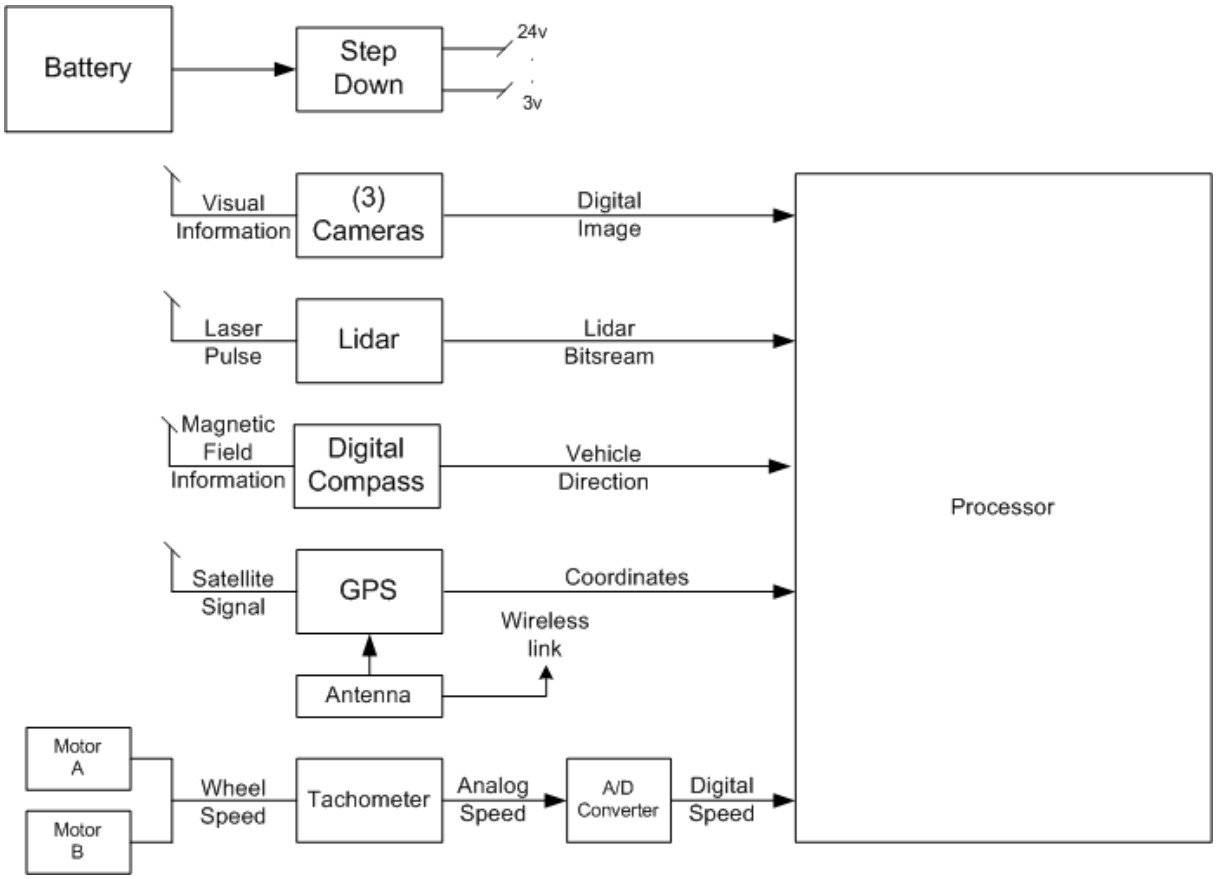


Figure 6. Block Diagram Level 2 (Sensors)

Table 7. Functional Requirements Level 2 (Sensors)

Module	Battery
Inputs	• None
Outputs	• 24v
Functionality	• Provides a source of energy that will later be stepped down and applied to sensors.

Module	Step Down
Inputs	• 24v
Outputs	• 12v, 3v, or any lesser voltage.
Functionality	• The step down from 24v will provide proper power to the sensors.

Module	(3) Cameras
Inputs	• Battery • Visual information
Outputs	• Digital Image
Functionality	• The (3) Cameras will be positioned one on each side of the vehicle and one in the front for line/pothole detection. The digital image will be sent to the processor for detection.

Module	LiDAR
Inputs	• Battery • Laser Pulse
Outputs	• LiDAR Bit stream
Functionality	• LiDAR (Light Detection and Ranging) is a sensor that employs a pulsed laser to measure or map an environment at 180 or 360 degrees at distances up to 80m. The LiDAR readings will be sent to the processor for object detection.

Module	Digital Compass
Inputs	• Battery • Earth's Magnetic Field
Outputs	• Direction of vehicle
Functionality	• The digital compass will give the exact direction the vehicle is facing at a given time. The digital aspect of the compass will eliminate any magnetic field interference from the motor.

Module	GPS
Inputs	• Satellite Signal • Antenna
Outputs	• GPS Coordinates
Functionality	• The GPS will send the vehicle's exact coordinates to the processor and compare it to the competition's waypoint (for extra points). After that calculation, the vehicle will maneuver the course to arrive at the waypoint.

Module	Antenna
Inputs	• Wireless Signal
Outputs	• GPS
Functionality	• The antenna receives information from GPS satellites and sends it to the GPS hardware.

Module	Tachometer
Inputs	• Motor A wheel speed • Motor B wheel speed • Battery
Outputs	• Analog Speed
Functionality	• The tachometer will measure the angular velocity of the wheels.

Module	A/D converter
Inputs	• Analog Speed
Outputs	• Digital Speed
Functionality	• The A/D converter will take the analog information from the tachometer and convert it to digital form for processing.

Module	Processor
Inputs	• Digital Image • LiDAR Bit stream • Vehicle Direction • Coordinates • Digital Vehicle Speed
Outputs	• Motor Driver
Functionality	• The processor will take the information provided by the sensors and apply it to the course mapping and object detection functions discussed in the software block diagram.

The sensor network is later organized to show power and interface connections. After performing loading calculations discussed later a 12V 8Ah (amp hour) battery. The sensor battery and the motor battery are separate to ensure data retention in case of an emergency stop, as in the motors will be shut down, however, power will remain to the sensor network to retain any course information gathered on the vehicles current run of the course. The 12V section will power the Pepperl + Fuchs R2000 LiDAR and the Novatel Propak V3 GPS system. Power will be passed from the 12V battery to a 5V DC-DC converter to provide power to a powered USB hub. The USB hub will have 5 outputs providing power and information transfer for the Logitech

C920 HD Pro webcams and the Arduino Mega. It should be noted that the cameras were changed from 3 to 1 when the LiDAR had been implemented and synched. Information will also be processed through the USB hub from the GPS system after running through a serial to USB converter. The Arduino MEGA 2560 will handle the power and processing of the digital Compass, tachometer and wireless Estop commands. The bulk of the processing will be handled by the ASUS laptop onboard the vehicle. The ASUS laptop accepts TCP protocol packets via an Ethernet connection from the Pepperl + Fuchs LiDAR. Information from the GPS, web cameras and Arduino MEGA 2560 are fed through a USB connection to the ASUS laptop. The sensor block diagram is shown in Figure 7 and discussed in Table 8.

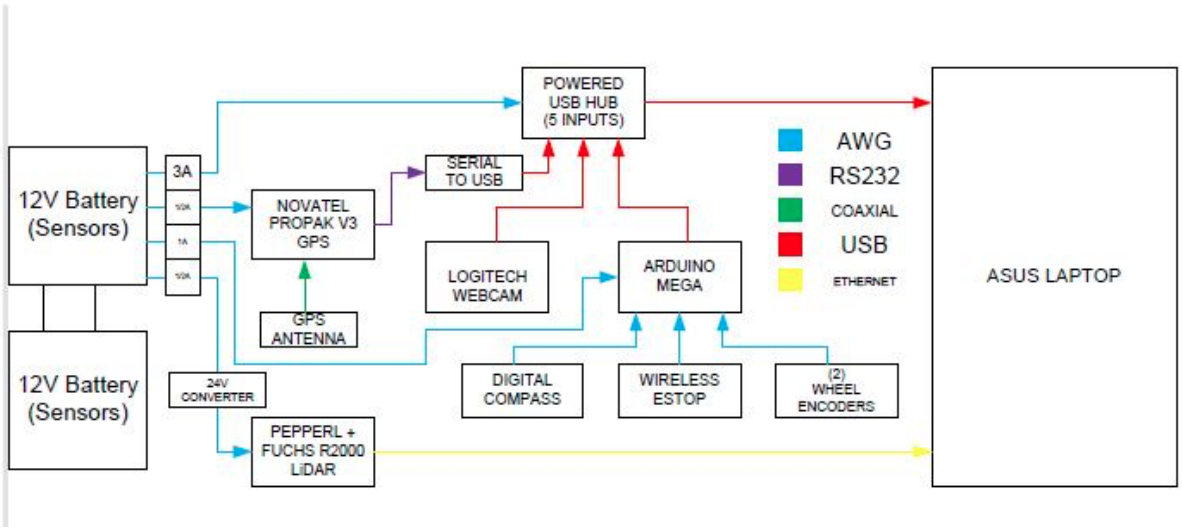


Figure 7. Block Diagram Level 3 (Sensors)

Table 8. Functional Requirements Level 3 (Sensors)

Module	(2) 12V Battery
Inputs	• None
Outputs	• (2) 12v 8aH
Functionality	• Provides a source of energy to the sensor network.

Module	24V Buck Converter
Inputs	• 12VDC
Outputs	• 24VDC
Functionality	• Converts 12VDC to 24VDC for Pepperl + Fuchs Lidar

Module	Powered USB Hub
Inputs	• 12VAC
Outputs	• Power to USB devices
Functionality	• The powered USB hub provides power to the sensors and allows information to be sent to the processor via a USB connection.

Module	Logitech C920 HD Pro webcam
Inputs	• Powered USB Hub • Visual information
Outputs	• Digital Image
Functionality	• The Cameras will be positioned one on each side of the vehicle and one in the front for line/pothole detection angled at 45°. The digital image will be sent to the processor for detection.

Module	Arduino MEGA 2560
Inputs	• Powered USB Hub, Digital Compass, Wheel encoder, Wireless Estop
Outputs	• Processing information to ASUS computer
Functionality	• The Arduino MEGA 2560 will power and process readings from the digital compass, wheel encoders and wireless Estop control.

Module	Digital Compass
Inputs	• Arduino MEGA 2560 • Earth's Magnetic Field
Outputs	• Direction of vehicle
Functionality	• The digital compass will give the exact direction the vehicle is facing at a given time. The digital aspect of the compass will eliminate any magnetic field interference from the motor.

Module	Wheel Encoders
Inputs	• Arduino MEGA 2560 • Wheel rotation
Outputs	• Speed of vehicle
Functionality	• The wheel encoders measure the rotation of the wheels on the vehicle to provide the vehicle's speed to the Arduino MEGA 2560 to maintain the allowed speeds between 1-5 MPH.

Module	Pepperl + Fuchs R2000
Inputs	• (2) 12V battery
Outputs	• TCP protocol packets via Ethernet connection
Functionality	• The Pepperl + Fuchs R2000 is powered via the 12V battery and reads in distances of an object to the vehicle via a rotating IR. The LiDAR is scans in 360° at frequencies adjustable to 10Hz and 50Hz. Data is transmitted via Ethernet connection to the ASUS laptop in the form of TCP protocol packets.

Module	Novatel Propak V3
Inputs	• (2) 12V Battery
Outputs	• RS 232 serial connection to USB
Functionality	• The Novatel Propak V3 provides access to GPS satellites with accuracy of roughly 4cm. Information is sent via the RS232 to USB converter into the USB hub where it is processed by the ASUS laptop using OMNISTAR's GPS software.

Module	Novatel GPS Antenna
Inputs	• Wireless Signal
Outputs	• Novatel Propak GPS
Functionality	• The antenna receives information from GPS satellites and sends it to the GPS hardware.

Module	ASUS laptop
Inputs	• Novatel Propak GPS • Pepperl + Fuchs LiDAR • (3) Logitech C920 HD Pro Webcam • Arduino MEGA 2560
Outputs	• Motor Driver
Functionality	• The processor will take the information provided by the sensors and apply it to the course mapping and object detection functions discussed in the software block diagram.

[ACG]

3.3 Software Block Diagrams and Functional Requirement Tables

The software for the intelligent ground vehicle is responsible for providing the autonomous navigation function required to navigate the IGVC course while avoiding obstacles, following a path, and navigating to waypoints. The software is also responsible for keeping track of the state of operation, providing visual feedback of the operation procedures, and meeting safety specifications. The top-level software block diagram is shown in Figure 8 and the functional requirements are listed in Table 9.

Figure 8. Block Diagram Level 0 (Software)

[ART]

Table 9. Functional Requirements Level 0 (Software)

Module	Intelligent Ground Vehicle Software
Inputs	• Power Button • Image Data • Vehicle Position (from Global Positioning System) • Vehicle Orientation (Compass Data) • Estop State • Autonomous Mode State • Tachometer Speed Data
Outputs	• Power Indicator/Safety Light • Motor Drive Signals (Left and Right) • Estop Trigger Signal
Functionality	• The vehicle will navigate an Intelligent Ground Vehicle Competition course by following lanes, avoiding obstacles, and using waypoint navigation.

[ART]

The software for the intelligent ground vehicle is further divided into five modules as depicted in Figure 9. The functional requirements for each block are listed in Table 10. The Object Detection and Recognition Block uses image and LiDAR data to perceive lines and objects in the real world and determine their location relative to the vehicle. The Course Mapping Block maps those objects in GPS space using information from a compass and GPS. The images, GPS data, and compass angle are time stamped to ensure everything is in sync and account for processing time on the images. The Course Mapping Block generates a 2-D birds eye view of the course. This map is used by the Path Decision Block to determine the best path to take to complete the course or get to the next waypoint. The Path Decision Block outputs a target speed and angle to the Motor Control Block. The Motor Control Block implements the control system for the intelligent vehicle. The target angle and speed are the inputs to the control system and the tachometer speed data and compass angle are the feedback signals. The outputs of the Motor Control Block are the left and right motor control signals for the motor driver. The Motor Control Block uses the control state signal to enable/disable the motors. The Control State Manager handles the emergency stop functions and the safety light operation.

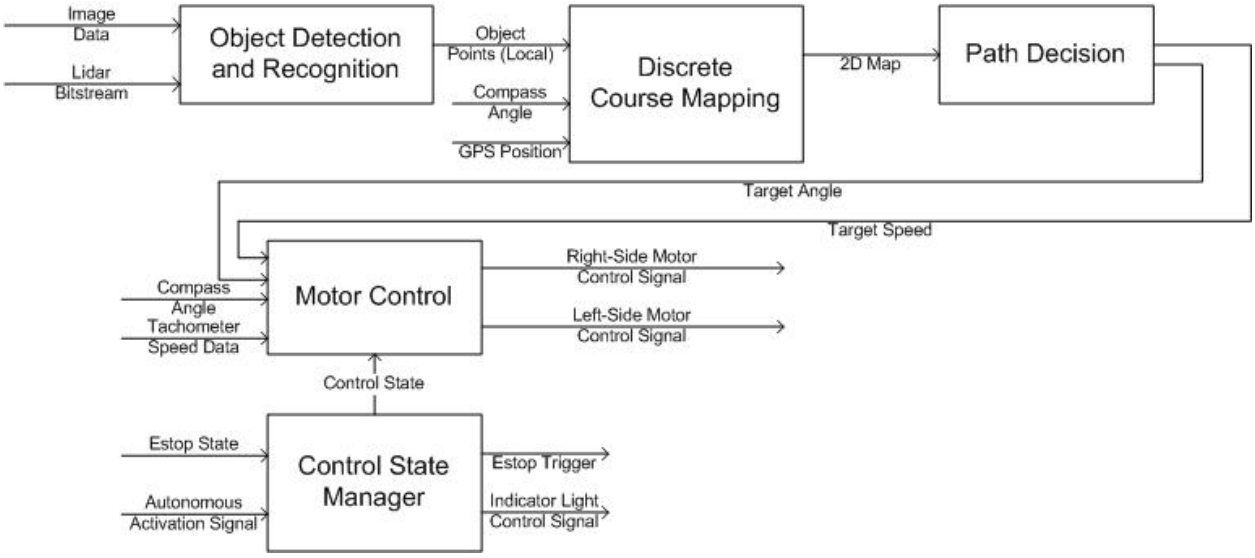


Figure 9. Block Diagram Level 1 (Software)

[ART]

Table 10. Functional Requirements Level 1 (Software)

Module	Object Detection and Recognition
Inputs	• Image Data (Front, left side, right side) • LiDAR Bit stream
Outputs	• Object Points (Local)
Functionality	• The object detection and recognition module uses environment data from three cameras and a LiDAR to determine the size and position of objects including barrels, lane lines, and potholes. Position is relative to the vehicle.

Module	Discrete Course Mapping
Inputs	• Object Points (Local) • GPS Position Data • Compass Angle
Outputs	• 2-D Course Map
Functionality	• The course mapping module uses the current location of the IGV and detected objects including barrels, potholes, and lane lines to construct a discrete map of the course.

Module	Path Decision
Inputs	• 2-D Course Map (Includes Vehicle Position)
Outputs	• Target Angle • Target Speed
Functionality	• The path decision module uses the 2-D course map to determine the best path to navigate towards the goal. The goal is determined as a point at the end of field of vision that keeps forward motion along the course. This module will also use the GPS waypoints but those are pre-set.

Module	Motor Control
Inputs	• Target Angle • Target Speed • Tachometer Speed • Compass Angle • Control State
Outputs	• Left motor control signal • Right motor control signal
Functionality	• The navigation control modules implement a control system to follow the given path using left and right side motors.

Module	Control State Manager
Inputs	• Hardware Estop State • Autonomous Activation Signal
Outputs	• Indicator Light Control Signal • Estop Trigger forces estop in hardware • Control State (Mode of Operation)
Functionality	• The control state manager module listens for the wireless estop to be activated and forces an e-stop if the estop has been pressed. The module also controls whether autonomous mode is active or inactive.

[ART]

The Object Detection and Recognition Block is further broken down by the type of objects that the vehicle must be able to detect as depicted in Figure 10. The types of objects that must be detected are specified as part of the design requirements in Table 2. The functional requirements of each block are listed in Table 11.

The camera images input to the system are expected to be HD quality images because HD quality cameras generally have higher quality lenses with less lens distortion. Lens distortion will make it more difficult to map objects in real space so a camera with quality lenses must be selected. The high pixel density of an HD camera is not needed for detecting objects at close

range, however, and using HD images costs more processing time. The Image Compression Block reduces the pixel density of the camera images for faster processing.

The three image filtering blocks transform the input image to highlight certain characteristic used to distinguish objects in the image. The potholes and lane lines are white lines on a dark background, likely grass or dirt. The High Contrast Filter Block manipulates the image to darken everything that is not a white. The Pothole Detection and Measurement Block and Line Detection and Measurement Block interpolate information about the location of the lines to reconstruct them in as real-space model. The blue and red filter blocks operate similarly for blue and red. The flag detection and measurement blocks identify and determine the location of flags in real-space with respect to the vehicle.

The Object Detection and Measurement Block uses the distances to objects measured by LiDAR and determines the location and size of the objects in x-y coordinate space with respect to the vehicle.

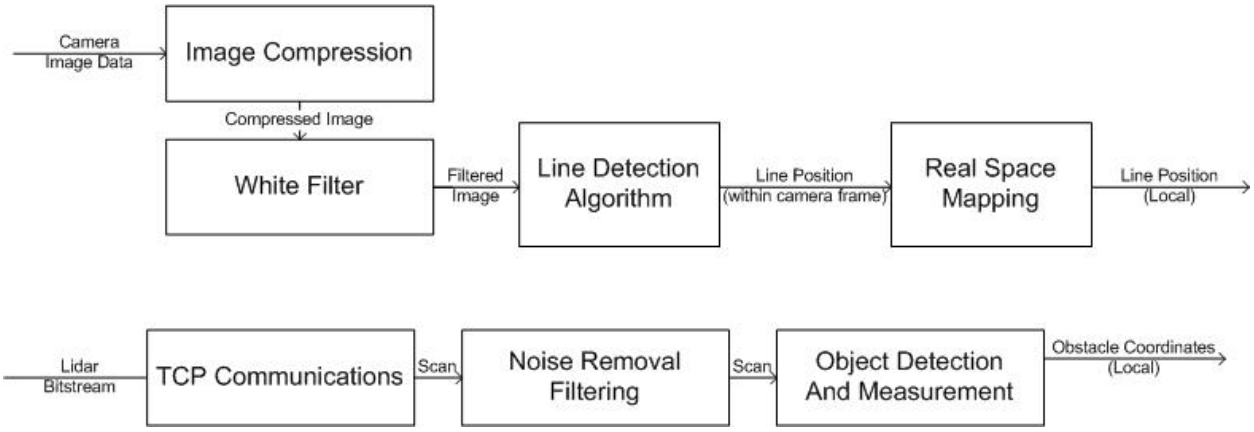


Figure 10. Block Diagram Level 2 (Software - Object Detection and Recognition)

[ART]

Table 11. Functional Requirements Level 2 (Software - Object Detection and Recognition)

Module	Image Compression
Inputs	• Camera Image Data (HD, RGB)
Outputs	• Compressed Image (RGB)
Functionality	• The image compression block will reduce the resolution of the input image to reduce the complexity of calculations for subsequent processing of the image.

Module	White Filter
Inputs	• Compressed Image (RGB)
Outputs	• Filtered Image (B/W)
Functionality	• The White Filter Block applies various filters to produce a black and white image for detecting the white lane lines and white pothole outlines on a black background.

Module	Line Detection Algorithm
Inputs	• Filtered Image
Outputs	• Line Position (within camera frame)
Functionality	• The line detection and measurement module will detect the high contrast lane lines as a series of points in positions relative to the vehicle's location when the image was captured.

Module	Real Space Mapping
Inputs	• Line Position (within camera frame)
Outputs	• Line Position (local)
Functionality	• The Real Space Mapping block maps the objects identified in the camera frame from pixel locations into real-world locations with respect to the vehicle.

Module	TCP Communications
Inputs	• Lidar Bit stream
Outputs	• Scan
Functionality	• The TCP communications module handles all I/O between the PC and the LiDAR and outputs a complete 360 degree scan when it is available.

Module	Noise Removal Filtering
Inputs	• Scan
Outputs	• Scan (filtered)
Functionality	• The Noise Removal Filtering removes errors from the scan and averages scan data between the most recent raw scan frames and averages scan points within the averaged frame.

Module	Object Detection and Measurement (LiDAR)
Inputs	• Scan
Outputs	• Object Coordinates (local)
Functionality	• The LiDAR object detection and measurement module will detect objects from the filtered LiDAR scan and determine their relative position to the vehicle's coordinate origin.

[ART]

A module in the level one software block diagram (Figure 9) is the path decision system. Shown below in Figure 11, the path decision algorithm will have four main components. The first of which is a shortest path algorithm. Two well-known algorithms can be used to compute the shortest path in a system of nodes and connections; Dijkstra's algorithm and Floyd's algorithm. Dijkstra's algorithm is more efficient, but cannot be parallelized over multiple processor threads. Floyd's algorithm is less efficient, but is extensively parallelizable. Following the shortest path algorithm is the routing mechanism. This component will perform a trace-back on the calculated shortest path from the previous component, and will output a set of navigation instructions for the vehicle to follow. The next component takes a rough path input and smooths it to prevent the vehicle from exhibiting 'jitter' when in motion. This will also prevent the vehicle from exhibiting any 'rocking' motion, which could inhibit its ability to accurately identify obstacles with cameras. The final component in the navigation system calculated the required speed and steering adjustments that need to occur to follow the route given by the previous component. This component will involve taking readings from the digital compass and Novatel GPS system to accurately calculate the direction adjustments that need to occur.

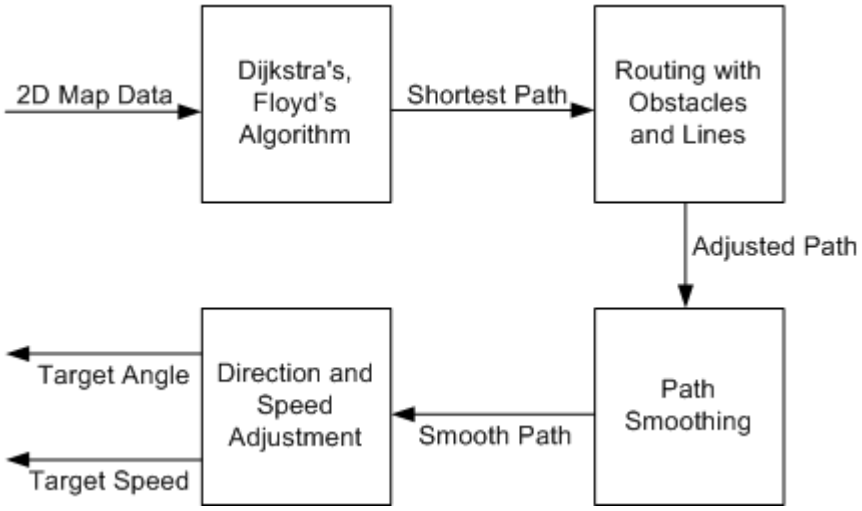


Figure 11. Block Diagram Level 2 (Software - Path Decision)

Table 12. Functional Requirements Level 2 (Software - Path Decision)

Module	Dijkstra's, Floyd's Algorithm
Inputs	• Raw 2D Map Data
Outputs	• Shortest Path
Functionality	• This module will use a variation of Dijkstra/Floyd's Algorithm to compute the shortest path to the next waypoint.

Module	Routing With Obstacles and Lines
Inputs	• Shortest Path Data
Outputs	• Adjusted Path
Functionality	• This is an intermediate software stage that will incorporate obstacle and line data into the path calculation. This block will also use a variation of Dijkstra's/Floyd's Algorithm to compute the shortest path to the next waypoint while compensating for obstacles and lines observed by the detection system.

Module	Path Smoothing
Inputs	• Adjusted Path
Outputs	• Smooth Path
Functionality	• This is an intermediate software stage that will manipulate the jagged-edged path from the "Routing with Obstacles and Lines" stage and will interpolate a smooth route in order to reduce zig-zag motion in the path to the next waypoint.

Module	Direction and Speed Adjustment
Inputs	• Smooth Path
Outputs	• Target Angle • Target Speed
Functionality	• This is the final module of the path decision block. This block will translate desired motion from the path data into delta angle and delta speed required to fulfil the path. While the input path is 'smooth,' it is important to allow for a margin of error in delta angle and delta speed in effort to avoid an over-under compensation loop.

[CRE]

3.4 Mechanical Block Diagrams and Functional Requirement Tables

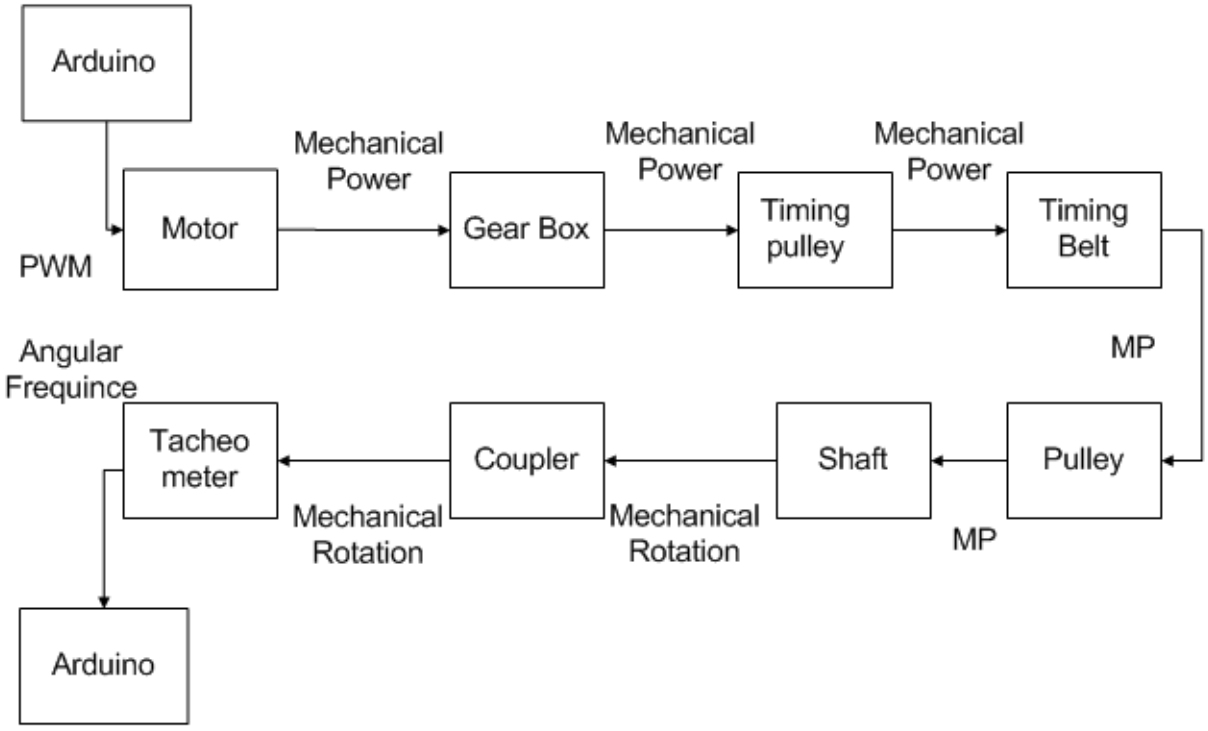


Figure 12. Block Diagram (Mechanical – Tachometer Mounting)

Module	Motor
Inputs	• PWM Power Signal
Outputs	• Mechanical Power (Rotating shaft)
Functionality	• Transforms Electrical power into mechanical power

Table 13. Functional Requirements Level 2 (Software - Path Decision)

Module	Gear Box
Inputs	• Mechanical Power (Rotating shaft)
Outputs	• Mechanical Power (Rotating shaft)
Functionality	• Increase Torque, decease speed

Module	Timing Pulley
Inputs	• Mechanical Power (Rotating shaft)
Outputs	• Mechanical Power (Spinning gear)
Functionality	• Transfers Mechanical Power

Module	Timing Belt
Inputs	• Mechanical Power (Spinning gear)
Outputs	• Mechanical Power (Moving Belt)
Functionality	• Transfers Mechanical Power

Module	Pulley
Inputs	• Mechanical Power (Moving Belt)
Outputs	• Mechanical Power (Rotating Shaft)
Functionality	• Transfers Mechanical Power, Increases Speed, Decrease Torque

Module	Coupler
Inputs	• Mechanical Power (Rotating Shaft)
Outputs	• Mechanical Power (Rotating Shaft)
Functionality	• Transfers Mechanical Power

Module	Tachometer
Inputs	• Mechanical Power (Rotating Shaft)
Outputs	• Electrical Digital Signal
Functionality	• Measures Speed of Rotating Shaft

Module	Arduino
Inputs	• Electrical Digital Signal
Outputs	• Electrical Digital Signal (PWM)
Functionality	• Controls Motor Steering

3.4 Motor Design

3.4.1 *Motor Analysis*

To analyze the motor, the key requirement is to determine the vehicle's ability based of the determined path that the vehicle will operate. As determined by the IGVC rule book, the vehicle must move between one and five miles an hour even while travelling on the steeper inclines of thirty degrees. Therefore, the analysis and calculations for the vehicle's operation must be effective under the most strenuous circumstances. Other considerations for the terrain and typical usage conditions can be found in Table 14 and will be reviewed in further details.

Table 14. Motor Analysis Assumptions

Mass	36.73	kg
Hill grade	0.10	rad
Wheel Radius	0.0762	m
# of Wheels	4	
# of Motors	2	
Motor efficency	80	% efficiency
Angular Velocity	11.73	rad/s
Voltage	24	V
Time in use	4	hours
Acceleration	0.06	m/s^2
Rolling Resistance Co.	0.1	
Velocity m (2.24m/s)	0.89	m/s

The mass of the vehicle was calculated to be at most 36.73kg from the consideration of the batteries (35%), chassis (12%), sensors (26%), drivetrain (4%), competition's load (19%), and a five percent error. To isolate the highly noisy motor drive power consumption from the sensors, the motor power will be separated from the sensing power consumption via different batteries. As discussed more in detail in the system voltage and battery capacity parameters, the motors will consume twenty-four volts which will be realized through two twelve volt batteries. Similarly, the sensor system may require twelve and/or twenty-four volts to maintain sensing. The battery mass was approximated at four batteries with the amp-hour capacity that could supply the motor's consumption.

The hill grade is the approximation of the course itself, and during the advanced competition the grade range was +/- 0.52 radians. The competitions general landscape was within 0.1 radians, however the advanced competition allowed for the more extreme 0.52 radian hill climbs. Because of this, required torque and power of the motor was mapped from positive to negative 0.52 radians. In addition, the anticipated speed of the vehicle was also included into the calculations, because on near flat surfaces, the vehicle should be maneuvering and max speed, whereas climbing a steep hill it can traverse at a slower pace. This variation of speed and hill gradient for three wheel diameters lead to the max required torque of 9Nm and average power required to operate eleven to fifty watts depending on the regeneration ability of the motor controller.

The wheel radius was based off the motor's torque, vehicle's speed, and load bearing capabilities. An observation noticed early on was, with a constant velocity, the increased wheel radius would lead to an increased required torque. Similarly, with a constant torque the larger wheel would allow the vehicle to travel faster, but because the competition limits the max velocity, the decision was to go with the smaller radius of wheels. The radiuses ranged from seven to thirty centimeters, from which the conclusion was the smaller radius had the smaller torque, but would have mounting and chassis constraints.

Initially, the number of wheels was three so that at any given moment the vehicle would be on the ground. However, due to the mass of the vehicle, the wheels needed to be able to support the weight of the vehicle, thereby requiring the vehicle to have four. Calculations for tipping forces and center of gravity has been conducting by the team's counterparts, but have not yet been implemented into our system.

The number of motors was chosen to be two, and the following factors lead to this conclusion: motor driver(s), power consumption, drive style, and ease to implement. Most commonly motor drivers come in single or dual motor capabilities especially when dealing with ten plus amps per motor. Next, the values that were calculated placed the torque per motor in the standard torque range of motors used in wheel chairs, which are aided with a torque converting worm drive system to increase the motor's output. When it came to the drive style, the possibilities of having a separate steering system were quickly eliminated due to the decision not to use a linear actuator or other power consuming device, due to the additional complexity and speed of the vehicle (0-2.25m/s). That being decided, the team chose to have a "tank drive" system in which both motors work together for forward and reverse directions. The tank drive turns consist of the outside wheel moving faster than the inside, or if approaching a tight turn, the inside motor turns in reverse. Pulling all the considerations together, this decision for two motors make the drive system easy to adjust, implement, and debug if a problem were to occur.

Motor efficiency varies upon the type of motor: DC or AC, brushed or brushless, as well as other parameters. For instance, the brushed DC motor varies from 75-85% whereas the brushless is more efficient ranging from 85-95%. Due to the supply and cost, the calculations assumed a DC brushed motor. Thereby, the motor efficiency was calculated at 80%.

Through observation, the power of the system varied directly with the amount of torque needed by the system. This power is represented by voltage multiplied by current, of which the design parameter became the system's voltage. The average power consumption of the motors was 50W. For DC motors, the common voltages are five, twelve, twenty-four, and forty-eight. Of which, the decision lied between the twelve and twenty-four-volt system, due to variety of motor choices, and clearance of lab restraints. The result was to have a twenty-four-volt system so that the current consumption would be cut in half. For instance, at fifty watts and twenty-four volts the current is around two amps, whereas the twelve-volt system would have a four amp draw Thereby, allowing the team to use thinner wires and less resistive loss through the system.

As this is a competition project, the vehicle would be operated under near continuous conditions while maneuvering the course as well as the facilities as the team went to different events and presentations. Because of this constraint and belief that the team would be away from a charging circuit for some time, the decision was that the vehicle should be available for use for approximately four hours at a time. This lead to the battery capacity of eight amp hours assuming no voltage regeneration from the motor driver. The four-hour charge time is for the extreme case when the vehicle is in constant use between the competition's beginning and lunch break or lunch and the end of the day.

[JPJ]

3.4.2 *Motor Calculations*

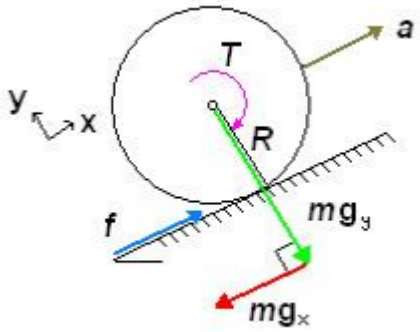


Figure 13. IGVC Force Diagram

Analysis for the power management and motor calculations began with Figure 13. IGVC Force Diagram with determining the forces acting on the vehicle. Those forces include gravity, normal, applied, and rolling resistance.

Force of Gravity Equation:

$$Fgx = m * g * \cos(\theta) \qquad (1)$$

From the mapping of the course, the overall incline of the field was within plus or minus five degrees. However, the competition has an advanced section which includes plus or minus thirty degree ramps to which the vehicle would have to overcome. The course mapping included calculations which were performed in increments of ten degrees to cover all possible variations of the course. Even though the vehicle will be in a potentially muddy field, the assumption is that the normal force will stay the same, but the rolling resistance force will vary.

Force of Rolling Resistance Equation:

$$Frr = Fgx * B \quad (2)$$

Rolling Resistance is dependent on the gravitational force as well as the surface. The gravitational force is pushing against the ground and in some cases, deforms the tires. This tire deformation increases the surface area on the road at a given point, which increases the rolling resistance. The other factor of rolling resistance is the coefficient of the ground. Because the vehicle is operating on a field, the intended coefficient is 0.3.

Force applied Equation:

$$Fa = ma = Fgx + Frr \quad (3)$$

The total force applied to the vehicle is mass times acceleration. The acceleration value is based off the vehicle's max speed of 2.25 m/s and declaring that the vehicle will take ten seconds to achieve that speed.

