Intelligent Ground Vehicle Competition

Robot Report

Mark Landergan

Aidan Cookson

Sean O’Neil

DATE: April 25, 2019

REPORT SUBMITTED TO:

Professor Xinming Huang

Professor William Michalson

Abstract

The goal of this project was to design and implement an autonomous robot capable of competing in the 2019 Intelligent Ground Vehicle Competition. The competition outlined several technical requirements including: lane detection and holding, obstacle avoidance, path planning, and GPS navigation. To achieve these tasks a Husky A100 platform was retrofitted with a more powerful computer and integrated with sensors such as a LIDAR, Camera, Satellite Compass, IMU, and GPS. A software architecture was developed based on Robot Operating System to reliably perceive the course lanes and obstacles, localize the robot through sensor fusion, and guide it through each waypoint.

Table of Contents

Abstract 3

1 Introduction 8

2 Background 10

2.1 - Intelligent Ground Vehicle Competition (IGVC) 10

2.2 - Auto Nav Qualification Requirements and Competition Tasks 11

2.2 - Previous IGVC Robots 13

2.2.1 - Georgia Institute of Technology - JAYMI (2017) 13

2.2.2 - Indian Institute of Technology - Bombay (2017) 14

3 System Architecture 16

3.1 - Hardware System Overview 16

3.2 - Software System Overview 16

4 Robot Development 18

4.1 - Research 18

4.1.1 - Robot Platform 18

4.2 - System Components 19

4.2.1 - Main Computer 19

4.2.2 - Nucleo Microcontroller 19

4.2.3 - Secondary Power Supply 20

4.2.4 - GPS 21

4.2.5 - Satellite Compass 22

4.2.6 - IMU 22

4.2.7 - Encoders and Motors 22

4.2.8 - Mono USB3.0 Camera 23

4.2.9 - Light Detection and Ranging (LIDAR) 23

4.2.10 - Sensor Tower 24

5 Software Development 27

5.1 Jetson TX2 Software 27

5.2 Sensor Communication 28

5.2.1 Nucleo Microcontroller 28

5.2.2 Ublox GPS 30

5.2.3 Simrad Satellite Compass 31

5.2.4 LORD Microstrain IMU 33
5.2.5 Hokuyo LIDAR 33
5.2.6 Point Grey Chameleon3 35
5.3 - Sensor Transformations 36
5.4 Software Stack 36
5.5 Lane and Pothole Detection 37
5.6 Navigation 39
5.7 GPS Waypoint Navigation 40

6 Software Validation 41
6.1 Odometry Testing 41
6.2 Course Test 42

7 Future Work 42

8 Software API 46
8.2 Wolfgang Bringup 46
8.3 GPS Navigation Node 47
8.4 Lane Following Node 48

Bibliography 49

Appendix 50
Lane Detection Source Code 50
Lane Holding Source Code 55
GPS Waypoint Node 59

List of Figures

Figure	Page
Figure 1: Example Auto-Navigation Course Map	12
Figure 2: System Hardware Block Diagram	16
Figure 3: ROS Node Software Hierarchy	17
Figure 4: Husky A100 Mobile Robotics Platform	18
Figure 5: Estimated Peripheral Power Consumption and Battery Operation Time	21
Figure 6: Hokuyo UXM-30LAH-EWA Scanning Laser Rangefinder	23
Figure 7: Sensor Tower	25
Figure 8: Udev Rules for Sensors	27
Figure 9: PlatformIO configuration file	28
Figure 10: ROS Encoders message	29
Figure 11. rostopic echo of *sensor_msgs/NavSatFix* message example	30
Figure 12: NMEA Network for Satellite Compass	32
Figure 13: Satellite Compass NMEA Messages	32
Figure 14: LIDAR ethernet rules	34
Figure 15: 2D laser scan data in RVIZ	34
Figure 16: Undistorted Chameleon3 Image	35

Figure 17: Sensor Transformations 36

Figure 18: Lane Detection Before Piecewise Linear Fit 38

Figure 19: Sensor Coordinate Transformations 39

Figure 20: Triangle Odometry Test 41

Figure 21: System Software Block Diagram 43

Figure 22: Odometry Pipeline 45

Figure 23: Wolfgang Bringup Launch File 48

1 Introduction

The goal of this project was to design and implement an autonomous robot capable of competing in the 2019 Intelligent Ground Vehicle Competition. The competition outlined several technical requirements including: lane detection and holding, obstacle avoidance, path planning, and GPS navigation. To achieve these tasks, a Husky A100 platform was retrofitted with a more powerful computer and integrated with sensors such as a LIDAR, Camera, Satellite Compass, IMU, and GPS. A software architecture was developed based on Robot Operating System to reliably perceive the course lanes and obstacles, localize the robot through sensor fusion, and guide it through each waypoint.

Every year in the United States, there is an average of 6 million reported car accidents, resulting in more than 90 deaths per day [1]. Unfortunately, many of these accidents are caused by preventable actions such as distracted driving. While educating people on safe driving habits is important, developing safer technology to aid drivers is arguably more important in preventing car accidents and injuries.

Within the past decade, technological advances such as higher precision cameras, more accurate GPSs, and exponentially increasing computational power have allowed the field of autonomous vehicles to grow from a niche market to a global technological race. Despite the recent leaps in progress, autonomous vehicles have a long way to go before they can be considered safe in the mainstream automotive market and become integrated with society.

One of the largest issues with current technologies is sensor reliability. When the sensors that autonomous vehicles rely on unexpectedly fail, the result can be catastrophic, causing haphazard driving or accidents. If these vehicles are to transport passengers in an everyday setting, such as to and from work, this is an unallowable risk. To mediate this risk, sensor redundancy and fault tolerance is required. This means having an intelligent system that detects faulty sensor data and changes its behavior accordingly is imperative to safe autonomous travel.

Many extremely complex problems are best solved by attacking individual aspects of the problem and combining the solutions to form a complete model of the problem. Autonomous vehicles, especially those exposed to unconstrained outdoor environments, is no exception. Because of this, it is common to develop and test self-driving technology on smaller models and scale them up. Student competitions such as the Intelligent Ground Vehicle Competition (IGVC) give aspiring autonomous vehicle engineers the opportunity to conduct research on some of the issues currently at the forefront of the field with little financial and human risk.

The goal of this project is to design, develop, and program an autonomous vehicle capable of competing in IGVC. While competing in IGVC is not the principal goal of the project, we aim to build a robot that accomplishes the tasks laid out in the competition while meeting the technical criteria outlined in the competition rules.

2 Background

2.1 - Intelligent Ground Vehicle Competition (IGVC)

The Intelligent Ground Vehicle Competition (IGVC) is an international robotics competition for undergraduate and graduate students which takes place annually at Oakland University in Rochester, Michigan. “The IGVC offers a design experience that is at the very cutting edge of engineering and education. It is multidisciplinary, theory-based, hands-on, team implemented, outcome assessed, and based on product realization.” [2]

There are two main competitions within the IGVC event: Auto-Navigation and Self-Drive. The Auto-Navigation challenge is concerned with testing a fully autonomous unmanned ground robotic vehicle in a closed grass course containing many obstacles similar to obstacles that could be found during military usage. The Self-Driving challenge involves a much larger vehicle capable of fitting two people with the goal of traveling down a closed road while maintaining lane position and avoiding hazards. Aside from the scale of the robots, both competitions require a similar technical toolkit comprised of: sensor fusion, object detection, path planning, and navigation.

A third competition, which teams from either of the previous challenges may choose to compete in, is the design competition. While the designs reviewed for this competition are from the auto-navigation or self-driving challenges, the design score is independent from vehicle performance. An oral presentation is given to a panel of design judges, and a written report is submitted. The judges are especially interested in innovative hardware or software that is new to the competition. We aimed to compete in the Auto-Navigation challenge, as the resources and scope were not available for our team to compete in the Self-Driving challenge.

2.2 - Auto Nav Qualification Requirements and Competition Tasks

The objective of the IGVC Auto-Navigation competition is to design a fully autonomous ground vehicle to navigate an outdoor obstacle course within a certain time limit while maintaining an average speed between one and five miles per hour. It must remain within the lane, and avoid all obstacles on the course. The competition is judged based on adjusted completion time, calculated by adjusting the raw completion time by factors determined by rule violations and uncompleted tasks. The robot has six minutes to complete the challenge.

The course takes place in a grassy field that is approximately 100 feet wide and 200 feet deep. The course length is approximately 600 feet. The lanes are between 10 and 20 feet wide, and are painted on the grass. Obstacles consist of natural objects such as trees and shrubs, in addition to unnatural objects such as construction drums, light posts, potholes, and street signs. The obstacles can be any color. Each team is provided with two sets of GPS coordinates. One set is the entrance and exit of the course. The other set is two waypoints in “no man’s land”, an unmarked area full of obstacles which the robot must navigate through. An example course map is shown in Figure 1 below [3].

The Auto-Navigation competition has strict qualifying technical requirements that each team must meet to ensure a safe and fair competition. The 2019 requirements and tasks are outlined below:

Qualification Technical Requirements

1. Design: Must be a ground vehicle
2. Dimensions:
    a. Length: Between three and seven feet
    b. Width: Between two and four feet
    c. Height: Shorter than six feet

3. Average Speed: Between one and five mph
    a. Over one mph for the first 44 feet of the course
4. Wireless Emergency Stop:
    a. Effective at a minimum of 100 feet
    b. Hardware-based and on a separate communication channel from RC control
5. Mechanical Emergency Stop:
    a. Push to stop, red in color, minimum of one inch diameter
    b. Mounted between two and four feet off the ground
6. Safety Light:
    a. Solid when vehicle is powered on
    b. Flashing when in autonomous control mode

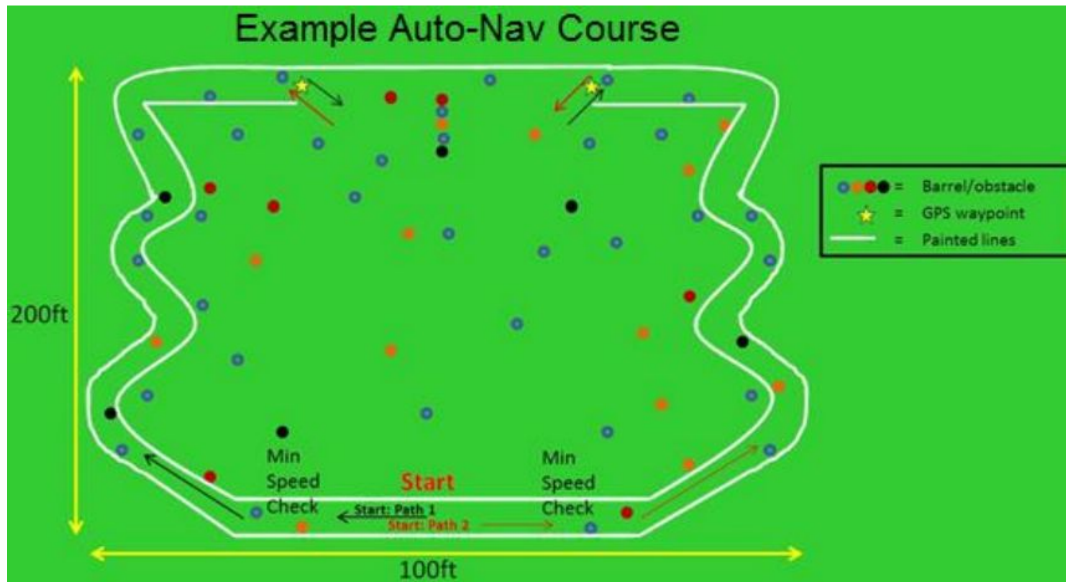

Figure 1: Example Auto-Navigation Course Map (IGVC Rules)

2.2 - Previous IGVC Robots

The IGVC competition requires that each team publish a design report presenting the conceptual design of the vehicle and its components. In the report, the team must include a description of their "electrical system, actuators, sensors, computers, signal processing, mapping, path following, control decisions, system integration, software strategy, and high speed operations" [3]. By analyzing previous teams' design reports, we can ascertain popular sensors, important design concepts, and get an idea of how to approach certain parts to the IGVC competition.

