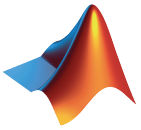
WPI



MathWorks®

Collaboratively Navigating Autonomous Systems

A Major Qualifying Project Report Submitted to the Faculty of the
WORCESTER POLYTECHNIC INSTITUTE
In partial fulfillment of the requirements for the
Degree of Bachelor of Science
By

Madeline Burris

Ryan Fawthrop

Kevin Janesch

Edward Murphy

Quinn Perry

Project Number: AW1 CNAS
Date: 05/01/2014

Sponsoring Organization:

The MathWorks

Project Advisors:

Professor Alexander M. Wyglinski

Professor Taskin Padir

Abstract

The objective of this project is to focus on technologies for enabling heterogeneous networks of autonomous vehicles to cooperate together on a specific task. The prototyped test bed consists of a retrofitted electric golf cart and a quadrotor designed to perform distributed information gathering to guide decision making across the entire test bed. The system prototype demonstrates several aspects of this technology and lays the groundwork for future projects in this area.

Acknowledgements

Professor Alexander M. Wyglinski, Ph.D.

Professor Taskin Padir, Ph.D.

MathWorks

Bob Boisse

Elizabeth Tomaszewski and WPI Facilities

WPI Robotics Learning Center

Tracey Coetzee

Cathy Emmerton

Joe St. Germain

Marleny Ortiz

Executive Summary

Communication between autonomous vehicles is becoming increasingly relevant as the number of robots in operation increases. The goal of this project is to create a platform system of autonomous robots, pooling information about their environment and using it as the basis for navigation decisions. This system will serve as a test platform to explore the benefits of collaborative decision-making across the distributed robotic agents. There are three main problems that this project will address:

1. Establishing communication between the distributed agents that make up the system
2. Contribution of sensory information from each distributed agent as to create a map of the environment.
3. Using the environmental map to make the best navigation decisions.

In order to create this test platform system, the team pinpointed three main objectives that must be met.

- The first objective consists of creating a physical ground vehicle capable of taking commands from a control system, and turning them into physical actions.
- The second objective is to create a collection of software that is capable of pooling environmental data, processing it, and providing navigation commands to the rest of the system.
- The third objective is to create an aerial platform that will send supplemental environmental information back the decision making engine.

In order to create this autonomous system, an electric golf cart will be used for the unmanned ground vehicle (UGV), and AR Parrot Drones will be used for the unmanned aerial vehicles (UAV). The ground vehicle will navigate by performing visual simultaneous localization and mapping (SLAM), using forward-facing stereo cameras to map the environment. The UAVs will have onboard cameras, collecting images of the environment and sending them back to the UGV. The UAVs will also send their positional data back to the ground vehicle. Once the images and positional information are received on the UGV, blob detection is applied so that obstacles, and their locations, can be identified. These obstacles are then added to the global map of the environment.

The electric golf cart had to be retrofitted as to incorporate drive-by-wire control of its functions. Linear actuators were used to control the steering and the braking of the golf cart. With feedback positioning, the computer knows how far the actuator arms are extended at all times. Since the throttle of the golf cart is controlled by a continuously variable potentiometer that is adjusted by the accelerator pedal, the team chose to use a digital potentiometer to emulate it.

The UGV, UAV, and computing systems are designed to be modular so that the individual aspects of the project can be improved separately, replaced, or upgraded as needed without requiring restructuring of the entire system.

This autonomous system uses a variety of software that utilizes MATLAB and ROS. An open source software package called ROS-MATLAB Bridge was used to allow MATLAB to take advantage of the publisher subscriber communication system that ROS creates. MATLAB was used for blob detection and control of the system. ROS was used to provide a standard system to transmit data between programs that can be written in C++, Python or MATLAB.

Open source software packages were used to try to move the project along, but due to a lack of documentation and conflicts between software the implementation of these open source packages proved difficult. By the projects conclusion, SLAM was implemented and worked with a Microsoft Kinect, but not the original Raspberry Pi cameras. The golf cart was controlled through MATLAB, and the cameras were able to detect orange blobs.

Table of Contents

Abstract i
Acknowledgements ii
Executive Summary iii
Table of Contents vi
List of Figures ix
List of Tables xiii
List of Acronyms xiv
1. Introduction 1
1.1 Motivation 1
1.2 Formation and Coordination of Autonomous Agents 3
1.3 Technical Challenges 6
1.4 Project Objectives 7
1.5 Report Structure 10
2 Current State of the Art 11
2.1 The DARPA Grand Challenge 11
2.2 Current Autonomous Vehicles 12
2.3 Chapter Summary 18
3 Principles of Autonomous Vehicles 19
3.1 External Sensing 20
3.2 Internal Sensing 31
3.3 Data Fusion and Path Planning 33
3.3.1 Simultaneous Localization and Mapping 33
3.3.2 Kalman Filters 35
3.3.3 Path Planning 37
3.4 Ground Vehicle Mechanics 39
3.4.1 Steering 39
3.4.2 Braking 42
3.4.3 Actuation 43
3.5 Aerial Platforms 45
3.5.1 Aerial Vehicle Motion 47
3.6 Chapter Summary 49

4 Proposed Implementation 51
4.1 Design Specifications 53
4.1.1 Experiment Specifications 53
4.1.2 Information and Sensor Connectivity 54
4.1.3 Power Management Constraints 56
4.2 Proposed Design 57
4.2.1 Aerial Platform 57
4.2.2 Ground Platform 58
4.2.3 Decision Making and Control Logic 60
4.3 Project Logistics 62
4.3.1 Financial Considerations 64
4.3.2 Time Considerations 65
4.3.3 Workspace 66
4.3.4 Safety 67
4.4 Chapter Summary 69
5 Ground Vehicle Hardware 70
5.1 Steering 70
5.1.1 Actuator Selection 71
5.1.2 Steering Mount Prototypes 74
5.1.3 Final Connection Design 77
5.2 Braking 80
5.3 Throttle 83
5.3.1 Throttle Control Testing 84
5.3.2 Throttle Switch 85
5.4 Safety 87
5.5 Chapter Summary 90
6 Ground Vehicle Software 91
6.1 Vehicle Control 91
6.2 Inertial Navigation 93
6.3 Stereo Vision and SLAM 95
6.4 Path Planning 104
6.5 ROS MATLAB Communication 105

6.6 Chapter Summary 106
7 Aerial Implementation 108
7.1 Drone Communication with ROS 108
7.2 Aerial Navigation 111
7.3 Cone Identification 121
7.4 Experimental Results 124
7.5 Chapter summary 125
8 Conclusions and Future Work 126
8.1 Future Work 127
9 Bibliography 131
Appendix A: Purchase List 135
Appendix B: MATLAB Classes for ROS-MATLAB Bridge 138
Appendix C: Drone Launch File 145
Appendix D: Python Drone Controller 147
Appendix E: Python Drone Navigation Data Extraction 150
Appendix F: Python Drone Flight Controller 151
Appendix G: MATLAB Drone Navigation 154
Appendix H: MATLAB Drone Blob Detection 157

List of Figures

Figure 1.1: An illustration of Moore's Law showing how the number of transistors per die increase at a steady logarithmic pace over time [2]. 2
Figure 1.2 : Diagram depicting the communication systems proposed by the DOT RITA. This is the layout that RITA plans to use to start in the standardization of autonomous vehicles across manufactures [6]. 4
Figure 1.3: California PATH platooning demonstration in San Diego in 1997. There are 8 Buick LeSabres driving in a platoon, each utilizing V2V communications [8]. 6
Figure 1.4: A diagram of the team's proposed system design. 9
Figure 2.2: The Oshkosh TerraMax Vehicle. This vehicle was designed to navigate extreme terrain, and comes equipped with a Velodyne HDL-64e Lidar module. The TerraMax also has wheels that can automatically change tire pressure to best suit its driving environment. This vehicle is capable of autonomous convoy driving [10]. 13
Figure 2.3: The 3D map of what the onboard computer of the Google Car sees. This is a combination of sensory information from its Lidar data overlaid on an in-depth map of the area [12]. 14
Figure 2.4: The Google Car equipped with the Velodyne HDL-64e Lidar Module [13]. 15
Figure 2.5: RQ-2A Pioneer Unmanned Aerial Vehicle 16
Figure 2.6: General Atomics MQ-1 Predator Drone 17
Figure 2.7: Amazon looks to use quadcopter drones to deliver packages in the near future. 17
Figure 3.1: This is a concept diagram for a generic independent autonomous agent. This diagram assumes that all of the agent's data collection and processing is done onboard, and that the agent does not collaborate with other autonomous agents. 19
Figure 3.3: This speckle pattern is an example from one of PrimeSense's patent applications [29] for 3D mapping using structured light. This pattern would be repeated when broadcast on a surface. Annotation 92 denotes a light-refracting bump. 27
Figure 3.4: Velodyne HDL-64E Lidar module, with cover (left) [35] and without (right) [36]. The top cylinder includes the laser emitters and receivers, and rotates at up to 900 rpm to acquire fifteen full 360 degree, 86,000-point scans per second. 29
Figure 3.5: Diagram of the radar systems in place in the Mercedes-Benz Intelligent Drive platform [37]. The orange swaths in the image indicate the areas covered by different radar modules. 30
Figure 3.6: Flow chart of how a SLAM algorithm can work. 34
Figure 3.8: Shows the new covariance (in green) after a measurement update. Illustrates that the confidence in the new estimate is greater than that of the prediction or the noisy measurement [39]. 36
Figure 3.9: A simple example of how A* works. This shows that A* can be get complicated when the space to search becomes very large [42]. 38
Figure 3.10: Ackerman Steering Instant Center of Rotation 40
Figure 3.11: Robot with parallel wheels and infinite Instant Centers of Rotation 41
Figure 3.12: Schematic of drum braking systems. The brake show is pushed outward, applying pressure to the interior wall of the brake drum, slowing the spinning of the drum [39]. 42

Figure 3.13: For disk brake systems, the calipers squeeze the brake pads onto the revolving disk, slowing the disk to a stop [39]. 43
Figure 3.14: Rack and pinion mechanical linear actuator. This actuator coverts the rotational motion enacted on the steering wheel into linear motion along the steering rack [42]. 44
Figure 3.15: The internal componenets of an electromechanical linear actuator. The rotational motion of the motor turns a gear, which rotates a screw. The action of the screw forcing itself through a fixed nut results in the linear motion of the actuator arm [43]. 45
Figure 3.16: The Asctec Hummingbird quadcopter 45
Figure 3.17: The Arducopter as built by a hobbyist . The body of this quadcopter was sold separately at the time of quadcopter selection. 46
Figure 3.18: The AR Parrot Drone 2.0 with the indoor shell equipped. 47
Figure 3.19: Six degrees of freedom in relation to aerial vehicles. This figure represents x as forward and back, y as right and left, and z as up and down. 47
Figure 3.20: Quadrotor torque generation and resulting motion. 48
Figure 4.1: The process a robot follows when navigating in an unknown environment. 51
Figure 4.2: Final cart, with electro-mechanical and safety systems installed. The drive-by-wire and computer controlled steering systems are used for motion control, and the cameras and sensors get mounted on the front to realize the real world environment. 52
Figure 4.4: Proposed project timeline for the first quarter of the project. 63
Figure 4.5: The proposed project timeline for the second quarter of this project. 63
Figure 4.6: Proposed project timeline for the final two quarters of this project. The final quarter was designated specifically for paper writing, and little to no technical work was to be done. ... 64
Figure 4.7: Photograph of one of the emergency stop switches installed in the vehicle. When the switch is activated, the powertrain of the vehicle is disabled, and the brakes are engaged to prevent injury. 68
Figure 5.1: View of the mounting point used to control the steering using the linear actuator. In connecting the linear actuator here, the original steering can remain installed. 71
Figure 5.2: Diagram of the large linear actuator the team intended to use for the steering (in inches). 71
Figure 5.3: The 560 lb linear actuator being held underneath the front bumper, with the actuator arm nearing the mounting point. The actuator barely fits and since there still would need to be a connection piece from the actuator to the steering rack, was deemed too large for use. 72
Figure 5.4: Diagram of the smaller actuator that was used to control the steering in the final revision (dimensions in inches). 73
Figure 5.6: Front view of the location that the linear actuator was mounted to. 75
Figure 5.10: A prototype revision of the steering mount and tie-rod. From left to right, the parts include the new female-threaded ball joint, the threaded stock connector, the inline ball joint, the stock that connects to the linear actuator, and the linear actuator. 77
Figure 5.11: The last prototype steering assembly installed. One more revision was made after this, in favor of more durable components, and less angles. 78
Figure 5.14: Actuator fully extended. The angle of the connection between the steering rack and the linear actuator has been reduced from the previous 80

Figure 5.15: The braking mechanism in place on the cart. This shows the device that actuates the cable, applying the brakes. 81
Figure 5.16: View of under the cart where the linear actuator is mounted for the braking. 82
Figure 5.17: Sabertooth Motor Controller circuit. The blue, yellow, and grey (representing white) potentiometer reference wires let the computer know the position of the actuator arms. For safety reasons, there is a closed emergency switch between the battery and the Sabertooth. 83
Figure 5.18: Wiring schematic of the golf cart without any modification from the group. This shows how the powertrain is connected to the battery system of the vehicle, and the location of the key switch in the wiring. 84
Figure 5.19: The throttle switch in "manual control" state. The original continuous potentiometer is in place. 86
Figure 5.20: The throttle switch in the "digital control" state. The new connections to the digital potentiometer are in place, and the pins 2 and 3 are tied together on the motor controller. 87
Figure 5.21: Schematic of the existing electrical system in place on the golf cart. Highlighted are the key switch and the solenoid. The emergency switch system was implemented in series with the key switch, so that when the switches were depressed, the power would be cut to the cart's drivetrain. 88
Figure 5.22: Emergency switch located at the front of the vehicle. 89
Figure 5.23: Emergency switch located in the footwell of the vehicle, for easy access to the driver. The button could be depressed by the driver's foot in emergency situations, allowing them to keep their hands safety within the vehicle. 89
Figure 5.24: Emergency switch at the rear of the vehicle. 90
Figure 6.1: Changes made to the compiler flags in rosserial_arduino/src/ros_lib/ArduinoHardware.h to accommodate the MK20DX128 (Teensy 3.0) and MK20DX256 (Teensy 3.1). These changes were submitted as pull request 90 on GitHub, and merged into the ros-drivers:hydro-devel branch of the project [50]. 91
Figure 6.2: Vehicle controller software organization and inputs/outputs. The calculated vehicle speed and throttle control loop were not implemented in this iteration of the project due to time constraints. 92
Figure 6.3: Pololu MinIMU-9 v2 IMU, with pinout and axes. The LSM303DLHC and L3GD20 are interfaced to the Teensy via I^2C. 93
Figure 7.1: Data transfer from the drone to ROS, through ROS-MATLAB Bridge, to MATLAB and vice versa. 111
Figure 7.2: Drone at a standard hover position 112
Figure 7.3: Abnormally low hover height caused by reflection of ultrasonic pings and non-solid surfaces. 113
Figure 7.4: Low battery can cause significant drift and launch errors. 113
Figure 7.5: Drone rolled to right and flying uncontrollably. 114
Figure 7.6: The drone is using ar_pose to localize to the black screen of the computer seen in the lower left hand corner of the image. A pose is returned giving the drone's position in relation to this screen. 117
Figure 7.7: A PD control loop uses the error of a system to determine the appropriate movements to reach a desired position. 119

Figure 7.8: Example code for the proportional derivative control loop utilized for waypoint navigation. 120
Figure 7.9: The drones are programmed to move via strafing. This means the drones will move diagonally to an end position and will always have the same heading. 121
Figure 7.10: The image data processed by the custom blob detection function displays only orange blobs currently within the view of the camera. 123

List of Tables

Table 3.1: Comparison of hobbyist-grade IMU modules to a commercial, automotive-grade module. Hobbyist-grade IMUs typically perform a power-on calibration, but noise and overall calibration need to be performed by the user. Often, noise cancels out additional precision from additional ADC bits. 21
Table 3.2: Comparison of different Lidar modules currently available. 28
Table 4.1: Specifications of different wireless standards. 55
Table 5.1: Table of different available actuator lengths. The 150 lb actuator with a stroke size of 4 inches is 9.51 inches in length when fully retracted, a whole 8 inches shorter than the 560 lb actuator. The actuator with a stroke size of 8 inches is 13.51 inches when fully retracted, 4 inches shorter than the 560 lb actuator [44]. 73
Table 7.1: Data available through the ardrone_autonomy package from the AR Drones as seen in the documentation. 109

List of Acronyms

Acronym	Definition	Context
AHRS	Attitude and Heading Reference System	Sensing
API	Application Programming Interface	Software programming
CAN	Controller Area Network	Device communications
CCD	Charge-Coupled Device	Sensor design
CMOS	Complementary Metal-Oxide Semiconductor	Sensor design
CNAS	Collaboratively Navigating Autonomous Systems	Project shorthand
CPU	Central Processing Unit	Computing, hardware
DARPA	Defense Advanced Research Projects Agency	Organization
DIY	Do It Yourself	Activity
DRC	DARPA Robotics Challenge	Event
EKF	Extended Kalman Filter	Computing, algorithm
FOV	Field of View	Sensor attribute
GPS	Global Positioning System	Sensing
GPU	Graphics Processing Unit	Computing, hardware
GUI	Graphical User Interface	Computing, software
I2C, I^2C	Inter-IC Communication	Device communications
IC	Integrated Circuit	Computing, hardware
IEEE	Institute of Electrical and Electronics Engineers	Organization
IMU	Inertial Measurement Unit	Sensing
IPC	Inter-Process Communication	Software
IR	Infra-red	Device communications
JPEG	Joint Photographic Experts Group	File format
LAN	Local Area Network	Distributed systems
MATLAB	Matrix Laboratory	Software
MEMS	Microelectromechanical Systems	Sensing
MMAL	Multi-Media Abstraction Layer	Software
MQP	Major Qualifying Project	Project shorthand

NATO	North Atlantic Treaty Organization	Organization
NDA	Non-Disclosure Agreement	Information
NMEA	National Marine Electronics Association	Organization
NTP	Network Time Protocol	Software
OS	Operating System	Software
PCL	Point Cloud Library	Software, library
PID	Proportional Integral Derivative	Computing, algorithm
PWM	Pulse-Width Modulation	Device communications
RAM	Random Access Memory	Computing, hardware
RLG	Ring Laser Gyroscope	Sensing
RPF	Raspberry Pi Foundation	Organization
ROS	Robot Operating System	Software, tool
SDK	Software Development Kit	Software, tool
SDR	Software-Defined Radio	Device communications
SLAM	Simultaneous Localization and Mapping	Software, algorithm
SLI™	Scalable Link Interface (Nvidia trademark)	Device communications
SPI	Serial Parallel Interface	Device communications
SSD	Solid-State Disk/Drive	Computing, hardware
SVN	Subversion	Software, version control
UAV	Unmanned Aerial Vehicle	System
UGV	Unmanned Ground Vehicle	System
USB	Universal Serial Bus	Device communications
V2I	Vehicle-to-Infrastructure	System topography
V2V	Vehicle-to-Vehicle	System topography
VGA	Video Graphics Array	Computing, hardware
WPI	Worcester Polytechnic Institute	Organization

1. Introduction

Robotics, unmanned systems, and automatic decision-making are becoming an integral component of the future of society. While the number of robots in daily operation increases, anywhere from manufacturing, on the roads, or in our homes, the need for robotic collaboration becomes increasingly apparent. As the industry gains popularity, research must be done on the methods and effectiveness of homogeneous collaboration among autonomous robotic systems.

1.1 Motivation

The current generation of robotics is based around higher-level computer decision-making. In conventional automation, the task to be performed would be completely predefined, every movement that the robot would make calculated through the programming code. As Bill Gates, Co-founder of Microsoft, stated in Scientific American, "One trend that has helped [robotics] is the increasing availability of tremendous amounts of [processing] power. One megahertz of processing power, which cost more than $7,000 in 1970, can now be purchased for just pennies" [1]. He goes on to explain how the costs of data storage and hardware have declined in a similar fashion. "As computing capacity continues to expand, robot designers will have the processing power they need to tackle issues of ever greater complexity" [1]. Due to the increase in the available amount of computing horsepower per dollar, more computationally expensive tasks have become achievable.

In 1965, Intel co-founder Gordon E. Moore made a prediction known as Moore's Law. He predicted that the number of transistors on integrated circuits would double approximately every two years [2]. This means that the performance of these integrated circuits will double every two years with respect to processing speed and memory capacity. This prediction has been used as a target for the computer sector for decades, defining growth and research and

development effects within this industry [2]. In fact, this prediction has held true for approximately 50 years, although it is anticipated to slow in the near future as the semiconductor devices are reaching their physical limits.

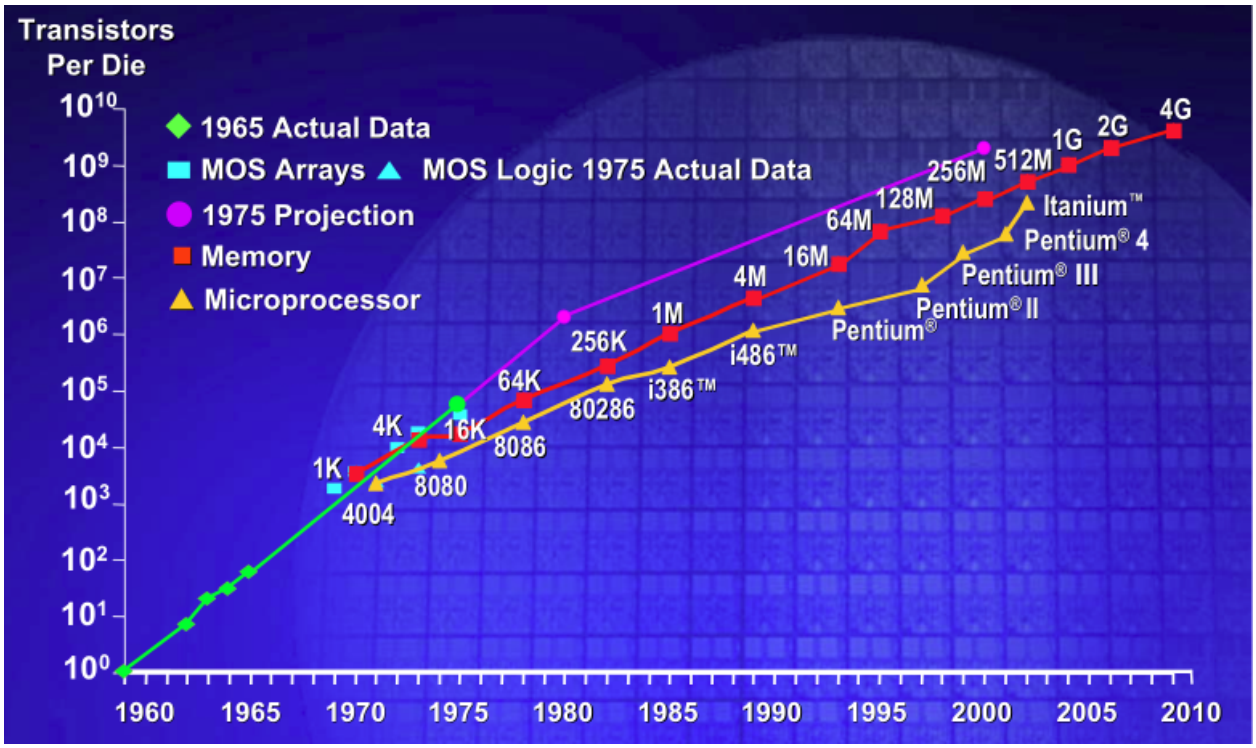


Figure 1.1: An illustration of Moore's Law showing how the number of transistors per die increase at a steady logarithmic pace over time [2].

The constant increase of computer processing power has directly impacted the robotic sector providing new possibilities for robot integration into the daily life of society. One area that has been significantly impacted by computer processing technology are autonomous vehicles. Due to the miniscule cost of processing power, nearly every major car manufacturer has an option for assistive braking, lane drift prevention, obstacle recognition, or any combination of these types of semi-autonomous features. Nissan has plans to sell a self-driven car in the market by the year 2020 [3], and the company currently has a test vehicle that is road legal in Japan. It utilizes a semi-autonomous system called the 'Advanced Driver Assist System',

which enables the vehicle to stop at red lights, slow down for traffic congestion, and even take an exit on a highway [4].

1.2 Formation and Coordination of Autonomous Agents

Once electronic assist features, like Nissan's 'Advanced Driver Assist System', become standard on the majority of automobiles on the road, the next step will be for the separate robotic systems to communicate data to one another, in order to better make decisions about the environment. A pilot project by University of Michigan is testing vehicle-to-vehicle (V2V) communications devices that broadcast location and speed information to other vehicles while gathering the same data [5]. This allows the car to notify the driver of potential threats, paving the way for pre-emptive safety measures in automobiles [5]. For example, if a driver encounters an obstacle in the road and brakes, the car would indicate to the vehicles in its vicinity that it is drastically slowing down, and the other drivers could react based on this information.

In addition to V2V communications, vehicle-to-infrastructure (V2I) communications allow cars to receive information from the road itself, providing warnings of accidents, cars in blind spots, school zones, etc [5]. This combination of information from other vehicles and the actual road itself will prove to be very helpful to drivers, but also offer autonomous cars "an abundance of trustworthy, useful information" [5]. This adds a new dimension of decision-making possibilities for the systems to deal with, but also will allow a well-designed system to operate much more safely. So, in the event of a car applying emergency braking to avoid collision with an obstacle, other cars approaching that location would slow down and move out of the path of the hazard. This could mitigate much of the cause of traffic, uniformed drives acting in a reactive scenario, as opposed to making informed decisions and planning to avoid trouble before it is reached.

The US Department of Transportation (DOT) Research and Innovative Technology Administration (RITA) recognizes the importance of connected vehicle technology, and is in the process of researching and defining technology standards to be implemented on vehicles sold commercially. The DOT has defined several applications that the connected vehicle technology must address: safety applications, mobility applications, and environmental applications [6].