Torque Equation:

$$T = F * r \quad (4)$$

The torque is then calculated using the above formula. To improve the accuracy of the torque equation the motor efficiency is inversely multiplied into the system (100/efficiency). However, this calculation gave us the total torque required for the vehicle to move at its given velocity. Therefore, the total torque that was calculated in the above equation was then divided by the number of motors. This will be realized using the two dc motors and a gearbox from the power wheels, which can carry 60kg at a up the 30-degree.

Power Equation:

$$P = T * w \quad (5)$$

The angular velocity (w) is found from the velocity divided by the radius of the tire.

Current Equation (6):

$$I = P * V \quad (6)$$

The voltage of the system was calculated at twelve volts as well as twenty-four. However, due to the larger current of the twelve-volt system, twenty-four volts was chosen and

returned the current required by the system. Note that, the motor itself will come with its own power consumption, to which the actual, as opposed to the theoretical, current can be calculated. Based on the vehicle's use of 12 AWG, the maximum current that can safely flow through the device is 30A. In addition, a similar vehicle has been tested and the stall current averaged four amps with a peak of twelve amps. Therefore, the Roboclaw 2x30A motor driver which can drive both motors at thirty amps will not exceed the allowed current. Similarly, the fuse will be 30 amps to protect the wiring of the device.

Capacity Equation (7)

$$Q = I * t \qquad (7)$$

Since the vehicle would need to be operational for four hours, the total capacity would be four hours times the motor's current draw. The power wheels' vehicle is rated for four hours of use. However, to achieve a full charge a four to six-hour charge is needed, which is larger than the anticipated afternoon break. To ensure the vehicle will continue to work, a spare battery will be purchased.

[JPJ, GWC]

3.4.3 *Motor and Drivetrain Schematics*

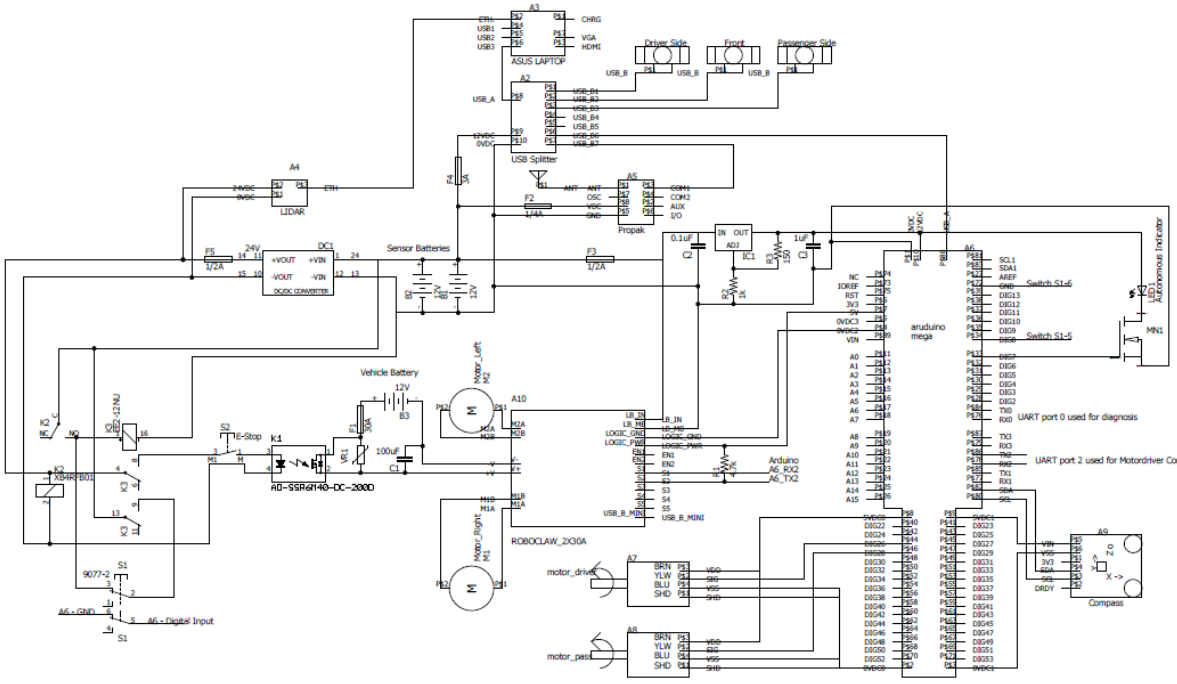


Figure 14. IGVC 2017 Complete Schematic

The 2017 intelligent ground vehicle will be comprised of a modified toy jeep donated by Power Wheels. Modifications will mainly include the changeover from a throttle made from a mechanical switch and a driver input required steering system to a tank drive vehicle. To do so, the existing motors will be used, the throttle and transmission switches will be bi-passed to the motor driver. A thirty-amp fuse, F1, has been chosen to protect the 14 AWG wires, as the wires max is thirty-two amps.

To control the emergency, stop system, a mechanical and electrical switch has been implemented. Using the sensor battery, B2, the isolated voltage will enter a wireless relay, K2 XB4RFB01 that can transmit up to 100m which is three times the IGVC rules requirement. This wireless relay will be mounted on the outside of the jeep using foam tape to reduce the vibrations. K3 is used to convert a singular pulse from the wireless relay into a held signal. That relay output connects to the mechanical kill switch, S2, and then to the control input relay, K1, which operates between a 3 and 32 Vdc signal. All connections to the relay K1 will be terminated with ring terminals. The relay will be attached to the main frame of the vehicle using foam tape. Upon firing the relay on, the current will flow into the Roboclaw motor driver. Upon closing the K2 relay, there can be a large transient voltage that will be suppressed with a Metal-Oxide Varistor, R2, which can protect up to a 50A surge.

The two motors will be from a Power Wheels jeep, and they will include the initial gear boxes. These motors will be wired to the motor driver using the manufacturer's wiring and ring terminals. The motor driver will be mounted to the vehicle's body between the two motors using #4-40 hardware. The Arduino will be communicating with the motor driver using the UART protocol over 24AWG wire. To reduce noise susceptibility, 4.7k pull-up resistors will be attached to the transmit and receive pins, logic voltage will be independent of the battery voltage, and the baud rate will be limited to 38400.

Lastly, cable ties will be used to mount the wire harnesses to the frame, and grommets will be used to safely pass through holes in the frame without damaging the harnesses. Specifics about the sensor part will be included in detail more in the sensor portion.

[JPJ]

3.5 Control Design

3.5.1 *Vehicle Motion Analysis*

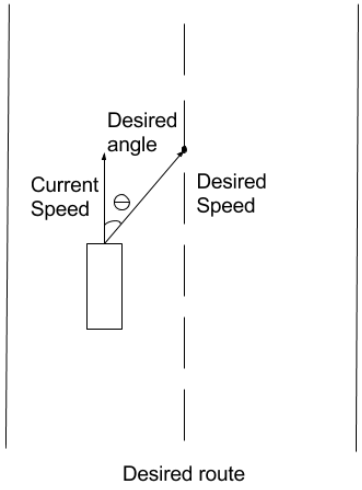


Figure 15. Vehicle Motion Diagram

The vehicle is controlled using a tank drive. This means both the angle (X, Y coordinates) and the overall speed of the vehicle are determined by the individual speeds of each wheel. Figure 15 shows the motion of the vehicle to its desired route. There are five inputs in this system, the desired speed and desired X, Y coordinates of the vehicle, the current speed of each wheel, feedback, and the control state. The output of the system is a pulse width modulated signal to a motor drive. The all the inputs are determined by the sensor data and the CPU. The transfer function for this system is a 4th order type system.

The desired speed and position are determined at separate locations within the system. The desired speed is the total speed of the vehicle and must be between 1-5 mph, taken as an average over the course. The desired speed will nominally be 5 mph, however due to the path that we must take this cannot be the case. The current position of the vehicle is take as a reference position and set equal to zero. The desired position can then be said to be the angle the vehicle must rotate and the distance the vehicle must travel. To find the Speed at which each wheel must travel to reach the desired destination in an allowed time we look at the equations of motion.

The Equation of Motion for a tank drive vehicle have two main parts. The first is the forward motion of the vehicle, or the velocity both wheels have in common. The second is the angular motion, the turning motion of the vehicle, this is due to the difference between each of the vehicle's wheels. The angular motion can be represented as an angle θ and can be found by

taking the difference in the wheel velocities. It is appropriate to model the angular motion of the vehicle by setting one of the wheels of the vehicle to zero and the other wheel to the difference of the two velocities. This model creates a circle with a radius equal to the vehicles width. Using the velocities in the wheels we can find the distance the vehicle travel around this circle; this is the arc length or distance traveled. From the Arc Length, we can calculate the angle change with respect to time. Figure 16 shows the derivation of arc length.

$$L = \frac{\theta}{360°} * 2 * \pi * r \qquad (8)$$

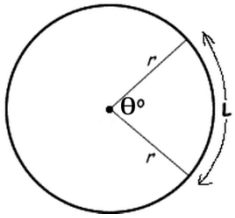


Figure 16. Arc Length.

Where r is the width of the vehicle and L is, the distance traveled. From Equation (8), we can find the angle theta, given by

$$\theta = \frac{L * 360°}{2 * \pi * r} \qquad (9)$$

The Distance traveled can be found by the integral of the difference in the wheel's velocities with respect to time, shown here

$$L = \int (V_L - V_R) dt \; . \qquad (10)$$

From this, Equation (9) can be rewritten.

$$\theta = \frac{\int (V_L - V_R) dt * 360°}{2 * \pi * r} \qquad (11)$$

The total motion of the vehicle is a convolution of the forward motion (Mf) and the angular motion or the common wheel velocities and the differential wheel velocities, respectively. The total motion of the vehicle can be represented as the motion around a circle of different radiuses. These radiuses are determined from the total arc length and the angle theta due to the angular motion. Figure 17 shows this relationship.

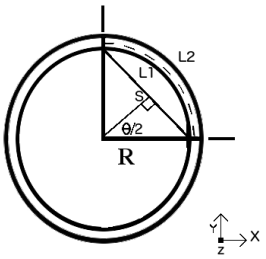


Figure 17. Total Motion.

Figure 17 shows the total motion of the vehicle. L2 and L1 are the distance each wheel travels, the wheel L2 has traveled farther than wheel L1. The Dotted line in the two circles represent the center of the vehicle and its movement. The value R is the radius of the circle that the vehicle travels around and can be found using the arc length formula.

$$R = \frac{L * 360°}{2 * \pi * \theta} \tag{12}$$

Where theta was calculated in Equation (12) and L is the arc length of the circle, L1 or L2 replace L in this equation. Equation (13) finds the radius of the circle for the center of the vehicle.

$$R = R \begin{matrix} - \\ + \end{matrix} \frac{W}{2} \tag{13}$$

Where w is the Width and the plus and minus depend on the arc length used.
The x and y distances can now be found using trigonometry, Figure 18 takes the triangle from Figure 17 and relates it to x and y coordinates.

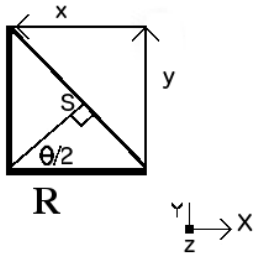


Figure 18. Triangle Motion.

Equation (14) and Equation (15) solve for x and y, respectively.

$$X(t) = Sin\left(-\frac{\theta}{2} + 180\right) * Sin\left(\frac{\theta}{2}\right) * R * 2 \quad (14)$$

$$Y(t) = Cos\left(-\frac{\theta}{2} + 180\right) * Sin\left(\frac{\theta}{2}\right) * R * 2 \quad (15)$$

For the Controller a state space model representation of the system is used, as the state space model is accurately portrays non-linear systems, discussed more in section 3.5.3 *Control Theory*.

The wheel speed and the current angle are both feedbacks in the system. The Wheel speed uses a tachometer to determine the angular frequency of the shaft and transforms that into a voltage. That voltage is used as feed back into the controller. The current angle is set to zero as discussed before. The current angle of the vehicle is found using a GPS and a digital compass.

The control state is for safety use and is used to turn on and of the movement of the vehicle. This can be done by a manual switch or a remote switch. The judges are in control of the vehicle's state. The control state is placed at the end of the motor controller.

The output of the motor controller is two pulse width modulated control signals, one for each motor. Each signal contains data on the speed that a motor needs to run. The UART was chosen because it allows for an easy and effective way to control a motor. The output is sent to a motor drive.

[GWC]

3.5.2 *Vehicle Motor Analysis*

The Motors are an essential part of the Vehicle's control system. Without an accurate model of the motors running each motor at the desired speed will be extremely difficult. Not just that as the system is a tank drive system any inaccuracy with the speed control of the motor is detrimental to the system. In essence the motors need to run at the speed requested by the controller.

The system uses a digital motor driver, because of this an accurate model of the overall system was not achieved. To account for the lack of an accurate model, feedback of the motors was the major priority. Experimental Gain was used alongside of a pure integrator. An appropriate gain that was found was approximately 1.8, this gain gave a quick and timely rise time with a small amount of overshot and the system was complete stable for all bounded inputs.

Tachometers where used to find the velocity's of each wheel. A sampling time of 100 milliseconds was used to find the current speed of the motors. Tachometers create a pulse

changes as the wheels spins, 200 pulse per revolution. The use of an interrupt was need to accurate count the number of pulses within the sampling time. This method allowed for a quick and precise result.

The The code shown below in Figure 19 shows the method of finding the distance traveled and the total angle change of the vehicle.

```
void LocationCalc(float* Speeds,int timeElapsed){  // FeedBack
  double CurrentAngle, CurrentDistance;
  double FeedBack[2];
  CurrentDistance=0;
  CurrentAngle = 0;
  float  DeltaSpeed = Speeds[0] - Speeds[1];
  float  TotalSpeed = (Speeds[0] + Speeds[1])/2;
  if(DeltaSpeed<=0.05 && DeltaSpeed>=-0.05){
    DeltaSpeed=0;
    CurrentDistance = TotalSpeed * (0.1)*(0.4469);
    CurrentAngle = 0;
    OverallAngle = (CurrentAngle + OverallAngle);
  }else{
    CurrentAngle = DeltaSpeed * (0.1)*(0.4469)* (180/(PI * WIDTH ));
    OverallAngle = (CurrentAngle + OverallAngle);
    CurrentDistance = sin((OverallAngle*PI)/360) * 2 *(TotalSpeed / DeltaSpeed) * WIDTH;
  }
  OverallDistance = (CurrentDistance-PrevDistance) + OverallDistance;
  PrevDistance=CurrentDistance;
  return 0;
}
```

Figure 19: Location Calculation from Tachometers

[GWC]

3.5.3 *Control Theory*

The Control Design for the system will use a Multi-Input Multi-output (MIMO), shown in Figure 20. The System only has two inputs a Desired Angle and a Desired Distance. The outputs will be a new position and orientation of the vehicle. There are two forms of feedback within the system, the Frequency of each motor determined by a tachometer and the current position of the vehicle. The Current position of the vehicle is found by the sensor data and the GPS, these are discussed in 4.6 Sensor System Design.

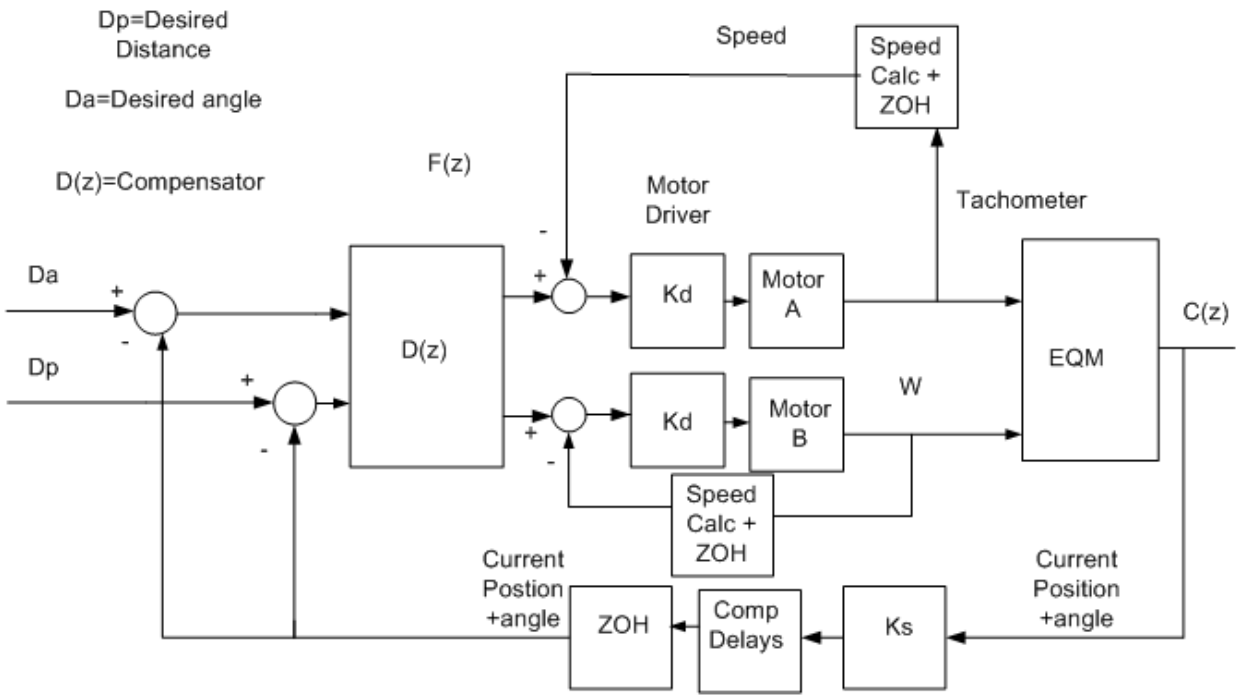


Figure 20. Controller Block Diagram.

The Uncompensated system has two motor drivers, two motors and a system of equations that govern the motion. The Equations of motion are discussed in 4.5.1 Control Analysis. The motor of the system is going to be a DC brushless motor, and the motor drive can just be interpreted as a gain add to the system. Because the Motors are DC brushless motors, they add a single pole to the system, changing the final order of the system from a 3rd order to 4th order and a 2nd to 3rd order for the x and y coordinates, respectively.

The Closed-Loop compensator will take care of any error within the system. The Controller is a state space controller meaning, the controller compensates for the error in the system by applying gain to each state feedback of the system. The controller improves the stability and response time of the system.

The State Space model of the system uses a linear approximation of the non-linear system at an infinitesimally small time. The rate of change of the Euclidean Space x and y axis shown in, Equations (16) and (17), respectively.

$$\dot{X}(t) = \cos\left(\frac{vr + vl}{2}\right) \tag{16}$$

$$\dot{Y}(t) = \sin\left(\frac{vr + vl}{2}\right) \tag{17}$$

These equation represent the error that the system while input into the controller. A polar cordate system is used here to limit the amount of calculations. Equation (18) shows the total distance traveled by the vehicle. Equation (19) shows the total angle change undergone by the vehicle.

$$R(t) = \sqrt{\overline{x}^2 + \overline{y}^2} \tag{18}$$

$$\emptyset(t) = \arctan\left(\frac{\overline{y}}{\overline{x}}\right) \tag{19}$$

Where $\overline{x}$ and $\overline{y}$ are the total Euclidian coordinate changes of the vehicle. Taking the derivative of Equation (18) and (19) leads to the position and angle change of the vehicle.

$$\dot{R}(t) = \cos(\gamma) * \frac{vr + vl}{2} \tag{20}$$

$$\dot{\emptyset}(t) = \frac{-\sin(\gamma)}{r} * \frac{vr + vl}{2} \tag{21}$$

Here γ is the difference between the Total angle change and the current angle, or the current error in the angle. The rate of change of γ is given by equation.

$$\dot{\gamma}(t) = \frac{-\sin(\gamma)}{r} * \frac{vr + vl}{2} - \gamma - \emptyset \tag{22}$$

A proportional feedback gain can now be applied. The state space feedback matrix uses a distance gain, K_r , an angle error gain and a current error gain, K_γ $K_\emptyset$, respectively. These gains make up the feedback matrix B.

$$\begin{vmatrix} K_r & 0 & 0 \\ 0 & 0 & -K_r \\ 0 & -K_\emptyset & -K_r - K_\gamma \end{vmatrix}$$

With the input matrix,

$$\begin{vmatrix} R \\ \emptyset \\ \gamma \end{vmatrix}$$

For the Stability of the system we place all the eigenvalues of the system inside the unit circle by modifying these gains. Taking the determinate of the system($\lambda I - B$), shows that for stability K_r must be lease than 0, $K_r + K_\gamma$ must be greater than 0, and $K_\emptyset * K_r$ must be lease than 0. K_r Was chosen to be -1, K_γ was chosen to be 1.1 and $K_\emptyset$ was chosen to be 1. These values where chosen because they offer a stable system with relatively fast response.

Another method of controlling the vehicle's position can be obtained from the equations of motion model. This method inverts the equations of motion and the beginning if statements of the model to create a controller method. This controller method finds the corresponding wheel velocity's for a given input. Then compares those velocity's with the feedback velocity's to find the error in the velocity's as talked about in section 3.5.2 *Vehicle Motor Analysis*. The Method is shown below in Figure

[GWC]

3.6 Sensor System Design

3.6.1 *Sensor Analysis*

The vehicle has three groups of sensing applications; Object Detection, Course Mapping and Speed Detection. These sensor groups are the tools used by the vehicle's software to navigate the course timely and safely. Each sensor is powered by the 12V battery specified in the hardware block diagram. The voltage is stepped down to 5V and properly fused after the battery via a dc-dc 5V switchable step down to ensure appropriate power is provided to each sensor.

Object detection is carried out by the camera and a LiDAR. The cameras are mounted five feet from the wheel base with one camera in front and one on each side of the vehicle and angled 45°. They take real time video of the course as the vehicle traverses it. The video is filtered as discussed in the software analysis to detect white course lines, red and blue flags and white potholes. Logitech C920 HD Pro webcams were chosen for the IGV. They have an adjustable base which makes mounting and setting the angle to 45° a simple task. The cameras will be powered via the USB hub drawing roughly 1.1 amps per camera. They will be wired from the USB hub to the ASUS laptop for image processing discussed in the software analysis.

LiDAR (Light Detection and Ranging), as discussed in the software analysis, detects the distance the vehicle is from an object with distances up to 30m and roughly 10mm resolution. The LiDAR is mounted to the front of the vehicle above the motor to provide a clean scanning area for the laser pulses. The distance information provided by the LiDAR is processed to determine an object's distance from the vehicle. The camera and the LiDAR, in concert, act as a

safety bubble around the vehicle protecting it from objects and obstructions throughout the course.

Two LiDAR's were reviewed for the IGV, the Pepperl + Fuchs R2000 and the Sick LS290. The Pepperl + Fuchs LiDAR offers a 360° scanning field at an industry leading angular resolution of $\geq 0.071°$. It offers an adjustable scanning frequency of 10Hz-50Hz. The Sick LS290 offers a 270° scanning field at an adjustable angular resolution of 0.25°, 0.5° and 1°. It should be mentioned that software is not available for either LiDAR but Sick offers a platform to take in the bit stream sent out by the Ethernet out of both LiDAR's. It is up to the user to program via their programming language of choice. Calculations were carried out and discussed in Equation (26) & (27) which concluded that the Pepperl + Fuchs R2000 was a better option than the Sick LS290 given its superior angular resolution and functionality.

Course Mapping is provided via GPS. The GPS hardware communicates with the provider's satellites through an onboard antenna. The coordinates are given via satellite pings sent back to the GPS providing the vehicle's current location in the world with up to 10cm resolution. The GPS software accepts this information and builds a map of the vehicle's surroundings. The GPS is mounted above the motor with the antenna placed perpendicular to the vehicle. The top most part of the antenna shall be no higher than six feet tall.

A Novatel Propak V3 was selected as the GPS system utilized on the IGV. It offers simplistic connection to the 12V battery and draws on average 2.8W. The Novatel Propak V3 will send its information through a serial to USB converter into the USB hub then to the ASUS laptop for software implementation. The GPS will have a matched Novatel antenna with compatibility for L1, L2, L2C, L5, L-Band, and SBAS tracking. OMNIstar's GPS/GNSS software will be purchased to allow access to its GPS/GNSS satellites around the world. OMNIstar offers superior accuracy of 4cm resolution when paired with the Novatel Propak GPS and antenna. These are top of the line components and should grant the IGV excellent real time locational information while traversing the course.

The vehicle's direction is crucial to the functionality of the GPS. A digital compass is implemented to insure the proper direction is sent to the GPS for navigation. The digital compass works in the same way a magnetic compass does, however it applies magnetic sensor technology to reduce magnetic field interference from the motor. The 3-Axis Digital Compass IC HMC5883L from Adafruit is utilized by the IGV. It offers an affordable accommodation for digital compass interfacing with a microcontroller opposed to a standalone unit which can run thousands of dollars. A pin layout for the HMC5883L is shown in Figure 21 which will be needed when discussing the schematic of the Arduino MEGA 2560 microcontroller.

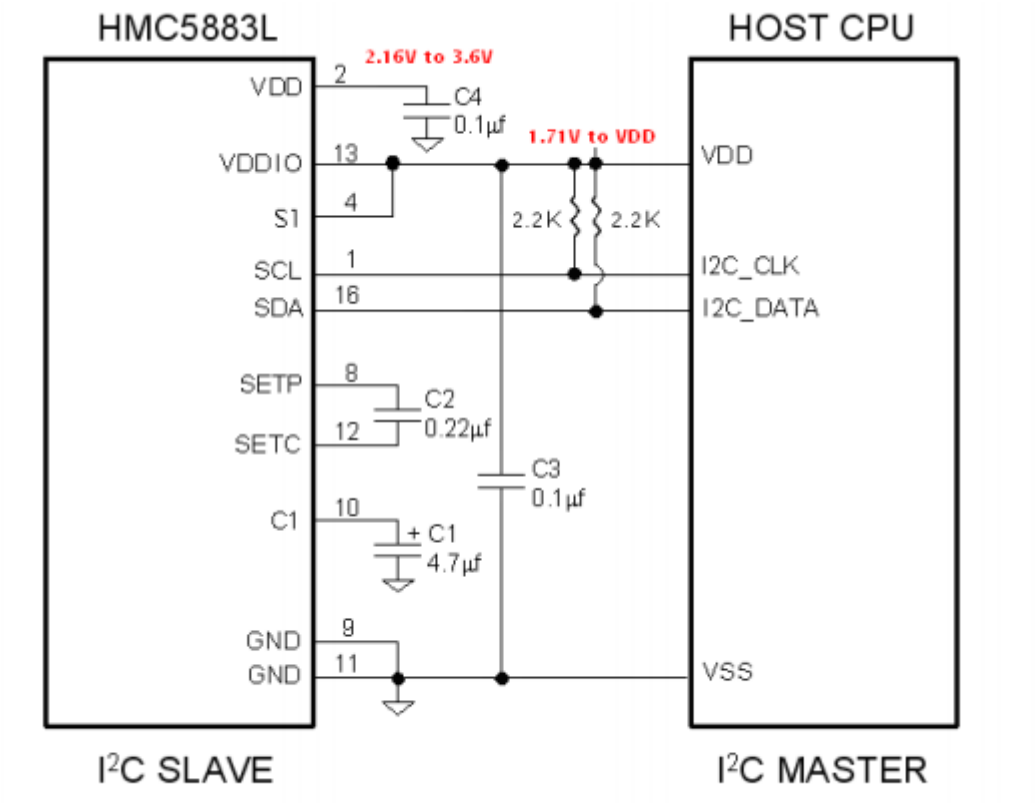


Figure 21. HMC5883L Pin Layout

Speed detection is executed through application of a tachometer. Motor A and Motor B respectively feed their angular frequency into the tachometer. The tachometer reads the angular frequency and translates it into an analog RPM signal. 2 YUMO-E6A2-CS3E rotary wheel encoders were selected to monitor the speed of the IGV. The YUMO encoders offer simple speed decoding with incorporation of the microcontroller. The YUMO is supplied with a power, ground, A & B output to the microcontroller. Its connection will be discussed in the microcontroller discussion.

To utilize the selected tachometer and digital compass a microcontroller must be employed in the sensor network. Two microcontrollers were considered for the processing; the Arduino MEGA 2560 and the Raspberry Pi V3. Two factors that were most heavily considered were analog compatibility and enough inputs for the components it will be using.

The Raspberry Pi offers only digital inputs which would require a digital to analog converter (DAC) for each input of the tachometers resulting in four (4) DAC's needed. The Raspberry Pi is limited by its input capabilities as well by only offering enough SPI inputs to

process one tachometer. An option would be to write commands in a different format for the second tachometer but seemed to be an unnecessary use of technology.

The Arduino MEGA 2560 employs up to 16 analog inputs on board as well as enough I^2C and SPI inputs to power and process the given sensors. **Error! Reference source not found.**14 shows the wiring schematic for the Arduino MEGA 2560 microcontroller and its connections to the tachometers, digital compass and handheld controller. The values in boxes to the right of the Arduino indicate the pin number on the microcontroller. The handheld described **Error! Reference source not found.**14 is only used for manual mode for transportation purposes and will not play a part in autonomous travel.

A layout of the sensor network on the vehicle is provided in Figure 22 below. The cabling is labeled and connected as shown in the figure. It should be noted that the figure is not to scale and sensors are placed for ease of wiring.

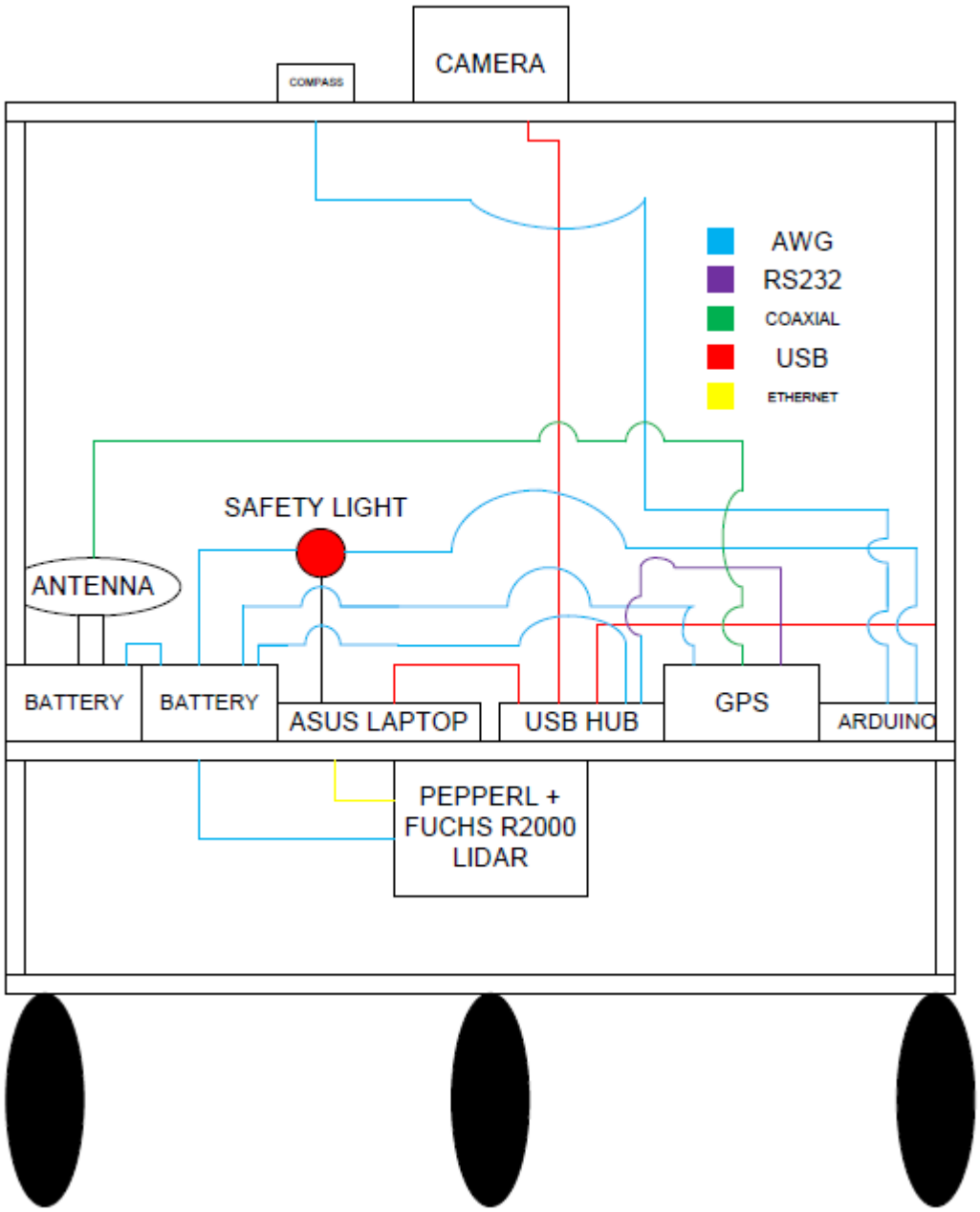


Figure 22: Sensor and wiring diagram

[ACG]

3.6.2 *Sensor Implementation*

The HMC5883L is powered and processed via the Arduino Mega. The digital compass measures Earth's magnetic field and relays its position based on x, y and z readings provided in microteslas. To make these numbers useful, an equation is performed taking the arctangent of the x and y coordinates and adding a declination angle predetermined by location, as found at http://www.magnetic-declination.com, to offset any imperfections in the readings. The number provided by this calculation is referred to as the Heading value, or facing direction of the vehicle, and is measured in degrees. A value of 0° is considered true north. All other degrees refer to a clockwise rotation about the sensor from this position. For example, a direction of west would yield a heading value of 270°. This value will be used in unison with the GPS to calculate movement of the vehicle.

The GPS will provide a means of locating the starting position of the vehicle and a waypoint for the vehicle to travel to. Latitude and Longitude readings will be provided for both positions. This grants a heading direction for the vehicle to travel i.e. vehicle is positioned at 41.0814° N, 81.5190° W (Akron) and is headed towards 42.6700° N, 83.2061° W (Oakland University) the heading value would be 1.5886° N, 1.6871° W. Given these coordinates a number of about 271° would be used for the GPS header value.

These two values will then be compared to ensure the vehicle is headed in the proper direction, the proper wheel is turned and if the vehicle is remaining on track while traversing the course. To ensure the proper wheel is turned to make the compass and GPS headers equal, the two values are compared in a circular manner. A brief overview of this process is depicted in Figure 23.

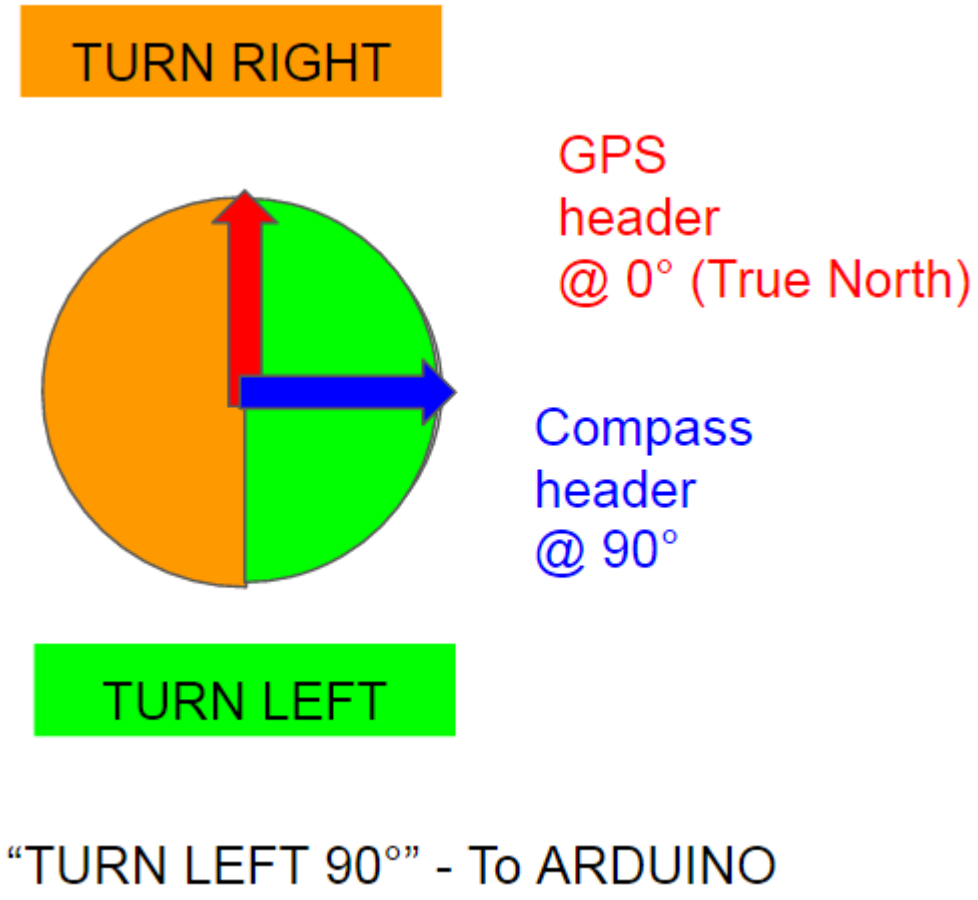


Figure 23: Compass and GPS header comparison

In this example, the compass header reads 90° or facing east. The GPS header reads 0° or saying to head north. To determine which direction the vehicle must turn, the circle is divided in half based on the GPS header. Whichever half of the circle the compass header is then located in will determine which direction the vehicle must turn. In this example the vehicle will turn left. A message is then sent to the Arduino to power the right wheel until the compass header is equal to the GPS header value.

A course example is shown in figures 23-26.