2.2.1 - Georgia Institute of Technology - JAYMI (2017)

Georgia Institute of Technology has a robotics organization, RoboJackets, that has been competing in every IGVC competition since 2006. The RobotJackets team is organized into mechanical, electrical and software subteams totaling 20 students all together. For the 2017 competition, their mechanical team decided to completely redesign their robot from the ground up, 3D modeled a base, then built it out of aluminum. Their senor suite consisted of a TiM5551 SICK AG LIDAR, Outback A321 Dual Frequency GPS, 9 Degree of Freedom Razor IMU, Logitech HD webcam, two Intel NUC5i3RYH computers, and US Digital EM1 optical quadrature encoders. It is important to note that their LIDAR is rated for outdoor use and is a popular choice for mobile robotics.

The RobotJacket's software team ported their code to use Robot Operating System (ROS), a standard middleware package commonly used for robotics research. They also spent a considerable amount of time developing a 3D model of the competition for Gazebo, ROS's built in simulator. Additionally, they created a model of their robot in the Unified Robot Description Format (URDF) so that they can simulate their robot inside the Gazebo simulator. For path planning, their team implemented an A* search algorithm to look through all valid nodes and determine an optimal path. They determined an optimal path by "using [a] soft threshold to weight the different paths and reward the paths that have the greatest minimum distance between

their nodes and an obstacle." [4]. Their vision software utilized a Canny edge detector to detect the lines for the course, and the Hough circle algorithm to detect circles that could be potholes. Once a circle was detected they compared the shape, size, and location against a set criteria to detect if it is a pothole. For mapping, their team created a global reference frame, drew in obstacles into it, assuming that their location was static, and their position error was reliable. This method allows for the creation of a static 3D map, however comes with the drawback that if there is any position errors it would be propagated throughout the map.

2.2.2 - Indian Institute of Technology - Bombay (2017)

In the 25th IGVC (2017) the IIT Bombay team came in first place for the Auto Nav Challenge and 2nd place in the Design Competition putting them in 1st place for the competition overall.

Mechanically, the chassis was designed with differential drive and suspension on the front caster wheel to more easily navigate around obstacles, and over bumps. The electronics are housed in a weather protective case and the GPS, LIDAR and camera sensors are mounted outside. The cameras are mounted high on an easily adjustable mount.

The IIT Bombay team used a SICK LMS111 LIDAR in combination with two SJ4000 Go-Pro style cameras mounted high up to detect obstacles and lanes as well as a Hemisphere Atlaslink GPS, Spartan AHRS 8 IMU, and US Digital S1 encoders to localize the robot. All the processing for theses sensors was conducted on board their Asus GR8 II mini-PC which is designed for gaming intensive applications and contains a GTX 1060 GPU and an i7 generation processor. Their wireless electronic safety system utilizes a transmitter which sends messages to the receiver on-board up to 1 kilometer away. If the robot does not receive heartbeat messages from this transmitter it shuts the robot down after 10 milliseconds.

The software architecture was organized with the ROS publisher/subscriber method entirely on their mini-pc on board.

The primary mapping strategy was to combine the detection of lanes from their cameras with their LIDAR point cloud data to populate an occupancy grid with lane and obstacle information. Adaptive thresholding is used with image data to detect lanes which are higher color intensity than the surrounding environment. The transformed image is then added to the occupancy grid as lanes the robot should avoid. IIT also implemented an iterative closed point scan-matching (ICP) algorithm to convert LIDAR data into a map of obstacles in the area.

They were not able to implement any SLAM algorithms due to high computation resources, so the localization is handled independently from mapping using an Extended Kalman Filter using the GPS, IMU, and encoder odometry data. To navigate to a desired waypoint, the D* algorithm is used to create a path to the goal which is then smoothed and straightened for more feasible motion. A planned velocity is calculated using difference in current and desired heading as well as distances to the nearest obstacles which allows them to be moving at the fastest safe speed.

3 System Architecture

3.1 - Hardware System Overview

The current hardware structure is outlined in the block diagram below. The five subsections are: Power System, Control, Actuators, Environmental Sensing, and External Peripherals.

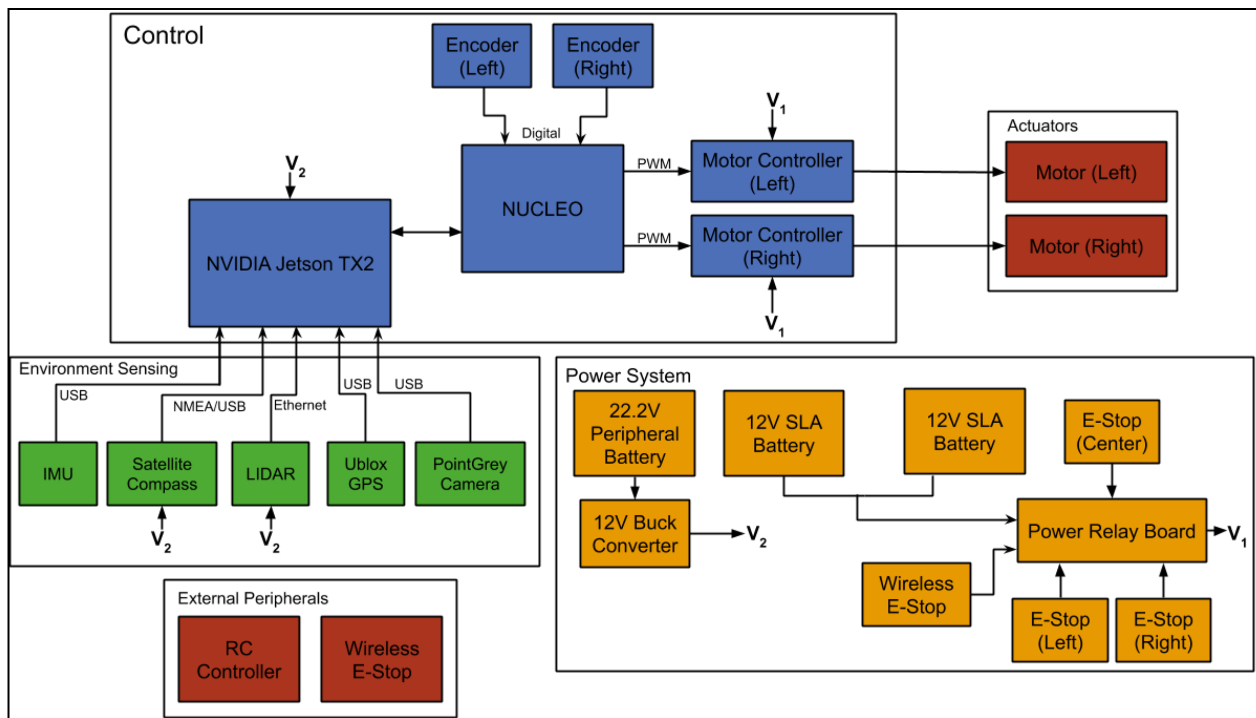

Figure 2: System Hardware Block Diagram

3.2 - Software System Overview

Our Husky's software structure was built using Robot Operating System (ROS). ROS is a "flexible framework for writing robot software. It is a collection of tools, libraries, and

conventions that aim to simplify the task of creating complex and robust robot behaviors" [8]. An autonomous vehicle software structure is very complex, and we decided to take advantage of ROS to help break our software into simpler pieces for an efficient software structure. An overview of our hierarchical software system is shown in the system block diagram below (Figure 3). A hierarchical software architecture decomposes functionality into layered subsystems where the lower-layered subsystems service the higher-level subsystems with services such as drivers and access to I/O.

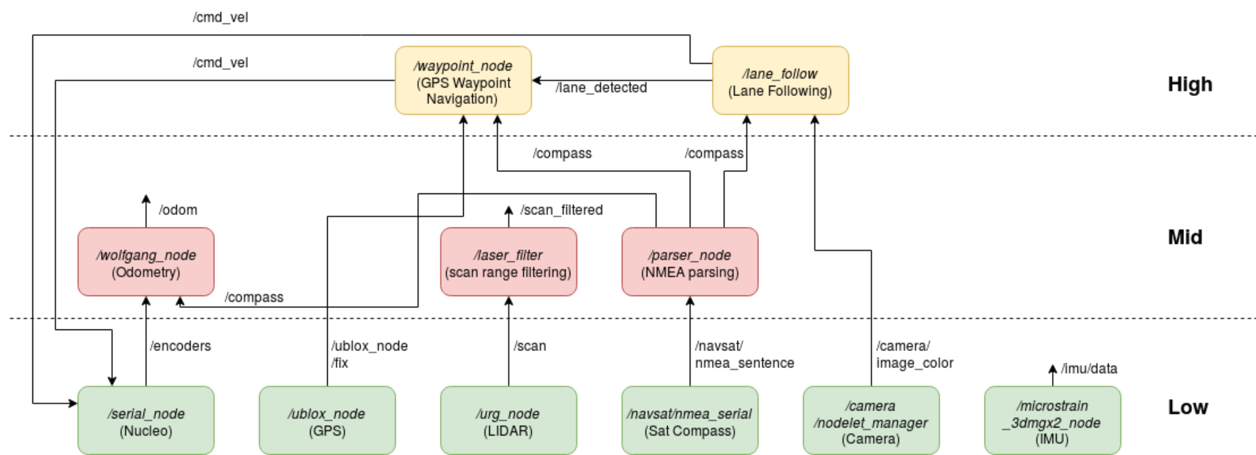

Figure 3. ROS Node Software Hierarchy

In our hierarchy, the subsystems are represented by ROS nodes. The lower layers primarily include nodes such as the *ublox_node* that read sensors and publish their data as more convenient ROS messages over topics that can be subscribed to by the higher level nodes such as *waypoint_node* and *lane_follow* which use that information to implement the logic that gives the robot autonomy. Mid-level nodes such as *wolfgang_node* may combine multiple lower-level sensing sources to produce to produce odometry that can be used by higher level subsystems.

4 Robot Development

4.1 - Research

4.1.1 - Robot Platform

A critical aspect of developing any autonomous vehicle system is having a mobile base that is consistent, reliable, and is physically capable of moving within its environment. Most teams competing in the IGVC end up using or developing their own mobile platform designed for this competition; however, we decided to go with a pre-built platform due to our small team size of 3 people compared to the 8 to 20 members on most IGVC teams, and because our skills and interests tended towards the electronics and software challenges of this competition.

Figure 4: Husky A100 Mobile Robotics Platform

We decided to use the Husky A100 base for a few key reasons. Firstly, it is proven to work reliably in terrain much rougher than that of the IGVC. Its maximum speed is 1.5 m/s (3.35 mph) unloaded which is below the hardware enforced speed limit of 5 mph [3]. It is also designed as a research platform so it is easy to adapt with our own sensors and control. Since the A100 is skid steering, it has the ability to rotate in place, allowing the kinematic model of the base to be simplified to a single point for path planning. This greatly reduces the complexity of path planning.

4.2 - System Components

4.2.1 - Main Computer

The robot's main computer is the NVIDIA Jetson TX2. The Jetson is an embedded system-on-module (SoM) with dual NVIDIA Denver2 and quad-core ARM Cortex-A57, 8GB 128-bit RAM and integrated 256-core Pascal GPU [6]. This platform is ideal for mobile robotics research due to its small size and powerful processing capabilities. Many researchers are using it for deep learning and computer vision tasks. Our team is using the Jetson, along with the Jetson Development board, as the central computer for our robot. We are using the Development board so the jetson can connect to a WiFi network without additional peripherals, and to allow easier access to the built in general purpose input/output (GPIO) pins. The Jetson will be responsible for reading in all of the sensor data, running all ROS nodes, computer vision algorithms and sending serial data to the Nucleo for motor control.

4.2.2 - Nucleo Microcontroller

To ensure that we have real-time control over the Husky A100 base, a STM32 Nucleo-144 F746ZG is used as a dedicated controller. The Nucleo hosts an embedded 216 MHz ARM processor with 1MB of Flash memory, 168 interruptible GPIO pins, and other features like an ethernet jack and capability to be programmed with the mbed framework which has an

intuitive API with extensive library support. The Nucleo communicates with the Jetson over serial with the *rosserial_mbed* library by receiving base control messages to interpret for the motors and reporting encoder odometry information to assist in localizing the robot.