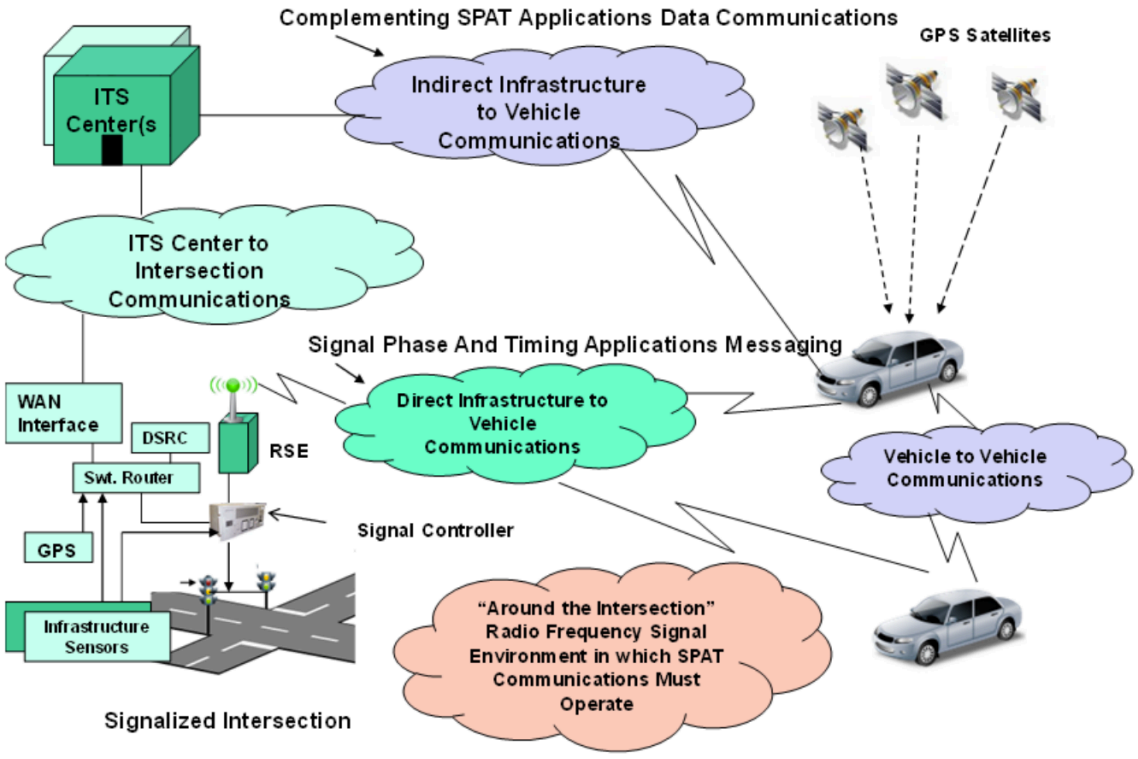


Figure 1.2 : Diagram depicting the communication systems proposed by the DOT RITA. This is the layout that RITA plans to use to start in the standardization of autonomous vehicles across manufactures [6].

According to the RITA, "connected vehicle safety applications are designed to increase situational awareness and reduce or eliminate crashes through vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2I) data transmission that supports: driver advisories, driver warnings, and vehicle and/or infrastructure controls." [6]. RITA also claims that these systems will have the ability to prevent up to 82 percent of accidents involving unimpaired drivers annually.

"Connected vehicle environmental applications both generate and capture environmentally relevant real-time transportation data and use this data to create actionable information" where this data is then used to make routing decisions that have the minimum environmental impact [6]. For example, this would prevent a traveler from driving into a traffic jam, but rather reroute them on the most efficient route around the congestion.

These technological advances in communication are essential steps towards the realization of a world of autonomous cars. A platooning project called SARTRE (Safe Road Trains for the Environment) has begun testing on roads in Europe using Volvo's specialized autonomous vehicles. Platooning is a system of linked vehicles that drive themselves based on their V2V connections to each other and the human-operated lead vehicle. The purpose of this platoon is to allow the human operator in each autonomous vehicle to be relived the task of driving, and to benefit the fuel economy of the vehicles by limiting the amount braking and acceleration seen in human operation [7].

The SARTRE Project is not the first of its kind, however. In 1997 California Partners for Advanced TecHnology (PATH) successfully demonstrated a platoon of 8 Buick LeSabres in San Diego, CA. By utilizing radar and radio V2V communication, the cars are able to coordinate maneuvers such as changing lanes, adjusting car spacing, accelerating and decelerating, and even changing positions in the platoon. California PATH reported that, "the platoon scenario at Demo '97 in San Diego did not include the full range of functions that would be needed for an operational automated highway system…" [8]. They set the groundwork for future platooning projects, and as technology advances, realizing these unmet goals become easier to realize.

Figure 1.3: California PATH platooning demonstration in San Diego in 1997. There are 8 Buick LeSabres driving in a platoon, each utilizing V2V communications [8].

1.3 Technical Challenges

The goal of this project is to create a platform made up of a system of autonomous robots, which utilize sensors to collect information about the environment and contribute that information to a central data pool. This pool of data is then used for making navigation decisions within the system. For the rest of this paper, the autonomous robots within the networked system will be referred to as distributed agents. This project will contribute a robotic system to serve as a test platform that enables distributed decision making across multiple homogeneous agents, ultimately allowing for testing of electronic security, electromagnetic security, and sensor security attacks on networked systems of distributed robotic agents.

- **Challenge 1:** Communication between agents. The most important aspect of this system is the communication between the distributed agents. This

communication is key to the transmission of sensor and location data from the distributed agents to the decision making engine. For systems like this one, it is important to explore requirements for the communication system, and design that system to be robust under maximum load.

- **Challenge 2:** Realizing the environment. Once communication is established between all of the distributed agents, the agents need to contribute their onboard sensor information to the system's environmental map. This map is used to make decisions about the actions that the system takes.
- **Challenge 3:** Acting upon the environment. Using the environmental map created by the fusion of all the sensor information within the system, the autonomous system must make navigation decisions to safely reach a desired destination. The system must weigh all the possible options, and choose the one that is safest and most efficient.

1.4 Project Objectives

This project is broken up into three main project objectives:

- **Objective 1:** Cyber-physical System for Autonomous Ground Vehicle. This portion of the project consists of the creation of the physical ground vehicle. The vehicle must be capable of operation in the defined test environment, and be able to take commands from a control system, and turn those commands into physical action.
- **Objective 2:** Navigation, Path Planning, and Data Fusion. This portion is the brain of the project. This collection of software must be capable of collecting data about the operating environment, process that data, and ultimately provide

navigation commands to the rest of the system so that physical actions can be performed.

- **Objective 3:** Aerial Platform for Expanded Environmental Awareness**.** This is the portion of the project that makes it stand out from other modern autonomous systems. The ground vehicle sensor information will be supplemented by auxiliary data, sent back to the decision engine by forwardly deployed aerial agents. These agents are responsible for contributing information to the environmental map that would otherwise go unrealized by the ground vehicle.

The end result of this project will be an autonomous system consisting of two distributed agents: one autonomous unmanned ground vehicle (UGV) and one autonomous unmanned air vehicles (UAV). This system will have the ability to navigate safely to a designated end position, while avoiding obstacles and detecting hazards. Data will be passed between the vehicles (UGV to UAVs and back) while performing the navigation and obstacle avoidance. This V2V communication between these two distributed agents demonstrates how the transmission of data is scalable, and that smarter and more efficient decisions can be made off of a more diverse pool of data. The distributed agents will be aware of their location in relation to the environment around them, while also tracking the locations of all the agents in the system. This design simulates the functionality of a system, which may be used, in the future, for an autonomous convoy. This will allow for a future project to use this unmanned autonomous system as a platform to begin testing the security and vulnerability concerns, which are associated with of these types systems.

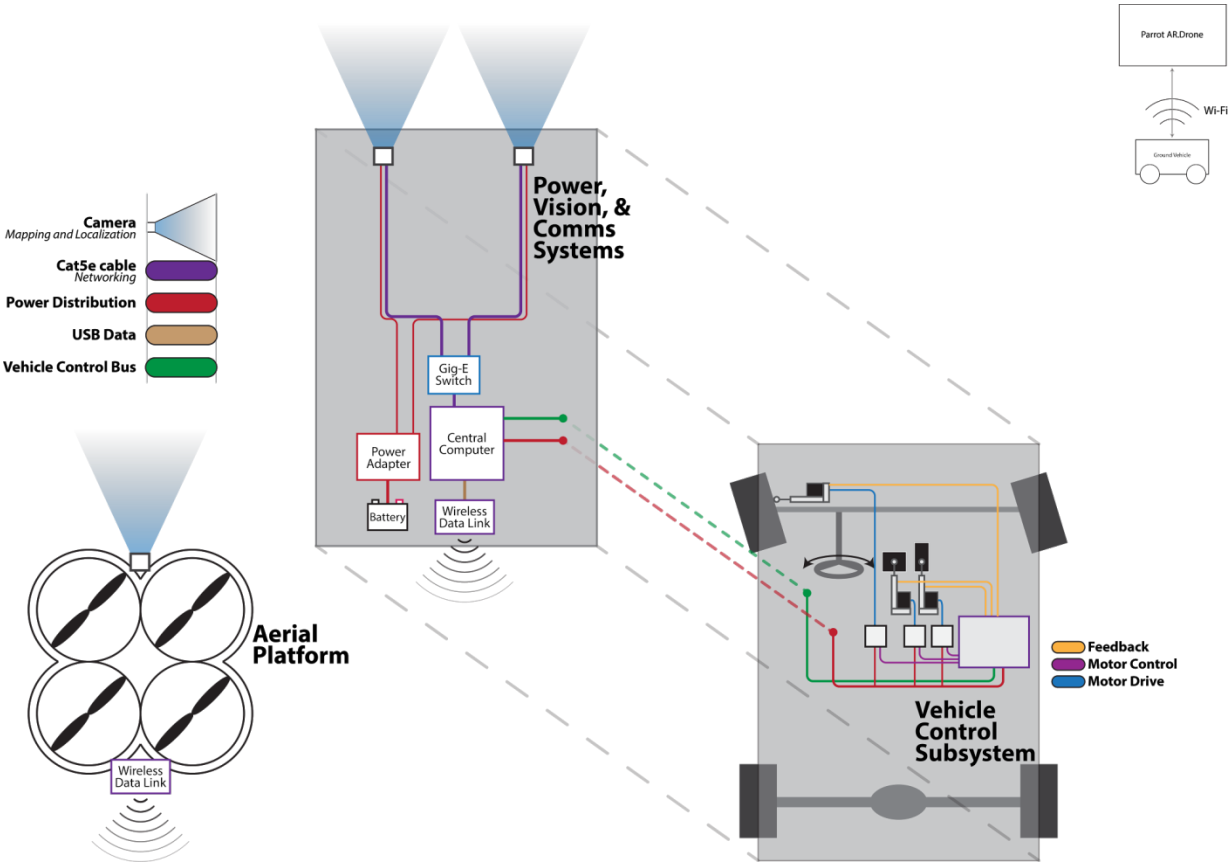


Figure 1.4: A diagram of the team's proposed system design.

In order to accomplish the goal of making a collaboratively navigating autonomous system, the project will implement a means of communications between a ground vehicle and auxiliary unmanned aerial vehicles. The ground vehicle will be able to navigate an obstacle course independently of aerial vehicles. The UAVs will send information about the environment to the ground vehicle, which the ground vehicle will then use to generate a more accurate navigation solution to its end goal.

For proof of concept, a used golf cart will be used for the ground vehicle, and AR Parrot Drones will be used for the aerial vehicles. The ground vehicle will navigate by performing visual simultaneous localization and mapping (SLAM), using forwards facing stereo cameras to map the environment. There will also be ultrasonic range finders, used as a failsafe, to prevent

collisions. The UAVs will have onboard cameras, collecting images of the environment and sending them back to the UGV. The UAVs will also send positional data back to the ground vehicle so the location can be calculated. Once the images and positional information are received on the UGV, blob detection is applied so that obstacles, and their locations, can be identified. These obstacles are then added to the global map to the environment.

The deliverables that this project contributes to the field are:

- A test bed system that demonstrates the ability for multiple homogeneous autonomous robots to share sensor information.
- A decision making engine that acts upon the shared pool of sensor data to make decisions about navigation.

1.5 Report Structure

The structure of this report is as follows: Chapter 2 will provide a detailed overview of the fundamentals involved in the design and fabrication of an autonomous system of robotic agents. Chapter 3 describes the proposed system to be delivered at the end of the project duration. Chapter 4 will delve into the details of the design for the aerial agent component of the collaboratively navigating autonomous system, while Chapter 5 breaks down the specifics of the ground agent. Chapter 6 details the logistics of the ground vehicle control systems, and Chapter 7 discusses the theory and challenges of path planning for this type of system. Chapter 8 contains a discussion and analysis of the final system delivered at the end of the product, highlighting the project team's challenges and their solutions. To wrap up this report, Chapter 9 concludes and discusses the futures uses of the deliverables contributed by the project team.

2 Current State of the Art

There is a great deal of research going into the creation and refining of autonomous ground vehicles in the world today. Much of this research is achieved through competitions that involve crowd sourcing from multiple teams. By pooling together such a wide range of talent, an accelerated progression has been made in many aspects autonomy.

2.1 The DARPA Grand Challenge

The Defense Advanced Research Projects Agency (DARPA) is a research organization of the United States Department of Defense. The DARPA Grand Challenge is a series of prize competitions that began in 2004 and focus on the development of autonomous robotics. DARPA awards the top teams of each competition with cash prizes, in hopes that they continue to develop their projects or compete in future challenges. By pooling together the work of thousands of talented engineers, great advances can be made in autonomous robotics while sparking the interest of more engineers.

The first DARPA Grand Challenge concentrated on traversing off-road terrain. Since no team was able to complete the course, the prize went unclaimed and many competed the following year in the next challenge. The DARPA Urban Challenge, held in 2007, was the third of DARPA's Grand Challenges, but the first to feature autonomous ground vehicles in a mock city environment. "The DARPA Urban Challenge is an autonomous vehicle research and development program with the goal of developing technology that will keep warfighters off the battlefield and out of harm's way" [9]. While the previous two challenges concentrated more on an off-road environment; the focus of the Urban Challenge was to obey traffic laws, detect and avoid other vehicles, and make intelligent decisions in real time.

Figure 2.1: Team Tartan Racing's Car "Boss", Winner of The DARPA Urban Challenge [57].

The goals of the program are as follows:

- Accelerate autonomous ground vehicle technology development within the areas of sensors, navigation, control algorithms, machine intelligence, and systems integration.
- Demonstrate an autonomous vehicle able to operate independently in a realistic urban environment through a field of live traffic.
- Attract and energize a wide community of participants to bring fresh insights to the problem of developing a truly robust autonomous vehicle and provide qualified performers to develop autonomous ground vehicles [for the Department of Defense].

2.2 Current Autonomous Vehicles

One vehicle, the TerraMax built by the Oshkosh Truck Corporation competed in the first two DARPA Grand Challenges and has been developed further alongside the United Department of Defense, despite its failure to win either of the competitions. The result is the Oshkosh Defense TerraMax UGV, which is a "vehicle kit system that advances perception, localization, and motion planning to… increase performance in autonomous missions" [10]. This new

technology is designed for use on any tactical wheeled vehicle, and is capable of supervised autonomous navigation in a leading or following role. Vehicles equipped with the Oshkosh TerraMax UGV become capable of forming an autonomous convoy that can adjust according to any obstructive situation. There are also features such as electronic stability control, collision mitigation braking, and adaptive cruise control, that provide for safer manual operation [10].

Figure 2.2: The Oshkosh TerraMax Vehicle. This vehicle was designed to navigate extreme terrain, and comes equipped with a Velodyne HDL-64e Lidar module. The TerraMax also has wheels that can automatically change tire pressure to best suit its driving environment. This vehicle is capable of autonomous convoy driving [10].

Google has made progress in building a fully operational autonomous car that can operate efficiently in a city environment. The Google Car is equipped with a Lidar, allowing it to gather information from the immediate environment and has a database is loaded with information about the streets in which it operates on. With this in-depth map of the roads and its visual system, it can navigate from one location to another with little to no interaction from a human operator. John Markoff of the New York Times writes, "Robot drivers react faster than humans, have 360-degree perception and do not get distracted, sleepy or intoxicated" [11]. In the far future, with this technology cars can be built with less passenger protection because they are less likely to crash, making them lighter and more fuel efficient [11].

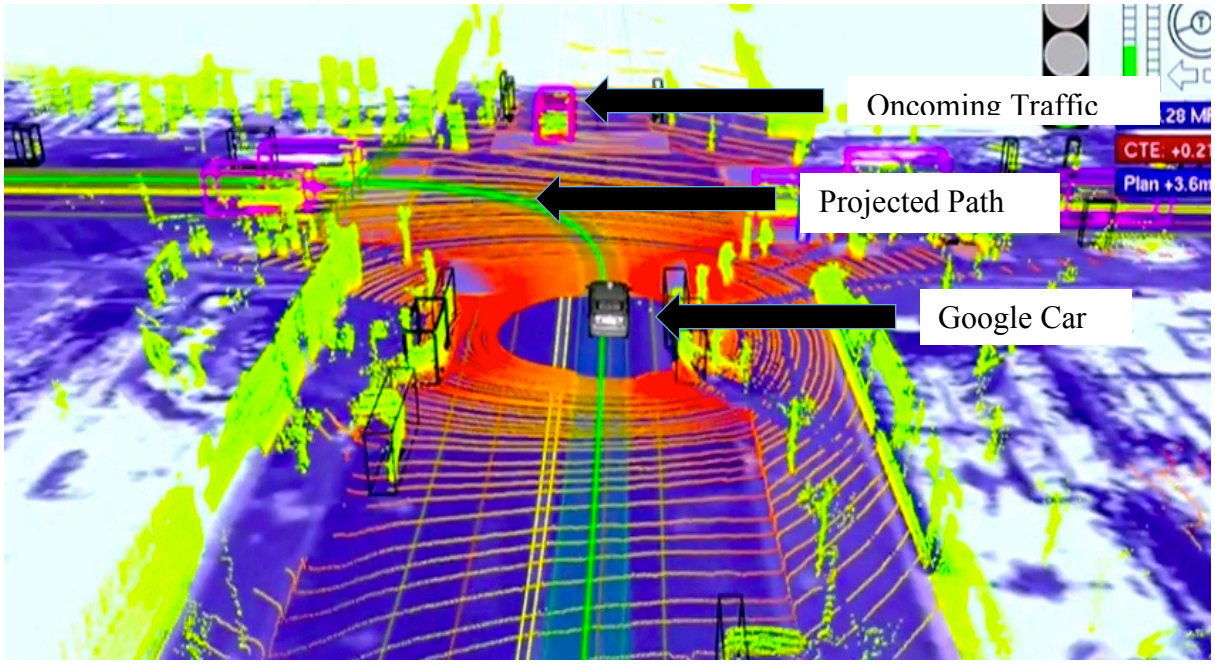


Figure 2.3: The 3D map of what the onboard computer of the Google Car sees. This is a combination of sensory information from its Lidar data overlaid on an in-depth map of the area [12].

There have been very few reports of accidents, and majority of them have been the fault of other drivers. For example, the only accident over the span of 150,000 miles in Mountain View, CA "was when one Google Car was rear-ended while stopped at a traffic light" [11]. There have also reports of accidents that have been determined to be the fault of the car, although the exact cause of the accident has been withheld. Despite this, the Google Car is a promising accomplishment in the world of autonomy. It demonstrates significant leaps and bounds in the ability to safely navigate a busy city at a rate faster than the DARPA Urban Challenge vehicles.

Figure 2.4: The Google Car equipped with the Velodyne HDL-64e Lidar Module [13].

The Artificial Vision and Intelligent Systems Laboratory (VisLab) of Parma University in Italy is a research laboratory that specializes in artificial vision. VisLab has worked with several automotive companies helping equip vehicles with a range of autonomous functions and environment perception all the way up to a fully autonomous vehicle. The VisLab ARGO autonomous vehicle was designed in 1998 and was tested on Italian highways with regular traffic for more than 2000 km, a milestone for vehicular robotics worldwide [14]. They also equipped the ENEA Surface Antarctic Robot (RAS) with cameras for vision-based sensing, and the Oshkosh TerraMax vehicle with a vision system for the DARPA Grand Challenge [14].

VisLab tested their system during the VisLab Intercontinental Autonomous Challenge. This challenge involved four autonomous vehicles travelling over 13,000 km from Parma, Italy to Shanghai, China from July 20, 2010 to October 28, 2010 with little to no human interaction [15]. This challenge differed from the DARPA Grand Challenges because it was not conducted

in a controlled environment, meaning that that the autonomous vehicles had to be ready for any real-life situations they may encounter.

Aircraft have been used for information reconnaissance since they were first introduced to combat situations in World War I [16]. Remotely piloted planes took the skies for reconnaissance shortly after in World War II [17]. During the Cold War and the Vietnam War, the United State realized a great risk of manned aerial surveillance when the U-2 vehicle was shot down in enemy territory. As a result, the US started a UAV program [18]. Since then, UAVs have become an active part of information reconnaissance. Drones such as the RQ-2A *Pioneer* Unmanned Aerial Vehicle [19], seen in Figure 2.5, and the General Atomics RQ-1 Predator drone [20], seen in Figure 2.6, are two drones, which have been used by the military autonomously for intelligence gathering.

Figure 2.5: RQ-2A Pioneer Unmanned Aerial Vehicle

These two drones have been used to scout and monitor hostile territory as well as to collect intelligence on unknown combat situations. However, drones are used in more than just combat situations. Predator drones and others like them are also used for border surveillance and to track drug trafficking. Drones have also been used to hinder poaching of elephants and rhinoceros by tracking their herds from the air [21].

Figure 2.6: General Atomics MQ-1 Predator Drone

Currently, quadcopter helicopters are used for research and educational purposes. Recommendations have been made for the use of quadrotors for a variety of purposes including search and rescue, surveillance, and package delivery. Figure 2.7 displays a quadrotor helicopter Amazon hopes to implement in the near future for home delivery.

Figure 2.7: Amazon looks to use quadcopter drones to deliver packages in the near future.

Quadrotors are at the cusp of the ability to be used for intelligence gathering. One quadrotor developed by DJI, a company out of Hong Kong, is developed for recording aerial shots for filming companies [22]. Hobbyists also use the AR Parrot Drone to film home videos.

2.3 Chapter Summary

While there are a multitude of autonomous vehicles currently in production today, there are limited technologies in place for data sharing between autonomous agents. This project will address these shortcomings, and explore the benefits of this collaborative decision-making.

3 Principles of Autonomous Vehicles

There are several challenges that must be considered when working with autonomous vehicles. These include the environment in which the vehicle will be operating, the type of vehicle, the motion constraints of the vehicle, and the sensors the vehicle will use to sense the environment and sense its movement in the environment. In addition, methods of analyzing the data collected to detect obstacles in the environment and to determine the current location of the robot in the environment must be considered. Finally, this analyzed data must then be used to make decisions about the movement of the robot within the environment. All these elements are discussed in this chapter.

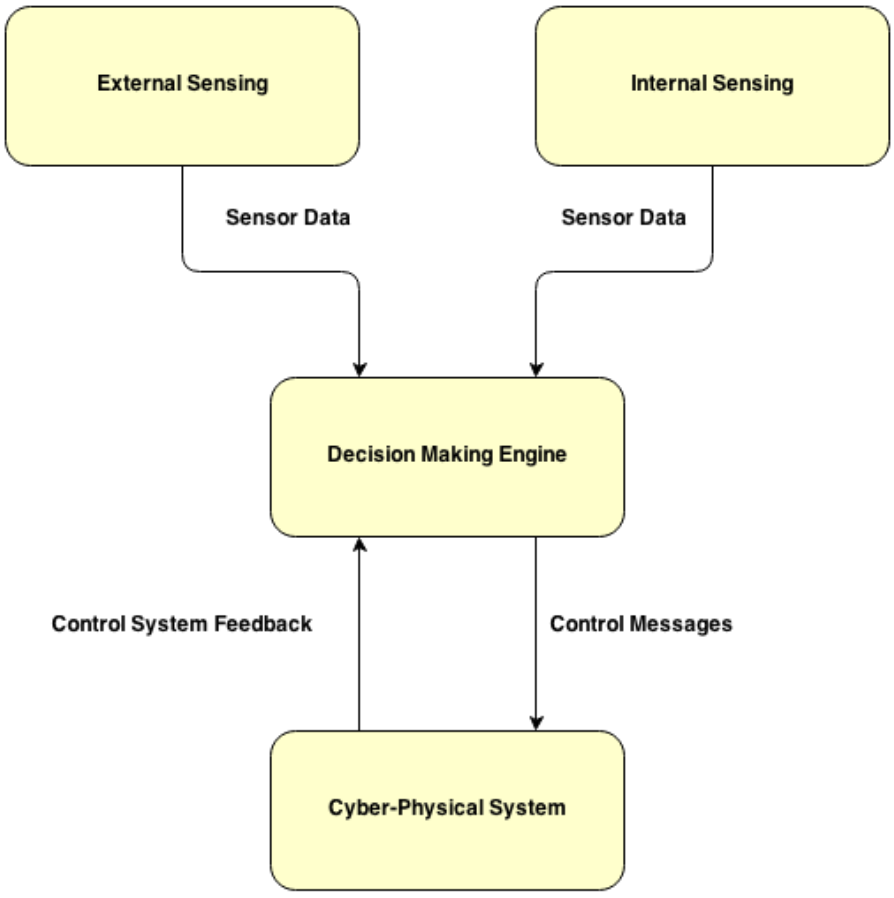


Figure 3.1: This is a concept diagram for a generic independent autonomous agent. This diagram assumes that all of the agent's data collection and processing is done onboard, and that the agent does not collaborate with other autonomous agents.

3.1 External Sensing

In order for an autonomous vehicle to navigate its environment, it must be able to sense and perceive the environment in a meaningful way. The specific tasks or environments expected to be encountered may dictate the exact choices of sensor or system organization, but the general sensor systems for autonomous vehicles involve perceiving the environment and determining what is an obstacle or objective, or assisting the vehicle in tracking its movements through the environment. Sensing can be performed either actively or passively. **Passive sensors** function as self-contained measurement instruments. They are used to perceive an aspect of the environment or the system's relation to the environment without broadcasting or emitting any form of radiation. **Active sensors** function by emitting a known signal and sensing how the signal is influenced by the environment. The most common type of active sensing is time of flight, wherein the round trip time of a known pulse or burst is measured, and is used for distance measurement and object tracking. Other approaches include measuring the change of a constant (not pulsed) signal. These include "structured light" sensors and some types of radar. Active sensors are often prone to interference in some manner, be it from uncontrollable environmental factors like weather or from one another when used multiple times on the same platform or in a given space.

An **inertial measurement unit (IMU)** can be a critical component in an autonomous system as it can be used to monitor the motion and orientation of the system where other sensors may fail to produce an accurate reading. IMUs have become popular components in consumer electronics like smartphones and video game consoles in the recent years for interacting with software through motion. An IMU can measure up to six degrees of freedom, and software can calculate change in position and orientation of a robot based upon these readings.

The six degrees of freedom refer to the three axes of linear motion along the X, Y, and Z axes, as well as rotation around each of the axes referred to as roll, pitch, and yaw. Typically, motion in these six degrees of freedom is combined over time to determine sensor position, speed, or orientation.

The current, popular iteration of this mechanism is the combination of a solid-state accelerometer and gyroscope. These systems are built as integrated circuits and referred to as MEMS ICs, and are small enough to be built into cell phones and video game equipment. By carefully integrating the angular velocity reported by the gyroscope and double-integrating the linear acceleration, accuracy of a MEMS-based IMU can be maintained for short periods of time, but the total noise can still induce drift or jitter into a reading.

Table 3.1: Comparison of hobbyist-grade IMU modules to a commercial, automotive-grade module. Hobbyist-grade IMUs typically perform a power-on calibration, but noise and overall calibration need to be performed by the user. Often, noise cancels out additional precision from additional ADC bits.