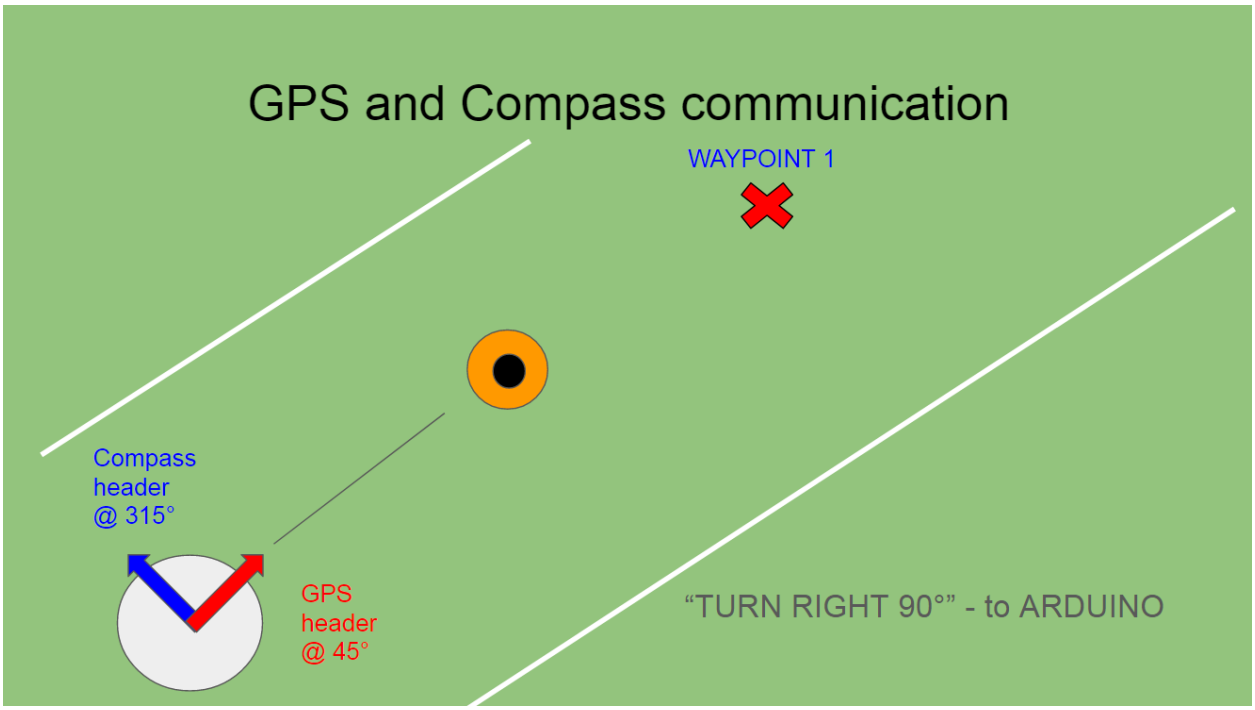


Figure 24: GPS and Compass Communication 1

In this example, the vehicle is facing NW at 315° from true north. The GPS yields a header value of NE at 45° from true north to get from the vehicles current location to waypoint 1. Dividing the circle in half from the GPS header would force a right turn if the compass header value is between 225°-360° and 0°-45° and a left turn from 45°-225°. Since the compass header is 315° this would yield a right turn of 90° to make the compass header = the GPS header.

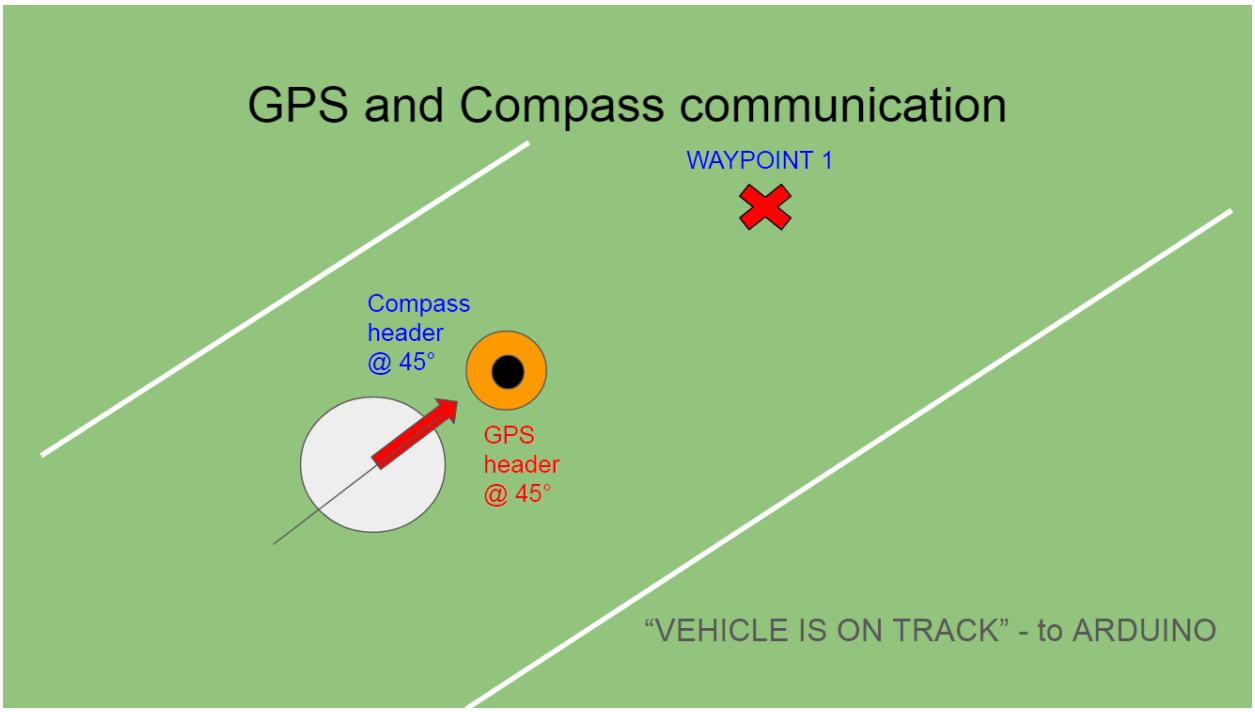


Figure 25: GPS and Compass Communication 2

In Figure 24, the vehicle is on track but the LIDAR detects a cone in route to Waypoint 1.

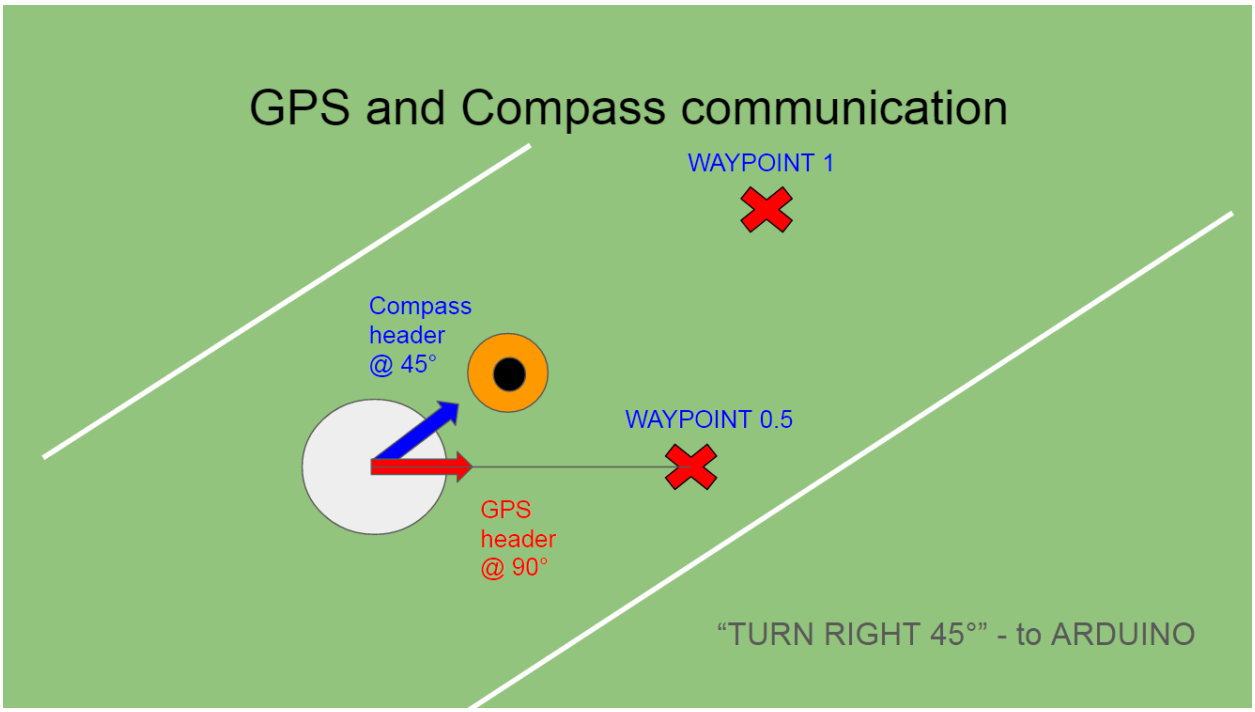


Figure 26: GPS and Compass Communication 3

Figure 25 depicts how the vehicle avoids the cone using GPS and compass communications. Once an object is detected obstructing the vehicles path to its determined waypoint, a “halfway” waypoint must be created to travel around the object, thus Waypoint 0.5 is created. The GPS and compass headers are then calculated and the same process is carried out as in figure 23.

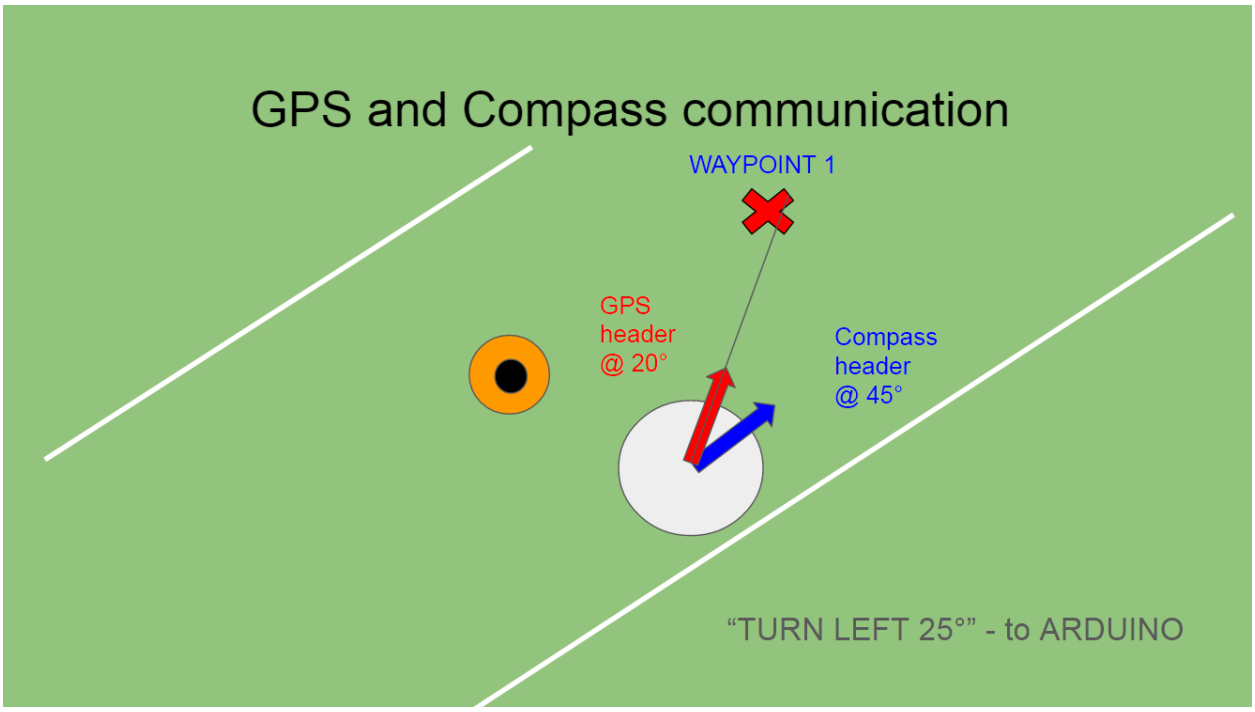


Figure 27: GPS and Compass Communication 4

Figure 26 shows the vehicle once it reaches Waypoint 0.5. Since the path is now clear to its original waypoint the vehicle may enter its calculations for header values and execute as before, however, this time it’s making a left turn of 25° to get on course. The vehicle may safely reach its Waypoint 1.

[ACG]

3.6.3 *Sensor Calculations*

The distance traveled between camera samples is important to take into consideration. Assuming the industry standard 24 frames/second frame rate on a webcam, the distance the vehicle travels between samples can be calculated at any speed. The distance traveled will be highest at max speed. The max speed permitted for the competition is 5 mph or 2.235 m/s. The max distance traveled between frames is 3.667 inches per frame, given by

$$5 \text{ mph} = \frac{88 \text{ in}}{\text{s}} \times \frac{\text{s}}{24 \text{ frames}} = 3.667 \text{ in/frame} \quad (23)$$

This is sufficiently small compared to the size of the vehicle and the width of the track. All the obstacles on the course are stationary so their speed does not have to be considered.

The range of distances where the cameras can see lines on the ground is determined by the height of the camera, field of vision of the camera, and the angle of the camera. Figure 28

shows the camera's field of vision. Here θ_c is the camera's angle, θ_{fv} is the vertical field of view of the camera, and h is the height at which the camera is mounted.

The maximum ground distance seen by the camera is given by:

$$d_{max} = h \tan\left(\theta_c - \frac{\theta_{fv}}{2}\right). \tag{24}$$

The minimum ground distance seen by the camera is given by:

$$d_{min} = h \tan\left(\theta_c + \frac{\theta_{fv}}{2}\right). \tag{25}$$

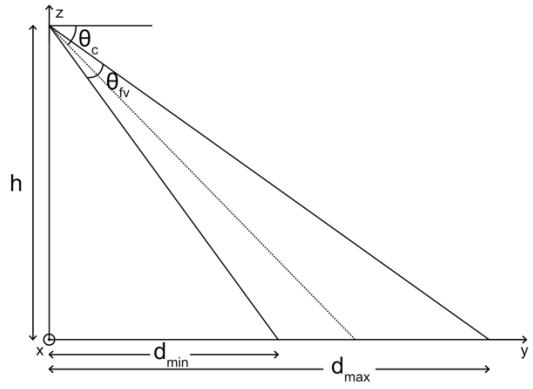


Figure 28. Camera Field of Vision

It is important to minimize the minimum distance to reduce blind spots near the vehicle. A larger maximum distance will give the vehicle more information to use when making decisions.

Two types of LiDAR were considered when designing the IGV, the Pepperl + Fuchs R2000 and the Sick LS290. The Pepperl + Fuchs offers 360° of scanning at selectable frequencies between 10Hz and 50Hz with an angular resolution of $\geq 0.071°$ whereas the Sick LS290 offers 270° of scanning at adjustable angular resolutions of 0.25°, 0.50° and 1°, respectfully. Initially, the precision scanning of the Pepperl + Fuchs R2000 appears to be superior given its angular resolution. Considering the degree of scanning to be more a luxury than a necessity, the speed of detection was the main factor examined for selection.

The Sick LS290 gives three options for angular resolution: 0.25°, 0.5° and 1°. Obviously, for the cleanest detection 0.25° is ideal, however, response time is increased from 13.33ms at 1°, 26.66ms at 0.5° and finally 53.33ms at 0.25°. To ensure the LiDAR will be able to scan quickly at this frequency, the vehicles max speed of 5mph must be considered. The framerate is calculated using Equation (26) given the provided information.

(26)

$$5mph = \frac{88in}{s} \times \frac{0.001s}{ms} \times \frac{53.33ms}{frame} = 4.69 \frac{in}{frame} at\ 0.25°$$
$$5mph = \frac{88in}{s} \times \frac{0.001s}{ms} \times \frac{26.66ms}{frame} = 2.34 \frac{in}{frame} at\ 0.5°$$
$$5mph = \frac{88in}{s} \times \frac{0.001s}{ms} \times \frac{13.33ms}{frame} = 1.17 \frac{in}{frame}\ at\ 1°$$

The Pepperl + Fuchs R2000 provides its scanning potential in an adjustable frequency ranging from 10Hz to 50Hz. The following equations (27) demonstrate the travel distance of the IGV operating at max speed (5 MPH) provided the maximum and minimum scanning frequencies provided by the R2000.

(27)

$$\frac{600\ Scans}{1\ Minute} \times \frac{1\ Minute}{60\ Seconds} = \frac{10\ Scans}{1\ Second} = \frac{1\ Scan}{0.1\ Second}\ @\ 10Hz$$

$$\frac{3000\ Scan}{1\ Minute} \times \frac{1\ Minute}{60\ Seconds} = \frac{50\ Scan}{1\ Second} = \frac{1\ Scan}{0.02\ Second}\ @\ 50Hz$$

Max speed = 5 MPH

$$5\ MPH = \frac{88\ Inches}{1\ Second} \times \frac{0.1\ Second}{1\ Scan} = 8.8\ Inches\ per\ Scan$$

$$5\ MPH = \frac{88\ Inches}{1\ Second} \times \frac{0.02\ Second}{1\ Scan} = 1.76\ Inches\ per\ Scan$$

Equation (26) shows that the vehicle will travel between 1.17 inches to 4.69 inches at max speed utilizing the Sick LS290 at its adjustable angular resolutions. Equipped with the Pepperl + Fuchs R2000 LiDAR the vehicle will travel between 1.76 inches and 8.8 inches as shown in Equation (27). These results may lead to believe that the Sick LS290 is clearly the leader in quickness, however, provided the Pepperl + Fuchs superior angular resolution of $\geq$ 0.071°, the ~0.6 inches per scan is worth the tradeoff for a more accurate reading, especially for detecting objects at a distance.

It should be noted that most LiDAR sensors have a range of well over 30m or ~100 feet. However, reflectivity plays a role in the effective range of the sensor. Upon review of sensor range given certain material, the worst case effective range would be 10m if the laser pulses reflect off a wooden chair or cardboard obstacle on the course. Since 10m is ~32 feet, it is safe to say that the vehicle traveling at max speed shall be able to safely and timely detect an object at any selected angular frequency given the response time of the sensor.

Based on the powering needs of the sensor network, a 12V battery was chosen to be appropriate. The current draw for each sensor was found and recorded in a spreadsheet and

shown in Table 15. The average run time of the vehicle per charge of the battery was estimated to be about 4 hours. This would allow for many runs of the course for testing and troubleshooting and yield plenty of time for the competition. The results in Table 15 reflect this time average for battery life.

Table 15. Sensor Network Power Consumption

	Voltage (V)	Power (W)	Current (A)
Lidar	24	8	0.33
Diff GPS	12	2.8	0.23
USB Hub	12	36	1.00
Cameras	5	10	2.00
Compass	5	0.5	0.01
Encoders	5	0.1	0.02
Arduino	5	2	0.40

$$Current = \frac{Power}{Voltage} = \frac{P_{12V}}{12V} + \frac{P_{24V} * \frac{1}{0.9}}{12V} + \frac{P_{5V} * \frac{1}{0.9}}{12V} \quad (28)$$

$$Capacitance = Current \times 4\ hours \quad (29)$$

Equation (23) assumes a voltage conversion efficiency of 90% as stated with the Anker USB hub's 12V to 5V buck conversion and the Ziumer B01M9JK9HF's 12V to 24V boost conversion. Utilizing information attained through Table 15 and Equations (28) and (29) the 12V battery, should be sized at a minimum of 12.3 Ah to provide power to the sensor network for 4 hours of run time, and will be implemented using two 8Ah batteries in parallel which will provide 16Ah of use or approximately 5 hours and 15 minutes of runtime.

[ACG, ART, JPJ]

3.7 Software Design

3.7.1 *Software Overview*

The software for the intelligent ground vehicle serves three purposes. The first purpose is to collect data from the environment to create a map of the course. The second purpose is to use the created map to make path decisions when navigating the course. The third purpose is to implement a feedback control system for the drive motors.

The environment sensing portion of the software uses input from three cameras pointed from the front and both sides of the vehicle. The primary purpose of the camera input is to detect the lane lines, potholes, and flags which cannot be picked up using distance sensors. The software will process each frame from the camera by first reducing the resolution to a less computationally expensive size. This image will be processed using three parallel filters: high contrast, red pass, and blue pass. The output of the high contrast filter will be used to determine the locations of lane lines and potholes in the image. The output of the red pass and blue pass filters will be used to determine the presence and rough location of flags in the image. Flags will not be located on the ground so location of the flags will be difficult to judge accurately but detecting whether they are in a frame or not in a frame for a particular camera should be enough precision to navigate the course. The lines will be less difficult to measure because they are located on the ground. Since the camera height and angle will be fixed and assuming a flat ground, the physical location of a pixel's color data in real space relative to the vehicle will remain the same. Determining the location of an object in real space (with respect to the vehicle) where its pixel location is known is as easy as looking up the location in a precomputed translation array.

LiDAR is used to detect the distance to 3D objects on the course. LiDAR works by measuring time-of-flight of a laser as it travels to objects and is reflected back to the LiDAR device. A laser will take less time to travel to and back from a nearby object than a distant object. The LiDAR uses a scanning laser and returns a stream of distances of objects at the scanned angles. This stream of distances will be used to measure the size and distance to objects relative to the vehicle.

Once the objects are mapped in relation to the vehicle their positions will be translated into global space. This is done using the GPS location and magnetic orientation of the vehicle at the time the frame was captured. With the objects always mapped in global space it is easier to keep track of objects between frames when the vehicle is in a different location and objects may be in sensor blind spots.

The path planning software module will use the map of objects in global space as well at the current position and orientation of the vehicle to determine the best path towards the GPS waypoints.

```
VideoCapture camera;
R2000 *lidar;
IGVC::Map *map;
main(){
     lidar = new R2000();
     initLidar();
     camera.open(CAMERA_NUMBER);
     gps = initGps();
     arduino = initArduino();
     map = initMap();

     while(1){
          // Handle Lidar
          lidar->ReadSensor(); // happens in another thread
          map->ProcessLidar(lidar->scan); // waits for mutex

          // Handle Cameras
          frame = camera->fetch();
          map->ProcessCamera(frame);

          // Get GPS location
          gps->locate();
          arduino->ReadCompass();

          // Build Map
          map->ProcessLidar(lidar->scan);
          map->ProcessCamera(cameras->objects)
          map->globalize(gps, compass);

          // Pathfinding
          target = map->findPath();

          // Send Data
          Serial.Send(target);
     }
}
initLidar(){
    lidar->Connect(IP_ADDRESS);
    lidar->StartStream();
}
```

Figure 29 – Pseudo Code for Software Main Function

The code for the main loop (main.cpp) and global constant definitions (Constants.h) are shown below.

High-Level Code

Constants.h

```
#define PI 3.1415926

/*****************************************************
 *      CAMERA MODULE
 *****************************************************/
#define CAMERA // enables the camera module

/*      Uncomment to directly put white points on map
        Otherwise line tracing algorithm is used */

// #define BYPASS_LINE_TRACING

/* Constants used for mapping pixel data to real space */
#define PIXEL_WIDTH 480
#define PIXEL_HEIGHT 270
#define FIELD_OF_VISION_H 70.42 // degrees
#define FIELD_OF_VISION_V 43.30 //degrees

#define CAMERA_HEIGHT 1.30175 // meters

static const char* CAMERA_NAME[3] = {
        "center",
        "left",
        "right"
};
static const float CAMERA_X_OFFSET[3] = { // meters
        0, // center camera
        0,//-0.23495, // left camera
        0.2413 // right camera
};

static const float CAMERA_Y_OFFSET[3] = { // meters
        0, // center camera
        0, // left camera
        0 // right camera
};

static const float CAMERA_HORIZONTAL_ANGLE[3] = { // radians from
        0, // center camera
        0, // left camera
        0.698132 // right camera
};

static const float CAMERA_ANGLE[3] = { // degrees from horizontal
        35, // center camera
        35, // left camera
        40 // right camera
};

/* Line tracing algorithm minimum line length */
#define MIN_LINE_LENGTH 50

/*****************************************************
 *      LiDAR MODULE
```

```
 ****************************************************/
#define LIDAR // enables the lidar module

/* Uncomment to disable code that groups lidar points into objects */
//#define REMOVE_LIDAR_OBJECT_DETECTION

#define DEFAULT_LIDAR_IP "169.254.12.9"

#define LIDAR_X_OFFSET 0 //meters
#define LIDAR_Y_OFFSET 0.3683; // meters

/****************************************************
 *      GPS MODULE
 ****************************************************/
//#define GPS // enables the GPS module
#define DEFAULT_GPS_COM "COM9"

/****************************************************
 *      Arduino MODULE
 ****************************************************/
#define ARDUINO // enables compass module
#define DEFAULT_ARDUINO_COM "COM4"


/****************************************************
 *      VISUALIZER
 ****************************************************/
#define VISUALIZER_SCALE 5.0

/* Used to draw the size of the vehicle in the visualizer*/
#define SAFE_FRONT_SPACE 92 //cm
#define SAFE_SIDE_SPACE 31 //cm

#define SAFE_BUBBLE 92 //cm
```

Main.cpp

```
#include "Constants.h"
#include <string>

#ifdef CAMERA
#include "LaneCamera.h"
#endif // CAMERA

#ifdef LIDAR
#include "R2000.h"
#endif // LIDAR

#ifdef GPS
#include "IGVC_GPS.h"
#endif // GPS

#ifdef ARDUINO
#include "IGVC_Arduino.h"
#endif // ARDUINO


#include "Map.h"

#include "Waypoints.h"

#include <stdio.h>
```

```
#include "GL/freeglut.h"

//#define MEM_DEBUG
#ifdef MEM_DEBUG
#define _CRTDBG_MAP_ALLOC
#include <stdlib.h>
#include <crtdbg.h>
#endif //MEM_DEBUG

#ifdef _MSC_VER        // Check if MS Visual C compiler
#   ifndef _MBCS
#      define _MBCS    // Uses Multi-byte character set
#   endif
#   ifdef _UNICODE     // Not using Unicode character set
#      undef _UNICODE
#   endif
#   ifdef UNICODE
#      undef UNICODE
#   endif
#endif // _MSC_VER

#ifdef _MSC_VER                         // Check if MS Visual C compiler
#include <windows.h>  // For MS Windows
#  pragma comment(lib, "opengl32.lib")  // Compiler-specific directive to avoid manually configuration
#  pragma comment(lib, "glu32.lib")     // Link libraries
#  pragma comment(lib, "freeglut.lib")
#endif // _MSC_VER

using namespace std;

// Global Connection Values
int CAM_NUM[3]; //camera id numbers
int CAMS = 1; //default number of cameras
char _LIDAR_IP[] = DEFAULT_LIDAR_IP; //default
char *LIDAR_IP = _LIDAR_IP;
char _GPS_COM[] = DEFAULT_GPS_COM;
char _ARDUINO_COM[] = DEFAULT_ARDUINO_COM;
char *GPS_COM = _GPS_COM;
char *ARDUINO_COM = _ARDUINO_COM;

// Global Objects
#ifdef LIDAR
R2000 *lidar;
#endif // LIDAR

IGVC::Map *mapp;

#ifdef GPS
igvc_gps *gps;
#endif // GPS

#ifdef CAMERA
VideoCapture cap(0); // Video object must be global because it cannot be passed between functions.
#endif // CAMERA

#ifdef ARDUINO
igvc_arduino *ard;
DWORD WINAPI arduinoThread(LPVOID lpParam);
#endif // ARDUINO

bool isRunning;
```

```
queue<waypoint> waypoints;

// Mutex for Multi-threading
HANDLE lidarMutex;
HANDLE arduinoMutex;

// Functions
#ifdef LIDAR
bool initLidar();
DWORD WINAPI lidarThread(LPVOID lpParam);
#endif // LIDAR

void close();
void display();
void displayGrid();

int main(int argc, char *argv[]) {

#ifdef MEM_DEBUG
		_CrtSetDbgFlag ( _CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF );
		_CrtSetReportMode( _CRT_ERROR, _CRTDBG_MODE_DEBUG );
#endif // MEM_DEBUG


	// Process arguments
	for (int i = 0; i < argc; i++){
		if (strcmp(argv[i], "-lip") == 0){
			LIDAR_IP = argv[i+1];
		}else if(strcmp(argv[i], "-cam") == 0){
			CAM_NUM[0] = atoi(argv[i+1]);
		}else if(strcmp(argv[i], "-ard") == 0){
			ARDUINO_COM = argv[i+1];
		}else if(strcmp(argv[i], "-gps") == 0){
			GPS_COM = argv[i+1];
		}else if(strcmp(argv[i], "-rcam") == 0){
			CAM_NUM[2] = atoi(argv[i+1]);
		}else if(strcmp(argv[i], "-lcam") == 0){
			CAM_NUM[1] = atoi(argv[i+1]);
		}else if(strcmp(argv[i], "-cams") == 0){
			CAMS = atoi(argv[i+1]);
		}
	}

	// Init Map
	mapp = new IGVC::Map();

#ifdef LIDAR
	// Init Lidar
	lidar = new R2000();
	if(!initLidar())
		return -1;
#endif // LIDAR

#ifdef GPS
	// Init GPS
	gps = new igvc_gps();
	gps->COM_PORT = GPS_COM;
	gps->setup();
#endif // GPS

#ifdef ARDUINO
	ard = new igvc_arduino();
```

```
        ard->COM_PORT = ARDUINO_COM;
        ard->setup();
        //CreateThread(NULL, 0, arduinoThread, 0 , 0 , NULL);
        //arduinoMutex = CreateMutex(NULL, false, NULL);
#endif // ARDUINO

#ifdef CAMERA
        /********************** Video Setup ***************************************/

        cap.open(CAM_NUM[0]);
                cap.set(CV_CAP_PROP_FRAME_WIDTH, FRAME_W); // 432x240 (x3) = 311,040 SubPixels
                cap.set(CV_CAP_PROP_FRAME_HEIGHT, FRAME_H);
                cap.set(CV_CAP_PROP_FPS, 30);

                if (!cap.isOpened()) {
                        CV_Assert("Camera failed to open!");
                        return -1;
                }

        //cap[0].set(CV_CAP_PROP_SETTINGS, 1);

        /****************** End Video Setup ***************************************/
#endif // CAMERA

#ifdef LIDAR
        CreateThread(NULL, 0, lidarThread, 0, 0, NULL);
        lidarMutex = CreateMutex(NULL, false, NULL);
#endif // LIDAR

        //Read Waypoints

        waypoints = readWaypoints("waypoints.txt");

        // Init GUI
        glutInit(&argc, argv);                 // Initialize GLUT
        glutCreateWindow("OpenGL Setup Test"); // Create a window with the given title
        glutInitWindowSize(320, 320);   // Set the window's initial width & height
        glutInitWindowPosition(50, 50); // Position the window's initial top-left corner
        glutDisplayFunc(display); // Register display callback handler for window re-paint
        glutCloseFunc(close);
        glutMainLoop();           // Enter the infinitely event-processing loop

        return 0;
}

#ifdef LIDAR
bool initLidar(){
        //Init Lidar
        //char ip[] = LIDAR_IP;

        bool success = true;
        success = lidar->Connect(LIDAR_IP);

        if(success){
                success = lidar->StartStream();
        }else{
                printf("ERROR: COULD NOT CONNECT TO R2000\n");
                return false;
        }

        if (!success) {
                printf("ERROR: COULD NOT START DATA STREAM FROM R2000\n");
```

```
        }
        return success;
}
#endif // LIDAR

void display() {
        glClearColor(0.0f, 0.0f, 0.0f, 1.0f); // Set background color to black and opaque
        glClear(GL_COLOR_BUFFER_BIT);         // Clear the color buffer

                /********************* Main Loop ***********************************/
#ifdef CAMERA
                /* Camera */
                Mat frame;
                // There is a built in buffer. Throw out two frames before processing the latest one.

                cap >> frame;
                cap >> frame;
                cap >> frame;
                mapp->ProcessCamera(frame, CAMERA_NAME[0], 0);

#endif // CAMERA
#ifdef LIDAR
                /* Lidar */
                WaitForSingleObject(lidarMutex, INFINITE);
                mapp->ProcessLidar(lidar->scan);
                ReleaseMutex(lidarMutex);
#endif // LIDAR
#ifdef GPS
                /* GPS */
                float system_latitude, system_longitude;
                gps->update_lat_long();
                system_latitude = gps->igvc_latitude;
                system_longitude = gps->igvc_longitude;
#endif // GPS

#ifdef ARDUINO
                /* Compass */

                /* Motor Control */
                /* Speed: -64 (Full Reverse) to +64 (Full Forward) */
                /* Angle: -5 (Full Left, No Right) to +5 (Full Right, No Left) with stepping amounts. Zero
is forward and backward. */


#endif // ARDUINO

                /* Mapping */
                mapp->FindPath();

                float slope = (mapp->tar_y-500.0-SAFE_FRONT_SPACE)/(mapp->tar_x-500.0);

#ifdef ARDUINO
                //WaitForSingleObject(arduinoMutex, INFINITE);
                // FOR DEMO -- PRESET ANGLE TO ZERO, WILL BE ADJUSTED BY PATH SLOPE
                if((slope >= 1.0) && (slope <= 1.5)){
                        ard->angle_toArduino = 5;
                }else if((slope > 1.5) && (slope <= 2.0)){
                        ard->angle_toArduino = 4;
                }else if((slope > 2.0) && (slope <= 2.5)){
                        ard->angle_toArduino = 3;
                }else if((slope > 2.5) && (slope <= 3.5)){
                        ard->angle_toArduino = 2;
```

```
                }else if((slope > 3.5) && (slope <= 5.0)){
                        ard->angle_toArduino = 1;
                }else if(slope > 5.0){
                        ard->angle_toArduino = 0;
                }else if(slope < -5.0){
                        ard->angle_toArduino = 0;
                }else if((slope > -5.0) && (slope <= -3.5)){
                        ard->angle_toArduino = -1;
                }else if((slope > -3.5) && (slope <= -2.5)){
                        ard->angle_toArduino = -2;
                }else if((slope > -2.5) && (slope <= -2.0)){
                        ard->angle_toArduino = -3;
                }else if((slope > -2.0) && (slope <= -1.5)){
                        ard->angle_toArduino = -4;
                }else if((slope > -1.5) && (slope <= -1.0)){
                        ard->angle_toArduino = -5;
                }else{
                        ard->speed_toArduino = 0;
                        ard->angle_toArduino = 0;
                }
                ard->speed_toArduino = 10;
                ard->send_speed_angle_distance();
                //ReleaseMutex(arduinoMutex);

                /* Send Path */
                // FOR DEMO -- KEEP THE SPEED LOW
#endif // ARDUINO

                /* Reset Map */
                mapp->Draw();
                mapp->Clear();


        displayGrid();
        glBegin(GL_POLYGON);
        glColor3f(0.0f,1.0f,0.0f);
        glVertex2d(-SAFE_SIDE_SPACE/CM_PER_M/VISUALIZER_SCALE,-
SAFE_FRONT_SPACE/CM_PER_M/VISUALIZER_SCALE);
        glVertex2d(-SAFE_SIDE_SPACE/CM_PER_M/VISUALIZER_SCALE,SAFE_FRONT_SPACE/CM_PER_M/VISUALIZER_SCALE);
        glVertex2d(SAFE_SIDE_SPACE/CM_PER_M/VISUALIZER_SCALE,SAFE_FRONT_SPACE/CM_PER_M/VISUALIZER_SCALE);
        glVertex2d(SAFE_SIDE_SPACE/CM_PER_M/VISUALIZER_SCALE,-SAFE_FRONT_SPACE/CM_PER_M/VISUALIZER_SCALE);
        glEnd();

   glFlush();  // Render now
   glutPostRedisplay(); // Call this function again (makes this a program loop)
}

void close(){
        // Clean up
#ifdef LIDAR
        delete lidar;
        lidar = NULL;
#endif

        delete mapp;
        mapp = NULL;
}

#ifdef LIDAR
DWORD WINAPI lidarThread(LPVOID lpParam){
        while(1){
                WaitForSingleObject(lidarMutex, INFINITE);
```

```
                lidar->Read_Sensor();
                ReleaseMutex(lidarMutex);
        }
        return true;
}
#endif

void displayGrid(){
        for(int i = -VISUALIZER_SCALE+1; i < VISUALIZER_SCALE; i++){
                // y meter lines
                glBegin(GL_LINE_STRIP);
                glColor3f(0.5f,0.5f,0.5f);
                glVertex2d(-1, i/VISUALIZER_SCALE);
                glVertex2d(1, i/VISUALIZER_SCALE);
                glEnd();
        }

        for(int i = -VISUALIZER_SCALE+1; i < VISUALIZER_SCALE; i++){
                // x meter lines
                glBegin(GL_LINE_STRIP);
                glColor3f(0.5f,0.5f,0.5f);
                glVertex2d(i/VISUALIZER_SCALE, -1);
                glVertex2d(i/VISUALIZER_SCALE, 1);
                glEnd();
        }
}

#ifdef ARDUINO
DWORD WINAPI arduinoThread(LPVOID lpParam){
        while(1){

                //Read
                //float heading;


                //heading = ard.compass_heading;
                //isRunning = ard.errorcode;
                WaitForSingleObject(arduinoMutex, INFINITE);
                //ard->get_compass_and_errorcode();
                Sleep(1);
                //Write
                ard->speed_toArduino = 10;
                ard->send_speed_angle_distance();

                ReleaseMutex(arduinoMutex);
        }
                return true;
}
#endif
```

[ART, CRE]

3.7.2 *Hardware Interfacing*

The software for the intelligent ground vehicle is divided between a PC and an Arduino microcontroller. The software on the PC is responsible for interfacing directly with the cameras, LiDAR, and GPS. The software on the Arduino will interface with the compass, tachometers, motor drivers, safety light, and estop circuit. The PC software handles the image acquisition,

object detection, real-world mapping transformations, and path planning. The software on the Arduino handles the motor control loop and receives input from the PC indicating the desired angle and speed of travel. The Arduino must also stream the compass data to the PC because it is required for mapping objects to real-space.