4.2.3 - Secondary Power Supply

The robot's primary power source is two 12V, 21Ah sealed lead-acid batteries. Despite the ability for the SLA batteries to deliver huge instantaneous currents, it was found that the when high torque is required, the motors are capable of drawing enough current to drop the supply voltage close to zero volts. This causes the entire system, including the NVIDIA Jetson TX2 to reset. This cannot occur during a competition run, therefore a secondary power supply is required to maintain a constant supply voltage. This system is the primary power source for the sensors, peripherals, Nucleo, and NVIDIA Jetson TX2 Development board.

The Jetson Development board contains a 5 V DC-DC converter capable of powering most sensors, so the only components which require direct power from the secondary power supply are the TX2 itself, the satellite compass, the wireless emergency stop receiver, and the LIDAR, all of which can be supplied with 12 VDC. The selected Lithium Polymer battery has 6 cells in series, meaning the nominal battery voltage is 22.2 V when fully charged. This means that a 12 V output buck converter is required to step down the DC voltage to a level safe for the components. The capacity is 18000mAh, leading to over 10 hours of continuous use with a 30% power margin, shown in Figure 5 below.

Components	Voltage (V)	Current (mA)	Power (W)
TX2	17	882	15
LIDAR	12	667	8
Camera	5	700	3.5
GPS	5	50	0.25
Satellite Compass	12	240	2.88
RC Receiver	5	0	0
RC E-Stop Receiver	5	0	0
		Total Power (W)	29.63
		Margin (%)	30.00%
		Power With Margin (W)	38.52
		Battery Voltage (V)	22.2
		Capacity (Ah)	18.00
		Operation Time (Hours)	10.37

Figure 5: Estimated Peripheral Power Consumption and Battery Operation Time

4.2.4 - GPS

In order for our robot to get a reliable position estimate, we are using a global positioning system (GPS) to read the robot's latitude and longitude coordinates. Our group is using the u-blox C94-M8P RTK sensor for our GPS. The C94-M8P is an evaluation board containing the NEO-M8P high precision GNSS(Global Navigation Satellite System) receiver module. The C95-M8P has a CEP (Circular Error Probability or standard deviation) of 2.5 m and one of 2.5 mm when using the Differential GPS corrections under a "clear sky"[10]. The u-blox provides coordinate data using the Proprietary Ublox UBX protocol that we are using over a serial USB connection directly with the NVIDIA Jetson TX2.

4.2.5 - Satellite Compass

To attain compass heading information the Simrad HS70 Satellite Compass is used. This gives position and compass data by using two GPS receivers on each end of the sensor. It also has a rate gyro and two tilt sensors inside to retain heading information if GPS signal is lost. Given that the Husky Drivetrain is skid steer it will inherently slip at times. The HS70 is used in conjunction with the encoders to attain odometry.

4.2.6 - IMU

As a redundant measure, a LORD Microstrain 3DM-GX3-45 inertial measurement unit (IMU) sensor is used to measure robot acceleration and orientation using inertial accelerometer and gyroscope sensors. Inertial techniques using this sensor provides an alternative method to odometry of localizing the robot that avoids the problem associated with wheel slippage and remains effective when GPS signal is lost (whereas the Satellite compass and GPS do not). On the robot, the IMU is used to detect faults such as wheel slippage and acts as a substitute for the Satellite Compass when GPS signal is lost.

The 3DM-GX3-45 has a small form factor of 37mm x 41mm and is mounted on the aluminum platform of the husky in between the center wheels and within the sensor tower structure.

4.2.7 - Encoders and Motors

The encoders are US Digital 360 CPR Quadrature encoders and they are attached to two brushed CIM motors which each drive one side of the drivetrain. These are the same that come with the Husky A100, and have 1800 counts per wheel rotation.

4.2.8 - Mono USB3.0 Camera

The Point Grey Chameleon 3 camera is a high quality camera that works well for our vision application. It has a large angle of view, and automatically adjusting parameters, such as iris, to help keep consistent colors when the brightness changes. We are mounting the Chameleon camera approximately 68” above the ground, with the help of an 8020 tower and a 3d-printed mount. The camera is also angled 30 degrees towards the ground to have an optimal view of the course directly in front of the robot base. The camera will be used to detect both the white potholes, and the white lanes. OpenCV was used to implement detection of the painted features, and a projection matrix was used to convert the 2D coordinate positions of the features to 3d world coordinates translated into the base link.

4.2.9 - Light Detection and Ranging (LIDAR)

Figure 6: Hokuyo UXM-30LAH-EWA Scanning Laser Rangefinder

Light Detection and Ranging (LIDAR) sensors are an integral component of many autonomous systems. LIDAR systems are used to get a series of x, y, and z points representing the various objects, and their distances from the sensor. Inside a conventional LIDAR, there is an optical laser that transmits pulses of light, and a spinning mirror reflects that beam of laser out to the world. Everytime the pulse of light is emitted, the precise time is recorded. Once the laser bounces of a nearby object and returns to the LIDAR sensor, the exact time of reflection is also recorded. By using the constant speed of light, the time difference can be converted into a “slant range” distance. By knowing the position and orientation of the sensor, the x,y,z coordinates of the reflected surface can be calculated [9]. For our project, we need to detect various obstacles in front of the robot to prevent any collisions. Our professor already had a Hokuyo UXM-30LAH-EWA Scanning Laser Rangefinder sensor, and allowed our team to borrow it for our project.

4.2.10 - Sensor Tower

For each of these sensors to work effectively they need to be mounted appropriately on the robot. To do this we designed and manufactured a versatile sensor tower out of 80/20 aluminum frame and laser cut wood to fit on the top plate of the Husky A100, Figure 7.

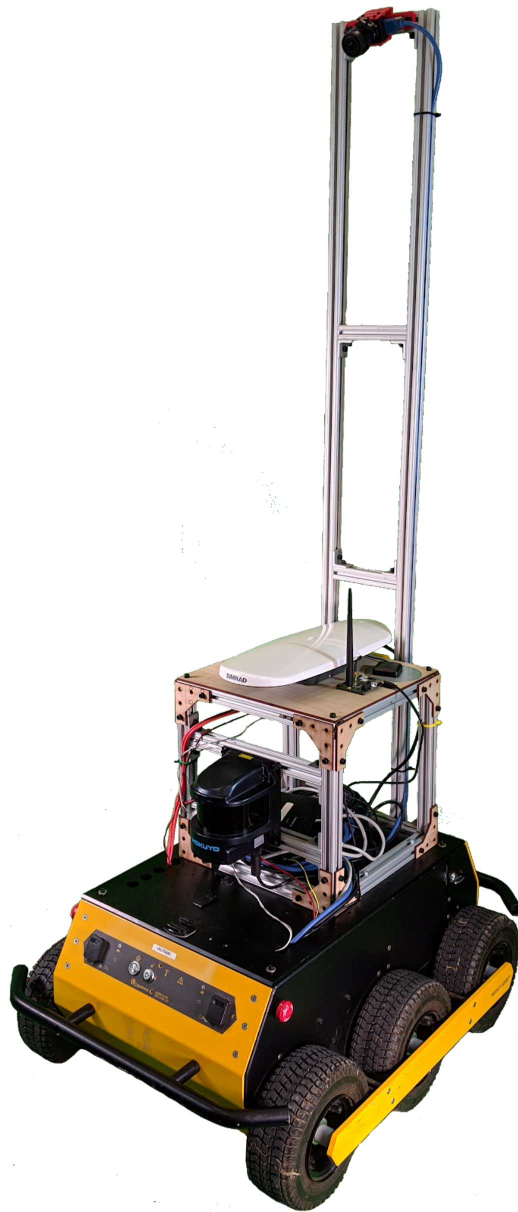

Figure 7: Sensor Tower

The tower had to fulfill a few requirements for sensing and for the competition. The red emergency E-Stop button "must be located in the center rear of vehicle at least two feet from ground" for easy accessibility in case of an emergency[3]. The Satellite compass and Ublox module needed to be far up on the robot to guarantee nothing is blocking their signal to the GPS Satellite constellation. The LIDAR needed to be low enough to detect the traffic barrels and the

Realsense had to be located far enough above the LIDAR such that it did not interfere with its field of view.

5 Software Development

5.1 Jetson TX2 Software

For our platform we have a multitude of sensors that each have their own USB device that are plugged into a USB hub that goes into the TX2. Everytime a USB is plugged in, if the correct drivers are installed, then the computer will assign the USB device a dev port. However, if you unplug and then plug back in the device, there is no guarantee that it will be the same dev port. It is important while interacting with USB devices for our system to be able to know exactly the path that each sensor will have for consistency. In order to remove this uncertainty we developed a series of udev rules which detects the specific USB device and will create a symbolic link to it.

```
SUBSYSTEM=="tty", ACTION=="add", ATTRS{idVendor}=="1546", ATTRS{idProduct}=="01a8", SYMLINK+="ublox"
SUBSYSTEM=="tty", ACTION=="add", ATTRS{serial}=="1163885", ATTRS{idProduct}=="03b1", ATTRS{idVendor}=="1576", SYMLINK+="compass"
SUBSYSTEM=="tty", ACTION=="add", ATTRS{serial}=="0675FF485550755187021745", ATTRS{idVendor}=="0483", ATTRS{idProduct}=="374b", SYMLINK+="nucleo"
SUBSYSTEM=="usb", ACTION=="add", ATTRS{idVendor}=="8086", ATTRS{idProduct}=="0ad6", KERNEL=="1-1.2", SYMLINK+="realsense"
```

Figure 8: Udev Rules for Sensors

On ubuntu distributions, the various rules for devices are stored in the directory */etc/udev/rules.d*. After reading many tutorials online, we were able to appended the *99-tegra-devices.rules* file and add four rules for each sensor which can be seen in Figure 8. The rules looks for information specific to each USB device and then assigns it a symbolic link in the format */dev/sensorName*. For example for the ublox GPS device, the rule checks for the idVendor of 1546, and the idProduct of 01a8 and then creates a symbolic link to */dev/ublox*. We were able to find the information for each sensor by running the command *udevadm info -a -p $(udevadm info -q path -n /dev/ttyACM0).* This lists all of the udev information for the usb devices under the *ttyACM0* USB branch. We followed the same format for the satellite compass sensor, the nucleo microcontroller, and the realsense camera. Now that each sensor has a

consistent dev path, we can write their USB paths in various roslaunch files and know that it will always be able to find the respective device.

5.2 Sensor Communication

5.2.1 Nucleo Microcontroller

The role of the Nucleo MCU is act as a low level controller that reads encoder data, deals with communication between the Jaguar motor controllers, and then forwards this information to the main TX2 controller. We began by writing C++ software that sends pulse width modulation (PWM) commands to the two digital pins that connect to the Jaguar Motor controller.

```
; PlatformIO Project Configuration File
;
;   Build options: build flags, source filter
;   Upload options: custom upload port, speed and extra flags
;   Library options: dependencies, extra library storages
;   Advanced options: extra scripting
;
; Please visit documentation for the other options and examples
; https://docs.platformio.org/page/projectconf.html

[env:nucleo_f746zg]
platform = ststm32
board = nucleo_f746zg
framework = mbed
```

Figure 9: PlatformIO configuration file

To flash the Nucleo, we installed and used Mbed CLI, which is the Arm Mbed command-line tool along with a compiler called platformio. After the installation, we created our own platformio package, *Nucleo-Husky-MQP*, and also made it a shared Github repository. We then edited the initialization file, Figure 9, which is used when it compiles the code inside the src folder. We are able to flash the code to the nucleo by running the command, *platformio run --target upload*, while inside the *Nucleo-Husky-MQP* directory.