		Pololu [23] MinIMU-9 v2	SparkFun [24] 9DOF Sensor Stick	Honeywell [25] 6DF-1N6-C2-HWL
Accelerometer	Technology	MEMS (LSM303DLHC)	MEMS (ADXL345)	MEMS
	Resolution	16-bit ±2*g*, ±4*g*, ±6*g*, or ±8*g*	13-bit ±2*g*, ±4*g*, ±8*g*, or ±16*g*	>12-bit ±6
Gyroscope	Technology	MEMS (L3GD20)	MEMS (ITG-3200)	MEMS
	Resolution	16-bit ±250, ±500, or ±2000 °/s	16-bit ±2000 °/s	>12-bit ±75°/s
Magnetometer	Technology	(LSM303DLHC)	(HMC5883L)	N/A
	Resolution	16-bit ±1.3 to ±8.1 gauss	12-bit ±1 to ±8 gauss	
Sample rate		Up to 100 Hz	Up to 100 Hz	1 – 100 Hz
Connectivity		I2C	I2C	SAEJ1939 CAN
Voltage input		2.5 – 5.5V	3.3 – 16V	7 – 32V
Features		Bare board	Bare board	IP67 or IP69k weatherproofing Factory calibrated
Price		$40	$50	$1,669

One drift compensation mechanism is to incorporate a multi-axis magnetometer into both the system and the orientation calculations. A magnetometer reports the sensor's absolute position relative to magnetic north, and can provide a suitable absolute reference for Z-axis rotation to help eliminate large accumulations of gyroscope drift. Another method is to accumulate the error over an extended period of time and attempt to compensate for it in software. A third method is known as "case flipping" and involves physically inverting the IMU at a fixed interval, causing the drift to accumulate in the opposite direction.

A **camera** is one of the most relatable types of sensor for a vehicle, as human operation of a similar system is often driven by what the human could see in its surroundings. Most sensors, be they active or passive, do not determine aspects of objects like color or brightness, and thus necessitate either an environmental supplement to allow a robot to coexist with humans, or a way to interpret the world as a human would. Such an example is the self-driving car, navigating a highway or other paved road among human-controlled cars. Features like brake lights and the lines painted on the road surface would go unnoticed by nearly every other kind of sensor available. Observing an adjacent wall could indicate if the vehicle is following the road and tracking the vehicle's current speed, and distance behind the car in front of it could be used to determine whether or not the brakes need to be engaged. These alternate approaches may work in some environments, but by removing the wall or the car in front, such a system would need either additional non-visual supplements to follow the lines, adding complexity and cost, or could be replaced with a camera.

Two main types of image sensor exist in commercial products: CCD (Charge Coupled Device) and CMOS (Complementary Metal-Oxide Semiconductor). These perform the same task, but have different advantages both in their own technology and the design of an individual

sensor. Pixels in CCD image sensors are charged when struck by light, and converted to voltage and digitized one at a time across the surface of the sensor. CMOS sensors have circuitry attached to each pixel in the sensor to convert the light received to a voltage, and then these voltages are read off. CCD sensors can encounter smearing vertically and horizontally in the presence of bright light sources, affecting large areas of the image, but CMOS sensors can experience distortions in the image called "rolling shutter" if the capture circuitry is slow. Small sensors of both varieties do not receive as much light as large sensors, which negatively impacts their low-light performance, but CMOS sensors generally have better low-light response than CCD sensors.

Multiple cameras around the vehicle (or a camera on a movable platform) can not only be used to further increase the field of view of the processing equipment, but can also be used to emulate human depth perception when oriented in the same direction and separated slightly. Human depth perception works by determining the slight differences in the images received by the eyes, referred to as parallax. Comparing the features of two stereo images captured from cameras can achieve similar results, and properly calibrated and synchronized cameras can be used to capture very accurate depth data. Calculating parallax between camera frames can also be used to determine motion of the attached system within a feature-rich environment.

Stereo vision has its appeal largely due to its relatively low cost ($100 or less) for its performance as a 3D depth sensor when compared to that of active structured light mapping sensors ($1,000) or compounded Lidar modules ($10,000 and up), or in environments where the environment's ambient radiation would otherwise interfere with an active sensor's measurements. Natural-light cameras are limited to use only in daylight, however, and by using

infrared cameras and artificial illumination to accomplish the same goal, the sensor system is now classified as an active one.

Stereo vision works by using two cameras that are mounted close together. The direct line of sight of the cameras should be close to parallel. In Figure 3.2, P represents the real word

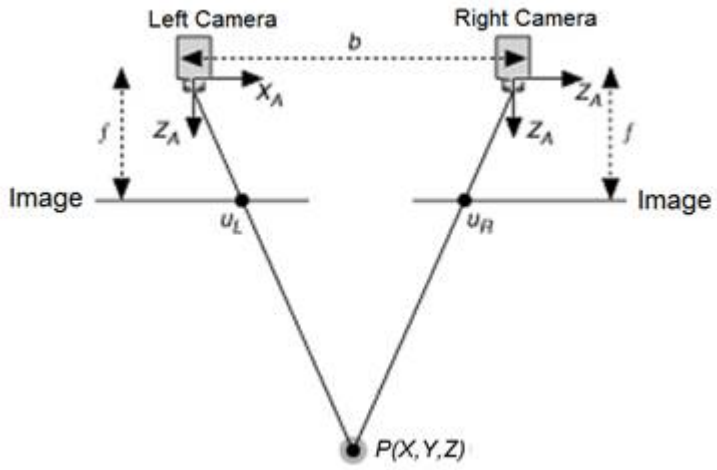


Figure 3.2: Diagram of how stereo vision works. This shows how a point from each image can be projected into 3D space [26].

point that is projected. U_L is the project of P in the left image and U_R is the projection of P in the right image. The disparity between these two points can be calculated by subtracting U_R from U_L. The depth image can be calculated by Equation 3.1, where f is the focal length of the camera, and b is the distance between the cameras.

Equation 3.1

$$Depth = f * b/disparity$$

Stereo vision can apply the depth information for use in path planning and obstacle avoidance which will help systems navigate unknown environments [26].

An **altimeter** measures the altitude of the system to which it is attached. Most digital sensing of altitude is based on a barometer, used to measure the ambient air pressure to determine altitude. These measurements are critical for airborne vehicles, as GPS-based altimeter

data can be inaccurate, and can use the accumulated Z-axis displacement as a check and source of finer altitude data if necessary. Proper sensing also requires a thermometer to measure the ambient air temperature, as the density of air changes with temperature as well as altitude.

The **global positioning system (GPS)** allows an autonomous system to determine not where in its environment it is located, but where on the surface of the planet it is located. US Air Force-maintained satellites have been placed into known and controlled orbits around the planet to facilitate this system. Each satellite is equipped with extremely precisely calibrated atomic clocks to maintain accuracy, and constantly broadcasts its time and location in its orbit to the surface, and a line of sight is required to receive it. Once a GPS receiver acquires these signals from at least four of these satellites, the time of flight (at the speed of light) of each message can be determined and the real present time extrapolated. These travel times, along with the satellites' locations, overlap in possible locations such that one point on the Earth in latitude, longitude, and altitude can be determined.

Civilian-grade GPS equipment can determine location down to the scale of meters, but some noise can occur and is expected between measurements, depending on the precision of the hardware and the presence of obstacles obstructing the exact line of sight to the satellite. For more rapid and precise position information, an IMU and appropriate software should be used.

Structured light sensors project a known pattern of light onto a surface or into an environment and record the changes with some form of optical sensor for the purpose of computing the difference between the observed pattern and the known structure to determine distance. A variety of commercially available approaches exist to project the light and record it for analysis, and can provide good depth resolution without an extremely large investment per

sensor. They are typically limited to indoor applications, however, due to their size, operating constraints, or being easily overridden by ambient outdoor light.

Perhaps the most popular construction of structured light mapping device is the 3D scanner. These often use a line of laser light and one or more orthogonally offset cameras to record the shape of the laser as projected on the object being scanned. This approach can be implemented by anyone from hobbyists scanning odds and ends [27] to companies like Shape Fidelity for use in scanning the outside and inside of all of the parts of a Saturn V F-1 rocket engine [28]. These scanners, however, require a known amount of movement to be able to assemble a 3D map of an object or environment from the two-dimensional data a given scan returns.

One of the most popular consumer-oriented implementations of a 3D structured light sensor is the Microsoft Kinect. Developed by PrimeSense, a Kinect broadcasts a unique "speckle" pattern in two dimensions in the IR spectrum and records it with a corresponding IR camera.

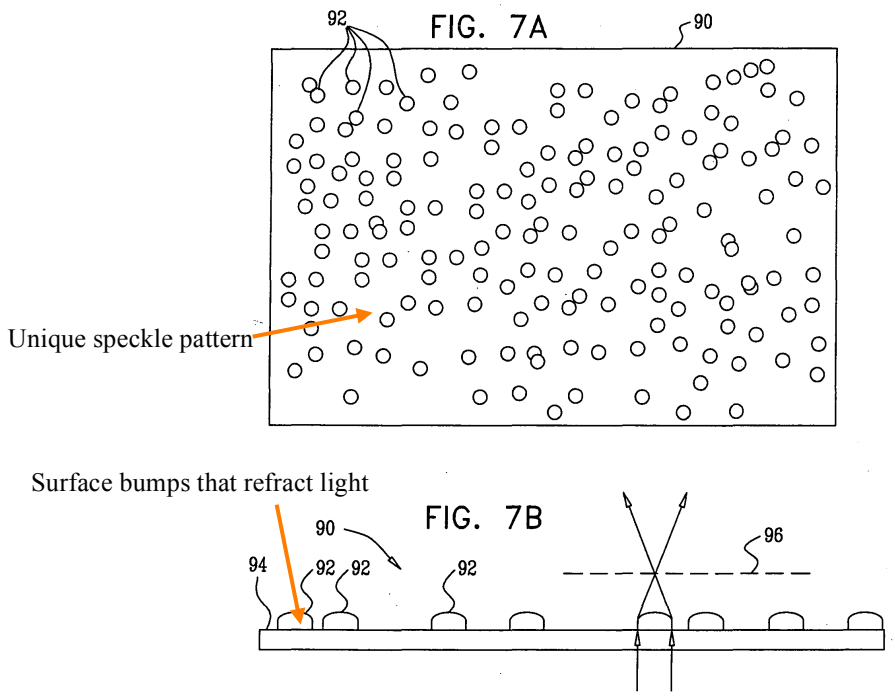


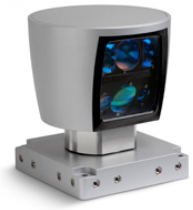
Figure 3.3: This speckle pattern is an example from one of PrimeSense's patent applications [29] for 3D mapping using structured light. This pattern would be repeated when broadcast on a surface. Annotation 92 denotes a light-refracting bump.

The changes in size and separation of these speckles and their relative intensity are compared to a calibrated baseline to determine the distance to a given point on an object.

Lidar is a depth mapping approach using the time-of-flight of an infrared laser. The laser is pulsed, and the time it takes the pulse to be received by the unit represents the distance to the object. The pulses are emitted at fixed angular increments throughout the sensor's field of view, and the flight times for those angles can be used to construct a multi-dimensional "scan" of the surrounding area. Most Lidar modules deliver a two-dimensional scan by sweeping through an angle, but several modules and arrangements are capable of providing 3D scans by performing multiple samplings at once (emulating several sensors with slightly offset scans) or changing the angle of a 2D sensor.

Lidar modules can have ranges up to 100 meters, with results accurate to centimeters or millimeters even in bright outdoor situations, given a high reflectivity of the surface of the object. These sensors are typically expensive due to the precise timing required, for instance accuracy to 1 cm resolution requires timing accuracy down to about 330 picoseconds.

Table 3.2: Comparison of different Lidar modules currently available.

	Hokuyo URG-04LX-UG01 [30]	**SICK LMS111-10100** [31]	**SICK LMS511-10100 PRO** [32]	**Velodyne HDL-64e S2** [33]
Application	Indoor	Outdoor (IP67-rated housing)	Outdoor (IP67-rated housing)	Outdoor (IP67-rated housing)
2D/3D	2D	2D	2D	3D
Points per scan	683	1080 or 590	1140, 760, 570, 380, 285, or 190	86,000
Scans per second	10	25 or 50	25, 35, 50, 75, or 100	5 to 15
Field of view	240°	270°	190°	Horizontal: 360° Vertical: +2°/-24.8°
Operating range	0.02 – 5.6 meters	0.5 – 20 meters	1 – 80 meters	120 meters
Angular resolution	0.352°	0.25° or 0.5°	0.167°, 0.25°, 0.333°, 0.5°, 0.667°, or 1°	Horizontal: 0.09° Vertical: 0.4°
Linear accuracy	<1m: ±30mm >1m: ±3%	±30 mm (typical)	<10 m: ±25mm 10-20m: ±35mm >20m: ±55mm	±10 mm (typical)
Data connection	USB 2.0 Full-Speed	CAN, Serial, 10/100 Mbit Ethernet	CAN, USB, Serial, 10/100 Mbit Ethernet	100Mbit Ethernet (UDP)
Price, USD	$1,100	$6,943	$10,206	$75,000
Product image	 [34]	 [31]	 [32]	 [35]

Commercially available products can vary widely in price and capability. Hokuyo makes a small, USB-driven model that could be considered suitable for the sophisticated hobbyist or small-scale product. They are capable of indoor and outdoor operation, but have no weatherproofing or fog compensation, and as of this writing still costs $1100 USD or more. SICK manufactures Lidar modules as well, and have been a common choice for past DARPA

Grand Challenge and Urban Challenge vehicles. By comparison, the Google driverless cars, Oshkosh TerraMax, and several DARPA Urban Challenge have used a Velodyne HDL-64E which uses 64 simultaneous lasers for a 28 degree vertical field of view, and rotates them around 360 degrees.

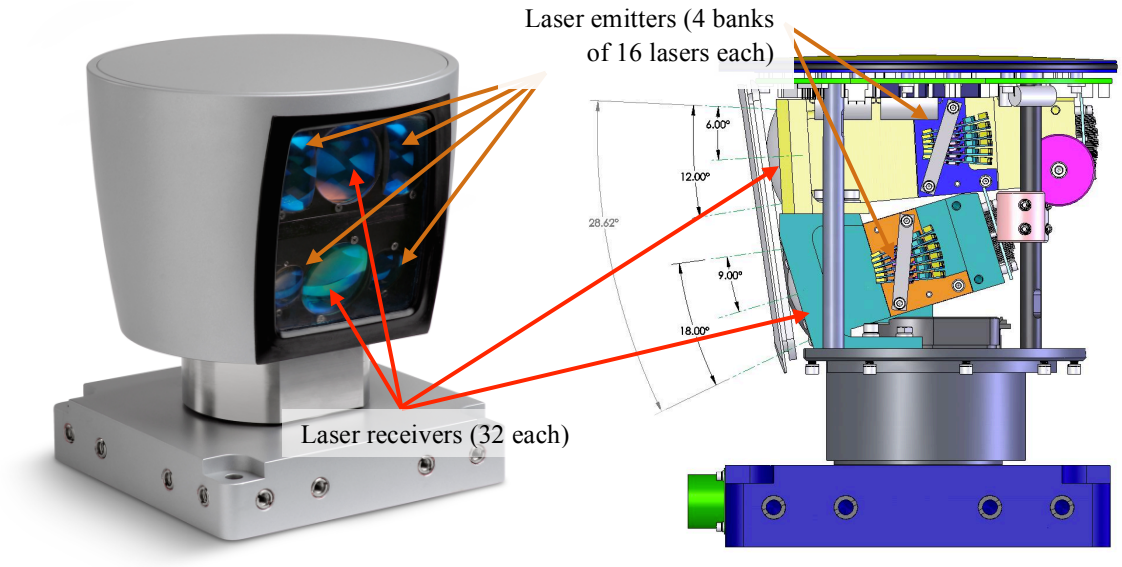


Figure 3.4: Velodyne HDL-64E Lidar module, with cover (left) [35] and without (right) [36]. The top cylinder includes the laser emitters and receivers, and rotates at up to 900 rpm to acquire fifteen full 360 degree, 86,000-point scans per second.

Radar works in a similar manner to Lidar in that it measures the time-of-flight of a signal, but radar uses short pulses of microwave radiation instead of IR light. The timing requirements for the detection hardware in a radar system are similar to those of a Lidar module since the microwave radiation also travels at the speed of light. Most radar modules only work well within a given range, as opposed to a Lidar's ability to work from very close ranges limited to the minimum measurable time of flight to the edges of the laser's range or the sensor's sensitivity.

Due to its longer wavelength it can be a more versatile sensor than just time-of-flight distance ranging a Lidar offers. By assembling an array of antennae that are independently controlled, the radar pulses can be not only steered due to constructive and destructive

interference, but the directionality of the antenna is increased to better block out ambient radiation. This implementation is referred to as a "phased array" due to the ability to change the phase of the signal at each antenna, and is typically used more in stationary radar locations.

Automotive radar has currently been on the rise in a number of uses, including adaptive cruise control, lane departure warnings, and automatic crash prevention measures. These are often distinct radar modules due to the limitations of the dynamic range of radar, adding to the cost of the vehicle, but allowing for better sensing across the different applications.

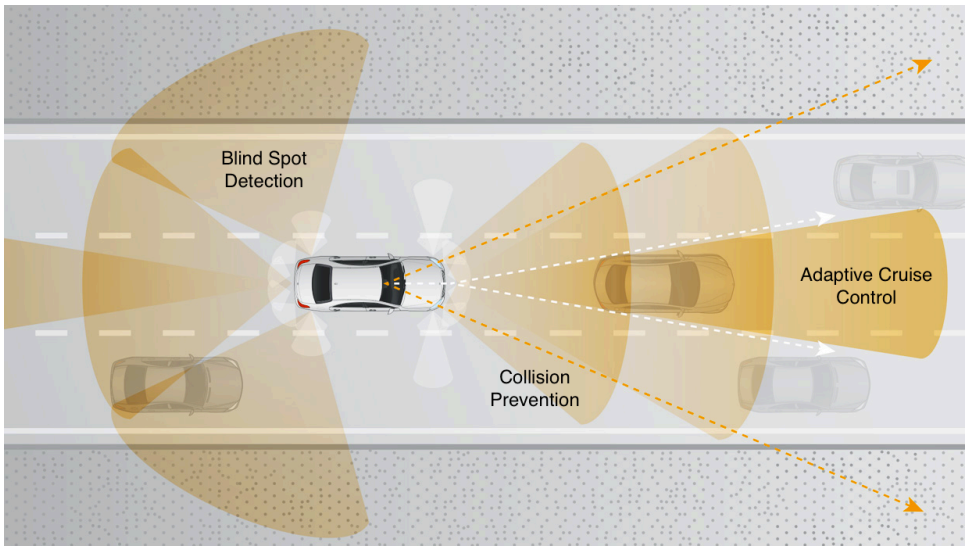


Figure 3.5: Diagram of the radar systems in place in the Mercedes-Benz Intelligent Drive platform [37]. The orange swaths in the image indicate the areas covered by different radar modules.

Figure 2.6 shows the areas covered by different radar modules in a Mercedes-Benz. The close-range and medium-range radars off the front of the vehicle are used for collision avoidance, including possible collisions with pedestrians. The long-range radar off the front of the vehicle detects cars up to 500 meters ahead and is used primarily for "adaptive" cruise control to maintain a set distance to the next vehicle in traffic and a safe following distance. The fields extending out from the rear quarter panels of the vehicle are used to prevent collisions with vehicles in blind spots when changing lanes.

Ultrasonic sensors are typically short-range time-of-flight distance sensors. They operate in the same manner as radar and Lidar, emitting a "chirp" and timing how long it takes for the chirp to return. Their short range, relatively low price, and ease of implementation makes these an appealing solution for detecting the presence of obstacles in a vehicle's immediate vicinity. They are typically susceptible to environmental factors, however, unless specialized environment-rated modules are purchased. They are also prone to interference from both other ultrasonic sound sources and other ultrasonic sensors, and thus a singular autonomous vehicle must coordinate its sensors to mitigate such interference.

3.2 Internal Sensing

Tracking the internal state of the system is also a critical aspect of autonomous systems, as the characteristics they observe may indicate whether or not a system is even capable of performing its designated tasks.

Switches exist in a variety of shapes and functions and are often used in the configuration or control of a system, but also have applications as sensors within a system. The "limit" switch is a type of snapping-action switch often used to detect when a moving part has reached the end of its range of travel. They can also be used with detents in a surface for simplistic detection of linear or angular displacement.

Potentiometers are variable resistors configured as voltage dividers and are applicable for sensing motion within a finite and typically small linear or angular range. As a sensor, these operate by moving a conductive wiper across the surface of a semi-conductive strip with a fixed resistance. As the wiper approaches one side or the other, the resistance between the wiper and the ends changes. This change in resistance can easily correspond to a change in voltage to be

interpreted by the controlling system. These are popular for measuring linear actuator displacement, as well as the angular displacement of joints in a system..

Encoders function by generating a "tick" for a given linear or angular displacement. These can function without limits to their range of motion, unlike potentiometers. These are popular sensors for measuring wheel speeds, and can function without inducing additional drag on the system. Depending on the application, the encoder can tick once per unit distance or revolution, or many times throughout the same range of motion. The higher the number of ticks the encoder can measure per given displacement, the more precise the sensor is. Given a constant, known number of ticks per unit, measuring the rate at which the ticks occur can indicate speed as well.

Encoders can create ticks in a number of different ways. The most common are the reflective and gated optical encoders. Reflective optical encoders bounce a light source off of a surface and measure the amount of light reflected to detect motion. Gated optical encoders place the emitter and receiver on opposite sides of a perforated strip or disk to sample the light/dark changes. These are typically used to sense rotation, as these can perform at many hundreds or thousands of revolutions per minute, and can be constructed to generate more than one tick per degree of movement.

Direction of rotation can be determined by generating two sets of ticks that are 90 degrees out of phase with one another, and sampling the state of one sensor on the changing (rising or falling) edge of the other. The high or low sampled state of the one tick is indicative of the direction.

Internal **temperature sensors** are often present in systems as safety or calibration instruments. Critical components, especially ones that are sealed or provided limited airflow, can

have temperature sensors mounted on or near them to ensure that they are not overheating, or operating within their allowed thermal limits. Preventative measures can also be taken if onboard hardware happens to fail, and sensors or subsystems shut down in order to avoid additional damage to the system. Temperature sensors are also used to ensure consistent data from other sensors across temperature gradients, as cold or warm temperatures can influence more sensitive sensor readings. High-end sensors can have temperature sensors built in and adjust themselves, or even heat the sensor to a known temperature to ensure consistent operation over all temperature conditions.

3.3 Data Fusion and Path Planning

Data fusion is the process of taking many sources of the same data into one piece of information that represents all of the sources. Kalman filters have the capability to combine data in the mentioned fashion by using mathematical models and knowledge about the system. One source of input for the filter will be the localized position from a SLAM (Simultaneous Localization and Mapping) algorithm which allows autonomous systems to navigate unknown environments. By leveraging this information, a path planning algorithm is able to plan a safe and accurate path through the mapped environment by using the estimated position from the Kalman filter.

3.3.1 Simultaneous Localization and Mapping

SLAM attempts to solve the problem of navigation without prior knowledge of the environment. The goal is to create a map of the environment by collecting data with sensors about the outside world as well as knowing the robot's location in this environment. SLAM is important to have on autonomous robots that have any intention of exploring unknown environments because it allows robots to function with a great deal of autonomy that otherwise cannot be attained [38]. SLAM is what robots use to know where they are in an environment,

and as the name implies SLAM is made up of two parts: localization and mapping. If an agent can obtain a very accurate initial position, it can localize via accumulation of motion. This can be accomplished through wheel odometry or an Inertial Measurement Unit (IMU). However, over time errors accumulate and eventually the position solution will diverge from ground truth unless corrected. A system will often take into account the distance moved along with how the environment has changed around it. They system might get this data from Lidar, cameras, or other sensors. This current perception can be compared to a global map in order for the system gain knowledge of where it is in the map [38].

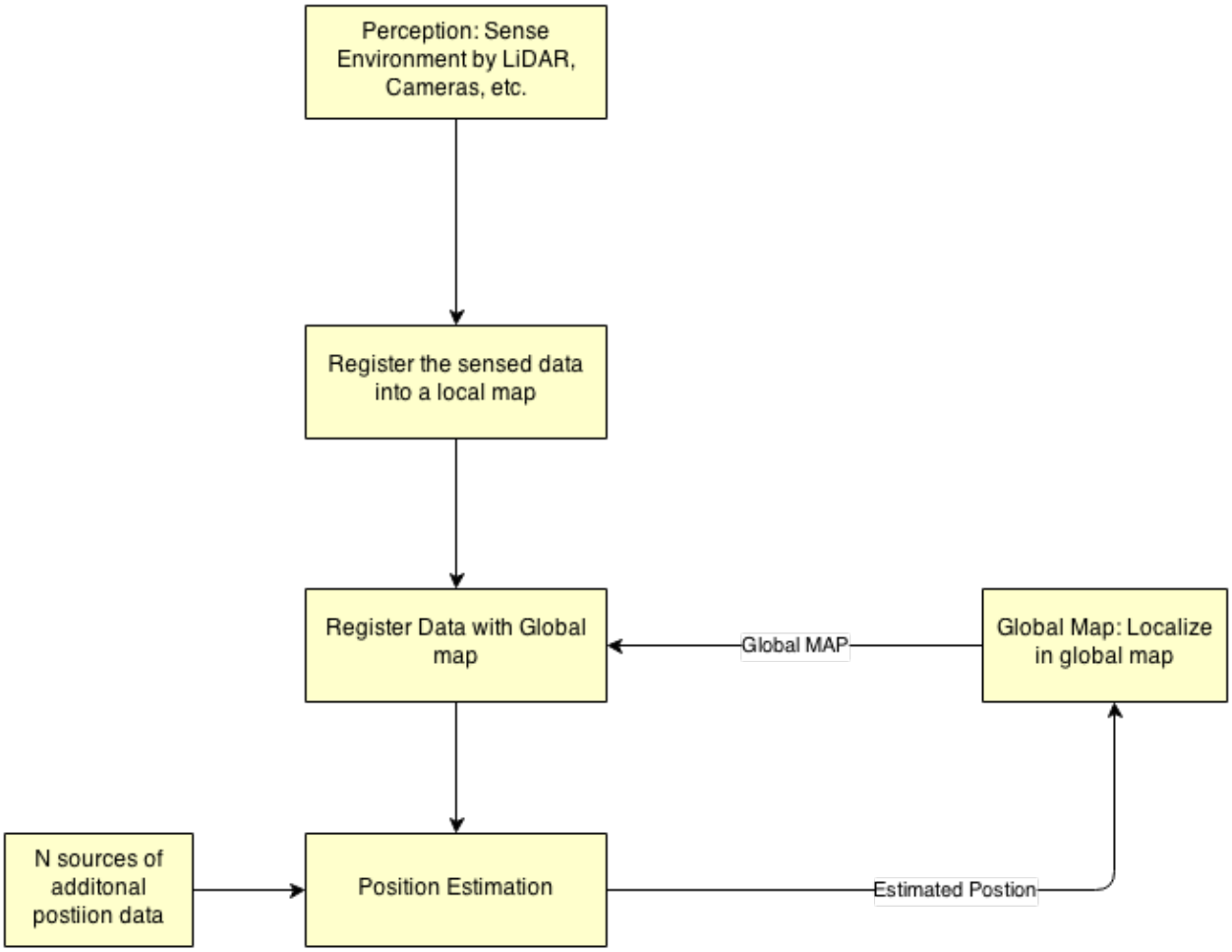


Figure 3.6: Flow chart of how a SLAM algorithm can work.