Each of the cameras, the LiDAR, the GPS, and compass reading routines run on a separate thread. GPS data and compass data must be stored on the PC software in a semaphore-locked buffer to be accessed by the mapping routine. This allows the sensors acquisition routines to run on their own threads while data can be accessed across threads.

Several drivers were written to interface with the various sensors onboard the vehicle. The serial interfaces between the Arduino micro controller, NovaTel Differential GPS and the PC require the creation of serial protocols included in the vehicle's software. Seen below, Figure 30 displays a snapshot of the C++ code used to connect to the Arduino via serial port.

```
#pragma once
class igvc_arduino{
public:
	igvc_arduino(void){};
	~igvc_arduino(void){ CloseHandle(this->COM_Handle); };

	std::string COM_PORT;

	HANDLE COM_Handle;
	DCB PortDCB;
	COMMTIMEOUTS CommTimeouts;

	char readBuff[500];

	// INCOMING PARAMETERS
	float compass_heading;
	bool errorcode;

	// OUTGOING PARAMETERS
	int speed_toArduino;
	int angle_toArduino;
	float distance_toArduino;

	bool flag_one;  // Used to signal that the bestposa write command has already been issued.

	void setup(){
		// Handle Setup
		this->handle_setup();
		// DCB Setup
		this->dcb_setup();
		// Timeout Setup
		this->timeout_setup();
		// Clear TX/RX Buffers
		this->purge_comms();
		// Initialize Vars.
		this->flag_one = 0;
		memset(readBuff, 0, sizeof(readBuff));
		this->compass_heading = 0.0;
		this->errorcode = 0;
	}

	void handle_setup(){
		// Performs a setup of the serial port's handle. Called in setup().
		this->COM_Handle = CreateFile(this->COM_PORT.c_str(), GENERIC_READ | GENERIC_WRITE, (DWORD)NULL, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
		//this->COM_Handle = CreateFile(TEXT("COM4"), GENERIC_READ | GENERIC_WRITE, (DWORD)NULL, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
		if(!SetupComm(this->COM_Handle, (DWORD)2048, (DWORD)2048)){
			cout << "Failed in Arduino SetupComm" << endl;
		}
	}
```

Figure 30 – Snapshot of Arduino Serial Interface Code (C++)

The 'igvc_arduino' class utilizes the technique of managing serial ports using handles. Handles are commonly used throughout many applications such as serial ports, ethernet ports/sockets and file interaction, amongst others. The serial timings must be carefully chosen on a case-by-case basis to be compatible with the device in which communication is taking place. For the Arduino, a baud rate of 9600 along with byte length of 8 and no parity and one stop bit creates a connection with the Arduino Mega onboard the vehicle. Shown below, Figure 31 illustrates the method behind setting up the parameters and timing for the Arduino serial communications port in C++.

```
void dcb_setup(){
        // Performs a setup of the serial port's DCB parameters. Called in setup().
        this->PortDCB.DCBlength = sizeof(DCB);
        GetCommState(this->COM_Handle, &this->PortDCB);
        this->PortDCB.BaudRate = 9600;                  // BAUD
        this->PortDCB.ByteSize = 8;                             // BYTE-LENGTH
        this->PortDCB.Parity = NOPARITY;                        // PARITY
        this->PortDCB.StopBits = ONESTOPBIT;            // STOP
        if(!SetCommState(this->COM_Handle, &this->PortDCB)){
                cout << "Failed in Arduino SetCommState" << endl;
        }
}

void timeout_setup(){
        // Performs a setup of the serial port's timeout parameters. Called in setup();
        // Note: These were determined to work through experimentation with the NovaTel ProPak V3
        GetCommTimeouts(this->COM_Handle, &this->CommTimeouts);
        this->CommTimeouts.ReadIntervalTimeout = 10;                    // Time between incoming chars.
        this->CommTimeouts.ReadTotalTimeoutConstant = 200;      // Maximum wait time for incoming msg.
        this->CommTimeouts.ReadTotalTimeoutMultiplier = 1;
        this->CommTimeouts.WriteTotalTimeoutConstant = 25;
        this->CommTimeouts.WriteTotalTimeoutMultiplier = 0;
        if(!SetCommTimeouts(this->COM_Handle, &this->CommTimeouts)){
                cout << "Failed in Arduino SetCommTimeouts" << endl;
        }
}
```

Figure 31 – Arduino Serial Parameter and Timing Setup Code (C++)

LiDAR

The Pepperl + Fuchs R2000 LiDAR connects to the PC via Ethernet and communicates using TCP. The PC software configures the LiDAR in a startup routine using HTTP requests. The computer sends the ‘request_handle_tcp’ command to the LiDAR to request a socket. Then the PC connects to the socket. Finally, the PC sends the ‘start_scanoutput’ command to the LiDAR, and the LiDAR starts sending scan packets to the PC. The LiDAR is configured to send Type A packets which only have distance data. Angles are calculated as an offset from the first distance of the sweep.

```
class R2000{
     has ip_address
     has private handle
     has private port
     has private ConnectSocket
     has public Scan

     R2000 init(){
          return self
     }
     void Connect(char* ip){
          Connect to R2000 http webpage
     }
     bool StartStream(){
          socket <- request handle
          if handle or socket is invalid
               return failure
          send start_scanoutput command with handle
          return success
     }
     bool StopStream(){
          send stop code
          return success
     }
     bool ReleaseHandle(){
          release socket handle on Lidar
          return success
     }
     void Read_Sensor(){
          FeedWatchdog();
          read socket
          interpret header data
          fill scan array with sweep data
     }
     bool FeedWatchdog(){
          send watchdog feed message
          return success
```

Figure 32 –Pseudo Code for Pepperl+Fuchs R2000 Lidar

```
class Scan{
     has private scan_distance
     has private scan_distance_n1
     has private scan_distance_n2
     has private scan_size
     has private derivative
     has private scan_x
     has private scan_y
     has private scan_magnitude;
     has private scan_angle;
     has public Scan

     Scan init(){
          allocate all necessary memory
          return self
     }
     UpdateScanData(){
          Put latest scan in scan_distance_n2
          TimeFilter();
          scan_distance_n1 = scan_distance_n2
          do a point average and remove error points and
          reduce number of points in scan
          calculate magnitude and angle in reduced scan
          calculate scan_x and scan_y from scan_m and
          scan_angle
     }
     TimeFilter(){
          for each distance in scan_distance{
               average scan_distance_n1 and scan_distance_n2
               store in scan_distance
          }
     }
}
```

The final code for the LiDAR sensor interfacing is contained in the R2000 class. The code is shown below.

R2000.h

```
#include <string>
#include <cstdint>
#include "Scan.h"

#ifdef WIN32
#include <WinSock2.h>
```

```
#include <ws2tcpip.h>
#pragma comment(lib, "Ws2_32.lib")
#endif

#define DEFAULT_BUFLEN 1404
#define TIMESTAMP_RAW_BYTE_OFFSET 14
#define FIRST_ANGLE_BYTE_OFFSET 44
#define HEADER_SIZE_BYTES 60

const uint16_t magic = 0xa25c;

typedef struct R2000_header{
        uint16_t magic;
        uint16_t packet_type;
        uint32_t packet_size;
        uint16_t header_size;
        uint16_t scan_number;
        uint16_t packet_number;
        uint64_t timestamp_raw;
        uint64_t timestamp_sync;
        uint32_t status_flags;
        uint32_t scan_frequency;
        uint16_t num_points_scan;
        uint16_t num_points_packet;
        uint16_t first_index;
        int32_t first_angle;
        int32_t angualar_increment;
        uint32_t output_status;
        uint32_t field_status;
        }header;

class R2000
{
private:
        std::string ip_addr;
        std::string handle;
        std::string port;
        int last_kick;
#ifdef WIN32
        WSADATA wsaData;
        SOCKET ConnectSocket;
#endif

public:
        Scan *scan;

        R2000(void);
        ~R2000(void);
        bool Connect(char *ip);
        void Read_Sensor(); //reads one scan
        bool StartStream();
        bool StopStream();
        bool FeedWatchdog();
        bool ReleaseHandle();
private:
        bool getPort(char* ip);
};
```

R2000.cpp

```
#include "R2000.h"
#include <string>
```

```
#include <stdio.h>
#include <cstdint>
#include <memory>
#include "curl/curl.h"
#include "json/json.h"
#include <time.h>

#define SOCKET_READ_TIMEOUT_SEC 1

using namespace std;
#pragma comment(lib, "libcurl.lib")

//#define DEBUG

#define URL_PREFIX "http://"
#define GET_SOCKET_REQUEST_URL "/cmd/request_handle_tcp"
#define START_STREAM_URL "/cmd/start_scanoutput?handle="
#define STOP_STREAM_URL "/cmd/stop_scanoutput?handle="
#define FEED_WATCHDOG_URL "/cmd/feed_watchdog?handle="
#define RELEASE_HANDLE_URL "/cmd/release_handle?handle="

#define WATCHDOG_KICK_PERIOD 1 //seconds

namespace
{
    std::size_t callback(
            const char* in,
            std::size_t size,
            std::size_t num,
            std::string* out)
    {
        const std::size_t totalBytes(size * num);
        out->append(in, totalBytes);
        return totalBytes;
    }
}

size_t write_data(void *buffer, size_t size, size_t nmemb, void *userp){
        return size*nmemb;
}

/*-----------------------------------------------------------------
 -      Windows Only Definitions
 -----------------------------------------------------------------*/
#ifdef WIN32
// Requires WinSock

/*****************************************
 *      Name:    R2000 [Constructor]
 *      Description: Constructs the R2000 Object
 *****************************************/
R2000::R2000(void)
{
        scan = NULL;
        int iResult;

        //initialize Windock
        iResult = WSAStartup(MAKEWORD(2,2), &wsaData);
        if(iResult != 0){
                printf("WSAStartup failed: %d\n", iResult);
                throw -1;
        }
```

```
}

/*****************************************
 *      Name:    R2000 [Destructor]
 *      Description: Destructs the R2000 Object
 *****************************************/
R2000::~R2000(void)
{
        //Remember to shut down the port...
        closesocket(ConnectSocket);
        WSACleanup();

        // Release handle to stop lidar stream
        this->ReleaseHandle();

        delete scan;
}

/*********************************************************************
 *      Name:    Connect
 *      Input:                  ip - Internet Protocol Address of R2000 Lidar
 *      Description:    Initializes and begins connection to R2000
 *********************************************************************/
bool R2000::Connect(char *ip){
        ip_addr = string(ip); // set ip member variable

        // try to get port number for connection via R2000 built in webpage
        if(!getPort(ip)){
                return false;
        }

        // Set up connection
        struct addrinfo *result = NULL,
                                        *ptr = NULL,
                                        hints;

        ZeroMemory(&hints, sizeof(hints));
        hints.ai_family = AF_INET;
        hints.ai_socktype = SOCK_STREAM;
        hints.ai_protocol = IPPROTO_TCP;

        int iResult = 0;
        // Resolve the server address and port
        iResult = getaddrinfo(ip, port.c_str(), &hints, &result);
        if (iResult != 0) {
                printf("getaddrinfo failed: %d\n", iResult);
                WSACleanup();
                return false;
        }

        ConnectSocket = INVALID_SOCKET;

        // Attempt to connect to the first address returned by
        // the call to getaddrinfo
        ptr=result;

        // Create a SOCKET for connecting to server
        ConnectSocket = socket(ptr->ai_family, ptr->ai_socktype,
                ptr->ai_protocol);

        if (ConnectSocket == INVALID_SOCKET) {
                printf("Error at socket(): %ld\n", WSAGetLastError());
```

```
                freeaddrinfo(result);
                WSACleanup();
                return false;
        }

        // Connect to server.
        iResult = connect( ConnectSocket, ptr->ai_addr, (int)ptr->ai_addrlen);
        if (iResult == SOCKET_ERROR) {
                closesocket(ConnectSocket);
                ConnectSocket = INVALID_SOCKET;
        }

        // If connection failed
        // free the resources returned by getaddrinfo and print an error message
        freeaddrinfo(result);

        if (ConnectSocket == INVALID_SOCKET) {
                printf("Unable to connect to server!\n");
                WSACleanup();
                return false;
        }

        // Copied from Microsoft tutorial (is this necessary?)
        // shutdown the connection for sending since no more data will be sent
        // the client can still use the ConnectSocket for receiving data
        iResult = shutdown(ConnectSocket, SD_SEND);
        if (iResult == SOCKET_ERROR) {
                printf("shutdown failed: %d\n", WSAGetLastError());
                closesocket(ConnectSocket);
                WSACleanup();
                return false;
        }

        DWORD timeout = SOCKET_READ_TIMEOUT_SEC * 1000;
        setsockopt(ConnectSocket, SOL_SOCKET, SO_RCVTIMEO, (char*)&timeout, sizeof(timeout));

        return true;
}

/**********************************************************************
 *      Name:   Read Sensor
 *      Description:    Reads sensor data and adds it to Scan object
 **********************************************************************/
void R2000::Read_Sensor(){
        // Kick Watchdog every second
        if(time(0) > last_kick + WATCHDOG_KICK_PERIOD){
                this->FeedWatchdog();
                last_kick = time(0);
        }

        int recvbuflen = DEFAULT_BUFLEN;
        char recvbuf[DEFAULT_BUFLEN];
        uint16_t magic_bytes;

        int iResult = 0;
        bool quit = false;
        int pointsPerScan = 1, pointsRead = 0;
        while(pointsRead < pointsPerScan && !quit){
                // Read Header
                iResult = recv(ConnectSocket, recvbuf, recvbuflen, 0);
                memcpy(&magic_bytes, recvbuf, 2);
```

```
                if (iResult > 0 && magic_bytes == magic ) // Check for magic byte 0xA25C (other data seems
to be getting in buffer?)
                {
                        struct R2000_header head;
                        //R2000Packet packet = R2000Packet(recvbuf, iResult / 2);
                        // Copy header to struct
                        // Adjust for 32-bit alignment on PC struct
                        memcpy(&head, recvbuf, TIMESTAMP_RAW_BYTE_OFFSET);
                        memcpy(&head.timestamp_raw, &recvbuf[TIMESTAMP_RAW_BYTE_OFFSET],
FIRST_ANGLE_BYTE_OFFSET - TIMESTAMP_RAW_BYTE_OFFSET);
                        memcpy(&head.first_angle, &recvbuf[FIRST_ANGLE_BYTE_OFFSET], HEADER_SIZE_BYTES -
FIRST_ANGLE_BYTE_OFFSET);

#ifdef DEBUG
                        printf("Magic %04X\n", head.magic);
                        printf("Packet Type %c\n", head.packet_type);
                        printf("Packet Size %d\n", head.packet_size);
                        printf("Header Size %d\n", head.header_size);
                        printf("Scan Number %d\n", head.scan_number);
                        printf("Packet Number %d\n", head.packet_number);
                        printf("Timestamp Raw %d\n", head.timestamp_raw);
                        printf("Timestamp Sync %d\n", head.timestamp_sync);
                        printf("Status Flags %04X\n", head.status_flags);
                        printf("Scan Frequency %d (1/1000 Hz)\n", head.scan_frequency);
                        printf("Number of Scan Points within Complete Scan: %d\n", head.num_points_scan);
                        printf("Number of Scan Points in Packet %d\n", head.num_points_packet);
                        printf("First Index: %d\n", head.first_index);
                        printf("First Angle: %d (1/10,000 deg)\n", head.first_angle);
                        printf("Angular Increment %d\n", head.angualar_increment);
                        printf("Output Status: %04X\n", head.output_status);
                        printf("Field Status: %04X\n", head.field_status);
#endif
                        if(scan == NULL){
                                // Allocate a new array for with packet.getNumPointsScan() elements
                                scan = new Scan(head.num_points_scan);
                        }

                        //copy buffer_size-header_size bytes from buffer[start_of_data] to scan array
                        scan->UpdateScanData(head.first_index, recvbuf, head.header_size,
(head.packet_size-head.header_size));
                        //memcpy(&scan[head.first_index], &recvbuf[head.header_size], (head.packet_size-
head.header_size));

#ifdef DEBUG
                        for(int i = 0; i <(iResult-head.header_size)/4; i++){
                                if(scan[head.first_index+i] != 0xFFFFFFFF){
                                        printf("%u mm @ %f\n", scan[head.first_index+i], -
180+0.1*(head.first_index+i));
                                }else{
                                        printf("ERROR POINT\n");
                                }
                        }
#endif
                        pointsPerScan = head.num_points_scan;
                        pointsRead += head.num_points_packet;


                }
                else if (iResult < 0){
                        printf("recv failed: %d\n", WSAGetLastError());
                        quit = true;
                }
```

```
        }
}
#endif //WIN32

/*-----------------------------------------------------------------
 -      Platform independent
 -----------------------------------------------------------------*/
 /*************************************************************************
 *      Name:   getPort
 *      Input:  ip - Internet Protocol Address of R2000
 *      Output:         returns true on success
 *      Description:    Asks the R2000 to open a TCP streaming port
 *************************************************************************/
bool R2000::getPort(char* ip){
        CURL *curl = curl_easy_init();
        std::string url = URL_PREFIX;
        url.append(ip);
        url.append(GET_SOCKET_REQUEST_URL);

        // Response information.
    int httpCode(0);
    std::unique_ptr<std::string> httpData(new std::string());

        if (curl)
        {
                CURLcode res;
                curl_easy_setopt(curl, CURLOPT_URL, url.c_str());
                curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, callback);
                curl_easy_setopt(curl, CURLOPT_WRITEDATA, httpData.get());
                res = curl_easy_perform(curl);
                curl_easy_getinfo(curl, CURLINFO_RESPONSE_CODE, &httpCode);
                curl_easy_cleanup(curl);

                //Handle Response
                if (httpCode == 200)
                {
                        printf("\nGot successful response from %s\n", url.c_str());

                        // Response looks good - done using Curl now.  Try to parse the results
                        // and print them out.
                        Json::Value jsonData;
                        Json::Reader jsonReader;

                        if (jsonReader.parse(*httpData, jsonData))
                        {
                                printf("Successfully parsed JSON data\n");
                                printf("\nJSON data received:\n");
                                printf("%s\n", jsonData.toStyledString().c_str());

                                port = jsonData["port"].asString();
                                handle = jsonData["handle"].asString();

                                printf("Parsed:\n");
                                printf("\tPort string: %s\n",port.c_str());
                                printf("\tHandle string: %s\n\n", handle.c_str());
                                return true;
                        }
                        else
                        {

                                printf("Could not parse HTTP data as JSON\n");
                                printf("HTTP data was:\n%s\n", *httpData.get());
                                return false;
```

```
                        }
                }
                else
                {
                        printf("Couldn't GET from %s - exiting\n", url);
                        return false;
                }
        }
        else
        {
                printf("problem with curl\n");
        }
        return false;
}

 /**************************************************************************
 *      Name:   StartStream
 *      Output:         returns true on success
 *      Description:    Asks the R2000 to start streaming
 **************************************************************************/
bool R2000::StartStream(){
        CURL *curl = curl_easy_init();
        std::string url = URL_PREFIX;
        url.append(ip_addr);
        url.append(START_STREAM_URL);
        url.append(handle);

        // Response information.
    int httpCode(0);
    std::unique_ptr<std::string> httpData(new std::string());

        if (curl){
                CURLcode res;
                curl_easy_setopt(curl, CURLOPT_URL, url.c_str());
                curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, write_data);
                res = curl_easy_perform(curl);
                curl_easy_getinfo(curl, CURLINFO_RESPONSE_CODE, &httpCode);
                curl_easy_cleanup(curl);

                //Handle Response
                if (httpCode == 200)
                {
#ifdef DEBUG
                        printf("\nGot successful response from %s\n", url.c_str());
#endif
                        return true;
                }
                else
                {
                        printf("Couldn't start stream %s\n", url.c_str());
                        return false;
                }
        }
        return false;
}

 /**************************************************************************
 *      Name:   StopStream
 *      Output:         returns true on success
 *      Description:    Asks the R2000 to stop streaming
 **************************************************************************/
bool R2000::StopStream(){
```

```
        CURL *curl = curl_easy_init();
        std::string url = URL_PREFIX;
        url.append(ip_addr);
        url.append(STOP_STREAM_URL);
        url.append(handle);

        // Response information.
    int httpCode(0);
    std::unique_ptr<std::string> httpData(new std::string());

        if (curl){
                CURLcode res;
                curl_easy_setopt(curl, CURLOPT_URL, url.c_str());
                curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, write_data);
                res = curl_easy_perform(curl);
                curl_easy_getinfo(curl, CURLINFO_RESPONSE_CODE, &httpCode);
                curl_easy_cleanup(curl);

                //Handle Response
                if (httpCode == 200)
                {
#ifdef DEBUG
                        printf("\nGot successful response from %s\n", url.c_str());
#endif
                        return true;
                }
                else
                {
                        printf("Couldn't stop stream %s\n", url.c_str());
                        return false;
                }
        }
        return false;
}

 /*****************************************************************************
 *      Name:    FeedWatchdog
 *      Output:          returns true on success
 *      Description:     Feeds the R2000's streaming watchdog
 *****************************************************************************/
bool R2000::FeedWatchdog(){
        CURL *curl = curl_easy_init();
        std::string url = URL_PREFIX;
        url.append(ip_addr);
        url.append(FEED_WATCHDOG_URL);
        url.append(handle);

        // Response information.
    int httpCode(0);
    std::unique_ptr<std::string> httpData(new std::string());

        if (curl){
                CURLcode res;
                curl_easy_setopt(curl, CURLOPT_URL, url.c_str());
                curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, write_data);
                curl_easy_setopt(curl, CURLOPT_TIMEOUT, 1);
                res = curl_easy_perform(curl);
                curl_easy_getinfo(curl, CURLINFO_RESPONSE_CODE, &httpCode);
                curl_easy_cleanup(curl);

                //Handle Response
                if (httpCode == 200)
```

```
                {
#ifdef DEBUG
                        printf("\nGot successful response from %s\n", url.c_str());
#endif
                        return true;
                }
                else
                {
                        printf("Couldn't feed watchdog. %s\n", url.c_str());
                        return false;
                }
        }
        return false;
}

 /**************************************************************************
 *      Name:   Release Handle
 *      Output:         returns true on success
 *      Description:    Closes connection with R2000
 **************************************************************************/
bool R2000::ReleaseHandle(){
        CURL *curl = curl_easy_init();
        std::string url = URL_PREFIX;
        url.append(ip_addr);
        url.append(RELEASE_HANDLE_URL);
        url.append(handle);

        // Response information.
    int httpCode(0);
    std::unique_ptr<std::string> httpData(new std::string());

        if (curl){
                CURLcode res;
                curl_easy_setopt(curl, CURLOPT_URL, url.c_str());
                curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, write_data);
                res = curl_easy_perform(curl);
                curl_easy_getinfo(curl, CURLINFO_RESPONSE_CODE, &httpCode);
                curl_easy_cleanup(curl);

                //Handle Response
                if (httpCode == 200)
                {
#ifdef DEBUG
                        printf("\nGot successful response from %s\n", url.c_str());
#endif
                        return true;
                }
                else
                {
                        printf("Couldn't release handle. %s\n", url.c_str());
                        return false;
                }
        }
        return false;
}
```

The code that handles the processing of the scan data successfully received from the R2000 LiDAR is contained in the Scan class. The Scan class code is shown below. The two most recent scans received from the lidar are averaged together and used as the current scan. The current scan is run through an averaging process which reduces the number of points in the scan to

reduce computation length while improving accuracy of the data and removing points which contain an error reading.

Scan.h

```
#include <cstdint>

class Scan
{
private:
        uint32_t *scan_distance; // in mm - at time N (the time that this class outputs though
Scan_Distance function)
        uint32_t *scan_distance_n1; // in mm - buffer N+1
        uint32_t *scan_distance_n2; // in mm - buffer N+2
        int scan_size; // number of points per lidar scan (unadjusted)

        float *der;
        float *scan_x;
        float *scan_y;
        float *scan_m;
        float *scan_angle;

        void TimeFilter();

public:
        Scan(int num_points_per_scan);
        ~Scan(void);
        void UpdateScanData(int start_index, char *buf, int buf_offset, int length); // Called by R2000 to
update the data in the scan
        const int ScanSize(); // Outputs adjusted scan size after filters
        float * ScanDistance(); // Outputs adjusted scan distances (filtered)
        float *ScanX(); // Outputs X coordinates at each angle (filtered)
        float *ScanY(); // Outputs Y coordinates at each angle (filtered)
        float *ScanAngle(); // Outputs angles corresponsing to ScanX, ScanY, and ScanDistances (filtered)
        float *Derivative(); // Outputs dy/dx derivative (filtered)
};
```

Scan.cpp

```
#include "Scan.h"
#include <memory>

#define MAX_RANGE_MM 20000 // filters extranous data points

#define MM_PER_METER 1000.0 // standard conversion between meters and millimeters
#define REDUCTION 5 // Number of points to average in filter
#define SCAN_RESOLUTION (360.0/scan_size)

/*****************************************
 *      Name:   Scan [Constructor]
 *      Inputs: Number of points per lidar scan
 *      Description: Constructs the Scan Object
 *****************************************/
Scan::Scan(int num_points_per_scan)
{
        // Allocate needed memory

        // Unfiltered Buffers (3)
        scan_distance = (uint32_t *) malloc(num_points_per_scan*sizeof(uint32_t));
        scan_distance_n1 = (uint32_t *) malloc(num_points_per_scan*sizeof(uint32_t));
        scan_distance_n2 = (uint32_t *) malloc(num_points_per_scan*sizeof(uint32_t));
```

```
        // Filtered Arrays (5)
        scan_x = (float *) malloc(num_points_per_scan*sizeof(float)/REDUCTION);
        scan_y = (float *) malloc(num_points_per_scan*sizeof(float)/REDUCTION);
        scan_m = (float *) malloc(num_points_per_scan*sizeof(float)/REDUCTION);
        der = (float *) malloc(num_points_per_scan*sizeof(float)/REDUCTION);
        scan_angle = (float *) malloc(num_points_per_scan*sizeof(float)/REDUCTION);

        // Store Number of Points per Complete Lidar Scan
        scan_size = num_points_per_scan;
}

/*****************************************
 *      Name:   Scan [Destructor]
 *      Description: Destructs the Scan Object
 *****************************************/
Scan::~Scan(void)
{
        // Free Memory
        // Unfiltered Buffers (3)
        free(scan_distance);
        scan_distance = NULL;

        free(scan_distance_n1);
        scan_distance_n1 = NULL;

        free(scan_distance_n2);
        scan_distance_n2 = NULL;

        // Filtered Arrays (5)
        free(scan_x);
        scan_x = NULL;

        free(scan_y);
        scan_y = NULL;

        free(der);
        der = NULL;

        free(scan_angle);
        scan_angle = NULL;

        free(scan_m);
        scan_m = NULL;
}

/*********************************************************************
 *      Name:   Upade Scan Data
 *      Inputs:
 *              - start_index:  index of first scan point in buf
 *              - buf:                  buffer of scan points
 *              - buf_offset:   index of first data point in buf
 *              - length:               length of scan point data to be used in bytes
 *      Output: None
 *      Description:    Copied data from network buffer to N+2 scan array
 *                                      Processes complete scan
 *********************************************************************/
void Scan::UpdateScanData(int start_index, char *buf, int buf_offset, int length){
        // Copy data from buf to N+2 scan array
        memcpy(&scan_distance_n2[start_index], &buf[buf_offset], length);

        if(start_index + length/sizeof(uint32_t) == scan_size){
```

```
                // Completed a scan
                TimeFilter(); // time blur filter

                // Shift Time Buffers
                uint32_t *temp = scan_distance_n1; // hold pointer to allocated buffer (saves from dealloc
and alloc)
                scan_distance_n1 = scan_distance_n2; // move scan data at time n+2 to n+1
                scan_distance_n2 = temp; // this will be overwritten before it is used again

                // Reduce number of points by averaging
                // Fill filtered arrays (angle, x, y, and m)
                for (int i = 0; i < scan_size/REDUCTION; i++){
                        // Error Filtered Average
                        float sum = 0;
                        int points_in_sum = REDUCTION;
                        for(int k = 0; k < REDUCTION; k++){
                                if(scan_distance[REDUCTION*i+k] < MAX_RANGE_MM){
                                        sum += scan_distance[REDUCTION*i+k];
                                }else{
                                        // No point detected (assume error for now)
                                        points_in_sum--;
                                }
                        }

                        if(points_in_sum > 0){
                                scan_m[i] = sum/points_in_sum/MM_PER_METER; // radius in meters
                        }else{
                                // Not an error - no point detected here
                                scan_m[i] = scan_distance[i*REDUCTION]/MM_PER_METER;
                        }

                        // Fill other arrays from filtered magnitudes
                        scan_angle[i] = -180+SCAN_RESOLUTION*REDUCTION*i;
                        scan_x[i] = cos(scan_angle[i]*3.14/180.0)*scan_m[i]; // x coord in meters
                        scan_y[i] = sin(scan_angle[i]*3.14/180.0)*scan_m[i]; // y coord in meters
                }

                // derivative
                for (int i = 1; i < scan_size/REDUCTION; i++){
                        float dx = scan_x[i] - scan_x[i-1];
                        float dy = scan_y[i] - scan_y[i-1];
                        if(dx == 0){
                                dx += 0.00001;
                        }
                        if(dy == 0){
                                dy += 0.00001;
                        }
                        der[i] = dy/dx;
                }
        }
}

/*****************************************************************
 *      Name:   ScanSize
 *      Description:    Returns size of filtered scan arrays
 *****************************************************************/
const int Scan::ScanSize(){
        return scan_size / REDUCTION;
}

/*****************************************************************
 *      Name:   ScanDistance
```

```
 *      Description:    Returns scan filtered scan distances (radius)
 ***********************************************************************/
float *Scan::ScanDistance(){
        return scan_m; // Scan distance in meters
}

/***********************************************************************
 *      Name:   ScanX
 *      Description:    Returns filtered X coordinates
 ***********************************************************************/
float *Scan::ScanX(){
        return scan_x;
}

/***********************************************************************
 *      Name:   ScanY
 *      Description:    Returns filtered Y coordinates
 ***********************************************************************/
float *Scan::ScanY(){
        return scan_y;
}

/***********************************************************************
 *      Name:   ScanAngles
 *      Description:    Returns angles corresponsing to filtered scan
 ***********************************************************************/
float *Scan::ScanAngle(){
        return scan_angle;
}

/***********************************************************************
 *      Name:   Derivative
 *      Description:    Returns dy/xy derivative of filtered scan
 ***********************************************************************/
float *Scan::Derivative(){
        return der;
}

/***********************************************************************
 *      Name:   Time Filter
 *      Description:    Applies time blur filter to buffered scan data
 *                                      Places result in scan_distance
 ***********************************************************************/
void Scan::TimeFilter(){
        for (int i = 0; i < scan_size; i++){
                if(scan_distance_n1[i] < MAX_RANGE_MM && scan_distance_n2[i] < MAX_RANGE_MM){
                        scan_distance[i] = (scan_distance_n1[i] + scan_distance_n2[i])/2;
                }else if(scan_distance_n2[1] < MAX_RANGE_MM){
                        scan_distance[i] = scan_distance_n2[i];
                }else{
                        scan_distance[i] = scan_distance_n1[i];
                }
        }
}
```

Cameras

Up to three Logitech C920 webcams are connected to the PC via USB. The camera data is read and processed using OpenCV.

[ART]

GPS

The Novatel GPS is connected to the PC via USB. The GPS calibration routine must generate latitude and longitude to feet conversions as described in Section 4.7.4.

```
class gps{
     has lat_to_feet
     has long_to_feet

     gps init(){
          initialize communications with gps
          calibrate lat_to_feet, long_to_feet
          call read_gps in new thread
          return self
     }
     void read_gps(){
          loop{
               get data (timestamp)
               add to gps data buffer
          }
     }
}
```

Figure 33 – Preliminary Pseudo Code for GPS Operation

The main part of the differential GPS code is embedded within its serial port class named ‘igvc_GPS’. This C++ class handles the setup, configuration of the unit and fetching of GPS coordinates from the device. The connection to the NovaTel GPS is handled in a similar way to the Arduino via handles. Shown below, Figure 34 shows the method used for retrieving GPS data from the NovaTel device and passing the information into class variables. These class variables will later be taken into consideration within the navigation system when determining the best course for reaching waypoints.

```
void update_lat_long(){
        // This will be the main function for updating the latitude and longitude of this GPS class.
        std::string read_string;
        std::vector<std::string> GPS_items;

        // Tell the GPS to output it's best position data once every 0.1 seconds. Use flag_one to do this once.
        if(this->flag_one == 0){
                this->write_line("LOG BESTPOSA ONTIME .25\r\n");
                this->flag_one = 1;
        }

        // Read a bunch of lines. Not all lines are guaranteed to contain GPS data, so data-checking is required.
        for(int idx = 0; idx < 5; idx++){
                this->read_line();
                read_string = std::string(this->readBuff);
                //std::cout << read_string << std::endl;

                GPS_items = this->split(read_string, ',');
                //std::cout << GPS_items.size() << std::endl;

                // Check if the line that was read actually contains the data we are looking for by checking vector size.
                if(GPS_items.size() > 12){
                        this->igvc_latitude = stod(GPS_items.at(11));
                        this->igvc_longitude = stod(GPS_items.at(12));
                }
                // Clear comms between reads.
                this->purge_comms();
        }
}
```

Figure 34 – Function for Fetching Latitude and Longitude from GPS

The main code for communicating with the GPS module, 'IGVC_GPS.h' is shown below. Its primary function is establishing and facilitating communication between the GPS and host PC.

```
/*
        Name:   IGVC_GPS.h
        Author: Chris Estock
        Brief:  Header file enclosing the class for the GPS serial port.
        Note:   Performs all setup necessary for retrieving the GPS's best latitude and longitude on
demand.

        EXAMPLE IMPLEMENTATION:

        igvc_gps gps;
        gps.COM_PORT = "COM9";
        gps.setup();

        [... somewhere in main loop ...]
        gps.update_lat_long()
        system_latitude = gps.igvc_latitude;
        system_longitude = gps.igvc_longitude;

*/

#include <windows.h>
#include <iostream>
#include <stdlib.h>
#include <time.h>
#include <vector>
#include <algorithm>
#include <iterator>
#include <sstream>

#pragma once
class igvc_gps{
public:
        igvc_gps(void){};
        ~igvc_gps(void){ CloseHandle(this->COM_Handle); };

        std::string COM_PORT;

        HANDLE COM_Handle;
        DCB PortDCB;
        COMMTIMEOUTS CommTimeouts;

        double igvc_latitude;
        double igvc_longitude;
        char readBuff[500];

        bool flag_one;  // Used to signal that the bestposa write command has already been issued.

        void setup(){
                // Handle Setup
                this->handle_setup();
                // DCB Setup
                this->dcb_setup();
                // Timeout Setup
                this->timeout_setup();
                // Clear TX/RX Buffers
                this->purge_comms();
                // Initialize Vars.
```

```
                this->flag_one = 0;
                this->igvc_latitude = 0;
                this->igvc_longitude = 0;
                memset(readBuff, 0, sizeof(readBuff));
        }

        void handle_setup(){
                // Performs a setup of the serial port's handle. Called in setup().
                this->COM_Handle = CreateFile((LPCWSTR)this->COM_PORT.c_str(), GENERIC_READ |
GENERIC_WRITE, (DWORD)NULL, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
                SetupComm(this->COM_Handle, (DWORD)2048, (DWORD)2048);
        }

        void dcb_setup(){
                // Performs a setup of the serial port's DCB parameters. Called in setup().
                this->PortDCB.DCBlength = sizeof(DCB);
                GetCommState(this->COM_Handle, &this->PortDCB);
                this->PortDCB.BaudRate = 115200;                // BAUD
                this->PortDCB.ByteSize = 8;                             // BYTE-LENGTH
                this->PortDCB.Parity = NOPARITY;                // PARITY
                this->PortDCB.StopBits = ONESTOPBIT;            // STOP
                SetCommState(this->COM_Handle, &this->PortDCB);
        }

        void timeout_setup(){
                // Performs a setup of the serial port's timeout parameters. Called in setup();
                // Note: These were determined to work through experimentation with the NovaTel ProPak V3
                GetCommTimeouts(this->COM_Handle, &this->CommTimeouts);
                this->CommTimeouts.ReadIntervalTimeout = 1;                     // Time between incoming
chars.
                this->CommTimeouts.ReadTotalTimeoutConstant = 1000;     // Maximum wait time for
incoming msg.
                this->CommTimeouts.ReadTotalTimeoutMultiplier = 0;
                this->CommTimeouts.WriteTotalTimeoutConstant = 25;
                this->CommTimeouts.WriteTotalTimeoutMultiplier = 0;
                SetCommTimeouts(this->COM_Handle, &this->CommTimeouts);
        }

        void purge_comms(){
                // Clears the TX and RX serial buffers. Called in setup().
                PurgeComm(this->COM_Handle, PURGE_TXCLEAR);
                PurgeComm(this->COM_Handle, PURGE_RXCLEAR);
        }

        void write_line(const char * line_to_write){
                // Attempts to write a line to this class' serial port.
                DWORD numberBytesWritten;
                // Uses numberBytesWritten via reference to perform a pass/fail check.
                WriteFile(this->COM_Handle, LPVOID(line_to_write), strlen(line_to_write),
&numberBytesWritten, NULL);

                // Post-write Status Check
                if(numberBytesWritten == strlen(line_to_write)){
                        std::cout << "Wrote              :       " << line_to_write << "\n";
                        return;
                }else{
                        std::cout << numberBytesWritten << " FAILED to Write ALL Bits!\n";
                        return;
                }
        }

        void read_line(){
```

```
                // Attempts to read a line from this class' serial port.
                DWORD dwReadBytes;
                if(!ReadFile(this->COM_Handle, LPVOID(this->readBuff), (DWORD)(499), &dwReadBytes, NULL)){
                        std::cout << "Failed to read!" << std::endl;
                }else{
                        //std::cout << "Read(" << dwReadBytes << ")              :       " << this-
>readBuff << std::endl;
                }
        }

        template<typename Out>
        void split(const std::string &s, char delim, Out result) {
                // Main string parsing function (with delimiter)
                // Note: This is a bit of a hack of the Python split() function.
                // Credit: Answer in http://stackoverflow.com/questions/236129/split-a-string-in-c
                std::stringstream ss;
                ss.str(s);
                std::string item;
                while (std::getline(ss, item, delim)) {
                    *(result++) = item;
                }
        }

        std::vector<std::string> split(const std::string &s, char delim) {
                // Helper function for string split with delimiters
                // Note: This is a bit of a hack of the Python split() function.
                // Credit: Answer in http://stackoverflow.com/questions/236129/split-a-string-in-c
                std::vector<std::string> elems;
                split(s, delim, std::back_inserter(elems));
                return elems;
        }

        void update_lat_long(){
                // This will be the main function for updating the latitude and longitude of this GPS
class.
                std::string read_string;
                std::vector<std::string> GPS_items;

                // Tell the GPS to output it's best position data once every 0.1 seconds. Use flag_one to
do this once.
                if(this->flag_one == 0){
                        this->write_line("LOG BESTPOSA ONTIME .25\r\n");
                        this->flag_one = 1;
                }

                // Read a bunch of lines. Not all lines are guaranteed to contain GPS data, so data-
checking is required.
                for(int idx = 0; idx < 5; idx++){
                        this->read_line();
                        read_string = std::string(this->readBuff);
                        //std::cout << read_string << std::endl;

                        GPS_items = this->split(read_string, ',');
                        //std::cout << GPS_items.size() << std::endl;

                        // Check if the line that was read actually contains the data we are looking for
by checking vector size.
                        if(GPS_items.size() > 12){
                                this->igvc_latitude = stod(GPS_items.at(11));
                                this->igvc_longitude = stod(GPS_items.at(12));
                        }
                        // Clear comms between reads.
```

```
                        this->purge_comms();
                }
        }
};
```

[CRE]

Compass

The compass is connected to the Arduino Mega via I2C. The data is acquired and used in real-time by the control loop in the Arduino, but it must also be streamed to the PC to be used by the mapping routine. The PC will collect the data and store it in a time stamped buffer. The communication with the compass is handled entirely within the Arduino Mega. The Arduino then forward the compass heading to the host PC over the serial port.

```
class compass{

     gps init(){
          connect to Arduino
          call read_data in new thread
          return self
     }
     void read_data(){
          loop{
               get data from stream (time stamp)
               add to compass data buffer
          }
     }
}
```

Figure 35 – Preliminary Pseudo Code for Digital Compass

Arduino Software

The Arduino runs the control loop and acquires data from the compass, PC (target angle and speed), estop, and tachometers. The Arduino interfaces with the motor driver via UART and controls the safety light. The Adafruit HMC5228L fetches the x, y, and z magnetic field readings in micro-teslas and sends this information to the Arduino. The header value is calculated by taking the arctangent of the x and y coordinates and applying a buffer for magnetic field declination. This information is translated to show the vehicles facing direction in real time. 0° is when the vehicle is facing true north and sweeps clockwise through 359° i.e. facing west would be 270°.