The Nucleo is connected over USB to a USB hub which is then connected to the Jetson TX2. In order to get the Nucleo to communicate with ROS, we needed to make the Nucleo code

into a ROS node. We ended up using the *rosserial_python* package to bridge the communication gap between the Nucleo and ROS. We are using the *serial_node.py* python script, and setting the default ROS parameter port to point to the symbolic link of the Nucleo's USB path. Additionally we set the default ROS parameter for baud rate to 57600.

```
[wolfgang/Encoders]:
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
int32 LeftEncoder
int32 RightEncoder
```

Figure 10: ROS Encoders message

The C++ code for the Nucleo is responsible for reading the values of the two encoders in the Husky, and then publishing the change in encoder ticks to a ROS topic on the Jetson TX2. We created a custom ROS message, Figure 10, that is used to keep track of the encoder ticks, and the exact time that they were read in at. The encoders we have are quadrature encoders, we decided to use the Mbed QEI library, which converts angular position to an integer value. Inside the main loop of our code, we keep track of the previous encoder readings and calculate the difference between the current encoder readings and the previous. We then package a new instance of the Encoders message and publish it via *rosserial_python* to the */encoders* topic.

The second responsibility for the Nucleo is to receive motor commands from the TX2 and then translate them into PWM signals to send to the Jaguar motor controllers. A ROS node sends messages on the /*cmd_vel* topic and the *rosserial_python* node transmits the data over serial to the nucleo. We used the standard differential drive model for the physics of our husky robot. We used the kinematics to form a C++ class, *base_kinematics.cpp* and *base_kinematics.h*. Inside our nucleo's main code, we have a ROS subscriber for the /*cmd_vel* topic. The callback

function, *twistCallback*, takes in a geometry message Twist and then writes the values we read in as a motor commands with the instance of our *base_kinematics* object.

5.2.2 Ublox GPS

The Ublox C94-M8P module is used as a standalone GNSS receiver in a competition setting but it can also be used with another base station u-blox in differential mode (DGPS) for testing. Differential GPS is not allowed in the IGVC competition; however, it can be used for testing. For future testing, the differential mode could be used in which the Ublox receives RTK correction signals from another Ublox set up as a base station to provide millimeter-level “truth” data to compare our localization algorithms with. This differential mode happens over a 915 MHz link between the base and rover modules so that the robot can continue receiving the same kind of position messages as if it were in standalone mode with simply more precision. The Ublox GNSS modules already have a ROS package created for interfacing with them on their proprietary UBX protocol over USB. The *ublox* ROS package provides premade configuration files to be loaded on to both the rover and the base station receivers which we modified slightly to fit our application.

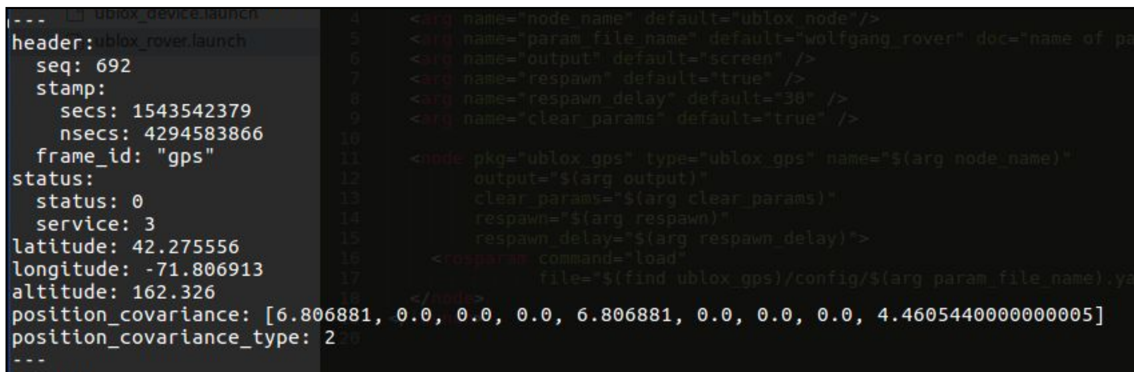

Figure 11. rostopic echo of *sensor_msgs/NavSatFix* message example

The basic ROS driver itself publishes position and velocity in the GNSS frame (i.e. lat-lon-alt) as well as some diagnostics information. The position is published on the as a

sensor_msgs/NavSatFix message on the *ublox_gps/fix* topic, Figure 11. However, the package comes with a sub-package *ublox_gps* that subscribes to these topics and publishes a myriad of more refined messages such as *ublox_msgs/navsat*, which gives information about all the satellites in the constellation that the GPS is using.

5.2.3 Simrad Satellite Compass

The satellite compass installed on the robot is a Simrad HS70. It is intended for use on boats, and thus uses the "National Marine Electronics Association" (NMEA) 2000 communication standard. A NMEA network uses a single-backbone structure in which each device acts as a "node", and every node is connected to a single cable, called a "backbone". The differential data wires are terminated with 120Ω resistors on each end to force the data wires into known states. The NMEA network is similar to a CAN bus, however power is also transmitted throughout the backbone cable. An outline of our NMEA network is shown in Figure 12 below. The NMEA 2000 messages will be transmitted using the ROS package *nmea_comms* and published onto the topic */navsat/nmea_sentence*. A sentence parser will convert the NMEA messages to a custom message format that will hold the relevant information such as heading, attitude, and speed. A list of useful NMEA sentences are shown in Figure 13 below. Every sentence has a unique "PGN" indicating the type of data the sentence holds, followed by the data itself. The highest frequency data recorded by the satellite compass is 10 Hz, so naturally our node will publish at the same frequency.

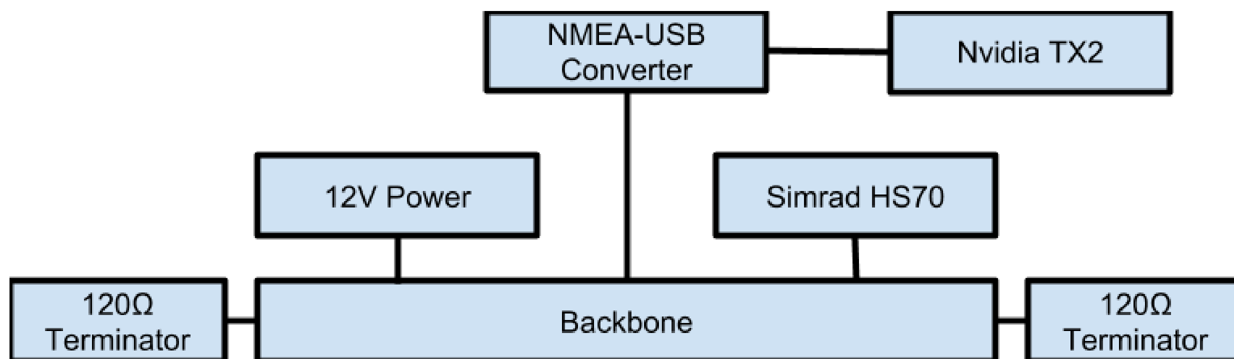

Figure 12: NMEA Network for Satellite Compass

PGN	Description	Frequeny (Hz)
126992	System Date/Time	1
127250	Vessel Heading	10
127251	Rate of Change of Heading	10
128259	Speed	1
129025	Position, Rapid Update	10
129029	GNSS Position Data	1
129027	Position Delta	10
127257	Attitude Field	1

Figure 13: Satellite Compass NMEA Messages

5.2.4 LORD Microstrain IMU

The 3DM-GX3-45 IMU is connected to the Nvidia Jetson TX 2 through USB and uses the microstrain_3dm_gx3_45 ROS Package Drivers and nodes to communicate data and publish to the */imu/data* topic. It will be used as an input to the fault-checking node to check other sensors and as a backup to wheel encoders and satellite compass source for odometry.

5.2.5 Hokuyo LIDAR

After mounting the LIDAR sensor on our custom sensor mount, we had to integrate it with our software system. The Hokuyo UXM-30LAH-EWA LIDAR that was given to our team had an ethernet cable attached, and a pair of wires with an AC power adapter. Since we are powering the entire robot through our lithium batteries, we removed the AC power adapter, and soldered on an anderson powerpole connectors. This allowed us to power the entire LIDAR from the 4-position power distribution block inside our Husky robot.

After the system was properly powered, we needed to connect the LIDAR to the NVIDIA jetson TX2, and interface it with ROS. On ROS's website, there is an existing software package called hokuyo_node, which translates serial data from any usb hokuyo LIDARs and populates ROS laser scan messages. Since our sensor uses ethernet, we needed to use a wrapper package, urg_node, which allows data to be transferred over ethernet and still use the hokuyo_node. Once the LIDAR is powered, we plugged in the LIDAR's ethernet into the TX2, and confirmed that the TX2 was able to see the LIDAR by running the ifconfig command in a terminal window. In order for the wrapper node, urg_node, to consistently connect to the LIDAR, we had to create a rule for the TX2 to automatically assign the LIDAR a static ip address. We did this by getting

into the interfaces directory by running the command cd /etc/network/interfaces.d/ in a terminal window. Then we created a new file, lidar.rules, and added the following lines:

```
auto eth0
iface eth0 inet static
address 192.168.0.15
netmask 255.255.255.0
gateway 192.168.0.1
metric 800
```

Figure 14: LIDAR ethernet rules

Once we created this rule, we confirmed we could communicate with the LIDAR by pinging the static ip address we set, by running *ping 192.168.0.10* and were able to see the packets we sent coming back.

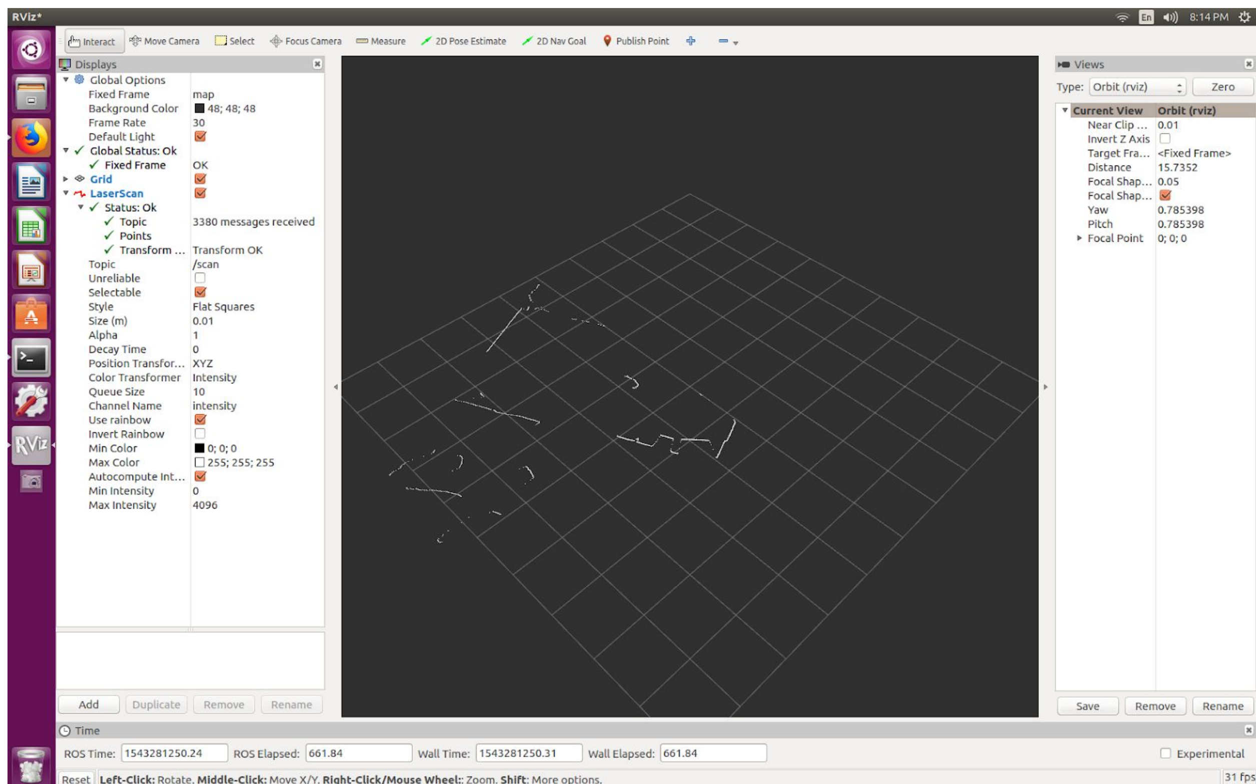

Figure 15: 2D laser scan data in RVIZ

Inside the ROS urg_node package, we then edited the launch file so the default ip address is set to the one we assigned, *192.168.0.10*. We were able to confirm the wrapper worked by powering the LIDAR and running the launch file by running *roslaunch urg_node urg_node*. Additionally, we created a static transform between the laser frame and the map frame to establish data for RVIZ. We were able to do this by running the command *rosrun tf static_transfrom_publisher 0 0 0 0 0 0 map laser 100*. This was just a temporary transform, and does not truly represent the translation offset from the robot. Once the transform was running and the urg_node was running, we ran RVIZ, Figure 15, and were able to see the walls of our lab populate.