Figure 3.6 gives an example of how SLAM predicts position. Perception is the act of the agent sensing the environment, which could be obtained through Lidar, stereo cameras, or other sensors. The data collected is processed and put into a local map which is the representation of what the robot perceives as the moment [38]. Using the global map, the robot tries to match what it sees with what is in the global map. This is when the agent predicts where it is in the frame of the global map. The agent may correct its position by methods such as Kalman filters. Map fusing is performed by determining what parts of the local map should be registered in the global map. Once the agent has the new global map it selects where to look next and executes its next motion [38]. SLAM is an important tool used in autonomous systems, but Kalman filters are often used to smooth and fuse noisy sensor data that is used in SLAM.

3.3.2 Kalman Filters

The Kalman Filter was first conceived by Rudolf E. Kalman over fifty years ago [39]. It is a data fusion algorithm that generates estimates from noisy data for a linear system.

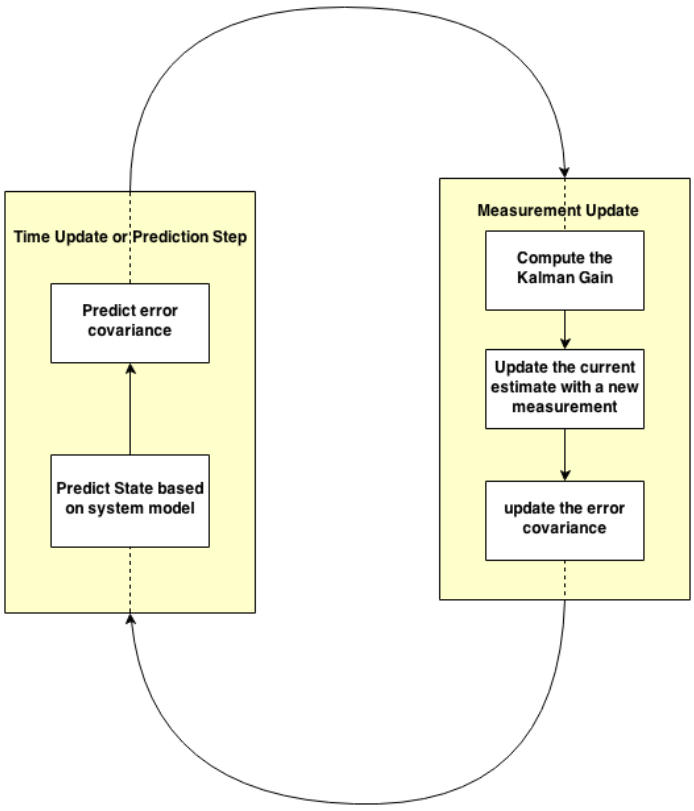


Figure 3.7: The Kalman Filter consists of a time update based on mathematical models and a measurement update that is the noisy sensor data

Applications for Kalman filters include GPS receivers, smoothing laptop trackpad output, and navigation. One of the most visible early implementations of the Kalman filter was in the navigation computer in Apollo 11 [39].

There are two main parts to a Kalman filter: a time update and a measurement update. Figure 3.7 shows the basic cycle that a Kalman filter goes through. In the time update, the state transition matrix, the mathematical system model, is propagated for one time step. The time update makes a prediction about the model and updates the covariance. After the time update, the Kalman filter performs the measurement update. It is as this point the Kalman filter fuses the measurement inputs and makes corrections to the states estimated in the prediction step.

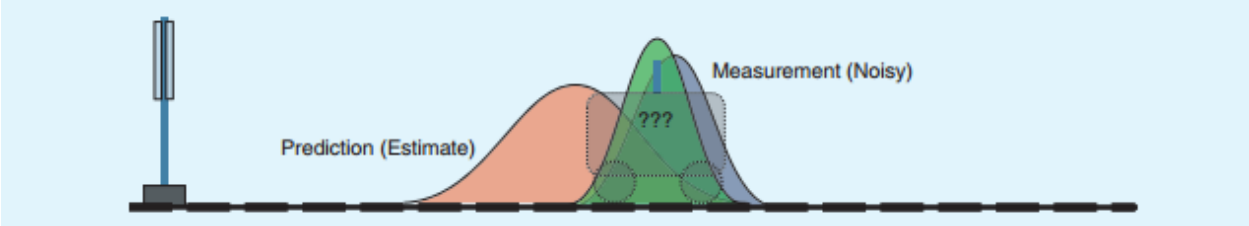


Figure 3.8: Shows the new covariance (in green) after a measurement update. Illustrates that the confidence in the new estimate is greater than that of the prediction or the noisy measurement [39].

In the case of this project, the Kalman filter is fusing position data from SLAM, the IMU and GPS. In the first step of the measurement update, the Kalman filter computes the Kalman gain using the error covariance and the measurement covariance. The figure above is a visual representation of the prediction, the noisy measurement, and the filtered output. They are modeled as probability density functions. The figure shows that after the measurement step the filtered output can be more trusted than either the prediction or the measurement alone. The value of the Kalman gain can show if the prediction or the measurement is more "trusted" [40]. The prediction is updated based on the Kalman gain, and then the covariance is once again updated. A unique feature of the Kalman filter is that after every measurement update the covariance will decrease. This means that as long as the filter receives sensor data every time

step and the state remains unchanged, the filter will be more confident in the estimation than the previous estimation. On the other hand, the covariance increases after every time update. If something were to happen and the robot no longer receives sensor data it would eventually have no idea where it is [39]. There are other forms of Kalman filters such as the Extended Kalman filter (EKF). This variant is made to operate on nonlinear systems, which is why this is the filter that will be used on this project for data fusion [40].

3.3.3 Path Planning

Path planning uses a map of its environment to navigate the environment. A good path planner must be able handle a dynamic world in order to prevent collisions between the robot and objects in the world. The path planner creates a path in configuration space (C-space) which is how the system represents the world [41]. In path planning, there is often a reference to cost, which is usually defined by the application. Cost may be defined as distance, or another variable that needs optimization. A path planning algorithm should meet the following requirements. One, the path computed should have the least cost. Two, the time to process a path should not create significantly slow down the rest of the system. The algorithm should be independent of the map that is used. One such generic path planner is the A* algorithm, which is often used to find the shortest path. It combines a heuristic and best first search [41]. A heuristic is a function that estimates cost, and in the case of finding the shortest path the heuristic is the distance formula shown in Equation 3.2:

Equation 3.2

$$h(n) = \sqrt{((x_2 - x_1)^2 + (y_2 - y_1)^2)}$$

A* uses this heuristic with a fitness function below in order decide what possible paths to search, where g(n) represents the total cost of the path up to the current point. The fitness function is represented in Equation 3.3:

Equation 3.3

$$f(n) = g(n) + h(n)\ [41]$$

A* maintains a list of possible nodes or points to travel to, and it searches the node that has the smallest fitness function value. This algorithm is guaranteed to find the optimal path as long as the heuristic underestimates the cost to the end goal [41]. However, underestimating by too much can cause the algorithm to search more paths than is needed, and over estimating may result in a suboptimal path.

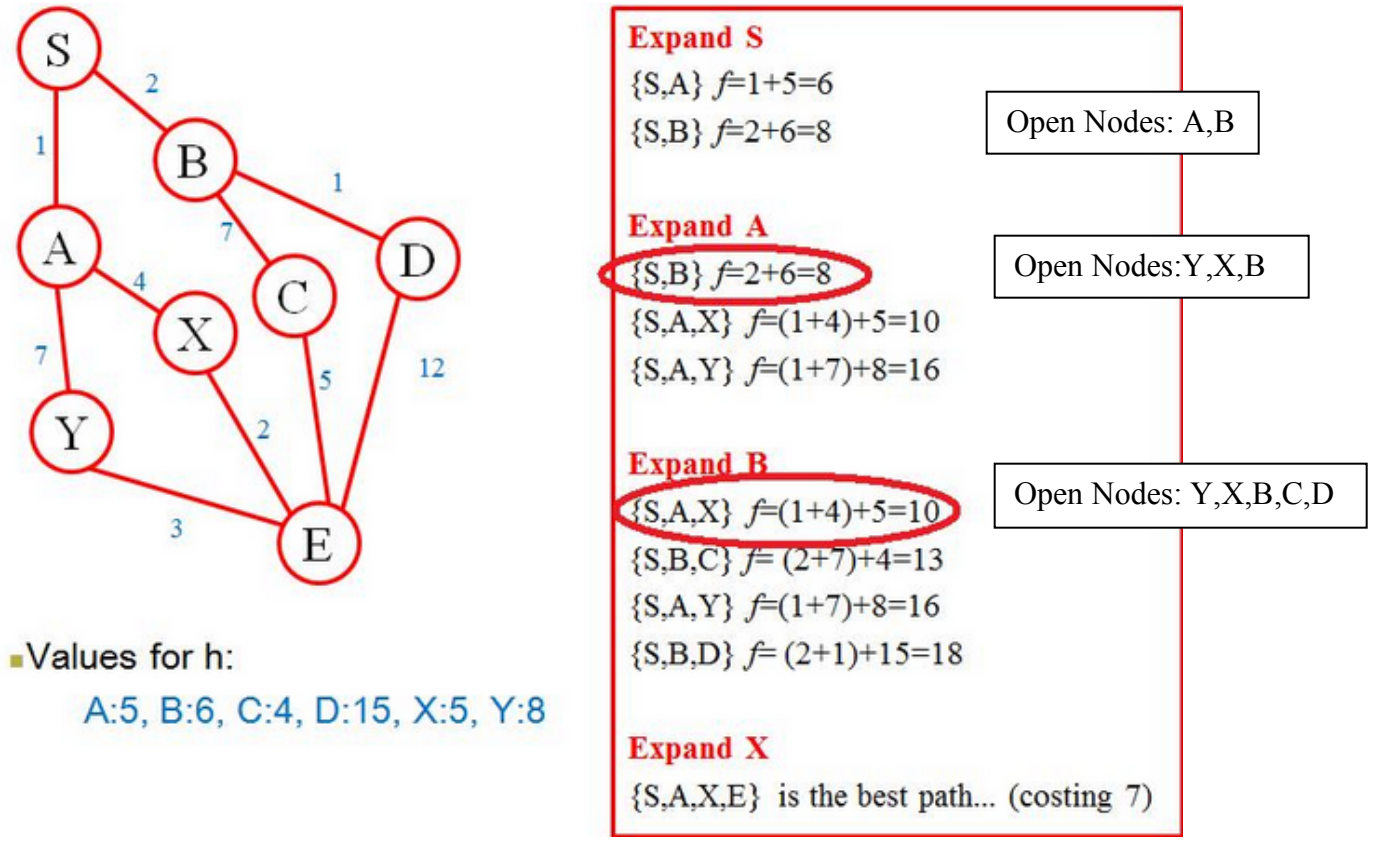


Figure 3.9: A simple example of how A* works. This shows that A* can be get complicated when the space to search becomes very large [42].

A* works by searching a list of open nodes, and expanding the node with the smallest fitness function. In the example in Figure 3.9, S expands to include nodes A and B. A is expanded first to include Y and X in the search, but because B now has the lowest fitness value, it is expanded next to include C and D in the list of open nodes. X is expanded last because it finally has the lowest fitness value, and the final path is S,A,X,E. While A* will find the path with the least cost, the downfall is that computing a path with a large number of nodes may take some time to compute. There are other path planning algorithms available that are more efficient, but they are more complex than A* [41].

3.4 Ground Vehicle Mechanics

Converting control signals into physical actions takes place in the electro-mechanical system of the vehicle. These systems control motors, brakes, and direction that the vehicle moves in.

3.4.1 Steering

There are several different types of ground vehicles and methods of locomotion of these vehicles. These different methods allow the ground vehicle to achieve different types of motion. There are six standard directions of motion: translation along the x, y, and z axis and rotation around each of those axes. Different drive trains have differing constraints on motion in each of these six directions. These are usually grouped into holonomic and nonholonomic vehicles [43]. With reference to the robot frame, a holonomic vehicle can move from rest in any direction on ground. This means from rest the vehicle can translate along the x and y-axis and rotate about the z-axis. In other words, the robot can drive forward, sideways, or spin in place. A nonholonomic vehicle does not have all of these abilities.

A standard car with Ackerman steering is classified as a nonholonomic vehicle as the drive train allows for only one degree of freedom: translation along the x axis. However due to the ability to steer, these vehicles have two degrees of mobility. This means a vehicle with Ackerman steering can move in the x and y direction with respect to the robot frame. However, in order to achieve movement in the y direction, the vehicle must turn about an instant center of rotation. This ICR is determined by extending lines perpendicular to all wheels of the vehicle and determining the point at which they intersect. This intersection is the ICR. Figure 3.6 displays the method of determining the ICR for a vehicle with Ackerman steering.

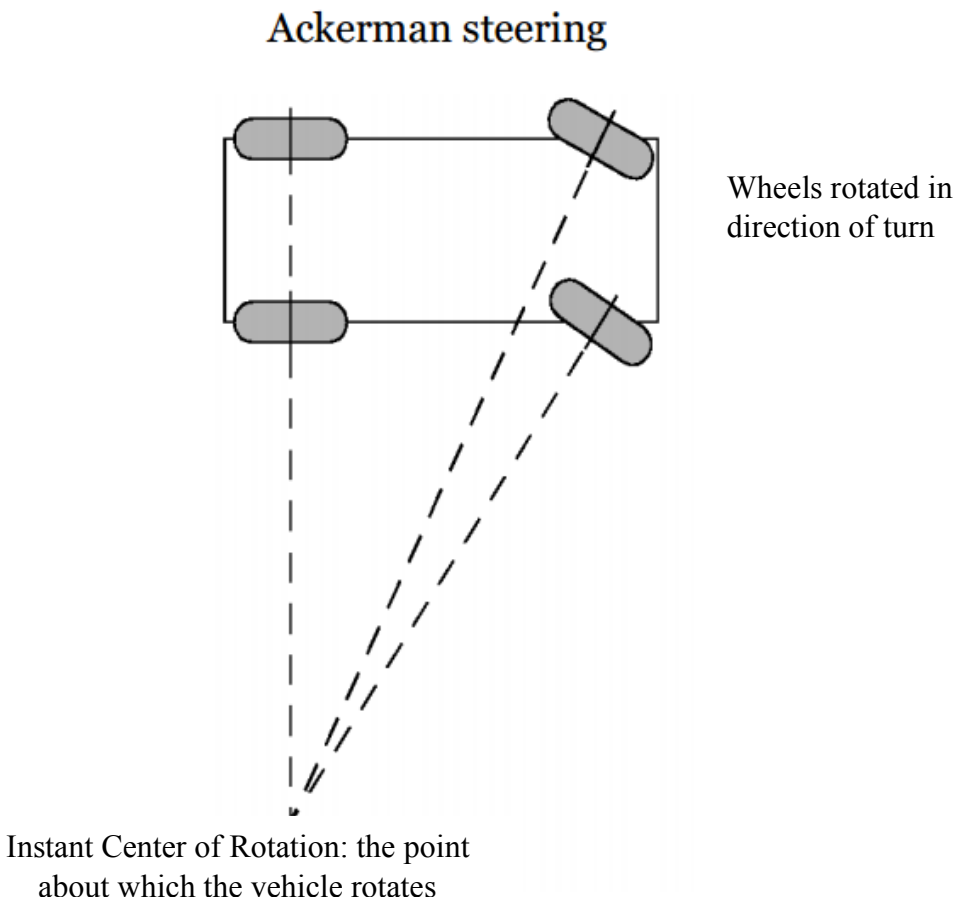


Figure 3.10: Ackerman Steering Instant Center of Rotation

The Ackermann steering method is performed using a collection of linkages so that the wheels turn together, and at the appropriate angles. These linkages then need to be actuated. In

the scope of this project, the Ackermann steering on the ground vehicle is actuated using a manual rack and pinion system. This system is described in full detail in Chapter 5.

From rest, some nonholonomic robots can rotate about the z-axis of the robot frame. As displayed in Figure 3.7, a robot with parallel treads or wheels have only one line between the wheels. This means an ICR can be located at any point along the line and that the robot can perform any degree of turn including zero thus allowing rotation about the z axis of the robot. These parallel-wheeled robots also have two degrees of mobility.

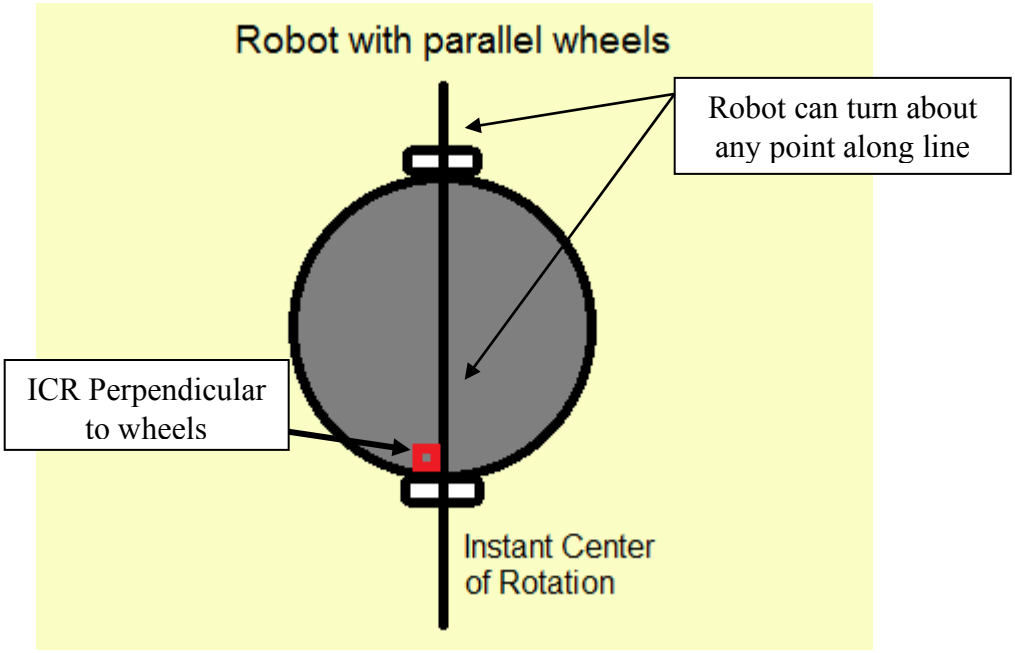


Figure 3.11: Robot with parallel wheels and infinite Instant Centers of Rotation

These motion constraints must be considered when planning the motions of an autonomous robot. Navigation must be customized per vehicle and the vehicle maneuverability to ensure the safest yet most efficient movement of the vehicle through the environment it is navigating. Paths planned for robots to avoid obstacles must include turns of a degree which can be executed by the drive train of the robot. If a path is planned around an obstacle at too tight of a turn, the robot will not be able to accomplish this movement and will likely collide with the

obstacle it was told to avoid. Programmers must account for the mobility of the robots when creating the path-planning algorithm for the autonomous navigation of the robot so that only paths that the robot is physically capable of following are generated.

3.4.2 Braking

The purpose of the braking system on a vehicle is to be able to reduce the speed, and eventually bring the vehicle to a stop. The two main methods used to do this on freewheeled ground vehicles; drum brakes, and disk brakes. Drum brakes (shown in Figure 3.12) use two brake shoes to apply pressure to the inside of a cylinder, causing friction and the wheels to slow. This project will be working with drum brakes, since they were already installed on the golf cart.

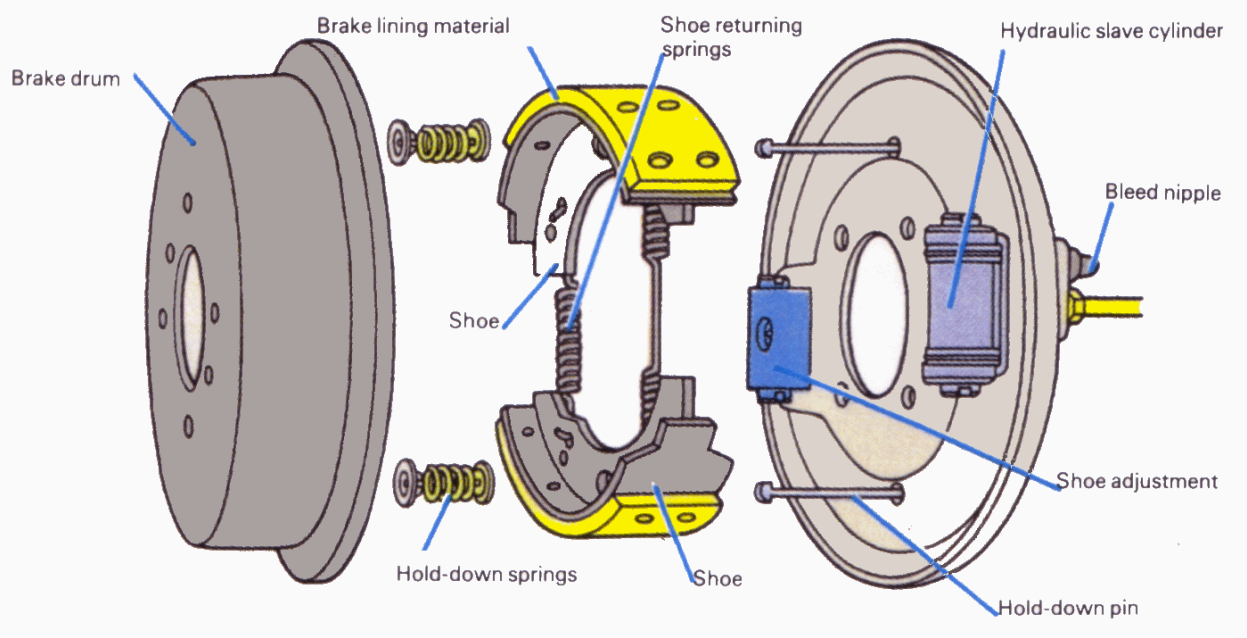


Figure 3.12: Schematic of drum braking systems. The brake show is pushed outward, applying pressure to the interior wall of the brake drum, slowing the spinning of the drum [44].

The second popular option for braking is disk brakes. Disk brakes work by using brake pads to squeeze onto a rotating disk, applying friction, and slowing the wheels to a stop.

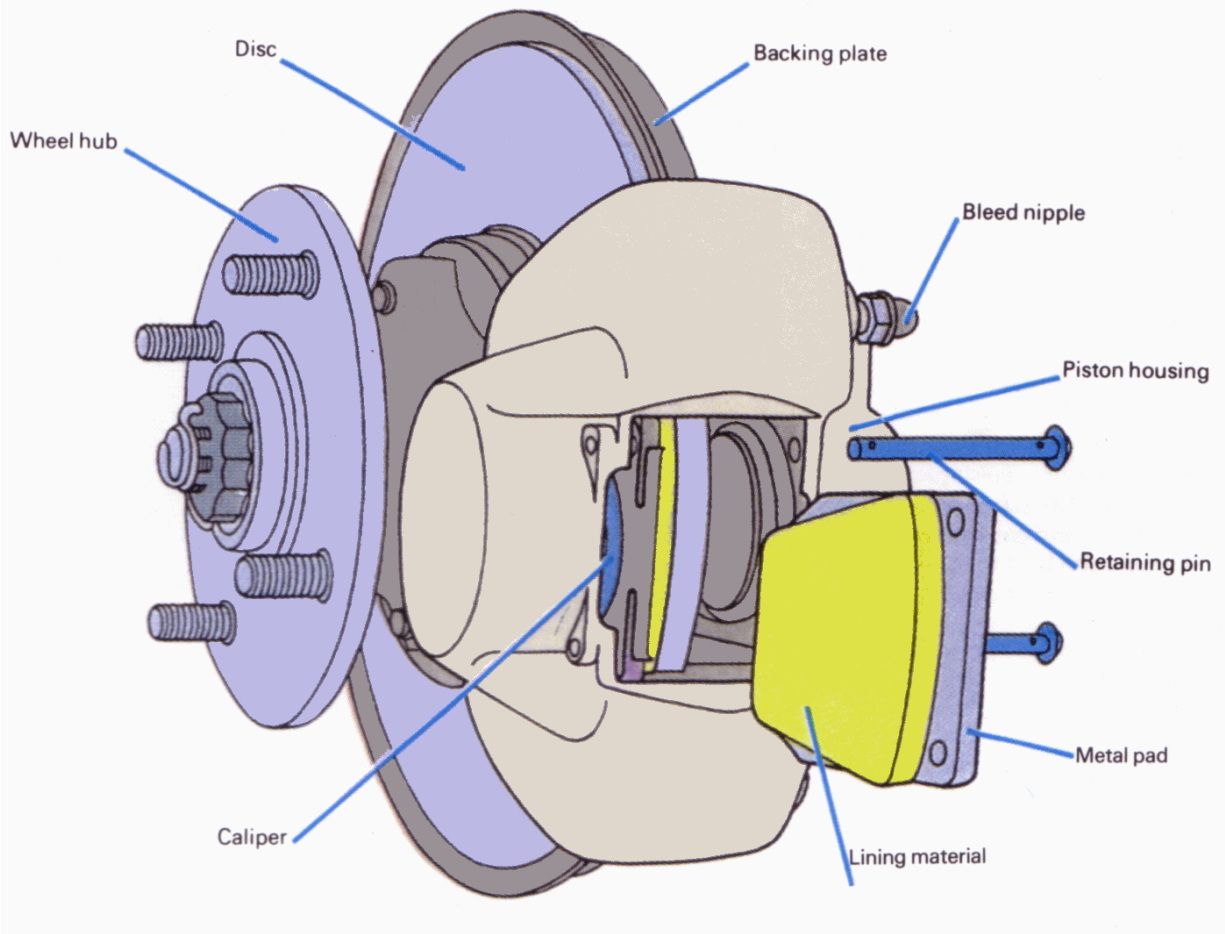


Figure 3.13: For disk brake systems, the calipers squeeze the brake pads onto the revolving disk, slowing the disk to a stop [44].

3.4.3 Actuation

An actuator is a mechanism that converts energy into motion, whether it be from an electric current, hydraulic fluid pressure, pneumatic pressure [45]. Mechanical actuators convert one form of directional motion into another [45]. Hydraulic actuators are limited in speed, but can exert a great deal of force due to the characteristics of fluid [45]. These move by applying fluid pressure to one, or two sides of a piston within a cylindrical tube [46]. Similar to hydraulic actuators are pneumatic actuators, which use compressed gas in place of fluid [45]. They can start and stop quickly, but are limited in strength [46].

Actuators that create motion in a straight line are known as linear actuators [45]. The original steering on the electric golf cart utilizes a mechanical linear actuator known as a rack and pinion. As seen in Figure 3.14, this actuator converts the rotational motion applied to the steering wheel, into linear motion of the steering rack [47]. As a result, the wheels turn to direct the car either left or right.