The main Arduino C++ (host PC) code file ‘IGVC_Arduino.h’ is shown below. It’s primary function is to establish and facilitate communication between the Arduino Mega and the host PC.

```
/*
        Name:   IGVC_Arduino.h
        Author: Chris Estock
        Brief:  Header file enclosing the class for the Arduino serial port.
        Note:   Performs all setup necessary for retrieving a message from the Arduino.

        EXAMPLE IMPLEMENTATION:

        igvc_arduino arduino;
        arduino.COM_PORT = "COM5";
        arduino.setup();

        [ ... Somewhere in Main Loop ... ]
        arduino.get_compass_and_errorcode(); // Update the arduino object's compass_heading and errorcode
variables.
        // dostuff with arduino.compassHeading

        // derive a speed, angle and distance to travel... ( arduino.speed_toArduino;
arduino.angle_toArduino; arduino.distance_toArduino; )
        arduino.send_speed_angle_distance(); // Send the arduino object's speed_toArduino,
angle_toArduino, distance_toArduino to the Arduino over serial.

*/

#include <windows.h>
#include <iostream>
#include <stdlib.h>
#include <time.h>
#include <vector>
#include <algorithm>
#include <iterator>
#include <sstream>

using namespace std;

#pragma once
class igvc_arduino{
public:
        igvc_arduino(void){};
        ~igvc_arduino(void){ CloseHandle(this->COM_Handle); };

        std::string COM_PORT;

        HANDLE COM_Handle;
        DCB PortDCB;
        COMMTIMEOUTS CommTimeouts;

        char readBuff[500];

        // INCOMING PARAMETERS
        float compass_heading;
        bool errorcode;

        // OUTGOING PARAMETERS
        int speed_toArduino;
        int angle_toArduino;
        float distance_toArduino;
```

```
        bool flag_one;  // Used to signal that the bestposa write command has already been issued.

        void setup(){
                // Handle Setup
                this->handle_setup();
                // DCB Setup
                this->dcb_setup();
                // Timeout Setup
                this->timeout_setup();
                // Clear TX/RX Buffers
                this->purge_comms();
                // Initialize Vars.
                this->flag_one = 0;
                memset(readBuff, 0, sizeof(readBuff));
                this->compass_heading = 0.0;
                this->errorcode = 0;
        }

        void handle_setup(){
                // Performs a setup of the serial port's handle. Called in setup().
                this->COM_Handle = CreateFile(this->COM_PORT.c_str(), GENERIC_READ | GENERIC_WRITE,
(DWORD)NULL, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
                //this->COM_Handle = CreateFile(TEXT("COM4"), GENERIC_READ | GENERIC_WRITE, (DWORD)NULL,
NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
                if(!SetupComm(this->COM_Handle, (DWORD)2048, (DWORD)2048)){
                        cout << "Failed in Arduino SetupComm" << endl;
                }
        }

        void dcb_setup(){
                // Performs a setup of the serial port's DCB parameters. Called in setup().
                this->PortDCB.DCBlength = sizeof(DCB);
                GetCommState(this->COM_Handle, &this->PortDCB);
                this->PortDCB.BaudRate = 9600;                  // BAUD
                this->PortDCB.ByteSize = 8;                             // BYTE-LENGTH
                this->PortDCB.Parity = NOPARITY;                // PARITY
                this->PortDCB.StopBits = ONESTOPBIT;            // STOP
                if(!SetCommState(this->COM_Handle, &this->PortDCB)){
                        cout << "Failed in Arduino SetCommState" << endl;
                }
        }

        void timeout_setup(){
                // Performs a setup of the serial port's timeout parameters. Called in setup();
                // Note: These were determined to work through experimentation with the NovaTel ProPak V3
                GetCommTimeouts(this->COM_Handle, &this->CommTimeouts);
                this->CommTimeouts.ReadIntervalTimeout = 10;                    // Time between incoming
chars.
                this->CommTimeouts.ReadTotalTimeoutConstant = 200;      // Maximum wait time for
incoming msg.
                this->CommTimeouts.ReadTotalTimeoutMultiplier = 1;
                this->CommTimeouts.WriteTotalTimeoutConstant = 25;
                this->CommTimeouts.WriteTotalTimeoutMultiplier = 0;
                if(!SetCommTimeouts(this->COM_Handle, &this->CommTimeouts)){
                        cout << "Failed in Arduino SetCommTimeouts" << endl;
                }
        }

        void purge_comms(){
                // Clears the TX and RX serial buffers. Called in setup().
                PurgeComm(this->COM_Handle, PURGE_TXCLEAR);
                PurgeComm(this->COM_Handle, PURGE_RXCLEAR);
```

```
        }

        void write_line(const char * line_to_write){
                // Attempts to write a line to this class' serial port.
                DWORD numberBytesWritten;
                // Uses numberBytesWritten via reference to perform a pass/fail check.
                if(!WriteFile(this->COM_Handle, LPVOID(line_to_write), strlen(line_to_write),
&numberBytesWritten, NULL)){
                        cout << "Failed to write to Arduino!" << endl;
                }

                // Post-write Status Check
                if(numberBytesWritten == strlen(line_to_write)){
                        std::cout << "Wrote              :       " << line_to_write << "\n";
                        return;
                }else{
                        std::cout << numberBytesWritten << " FAILED to Write ALL Bits!\n";
                        return;
                }
        }

        void read_line(){
                // Attempts to read a line from this class' serial port.
                DWORD dwReadBytes;
                // Clear the read buffer first.
                memset(readBuff, 0, sizeof(readBuff));
                if(!ReadFile(this->COM_Handle, LPVOID(this->readBuff), (DWORD)(499), &dwReadBytes, NULL)){
                        std::cout << "Failed to read!" << std::endl;
                }else{
                        //std::cout << "Read(" << dwReadBytes << ")              :       " << this-
>readBuff << std::endl;
                }
        }

        template<typename Out>
        void split(const std::string &s, char delim, Out result) {
                // Main string parsing function (with delimiter)
                // Note: This is a bit of a hack of the Python split() function.
                // Credit: Answer in http://stackoverflow.com/questions/236129/split-a-string-in-c
                std::stringstream ss;
                ss.str(s);
                std::string item;
                while (std::getline(ss, item, delim)) {
                    *(result++) = item;
                }
        }

        std::vector<std::string> split(const std::string &s, char delim) {
                // Helper function for string split with delimiters
                // Note: This is a bit of a hack of the Python split() function.
                // Credit: Answer in http://stackoverflow.com/questions/236129/split-a-string-in-c
                std::vector<std::string> elems;
                split(s, delim, std::back_inserter(elems));
                return elems;
        }

        void send_speed_angle_distance(){
                // This function will send this object's 'speed_toArduino' 'angle_toArduino' and
distance_toArduino'
                // over the serial port to the Arduino in a packet formed like:
                //              "speed_toArduino,angle_toArduino,distance_toArduino\n"
```

```
                // Variable creation, convert the floating point values to strings before forming the
packet.
                std::string outstring = "";
                std::string speed_as_string = std::to_string(speed_toArduino);
                std::string angle_as_string = std::to_string(angle_toArduino);
                //std::string distance_as_string = std::to_string(distance_toArduino);

                outstring.append("S");
                outstring.append(speed_as_string);
                outstring.append("A");
                outstring.append(angle_as_string);
                //outstring.append(",");
                //outstring.append(distance_as_string);
                outstring.append("\n");

                //std::cout << outstring.c_str() << std::endl;
                this->write_line(outstring.c_str());
        }

        void get_compass_and_errorcode(){
                // This will be the main function for updating the latitude and longitude of this GPS
class.
                std::string read_string;
                std::vector<std::string> Arduino_items;

                // Read a bunch of lines. Not all lines are guaranteed to be valid, so data-checking is
required.
                for(int idx = 0; idx < 3; idx++){
                        this->read_line();
                        read_string = std::string(this->readBuff);
                        //std::cout << read_string << std::endl;

                        Arduino_items = this->split(read_string, ',');
                        //std::cout << Arduino_items.size() << std::endl;

                        // Check if the line that was read actually contains the data we are looking for
by checking vector size.
                        //if(Arduino_items.size() != 0){
                                //this->compass_heading = stod(Arduino_items.at(0));
                                //if(Arduino_items.at(1) == "0"){
                                        //this->errorcode = 0;
                                //}else{
                                        //this->errorcode = 1;
                                //}
                        //}
                        // Clear comms between reads.
                        this->purge_comms();
                }
        }
};
```

[CRE]

Shown below is C code that is used with the Arduino Mega micro controller to handle the motor control and compass communications.

```
/*
  IGVC Master Code
  Rev 2
  Chris Estock, Garrett Chonko, & John Johenning
  4/22/17

  All refrences to vehicle are taken with repspect to forward facing: M1 =pass side, M2 =driver side
  Uses speed and angle to adjust motor speeds
*/

#include <Wire.h>
#include <SoftwareSerial.h>

#include "RoboClaw.h"
#include "Adafruit_Sensor.h"
#include "Adafruit_HMC5883_U.h"

//Compass Constants
#define DECLINATION_ANGLE -0.146025 // Akron U = -0.146025 radians, Oakland U = -0.130609
#define INTERVAL 500 //value set to send compass data  every 1/2 sec

//ARDUINO Connections
#define PASS_ENCODE_PIN 10  //25
#define DRIVER_ENCODE_PIN 12 //24

//Motor Driver Constants
#define motorDriverAddress 0x80
#define ROBOCLAW_GAIN ((63*3.52)/1675) // Max channel value over max motor driver gain

//Vehicle Constants
#define WIDTH 0.786 //Half of the wheel base
#define GR_TRANS 103 // motor gear to drive gear ratio
#define PPR 400 //Pulses per Revolution of the encoders

//Transfer Function Constants
#define CURRENT_INPUT_GAIN 0.018
#define PREVIOUS_INPUT_GAIN 0.014


SoftwareSerial RBCC(8,9);
RoboClaw roboclaw(&RBCC,10000);

Adafruit_HMC5883_Unified mag = Adafruit_HMC5883_Unified(12345); // Unique ID for the Compass

unsigned long currentTime = 0; // records runtime of the processor, used for determining data sending
frequency
unsigned long previousTime = 0;
unsigned long accumulatedTime = 0;

// Floats for Motor Controller
float InputR =  0;
float InputL =  0;
float OutputR =  0;
float OutputL =  0;
float PrevOutR = 0;
float PrevOutL = 0;

// bools for Detecting negative and positive Wheel Rotations
```

```
bool isRNegative =   false;
bool isLNegative =   false;
bool isRChangeDirec = false;
bool isLChangeDirec = false;
bool FirstCrossOverR;
bool FirstCrossOverL;

// Indicators if there is a problem
bool errorFlag = 0;

//Values Recieved from Computer
byte DesiredDistance = 0;
byte DesiredAngle = 0;
byte DesiredSpeed = 0;

// Ints for Tach read
volatile unsigned long TachRPulseCount = 0;
volatile unsigned long TachLPulseCount = 0;
int TachLPulseChange =   0;
int TachRPulseChange =   0;
int TachLPulse =  0;
int TachRPulse =  0;


// Doubles for Tach controlled position feedback
double OverallDistance = 0;
double OverallAngle =0;
double PrevDistance=0;

// Input variables
int requestedDistance = 0;
int requestedAngle = 0;
int requestedSpeed = 0;
int MotorChange = 0;

// Floats for ZeroCross Detection
float PrevRSpeed;
float RSpeedDiff;
float PrevLSpeed;
float LSpeedDiff;

// Ints for ZeroCross Detection
int RHasChanged;
int LHasChanged;

// Float for Speed Calculation
float GlobalSpeed[2];

// Doubles for Wheel Speed FeedBack
double GlobalRightWheelSpeed=0;
double GlobalLeftWheelSpeed=0;

// === PROTOTYPES ===
void MovementController(int Distance, int Angle, int Speed);
void FindZeroCrossover(float* Speeds,int SpeedType);
void MotorController(float RightWheelSpeed, float LeftWheelSpeed);
void TachCount();
void SpeedCalc();
void LocationCalc(float* Speeds,int timeElapsed);
void left_tach_isr();
void right_tach_isr();
```

```
#define LEFT_TACH_PIN 2
#define RIGHT_TACH_PIN 3
//volatile unsigned long left_tach_counter = 0;
//volatile unsigned long right_tach_counter = 0;


/*****************************************************************************************************
*****************************************************/
void setup(){
  Serial.begin(9600);
  roboclaw.begin(38400);
  Serial.println("IGVC Master Code Rev1\n\n");

  // Attach interrupts to pins 2 and 3 for the Right and Left Tachometers.
  // These will call interrupt routines for increment a counter which will determine speed via comparison
to time.
  attachInterrupt(digitalPinToInterrupt(LEFT_TACH_PIN), left_tach_isr, CHANGE);   // Pin2
  attachInterrupt(digitalPinToInterrupt(RIGHT_TACH_PIN), right_tach_isr, CHANGE); // Pin3

  pinMode(5, OUTPUT);
  pinMode(6, OUTPUT);

}
/*****************************************************************************************************
*****************************************************/
void loop(void){

  // This snippet tracks the current time, previous time, and accumulated time which will trigger the
sampling portion of this loop every X seconds.
  previousTime = currentTime;
  currentTime=millis(); //counts milliseconds of run time
  accumulatedTime += (currentTime - previousTime);


  requestedAngle = analogRead(5)*(0.3516);
  requestedDistance = analogRead(4)/100;
 // double Speed=analogRead(8);

  /*
  Serial.println("requestedDistance");
  Serial.println(requestedDistance);
  Serial.println("requestedAngle");
  Serial.println(requestedAngle);
  */
   /*
  if(Serial.available()){
    char valueType = Serial.read();
    if((valueType == 'D') || (valueType == 'd')){
      requestedDistance = Serial.read(); //read byte in buffer, Distance in Meters
    }
    valueType = Serial.read(); //read char in buffer
    if((valueType == 'A') || (valueType == 'a')){
      requestedAngle = Serial.read();// angle in degrees
    }
    valueType = Serial.read(); //read char in buffer
    if((valueType == 'S') || (valueType == 's')){
      requestedSpeed = Serial.read();// Speed in degrees mph
    }
  }*/

  if(accumulatedTime >= 100){
    SpeedCalc();
```

```
    if(isRChangeDirec == true){
      MotorChange = MotorChange + 1;
    }
    if(isLChangeDirec == true){
      MotorChange = MotorChange + 2;
    }
    FindZeroCrossover(GlobalSpeed,MotorChange);

    MotorChange = 0;

    LocationCalc(GlobalSpeed,accumulatedTime);

    // Reset the tachometer counters to zero in order to prepare to count pulse edges for the next cycle.
    TachRPulseCount = 0;
    TachLPulseCount = 0;
    accumulatedTime = 0;

    if(((DesiredAngle-OverallAngle)!=requestedAngle) || ((DesiredDistance-
OverallDistance)!=requestedDistance)){// input Different from Current Feedback Model
      DesiredAngle = requestedAngle;
      DesiredDistance = requestedDistance;
      OverallAngle = 0;
      OverallDistance = 0;
    }

    int  InputAngle = DesiredAngle - OverallAngle;
    int  InputDistance= DesiredDistance - OverallDistance;

    MovementController(InputDistance, InputAngle, DesiredSpeed);

    float  InputRSpeed = GlobalRightWheelSpeed - GlobalSpeed[0];
    float  InputLSpeed = GlobalLeftWheelSpeed - GlobalSpeed[1];
    MotorController(InputRSpeed, InputLSpeed);
  }


  //Send Read values to Laptop
  //wherePointing();  //0-3600 tenths of a degree
}

// === LEFT TACHOMETER INTERRUPT SERVICE ROUTINE ===
void left_tach_isr(){
  TachLPulseCount += 1;
  return;
}
// === RIGHT TACHOMETER INTERRUPT SERVICE ROUTINE ===
void right_tach_isr(){
  TachRPulseCount += 1;
  return;
}

/******TACH
COUNT*************************************************************************************************
*********************************/
void TachCount(){

//Read Tachs
  TachRPulse = digitalRead(PASS_ENCODE_PIN);
  TachLPulse = digitalRead(DRIVER_ENCODE_PIN);

  if (TachLPulse != TachLPulseChange){ // Count pulse Changes in Left Wheel Tach
```

```
    TachLPulseChange = TachLPulse;
    TachLPulseCount++;
  }
  if (TachRPulse != TachRPulseChange){ // Count pulse Changes in Right Wheel Tach
    TachRPulseChange = TachRPulse;
    TachRPulseCount++;
  }
}

/****** SPEED CALC
*****************************************************************************************************
*************************/
void SpeedCalc(){
  double MPHL, MPHR;
  double RPML = TachLPulseCount*1.5*0.3;
  double RPMR = TachRPulseCount*1.5*0.3;

  //Convert RPM to MPH
  MPHR=RPMR*(0.0279);//10 inch Diameter wheels , rpm to mph using feet to miles
  MPHL=RPML*(0.0279);

  //Convert to Radians Per Second
  GlobalSpeed[0]=MPHR;
  GlobalSpeed[1]=MPHL;

  return 0;
}

/********LOCATION
CALC*****************************************************************************************************
**************************/
void LocationCalc(float* Speeds,int timeElapsed){  // FeedBack
  double CurrentAngle, CurrentDistance;
  double FeedBack[2];
  CurrentDistance=0;
  CurrentAngle = 0;
  float  DeltaSpeed = Speeds[0] - Speeds[1];
  float  TotalSpeed = (Speeds[0] + Speeds[1])/2;
  if(DeltaSpeed<=0.05 && DeltaSpeed>=-0.05){
    DeltaSpeed=0;
    CurrentDistance = TotalSpeed * (0.1)*(0.4469);
    CurrentAngle = 0;
    OverallAngle = (CurrentAngle + OverallAngle);
  }else{
    CurrentAngle = DeltaSpeed * (0.1)*(0.4469)* (180/(PI * WIDTH ));
    OverallAngle = (CurrentAngle + OverallAngle);
    CurrentDistance = sin((OverallAngle*PI)/360) * 2 *(TotalSpeed / DeltaSpeed) * WIDTH;
  }
  OverallDistance = (CurrentDistance-PrevDistance) + OverallDistance;
  PrevDistance=CurrentDistance;


  /*
  Serial.print(OverallDistance);
  Serial.print(" ");
  Serial.println(TotalSpeed);
  */


  return 0;
}
```

```
/******** MOVEMENT CONTROLLER
**************************************************************************************************
***************/
void MovementController(int Distance, int Angle, int Speed){
  float LeftWheelSpeed =  0;
  float RightWheelSpeed = 0;

  Speed=1;
  float WheelSpeeds[2];
  float RotationRadius;
  float ThetaMax;
  float VehicleForwardSpeed;

  if(Angle==0){
    float RotationRadius=0;
    float ThetaMax =(0);
    float VehicleForwardSpeed = Speed;
  }else{
    float RotationRadius = (Distance/sin(Angle/2)*2);
    float ThetaMax = (Speed*180/(RotationRadius*PI));
    float VehicleForwardSpeed = RotationRadius*PI*Angle/180;
  }

  float WheelSpeedDifference = ThetaMax*PI*WIDTH/180;
  float WheelSpeedSpin = Angle*PI*WIDTH/180;
  /*
  Serial.println("ThetaMax");
  Serial.println(ThetaMax);
  Serial.println("WheelSpeedDifference");
  Serial.println(WheelSpeedDifference);
  */
  if(RotationRadius == 0){
    if(ThetaMax == 0){
      if(WheelSpeedDifference < WIDTH){
        RightWheelSpeed = -1*Speed;
        LeftWheelSpeed =  -1*Speed;
      }
      if(WheelSpeedDifference > WIDTH){
        RightWheelSpeed = +1*Speed;
        LeftWheelSpeed =  +1*Speed;
      }
    }else{
      RightWheelSpeed = +1*Speed;
      LeftWheelSpeed =  -1*Speed;
    }
  }else{
    if(RotationRadius < (-WIDTH)){
      RightWheelSpeed = VehicleForwardSpeed;
      LeftWheelSpeed  = VehicleForwardSpeed + WheelSpeedDifference;
    }
    if(RotationRadius == (-WIDTH)){
      RightWheelSpeed = VehicleForwardSpeed;
      LeftWheelSpeed  = 0;
    }
    if((RotationRadius > (-WIDTH)) && (RotationRadius < WIDTH)){
      RightWheelSpeed = (VehicleForwardSpeed - WheelSpeedDifference)/2;
      LeftWheelSpeed  = (VehicleForwardSpeed + WheelSpeedDifference)/2;
    }
    if(RotationRadius == WIDTH){
      RightWheelSpeed = 0;
      LeftWheelSpeed  = VehicleForwardSpeed;
    }
```

```
    if(RotationRadius > WIDTH){
      RightWheelSpeed = VehicleForwardSpeed - WheelSpeedDifference;
      LeftWheelSpeed  = VehicleForwardSpeed;
    }
  }

  // Detect direction negative change
  if(((RightWheelSpeed < 0) && (isRNegative = false) )||(( RightWheelSpeed >= 0) && (isRNegative=true))){
    isRChangeDirec=true;
  }
  if(LeftWheelSpeed < 0 && isLNegative == false || LeftWheelSpeed >= 0 && isLNegative == true ){
    isLChangeDirec=true;
  }

  //  Adjust Speed for each motor
  //  adjustedPassMotor = speedByte*(+1*adjustPercentage);
  //  adjustedDriverMotor = speedByte*(-1*adjustPercentage); //adjust for opposite side motor


  GlobalRightWheelSpeed=RightWheelSpeed;
  GlobalLeftWheelSpeed=LeftWheelSpeed;

  /*
  Serial.println(" RightWheelSpeed");
  Serial.println(RightWheelSpeed);
  Serial.println("LeftWheelSpeed");
  Serial.println(LeftWheelSpeed);
  */

  return 0;
}

/*********FIND ZERO CROSSOVER
*************************************************************************************************************
***************/
void FindZeroCrossover(float* Speeds,int SpeedType){

  // Right Wheel Crossover
  if((SpeedType == 1) || (SpeedType >= 3)){//1 = right turn, 3 = both turn
    if(FirstCrossOverR==true){

      float PrevRSpeed = 0;
      float RSpeedDiff = 0;
      int RHasChanged  = 0;

      FirstCrossOverR = false;
    }

     RSpeedDiff=Speeds[0]-PrevRSpeed;

     PrevRSpeed=Speeds[0];

     RHasChanged = ((RSpeedDiff>=0) ? RHasChanged++ : RHasChanged--);


    if(RHasChanged >= 3){
      isRChangeDirec=false;
      FirstCrossOverR=true;
      //change Direction
      isRNegative = ((isRNegative == true) ? false : true);
    }
  }
```

```
  // Left Wheel Crossover
  if(SpeedType==2 || SpeedType>=3){//2 = left turn, 3 = both turn
     if(FirstCrossOverL==true){

      float PrevLSpeed = 0;
      float LSpeedDiff = 0;
      int LHasChanged  = 0;

      FirstCrossOverL = false;
    }

     LSpeedDiff=Speeds[1]-PrevLSpeed;
     PrevLSpeed=Speeds[1];

     LHasChanged = ((LSpeedDiff>=0) ? LHasChanged++ : LHasChanged--);


    if(LHasChanged >= 3){
      isLChangeDirec=false;
      FirstCrossOverL=true;
      //change Direction
      isLNegative = ((isLNegative == true) ? false : true);
    }
  }


  return 0;
}

/**************MOTOR
CONTROLLER*************************************************************************************************
************************/
void MotorController(float RightWheelSpeed, float LeftWheelSpeed){
  byte passengerMotorSpeed, driverMotorSpeed;
  int adjustedPassMotor, adjustedDriverMotor;

  /*Serial.println("RightWheelSpeed");
  Serial.println(RightWheelSpeed);
  Serial.println("LeftWheelSpeed");
  Serial.println(LeftWheelSpeed);
  */

  InputR = (RightWheelSpeed * 0.0625)+64;
  InputL = (LeftWheelSpeed  *0.0625)+64;

  /*
  Serial.println("InputL");
  Serial.println(InputL);
  Serial.println("InputR");
  Serial.println(InputR);
  */

  //Transfer Function for motor
  //OutputR = CURRENT_INPUT_GAIN * InputR - PREVIOUS_INPUT_GAIN * PrevR + PrevOutR;
  //OutputL = CURRENT_INPUT_GAIN * InputL - PREVIOUS_INPUT_GAIN * PrevL + PrevOutL;

  OutputR = InputR+PrevOutR;
  OutputL = InputL+PrevOutL;

  //PrevR = InputR;
```

```
  //PrevL = InputL;

  PrevOutL = OutputL;
  PrevOutR = OutputR;

  /*
  Serial.println("OutputR");
  Serial.println(OutputR);
  Serial.println("OutputL");
  Serial.println(OutputL);
  */

  //Keep speed between 0 and 127
  adjustedPassMotor = constrain(InputR, 0, 127);
  adjustedDriverMotor = constrain(InputL, 0, 127);

  //Convert to bytes to send to motor driver
  passengerMotorSpeed = byte(adjustedPassMotor);
  driverMotorSpeed = byte(adjustedDriverMotor);

  /*
  Serial.println("driverMotorSpeed");
  Serial.println(driverMotorSpeed);
  Serial.println("passengerMotorSpeed");
  Serial.println(passengerMotorSpeed);
  */

  //Send values to motordriver: M1=pass motor, M2=driver motor
  roboclaw.ForwardBackwardM1(motorDriverAddress,passengerMotorSpeed);
  roboclaw.ForwardBackwardM2(motorDriverAddress,driverMotorSpeed);
}

/******Digital
Compass*************************************************************************************************
**********************************/
void wherePointing(){

  //INSERT COMPASS CODE
  //Read Sensor
  sensors_event_t event;
  mag.getEvent(&event);

  // Hold the module so that Z is pointing 'up' and you can measure the heading with x&y
  // Calculate heading when the magnetometer is level, then correct for signs of axis.
  float heading = atan2(event.magnetic.y, event.magnetic.x);

  //Adjust Heading by the 'Declination Angle' (error in magnetic field based off location)
  heading += DECLINATION_ANGLE;

  //Keep heading between 0 and 2*PI
  if(heading < 0)
    heading += 2*PI;
  if(heading > 2*PI)
    heading -= 2*PI;

  // Convert radians to degrees for readability.
  float headingDegrees = heading * 180/M_PI;

  // This snippet sends the compass data and error flags to the host computer.
  String outstring = "";
  outstring += headingDegrees;
  outstring += ',';
```

```
  outstring += errorFlag;
  Serial.println(outstring);

  //Reset Values
  errorFlag=0;
}
```

[GWC, JPJ, CRE]

```
Init{
     Initialize GPIO pins for estop trigger and sense
     Initialize GPIO pins for safety light safety lights
     T = sampling time for control system
}

Loop{
     While not T elapsed{
          if off{
               safety light off
          }else if(autonomous){
               safety light flashing
          }else{
               safety light on
          }
     }
     acquire tachometer speeds
     acquire compass heading
     acquire target speed and angle
     evaluate control compensator function
     output signal for each motor (UART)
     send compass data to PC
}


Function{
     get magnetic field information from compass
     evaluate header
```

Figure 36 – Preliminary Pseudo Code for Arduino Mega Micro Controller

3.7.3 *Image Processing*

Image Compression

Most HD USB webcams operate at with 10-bit color depth. This means that each pixel has 10 bits representing the red subpixel, 10 bits representing the green subpixel, and 10 bits

representing the blue subpixel. In total that means each pixel in an image is represented by 30 bits of color data. For 1080p pictures the pixel dimensions are 1920x1080 at a 16:9 aspect ratio. A 1080p picture will contain 2,073,600 pixels. At 10-bit color depth that is 62,208,000 bits or 7,776,000 bytes or 7.42 Mb. With three cameras, the software will collect 22.26 Mb of raw camera data 24 times per second. Using the processing methods called out in the design requires three filtered copies to be made for each camera. Using these images at full resolution requires 200.32 Mb of additional picture data to be processed and stored in memory every 0.041 seconds. Full 1080p resolution is not required to detect 3-inch wide lines at distances on the order of 10 feet used on the vehicle.

A 240p picture has enough resolution to determine 3-inch wide lines at the distance the vehicle's cameras see. The pixel dimensions at 240p and 16:9 resolution are 426x240. This image has 102,240 pixels per image or 3,067,200 bits per image at 10-bit per channel color depth. This image only requires 375 Kb of space in memory. Three filtered copies per camera would only require processing 3.3 Mb of image data per 0.041 seconds.

[ART]

Image Filtering

Image filtering is used to enhance pictures captured by the camera to make it easier to distinguish between objects of interest and unimportant data. The intelligent vehicle must be able to detect white lines on a dark background, blue flags, and red flags.

Filtering for White Lines

Camera images are recorded in RGB color space where the white is achieved when the red, green, and blue channels are at 100%. Figure 37 depicts RGB color space a cube where white sits on a corner. If white was always captured as perfect 100% red, 100% green, 100% blue, it would be easy to distinguish white objects in an image. Unfortunately, exposure, lighting conditions, and color casting all contribute to white not always being captured as true white on the camera sensor. Trying to identify off-white as white in RGB color space is difficult because white is a combination of three color channels and lives on a diagonal across three dimensions. HSV color space models color as a cylinder with white and near white colors in the center. Identifying white in HSV color space is computationally easier because white and white like colors sit at the low end of the saturation spectrum and the high end of the value spectrum. The hue channel can be completely ignored when detecting white.

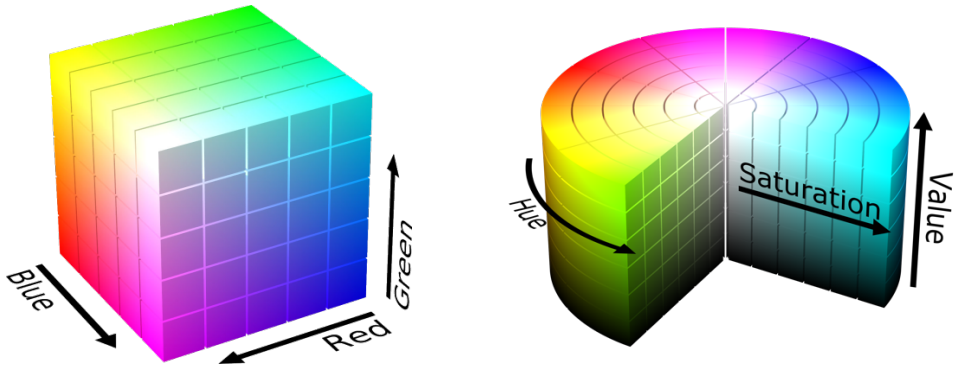


Figure 37. RGB and HSV Color Space Models [9].

The filter process for isolating white lines consists of three steps. First, the RGB image is converted to HSV color format. Only the saturation and value channels are used. The value channel is given by

$$V = \max(R, G, B) \tag{30}$$

where *R*, *G*, and *B* are the red, green, and blue channel values respectively. The saturation is calculated using

$$S_{HSV} = \begin{cases} 0, & \text{if } V = 0 \\ \frac{C}{V}, & \text{otherwise} \end{cases} \tag{31}$$

where *C* is the chroma component.

Chroma is defined in Equation (32).

$$C = \max(R, G, B) - \min(R, G, B) \tag{32}$$

Then, the average saturation, minimum saturation, and average value for the image are calculated. Finally, each pixel that has a saturation value less than the mean of the average and minimum saturation and a value greater than the average value is set to white. All pixels that do not meet this criterion are set to black. This process is demonstrated in Figure 38.

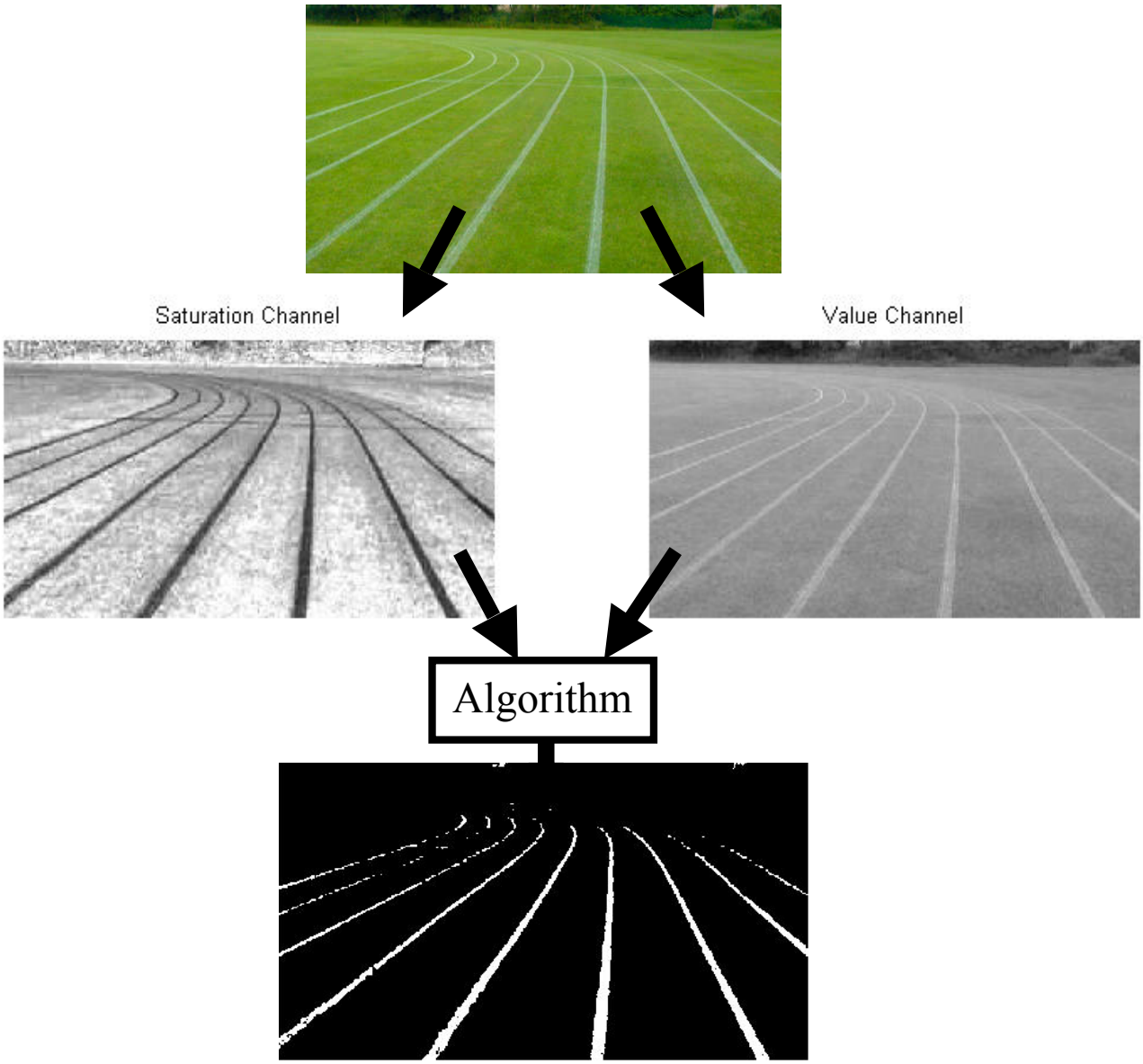


Figure 38. Progression of Image Filtering [10].