5.2.6 Point Grey Chameleon3

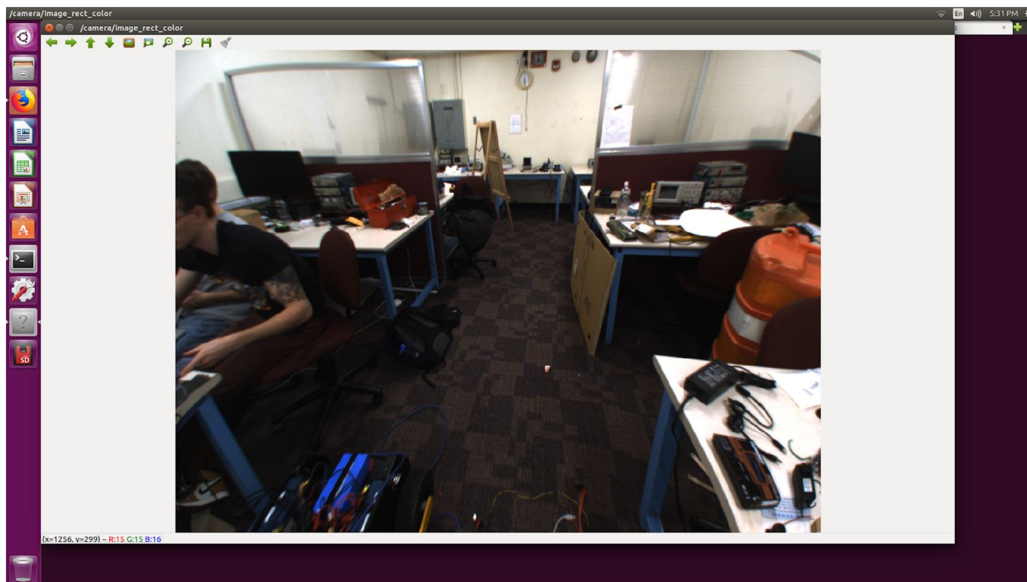

Figure 16: Undistorted Chameleon3 Image

The Point Grey Chameleon3 camera is used on our platform to identify potholes and lanes. Before the image could be used for detection, it had to first be undistorted. Using a checkerboard and the ROS package "camera_calibration", a yaml file containing the intrinsic camera matrix was generated. The intrinsic camera matrix contains properties of the camera and lens, such as the optical center and focal lengths.

Several topics are used to generate and use the image. First, the camera publishes buffered frames to “/camera/image_mono”. Next, the ROS package “image_proc” subscribes to this topic and publishes an undistorted image (using the intrinsic matrix), called “/camera/image_proc/image_rect”. Finally, the wolfgang_vision node subscribes to the rectified image topic and performs feature detection.

5.3 - Sensor Transformations

The transformations for each sensor from the robot’s base frame are shown in Figure 17 below.

X (Forward) (m)	Y (Left) (m)	Z (Height) (m)	frame_id
-0.075	0.000	0.708	base_compass
-0.176	0.105	0.683	base_gps
0.151	0.000	0.506	lidar
0.114	0.000	0.659	pointgrey
.264	.073	.345	imu

Figure 17: Sensor Transformations

5.4 Software Stack

For the IGVC autonomous navigation competition, our robot has to navigate a field, staying in between the white lanes, avoiding obstacles and completing the course as quickly as possible. Additionally there is a section of the course in which there are no lanes and only objects such as traffic barrels, a ramp and shrubbery. Prior to the competition we will be given four GPS waypoints that are within the no lane section to aid in our navigation. For our high level software we break up the competition into a series of steps to follow:

1. Detect traffic barrels, white lanes and potholes

2. Determine the best point in between the white lines
3. Set point as the next waypoint
4. Continue with logic until the robot gets to the lane free zone
5. Then have the robot use GPS waypoint navigation until it finds the lane
6. Return to steps 1-3

Our first step begins with detecting both the white lanes and the potholes. We are using OpenCV and our own computer vision algorithm to detect these lanes. We then determine a point in between the lanes by looking at all of the points and averaging them. We calculate the difference in angle between this point and the center of the image and consider this the error in the robots heading. The heading is controlled to minimize this error using a proportional controller as seen in Figure 18 below that keeping this point at the center of the image and directly in front of the robot. Once the robot drives to this point we repeat the previous step looking for another waypoint. We detect when we are going to enter the no lane section when an insufficient number of lane hough lines is detected and our GPS is within a certain range of our first waypoint. Once we lose these lanes, we publish a false boolean under the */lane_detection* topic. The GPS node listens to this topic and executes when this flag is set to false. We follow a set of predetermined waypoints until we re-enter the lane section. We then continue our previous method until the robot makes it to the finish section.

5.5 Lane and Pothole Detection

Once the image has been rectified, covered in Section 5.2.5, the image is ready to be used for lane and pothole detection. The camera's angle of view is so high that the image must be cropped to ensure the sky is not in the image. Next, since both lanes and potholes are white, a simple brightness threshold is applied to create a binary mask. This leaves the image with just potholes, lanes, and any other white objects in the image. To filter out any smaller unwanted white noise, a morphological transformation is applied to the image known as "opening", which consists of an erosion stage followed by a dilation stage. This results in a loss of detection range,

but is required for edge detection. Next, Canny Edge Detection is used to extract just the edges of the lanes. Finally, a Probabilistic Hough Line Transformation is applied to the detected edges to determine the start and end pixel coordinates of the lanes. However, the lines are not consistent due to several factors, such as the lanes being sinusoidal as opposed to linear, and they must be cleaned up. A piecewise linear function is fit to the detected points to produce a messy, but complete, line representing each lane.

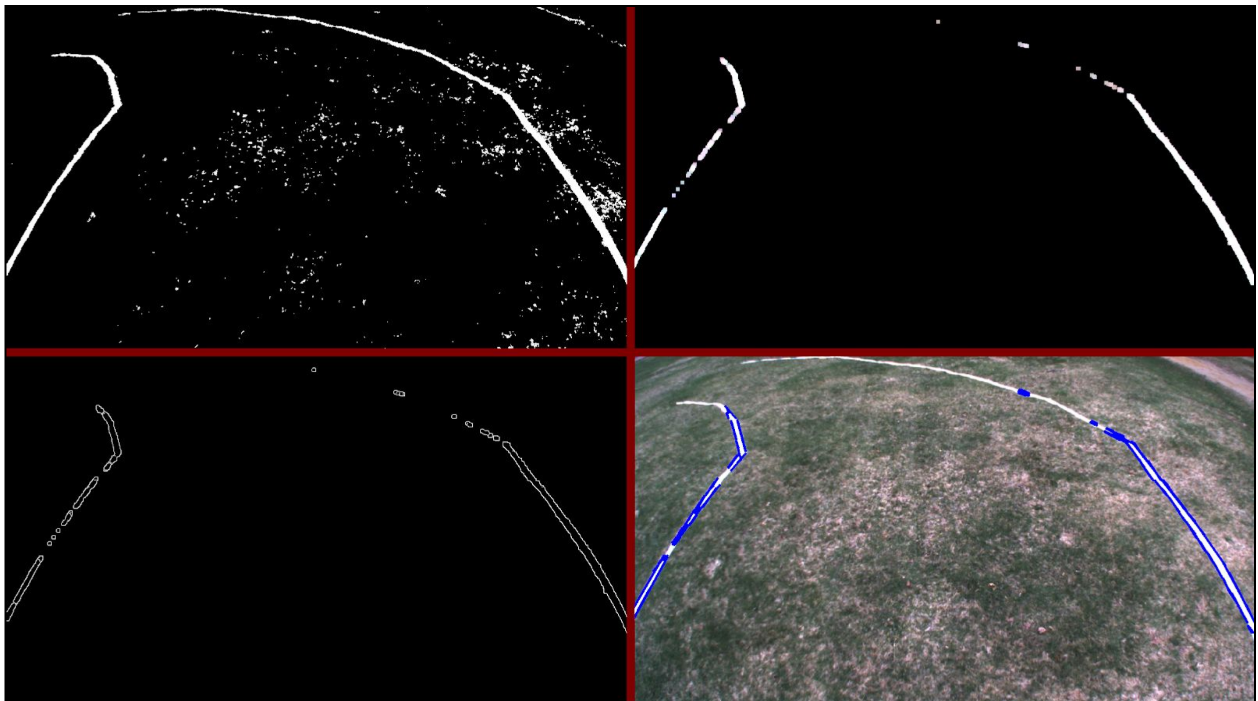

Figure 18: Lane Detection Before Piecewise Linear Fit

Several strategies can now be employed in integrating these features with the autonomous navigation code. One rudimentary strategy is to find a point in the middle of the lines and use this as a waypoint. Another option is to use a projection matrix to convert the 2d coordinates to base-link coordinates and add the features to the occupancy map,

Typically, it is not possible to calculate 3D positions of objects based on a 2D, mono image, because it is impossible to determine lambda, the distance an object is away from the camera. However, since the competition is on a field and all objects detected by the camera are assumed to be on the ground plane, one can calculate lambda, and thus the 3D position relative

to base_link. This can be achieved knowing things: the camera's intrinsic matrix, the extrinsic matrix, and an equation for the ground plane in camera frame. The intrinsic matrix includes the focal length of the camera, and the optical center. The extrinsic matrix contains the rotational and translational information necessary to convert the coordinates from camera frame to local frame.

First, the intrinsic parameter matrix is inverted and multiplied by the pixel coordinates, with a pseudo z-coordinate (typically 1, but could be any scalar). The result is a vector, Qc, beginning at the optical center and pointing towards the 3D coordinate, such that the pixel coordinate is on the vector. Next, the intersection point between Qc and the ground plane is calculated. This is the actual 3D position of the object. However, the coordinates are still relative to the local frame. After multiplying by the rotation matrix and adding the translation, the axis are swapped to match base_link. The points can now be offset by the robot's current position and added to the occupancy grid for further processing.

5.6 Navigation

In order to move between GPS waypoints using path planning algorithms the waypoints need to be converted to the robots cartesian global reference frame *odom*. The waypoints are first be converted geodetic coordinates (lat., lon., atl.) to the cartesian earth-fixed East-North-Up (ENU) frame (x, y, z), given the GPS readings upon startup. Thus GPS waypoints can be converted to an XYZ position relative to the ENU and rotated to *odom* which can be directly added to the occupancy grid for the *move_base* stack to handle.

Sensor	Encoders	IMU	Satellite Compass	GPS
Sensor Type	Odometrical	Inertial	Differential Multilateration	Multilateration
Coordinate System Type	2D-Cartesian	3D-Cartesian	Geodetic	Geodetic
Relative To...	Starting	Robot Frame	WGS 84 North pole	WGS 84

	Position			
Conversion to East-North-Down (ENU)	Rotation from initial heading about Z (Up) axis	Mechanization equations and homogeneous transformation using EKF odometry estimate	None	*geodetic2enu* transform

Figure 19: Sensor Coordinate Transformations

5.7 GPS Waypoint Navigation

The vast majority of the course has complete lanes formed by two separate white lines. However there is a small portion of the course where the lanes disappear and the robot must navigate based on four GPS waypoints given. We developed a ROS node, *waypoint_node*, to handle this GPS waypoint navigation portion of the course. The node subscribes to the GPS, satellite compass, and lane detection status. Since the GPS code is a lower priority than lane following, the code will only be run when a lane is not detected. Inside the GPS callback, the first condition checks the status of the lane detection topic. This message is a boolean representing if the lane is detected or not. If the boolean is false then the code runs. The software calculates the distance between the current GPS reading and the first goal waypoint. It then calculates the angle needed to turn based on the difference between the calculated bearing and the current yaw from the satellite compass. We then create a command vel message with a linear speed of 0.25 meters per second and then calculate the angular speed based off of the calculated angle.