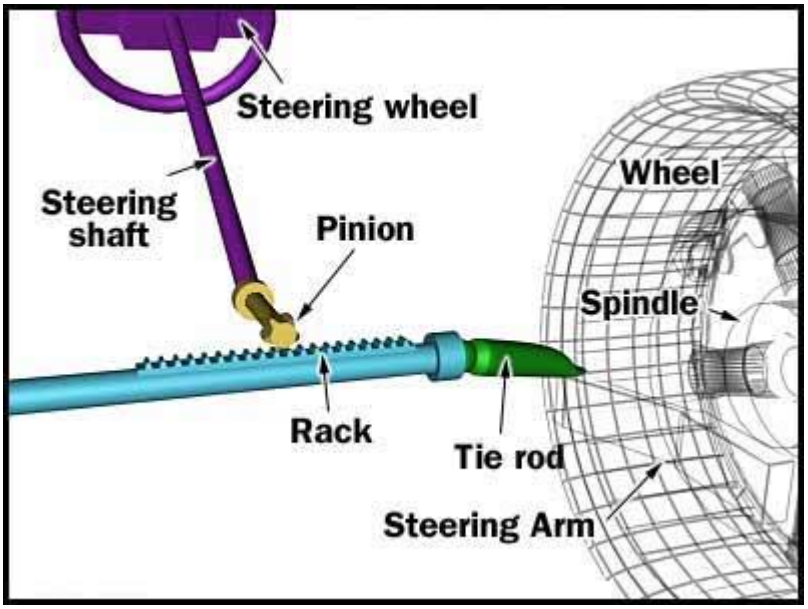


Figure 3.14: Rack and pinion mechanical linear actuator. This actuator coverts the rotational motion enacted on the steering wheel into linear motion along the steering rack [47].

Electromechanical linear actuators utilize electric motors in order to create rotational motion and convert it to linear motion [48]. The rotational motion of the motor turns a gear, which in turn rotates a screw [48]. The screw then forces itself through a nut, pushing the screw forward and extending the actuator arm [48]. Figure 3.15 depicts an electromechanical linear actuator, like the ones discussed in later chapters. Some of these actuators also come with a potentiometer in order to enable the system to know how much the actuator arm is extended, also known as feedback positioning [49].

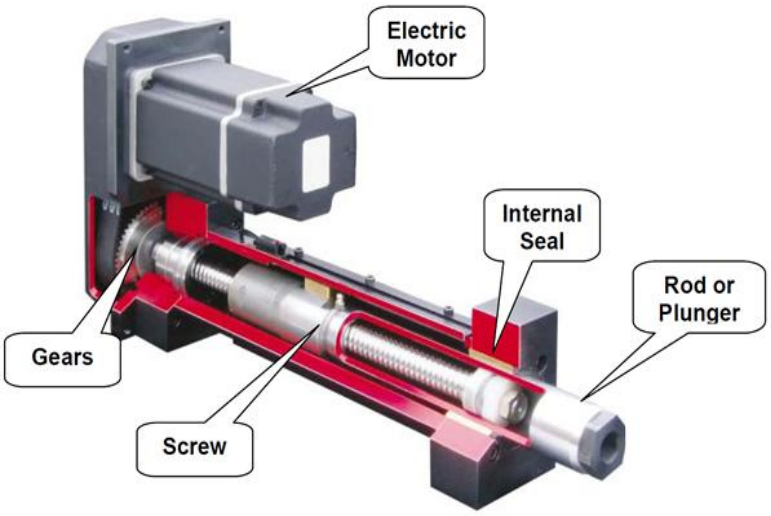


Figure 3.15: The internal componenets of an electromechanical linear actuator. The rotational motion of the motor turns a gear, which rotates a screw. The action of the screw forcing itself through a fixed nut results in the linear motion of the actuator arm [48].

3.5 Aerial Platforms

Quadcopters, like the AR Parrot Drone, the Asctec Hummingbird, and the ArduCopter are options for research for forward deployed reconnaissance. The WPI Robotic Department currently already owns the Asctec Hummingbird: an approximately $7000 quadcopter which is programmable using the Arduino environment, and thus with MATLAB through the implementation of the Simulink Arduino Interface.

Figure 3.16: The Asctec Hummingbird quadcopter

The project team decided not to use the Asctec Hummingbird, seen in Figure 3.16, due to a lack of onboard sensors, the lack of a camera, the lack of support and informative materials

about the specifications of the quadcopter, and the potential cost of repairs in the event of a crash and damage. Additionally, the Asctec Hummingbird does not fly-out-of-box.

At the time of the team research and decision making process, the Arducopter was a do-it-yourself quadcopter kit which included the control hardware and electronics. A quadcopter made by a hobbyist using the Arducopter components can be seen in Figure 3.17. The kit cost around $300 and contained an ArduPilot, a gyroscope, a barometer, a magnetometer, an accelerometer, and GPS. However, the kit did not contain any body elements of the quadcopter.

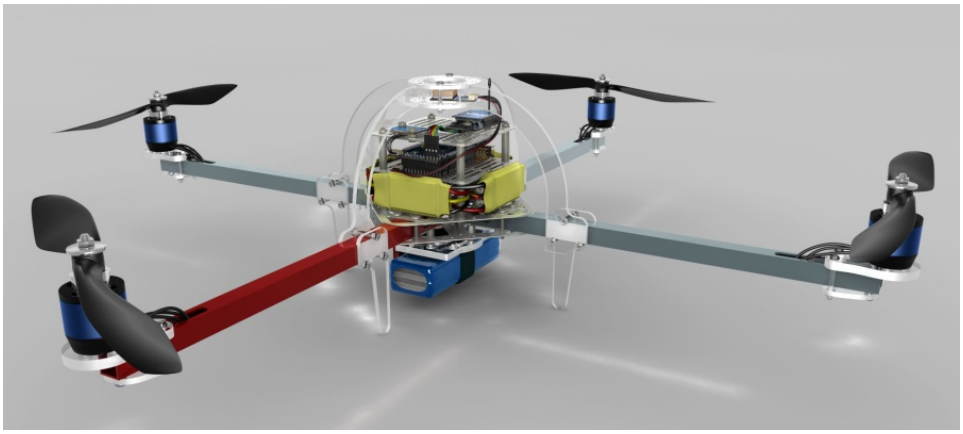


Figure 3.17: The Arducopter as built by a hobbyist . The body of this quadcopter was sold separately at the time of quadcopter selection.

The ArduPilot can be programmed using the Arduino environment. Because of the lack of ability to fly out of box and the limited time of the project, the team decided not to use the Arducopter.

The AR Parrot Drone 2.0 is a quadcopter designed to take HD video and to perform aerial stunts controlled by an app on an Android or IOS mobile device and is displayed in Figure 3.18. The AR Parrot Drone 2.0 costs $400 and includes a forward facing high definition camera, a downward facing camera used for optical flow to calculate ground speed, an IMU, an ultrasonic sensor, a magnetometer, an altimeter, and two shells for indoor and outdoor operation.

Figure 3.18: The AR Parrot Drone 2.0 with the indoor shell equipped.

The AR Drone also has a SDK and an API available online for consumer use. The team additionally found ROS packages which had been developed by other ROS users to communicate with and control the drone.

3.5.1 Aerial Vehicle Motion

There are different types of aerial vehicles with different dynamical features. When working with aerial vehicles, there are six different types of movement which can be achieved. These are labeled as x, y, z, roll, pitch, and yaw which are shown in Figure 3.19. It can be seen that x, y, and z are translational movements where roll, pitch, and yaw are rotational movements. Roll is the rotation about the x-axis. Pitch is the rotation about the y axis and yaw is the rotation about the z axis.

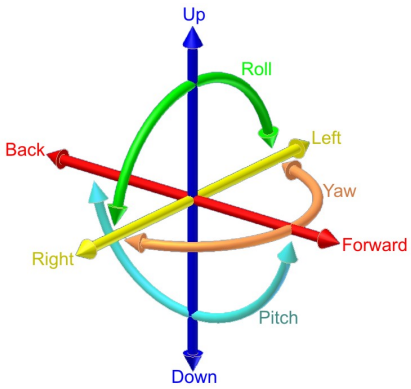


Figure 3.19: Six degrees of freedom in relation to aerial vehicles. This figure represents x as forward and back, y as right and left, and z as up and down.

An ability to move in the directions of these movements independently from the others is defined as a degree of freedom. Different types of aerial vehicles have different constraints on these degrees of freedom. Traditional planes, helicopters, and quadcopters have four degrees of freedom: x, y, z, and yaw. These aerial vehicles can in fact move in all six directions however, they do not have all six DOF because roll results in a translation in the y direction and pitch results in a translation in the z direction.

A quadrotor helicopter combines the torques generated by the four rotors to create forces in the desired directions. The generation of these forces can be viewed in Figure 3.20. It can be seen in this figure that a larger force in a pair of diagonal rotors results in a yaw motion. A larger force generated by a pair of contiguous rotors, two and three for example, results in a translation in the direction opposite the pair. For example, a larger force in rotors two and three would cause the quadrotor to fly forward.

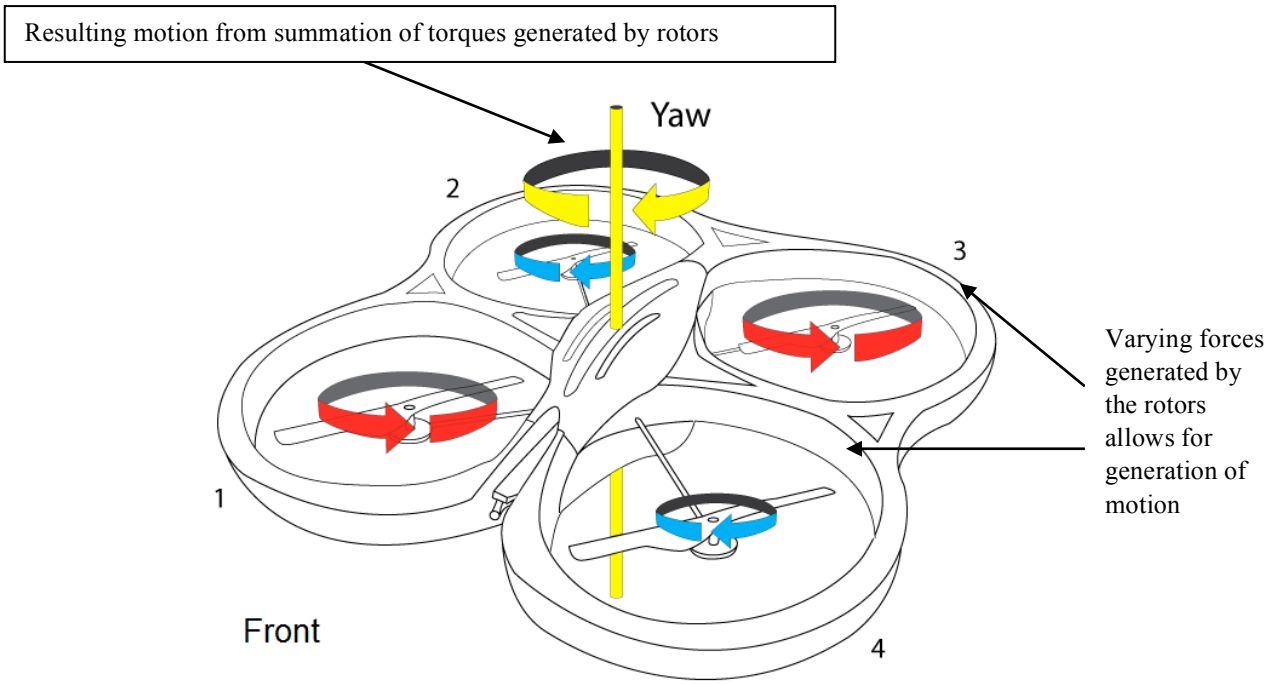


Figure 3.20: Quadrotor torque generation and resulting motion.

These movement capabilities of the quadrotor allow for strafing. This means the quadrotor can fly on a diagonal to translate along the x and y axes simultaneously. When faced with navigation, a quadrotor presents many options for methods of maneuvering to a goal. The quadrotor may strafe to move to a point some distance along the x and y axes or the quadrotor may rotate about the z axis until the point is directly ahead then translate along the x axis. A third option of navigation for the quadrotor is to translate the required distance along the x axis and then along the y axis or vice versa.

3.6 Chapter Summary

This chapter gave an overview of the technologies and background knowledge needed to design an autonomous system. Each portion of the design plays in integral role in the functionality of the system: sensors for realizing the environment, electro-mechanical systems for controlling the vehicle hardware, and control software for the decision-making. Each of these portions must be designed for robust operation in the desired operating environment.

Aerial vehicles have been traditionally used for forward deployed intelligence collection. Since the Cold War, Unmanned Aerial Vehicles have been the choice vehicle for this reconnaissance task. These UAVs are traditionally large and in the style of a traditional airplane. Quadcopter helicopters are on the cusp of entering the forward deployed information reconnaissance scene. Currently, quadcopter helicopters are more popular among the hobbyist crowd.

Quadcopters have unique movement capabilities. The combination of the thrust forces generated by the four rotors results in rotational and translational movements. This unique method of generating motion also allows for the quadrotor to travel in two directions at the same time. When these two directions are along the x and y axes, the motion is called strafing. Strafing

allows for the drone to remain at the same yaw angle while changing position which can be useful when cameras are being used.

4 Proposed Implementation

The system designed and implemented for this Major Qualifying Project was created as an experimentation of a multi-robot system, which uses a shared pool of data to make navigational decisions. Each robot in the system contributed to this collection of data from their various sensors.

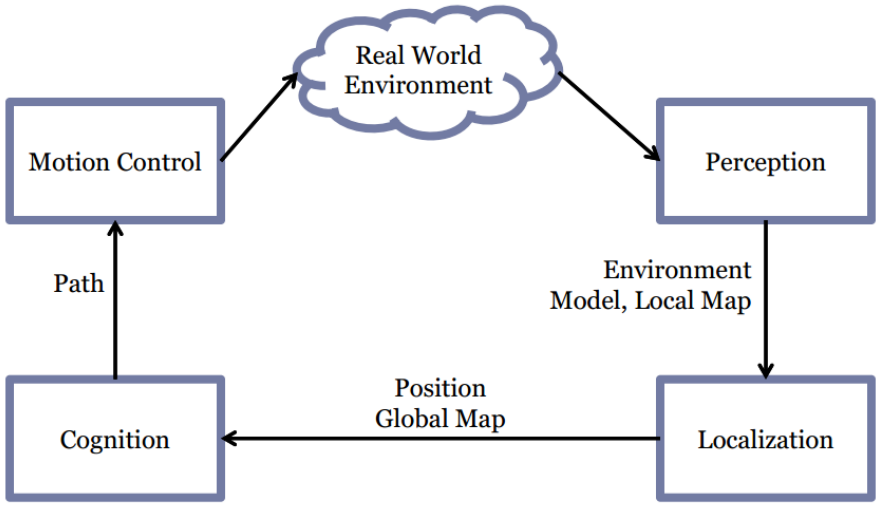


Figure 4.1: The process a robot follows when navigating in an unknown environment.

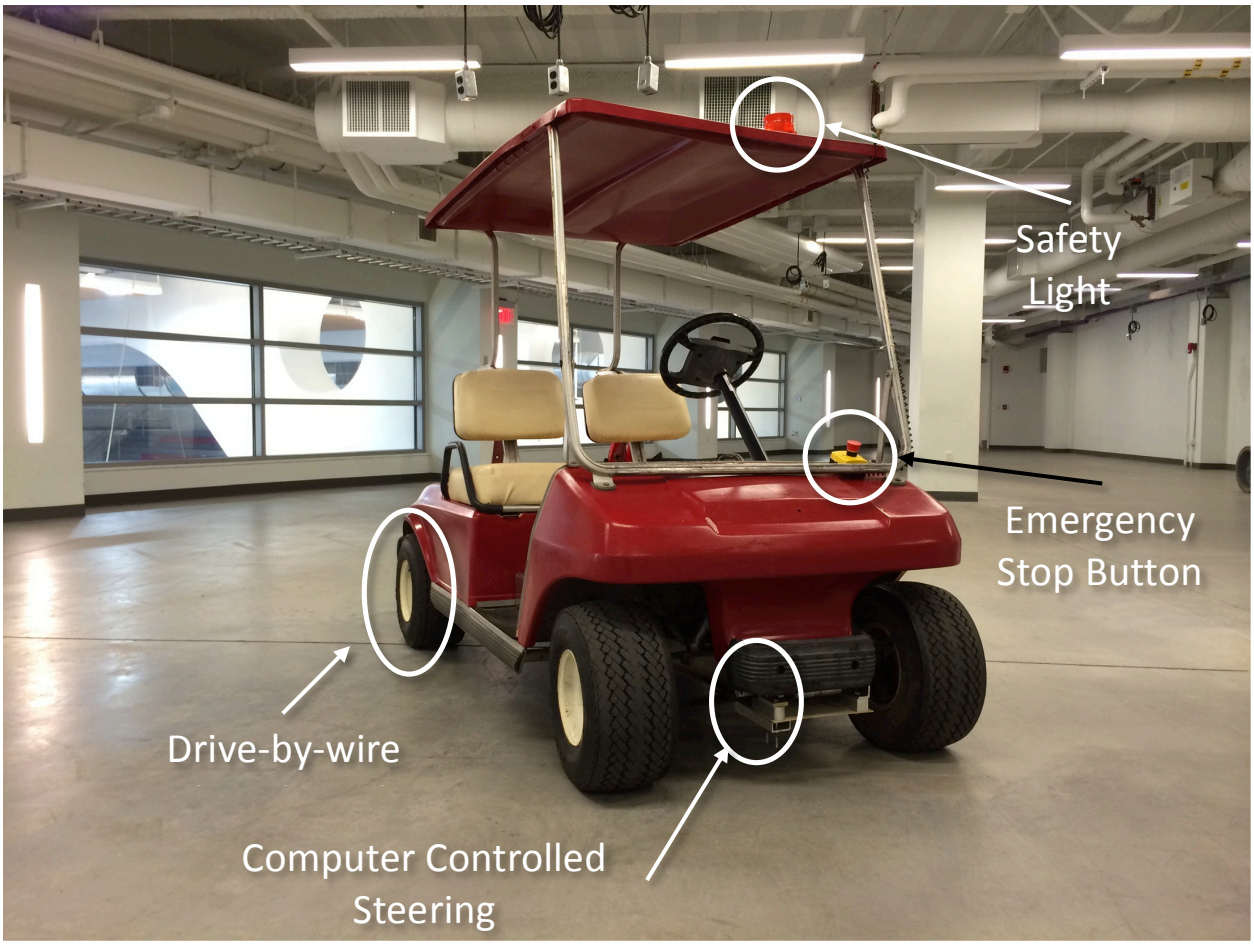


Figure 4.2: Final cart, with electro-mechanical and safety systems installed. The drive-by-wire and computer controlled steering systems are used for motion control, and the cameras and sensors get mounted on the front to realize the real world environment.

This combined data was then used to enable all robots to safely and effectively navigate through any environment which met the predefined project constraints. Figure 4.1 displays the process each robot executes while performing the task of navigation. The project team examined and observed the benefits and challenges associated with a system of collaboratively navigating autonomous robots. Assumptions were made for the system and testing environment to simplify the complexity of the tasks to allow for the system to be successful under the constraints the project team faced.

The system developed to meet the requirements and constraints of this project consisted of one autonomous ground vehicle, a golf cart, and four quadrotor helicopters, AR Parrot 2.0

drones. The golf cart performed SLAM, planned paths to user input destinations, and avoided obstacles as it navigated. The quadrotor helicopters individually flew to waypoints along the planned path of the vehicle and identified and located orange traffic cones as hazardous obstacles for the golf cart to avoid. All data collected by the sensors of the cart and the drones was communicated to the central computer. This computer, using Robot Operating System (ROS) and MATLAB, interpreted the data and used it to develop a global map of the environment. Tasks were performed based on the information displayed in this map.

The system was developed under various constraints. These constraints and the resulting system design are described in below in the Design Constraints section and the Proposed Design section. Also in the Proposed Design section, tasks defined for the individual robots and assumptions about the robots and the environment made by the team are reviewed below.

4.1 Design Specifications

In order to be able to accurately define the needs in the design of the project deliverable, the team had to consider numerous constraints. These constraints can be grouped into seven distinct categories: financial constraints, time limitations, available workspace, safety concerns, operational environment, wireless communication protocols, and power distribution. Each of these categories had an effect on how every aspect of the project was approached, and overall determined the outcome of the project deliverable. This section will explore each of these categories in depth, and discusses the impact on the design choices made in the development of the collaboratively navigating autonomous system.

4.1.1 Experiment Specifications

In order to simplify the requirements of the vehicle for the task of autonomous navigation, the project team had to simplify the environment in which the vehicle would be operating. The team made decisions on which environmental obstacles would be accounted for in

the design. The team also made assumptions as to which obstacles would never be encountered and thus would not be considered when making decisions about the vehicle design. One simplification made was the decision to not be concerned with abiding by the rules of the road. Another assumption the team decided to make about the operating environment is that the vehicle will only traverse flat terrain. Additionally, it has been assumed that the vehicle will never be required to make turns sharper than that of its turning radius and will never be required to reverse.

The vision system must be able to operate in an outdoor environment, which means a vision system cannot be used that utilizes infrared sensors, such as the Microsoft Kinect because the sun produces light in the infrared spectrum. The Kinect is a camera that uses an infrared sensor to gather depth data about the surrounding environment. The noise induced by the Sun's infrared light will cause the depth data from a sensor like the Kinect useless. This was a driving factor to use RGB cameras. The team did not restrict the environment to indoors or outdoors, and thus the sensors used must be able to function properly in both types of lighting and must be somewhat weather resistant. This assumption meant we could not use any sensors that depend on infrared light due to the interference from sunlight.

4.1.2 Information and Sensor Connectivity

To successfully communicate with the UAVs engaged in the project without hindering their movements or range, they were operated via a wireless connection to the ground vehicle. Ultimately, the project is intended to function independent of the type of connection, and only needs a respectable range and a minimum speed. This is achieved by allowing the drones function as separate, self-contained systems that take positional commands from the ground vehicle and relay back information about the observed environment.

The AR Parrot drones acquired for this project already used Wi-Fi to communicate with a phone or computer, but also contained several other wired hardware interfaces onboard. These were pursued in the hope that they may be used to control the drone (and thus allow the team to attach a controller and use an independent radio), but were dismissed due to the short time and lack of documentation. An alternative proposal was to attach a small Linux-based computer to the drone (ie a Raspberry Pi) and equip it with a Wi-Fi adapter to connect to the drone and a Wi-Fi, Zigbee, Bluetooth, SDR, or other radio to communicate with the ground vehicle.

Table 4.1: Specifications of different wireless standards.

Wireless Standard	**Data Bandwidth**	**Range** (line-of-sight)	**Transceiver power**	**Price**
Wi-Fi (IEEE 802.11g-2003)	Up to 54 Mbps	About 140 m	Up to 125 mW+	$20+
Wi-Fi (IEEE 802.11n-2007)	Up to 600 Mbps	About 250 m	Up to 700 mW+	$30+
Bluetooth	Up to 2.1Mbps (2.0+ EDR) Up to 24Mbps (3.0+ EDR)	Up to 60 m	100 mW (Class 1) 2.5 mW (Class 2) 1 mW (Class 3)	$20+
ZigBee (IEEE 802.15.4)	Up to 250 kbps	10-100 m (Up to 1500 m)	1, 2, 30, 50, 60 mW (up to 250 mW)	$20+

Due to the team's concerns over the AR Drone payload capacity, controllability, and reduced flight times of implementing such solutions, the team decided to use the Wi-Fi radios on the drones to communicate with them, and conduct all of the position control and data processing on the ground vehicle. The advantage of the ROS environment is that the AR Drone API and functions could be offloaded to a computer attached to the drone, or replaced with another software stack as needed to accommodate other drones or configurations.

4.1.3 Power Management Constraints

The electronics and electromechanical systems added to the ground vehicle demanded consideration regarding the ability of the vehicle to power itself and the added peripherals for periods of extended testing. An internal combustion engine, powering the vehicle drivetrain or solely generating power, was disallowed due to safety concerns over storing gasoline and hot engines (see Section 4.3.4, Safety). When compared to an internal combustion engine powering a generator, however, the low energy density of the battery system on the ground vehicle leaves a minimal overhead for accessories before the performance of the vehicle is negatively affected. Additionally, the time needed to recharge the batteries would potentially interfere with the team testing the system, and as such required either conservative power use or supplementation.

Supplying power to the computer was the first aspect the team addressed as it had the most potential to discharge the vehicle quickly. The two ways to power the computer from the batteries on the vehicle involved adapting the 48V DC vehicle power to either a 120V, 60Hz AC mains source, allowing the power supply of the computer to be plugged in with no modification, or sourcing or building a 48V-powered, ATX compatible power supply and installing it in the computer. Both such solutions exist, but were ruled out due to their cost. As the constraints for the rest of the project narrowed, the decision was made to provide power to the computer and other mains-powered devices via an extension cord long enough for the vehicle to complete its course.

For the electromechanical systems mounted to the ground vehicle, the least expensive options for actuators and motor drivers consistently required 12V or 24V (or up to 30V) instead of 48V, as most hobbyist-level and light industrial hardware operates at these lower voltages. Additionally, motors and linear actuators capable of manipulating the controls for the ground vehicle were rated at drawing more current than most cheap step-down voltage regulators would

supply. Many of the high-current solutions required the construction of some manner of mounting that would provide sufficient heat dissipation. To keep the system simple, the team chose to purchase 12V batteries, manipulators, and controllers rated for 12V.

By isolating the computer and actuator power systems from that of the vehicle, its failsafe mode could be implemented to cut power only to the drive motor of the vehicle, while the electromechanical system can still steer and actuate the brakes as necessary. The computer can also continue operating without the need to re-initialize the software after every emergency stop, preserving the state of the executing programs to better diagnose the problem and fix it.

4.2 Proposed Design

The project began with a basic idea of purpose and goals which were proposed to the project team. The team then translated these goals into an overall system design and had to determine tasks for each robot in the system to perform. The project team identified many options for completing the desired tasks of this project. These methods of completion were developed and analyzed for efficiency, complexity, and ability to effectively complete the tasks of the project. The proposed methods and chosen project design for each element are described in the respective sections below.

4.2.1 Aerial Platform

Because of its ability to fly out of the box, the HD camera, the basic sensors, the relatively inexpensive price, and the availability of prewritten code, the team decided to purchase and implement the AR Parrot Drones for the aerial vehicle of the project.

The team had many options for the implementation and use of the aerial vehicles in this project. The first of these uses would be to simply use the drones to assist with the mapping of the environment. Due to the quality of the drones and the difficulties associated with monocular SLAM, the team decided not to attempt to use the drones as an additional method of populating

the global map. Another method of use would be to have all drones search for hazards in a general area ahead of the vehicle. However, accurate flight of the drone is very limited by the battery life and thus the team determined this method to be inefficient.