Original Image (top), Lightness Channel (middle), Adjusted Black/White Points (bottom)

```
mask whiteFilter(RGBImage* RGB){
     mask = array(RGB.width, RGB.height);
     HSVImage HSV = rgb2hsv(RGB);
     // assume H is in degrees
     // S and V are in percent
     value = HSV.valueChannel();
     saturation = HSV.saturationChannel();
     avg_sat = sum(pixels in saturation)/pixels.count();
     avg_val = sum(pixels in value)/pixels.count();
     min_sat = min(pixels in saturation);

     mask.inRange(HSV, (0-360,0-((avg_sat+min_sat)/2),avg_val-
          100); //generate mask everywhere a pixel falls
          within (H1-H2,S1-S2,V1-V2) ranges
     return mask;
```

Figure 39 – Preliminary Pseudo Code for White Color Filtering

Filtering for Red and Blue Flags

Like white, red and blue exist on corners of the RGB color cube. Due to the geometries of the cube, it is difficult to mathematically describe all the shades of red or blue as RGB values. The resulting function is complex. It is much easier to transform the image to HSV color space. In HSV space, red and blue are each expressed as a range of hues at the upper end of the saturation and lightness spectrums. Hue is defined as

$$H = 60^\circ \times \begin{cases} \text{undefined}, & \text{if } C = 0 \\ \frac{G-B}{C} \mod 6, & \text{if} \max(R,G,B) = R \\ \frac{B-R}{C} + 2, & \text{if} \max(R,G,B) = G \\ \frac{R-G}{C} + 4, & \text{if} \max(R,G,B) = B \end{cases} \tag{33}$$

Red is defined as HSV ranges (0-20°, 40-100%, 50-100%) and (340-360°, 40-100%, 50-100%). Blue is defined as HSV ranges (220-260°, 40-100%, 50-100%).

```
mask redFilter(RGBImage* RGB){
     mask1 = array(RGB.width, RGB.height);
     mask2 = array(RGB.width, RGB.height);
     // assume H is in degrees
     // S and V are in percent
     HSVImage HSV = rgb2hsv(RGB);

     mask1.inRange(HSV, (0-20,40-100,50-100);
     mask2.inRange(HSV, (340-360,40-100,50-100);

     return mask1||mask2;
}

mask blueFilter(RGBImage* RGB){
     mask = array(RGB.width, RGB.height);
     // assume H is in degrees
     // S and V are in percent
     HSVImage HSV = rgb2hsv(RGB);

     mask.inRange(HSV, (220-260,40-100,50-100);
     return mask;
}
```

Figure 40 – Preliminary Pseudo Code for Red and Blue Color Filtering

From the white, red, and blue filters, points are extrapolated from areas where high densities of white, red, or blue detected. These points are used when mapping to real-space.

After further experimentation with the method of filtering white in the image, a new method was devised and implemented using RGB color space. The pseudo-code for the new method is shown in Figure 43.

A line tracing algorithm is used to detect and trace the white lines in the image. The classification of white points into line objects helps filter out some of the other noise in the image. The lines that are used in the later mapping can be guaranteed to have a certain number of points in them. The method of curve tracing used is described in a paper called “Curve Tracing and Curve Detection in Images” by Karthik Raghapathy. The method involves calculating the second-order partial derivatives and finding the Eigen values of the Hessian matrix to find the normal to the points on the line. The angle of the line is calculated at any point where the line exists and is used to string the points together. For efficiency, the location of the next curve point can be predicted at one of three pixels in the direction of the curve as shown in Figure 41 [11]. In the case that the curve point is missing in this location, the pixels in the same direction but one-hop away can be checked, as shown in Figure 42. The retry point with the smallest difference in

angle is chosen if a curve point exists at multiple retry locations. The algorithm is implemented in the igvc_lines function in LaneCamera.cpp.

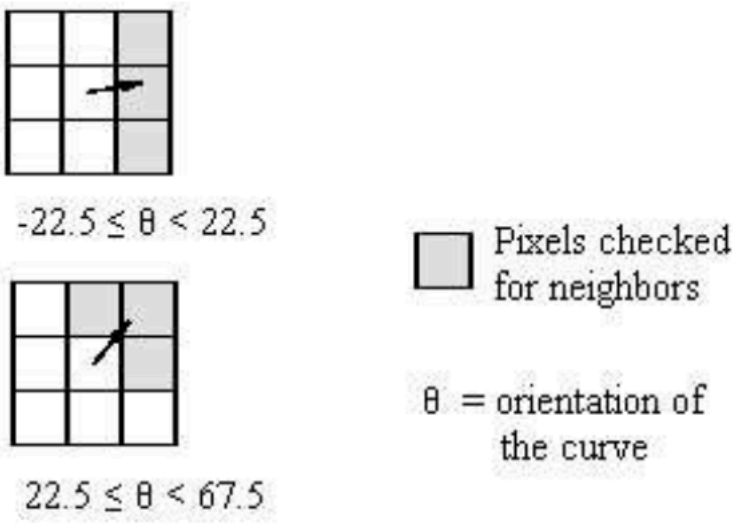


Figure 41. Linking Curve Points.

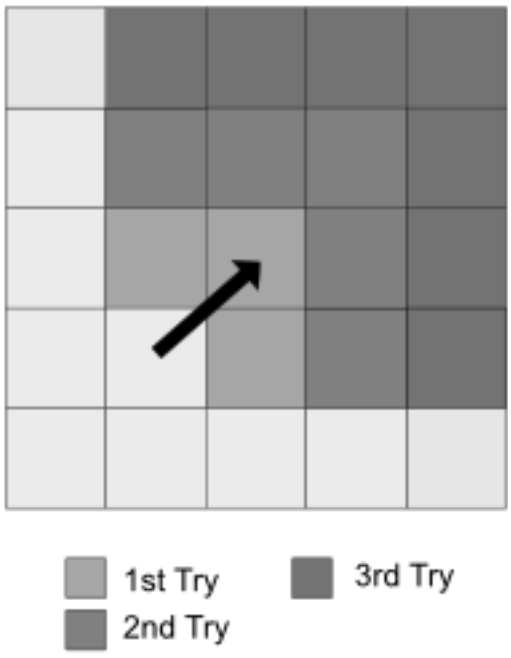


Figure 42. Retries for Missing Curve Points.

```
Mat color_filter(Mat image){
     Use OpenCV MorthologyEx
     Threshold blue channel 0-100 to 0 (out of 255)
     Threshold red channel 0-200 to 0 (out of 255)
     Bitwise_and all color channels
     Threshold 0-100 to 0 100-255 to 255
     return filtered image
}

// takes a B/W input image, minimum number of points in a
line, and the camera number (used for real-world mapping)
// returns list of lines
list<Line> igvc_lines(Mat input, minLength, camera_number){
     Perform algorithm in "Curve Tracing and Curve Detection
     in Images" by Karthik Raghapathy

     1st calculate dxx, dxy, dyx, dyy for image
     find normal to line
     string lines together
     if line is longer than minLength transform into real
     space and add to return list
     return list
}
```

Figure 43 – Pseudo Code for OpenCV Image Processing

The code that handles the processing of camera data is contained in the LaneCamera module (LaneCamera.h and LaneCamera.cpp). This code is shown below.

LaneCamera.h

```
#include <opencv\cv.h>
#include <opencv2\core\operations.hpp>
#include <opencv2\opencv.hpp>
#include <opencv2\core\core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv\highgui.h>
#include <iostream>
#include <stdlib.h>
#include <list>
#include <Polygon.h>

using namespace cv;
using namespace std;

#define FRAME_W 432
#define FRAME_H 240
#define FRAME_SIZE FRAME_W*FRAME_H
```

```
extern Mat color_filter(Mat &input, const char* name);
extern std::list<IGVC::Polygon *> igvc_lines(Mat &input, int minLength, int camera_number);
```

LaneCamera.cpp

```
#include "LaneCamera.h"
#include "Constants.h"
#include "PixelMap.h"

//#define RETRY_VIS

#define INVALID_ANGLE -360
#define INVALID_POINT Point(-1,-1)
#define MAX_RETRY 8 // maximum number of missing points allowed before end of line
#define MAX_CURVES 100 // maximum number of curves allowed to be found in image

// Internal Functions
deque<Point> igvc_lines_target(int x, int y, float curveDir, float curAngle, float* ang_mat, int cols, int
rows, list<int> *curvePoints, char* img, int retry_num = 0);
deque<Point> igvc_lines_target_edge(int x, int y, int d_x, int d_y, float* ang_mat, int cols, int rows,
list<int> *curvePoints, char* img, int retry_num);


/**********************************************************************
 *      Name:   color filter
 *      Inputs:
 *              - input:        input image matrix
 *              - name:                 name to display on output window
 *      Return:         image post filter
 *      Description:
 *              filters the image for the line detection function
 **********************************************************************/
Mat color_filter(Mat &input, const char* name) {
        resize(input, input, Size(FRAME_W, FRAME_H), 0, 0, INTER_CUBIC);
        char names[255] = "Unfiltered ";
        strcat(names, name);
        imshow(names, input);
        char * img = (char*)input.ptr();

        morphologyEx(input, input, MORPH_CLOSE, getStructuringElement(MORPH_ELLIPSE, Size(2,2)));

        morphologyEx(input, input, MORPH_OPEN, getStructuringElement(MORPH_ELLIPSE, Size(2,2)));

                vector<Mat> BGR_channels;
                split(input, BGR_channels);

                threshold(BGR_channels[0], BGR_channels[0], 100, 255, THRESH_TOZERO);
                threshold(BGR_channels[2], BGR_channels[2], 200, 255, THRESH_TOZERO);

                bitwise_and(BGR_channels[0], BGR_channels[1], input);
                bitwise_and(BGR_channels[2], input, input);

                threshold(input, input, 100, 255, THRESH_BINARY);

        char names2[255] = "Filtered ";
        strcat(names2, name);
        imshow(names2, input);

        return input;
}
```

```
/*********************************************************************
 *      Name:   igvc_lines
 *      Inputs:
 *              - input:                        input image matrix
 *              - minLength:                    minimum length of a recognized line
 *              - camera_number:        camera number to use in offset table
 *      Return:                 list of lines
 *      Description:
 *              Line Tracing Derived from "Curve Tracing and Curve Detection In Images" by Karthik
Raghapathy (2004)
 *              Hessian Matrix
 *              Steger's algorithm
 *********************************************************************/
std::list<IGVC::Polygon *> igvc_lines(Mat &input, int minLength, int camera_number){
        list<IGVC::Polygon *> ret;

        char ptr_dx[FRAME_SIZE];
        char ptr_dy[FRAME_SIZE];
        float ang_mat[FRAME_SIZE];
        //float *ang_mat = (float *)malloc(FRAME_SIZE*sizeof(float));

        uchar *image = input.ptr();
        int rows = input.rows;
        int cols = input.cols;
        int type = input.type();

        // Calculate First Order Derivatives
        for( int y = 1; y < rows; y++){
                for( int x = 1; x < input.cols; x++){
                        ptr_dx[y*cols+x] = image[(y*cols+x)] - image[y*cols + x-1];
                        ptr_dy[y*cols+x] = image[y*cols+x] - image[(y-1)*cols+x];
                }
        }

        // Max Strenght Point is used to begin line tracing
        deque<int> localCurvePoints;

        // Calculate Second Order Derivatives
        for( int y = 2; y < rows; y++){
                for( int x = 2; x < input.cols; x++){
                        float a = ptr_dx[y*cols+x] - ptr_dx[y*cols + x-1]; //dxx
                        float b = ptr_dx[y*cols+x] - ptr_dx[(y-1)*cols+x]; //dxy
                        float c = b;//ptr_dy[y*cols+x] - ptr_dy[y*cols + x-1]; //dyx
                        float d = ptr_dy[y*cols+x] - ptr_dy[(y-1)*cols+x]; //dyy

                        float gamma = sqrt(abs((a+d)*(a+d) - 4*b*c));
                        float strength = max((a+d+gamma)/2, (a+d-gamma)/2); // Maximum Eiganvalue

                        if ((strength - a) != 0){// && ((a + d)*(a + d) - 4 * b*c) >= 0 && b!=0) {
                                float n = b / (strength - a);
                                float nx = n / sqrt(1 + n*n); // Normalized Eiganvector_x
                                float ny = 1 / sqrt(1 + n*n); // Normalized Eiganvector_y
                                float t = -(nx*ptr_dx[y*cols + x] + ny*ptr_dy[y*cols + x]) / (nx*nx*a + 2
* nx*ny*b + ny*ny*d);
                                float angle = (int)cvFastArctan(ny, -nx);
                                if (x == 108 && y == 102) {
                                        int r = 7;
                                }

                                if (t*nx >= -0.5 && t*nx <= 0.5 && t*ny >= -0.5 && t*ny <= 0.5) {
                                        // Is a curve point
                                        image[y*cols + x] = 50; // used for debug display
```

```
                                        ang_mat[y*cols + x] = angle;

                                        if ((x > 5 && y > 5 && y < rows - 5 && x < cols - 5 && strength
>1.0)){
                                                localCurvePoints.push_front(y*cols + x); // Place all
start points for line construction in this list
                                        }

                                }
                                else {
                                        // Is not a curve point
                                        image[y*cols + x] = 0;
                                        ang_mat[y*cols + x] = INVALID_ANGLE;
                                }
                        }
                        else {
                                // Divide by Zero error - not a curve point
                                image[y*cols + x] = 0;
                                ang_mat[y*cols + x] = INVALID_ANGLE;
                        }

                }
        }

        // Make pixels that are not processed by point detection INVALID (-360)
        for (int y = 0; y < rows; y++) {
                ang_mat[y*cols] = INVALID_ANGLE;
                ang_mat[y*cols+1] = INVALID_ANGLE;
                ang_mat[y*cols+2] = INVALID_ANGLE;

                // Also clear from debug output
                image[y*cols] = 0;
                image[y*cols + 1] = 0;
                image[y*cols + 2] = 0;
        }
        for (int x = 0; x < cols; x++) {
                ang_mat[x] = INVALID_ANGLE;
                ang_mat[cols + x] = INVALID_ANGLE;

                // Also clear from debug output
                image[x] = 0;
                image[cols + x] = 0;
        }

        // Curve Tracing
        Mat temp = Mat(rows, cols, input.type()); // Used to hold curve data temporarily
        int foundCurves = 0;

        while (!localCurvePoints.empty() && foundCurves <= MAX_CURVES) // While start points exist
        {
                int index = localCurvePoints.front();
                int x = index%cols; int y = index/cols;
                localCurvePoints.pop_front();

                int count = 0; // Counts number of points in curve
                int x_prev, y_prev;
                list<int> _points; // Keeps track of points in current line;
                list<int> *points = &_points;
                temp.setTo(0); // clear temporary line visualization matrix
                bool end = false; // Marks end of curve
                float curAngle, angle1 = 0, angle2 = 0, curveDir;
```

```
		// Curve Tracing loop
		while (!end && count < 500) {
			count++;
			if (count > 3) {
				curAngle = (ang_mat[x + cols*y] + angle1 + angle2) / 3;
			}
			else {
				curAngle = ang_mat[x + cols*y];
			}
			x_prev = x; y_prev = y;

			angle1 = angle2;
			angle2 = curAngle;

			curveDir = curAngle + 90;
			if (curveDir > 180) curveDir -= 360; // Put curve direction in range -180 to 180
degrees

			points->push_front(y*cols+x); // Add point to curve list

			temp.ptr()[x + cols*y] = 255; // Visualize curve point

			deque<Point> point_buf = igvc_lines_target(x, y, curveDir, curAngle, ang_mat,
cols, rows, points, (char*)temp.ptr());
			if (point_buf.empty()) {
				end = true;
			}
			else {
				count += point_buf.size();
				Point temp_p = point_buf.back();
				point_buf.pop_back();
				x = temp_p.x;
				y = temp_p.y;
			}

			// Draw

			while (!point_buf.empty()) {
				Point temp_p = point_buf.back();
				points->push_front(temp_p.x + cols*temp_p.y);
				temp.ptr()[temp_p.x + cols*temp_p.y] = 255; // Visualize curve point
				point_buf.pop_back();
			}


			if (find(points->begin(), points->end(), y*cols + x) != points->end()) {
				end = true;
			}

			if (ang_mat[x + cols*y] == INVALID_ANGLE) {
				// Search for continuation
				end = true;
			}
		}

		// Only add lines to list if they are longer than the minumim length
		if (count > minLength) {
			bitwise_or(temp, input, input);
			foundCurves++;

			IGVC::Polygon* line = new IGVC::Polygon();
```

```
                        while (!points->empty()) {
                                int f_index = points->front();

                                // Translate to local coordinates
                                float translated_x, translated_y;
                                pixelMap(f_index % cols, f_index / cols, camera_number, translated_x,
translated_y);
                                IGVC::Point* pt = new IGVC::Point();
                                pt->x = translated_x; pt->y = translated_y;
                                line->AddPoint(pt); // Add point to Polygon

                                // For speed improvements don't process start points that are already in
curves
                                deque<int>::iterator f_remove = find(localCurvePoints.begin(),
localCurvePoints.end(), f_index);
                                if (f_remove != localCurvePoints.end()) {
                                        localCurvePoints.erase(f_remove);
                                }
                                points->pop_front();
                        }

                        ret.push_front(line); // Add Polygon to return list
                }
        }

        //namedWindow("Contours", WINDOW_AUTOSIZE);
        imshow(CAMERA_NAME[camera_number], input);

        return ret;
}


/*********************************************************************
 *      Name:   igvc_lines_target_edge
 *      Inputs:
 *              - x:            last confirmed point x
 *              - y:            last confirmed point y
 *              - d_x:          x derivative at curve point x,y
 *              - d_y:          y_derivative at curve point x,y
 *              - ang_mat:      matrix of curve angles
 *              - cols:         number of columns in image
 *              - rows:         number of rows in image
 *              - curvePoints:  list of curve points
 *              - img:          image (only used for debug)
 *              - retry_num:    number of retries when points are missing from the curve
 *      Return:                 deque of next points in line
 *      Description:    Recursive function finds the next point in the curve when the point is in the
middle of the possible field
 *********************************************************************/
deque<Point> igvc_lines_target(int x, int y, float curveDir, float curAngle, float* ang_mat, int cols, int
rows, list<int> *curvePoints, char* img,  int retry_num) {
        float ang1, ang2, ang3;
        Point p1, p2, p3;
        int d_x1, d_y1, d_x2, d_y2, d_x3, d_y3;

        /*Uncomment to visualize retry pattern*/
#ifdef RETRY_VIS
        img[x + cols*y] = 200; // Visualize curve point
#endif

        // Safety Check
        if (x > cols - 2 || y > rows - 2 || x < 2 || y < 2) {
```

```
        // Get rid of egde conditions
        deque<Point> r; //empty
        return r;
}

// Pixel Canidates
if (curveDir >= -22.5 && curveDir < 22.5) {
        // Check pixels UR, R, and DR
        d_x1 = 1;
        d_y1 = -1; // UR
        d_x2 = 1;
        d_y2 = 0; //R
        d_x3 = 1;
        d_y3 = 1; //DR
}
else if (curveDir >= 22.5 && curveDir < 67.5) {
        // Check pixels U, UR, R
        d_x1 = 0;
        d_y1 = - 1; // U
        d_x2 = 1;
        d_y2 = -1; // UR
        d_x3 = 1;
        d_y3 = 0; //R
}
else if (curveDir >= 67.5 && curveDir < 112.5) {
        // Check pixels UL, U, UR
        d_x1 = -1;
        d_y1 = - 1; // UL
        d_x2 = 0;
        d_y2 = -1; // U
        d_x3 = 1;
        d_y3 = -1; // UR
}
else if (curveDir >= 112.5 && curveDir < 157.5) {
        // Check pixels L, UL, U
        d_x1 = -1;
        d_y1 = 0; // L
        d_x2 = -1;
        d_y2 = -1; // UL
        d_x3 = 0;
        d_y3 = -1; // U
}
else if (curveDir >= -67.5 && curveDir < -22.5) {
        // Check pixels D, DR, R
        d_x1 = 0;
        d_y1 = 1; // D
        d_x2 = 1;
        d_y2 = 1; //DR
        d_x3 = 1;
        d_y3 = 0; //R
}
else if (curveDir >= -112.5 && curveDir < -67.5) {
        // Check pixels D, DL, DR
        d_x1 = -1;
        d_y1 = 1; //DL
        d_x2 = 0;
        d_y2 = 1; // D
        d_x3 = 1;
        d_y3 = 1; //DR
}
else if (curveDir >= -157.5 && curveDir < -112.5) {
        // Check pixels D, DL, L
```

```
                d_x1 = -1;
                d_y1 = 0; // L
                d_x2 = -1;
                d_y2 = 1; //DL
                d_x3 = 0;
                d_y3 = 1; // D
        }
        else if (curveDir >= -180 && curveDir <= 180) {
                // Check pixels DL, L, UL
                d_x1 = -1;
                d_y1 = -1; // UL
                d_x2 = -1;
                d_y2 = 0; // L
                d_x3 = -1;
                d_y3 = 1; //DL
        }
        else {
                deque<Point> r;
                return r; // Bad Angle return empty stack
        }

        p1 = Point(x + d_x1, y + d_y1);
        p2 = Point(x + d_x2, y + d_y2);
        p3 = Point(x + d_x3, y + d_y3);

        ang1 = ang_mat[p1.x + cols *p1.y];
        ang2 = ang_mat[p2.x + cols *p2.y];
        ang3 = ang_mat[p3.x + cols *p3.y];

        deque<Point> d1, d2, d3;

        if (ang1 == INVALID_ANGLE && ang2 == INVALID_ANGLE && ang3 == INVALID_ANGLE) {
                // Retry
                if (retry_num < MAX_RETRY-1) {
                        retry_num++;

                        d1 = igvc_lines_target_edge(p1.x, p1.y, d_x1, d_y1, ang_mat, cols, rows,
curvePoints, img, retry_num);
                        d2 = igvc_lines_target(p2.x, p2.y, curveDir, curAngle, ang_mat, cols, rows,
curvePoints, img, retry_num);
                        d3 = igvc_lines_target_edge(p3.x, p3.y, d_x3, d_y3, ang_mat, cols, rows,
curvePoints, img, retry_num);
                        Point tp1, tp2, tp3;
                        if(!d1.empty())
                                tp1 = d1.back();
                        if(!d2.empty())
                                tp2 = d2.back();
                        if(!d3.empty())
                                tp3 = d3.back();
                        ang1 = ang_mat[tp1.x + cols *tp1.y];
                        ang2 = ang_mat[tp2.x + cols *tp2.y];
                        ang3 = ang_mat[tp3.x + cols *tp3.y];
                }
                else {
                        deque<Point> r;
                        return r; // return empty stack
                }
        }

        // return the winning point (lesser angle)
        if (abs(curAngle - ang1) < abs(curAngle - ang2) && abs(curAngle - ang1) < abs(curAngle - ang3)) {
                d1.push_front(p1);
```

```
                return d1;
        }
        else if (abs(curAngle - ang2) < abs(curAngle - ang1) && abs(curAngle - ang2) < abs(curAngle -
ang3)) {
                d2.push_front(p2);
                return d2;
        }
        else {
                d3.push_front(p3);
                return d3;
        }
}

/*********************************************************************
 *      Name:   igvc_lines_target_edge
 *      Inputs:
 *              - x:            last confirmed point x
 *              - y:            last confirmed point y
 *              - d_x:          x derivative at curve point x,y
 *              - d_y:          y_derivative at curve point x,y
 *              - ang_mat:      matrix of curve angles
 *              - cols:         number of columns in image
 *              - rows:         number of rows in image
 *              - curvePoints:  list of curve points
 *              - img:          image (only used for debug)
 *              - retry_num:    number of retries when points are missing from the curve
 *      Return:                 deque of next points in line
 *      Description:    Recursive function finds the next point in the curve when the point is on the
edge of the possible field
 *********************************************************************/
deque<Point> igvc_lines_target_edge(int x, int y, int d_x, int d_y, float* ang_mat, int cols, int rows,
list<int> *curvePoints, char* img, int retry_num) {
        // Safety Check
        if (x > cols - 2 || y > rows - 2 || x < 2 || y < 2) {
                // Get rid of egde conditions
                deque<Point> r; //empty
                return r;
        }

        /*Uncomment to visualize retry pattern*/
#ifdef RETRY_VIS
        img[x + cols*y] = 200; // Visualize curve point
#endif

        deque<Point> d;
        Point p = Point(x + d_x, y + d_y);
        float ang = ang_mat[p.x + cols *p.y];
        if (ang == INVALID_ANGLE) {
                if (retry_num < MAX_RETRY - 1) {
                        retry_num++;
                        d = igvc_lines_target_edge(p.x, p.y, d_x, d_y, ang_mat, cols, rows, curvePoints,
img, retry_num);
                }
                else {
                        return d; // return empty stack
                }
        }

        d.push_front(p);
        return d;
}
```

3.7.4 *Object Mapping*

Mapping with Respect to Camera

Line and pothole perception is accomplished through camera images. Geometry is used to map each pixel in the image to a point in real space. Some assumptions are necessary.

Assuming the pixel sensors on the camera are much closer together than the distance to the image that they are capturing, the camera sensor can be approximated as a point where each pixel captures light at an angle from the center of the camera's field of vision. The field of vision of a camera consists of a horizontal and vertical component, θ_{fh} and θ_{fv} respectively. These angles are included with camera specifications or can be measured. Figure 44 demonstrates this model, and will be demonstrated using white tape as it will be in the competition. The angles are calculated from the center of the pixel. The angle θ_x for each pixel is calculated using

$$\theta_x(w) = \frac{\theta_{fh}(2w - W - 1)}{2W}. \quad (34)$$

Here *w* is the pixel number from the left side of the image and W is the width of the image sensor in pixels.

The angle θ_y is calculated similarly using

$$\theta_y(y) = \frac{\theta_{fv}(2h - H - 1)}{2H}. \quad (35)$$

Here *h* is the pixel number from the top of the image and H is the height of the image sensor in pixels.

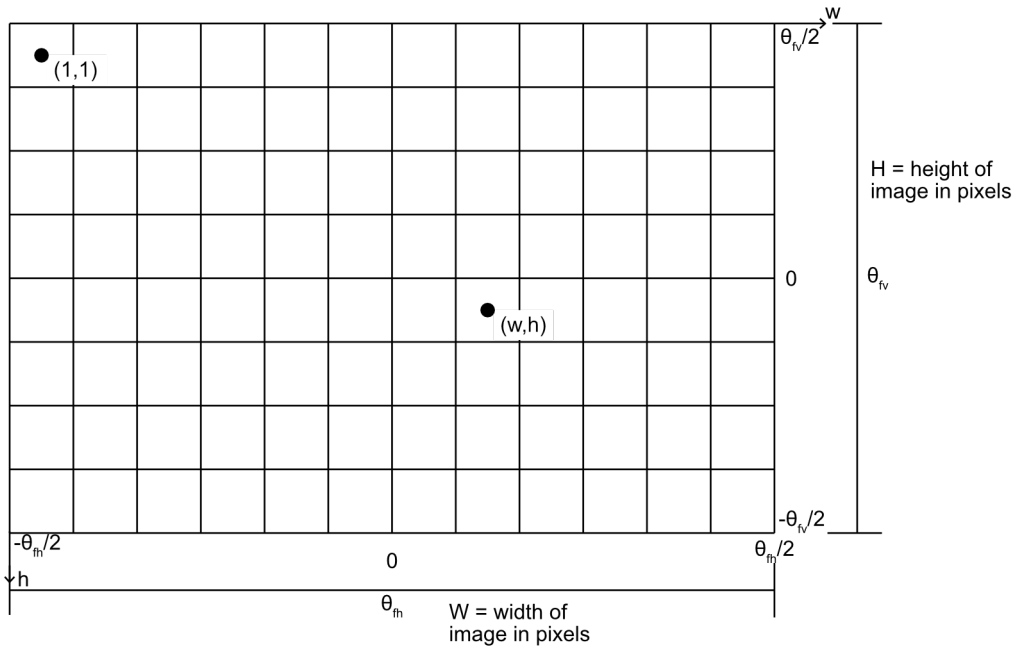


Figure 44. Camera Pixel Grid.

In addition to the camera model, other assumptions must be made about the configuration of the vehicle. First, all objects of interest for camera detection are assumed to be located on the ground plane which is flat always perpendicular to the mast holding the camera. In the following figures the ground plane is the x-y plane. The camera is mounted at height *h* from the ground plane. The normal to the camera and the normal to the ground must both exist exclusively in the y-z plane. The camera angle θ_c is defined as the angle between the plane perpendicular to the mast at height *h* and the center of the camera's field of vision. Figure 45 shows the geometries used in the derivation of

$$y(\theta_y) = \frac{h}{\tan(\theta_c + \theta_y)}, \tag{36}$$

the formula for mapping pixels to real space y-coordinates from the pixel angle approximation θ_y.

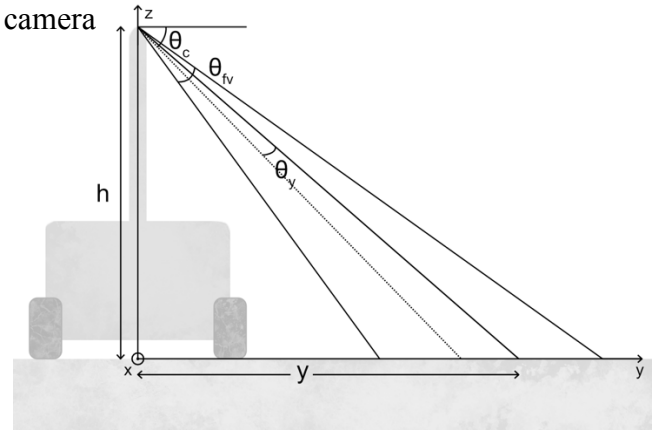


Figure 45. Geometry for Y Coordinates.

Figure 46 shows the geometry for the derivation of

$$x\left(\theta_x, \theta_y\right) = y\left(\theta_y\right) \cdot \tan\left(\theta_x\right), \tag{37}$$

the formula used to compute the real space x-coordinate for each pixel given the pixel's x-angle θ_x and corresponding real-world y coordinate.

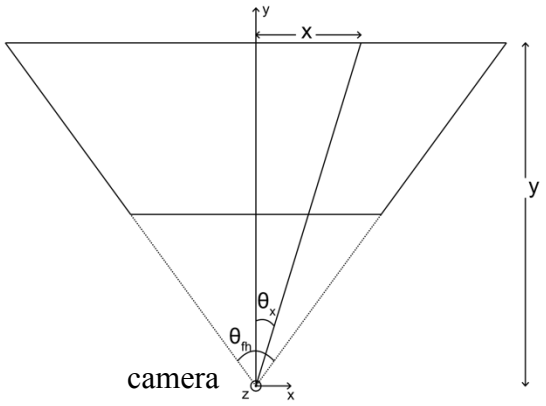


Figure 46. Geometry for X Coordinates.

The real-world mapping of each pixel in a compressed 270x480 pixel image from the Logitech C920 webcam is shown in Figure 47 with the camera mounted at 45° and height 5ft. The C920 camera has a horizontal field of view θ_{fh} = 70.42° and vertical field of view θ_{fv} = 43.30°.

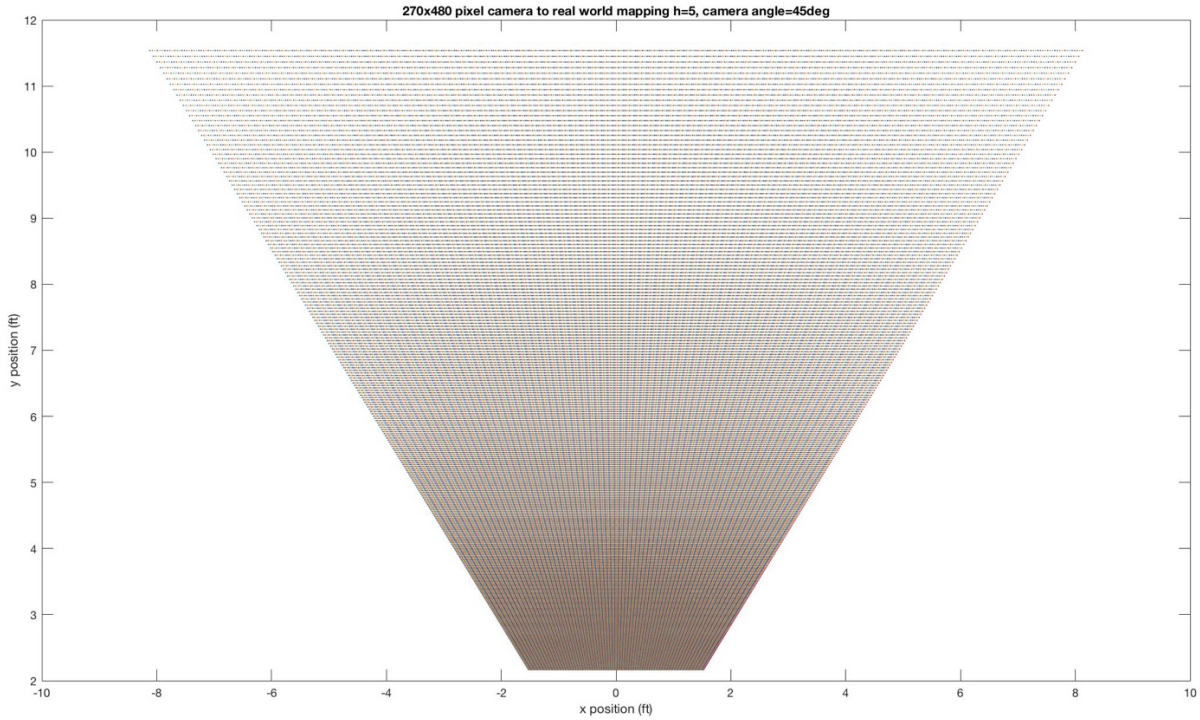


Figure 47. Pixel Mapping to Real World Position.

Mapping with Respect to GPS coordinates

Three cameras are mounted to the vehicle as shown in Figure 48. The mast holding the cameras and GPS antenna is designated by the solid black dot in the center of the vehicle. The local coordinate axis for the vehicle is defined so the direction of motion is in the negative y direction. The positive y-axis is designated as 0° and the direction of forward motion as 180°. Camera 1 is located at 90°, camera 2 at 180°, and camera 3 at 270°.

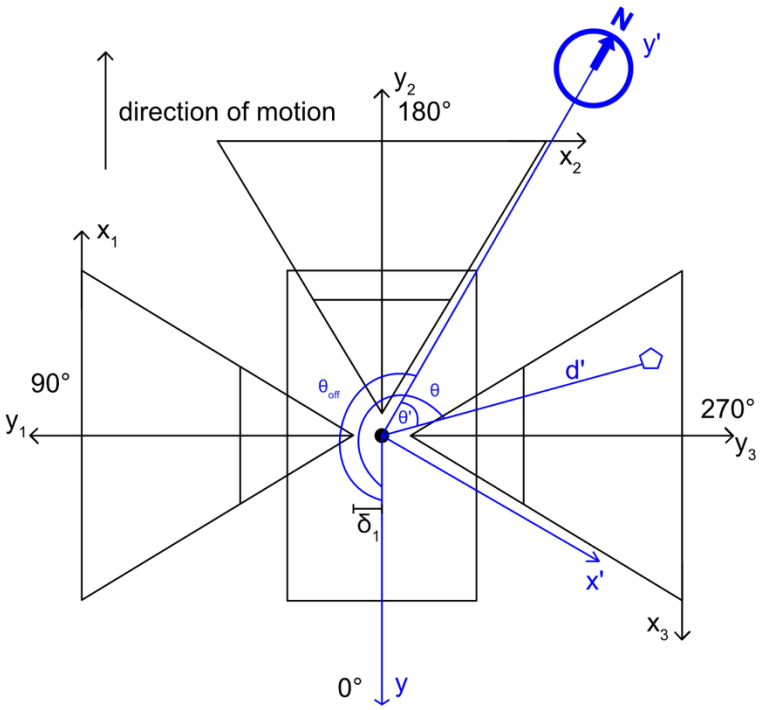


Figure 48. Camera Position to Global Position Mapping.

For each camera, there is an offset δ_i from the vehicle's GPS reference point (black dot) and the camera's origin. This offset only exits in the y direction of each camera. The process for translating the position of an object from camera begins by adding δ to the y coordinate and finding the distance and angle to the object from the GPS reference point. The distance *d'* is given by

$$d' = \sqrt{(y_i + \delta_i)^2 + x_i^2}. \tag{38}$$

Here the subscript *i* refers to the camera number. The angle from the y-axis reference is given by

$$\theta = \tan^{-1}\left(\frac{x_i}{y_i + \delta_i}\right) + \theta_i \tag{39}$$

where θ_i is the offset angle for the camera from the y-axis. The heading angle for the vehicle θ_{off} is acquired from the compass. The object's offset from north θ' is calculated by

$$\theta' = \theta - \theta_{off}. \tag{40}$$

The latitudinal and longitudinal coordinate offsets for the object from vehicle can be determined by projecting the distance vector onto the y' (latitude) and x' (longitude) axes using Equation (41) and Equation (42). The GPS must be accurate to a few inches for this method to effectively plot the location of objects.

$$\delta_{lat} = d' \cos\theta' \tag{41}$$
$$\delta_{long} = d' \sin\theta' \tag{42}$$

The values of δ_{lat} and δ_{long} have the same units as the height of the cameras, likely feet. These values need to be converted to GPS angles. Since the curvature of the earth's surface is very small over the distance the vehicle will travel, a flat earth can be assumed. The conversion between feet and GPS angles can remain constant over the entire course, however, these constants will change depending on the location on Earth. These values will be calculated in a calibration routine when the GPS takes its first reading. The distance between two points on Earth specified as latitude and longitude can be calculated using the haversine formula in Equation (43).

$$a = \sin^2\left(\frac{\Delta\phi}{2}\right) + \cos\phi_1 \cdot \cos\phi_2 \cdot \sin^2\left(\frac{\Delta\lambda}{2}\right) \tag{43}$$
$$c = 2 \cdot \operatorname{atan2}(\sqrt{a}, \sqrt{1-a})$$
$$d = R \cdot c$$

Here φ is latitude, λ is longitude, and R is the earth's radius (6,371km) [12]. These displacements are added to the GPS coordinates of the vehicle to get the GPS position of the object.