6 Software Validation

6.1 Odometry Testing

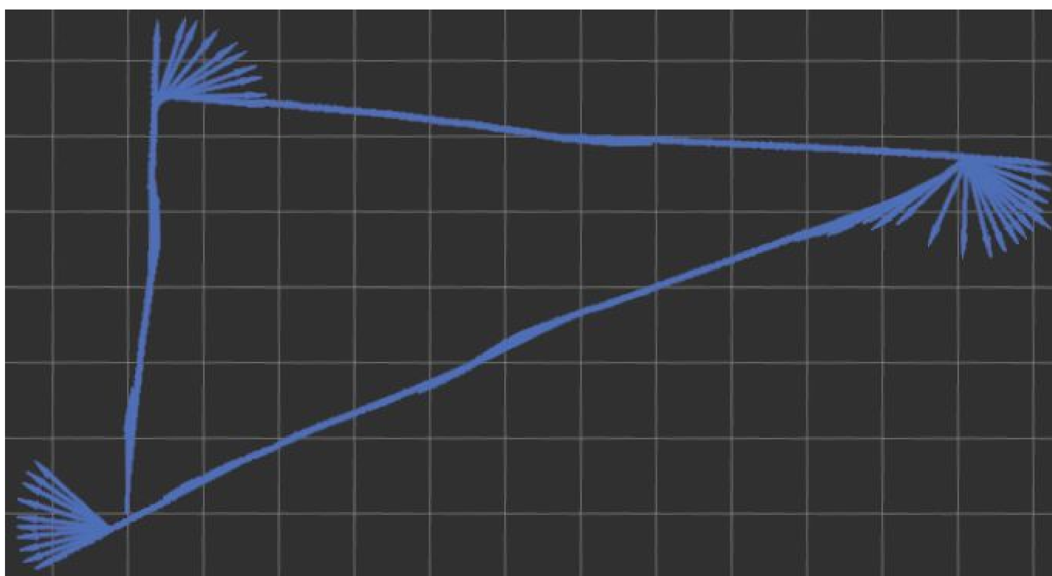

Figure 20: Triangle Odometry Test

Odometry is an integral component of our software stack and needs to be as accurate as possible for navigation. We ran a number of tests in order to assess the accuracy of our odometry software. One of those tests included manually driving the robot in a triangle configuration. We used an xbox controller to send command velocity commands and tested the robot outdoors. We drove straight 7.62 meters, took a 90 degree turn, then drove 15.24 meters straight. Next we turned the robot so it was facing the starting location and drove it straight until it reached the origin. At the end of the first segment of the triangle, the odom topic reported 6.67 meters, which is an error of 0.95 meters. The next section reported a distance of 13.64, which is an error of 1.6 meters.

6.2 Course Test

In order to test out the lane following and GPS course, we set up a mock course in a park. We used white tape to act as our lanes and separated them with about 10 to 15 feet clearance. The mock course mimics the portion of the competition where the lanes disappear and the robot must navigate to the next GPS waypoint. Our test course successfully emulates every portion of this task, with the exception of object detection.

7 Future Work

While our hierarchical software structure works for simple path planning, in order to develop a more robust system capable of handling all of the variables within an outdoor course, a more complex software structure must also be developed. The pre-built *move_base* stack meets all the criteria for a robust, complete, software architecture.

The *move_base* stack will allow the husky robot to drive to various *move_base* goal points. The *move_base* stack is a collection of nodes that links together sensor data, a 2D map, controller, global planner, and local planner. The sensor data that the *move_base* stack takes in is odometry in the form of a navigation message, and point cloud laser scan data. All of the sensors used have different frame transformations from their sensor location to the robot base frame. By having these transformations, ROS's built in transform function, *tf2*, is able to convert all sensor information into the same robot frame.

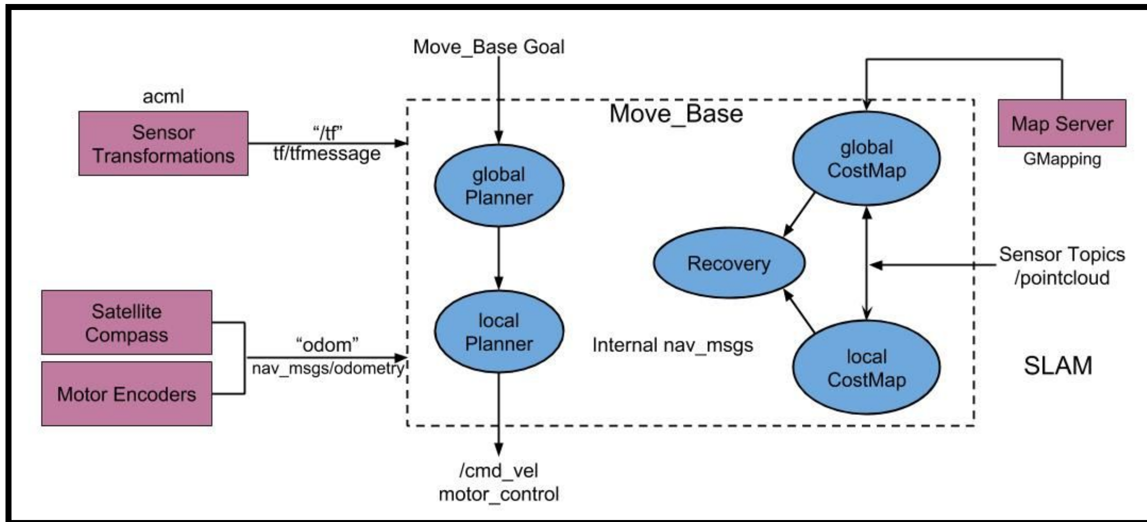

Figure 21: System Software Block Diagram

In order for the *move_base* stack to know its environment, a 2D map is needed under the *map* topic, which is a 2D occupancy grid representing all known obstacles and free space. The most popular method to produce this 2D occupancy grid is through ROS's built-in simultaneous localization and mapping (SLAM) algorithm called GMapping. Typically, the robot is driven manually through the desired area, recording odometry and laser scan data. The GMapping algorithm can produce the map live as the robot is being driven around, or offline if the odometry and laser scan data is recorded and played back with the algorithm.

The odometry message stores an estimate of the position and velocity (pose) of the robot in free space. It contains the estimated position of the robot in the odometric frame, along with an optional covariance matrix for the certainty of that pose estimate. The twist message corresponds to the robot's velocity in the child frame, normally the coordinate frame of the mobile base, along with the optional covariance matrix for the certainty of the velocity estimate.

To provide a better estimation of robot pose, an Extended Kalman Filter would be used from the *robot_pose_ekf* package. This takes the difference between the 2D odometry, 3D orientation, and 3D pose vectors from different sensors and publishes a pose estimation with a velocity covariance, a measure of estimation confidence. However, if a sensor fault occurs, the *robot_pose_ekf* estimation would be drastically skewed. To combat this we would create a

fault-checking node that acts as a an additional filtering step prior to using the Extended Kalman Filter itself. Given that Wolfgang contains multiple redundant sensors and that those sensors' fault profiles are roughly known or can be tested, the fault-checking node can decide whether a sensor is failing and alter the odometry, orientation, or pose data given to the Kalman Filter to use data without faults.

The EKF outputs a 3D pose message to the *robot_pose_ekf/odom_combined* topic which can then be used as an starting position for any path planning maneuvers and as a position that perception sensors record data relative to when building up the course's occupancy grid.

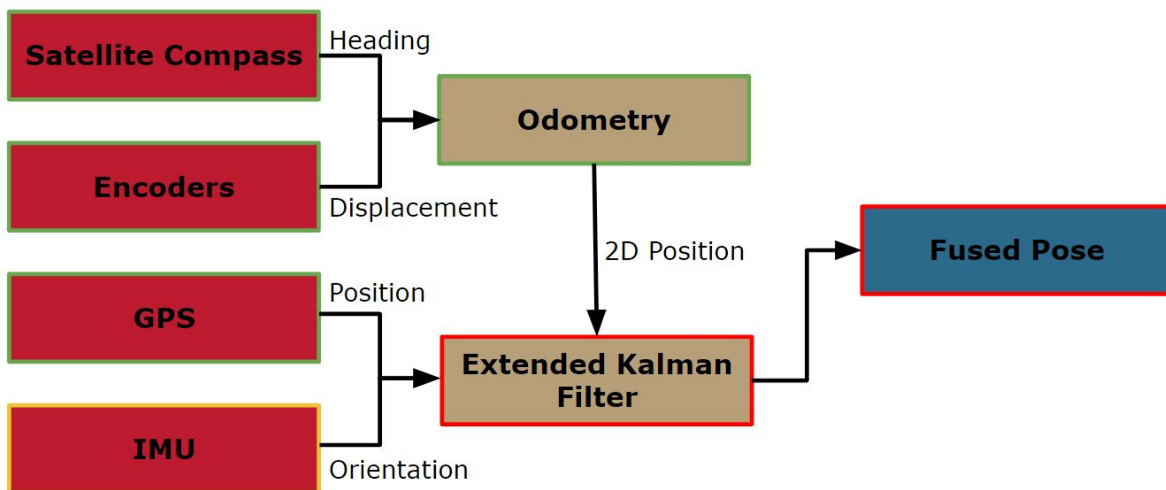

Figure 22: Odometry Pipeline

Once the *move_base* stack has the robot's odometry and a static map, it needs current point cloud data from the lidar and camera in order to detect where it is on the map. In the stack, this point cloud data is overlaid onto the occupancy grid from the map server. The stack supports both a global costmap and local costmap, which allows the software to have a consistent map of known obstacles and a local costmap to be updated with new detected obstacles while the robot is operating. It can be configured to add obstacles from the local costmap to the global costmap if the object has persisted for a set period of time across multiple scans. Additionally, the current laser scan data along with the odometry is used with an Adaptive Monte Carlo Localization

(AMCL) algorithm to localize the robot on the map. The algorithm uses a particle filter with a scan matching algorithm to determine how closely the current scan data lines up with the map.

When the stack has the map, sensor information, and costmaps, it uses a planning algorithm to determine an optimal route from its current location to a desired *move_base* goal position and orientation. The stack can be configured to either use a global and local planner as separate planners or have one single planner. Our team will use global planner for long term planning, and the local planner to route around undetected obstacles that appear in the point cloud data. ROS has a number of built in path planning algorithms that work with the global and local planners. The most common is the A* algorithm, which if implemented properly, will guarantee that the shortest possible path is found. With the *move_base* stack integrated, the robot should be able to navigate the obstacle course predictably and reliably.