The final decision on implementation was to use the drones to detect hazards along the path planned by the ground vehicle. The drones are dispatched one by one and transmit image data back to the main computer over Wi-Fi. This image stream is processed in MATLAB using blob detection. As the drone navigates to waypoints along the planned path, the blob detection code searches for blobs of pixels which fall within a predetermined color range of orange. Once a blob of a certain size is identified, the drone alters its flight path to the direction of the blob. Once it was determined that the drone is a desired distance from the cone, based on the size of the blob in the total frame of the image, the drone then lands and sends its location, based on its IMU data, back to the main computer. The location of the drone, and thus the identified cone, is populated in the global map. At this point, the next drone is dispatched to the next waypoint along the planned path of the vehicle and continues the process of blob detection and cone location as described.

4.2.2 Ground Platform

In order to control the steering and braking of the electric golf cart, the team decided to use two linear actuators, controlled by a Sabertooth 2x60 Motor Controller. In determining the required load rating of the linear actuator to be used for the steering, several test measurements were taken of the force required to move the steering rack from either lateral limit. These test measurements yielded an average of 118.1 lbs. Unfortunately, the team could not find any actuators between the load ratings of 110 lbs and 560 lbs, so the latter of the two was selected. The displacement of the steering rack from one lateral limit to the other was also measured, as to determine the stroke size of the actuator. With a displacement of 6 inches, the team decided to

purchase a 560 lbs linear actuator with a 6 inch stroke for the steering. Similar measurements were taken of the braking system on the cart. The average force required to move the brakes into a fully depressed position was about 94 lbs and the displacement of the braking clamp was measured to be 4 inches. The team decided to purchase a 110 lbs linear actuator with a 4 inch stroke for the braking. Each of these linear actuators is equipped with a reference potentiometer for feedback positioning.

The throttle control of the golf cart is adjusted by a three-wire potentiometer. When the accelerator pedal of the cart is in its upright (rest) position, the resistance of this potentiometer ranges from 0 to 300 ohms. When fully depressed, the cart enters a full-speed operation with a potentiometer resistance of around 5500 ohms. In order to autonomously control the throttle of the cart, the team decided to use a digital potentiometer to mimic the accelerator potentiometer. The resistance of the digital potentiometer would be adjusted in increments of 50 ohms from 0 to 10k ohms by using a Teensy 3.0 microcontroller, thus controlling the speed of the electric golf cart.

Sensors are installed on the golf cart to collect information about the environment and its movement in it. These sensors include a stereo camera system, a Global Positioning System (GPS), and an Inertial Measurement Unit (IMU). The information collected from these sensors is fused together using a Kalman filter in order to provide position estimation used in the SLAM and path planning algorithms.

The stereo vision system consists of two Raspberry Pi cameras. Each camera is connected to a Raspberry Pi which transmits compressed image data to the main computer over ethernet. This setup was chosen in part because the price was comparable to the price of buying two webcams, while retaining similar levels of customizability (if not more) than comparable

webcams. The biggest deciding factor was in the USB bus design, where connecting more than one webcam to one bus (which could be serving the entire computer) would saturate it and cause frame rate issues. There are also drivers written in ROS (Robot Operating System) for the Raspberry Pi cameras, and while the majority of webcams have functioning Linux drivers, the capability or stability of these drivers can be limited as they lack official support.

Once the camera data has been transmitted back to the main computer, position measurements are processed using a Kalman filter. In a process called data fusion, the filter fuses position measurements recorded from the stereo system, GPS, and the IMU to find an estimated position. The position measurements from the IMU are obtained by removing the gravity vector and compensating for accelerometer drift before the data is passed to the Kalman filter. The filter combines all measurements to form an estimated position that is used for the vehicle localization. This step is key because, at any given time, not all position measurements may be accurate or available (ie operating GPS indoors or with too few visible satellites), and Kalman filter will be able to account for this.

4.2.3 Decision Making and Control Logic

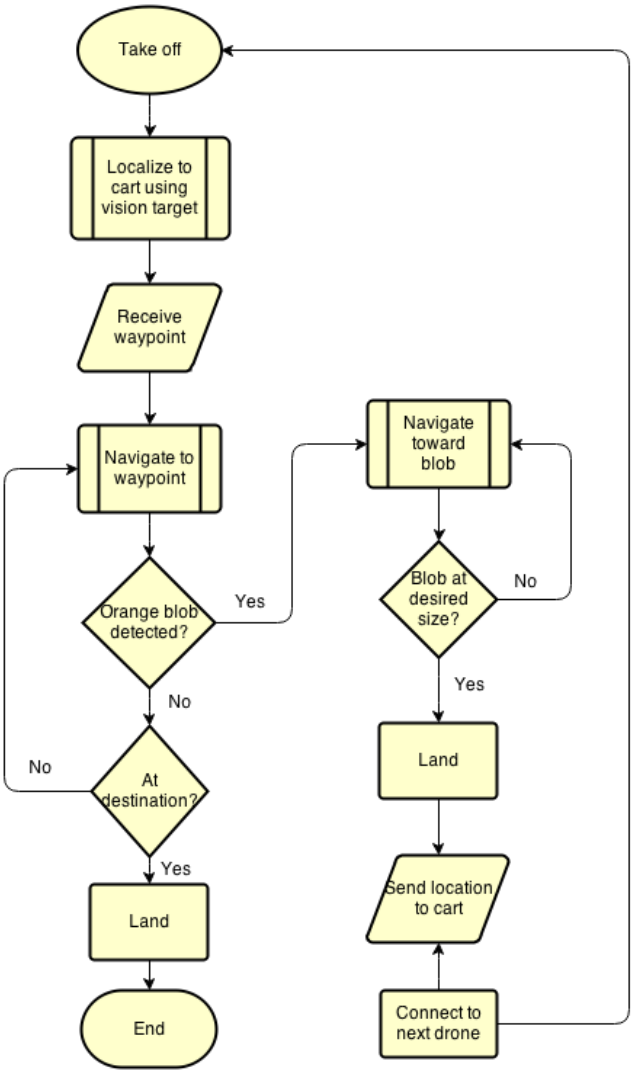
The decision making process is computed on the main computer, which is mounted on the rear of the golf cart. This computer is responsible for path planning and control of the drones. The main computer will evaluate data from the drones and sensors on the golf cart to find a path to a target.

4.2.3.1 Path Planning

The most important decision that the system makes is how to get from where it is to where it needs to be. It will do this by employing a path planning algorithm that will optimize a path based on distance to the destination. The team decided to use the A* algorithm because this algorithm optimizes for the shortest distance to reach the goal [41]. There are other algorithms

that will plan a path based on risk, but these are not the most efficient choice for this application. Risk can defined by how close the vehicle comes to other objects in the map. Optimizing for least risk means that the vehicle would plan to stay as far away as possible from other obstacles in the map. Based on the environmental assumptions, there will be no obstacles that can damage the cart or people to hit, so algorithms which take these factors into account are not applicable. When performing path planning, it is important to make a path that the vehicle is capable of following. Knowing this, an understanding of the vehicle's physical limits must be had. This project uses a vehicle with Ackerman steering, which means that the vehicle has a turning radius greater than 0 degrees. Taking this into account can increase the complexity of planning a path through the environment. According to the project constraints, the path also cannot include motions that would require the vehicle to go in reverse, even though the golf cart is technically capable of driving in reverse. This project only equipped sensors that are forward facing, so it has no information about what is behind. Knowing this, the control system was designed without reverse functionality, and the golf cart needs to be manually put into reverse.

Figure 4.3: Decision making process of the drones



Once the path is created, a series of ROS Ackerman messages is sent to a Teensy which parses the messages and performs the desired action.

4.2.3.2 Drone Flight Decisions

The drones use IMU data and visual data to make navigational decisions. Waypoints are received from the vehicle to which the drone navigate by calculating distance traveled from the IMU data. The drones navigate from waypoint to waypoint until an orange blob larger than a set minimum size is detected. At the detection of a blob, the drone centers the blob in the field of view of the camera. The drone then navigates to the detected blob, presumably a cone, until the blob reaches a designated size in the image. At this point, the drone lands and sends the calculated location of the cone back to the main computer to be registered into the global map. This process is continued with the next drone. Once a drone reaches the desired final position of the cart, the process ends. Figure 4.4 provides and in-depth flow diagram of the decision making process that the drones follow to perform the desired tasks of the aerial platform.

4.3 Project Logistics

In order to assure that the project was making progress in a timely fashion, the project team created timelines to establish an idea about the amount of time to designate to each task. These timelines used WPI's academic calendar as a reference, setting major milestones to be due at the end of each academic term. From there, smaller, more frequent goals were created so that the team could monitor its progress. The following Gantt charts show the projected timeline for the project, broken down into three terms. The first term was focused on the planning stages of the project, while the subsequent two were strictly for fabrication and testing.

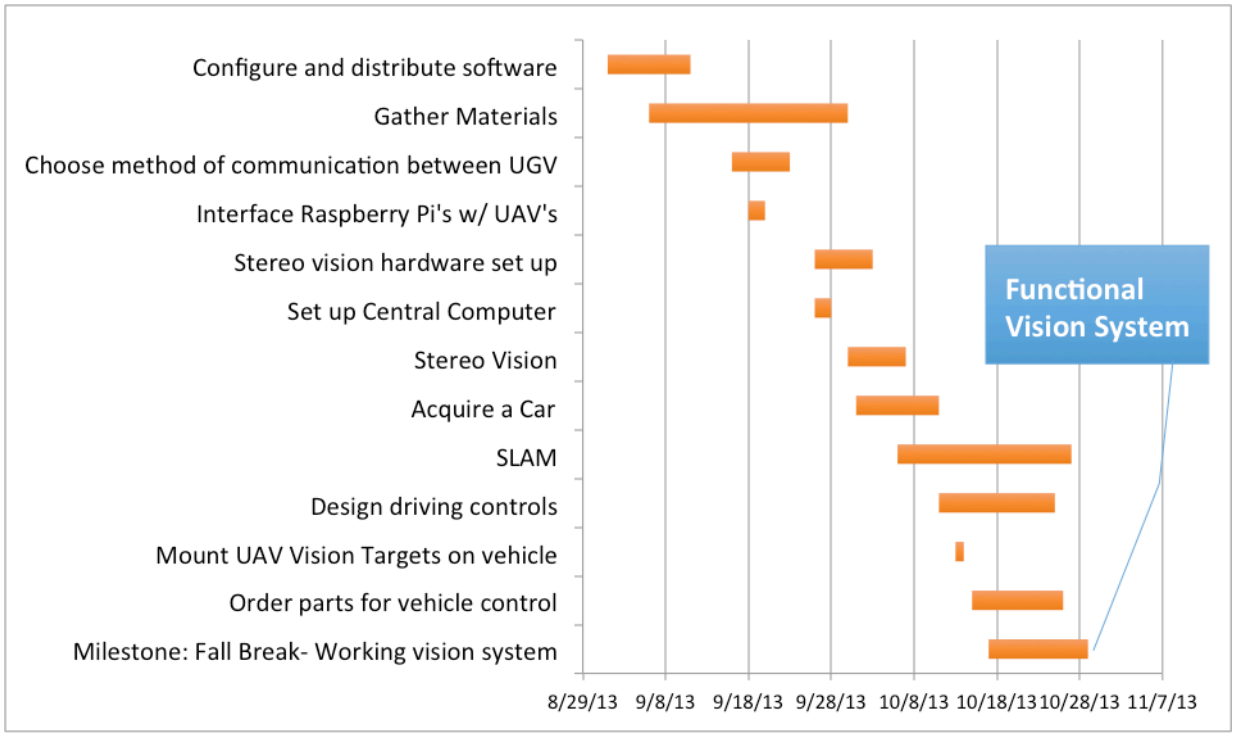


Figure 4.4: Proposed project timeline for the first quarter of the project.

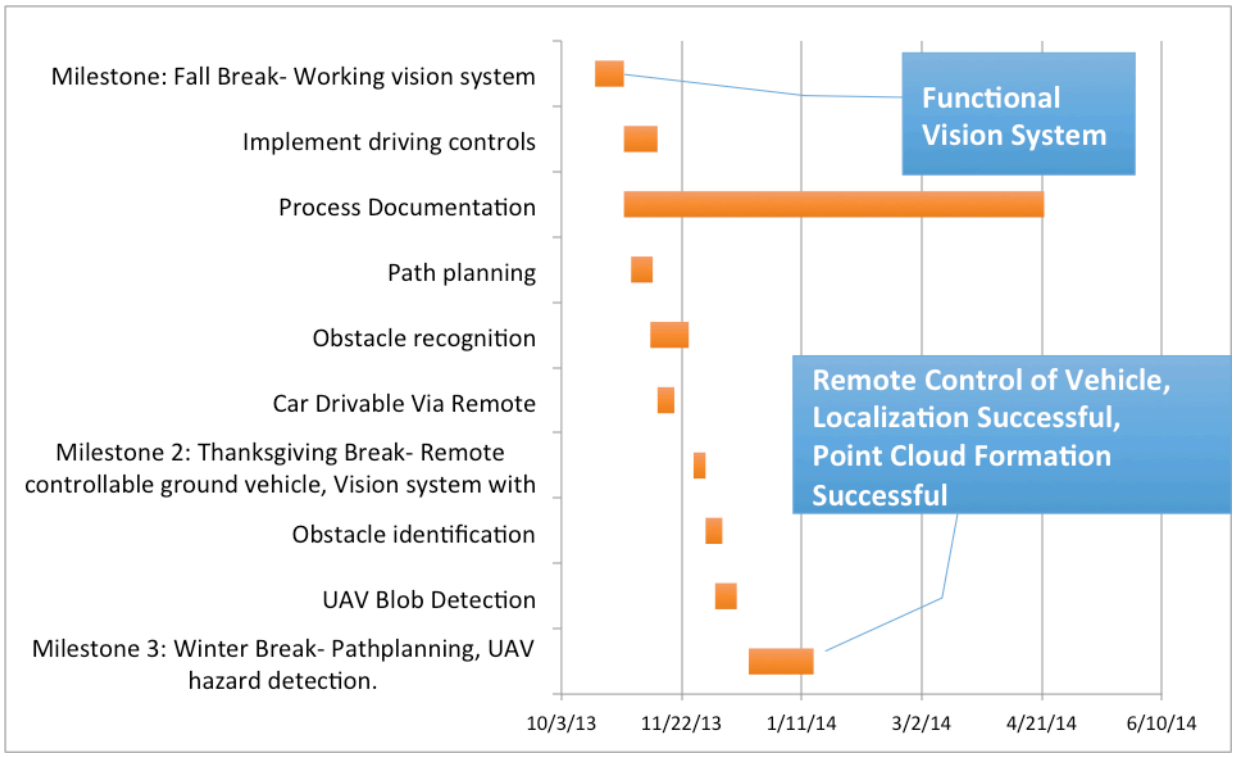


Figure 4.5: The proposed project timeline for the second quarter of this project.

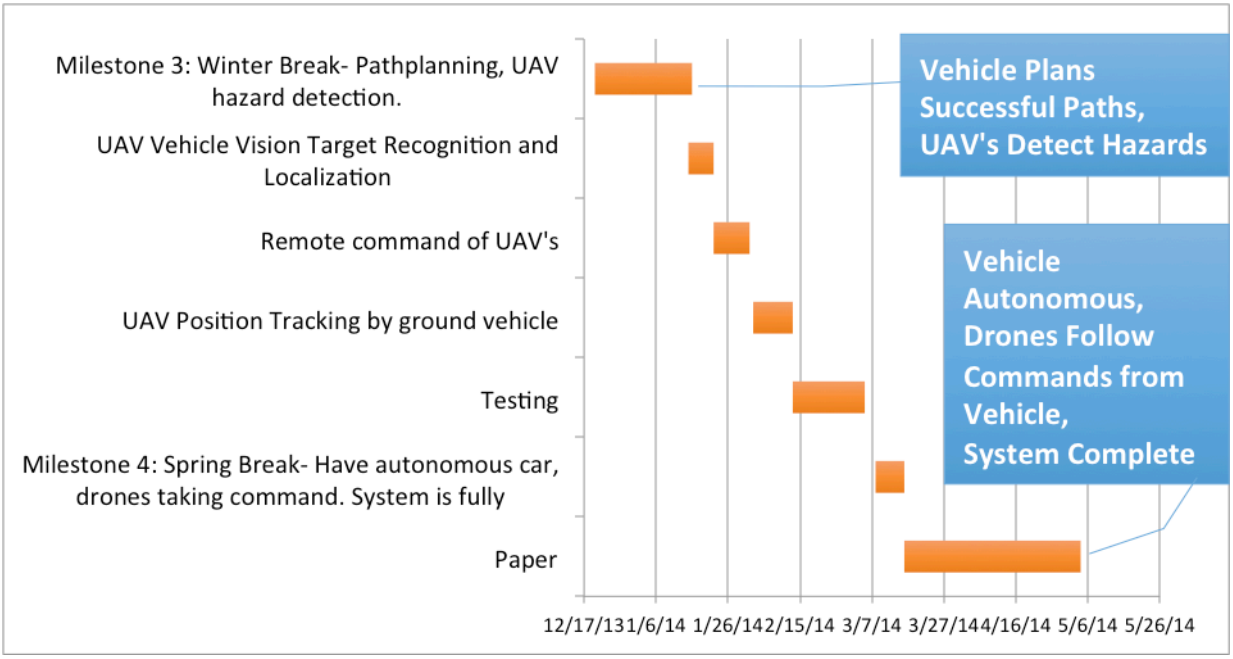


Figure 4.6: Proposed project timeline for the final two quarters of this project. The final quarter was designated specifically for paper writing, and little to no technical work was to be done.

A detailed list of all materials purchased has been attached in Appendix A of this report. This list outlines the specific components that the team chose, along with the price of each purchase.

4.3.1 Financial Considerations

Financial considerations had a large role in the selection of sensors and ground vehicle. There was a choice made to not use Lidar in order to stay within the budget due to the cost of such a system. There was a laser range finder available, but the maximum range was about 5 meters, and the cost was approximately $1100 before shipping. This unit was not used because the range was not long enough for an outdoor application. The next set of Lidar units cost between $6000 and $10000 dollars. While the range for these units were up to 80 meters, the cost would be most if not all of the budget.

With this knowledge, the decision was made to use cameras as a primary sensor on the golf cart. One camera option was to buy pre calibrated stereo cameras. Stereo cameras can be

two cameras separated by some distance with overlapping fields of view. These pre-built systems can come with built in software, but they are very expensive compared to the cost of building a system with off the shelf components. The decision was made to build and calibrate a custom stereo system because the price to implement a custom system was cheaper than a pre-built system, and there is open source software available for stereo systems.

Other than sensor selection, money influenced vehicle selection. With the available budget, buying a car was not feasible because overhead costs such as repairs and insurance. This led the team to choose between a gas powered go kart, power wheels, and a golf cart. The decision was made to use a golf cart due to non-financial factors that will be discussed later in the chapter.

Since the original quote of the vehicle was $3,000, the budget for the computer was set $1,500 so there would be money left over to cover shipping, sensors, and other parts that are required for this project. The main computer was designed to have a fast processor with plenty of RAM. The hard drive did not need to have a large capacity because the computer only needed enough hard drive space for the operating system and installed software.

4.3.2 Time Considerations

In order to complete this project within the allotted period of time, the team realized that it would be impossible to design and build everything from the ground up. Knowing this, there were some considerations in the design made in the interest of time. Some examples of these are using pre-existing libraries, buying a golf cart, wireless communications protocols, and the selection of a drone.

The amount of software that would need to be written to complete a project of this magnitude cannot be written, tested, and documented by five people in less than eight months. To get around this, many pre-built software libraries were used. This benefited the project

because it reduced the amount of code that needed to be written, with the downfall being that there is a learning curve to using libraries and this can prove to be difficult to use if the libraries are poorly documented.

The main options for drones were to build a drone(s) for the project or to buy and implement a pre-built drone. Building a drone would take time that is not available. Building a drone could potentially also cost more money, but it could be controlled to remain within budget. Buying a drone ensures that the project has a tested and proven platform, but control software could be closed source. In some cases however, an API is provided giving the ability to send commands to the drone. The team decided to buy a drone to save time because after discussion building, testing, and writing software for the drone could turn into another project on its own.

4.3.3 Workspace

Another logistical concern that the team faced during vehicle selection was finding adequate workspace for the ground vehicle. If the team were to explore the Power Wheels option, the vehicle could likely be stored in a room in Atwater Kent Laboratories on the WPI campus because of its smaller size and lighter weight. This option was later disregarded, despite the potential simplicity of storage, due to concerns of the durability of a plastic frame that was designed as a toy to carry small children, and not the constant weight of computers and a control system. Entertaining a gas-powered go-kart introduced the issue of finding a properly ventilated location at WPI. Also, attempting to store a vehicle with an internal combustion engine only added further complications due to the emission of exhaust fumes while running and the increased risk of fire caused by a hot engine if kept running for an extended period. Naturally, initial concerns of storage led the team away from the go-kart option, and towards that of an electric golf cart. The size of each vehicle is comparable, so the main difference in storage of the golf cart as opposed to the go-kart was the lack of ventilation requirements.

4.3.4 Safety

In order to conduct this project within the safety regulations of the campus, maintaining a safe testing and storage environment played a key role in the implementation of this project. The team assessed the potential risks that were specific to our project and developed methods of avoiding and responding to specific incidences.

In assessing pedestrian safety, most of the attention was paid towards enabling human override of the ground vehicle. The golf cart is equipped with three different emergency stop buttons, which can be used to halt vehicle operation should it become dangerous. These buttons kill power to the electronics on the vehicle thus effectively ending all operations. The buttons are located on the front of the cart, in the foot well within reach of the driver's foot, and on the back of the cart. In addition, the method of attachment for the linear actuator controlling the brake cable allows for the brakes to be used manually at any time by the driver without interfering with the operation of the actuator. This is an important inclusion to the braking system because the actuator requires power to actuate and thus if an emergency stop is initiated, the actuator will remain in the current position at the time of the emergency stop. At this point, it is important for the human operator to have the ability to manually engage the brakes. To ensure pedestrians do not unintentionally endanger themselves in proximity to the vehicle, there is a safety light atop the golf cart, making it visible from a distance. For fire safety, one fire extinguisher was strapped on the cart and another was within reach of the non-driving team members.

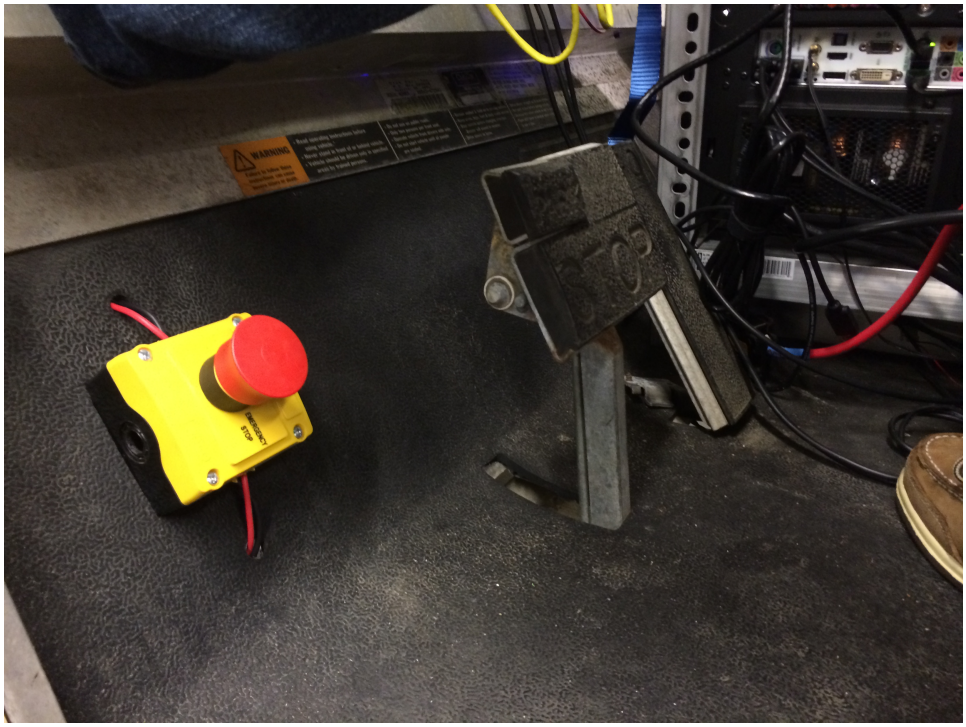


Figure 4.7: Photograph of one of the emergency stop switches installed in the vehicle. When the switch is activated, the powertrain of the vehicle is disabled, and the brakes are engaged to prevent injury.

Safety precautions were also implemented for the drones. During testing, team members wore gloves to protect their hands in the event that the drones had to be physically caught and disabled. Additionally to protect the drones and the environment in which they were being tested, the drones were always flown with the indoor shell in place. There are also safety features built and programmed into the drones by Parrot. The drones will enter an emergency state if the IMU senses a rotation beyond a pre-programmed angle. Additionally, an alarm sounds and the drone LEDs signal during flight if the drone battery drops below a pre-programmed level. Finally, in the event of a failure where the drone cannot be reached by a team member, the drones are programmed to automatically land when the program is terminated.

4.4 Chapter Summary

Based on the test environment the group decided upon, the proposed a design that consisted of one autonomous ground vehicle (a golf cart), and four quadrotors (AR Parrot 2.0 drones). The system would be tasked with navigating an obstacle course under controlled conditions, and the system must be outdoor capable. The team is assuming a safe flying environment for the aerial agents, and a low risk environment with no moving obstacles for the ground agents.

5 Ground Vehicle Hardware

Before the electric golf cart could be controlled autonomously, the interfaces that manipulate the steering, braking, and throttle of the cart had to be accessible by the computer. This meant that the team had to retrofit a custom robotic solution to fit the existing mechanical systems on the golf cart in order to electronically control the vehicle. This chapter will break down the modifications made to the vehicle into the four major challenges: steering, braking, throttle control, and safety systems. In each section, the evolution of the design is followed, from initial implementation prototypes, up to the final installation.

5.1 Steering

After evaluating all of the potential solutions, the team decided to implement a linear actuator as the means of controlling the steering. Given the force requirements gathered after measuring the force needed to manipulate the steering when stopped, the team selected a 560 lb linear actuator with a 6 inch stroke in order to power the steering. The plan was to replace the original rack and pinion on the golf cart with this actuator.

The team encountered several problems with this idea, once the cart and the actuator were actually acquired. The first issue to be realized was that this design would eliminate the possibility of any manual steering once the actuator was implemented. Provided that the cart would have to be moved to a proper testing location from the workstation, a lack of manual control would prove to be troublesome. In attempting to find a workaround for this, the team noticed that there was a separate spot on the steering rack shown in Figure 5.1 where the actuator could potentially be linked to, as to turn the wheels of the cart.