The same process is used for mapping objects detected by LiDAR to GPS coordinates. The LiDAR's angles match the angles designated in Figure 48.

Code

The code for mapping image points in real space is contained in PixelMap.h.

PixelMap.h

```
#ifndef PIXEL_MAP_H
#define PIXEL_MAP_H

#include <math.h>
#include "Constants.h"


/*********************************************************************
 *      Name:   pixelMap
 *      Inputs:
 *              - pixel_X: pixel x location
 *              - pixel_Y: pixel y location
 *              - camera_number: camera number in offset tables
 *              - &x: reference to float (will return real-world x coordinate)
 *              - &y: reference to float (will return real-world y coordinate)
 *                      units of x and y will match CAMERA_HEIGHT units
 *      Description: Converts pixel coordinate to real world coordinate
 *********************************************************************/
static void pixelMap(int pixel_X, int pixel_Y, int camera_number, float& x, float& y){
        float angle_x = FIELD_OF_VISION_H*(2.0*pixel_X-PIXEL_WIDTH-1.0)/(2.0*PIXEL_WIDTH);
        float angle_y = FIELD_OF_VISION_V*(2.0*pixel_Y-PIXEL_HEIGHT-1.0)/(2.0*PIXEL_HEIGHT);

        y = CAMERA_HEIGHT/tan(PI*(angle_y+CAMERA_ANGLE[camera_number])/180.0);
        x = y * tan(PI*angle_x/180.0) ;

        // If camera is not facing heading direction compute the offset
```

```
        if(CAMERA_HORIZONTAL_ANGLE[camera_number] != 0){
                float m = sqrt(x*x + y * y);
                float phi = atan2f(x, y); //in radians

                x = m*sin(CAMERA_HORIZONTAL_ANGLE[camera_number] + phi);
                y = m*cos(CAMERA_HORIZONTAL_ANGLE[camera_number] + phi);
        }
        y =  y + CAMERA_Y_OFFSET[camera_number];
        x = x - CAMERA_X_OFFSET[camera_number];
}
#endif
```

The code used to map LiDAR and camera points and draw them in the visualizer is in Map.h and Map.cpp below. The LiDAR mapping code groups points together into objects based on the difference in x-y derivatives between points and the distances between points. Like the line tracing algorithm, the algorithm is robust enough to eliminate objects that do not contain a greater than the minimum number of points. This eliminates erroneous reflections from being treated as objects.

Map.h

```
#include <cstdint>
#include <list>
#include "Polygon.h"
#include "Scan.h"
#include "LaneCamera.h"

#define MAP_SIZE 1000
#define CM_PER_M 100.0

namespace IGVC{
class Map
{
private:
        std::list<IGVC::Polygon *> objects;
        void leftPath(int x, int y);
        void rightPath(int x, int y);
        void middlePath(int x, int y);

public:
        Map(void);
        ~Map(void);
        void ProcessLidar(Scan *scan);
        void Map::ProcessCamera(Mat frame, const char* name, int camera_number);
        void Draw();
        void Clear();
        void FindPath();
        char cam_map[MAP_SIZE][MAP_SIZE];
        int tar_x, tar_y;//cm
};
}
```

Map.cpp

```
#include "Map.h"
#include <math.h>
#include "Polygon.h"
#include "Constants.h"
#include "PixelMap.h"
```

```
#include "GL/freeglut.h"

#define OBJECT_SENSITIVITY 0.3 //meters
#define MIN_SHAPE_POINTS 3
#define ANGLE_SENSITIVITY 0.5

#define OBSTACLE -1
#define LIDAR_OBSTACLE -2
#define DANGER_ZONE -3


using namespace std;
using namespace IGVC;

/*****************************************
 *      Name:   Map [Constructor]
 *      Description: Constructs the Map Object
 *****************************************/
Map::Map(void)
{
}

/*****************************************
 *      Name:   Map [Destructor]
 *      Description: Frees memory from object list
 *****************************************/
Map::~Map(void)
{
        for(std::list<IGVC::Polygon *>::iterator it = objects.begin(); it != objects.end(); ++ it){
                        delete *it;
                }
        objects.clear();
}

#ifndef REMOVE_LIDAR_OBJECT_DETECTION
/*****************************************************************************
 *      Name:   Process Lidar
 *      Inputs:
 *              - scan:         Scan object
 *      Description:    Uses data from Scan object to construct map
 *****************************************************************************/
void Map::ProcessLidar(Scan *scan)
{
        // Detect Object
        float start_point = scan->ScanDistance()[0];
        IGVC::Polygon *shape = new IGVC::Polygon();
        shape->type = LIDAR_TYPE;
        int points_in_shape = 0;
        bool first = true;
        bool corner = false;
        for (int i = 4; i < scan->ScanSize(); i++){
                corner = false;
                float d1 = scan->Derivative()[i], d2 = scan->Derivative()[i-1], d3 = scan->Derivative()[i-
3], d4 = scan->Derivative()[i-4];
                float a1 = atan((d1+d2)/2);
                float a2 = atan((d3+d4)/2);

                float dif = abs(a1 - a2);
                if (dif > ANGLE_SENSITIVITY) corner = true;

                // Check for end of object
```

```
                // Block out area behind lidar from -140 to -180 and 140 to 180
                if (scan->ScanDistance()[i] <= (OBJECT_SENSITIVITY/2 + start_point) && scan-
>ScanDistance()[i] >= (start_point - OBJECT_SENSITIVITY/2) && scan->ScanAngle()[i] < 140 && scan-
>ScanAngle()[i] > -140){
                        // Add first point or "corner" point
                        if(first || corner){
                                Point *point = new Point();
                                point->x = -scan->ScanY()[i] + LIDAR_X_OFFSET; //meters
                                point->y = scan->ScanX()[i] + LIDAR_Y_OFFSET; //meters
                                shape->AddPoint(point);
                        }
                        first = false;
                        points_in_shape++;
                        start_point = scan->ScanDistance()[i];
                        if(i == scan->ScanSize()-1){
                                // DON'T LOSE LAST OBJECT
                                if(i>0){
                                        //add end point
                                        Point *point = new Point();
                                        point->x = -scan->ScanY()[i-1] + LIDAR_X_OFFSET; //meters
                                        point->y = scan->ScanX()[i-1] + LIDAR_Y_OFFSET; //meters
                                        shape->AddPoint(point);
                                }
                                objects.push_back(shape);
                                // Create new shape for deletion (wasteful but necessary for now)
                                shape = new IGVC::Polygon();
                                start_point = scan->ScanDistance()[i];
                        }
                }else{
                        if(points_in_shape >= MIN_SHAPE_POINTS){
                                if(i>0){
                                        //add end point
                                        Point *point = new Point();
                                        point->x = -scan->ScanY()[i-1] + LIDAR_X_OFFSET; //meters
                                        point->y = scan->ScanX()[i-1] + LIDAR_Y_OFFSET; //meters
                                        shape->AddPoint(point);
                                        shape->type = LIDAR_TYPE;
                                }
                                objects.push_back(shape);
                        }else{
                                delete shape;
                                shape = NULL;
                        }
                        // Clean Up and Reset for next point
                        shape = new IGVC::Polygon();
                        shape->type = LIDAR_TYPE;
                        start_point = scan->ScanDistance()[i];
                        first = true;
                        points_in_shape = 0;
                }
        }
        delete shape;
        shape = NULL;
}
#else
// discrete version
void Map::ProcessLidar(Scan *scan)
{
        float x,y;
        float start_point = scan->ScanDistance()[0];
        for (int i = 0; i < scan->ScanSize(); i++){
                // Block out area behind lidar from -140 to -180 and 140 to 180
```

```
                if (scan->ScanDistance()[i] <= (OBJECT_SENSITIVITY/2 + start_point) && scan-
>ScanDistance()[i] >= (start_point - OBJECT_SENSITIVITY/2) && scan->ScanAngle()[i] < 140 && scan-
>ScanAngle()[i] > -140){
                        x = -scan->ScanY()[i] + LIDAR_X_OFFSET; //meters
                        y = scan->ScanX()[i] + LIDAR_Y_OFFSET; //meters

                        if(x > (-MAP_SIZE+1)/2/CM_PER_M && x < (MAP_SIZE-1)/2/CM_PER_M && y > (-
MAP_SIZE+1)/2/CM_PER_M && y < (MAP_SIZE-1)/2/CM_PER_M){
                                cam_map[(int)(x*CM_PER_M)+MAP_SIZE/2][(int)(y*CM_PER_M)+MAP_SIZE/2] =
LIDAR_OBSTACLE;
                        }
                }
                start_point = scan->ScanDistance()[i];
        }
}
#endif
/*************************************************************************
 *      Name:   Draw
 *      Description:    Draws the objects in the object list
 *************************************************************************/
void Map::Draw(){
        for(std::list<IGVC::Polygon *>::iterator it = objects.begin(); it != objects.end(); ++ it){
                (*it)->Draw();

        }


        //draw discrete points
        glLineWidth(2.0f);


        for(int y = 0; y < MAP_SIZE; y++){
                for (int x = 0; x < MAP_SIZE; x++){
                        if(cam_map[x][y] == OBSTACLE){
                                glBegin(GL_LINE_STRIP);
                                glColor3f(1.0, 1.0, 0.0);
                                glVertex2d((x-MAP_SIZE/2)/VISUALIZER_SCALE/CM_PER_M, (y-
MAP_SIZE/2)/VISUALIZER_SCALE/CM_PER_M);
                                glVertex2d((x-MAP_SIZE/2)/VISUALIZER_SCALE/CM_PER_M+0.001, (y-
MAP_SIZE/2)/VISUALIZER_SCALE/CM_PER_M+0.001);
                                glEnd();
                        }else if(cam_map[x][y] == LIDAR_OBSTACLE){
                                glBegin(GL_LINE_STRIP);
                                glColor3f(1.0, 0.0, 0.0);
                                glVertex2d((x-MAP_SIZE/2)/VISUALIZER_SCALE/CM_PER_M, (y-
MAP_SIZE/2)/VISUALIZER_SCALE/CM_PER_M);
                                glVertex2d((x-MAP_SIZE/2)/VISUALIZER_SCALE/CM_PER_M+0.001, (y-
MAP_SIZE/2)/VISUALIZER_SCALE/CM_PER_M+0.001);
                                glEnd();
                        }
                }
        }

        //draw path
        for(int y = 0; y < MAP_SIZE; y++){
                for (int x = 0; x < MAP_SIZE; x++){
                        if(cam_map[x][y] > 0){
                                glBegin(GL_LINE_STRIP);
                                glColor3f(cam_map[x][y]/255.0, cam_map[x][y]/255.0, cam_map[x][y]/255.0);
                                glVertex2d((x-MAP_SIZE/2)/VISUALIZER_SCALE/CM_PER_M, (y-
MAP_SIZE/2)/VISUALIZER_SCALE/CM_PER_M);
```

```
                                glVertex2d((x-MAP_SIZE/2)/VISUALIZER_SCALE/CM_PER_M+0.001, (y-
MAP_SIZE/2)/VISUALIZER_SCALE/CM_PER_M+0.001);
                                glEnd();
                        }
                }
        }

        //target
        glBegin(GL_LINE_STRIP);
        glColor3f(1.0, 0.0, 1.0);
        glVertex2d(0, SAFE_FRONT_SPACE/VISUALIZER_SCALE/CM_PER_M);
        glVertex2d((tar_x - (MAP_SIZE/2))/VISUALIZER_SCALE/CM_PER_M, (tar_y -
(MAP_SIZE/2))/VISUALIZER_SCALE/CM_PER_M);
        glEnd();

}

/*********************************************************************
 *      Name:   Process Camera
 *      Input:  OpenCV Mat - frame from camera
  *     Description:    Processes camera frame to find lane lines in real space
 *********************************************************************/
void Map::ProcessCamera(Mat frame, const char* name, int camera_number){
        Mat  IMG_OUT;
        // Read the video capture stream into the frame matrix, change to HSV colorspace.

        IMG_OUT = color_filter(frame, name);

#ifdef BYPASS_LINE_TRACING
        // DIRECT TO POINT
        for (int y = 0; y < frame.rows; y++) {
                for (int x = 0; x <  frame.cols; x++) {
                        if(frame.ptr()[frame.cols*y+x] !=0){
                                float translated_x, translated_y;
                                pixelMap(x, y, camera_number, translated_x, translated_y);
                                if(translated_x > (-MAP_SIZE+1)/2/CM_PER_M && translated_x < (MAP_SIZE-
1)/2/CM_PER_M && translated_y > (-MAP_SIZE+1)/2/CM_PER_M && translated_y < (MAP_SIZE-1)/2/CM_PER_M){

        cam_map[(int)(translated_x*CM_PER_M)+MAP_SIZE/2][(int)(translated_y*CM_PER_M)+MAP_SIZE/2] =
OBSTACLE;
                                }
                        }
                }
        }
#else
        //Line Tracing
        list<IGVC::Polygon *> obj = igvc_lines(IMG_OUT, MIN_LINE_LENGTH, camera_number);
        while(!obj.empty()){
                Polygon * poly = obj.front();
                if (camera_number == 0)
                        poly->type = CAMC_TYPE;
                else if(camera_number == 1)
                        poly->type = CAML_TYPE;
                else if(camera_number == 2)
                        poly->type = CAMR_TYPE;
                objects.push_front(poly);
                obj.pop_front();
        }
#endif
}

/*********************************************************************
```

```
 *      Name:   Clear
 *      Description:    Clears the map
 ***********************************************************************/
void Map::Clear(){
        for(std::list<IGVC::Polygon *>::iterator it = objects.begin(); it != objects.end(); ++ it){
                delete *it;
        }
        objects.clear();

        //clear
        for(int y = 0; y < MAP_SIZE; y++){
                for (int x = 0; x < MAP_SIZE; x++){
                        cam_map[x][y] = 0;
                }
        }
}

void Map::FindPath(){
        int x=MAP_SIZE/2, y=MAP_SIZE/2; // start point
        int prev_x = x, prev_y = y;
        char prev_point = cam_map[x][y];

        bool safe = true;

        // first run check y+front+1, x-side-1, x-side, x+side+1
        middlePath(x+SAFE_SIDE_SPACE,y+1);
        middlePath(x-SAFE_SIDE_SPACE, y+1);

        int off = 0;
        for(int front = 0; front < 10; front++){
                char f1 = cam_map[x+off+1][y+SAFE_FRONT_SPACE+front];
                char f2 = cam_map[x+off][y+SAFE_FRONT_SPACE+front];
                if(f2 - f1 > 0){ //derivative
                        //go right
                        off += 1;
                }else if(f2 - f1 < 0){
                        //go left
                        off -= 1;
                }
        }

        tar_x = x+off;
        tar_y = y+10+SAFE_FRONT_SPACE;
}

void Map::leftPath(int x, int y){
        if(x == 1 || x == MAP_SIZE-1 || y ==1 || y==MAP_SIZE-1) return;//out of bounds
        // Check for Obstacles
        if(cam_map[x-1][y] < 0 || cam_map[x-1][y+1] < 0 || cam_map[x][y+1] < 0){
                // Yes we are beside an obstacle
                cam_map[x][y] = 127;
        }else{
                leftPath(x-1, y+1);
                if(cam_map[x-1][y+1] > 0){
                        char val = max((char)(cam_map[x-1][y+1] - 1), cam_map[x][y]);
                        cam_map[x][y] = val;
                }
        }
}

void Map::rightPath(int x, int y){
        if(x == 1 || x == MAP_SIZE-1 || y ==1 || y==MAP_SIZE-1) return;//out of bounds
```

```
        // Check for Obstacles
        if(cam_map[x][y+1] < 0 || cam_map[x+1][y+1] < 0 || cam_map[x+1][y] < 0){
                // Yes we are beside an obstacle
                cam_map[x][y] = 127;
        }else{
                rightPath(x+1, y+1);
                if (cam_map[x+1][y+1] > 0){
                        char val = max((char)(cam_map[x+1][y+1] - 1), cam_map[x][y]);
                        cam_map[x][y] = val;
                }
        }
}

void Map::middlePath(int x, int y){
        if(x == 1 || x == MAP_SIZE-1 || y ==1 || y==MAP_SIZE-1) return; //out of bounds
        // Check for Obstacles
        if(cam_map[x][y+1] < 0 || cam_map[x-1][y+1] < 0 || cam_map[x+1][y] < 0){
                // Yes we are beside an obstacle
                cam_map[x][y] = 127;
        }else{
                middlePath(x,y+1);
                leftPath(x-1,y+1);
                rightPath(x+1,y+1);
                char val = max(cam_map[x][y+1], cam_map[x-1][y+1]);
                val = max(val, cam_map[x+1][y+1]);
                val = max(val, cam_map[x][y]);
                if(val > 0)
                        cam_map[x][y] = val-1;
        }

}
```

Utility Code

Map.h and Map.cpp and the LiDAR and camera code require some utility objects Polygon, and Point. The code for these classes are shown below in Polygon.h, Polygon.cpp, Point.h, and Point.cpp.

Polygon.h

```
#include "Point.h"
#include <list>

namespace IGVC{
        enum poly_type {NULL_TYPE = 0, LIDAR_TYPE, CAMC_TYPE, CAML_TYPE, CAMR_TYPE};

class Polygon
{
private:
        std::list<IGVC::Point *> points;

public:
        Polygon(void);
        ~Polygon(void);
        void AddPoint(IGVC::Point *point);
        void Draw();
        poly_type type;
```

```
};
}
```

Polygon.cpp

```
#include "Constants.h"
#include "Polygon.h"

#ifdef OPENGL
#include "GL/freeglut.h"
#endif
#include <random>

using namespace IGVC;

/****************************************
 *      Name:    Polygon [Constructor]
 *      Description: Constructs the Polygon Object
 ****************************************/
IGVC::Polygon::Polygon(void)
{
}

/****************************************
 *      Name:    Polygon [Destructor]
 *      Description: Destructs the Polygon Object
 ****************************************/
IGVC::Polygon::~Polygon(void)
{
        for(std::list<Point *>::iterator it = points.begin(); it != points.end(); ++ it){
                delete *it;
        }
        points.clear();
}

 /**********************************************************************
 *      Name:                   Add Point
 *      Output:                 point - Point object to add
 *      Description:    Adds point to polygon
 **********************************************************************/
void IGVC::Polygon::AddPoint(Point *point){
        points.push_front(point);

}

/**********************************************************************
 *      Name:    Draw
 *      Description:    Draws the object
 **********************************************************************/
void IGVC::Polygon::Draw(){
#ifdef OPENGL
        float col_r = 1, col_g = 1, col_b = 1;
        if(type == LIDAR_TYPE){
                col_r = 1, col_g = 0, col_b = 0;
        }else if(type == CAMC_TYPE){
                col_r = 1, col_g = 1, col_b = 0;
        }else if(type == CAMR_TYPE){
                col_r = 1, col_g = 0, col_b = 1;
        }else if(type == CAML_TYPE){
                col_r = 0, col_g = 1, col_b = 0;
        }
        glLineWidth(2.0f);
```

```
        //glBegin(GL_LINES);
        glBegin(GL_LINE_STRIP);
        glColor3f(col_r, col_g, col_b);
        for(std::list<Point *>::iterator it = points.begin(); it != points.end(); ++ it){
                glVertex2d((*it)->x/VISUALIZER_SCALE, (*it)->y/VISUALIZER_SCALE);
        }
        glEnd();
#endif
}
```

Point.h

```
namespace IGVC{
        class Point
        {
        public:
                Point(void);
                ~Point(void);
                float x;
                float y;
        };
}
```

Point.cpp

```
#include "Point.h"
using namespace IGVC;

Point::Point(void)
{
}


Point::~Point(void)
{
}
```

[ART]

3.7.5 *Path Finding*

Floyd's Algorithm - Outline

The path finding and planning module will be implemented using the C programming language. Shortest-path navigation will be calculated using a modified version of the Floyd-Warshall algorithm. The Floyd-Warshall algorithm (also known as Floyd's algorithm) operates in O(n^3) time, and is parallelizable across a few message-passing standards. Floyd's algorithm has many applications in computer networking, but can also be used to find the shortest path between two nodes in many other scenarios including vehicle navigation.

The basic operation of Floyd's algorithm is as follows. First, a structure of arrays is defined for containing several nodes as well as each node's position. There is one array to hold x positions and one array to hold y positions. A two-dimensional matrix is also defined to hold the distances, also known as edge weights, between each node. The matrix holding the edge weights is known as an 'adjacency' matrix. Two nodes that do not have a direct connection between them are defined to have an infinite (or very high) edge weight. Once the adjacency data has been populated with edge weights, another matrix known as the 'next' matrix is defined. The next matrix is utilized for re-routing instructions. Prior to Floyd's algorithm being run, the re-routing matrix remains populated with null values indicating no re-routes have occurred. An example of the data structures required to perform Floyd's algorithm in Figure 49 and Figure 50

```
typedef struct IGVC_node{
    float x;
    float y;
} IGVC_node;
```

Figure 49. Sample code, node structure.

```
// Define and allocate memory for adjacency matrix.
float **adjacency = malloc(sizeof(float*) * numNodes);
for(i = 0; i < numNodes; i++){
adjacency[i] = malloc(sizeof(float) * numNodes);
for(j = 0; j < numNodes; j++){
adjacency[i][j] = 0.0;
}
}

//Define and allocate memory for next matrix.
int **next = malloc(sizeof(int*) * numNodes);
for(i = 0; i < numNodes; i++){
next[i] = malloc(sizeof(int) * numNodes);
for(j = 0; j < numNodes; j++){
next[i][j] = -1;
}
}
```

Figure 50. Sample code, creation and allocation of 'adjacency' and 'next' matrices.

Now that the data structures have been constructed, Floyd's algorithm can be performed. Floyd's algorithm iterates through the entirety of the array of nodes. Floyd's algorithm selects a node as a middle point and calculates the combined edge weight between itself and two other end nodes. This new value is then compared to the existing value between the two end nodes. If the value of the combined edge weights through the middle point node is less than the existing value, the new combined edge weight value overwrites the existing value in the 'adjacency' matrix and a re-route is created in the 'next' matrix. This process is repeated until all possible re-routes have been exhausted. An example of the first two nodes being tested as middle points is shown in Figure 51.

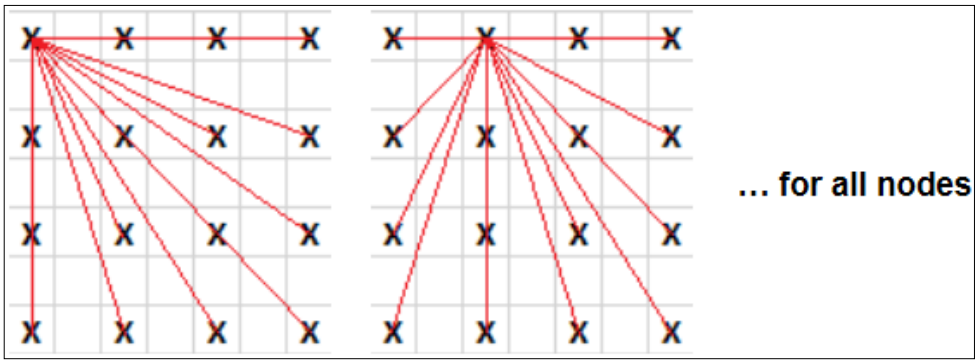


Figure 51. Concept example, first two nodes in a Floyd's algorithm system.

Floyd's Algorithm - Implementation

The implementation of Floyd's algorithm on the IGVC can be done in many ways. The primary factor to consider when creating this module is computing time. The IGVC must be able to identify, process, and react in near real time. Therefore, the processing time of Floyd's (and the rest of the navigation module) needs to be fast enough to smoothly operate a vehicle moving at a maximum of 5 miles per hour.

An ideal approach (assuming infinite processing power) would be to consider every possible node in the entire IGVC course defined by a resolution between each node that would allow for navigation around obstacles. Being able to implement such a system would require processing power sufficient to calculate the shortest route between two (or more nodes) out of potentially hundreds of thousands of nodes. For example, assume the competition course is 50x50 meters and the desired node resolution is 10 centimeters. This results in the existence of 250,000 nodes. For this scenario, the number of unique connections is 31,250,000,000. Clearly, this large number of connections cannot be implemented due to processing power restrictions of the vehicles onboard computer (Core i5-6200, 8 GB RAM, NVidia 940M), because it will not process the possible routes at a fast-enough rate to ensure smooth travel.

Noticing that the 'ideal' approach is not able to be implemented due to computational constraints, Floyd's algorithm needs to be reduced to a more manageable scale for the vehicle's onboard computer to handle. To carry out this reduction of data, a restructuring of the node arrangement in space needs to occur. Thus, it was decided that only the nearby area with respect to the vehicle needs to be mapped with high-precision to navigate around obstacles. This arrangement can be thought of as a series of node 'rings' with a constant node resolution as the radius from the vehicle increases. For example, the nearest ring of nodes at one meter contains 72 nodes for a precision of 5 degrees at the closest ring. The next ring, at 2 meters, contains 36 nodes for a precision of 10 degrees. The rings outside of two meters contain 18 nodes, for a precision of 20 degrees. The total number of rings created will be determined by the distance from the starting point to the end destination. The radius 'R' of each ring will be determined by its identifier 'n' using Equation (44).

(44)

$$R = 2^n$$

Using the previous example, the same 50x50 meter course in a worst-case scenario (start and end points are at opposite corners, distance of 70.7 meters) will result in six rings using the following method: If the identifier of a potentially newly created ring is 'n', create a new ring if (45) is satisfied.

(45)

$$2^n \leq \sqrt{((x_{end} - x_{start}) + (y_{end} - y_{start}))}$$

Also, the number of nodes 'N' in the example system can be defined using:

$$N = 72 + 36 + (18(max(n) - 2)) \quad (46)$$

For the example system of six rings, 180 nodes will be created; a substantial reduction from the 2500 nodes created in the previous system.

The next parameter of Floyd's algorithm that must be reduced to increase processing speed is the number of connections between nodes. To handle the reduction of connections, each node in a ring will be connected to its two nearest neighbors of the same ring and every node from the two adjacent rings. This will reduce the number of possible connections in the system, and thus reduce processing time. A visualization of this connection layout is provided in Figure 52, showing the all connections to the node highlighted in orange.

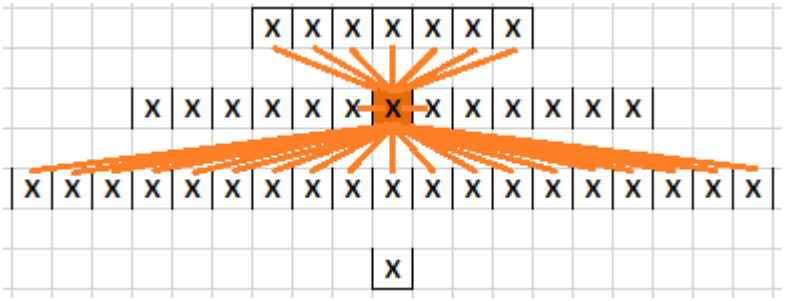


Figure 52. Concept visualization, connections to one node.

The total number of connections 'C' in the new system of mapping nodes and connections can be modeled using Equation (47).

$$C = 1(72) + 72 + 72(36) + 36 + 36(18) + 18 + (max(n) - 3)(18^2) + (max(n) - 3)(18) \quad (47)$$

Therefore, the example system (with six rings) contains 4,464 connections. This is a significant decrease in connections compared to the previous method of mapping nodes and connections which resulted in 31,250,000,000 connections.

Floyd's Algorithm – Obstacle Avoidance

To effectively compete in the Intelligent Ground Vehicle Competition, the vehicle must be able to identify and avoid obstacles in its desired path. Implementation of such feature entails creating a path collision algorithm. The path collision algorithm will test the current path by sampling several points along the path and calculating distance to each obstacle. Since an array of obstacle 'nodes' exist in an independent data structure, the GPS coordinates of each obstacle will first need to be translated to relative x and y positions in relation to the vehicle. This conversion can be performed using the haversine formula (43) using independent calculations for latitude and longitude differences. The output of the haversine formula can be used to determine a relative x and y position. Next, the Cartesian coordinate distance formula (48) can be used to compute the distance of each path sample from the obstacle node.

$$\boldsymbol{D = \sqrt{(X_2 - X_1) + (Y_2 - Y_1)}} \qquad (48)$$

Path collision testing should be performed immediately following the execution of Floyd's algorithm to prevent the vehicle from inadvertently traveling along a collision-bound path. Code for implementing path-obstacle collision detection is shown in Figure 53.

```
int obstacleContactTest(IGVC_node nodeA, IGVC_node nodeB, IGVC_node obstacle, int samples){
   // Helper function for scanObstacles().
   // This function will sample a number of points on a line between nodes
   // A and B, then calculate the distance between the points and the obstacle.
   // If this distance is less than IMPACT_RADIUS, an impact is detected.
   // Impact -> return 1
   // Clear  -> return 0

   float sampleSpacingX, sampleSpacingY, testPointX, testPointY, sampleDistance;
   sampleSpacingX = (nodeA.x - nodeB.x) / samples;
   sampleSpacingY = (nodeA.y - nodeB.y) / samples;

   int i, returnVal = 0;
   #pragma omp parallel for private(i, testPointX, testPointY, sampleDistance) num_threads(OMP_THREADS)
   for(i = 0; i < samples + 1; i++){
      testPointX = nodeB.x + (sampleSpacingX * (float)i);
      testPointY = nodeB.y + (sampleSpacingY * (float)i);

      sampleDistance = sqrt((obstacle.x - testPointX)*
                        (obstacle.x - testPointX)
                        +
                        (obstacle.y - testPointY)*
                        (obstacle.y - testPointY));
      if(sampleDistance <= IMPACT_RADIUS){
         returnVal = 1;
      }
   }
   return returnVal;
}
```

Figure 53. Sample code, path-obstacle collision detection.

The code in Figure 53 operates by tracing a path between two nodes and take a passed number of samples along that path. Each sample's distance from the obstacle node is calculated and compared to a defined impact radius. If any of the calculated distances is less than the impact radius, the function returns a value to indicate that an impact has been detected.

An example of the type of methodology which will be used to scan connections for intercept with obstacle nodes is shown in Figure 54. This code segment is used within several for-loops to iterate through the entirety of the adjacency matrix. Using the code in Figure 53, the code in Figure 54 tests each path for collision with an obstacle. If a collision is detected, the connection edge weight is set to a high value (INT_MAX) which effectively removes that connection from the system via Floyd's algorithm.

```
// Scan realistic paths AND paths not to self AND only new paths.
if((adjacency[i][j] != INT_MAX) && (i != j) && (j > i)){
    if(obstacleContactTest(nodeArray[i], nodeArray[j], nodeArray[obst], 100) == 1){
        adjacency[i][j] = INT_MAX; adjacency[j][i] = INT_MAX;
        // Set found flag, flags a necessary redo of floyd's.
        found_flag = 1;
        nodeArray[obst].isObstacle = 3;
    }
}
```

Figure 54. Sample code, path scanning for obstacles.

[CRE]

Waypoints.h, shown below, reads the waypoints from a file that are used as target locations in the pathfinding algorithm.

Waypoints.h

```
#include <queue>
#include <stdio.h>

using namespace std;

typedef struct waypoint{
float latitude;
float longitude;
} waypoint;

/****************************************
 *      Name:    readWaypoints
 *  Inputs:      filename - name of file containing GPS Waypoints
 *      Description: Reads GPS Waypoints from file
 ****************************************/
extern queue<waypoint> readWaypoints(const char* filename){
        queue<waypoint> waypoints;

        FILE *fp;
        waypoint wp  = {0,0};
```

```
        fopen_s(&fp, filename, "r");
        if(fp == NULL)
                return waypoints;
        while (!feof(fp)){
                fscanf_s(fp, "%f, %f", &wp.latitude, &wp.longitude);
                waypoints.push(wp);
        }

        return waypoints;
}
```

[ART]

3.8 Mechanical Design

3.8.1 Mechanical Analysis

The mechanical design for the IGV is a major contributor to the success of the vehicle. The design needs to be able to hold all components in a stable and weight efficient manner. There are two main parts to the Mechanical design of the vehicle; the chassis and the mounting frame for the sensors. The chassis is a modified version of a Powers Wheels Jeep Wrangler frame. The sensor mounting frame rises up from this frame as shown inFigure 22: Sensor and wiring diagram Figure 22.

The chassis is a metal case that re-enforces the weak points of the Jeeps frame. The chassis consist of two pieces of sheet metal that act as the base of the vehicle and a single metal rode extends vertically throw the center of them, holding them in place. An outer metal square wraps around the inside of the Jeeps frame and connects to the center rode at both ends and in the middle. The axels for the wheels are individual and are mounted on top of the frame. The axels hold the weight of the wheels and the gear boxes. The gear box is used to increase the torque of the motor 103:1. From the gear box a custom 3d printed mount will be implemented connecting the back wheel, the tachometer, and the front wheel. This allows for a tank drive system to be fully implemented and for the speed of the wheels to be measured.

4. Operation, Maintenance, Repair Instructions

To operate the intelligent ground vehicle, the charged batteries must first be installed into their corresponding connectors. Next, the operator needs to install the main USB cable and Ethernet cable into the jacks on the ASUS laptop. Afterwards, the user can open the computer program allowing the computer to connect to the camera to ensure that the device is operational. From there the vehicle is ready to operate. Note that, the vehicle requires a special Omnistar Software license to operate as well as the sponsored Arduino will need to be returned to John, the Asus laptop will need to be returned to Garrett, and the antenna will need to be returned to NovAtel. Thereby, the vehicle will only be in a usable state until mid-June when the license has expired and the antenna is returned.

In order to run the executable IGVC_main.exe, OpenCV 2.4.13 must be installed on the onboard PC. The dynamic-link libraries freeglut.dll and libcurl.dll (available online) and static library PF_2000.lib (which is part of this project) must be in the same directory as the executable. The executable itself can take several command line arguments when launched to specify how to connect to the various sensors. The arguments accepted by the program are listed in Table 16.

Table 16. Table of Command Line Arguments

Argument flag	Parameter
-lip	IP Address in XXX.XXX.XXX.XXX format e.g. 169.254.12.9
-cams	Number of cameras to use (the current state of the source code only allows one camera)
-cam	The camera identifying number assigned by the OS e.g. 0
-ard	The Arduino COM port e.g. COM4
-gps	The GPS COM port e.g. COM7

The program may take some time to load all the necessary components. If the program fails to launch, errors will explain what module is not connecting correctly or which libraries are missing. The program can be compiled to exclude certain modules by commenting out or uncommenting the enable lines in Constants.h if certain devices are not connected to the computer.

Maintenance on the vehicle is quite simple. First, the batteries will need to be charged up to 13.5V which takes approximately five and a half hours when charging at one amp. One amp has been chosen as a safe long term charging rate so that the batteries cells can stay alive as long as possible. Second, the camera needs to be checked for proper alignment and angle to ensure that camera and LIDAR obstacle positions are representative on the actual objects. To do so, place the vehicle in an open area the is relatively monochromatic. Next, place a white object such as a pvc pipe so that the object can be picked up by the LIDAR and the camera. The third step for maintenance is that the rubber tires will need to remain at the pressure of 30 PSI for peak performance and optimal grip capabilities. Finally, if the vehicle is to be shown, then a waxing will be required to ensure top notch quality to proudly represent the university.

As a future notice, the drive train will undergo a full inspection to ensure that there has been no damage and limited wear to the gearbox mounts that are under torsion, gears, and motors. If the drive train components are unable to withstand three weeks of testing, the team's conclusion is to replace or redesign the components in question depending on the severity. In most cases, spare parts are currently ready to substitute so that testing can resume within an hour of its potential failure.

[JPJ, ART]

5. Testing Procedures

Individual testing procedures:

Garrett Parameter tests:

John Circuit Tests:

Voltage (V)	(1/200) Rotations per Minute (No load)	Pulses Per Revolution
1	0.03 - 0.04	6 - 8
2	0.08 - 0.09	16 - 18
3	0.12 - 0.13	24 - 26
4	0.17 - 0.18	34 - 36
5	0.22	44
6	0.25 – 0.27	50 - 54
7	0.30 – 0.32	60 - 64
8	0.35 – 0.37	70 - 74
9	0.40 – 0.41	80 - 82
10	0.45	90
11	0.49 – 0.50	98 - 100
12	0.55	110

Table 17: Input Motor Voltage to Encoder Output

The first series of motor testing, seen in Table 17:, involved relating the encoder readings to the input voltage. This showed the linear relationship of pulses per revolution is 9.2 times the input voltage. Through this relationship, the encoder values can be cross referenced to the actual voltage seen at the motor terminals.

The next test procedure was on the relay system. In order to satisfy the competition's rules a wireless relay with communication of over 90 meters was installed, but the relay could only handle a three-amp maximum and the signal switching only lasted two milliseconds. The first problem was handled by installing a power solid state relay that could handle more than the maximum current. This relay closing signal required a constant high voltage to allow current to flow. The second problem was that the relay had both a normally open and closed signal that would only disconnect for two milliseconds. Because both the on and off signal needed to remain at that state a third relay that could handle to signal wires was added. The 24-volt signal is power relay drive signal, and the twelve volt signal keeps the relay switching signal constant.