8 Software API

8.2 Wolfgang Bringup

The main software architecture is run with Robot Operating System (ROS). In order to bring up all of the sensors on wolfgang, we created a launch file, *wolfgang_bringup.launch*, to startup all the necessary nodes. To run this launch file, connect to the Jetson via SSH and run *roslaunch wolfgang wolfgang_bringup.launch*.

```
<launch>

  <!-- Start wolfgang node responsible for calculating odometry messages -->
  <node name="wolfgang_node" pkg="wolfgang" type="wolfgang_node" output="screen"/>

  <!-- Start rosserial node responsible for nucleo communication -->
  <node pkg="rosserial_python" name="serial_node" type="serial_node.py" output="screen">
    <param name="port" value="/dev/nucleo" />
  </node>

  <!-- start GPS -->
  <include file="$(find ublox_gps)/launch/ublox_rover.launch">
    <param name="node_name" value="ublox_node" />
    <param name="param_file_name" value="wolfgang_rover" />
  </include>

  <!-- start LIDAR -->
  <include file="$(find urg_node)/launch/urg_lidar.launch" />
  <include file="$(find laser_filters)/launch/igvc_range_filter.launch"/>

 <!-- start compass -->
  <include file="$(find nmea_comms)/launch/tee.launch" />

  <!-- start imu -->
  <include file="$(find microstrain_3dmgx2_imu)/microstrain_3dmgx2.launch" />

  <!-- start parser for compass and GPS -->
  <node pkg="nmea_navsat_driver" name="parser_node" type="nmea_topic_driver" />

  <!-- start tf broadcaster -->
  <node pkg="robot_setup_tf" name="tf_node" type="tf_broadcaster" />

  <!-- start GPS transform -->
  <node pkg="geodetic_utils" name="gps_conversion" type ="gps_to_pose_conversion_node" />

</launch>
```

Figure 23: Wolfgang Bringup Launch File

The first component of this launch file is bringing up the wolfgang node. The wolfgang node listens to the encoder and compass data and calculates odometry messages. The next part of the launch file starts up a rosserial node to handle the communication between the Nucleo microcontroller and the Jetson TX2. The nucleo code publishes a custom ROS encoder message and subscribes to command velocity message which it converts into motor commands. The next component is the Ublox node which communicates with the ublox GPS and converts the data into ROS ublox messages. After that we begin running the *urg_lidar.launch* file which is responsible for communicating with the Hokuyo LIDAR and publishes scan messages. Next we run *igvc_range_filter.launch* file which cuts off scan data past 10 meters. We run this because our LIDAR is capable of seeing up to 80 meters, but for our purposes we only are interested in scan data between 0 and 10 meters. Next we run *microstrain_3dmgx2.launch*, which handles communication between our IMU and ROS, publishing ROS IMU messages. Furthermore we run the *nmea_topic_driver* python script which parses the NMEA messages from the satellite compass and publishes a Magnetic Field message. The following node, *tf_brodcaster*, is a custom node that publishes the transforms of the different sensor frames. The last node that we run is *gps_to_pose_conversion_node*, which converts GPS data into a local frame of the robot.

8.3 GPS Navigation Node

We developed a ROS node, *waypoint_node*, to execute our GPS waypoint code. The waypoint node subscribes to the satellite compass, GPS, and lane detection status. The GPS waypoints are set manually at the top of the *NavigateCourse.cpp* file. The main logic of the code will only run when the lane detection status reads false. To run this node, SSH into the Jetson TX2 and run the command: *rosrun wolfgang waypoint_node*.

8.4 Lane Following Node

Additionally we developed software using OpenCV to handle lane following. In order for this to run properly, the nucleo and camera must be connected to the Jetson TX2. This node subscribes to the camera feed and publishes command velocities which are sent to the nucleo and converting into motor commands. To run this node, SSH into the Jetson TX2 and run the command: *rosrun wolfgang wolfgang_node*.

Bibliography

[1] "Car Accident Statistics in the U.S." *Driver Knowledge*. [Online]. Available: www.driverknowledge.com/car-accident-statistics/

[2] "The 27th Annual Intelligent Ground Vehicle Competition," *Intelligent Ground Vehicle Competition*. [Online]. Available: http://www.igvc.org/

[3] "The 26th Annual Intelligent Ground Vehicle Competition (IGVC) & Self-Drive," *Intelligent Ground Vehicle Competition.* [Online]. Available: http://www.igvc.org/2018rules.pdf

[4] "Jaymi," *Robojackets, Competitive Robotics at Georgia Tech.* [Online]. Available: http://www.igvc.org/design/2017/5.pdf

[5] "Design Report, SeDriCa," *Indian Institute of Technology.* [Online]. Available: http://www.igvc.org/design/2017/8.pdf

[6] "Jetson TX2," *eLinux*. [Online]. Available: https://elinux.org/Jetson_TX2

[7] "Intel RealSense Technology," *Intel*. [Online]. Available: https://software.intel.com/en-us/realsense/d400

[8] "About ROS," *Open Source Robotics Foundation.* [Online]. Available: http://www.ros.org/about-ros/

[9] "Lidar 101: An Introduction To LIDAR Technology, Data, and Applications" National Oceanic and Atmospheric Administration (NOAA) Coastal Services Center. 2012. Available: https://coast.noaa.gov/data/digitalcoast/pdf/lidar-101.pdf

[10]https://www.u-blox.com/sites/default/files/NEO-M8P_DataSheet_%28UBX-15016656%29.pdf

Appendix

Lane Detection Source Code

```
#include <ros/ros.h>
#include <math.h>
#include <image_transport/image_transport.h>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/features2d/features2d.hpp>
#include <opencv2/objdetect/objdetect.hpp>
#include <cv_bridge/cv_bridge.h>
#include "spline/src/spline.h"
#include "nav_msgs/OccupancyGrid.h"
#include "nav_msgs/GridCells.h"
#include <tf/transform_listener.h>
#include <std_msgs/Bool.h>

#ifndef LANE_H
#define LANE_H
#include "wolfgang_vision/lane.hpp"
#endif

// IN DEGREES
const float ANGLE_MOUNTED = 30.0f;
const float ANGLE_HORIZONTAL = 73.35f;
const float ANGLE_VERTICAL = 100.0f;

const float INV_FL = .001310616f; // 1 / focal length
const float OPT_CENTER_X = -.8757f; //Optical Center (X)
const float OPT_CENTER_Y = -.6353f; // Optical Center (Y)
const float NORMAL[3] = {0.0f, 136.0f, 78.52f}; // Vector Normal to Ground
const float TRANSLATION[3] = {-.181f, 0.0f, 1.722f}; // Translation from Camera to
Local Frame

// How much of the top screen to crop off
const int ROI_CROP_Y = 350;

using namespace cv;
using namespace std;

SimpleBlobDetector::Params params;
float current_yaw_;

ros::Publisher map_pub;
ros::Publisher lane_detection;
```

```
nav_msgs::GridCells gridMap;


//SimpleBlobDetector detector(params);

float quatToEuler(float x, float y, float z, float w){
  return atan( 2*(x*w + y*z) / (1-2*(pow(z,2) + pow(w,2))) );

}

void imageCallback(const sensor_msgs::ImageConstPtr& msg) {
  try
  {
    // Read image from Topic
    Mat img = (cv_bridge::toCvShare(msg, "bgr8")->image).clone();
    Mat cropped_img, hsv, res, mask, edges, res_lanes;

    std::vector<Vec4i> lines;

    // Crop off Top of Image
    Rect roi(0, ROI_CROP_Y, img.cols, img.rows-ROI_CROP_Y);
    cropped_img = img(roi);

    imshow("Result", cropped_img);

    // Filter White from HSV Image
    cvtColor(cropped_img, hsv, CV_BGR2HSV);
    inRange(hsv, Scalar(0, 0, 225), Scalar(179, 50, 255), mask);
    bitwise_and(cropped_img, cropped_img, res, mask=mask);



    // Canny Edge Detection then Probabilistic Hough Transform from edges
    Canny(res, edges, 100, 200);
    HoughLinesP(edges, lines, 3, CV_PI/180, 30, 10, 10);

    // Just take a copy of the image to display later
    res_lanes = cropped_img.clone();

    std::vector<float> odom_x_values;
    std::vector<float> odom_y_values;

    std_msgs::Bool lane_status;
    if(lines.size() < 6){
        lane_status.data = false;
    }
    else{
       lane_status.data = true;
    }
    lane_detection.publish(lane_status);

    for( size_t i = 0; i < lines.size(); i++ )
    {
```

```
        // Calculate Length of Line Using Distance Formula
        float dist = distance(lines[i][0], lines[i][1], lines[i][2], lines[i][3]);

        // Pixel Increments Using Line Slopes and Distance
             float delta_y = (lines[i][3] - lines[i][1]) / dist;
             float delta_x = (lines[i][2] - lines[i][0]) / dist;
       // line(res_lanes, Point(lines[i][0], lines[i][1]), Point(lines[i][2],
lines[i][3]), Scalar(255,0,255), 3, 8 );

        float x = lines[i][0];
        float y = lines[i][1];

        tf::StampedTransform odom_transform;
             tf::TransformListener listener;
        if(!listener.waitForTransform("/base_link", "/odom", ros::Time::now(),
ros::Duration(3.0))){
         try{
           listener.lookupTransform("/base_link","/odom", ros::Time(0),
odom_transform);
         }
         catch (tf::TransformException ex){
           ROS_ERROR("%s",ex.what());
           ros::Duration(1.0).sleep();
         }
         }

        for (int i = 0; i < int(dist+.5); i++) {
            // Calculate Local Frame Point Location
            float local_point[3];

           cameraToLocal(int(x+.5), int(y-ROI_CROP_Y+.5), local_point);

            float z_rot = quatToEuler(odom_transform.getRotation().x(),
odom_transform.getRotation().y(), odom_transform.getRotation().z(),
odom_transform.getRotation().w());
            float x_trans = odom_transform.getOrigin().x();
            float y_trans = odom_transform.getOrigin().y();

            float odomX = cos(z_rot)*local_point[0] + sin(z_rot)*local_point[1] +
x_trans;
            float odomY = -sin(z_rot)*local_point[0] + cos(z_rot)*local_point[1] +
y_trans;

            odom_x_values.push_back(odomX);
            odom_y_values.push_back(odomY);

            circle(res_lanes, Point(x, y), 2.0, Scalar(255, 0, 0), -1);
            x += delta_x;
            y += delta_y;

        }
```

```
    }
    // Publish Occupied Grid Cells
    publishPoints(odom_x_values, odom_y_values);
    // publishPointStamped(odom_points);
    //imshow("lanes", res_lanes);
    waitKey(10);
    }
    catch (cv_bridge::Exception& e)
    {
      ROS_ERROR("Could not convert from '%s' to 'bgr8'.", msg->encoding.c_str());
    }

}

// Calculate absolute distance between two points
float distance(int x1, int y1, int x2, int y2) {
    return sqrt(pow(y2 - y1, 2) + pow(x2 - x1, 2));
}

// Converts 2D Pixel Coordinate to 3D Local Frame Point
void cameraToLocal(int x, int y, float local_point[3]) {

   // Use Inverse Intrinsic parameters to get Camera Frame Vector with unknown
lambda (scalar length)
   float kqx = x*INV_FL + OPT_CENTER_X;
   float kqy = (ROI_CROP_Y + y)*INV_FL + OPT_CENTER_Y;

   // Calculate length of Camera Frame Vector to Ground Plane Using vector normal
to ground
   float S1 = 10678.72f / (kqy*NORMAL[1] + NORMAL[2]);

   // Calculate Intersection Point of Vector and Ground Plane in Camera Frame
   float intersection_x = kqx * S1;
   float intersection_y = kqy * S1;
   float intersection_z = S1;

   // Rotate Intersection Point so its parallel with the ground and swap axis to
match local axis
   float camera_frame_x = intersection_x;
   float camera_frame_y = intersection_y*.866f + intersection_z*.5f;
   float camera_frame_z = -intersection_y*.5f + .866f*intersection_z;

   // Translate to base_link origin and convert to meters
   local_point[0] = camera_frame_z / 39.37f + TRANSLATION[0];
   local_point[1] = -camera_frame_x / 39.37f + TRANSLATION[1];
   local_point[2] = -camera_frame_y / 39.37f + TRANSLATION[2];
}



// Publishes list of x/y coordinate pairs to grid map
void publishPoints(std::vector<float> x_points, std::vector<float> y_points) {
  gridMap.header.stamp = ros::Time::now();
```

```
   for(int i=0; i < x_points.size(); i++){
       geometry_msgs::Point point;
       point.x = x_points.at(i);
       point.y = y_points.at(i);
       point.z = 0;
       gridMap.cells.push_back(point);
   }

  cout << "Size of grid cells: " << gridMap.cells.size() << endl;
  map_pub.publish(gridMap);
}

// Publishes list of points to grid map
void publishPointStamped(std::vector<geometry_msgs::PointStamped> &odom_points) {
  gridMap.header.stamp = ros::Time::now();
   for(int i=0; i < odom_points.size(); i++){
       geometry_msgs::Point point;
       point.x = odom_points.at(i).point.x;
       point.y = odom_points.at(i).point.y;
       point.z = 0;
       gridMap.cells.push_back(point);
   }

  cout << "Size of grid cells: " << gridMap.cells.size() << endl;
  map_pub.publish(gridMap);
}

void setupBlobDetector() {
  params.filterByArea = true;
  params.minArea = 1000;
  params.maxArea = 15000;
  params.filterByCircularity = false;
  params.minCircularity = .75;
  params.filterByConvexity = false;
  params.filterByInertia = false;
}

int main(int argc, char **argv)
{
  cout << "Major version : " << CV_MAJOR_VERSION << endl;
  setupBlobDetector();
  ros::init(argc, argv, "image_listener");
  ros::NodeHandle nh;

  namedWindow("lanes");
  startWindowThread();
  image_transport::ImageTransport it(nh);
  image_transport::Subscriber sub = it.subscribe("camera/image_color", 1,
imageCallback);
  gridMap.header.frame_id = "map";
  gridMap.cell_width = 0.05;
  gridMap.cell_height = 0.05;
```

```
  map_pub = nh.advertise<nav_msgs::GridCells>("/detected_lanes", 1000);
  lane_detection = nh.advertise<std_msgs::Bool>("/lane_detected", 1);

  ros::spin();
  destroyWindow("lanes");
}
```