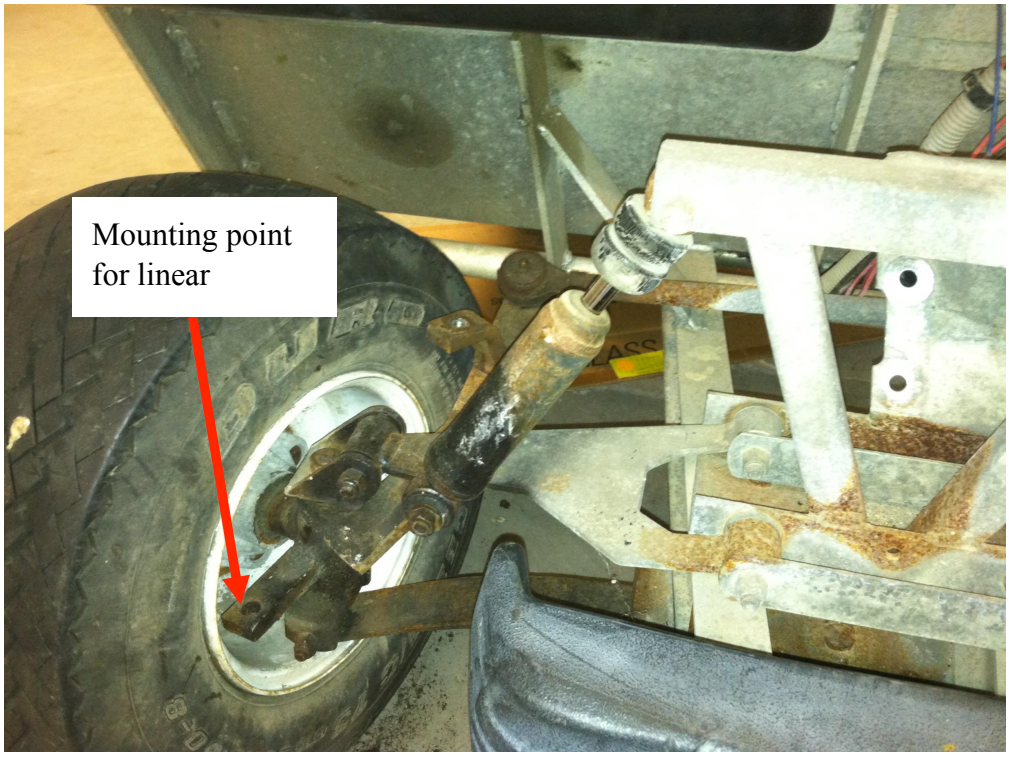


Figure 5.1: View of the mounting point used to control the steering using the linear actuator. In connecting the linear actuator here, the original steering can remain installed.

5.1.1 Actuator Selection

The potential mounting location was in the interior of the front passenger-side wheel. In attempting to connect the actuator to this point, the team found that it was too bulky to fit properly underneath the front bumper. Furthermore, the actuator was 17.72 inches in length while the distance from the interior of the two front wheels was 24 inches.

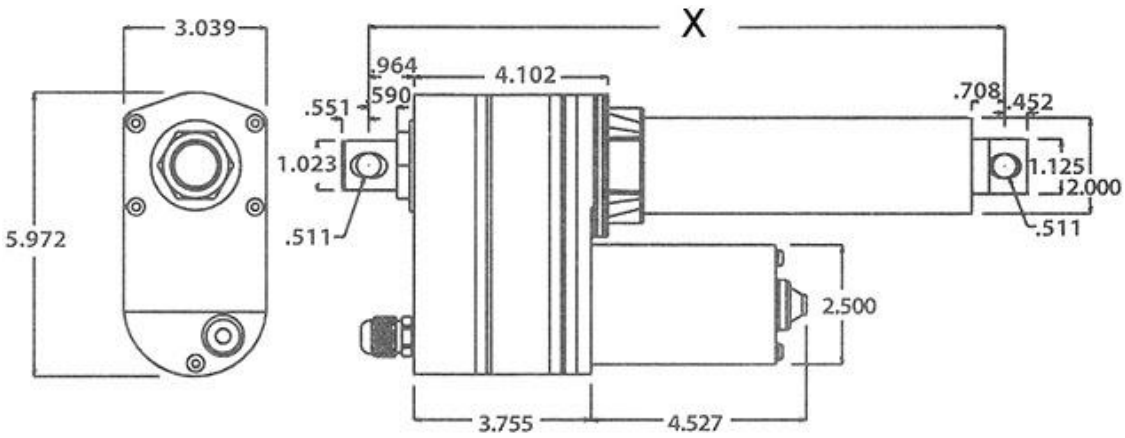


Figure 5.2: Diagram of the large linear actuator the team intended to use for the steering (in inches).

In the optimal position for the 560 lb actuator, the team could see that the wheels would bump into the actuator after moving just about an inch. As seen in Figure 5.3, the actuator was too large for use.

Figure 5.3: The 560 lb linear actuator being held underneath the front bumper, with the actuator arm nearing the mounting point. The actuator barely fits and since there still would need to be a connection piece from the actuator to the steering rack, was deemed too large for use.

The team began to search for a smaller linear actuator capable of applying more than the 118.1 lbs of force required to move the steering rack with a 6-inch stroke. Consequently, the team acquired two 150 lb. linear actuators, one with an 8-inch stroke and the other with a 4-inch stroke.

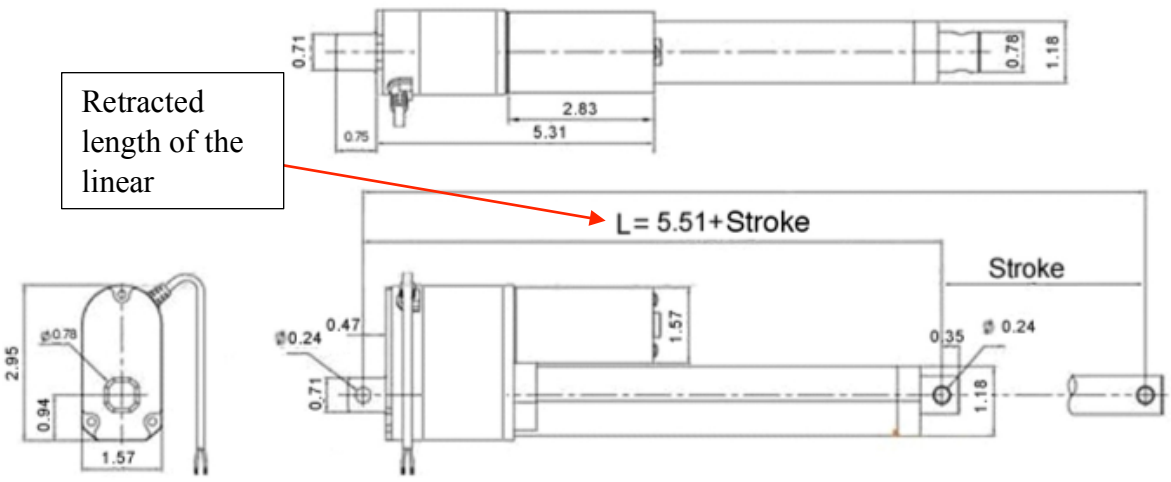


Figure 5.4: Diagram of the smaller actuator that was used to control the steering in the final revision (dimensions in inches).

Table 5.1: Table of different available actuator lengths. The 150 lb actuator with a stroke size of 4 inches is 9.51 inches in length when fully retracted, a whole 8 inches shorter than the 560 lb actuator. The actuator with a stroke size of 8 inches is 13.51 inches when fully retracted, 4 inches shorter than the 560 lb actuator [49].

Stroke Size (inches)	Length, Fully Retracted (inches)	Length, Fully Extended (inches)
0	5.51	5.51
1	6.51	7.51
2	7.51	9.51
3	8.51	11.51
4	**9.51**	**13.51**
6	11.51	17.51
8	**13.51**	**21.51**
9	14.51	23.51
10	15.51	25.51

As can be seen in Table 5.1, each of these actuators were significantly shorter in length than their 560 lb counterpart when fully retracted. Each of the actuators also fit underneath the front bumper and did not impede the motion of the wheels at any point in their range of motion, making the stroke size the deciding factor between the two options. It was determined that full range of the wheels is not necessarily needed in order to efficiently turn the cart, especially if operation is done at a slow pace. It also appeared safer to compromise 1 inch of stroke on either side of lateral movement of the steering rack than to have 1 inch more than enough on either side and risk damage to the linear actuator if something went wrong in controlling the length of the

actuator arm. The team determined that the 150 lb linear actuator with the 4 inch stroke size was the best option of the two linear actuator solutions.

5.1.2 Steering Mount Prototypes

The next major challenge for implementing the steering mechanism was to find a way to connect the linear actuator to the steering rack. At first, the team tried to use two male-threaded shank ball joint linkages such as the one depicted in Figure 5.5. A bar of aluminum stock was manually threaded on each end in order to screw in the stud ends of the joints. Each of the shank ends had a nut on the other end in order to secure its place as one rested in the steering rack hole and the other in the hole of the linear actuator.

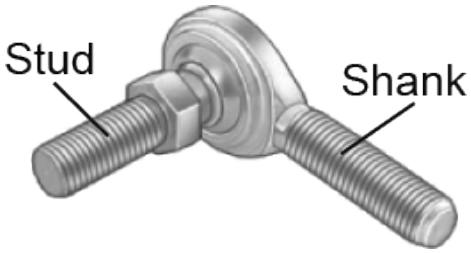


Figure 5.5: Initial tie-rod end implemented to connect the actuator to the wheels. The male shanks connect to mounting points on the steering rack and the linear actuator. The studs screw into a threaded bar of stock in order to connect the two joints.

The team created a mount for the linear actuator using an aluminum plate in order to mount it onto the cart. The plate was installed below the suspension on the cart, as depicted in Figure 5.6. Preliminary prototypes involved bending the plate at the front, as to avoid the need to cut out part of the steel frame of the cart.

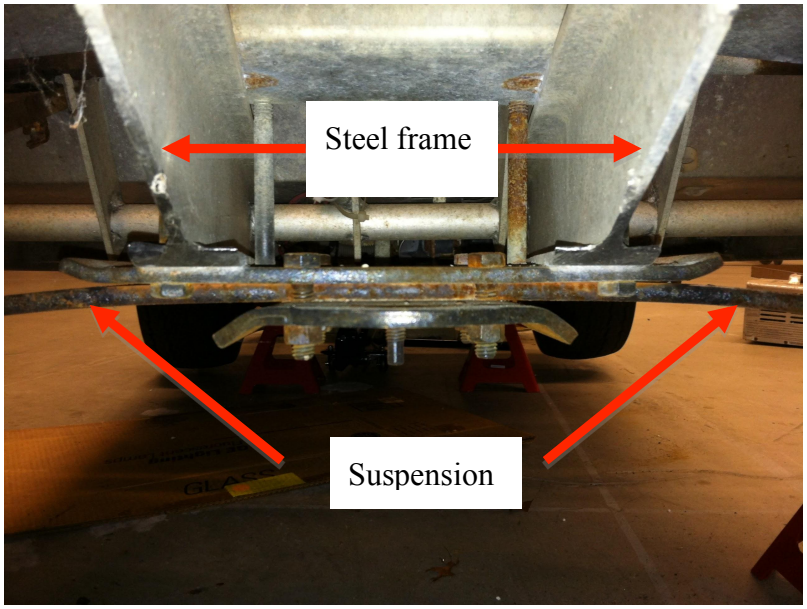


Figure 5.6: Front view of the location that the linear actuator was mounted to.

In preliminary testing, the cart was jacked up off of the ground and the linear actuated steering worked as expected. However, when the cart had all four wheels on the ground, problems were experienced. Pulling the steering rack from its rightmost position to its leftmost resulted in no trouble whatsoever. However, when attempting to push the steering rack in the opposite direction, the linear actuator arm began to twist. After further research was done about the linear actuator, the team learned that it is not built to withstand any significant amount of torque. This meant that in order to refine the design, the angle of the connection piece from the actuator arm to the rack and pinion must be reduced.

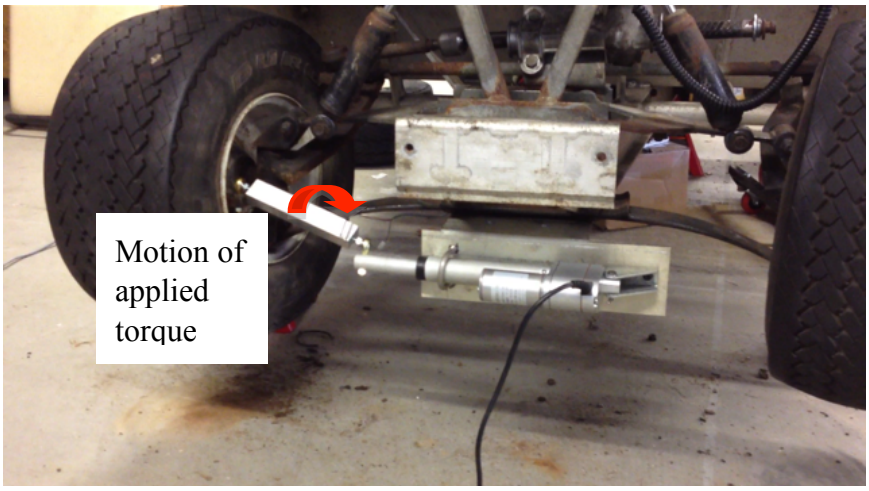


Figure 5.7: The initial system used to connect the linear actuator to the wheels. Ended up being too weak and introducing too many angles, creating unwanted torques.

The best solution to do this was to redesign a flat version of the actuator mount plate, and to cut out a section of the steel frame such that the actuator could fit underneath. The angle of the linkage was greatly reduced, but the actuator arm continued to twist when pushing the steering rack.

Given these issues with the design, the team began to seek alternative solutions in order to counter the twisting of the actuator arm, such as the implementation of a tracked method, along with reinforcing the underside of the aluminum plate. Again, the same ball joints were used, but the method failed on the first attempt, breaking one of the ball joints. The team decided to completely scrap the tracked method, and revisit the connected linkage with new joints. The joints were selected to emulate those found on the manual steering. A slightly different ball joint linkage was used at the rack and pinion end with a female-threaded shank as shown in Figure 5.8. The stud was placed into the rack and pinion hole, and a custom threaded rod was screwed into the shank. This connected to a piece of stock that was bored and threaded in order to fit the rod. At the other end of the stock was an inline ball joint with a male-threaded shank as shown in Figure 5.9. The shank screwed into a connector piece that was threaded to receive the shank, and had a pin-drop hole to connect it to the linear actuator.

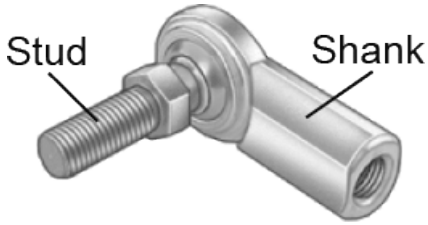


Figure 5.8: New female tie-rod end. The male stud fits into the mounting point on the steering rack. The female shank receives a custom-threaded rod that links it to the connecting stock [58].

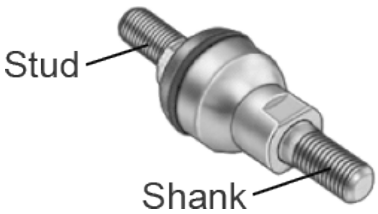


Figure 5.9: In-line ball joint. The male shank fits into the threaded stock while the male stud fits into the actuator connecting piece [58].

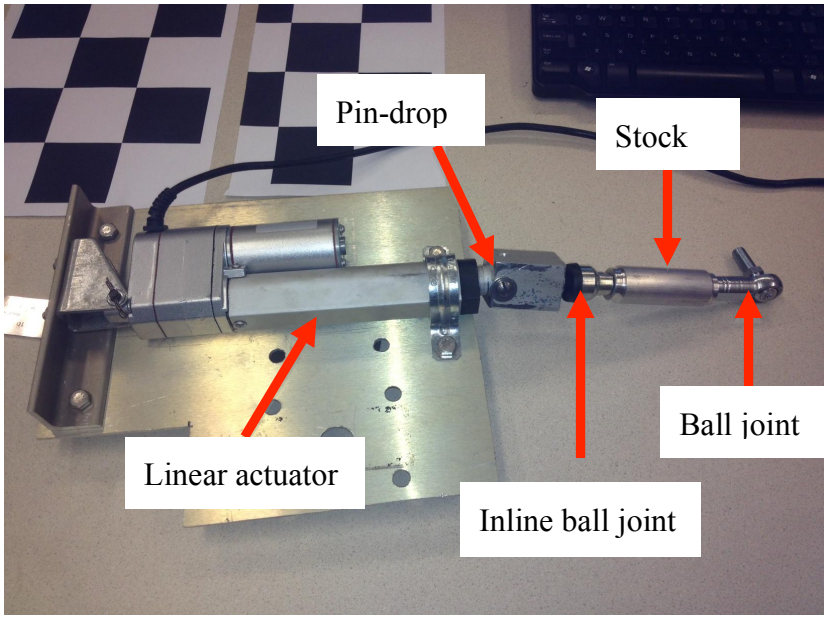


Figure 5.10: A prototype revision of the steering mount and tie-rod. From left to right, the parts include the new female-threaded ball joint, the threaded stock connector, the inline ball joint, the stock that connects to the linear actuator, and the linear actuator.

5.1.3 Final Connection Design

This new design depicted in Figure 5.10 was promising, as it emulated the original steering on a smaller scale. Installing the new mount onto the cart showed that some of the angle of the connection that was reduced by utilizing a flat plate had been reintroduced, as one can see in Figure 5.11. This was due to the lack of the ball joints being connected to the actuator and steering rack with the ball above the point of linkage. The inline joint lowered the connection, thus bringing an angle back.

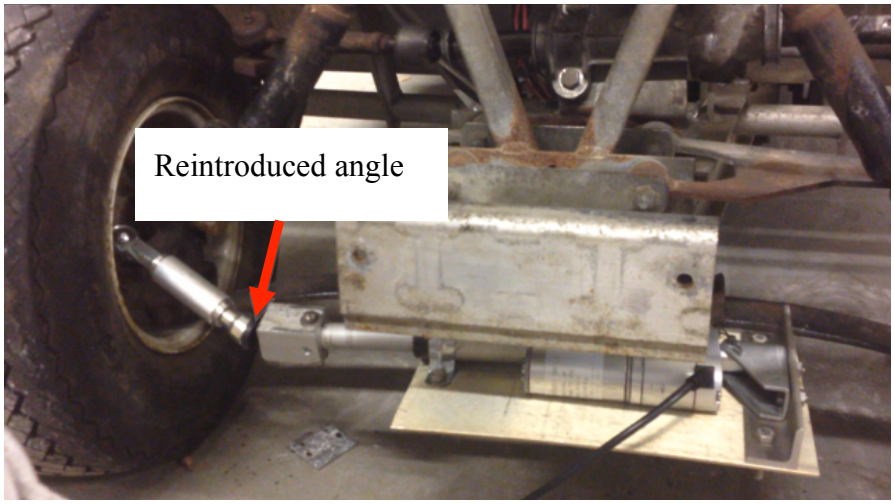


Figure 5.11: The last prototype steering assembly installed. One more revision was made after this, in favor of more durable components, and less angles.

Despite this, the linear actuator could now push and pull the steering rack, although the motion was slow. This design worked very well and was tested several times, both with the cart jacked up and with all four wheels on the ground. However, the retaining ring of the inline ball joint broke during one test and the joint was separated. Since it had been tested so often with the reintroduced angle, the ring was weakened and broke not long after. In order to reduce this angle, the team added standoffs of aluminum stock underneath the points where the actuator was connected to the aluminum plate, and cut

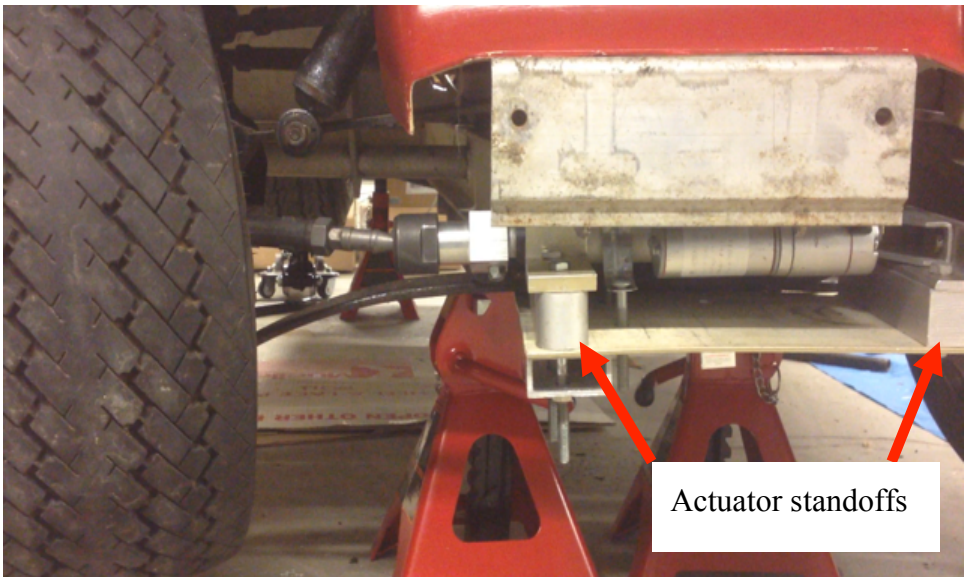


Figure 5.12: Redesigned mount with actuator standoffs. The actuator arm is fully retracted.

more of the frame out. This can be seen in Figure 5.12. The broken joint was replaced by the supplier with no questions asked, but the team wanted a more solid solution to the steering.

At this time, the team looked into purchasing similar joint linkages to the ones used in the previous design. The replacement for the female-threaded shank ball joint came from a 1996 Honda Passport and the male-threaded inline ball joint came from a 1997 Honda Accord. The two pieces can be seen in Figure 5.13, connected to the rack and pinion and juxtaposed with the previous design. This was a much more robust design, which was proved over a series of tests. As one can see in Figure 5.14 much of the angle of the connection was reduced. The team was satisfied with the results of this design, and it was ultimately used for the steering.

Figure 5.13: Final install (top) side-by-side with the final prototype (bottom). The final design has far superior strength and durability.

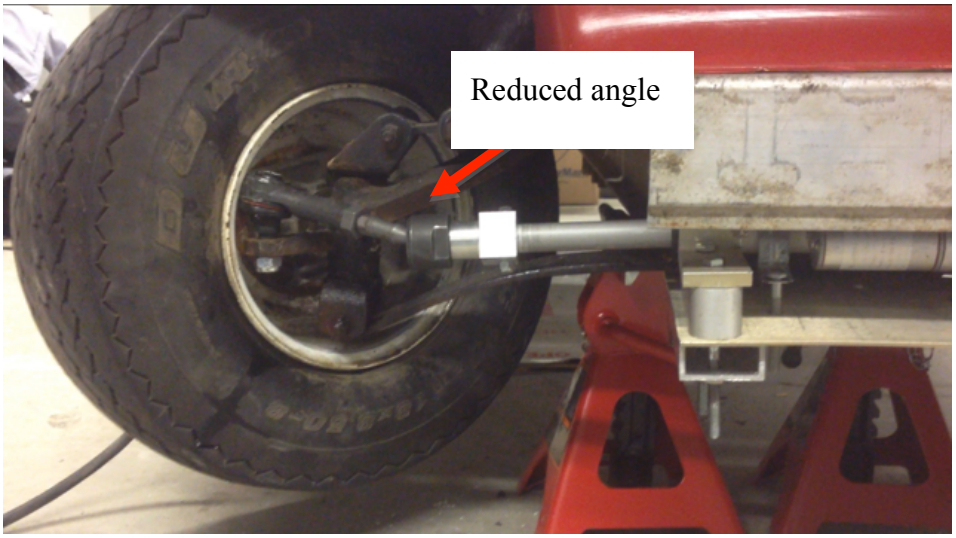


Figure 5.14: Actuator fully extended. The angle of the connection between the steering rack and the linear actuator has been reduced from the previous

5.2 Braking

The existing braking system in place on the cart were cable actuated drum brakes, located on the back axle of the vehicle. The cable is pulled when the brake pedal is depressed, engaging the brakes. The team needed a way to mimic this motion with a computer controlled device, but still wanted the physical brake pedal to be usable in emergency situations. To accomplish this, the team implemented a linear actuator that worked in parallel to the physical foot pedal. This way, if either system would be able to engage the brakes at any time.

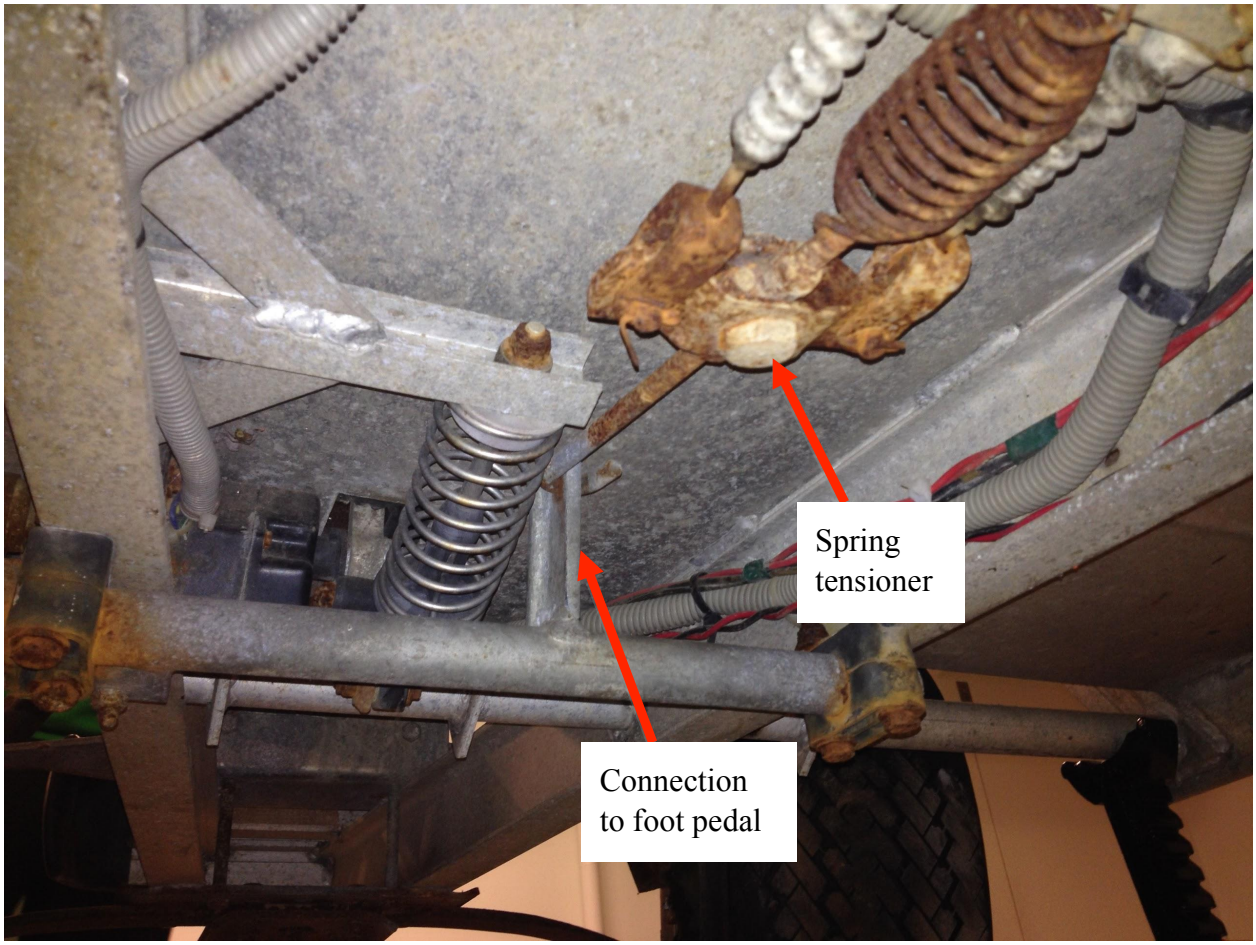


Figure 5.15: The braking mechanism in place on the cart. This shows the device that actuates the cable, applying the brakes.