Softwares' Entire system test:

The positioning of the sensors (LiDAR and cameras) were verified and updated in Constants.h. The program was recompiled and loaded onto the onboard laptop as an executable. Once the

program is launched, sensor mapping calibration can be verified by placing objects at known distances (preferably at an integer meter) from the IGV's origin point. The objects detected by the LiDAR will show up red on the visualized map while lines detected by the camera will show up yellow or green. If the objects and lines map in the visualizer to the proper locations in real space the sensor mapping calibration is good. Otherwise, the sensors may need to be moved or the constants be adjusted to fix where objects appear on the visualizer. Testing involved placing objects at various distances and in different configurations and verifying that they were detected and mapped to the correct location in the visualizer. Pictures of these tests being performed are included in Figure 55.

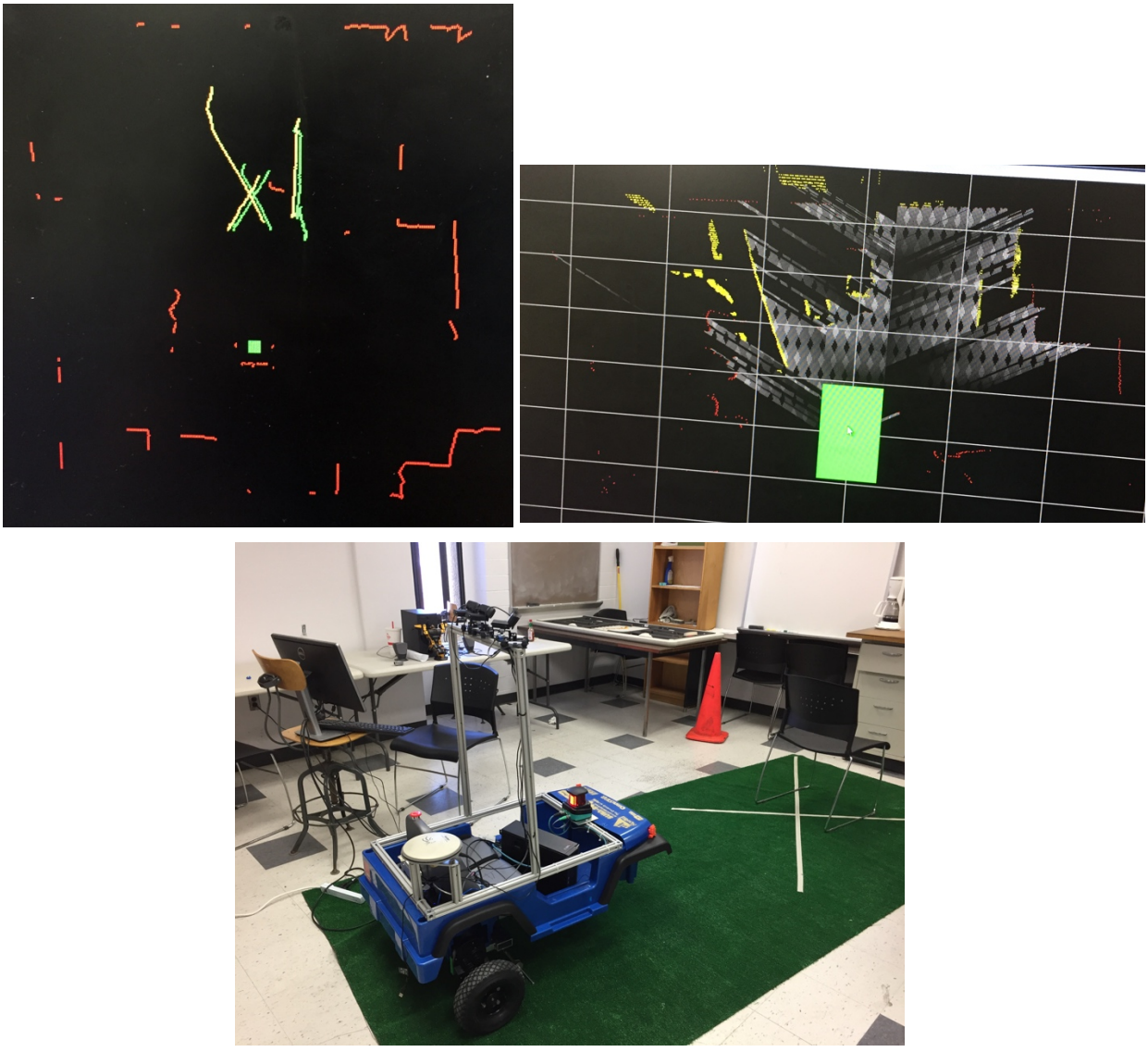


Figure 55. Images from Computer Vision Test

6. Financial Budget

Qty.	Refdes	Part Num.	Description	Vendor	Vendor Part Num.	Cost	Total
2	BT1	Q02BLMFM12_8	ExpertPower EXP1280 12V8AH Rechargeable Battery (2 pack	Amazon	Q02BLMFM12_8	34.99	
1	A2	AK-68ANHUB-BV7A	Anker USB 3.0 7-Port Hub	Amazon	AK-68ANHUB-BV7A	29.99	
3	A10,11,12	960-000764	Logitech C920 HD Pro Webcam	Amazon	960-000764	62.64	1
1	A9	1746	Triple-axis Magnetometer (Compass) Board	ADAfruit	1746,	9.95	
2	A7,8	E6A2-CS3E	YUMO Rotary Encoder 200 P/R	RobotShop	COM-10790	29.95	
1		TL296-ND	INVERTER 375W 12VDC 2OUT CIGPLUG	DigiKey	TL296-ND	66.05	
2		DK-1511-003/BL	CABLE RJ45 CAT5E W/BOOT 3' BLK	Digikey	AE10480-ND	2.19	
1		U209-005-DB25	ADAPTER USB TO SERIAL 5 FEET	Digikey	U209-005-DB25-ND	19.02	
3		3023013-02M	CABLE USB 3.0 A TO A 6.56'	Digikey	Q543-ND	8.07	
1		AK-300114-010-S	CABLE USB 3.0 TYPE-A M-M 1M	Digikey	AE10412-ND	6.94	
3	RV1	V12ZA05P	VARISTOR 12.5V 50A DISC 5MM	Digikey	F5351-ND	1.83	
10	H1-H10	745	GROMMET 0.375" RUBBER BLACK	Digikey	36-745-ND	2.27	
10	H11-H20	732	GROMMET 0.437" RUBBER BLACK	Digikey	36-732-ND	2.77	
10	H21-H30	739	GROMMET 0.250" RUBBER BLACK	Digikey	36-739-ND	2.09	
1	LED1	1092D1-12V	LED PANEL INDICATOR RED 12V	Digikey	1092D1-12V-ND	3.91	
3	Q1	TN0702N3-G	MOSFET N-CH 20V 530MA TO92-3	Digikey	1092D1-12V-ND	3.27	
1	A4	OMD10M-R2000-B23-V1V1D	Pepperl + Fuchs R2000 LiDAR	Pepperl + Fuchs	-		
1	A5		Novatel Propak V3 GPS		-	-	
1	ANT1	GPS-701-GG	Novatel GPS-701-GG-GPS antenna	Arrow Electronics		751.75	7
1	A6	A000069	Arduino MEGA 2560	Digikey		-	
1	K1	AD-SSR6M40-DC-200D	SSR 40A, 3-200Vdc load, 3-32Vdc control	Automation Direct		$40.00	
1	MP1	N/A	Toy Riding Jeep for ages 3-7	Fisher Price			
1	A1	N/A	2 Channel Motor Driver 30A each	Pololu	Roboclaw2x30A	124.95	1
1	SW1		NC N/C Emergency Stop Switch Push Button Mushroom Push	Ebay		1.22	

			Button				
1	F1	ATO30	ATO blade fuse 30A	UofA		1	
1		IHWO12	12AWG fuse holder waterproof	UofA		1.25	
4	F2,3,4,5	ATO05	ATO blade fuse 5A	UofA		0.25	
4		IHWO18	18AWG fuse holder waterproof	UofA		1.25	
1		90272A106	100 Screws 4-40 Thread Size, 1/4" Long	McMaster		1.46	
20		91780A529	Hex Standoff 4-40 3/8”	McMaster		.48	
1		90272A108	100 Screws 4-40 3/8”	McMaster		1.60	
10		93505A103	10 M-F Standoff 4-40 3/16	McMaster		.36	
1		7598A904	Foam Tape, 5yds	McMaster		1.90	
1		94985K614	10’ Velcro hook	McMaster		8.59	
1		94985K655	10’ Velcro loop	McMaster		8.59	
1		90480A005	100 hex nuts 4-40	McMaster		.87	
1		71295K62	100 cable ties 4”	McMaster		2.20	
1		71295K69	100 cable ties 7.5”	McMaster		3.38	
2		76255A14	White tape roll 2” width, 110yd	McMaster		5.68	
						Total	**$1,5**

7. Project Schedules

7.1 Final Design Schedule

The Grant Chart that was used by the senior design team #16 to design the IGVC in fall of 2016.

Name	Begin date	End date	Resources
SDP1 fall2016	9/1/16	12/13/16	
Project Design	9/1/16	12/13/16	
Preliminary Design Report	9/1/16	9/15/16	
Problem Statement	9/1/16	9/15/16	Garrett Chonko,Austin Tyler,Johnathan Johenning,Chistopher Estock ,Allen Gilleland
Need	9/1/16	9/8/16	
Objective	9/1/16	9/8/16	
Background	9/1/16	9/8/16	
Marketing Requirements	9/1/16	9/8/16	
Objective Tree	9/1/16	9/15/16	
Preliminary Design Gantt Chart	9/1/16	9/15/16	
Block Diagrams Level 0 w/ FR tables	9/1/16	9/15/16	
Hardware modules	9/1/16	9/15/16	Garrett Chonko,Johnathan Johenning
Software modules	9/1/16	9/15/16	Austin Tyler,Chistopher Estock
Preliminary Design Presentation 3:20PM	9/15/16	9/15/16	Garrett Chonko,Austin Tyler,Johnathan Johenning,Chistopher Estock ,Allen Gilleland
Midterm Report	9/19/16	10/20/16	
Design Requirements Specification	9/19/16	10/2/16	
Midterm Design Gantt Chart	9/19/16	10/2/16	
Design Calculations (Midterm)	10/2/16	10/20/16	
Electrical Calculations	10/2/16	10/20/16	
Computing Calc	10/2/16	10/20/16	Austin Tyler,Chistopher Estock
Camra Calc	10/2/16	10/20/16	Garrett Chonko,Allen Gilleland
Motor Calc	10/2/16	10/20/16	Garrett Chonko,Johnathan Johenning
Battery Calc	10/2/16	10/20/16	Johnathan Johenning
Department Budget Presentation	10/3/16	10/14/16	
Sensor Cost Estimations	10/3/16	10/14/16	Allen Gilleland
Motor/Power Cost Estimations	10/3/16	10/14/16	Johnathan Johenning
HardWare Cost Estimations	10/3/16	10/14/16	Garrett Chonko
Software Cost Estimations	10/3/16	10/14/16	Austin Tyler,Chistopher Estock
Travel Cost Estimations	10/3/16	10/14/16	Johnathan Johenning
PowerPoint Presentation (15min)	10/3/16	10/14/16	Garrett Chonko,Austin Tyler,Johnathan Johenning,Chistopher Estock ,Allen Gilleland
Department Presentation 3:20	10/14/16	10/14/16	Garrett Chonko,Austin Tyler,Johnathan Johenning,Chistopher Estock ,Allen Gilleland
Block Diagrams Level 1 w/ FR tables & ToO	10/3/16	10/9/16	
Sensor Data Modules	10/3/16	10/9/16	Austin Tyler,Allen Gilleland
Software Modules	10/3/16	10/9/16	Garrett Chonko,Austin Tyler,Chistopher Estock
Power/Controls Modules	10/3/16	10/9/16	Garrett Chonko,Johnathan Johenning
Block Diagrams Level 2 w/ FR tables & ToO	10/10/16	10/20/16	
Sensor Data Modules	10/10/16	10/20/16	Allen Gilleland
Software Modules	10/10/16	10/20/16	Austin Tyler,Chistopher Estock
Power/Controls Modules	10/10/16	10/20/16	Garrett Chonko,Johnathan Johenning
Midterm Design Presentations 3:20-5:00PM Part 1	10/20/16	10/20/16	Garrett Chonko,Austin Tyler,Johnathan Johenning,Chistopher Estock ,Allen Gilleland
Project Poster	10/3/16	11/3/16	Garrett Chonko,Austin Tyler,Johnathan Johenning,Chistopher Estock ,Allen Gilleland
Parts Ideas list	10/3/16	11/21/16	Garrett Chonko,Austin Tyler,Johnathan Johenning,Chistopher Estock ,Allen Gilleland
Final Design Report	10/21/16	12/4/16	Garrett Chonko,Austin Tyler,Johnathan Johenning,Chistopher Estock ,Allen Gilleland
Abstract	10/21/16	12/4/16	Garrett Chonko,Johnathan Johenning,Chistopher Estock ,Allen Gilleland
Software Design	10/21/16	12/4/16	
Modules 1...n	10/21/16	12/4/16	
Psuedo Code	10/21/16	12/4/16	Austin Tyler,Chistopher Estock
Design Calculations	10/21/16	12/4/16	Garrett Chonko,Austin Tyler,Chistopher Estock
Software Calc	10/21/16	12/4/16	Austin Tyler,Chistopher Estock
Equation Development	10/21/16	12/4/16	Garrett Chonko,Austin Tyler,Chistopher Estock
Algorithem Development	10/21/16	12/4/16	Austin Tyler,Chistopher Estock
Hardware Design	10/21/16	12/4/16	
Modules 1...n	10/21/16	12/4/16	
Simulations	10/21/16	12/4/16	Garrett Chonko,Austin Tyler,Chistopher Estock ,Allen Gilleland
Design Calculations	10/21/16	12/4/16	
Control Calc	10/21/16	12/4/16	Garrett Chonko,Austin Tyler
Power Consumtion Calc	10/21/16	12/4/16	Johnathan Johenning,Allen Gilleland
Motor/Torqe Calc	10/21/16	12/4/16	Garrett Chonko,Johnathan Johenning
Sensor Calc	10/21/16	12/4/16	Allen Gilleland
Parts Request Form	10/21/16	12/4/16	Garrett Chonko
Budget (Estimated)	10/21/16	12/4/16	Garrett Chonko,Austin Tyler,Johnathan Johenning,Chistopher Estock ,Allen Gilleland
Implementation Gantt Chart	10/21/16	12/4/16	Garrett Chonko,Austin Tyler,Johnathan Johenning,Allen Gilleland
Conclusions and Recommendations	10/21/16	12/4/16	Garrett Chonko,Austin Tyler,Johnathan Johenning,Chistopher Estock ,Allen Gilleland
Final Design Presentation Part 3 5:15PM-7:15PM	12/14/16	12/14/16	Garrett Chonko,Austin Tyler,Johnathan Johenning,Chistopher Estock ,Allen Gilleland

7.2 Proposed Implementation Schedule

Gantt Chart that will be implemented to build the 2017 SDP IGV.

Name	Begin date	End date	Resources
SDPII Implementation 2017	12/25/16	5/11/17	
Revise Gantt Chart	1/17/17	1/24/17	Garrett Chonko
Implement Project Design	12/25/16	5/11/17	Garrett Chonko,Austin Tyler,Johnathan Johenning,Christopher Estock,Allen Gilleland,Mark Naim,Beong Ku...
Hardware Implementation	12/25/16	4/6/17	Garrett Chonko,Johnathan Johenning,Christopher Estock,Allen Gilleland,Jacob Lukachinsky,Mark Naim,Be...
Work with ME's on Vehcle build	12/25/16	4/6/17	Garrett Chonko,Austin Tyler,Johnathan Johenning,Christopher Estock,Allen Gilleland,Jacob Lukachinsky,M...
ME's Design	12/25/16	1/30/17	Garrett Chonko,Jacob Lukachinsky,Mark Naim,Beong Kushington,Johnathan Johenning,Allen Gilleland
ME's Build	1/25/17	4/6/17	Garrett Chonko,Jacob Lukachinsky,Mark Naim,Beong Kushington,Johnathan Johenning,Allen Gilleland
Plan Layout of Devices	12/25/16	3/10/17	Garrett Chonko,Johnathan Johenning,Allen Gilleland,Jacob Lukachinsky,Mark Naim,Beong Kushington
Resive and test device	12/25/16	3/31/17	Garrett Chonko,Austin Tyler,Allen Gilleland
Implement Hardware Layout	3/11/17	3/17/17	Johnathan Johenning,Allen Gilleland
Test Hardware	3/18/17	3/31/17	Garrett Chonko,Johnathan Johenning,Allen Gilleland
Revise Hardware	3/18/17	3/31/17	Garrett Chonko,Johnathan Johenning,Allen Gilleland
MIDTERM: Demonstrate Hardware	4/1/17	4/5/17	Garrett Chonko,Johnathan Johenning,Allen Gilleland
SDC & FA Hardware Approval	4/6/17	4/6/17	Garrett Chonko,Johnathan Johenning,Allen Gilleland
Software Implementation	1/17/17	3/10/17	Garrett Chonko,Austin Tyler,Christopher Estock,Allen Gilleland
Create Object detection software	1/17/17	2/12/17	Christopher Estock,Garrett Chonko
Create Controller Software	1/17/17	2/12/17	Garrett Chonko
Creat maping/ pathfinding	1/17/17	2/12/17	Austin Tyler,Christopher Estock
Test Software	2/13/17	3/5/17	Garrett Chonko,Austin Tyler,Christopher Estock
Revise Software	2/13/17	3/5/17	Garrett Chonko,Austin Tyler,Christopher Estock
MIDTERM: Demonstrate Software	3/6/17	3/10/17	Austin Tyler,Christopher Estock
SDC & FA Software Approval	3/11/17	3/11/17	Austin Tyler,Christopher Estock
System Integration	4/6/17	5/11/17	Garrett Chonko,Austin Tyler,Johnathan Johenning,Christopher Estock,Allen Gilleland,Jacob Lukachinsky,Be...
Assemble Complete System	4/6/17	4/20/17	Garrett Chonko,Austin Tyler,Johnathan Johenning,Christopher Estock,Allen Gilleland,Jacob Lukachinsky,M...
Test Complete System	4/21/17	5/11/17	Garrett Chonko,Austin Tyler,Johnathan Johenning,Christopher Estock,Allen Gilleland,Jacob Lukachinsky,M...
Demonstration of Complete System	5/12/17	5/12/17	Garrett Chonko,Austin Tyler,Johnathan Johenning,Christopher Estock,Allen Gilleland,Jacob Lukachinsky,M...
Develop Final Report	4/21/17	5/11/17	Garrett Chonko,Austin Tyler,Johnathan Johenning,Christopher Estock,Allen Gilleland
Write Final Report	4/21/17	5/11/17	Garrett Chonko,Austin Tyler,Johnathan Johenning,Christopher Estock,Allen Gilleland
Revise Complete System	4/21/17	5/11/17	Garrett Chonko,Austin Tyler,Johnathan Johenning,Christopher Estock,Allen Gilleland
Submit Final Report	5/12/17	5/12/17	Garrett Chonko,Austin Tyler,Johnathan Johenning,Christopher Estock,Allen Gilleland
Spring Recess	3/27/17	4/2/17	
Project Demonstration and Presentation	4/24/17	4/24/17	Garrett Chonko,Austin Tyler,Johnathan Johenning,Christopher Estock,Allen Gilleland

[GWC]

7.3 Actual Implementation Schedule

Untitled Gantt Project Apr 27, 2017

2

Tasks

Name	Begin date	End date	Resources
SDPII Implementation 2017	1/17/17	4/30/21	
Revise Gantt Chart	4/27/17	4/27/17	Austin Tyler
Implement Project Design	1/17/17	5/13/17	
Hardware Implementation	1/17/17	4/24/17	
Work with MEs on Vehicle Build	1/17/17	4/22/17	
Mechanical Design	1/17/17	4/15/17	Garrett Chonko, Johnathan Johenning, Jacob Lukachisky
Mechanical Build	3/13/17	4/22/17	Garrett Chonko, Austin Tyler, Chris Estock, Allen Gilleland, Johnathan Johenning
Plan Layout of Devices	1/17/17	3/10/17	Austin Tyler, Chris Estock, Allen Gilleland, Johnathan Johenning
Receive and Test Sensors	1/17/17	1/30/17	Garrett Chonko, Austin Tyler, Chris Estock, Allen Gilleland
Motor Testing and Modeling	1/17/17	4/17/17	Garrett Chonko
MIDTERM: Demonstrate Hardware	3/15/17	3/15/17	Garrett Chonko, Austin Tyler, Chris Estock, Allen Gilleland, Johnathan Johenning
SDC & FA Hardware Approval	3/15/17	3/15/17	Garrett Chonko, Allen Gilleland, Johnathan Johenning
Assemble Hardware	4/17/17	4/23/17	Allen Gilleland, Johnathan Johenning
Test Hardware	4/22/17	4/24/17	Allen Gilleland, Johnathan Johenning
Software Implementation	1/17/17	4/24/17	
Develop Software	1/17/17	4/24/17	
LIDAR Interfacing + Mapping	1/17/17	2/16/17	Austin Tyler
Line Detection + Mapping	2/11/17	4/5/17	Austin Tyler, Chris Estock
GPS Interfacing	4/3/17	4/13/17	Chris Estock
Compass Interfacing	2/20/17	2/27/17	Allen Gilleland
Controls Modeling and Programming	3/15/17	4/24/17	Garrett Chonko
Arduino Interfacing	4/17/17	4/24/17	Austin Tyler, Chris Estock, Johnathan Johenning
Pathfinding	4/23/17	4/23/17	Austin Tyler
System Integration	4/6/17	4/22/17	Austin Tyler
Test Software	4/22/17	4/24/17	Austin Tyler, Chris Estock, Johnathan Johenning
Revise Software	4/22/17	4/24/17	Austin Tyler, Chris Estock, Johnathan Johenning
MIDTERM: Demonstrate Software	3/15/17	3/15/17	Austin Tyler, Chris Estock
SDC & FA Software Approval	3/16/17	3/16/17	Austin Tyler, Chris Estock
System Integration	4/20/17	5/13/17	
Assemble Complete System	4/20/17	4/22/17	Garrett Chonko, Austin Tyler, Chris Estock, Allen Gilleland, Johnathan Johenning
Test Complete System	4/23/17	5/13/17	Austin Tyler, Chris Estock, Johnathan Johenning
Revise Complete System	4/23/17	5/13/17	Garrett Chonko, Austin Tyler, Chris Estock, Allen Gilleland, Johnathan Johenning
Demonstration of Complete System	5/14/17	5/14/17	Garrett Chonko, Austin Tyler, Chris Estock, Johnathan Johenning
Develop Final Report	1/17/17	4/30/21	
Write Final Report	1/17/17	4/30/21	Garrett Chonko, Austin Tyler, Chris Estock, Allen Gilleland, Johnathan Johenning
Submit Final Report	5/1/21	5/1/21	Garrett Chonko, Austin Tyler, Chris Estock, Allen Gilleland, Johnathan Johenning
Spring Recess	3/27/17	4/2/17	
Project Demonstration and Presentation	4/24/17	4/24/17	Garrett Chonko, Austin Tyler, Chris Estock, Allen Gilleland, Johnathan Johenning

8. Design Team Information

Garrett Chonko, Electrical Engineer

Christopher Estock, Computer Engineer

Allen Gilleland, Electrical Engineer

Johnathan Johenning, Electrical Engineer

Austin Tyler, Electrical and Computer Engineer

9. Conclusions and Recommendations

The sensors utilized on the IGV will achieve the three goals of course mapping, object detection and speed monitoring. The implementation of these processes should be seamless from a power and functionality standpoint. Implementation of the microprocessor and its components, the digital compass and tachometer may run into calibration and communication issues on the processing side but should be a trial and error type of fix. Communication between the GPS software and hardware must be achieved and tested prior to going to competition to ensure the vehicle can find a geological waypoint.

10. References

[1] D. J. Bruemmer and D. A. Few, "Autonomous navigation system and method," U.S. Patent 7 587 260, Sept. 8, 2009.

[2] D. A. Pomerleau, "System and method for estimating lateral position," U.S. Patent 5 675 489, Oct. 7, 1997.

[3] P. G. Trepagnier, et al., "Navigation and control system for autonomous vehicles," U.S. Patent 8 050 863, Nov. 1, 2011.

[4] M. L. Nelson, "A Design Pattern for Autonomous Vehicle Software Control Architectures," in *Computer Software and Applications Conf.* Phoenix, AZ, 1999, pp. 172-177.

[5] M. Huang et al., "Research on Autonomous Driving Control Method of Intelligent Vehicle Based on Vision Navigation," in *2010 Int. Conf. on Computational Intelligence and Software Engineering*, Wuhan, 2010, pp. 1-7.

[6] B. Dumitrascu et al., "Laser-based Obstacle Avoidance Algorithm for Four Driving/Steering Wheels Autonomous Vehicle," in *2013 17th Int. Conf. on System Theory, Control and Computing*, Sinaia, 2013, pp. 187-192.

[7] G. Close, J. Cutright, K. Gee, and G. Rocco, "Intelligent Ground Vehicle Competition," unpublished.

[8] *HSL and HSV* [Online]. Available: https://en.wikipedia.org/wiki/HSL_and_HSV.

[9] SharkD, RGB_color_solid_cube.png and HSV_color_solid_cylinder.png. 2010.

[10] Peter Villars Sportsground Maintenance, school-white-line-marking.jpg, 2013.

[11] K. Raghupathy, "Curve Tracing and Curve Detection in Images," Dept. Elect. Eng., Cornell Univ., Ithaca, NY, Aug. 2004.

[12] C. Veness. (2016). *Calculate Distance, Bearing and More Between Latitude/Longitude Points* [Online]. Available: http://www.movable-type.co.uk/scripts/latlong.html

[ART, CRE]

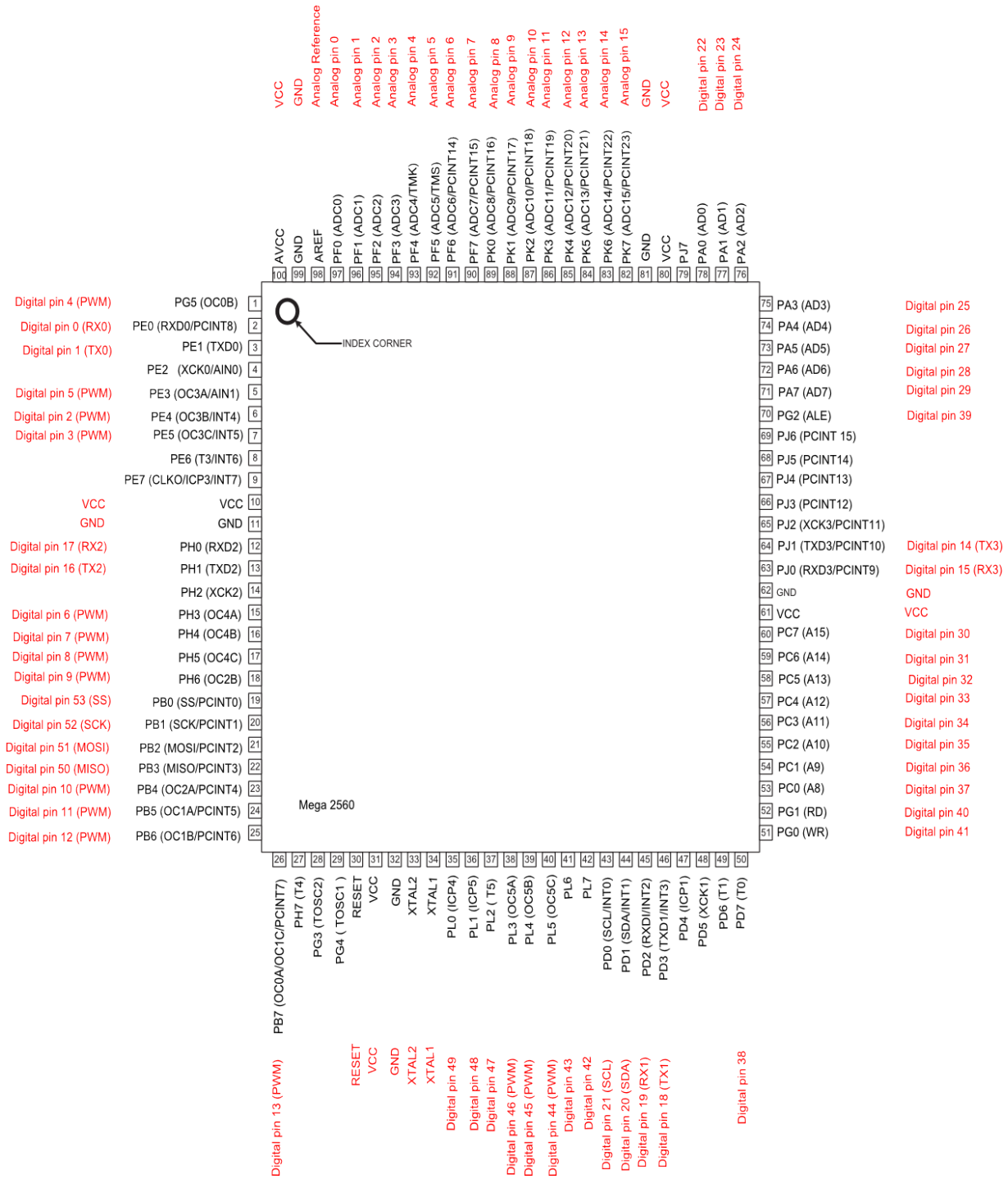
11. Appendix

Pepperl + Fuchs R2000:

General Technical Data	
Detection/ measuring range	0.1 ... 30 m to object; 0.1 ... 100 m to reflector
Light type	Red laser light, infrared (IR) laser light; laser class 1
Light spot diameter	< 15 mm at 10 m; 25 mm x 105 mm at 10 m (IR)
Rotational speed/ detection rate	R2000 UHD 10–50 Hz 600–3,000 RPM R2000 HD 10–50 Hz 600–3,000 RPM R2000 Detection 10–30 Hz 600–1,800 RPM
Dimension (W x H x D)	106 x 116.5 x 106 mm
Operating voltage	10 V ... 30 V

Accessories	Description
PACT*ware*	FDT framework
Device Type Manager	DTM R2000 Series
MH-R2000	Mounting aid
V1SD-G-ABG-PG9	Field-attachable male connector
V1-G-5M-PUR	Female connector
V1SD-G-2M-PUR-ABG-V45-G	Connection cable 2 m
V1SD-G-5M-PUR-ABG-V45-G	Connection cable 5 m
V17-G-2M-PUR	Connection cable 2 m (R2000 Detection)
V17-G-5M-PUR	Connection cable 5 m (R2000 Detection)

Arduino MEGA 2560:



Novatel Propak V3:

Performance[1]

Channel Configuration
72 Channels
Signal Tracking

GPS	14 L1, 14 L2, 6 L5
GLONASS	12 L1, 12 L2
SBAS	12 L1, 12 L2
L-band	1

Horizontal Position Accuracy (RMS)

Single Point L1	1.5 m
Single Point L1/L2	1.2 m
SBAS[2]	0.6 m
DGPS	0.4 m
OmniSTAR[2]	
VBS	0.6 m
XP	0.15 m
HP	0.1 m
RT-20®[3]	0.2 m
RT-2™	1 cm+1 ppm

Measurement Precision

	GPS	GLO
L1 C/A Code	4 cm	15 cm
L1 Carrier Phase	0.5 mm	1.5 mm
L2 P(Y) Code	8 cm	8 cm
L2 Carrier Phase	1.0 mm	1.5 mm

Data Rate[4]

Measurements	50 Hz
Position	50 Hz
OmniSTAR HP/XP	20 Hz

Time to First Fix

Cold Start[5]	60 s
Hot Start[6]	35 s

Signal Reacquisition

L1	0.5 s (typical)
L2	1.0 s (typical)

Time Accuracy[7] 20 ns RMS

Velocity Accuracy 0.03 m/s RMS

Dynamics

Velocity[8]	515 m/s
Vibration	4 G (sustained tracking)

Physical and Electrical

Dimensions 185 x 160 x 71 mm

Weight 1.0 kg

Power

Input Voltage[9]	+9 to +18 VDC
Power Consumption	2.8 W (typical)[10]

Antenna Port Power Output

Output Voltage	+5 VDC
Maximum Current	100 mA

Connectors

Power	4-pin LEMO
Antenna Input	TNC female
External Oscillator	BNC female
COM1	DB9 male
COM2	DB9 male
AUX (COM3)	DB9 male
I/O	DB9 female

Communication Ports

2 RS-232 or RS-422
1 RS-232 serial port
1 USB 1.1 port

Environmental

Temperature

Operating	-40°C to +75°C
Storage	-45°C to +95°C

Humidity 95% non-condensing

Waterproof IEC 60529 IPX7

Dust IEC 60529 IP6X

Vibration (operating)

Random	MIL-STD-810F, 514.5
Sinusoidal	SAE J1211 4.7

Shock (non-operating) IEC 68-2-27 Ea

Compliance FCC, CE

Included Accessories

- Automotive 12 VDC power adapter with 3A slow-blow fuse
- Mounting bracket
- Straight serial cable
- Null-modem cable
- I/O interface cable
- USB cable

Optional Accessories

- GPS-700 series antennas
- ANT series antennas
- RF Cables—5, 10 and 30 m lengths
- AC adapters—International and North American

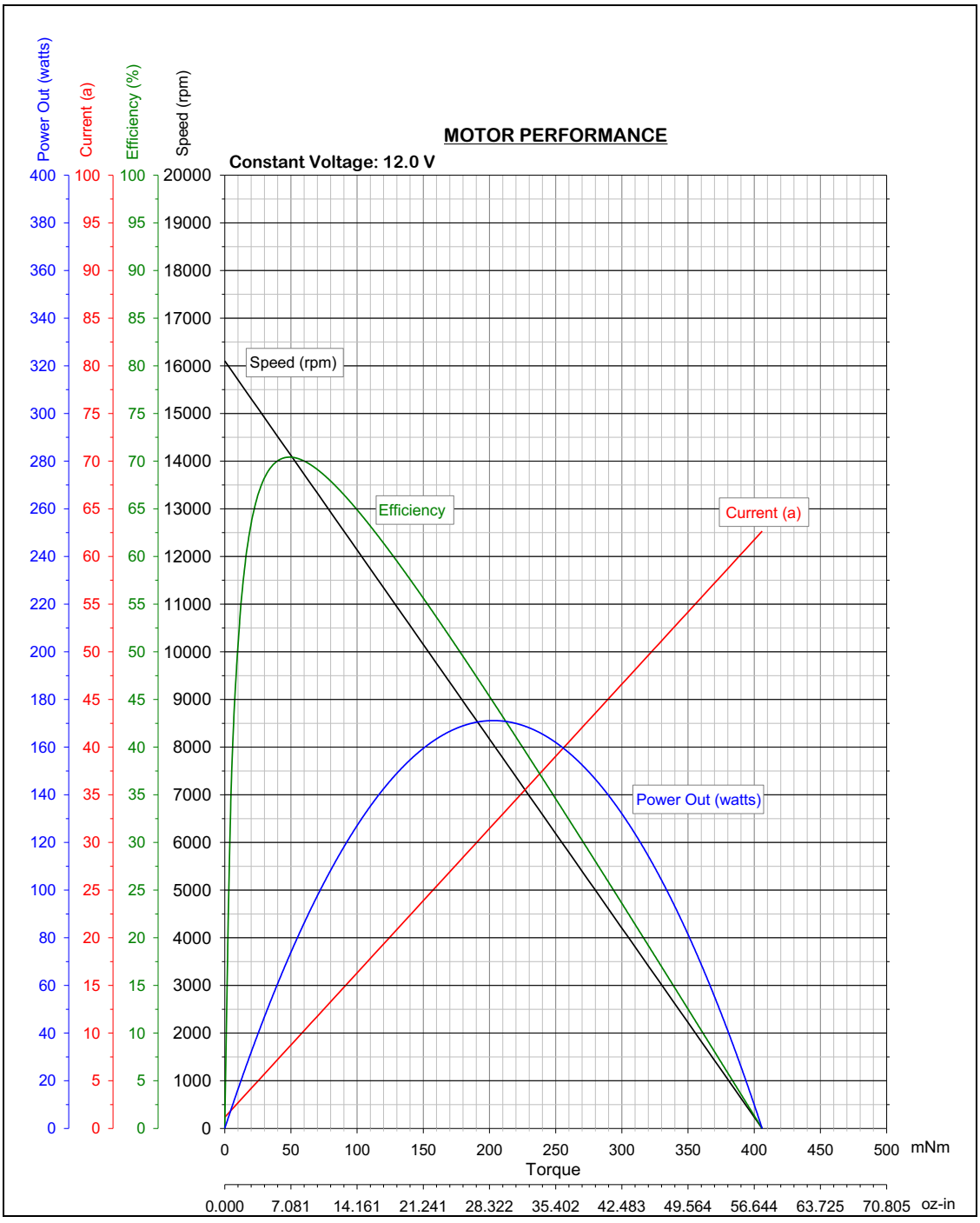
Features

- Multiple software models, including L1 GPS or GLONASS, L1/L2 GPS + GLONASS, and carrier-phase positioning
- Auxiliary strobe signals, including a configurable PPS output and two mark inputs
- Field-upgradeable firmware
- Supports RTCM SC-104 version 3.0, CMR version 3.0, CMR+, NMEA 0183 version 3.01, and RTCA DO-217 message types
- Application Programming Interface (API)

Firmware Options

- RT-20
- ALIGN®
- GL1DE®
- OmniSTAR HP, XP, VBS, G2
- L5 signal tracking
- Pseudo Range/Delta-Phase (PDP) Positioning

Power Wheels Motor 9015 Motor Performance Curve:



As stated from the manufacturer, there are two motors on the vehicle each of which are attached to a gear box with a 103:1 ratio.