Lane Holding Source Code

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/features2d/features2d.hpp>
#include <opencv2/objdetect/objdetect.hpp>
#include <cv_bridge/cv_bridge.h>
#include "spline/src/spline.h"
#include <math.h>
#include <geometry_msgs/Twist.h>
#include <std_msgs/Bool.h>
#include "sensor_msgs/LaserScan.h"

#define PI 3.14159265

// IN DEGREES
const float ANGLE_MOUNTED = 30.0f;
const float ANGLE_HORIZONTAL = 73.35f;
const float ANGLE_VERTICAL = 100.0f;

// IN METERS
const float MOUNT_HEIGHT = 1.7272f;


// How much of the top screen to crop off
const int ROI_CROP_Y = 400;
const int ROI_CROP_X = 0;

using namespace cv;
using namespace std;

SimpleBlobDetector::Params params;
ros::Publisher lane_detection;
ros::Publisher pub;
image_transport::Publisher pub_debug;

//SimpleBlobDetector detector(params);
/*
void scanCallback(const sensor_msgs::LaserScan::ConstPtr& msg){
   for(int i=0; i < msg.
```

```
}
*/
void imageCallback(const sensor_msgs::ImageConstPtr& msg){
  geometry_msgs::Twist twist_msg;
  try
  {
    Mat img = (cv_bridge::toCvShare(msg, "bgr8")->image).clone();
    Mat cropped_img, hsv, res, mask, edges, res_lanes;
    std::vector<Vec4i> lines;

    Rect roi(ROI_CROP_X, ROI_CROP_Y, img.cols-ROI_CROP_X*2, img.rows-ROI_CROP_Y);

    cropped_img = imread("img_raw_cropped.png", 0);//img(roi);



    cvtColor(cropped_img, hsv, CV_BGR2HSV);
    inRange(hsv, Scalar(0, 0, 200), Scalar(179, 50, 255), mask);
    bitwise_and(cropped_img, cropped_img, res, mask=mask);

    imwrite("Filtered.bmp", cropped_img);

    int morph_size = 3;
    Mat element = getStructuringElement(0, Size( 2*morph_size + 1, 2*morph_size+1
), Point( morph_size, morph_size ) );

    morphologyEx(res, res, MORPH_OPEN, element);

    imwrite("eroded.bmp", res);

    Canny(res, edges, 100, 200);

    imwrite("canny.png", edges);
    HoughLinesP(edges, lines, 3, CV_PI/180, 30, 10, 10);
    //imshow("result", res);

    res_lanes = cropped_img.clone();

    imshow("Result_lanes.bmp", res_lanes);

    double x_accum = 0.0;
    double y_accum = 0.0;
    int num_points = 0;

    std_msgs::Bool lane_status;
    if(lines.size() < 20){
        lane_status.data = false;
    }
    else{
       lane_status.data = true;
    }
    lane_detection.publish(lane_status);
```

```
    for( size_t i = 0; i < lines.size(); i++ )
    {

       if (lines[i][1] < img.rows*.75 && lines[i][1] > 0) {
           x_accum += lines[i][0];
           y_accum += lines[i][1];
            circle(res_lanes, Point(lines[i][0], lines[i][1]), 3.0, Scalar(255, 0,
0), -1);
           num_points++;
       }
       if (lines[i][3] < img.rows*.75 && lines[i][3] > 0) {
           x_accum += lines[i][2];
           y_accum += lines[i][3];
            circle(res_lanes, Point(lines[i][2], lines[i][3]), 3.0, Scalar(255, 0,
0), -1);
           num_points++;
             }
    }

    double x = (double)(x_accum / num_points);
    double y = (double)(y_accum / num_points);

    circle(res_lanes, Point(x >= res_lanes.cols ? res_lanes.cols-1:x, y), 4.0,
Scalar(255, 0, 255), -1);
    circle(res_lanes, Point(res_lanes.cols / 2, res_lanes.rows-1), 2.0, Scalar(255,
0, 255), -1);

    if (lane_status.data) {
           double angle = 90 - atan(-(res_lanes.rows - y) / (res_lanes.cols / 2 -
x)) * 180 / PI;

           if (angle > 90) {
              angle -= 180.0;
           }
           cout << "Angle: " << angle << endl;
           twist_msg.linear.x = 0.25;
           twist_msg.angular.z = angle*0.03;
           pub.publish(twist_msg);
    }

    sensor_msgs::ImagePtr msg = cv_bridge::CvImage(std_msgs::Header(), "bgr8",
res_lanes).toImageMsg();
    pub_debug.publish(msg);
    waitKey(30);
  }
  catch (cv_bridge::Exception& e)
  {
    ROS_ERROR("Could not convert from '%s' to 'bgr8'.", msg->encoding.c_str());
    twist_msg.linear.x = 0.0;
    twist_msg.angular.z = 0.0;
    pub.publish(twist_msg);
  }
}
```

```
void setupBlobDetector() {
  params.filterByArea = true;
  params.minArea = 1000;
  params.maxArea = 15000;
  params.filterByCircularity = false;
  params.minCircularity = .75;
  params.filterByConvexity = false;
  params.filterByInertia = false;
}


int main(int argc, char **argv)
{
  cout << "Major version : " << CV_MAJOR_VERSION << endl;
  setupBlobDetector();
  ros::init(argc, argv, "image_listener");
  ros::NodeHandle nh;

  startWindowThread();
  image_transport::ImageTransport it(nh);
  image_transport::Subscriber sub = it.subscribe("camera/image_color", 1,
imageCallback);
  pub_debug = it.advertise("camera/debug", 1);
  pub = nh.advertise<geometry_msgs::Twist>("cmd_vel",1000);
  lane_detection = nh.advertise<std_msgs::Bool>("/lane_detected", 1);
  ros::spin();
  destroyWindow("lanes");
}
```

GPS Waypoint Node

```
#include <wolfgang/NavigateCourse.h>
#define PI 3.14159265

// GPS goal points
float goal_lat1 = 42.2758496;//42.276096;
float goal_long1 = -71.8064903; //-71.806412;

float goal_lat2 = 42.2760963;
float goal_long2 = -71.806446;

float goals_lat[] = {goal_lat1, goal_lat2};
float goals_long[] = {goal_long1, goal_long2};

float x_vel = 0.25;
float angleDiff;
float z_angular_vel;
int gps_waypoint_index = 0;

NavigateCourse::NavigateCourse(){
  cmd_vel_pub_ = nh_.advertise<geometry_msgs::Twist>("/cmd_vel", 1);
  object_pub_ = nh_.advertise<std_msgs::Bool>("/detected_objects", 1);
  gps_sub_ = nh_.subscribe("/ublox_node/fix", 1, &NavigateCourse::gpsCallback, this);
  compass_sub_ = nh_.subscribe("/compass", 1, &NavigateCourse::compassCallback, this);
  lane_sub_ = nh_.subscribe("/lane_detected", 1, &NavigateCourse::laneCallback, this);
  scan_sub_ = nh_.subscribe("/scan_filtered", 1, &NavigateCourse::scanCallback, this);
}

// Save lane status callback
void NavigateCourse::laneCallback(const std_msgs::Bool& msg){
   lane_status_ = msg.data;
}

// calculate angular velocities to next waypoint
void NavigateCourse::compassCallback(const sensor_msgs::MagneticField& msg){
    current_yaw_ = msg.magnetic_field.z;
    if(!lane_status_){
        angleDiff = (int)(bearing_ - current_yaw_) % 360;
        z_angular_vel = -angleDiff*(2.5/180)*1.6;
        geometry_msgs::Twist cmd;
        cmd.linear.x = x_vel;
```

```
        if (z_angular_vel > 2.5) z_angular_vel = 2.5;
        if (z_angular_vel < -2.5) z_angular_vel = -2.5;
        cmd.angular.z = z_angular_vel;
        cmd_vel_pub_.publish(cmd);
  }
}

float NavigateCourse::degreesToRadians(float angle){
  return angle * (PI/180.0f);
}

float NavigateCourse::RadiansToDegrees(float angle){
  return angle * (180.0f/PI);
}

// calculate bearing between current GPS reading to next waypoint
float NavigateCourse::bearing(float lat, float lon, float lat2, float lon2){
    float latrad1 = degreesToRadians(lat);
    float latrad2 = degreesToRadians(lat2);
    float delta1 = degreesToRadians(lat2-lat);
    float delta2 = degreesToRadians(lon2-lon);

    float y = sin(delta2) * cos(latrad2);
    float x = cos(latrad1)*sin(latrad2) - sin(latrad1)*cos(latrad2)*cos(delta2);
    float brng = atan2(y,x);
    brng = RadiansToDegrees(brng);
    return brng;
}

// calculate distance between current GPS reading to next waypoint
float NavigateCourse::distance(float lat, float lon, float goalLat, float goalLon){
    float lat1 = degreesToRadians(lat);
    float long1 = degreesToRadians(lon);
    float lat2 = degreesToRadians(goalLat);
    float long2 = degreesToRadians(goalLon);

    // Haversine Formula
    long double dlong = long2 - long1;
    long double dlat = lat2 - lat1;

    float ans = pow(sin(dlat / 2), 2) +
                     cos(lat1) * cos(lat2) *
                     pow(sin(dlong / 2), 2);
```

```
    ans = 2 * asin(sqrt(ans));

    // Radius of Earth in
    // Meters, R = 6371
    long double R = 6371000;

    // Calculate the result
    ans = ans * R;

    return ans;
}

void NavigateCourse::gpsCallback(const sensor_msgs::NavSatFix& msg){
  current_gps_ = msg;
  if(!lane_status_){
    float current_distance = distance(msg.latitude, msg.longitude,
goals_lat[gps_waypoint_index], goals_long[gps_waypoint_index]);
    std::cout << "distance: " << current_distance << std::endl;
    bearing_ = bearing(msg.latitude, msg.longitude, goals_lat[gps_waypoint_index],
goals_long[gps_waypoint_index]);

   // if less than 3 meter from goal stop
    if(current_distance < 3.0){
      gps_waypoint_index ++; // Advance gps waypoint
      set_bearing_ = false;
      std::cout << "waypoint #: " << gps_waypoint_index << std::endl;
      if(gps_waypoint_index > 1)
        x_vel = 0.0;
    }
    else{
      angleDiff = (int)(bearing_ - current_yaw_) % 360;
      if(angleDiff < -180) angleDiff += 360;
      std::cout << "yaw: " << current_yaw_ << std::endl;
      std::cout << "angle: " << angleDiff << std::endl;
    }
  }
  else{
    std::cout << "Lane detected, no GPS waypoint navigation" << std::endl;
  }
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "waypoint_node");
```

```
  NavigateCourse nav;
  ros::spin();
  return 0;
}
```