The linear actuator was mounted to the underside of the floorboard in the golf cart, and was then connected to the brake cable using short length of braided steel cable. The wires to control the actuator were then run into the cab of the golf cart through a hole drilled in the floorboard.

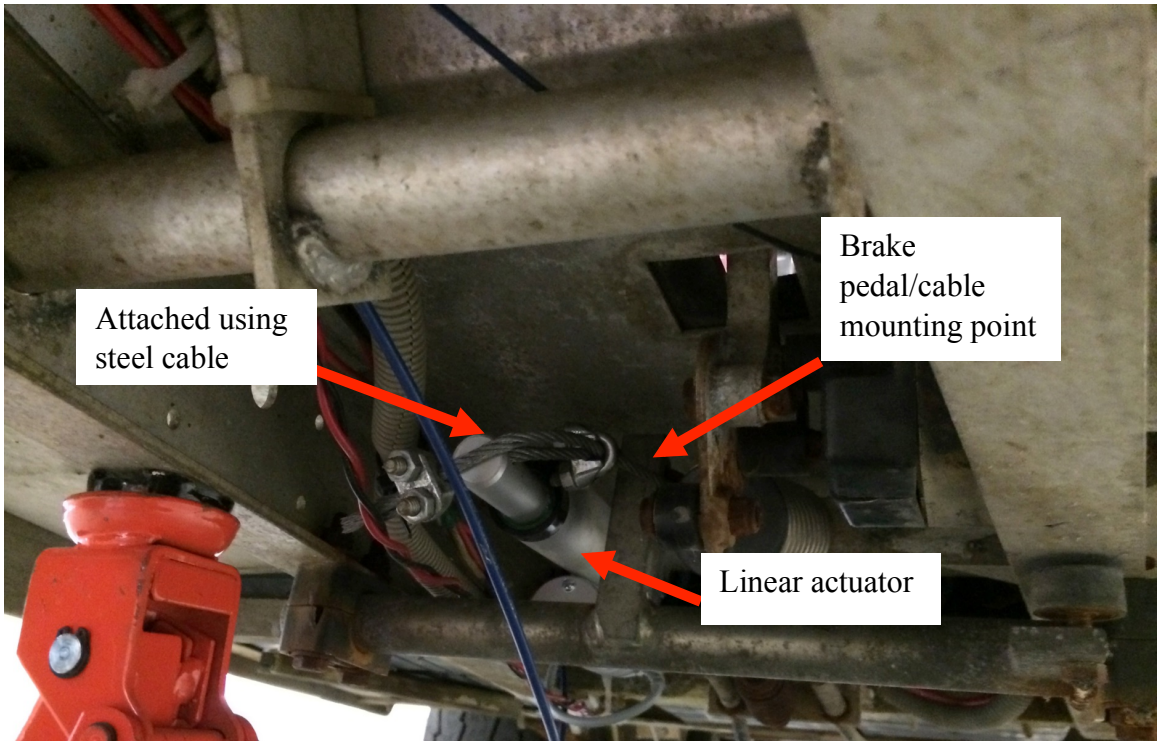


Figure 5.16: View of under the cart where the linear actuator is mounted for the braking.

In order to control the motion of the braking and steering actuators, the team designed a power system isolated from the main power system on the vehicle. A Sabertooth 2x60 Motor Controller powered by a 12V rechargeable battery was installed as shown in Figure 5.17. The Teensy 3.0 accepts commands from the computer on the cart, and relays them to the Sabertooth. The Sabertooth, in turn, controls the speed and direction that the linear actuators operate in.

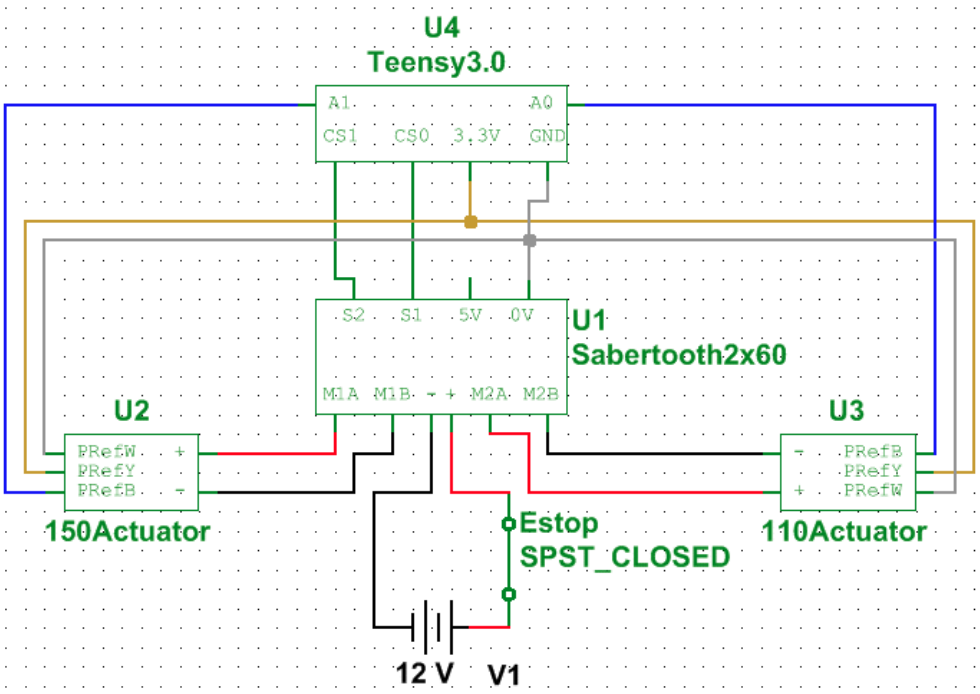


Figure 5.17: Sabertooth Motor Controller circuit. The blue, yellow, and grey (representing white) potentiometer reference wires let the computer know the position of the actuator arms. For safety reasons, there is a closed emergency switch between the battery and the Sabertooth.

5.3 Throttle

The original throttle control of the electric golf cart is adjusted by a three-wire potentiometer. The accelerator pedal controls the resistance output by the potentiometer with a range from 0 to 5500 ohms. This resistance was be measured between the purple and yellow wires coming from the continuous potentiometer.

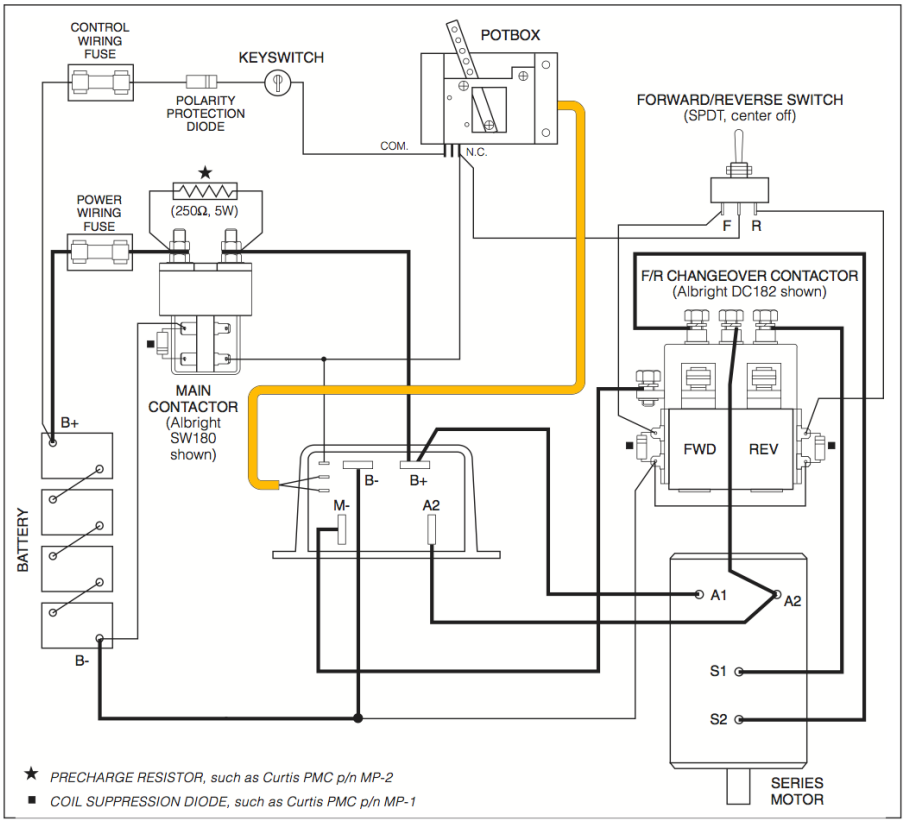


Figure 5.18: Wiring schematic of the golf cart without any modification from the group. This shows how the powertrain is connected to the battery system of the vehicle, and the location of the key switch in the wiring.

The yellow wire connected to pin 3 of the cart's motor controller. This pin and pin 2 connected across the potentiometer via limit switches. The purple wire connected to the B- pin of the motor controller, which was connected to the negative side of the power source on the golf cart, and the forward/reverse switches.

5.3.1 Throttle Control Testing

In attempting to gain control of the throttle, the team first tested the original plan of replacing the continuous potentiometer with a manual hand potentiometer. This potentiometer should have been able to represent the function of the digital potentiometer. With the wiper connected to the purple wire of the continuous potentiometer and the low resistance terminal connected to the yellow wire, there was no control of the throttle with the manual potentiometer.

The team decided to find alternate methods of controlling the throttle, while attempting to debug the potentiometer method.

The voltage across the input pins 3 and 2 of the motor controller was 7.6V when the pedal was in the rest position, but dropped to 0V when the pedal was fully depressed. The team attempted to control the throttle by applying a voltage of 7.6V across pins 2 and 3, in order to replicate the voltage drop seen with the pedal control. This did not work initially, but when the pedal was fully depressed, it worked as expected. By applying a range of voltage from 7.6V to 0V, the team had access to the full range of speed.

Despite this success, the potentiometer method was experimented with further. In lieu of the success of the voltage method, the team learned that the solution to the potentiometer method lied in making sure to complete the circuit. By linking pins 2 and 3 directly, the full range of speed was entirely accessible by adjusting the manual potentiometer. This manual method served as a proof-of-concept for the digital potentiometer method that the team wanted to implement.

5.3.2 Throttle Switch

As with the other systems on the cart, the ability to toggle between manual and computer control of the throttle was desired. The team designed and installed a throttle switch that would allow this to happen.

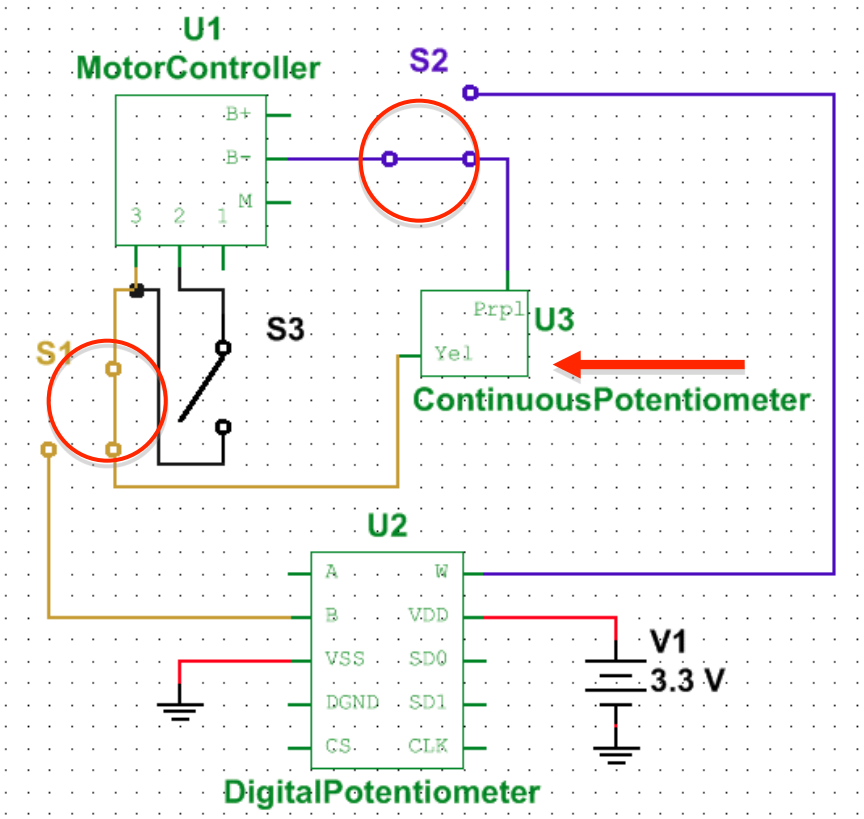


Figure 5.19: The throttle switch in "manual control" state. The original continuous potentiometer is in place.

In Figure 5.19 the original connections of the continuous potentiometer are in place, and manual control is possible. There are three switches, labeled S1-S3, that toggle between manual and digital control of the throttle. In the “manual control” state, the purple wire of the three wire connector to the continuous potentiometer is connected to pin B- on the cart’s motor controller via S2. S1 connects the yellow wire of the continuous potentiometer to pin 3 on the motor controller and the third switch (S3) is left open.

Figure 5.20 depicts the “digital control” state, in which S1 now connects pin 3 of the motor controller and the B terminal of the digital potentiometer. S2 connects the wiper (W) of the digital potentiometer and pin B- of the motor controller. Pins 2 and 3 are now connected via S3. This design effectively replaces the continuous potentiometer with the digital potentiometer.

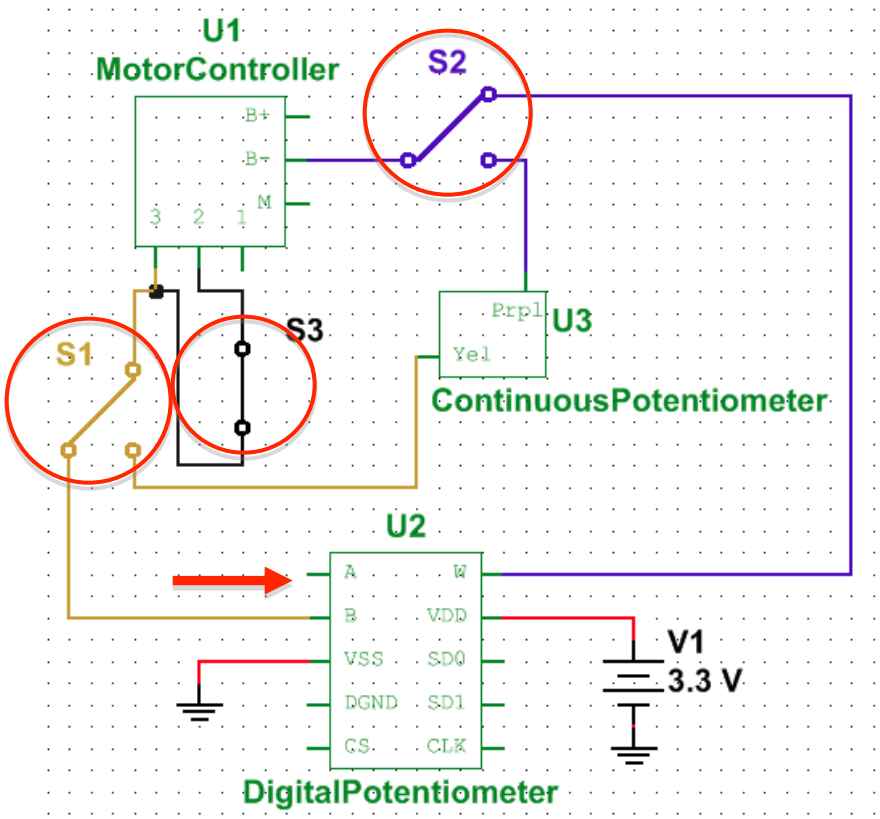


Figure 5.20: The throttle switch in the "digital control" state. The new connections to the digital potentiometer are in place, and the pins 2 and 3 are tied together on the motor controller.

5.4 Safety

Once the team began to physically implement some of the safety features of the golf cart, some minor issues arose. After the installation of the safety harness for the human operator, it became apparent that the operator would have a tough time reaching an emergency stop (E-stop) button located on the dashboard. The E-stop was subsequently moved to a location on the foot well of the cart, that could be accessed by the operator's foot (Figure 5.23). Also, The front E-stop was moved from the front bumper to a location closer to the frame because it would be safer for a team member on the ground to access this location in the event of an emergency (Figure

5.22). If the emergency stemmed from throttle control, attempting to hit an E-stop on the front bumper could result in injury.

As the project progressed, the performance requirements of the team's software grew to the point that a second computer was deemed a necessity in order to successfully implement a full software stack. The team decided to relocate the computers from the passenger's seat to the back of the cart. The electronics hardware involved in the throttle control was located in the foot well of the passenger's side of the cart.

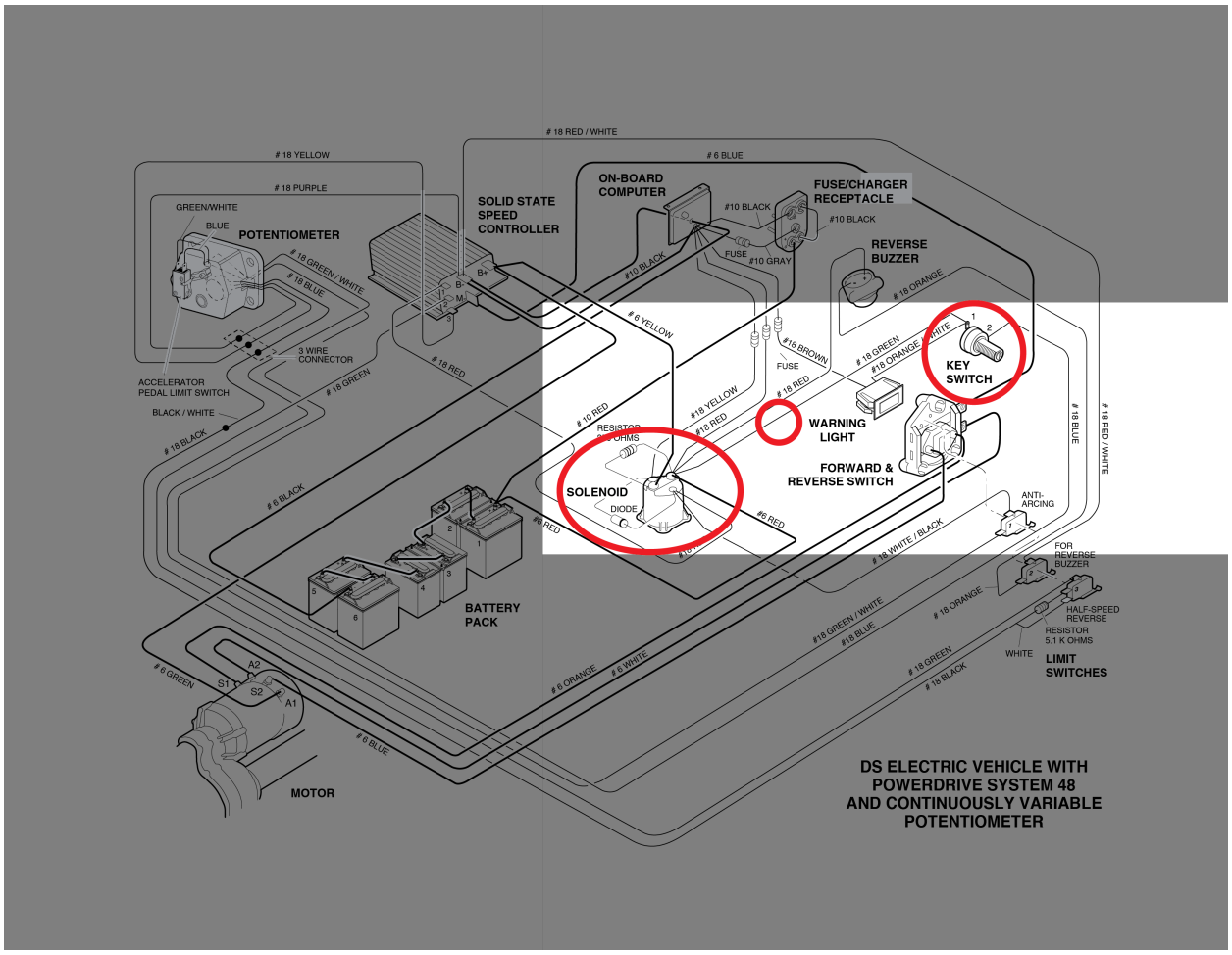


Figure 5.21: Schematic of the existing electrical system in place on the golf cart. Highlighted are the key switch and the solenoid. The emergency switch system was implemented in series with the key switch, so that when the switches were depressed, the power would be cut to the cart's drivetrain.

Figure 5.22: Emergency switch located at the front of the vehicle.

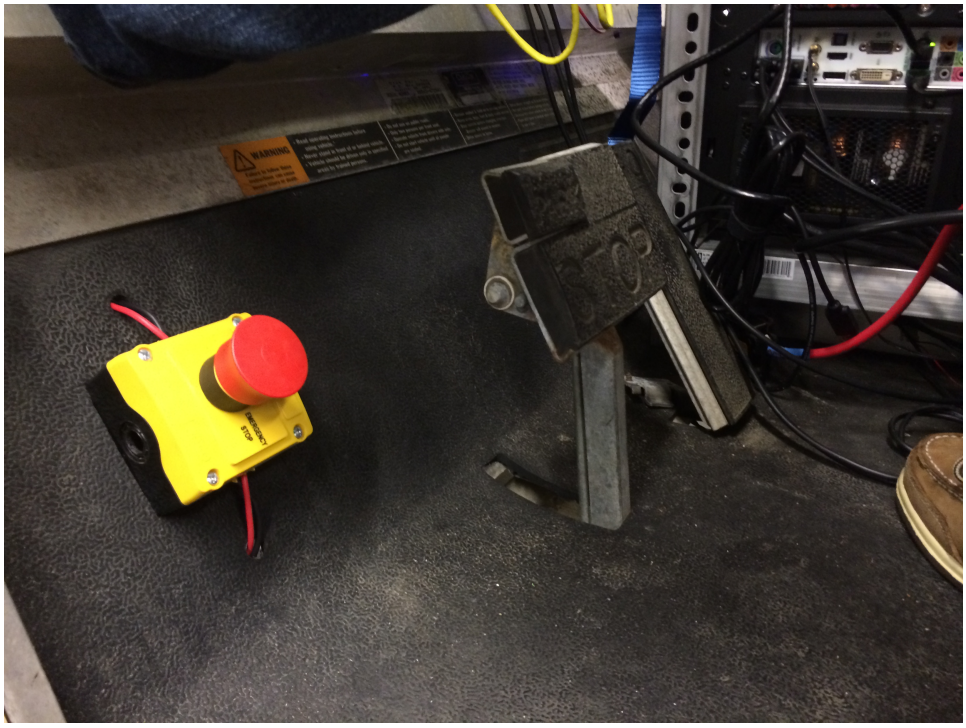


Figure 5.23: Emergency switch located in the footwell of the vehicle, for easy access to the driver. The button could be depressed by the driver's foot in emergency situations, allowing them to keep their hands safety within the vehicle.

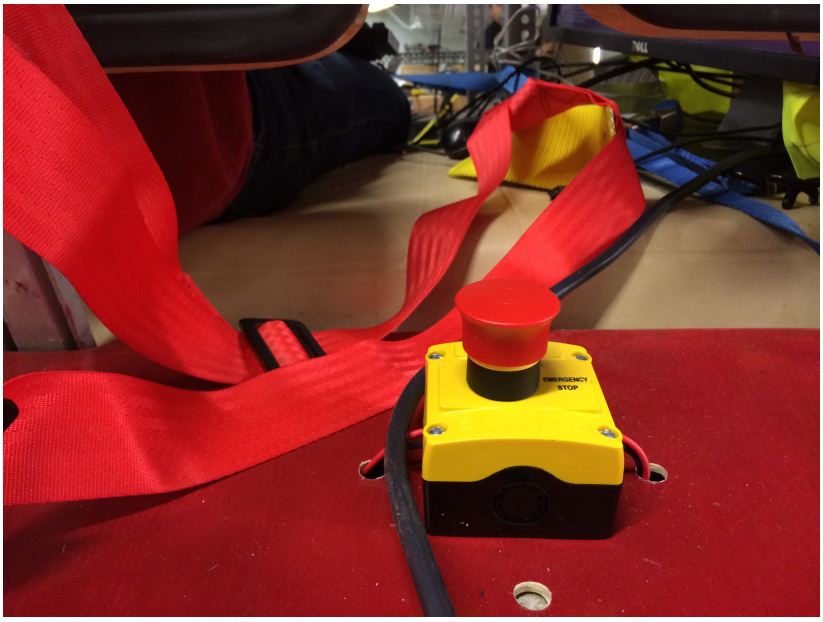


Figure 5.24: Emergency switch at the rear of the vehicle.

5.5 Chapter Summary

Many of the initial designs proposed for the ground vehicle hardware evolved over several prototyping phases. The steering system went through several iterations before a final design was installed. Impediments such as actuator size, and connecting the actuator to the rack and pinion were the most difficult to overcome. The braking system worked as expected, with the ability for manual control at all times. The team had two solutions to controlling the throttle of the cart, but ultimately chose the method that was the simplest to implement. There were some adjustments made to the original safety features of the cart, in the interest of accessibility.

6 Ground Vehicle Software

This chapter will explore the software used for vehicle control and navigation. Writing and choosing the correct software was difficult and very time consuming. Below are details describing the software used.

6.1 Vehicle Control

Control of the vehicle actuators was implemented on a Teensy 3.0 microcontroller board connected to the ROS environment. The rosserial_arduino ROS package allows a given microcontroller programmed in the Arduino environment to publish and subscribe to ROS topics, as well as provide ROS services, directly over a serial connection. The library required several small tweaks to function with the Teensy 3.0 architecture to select the correct serial interface, as seen in Figure 6.1.

```
44      -#ifdef _SAM3XA_
    44  +#if defined(__MK20DX128__) || defined(__MK20DX256__)
    45  +  #include <usb_serial.h>  // Teensy 3.0 and 3.1
    46  +  #define SERIAL_CLASS usb_serial_class
    47  +#elif defined(_SAM3XA_)
45  48     #include <UARTClass.h>  // Arduino Due
46  49     #define SERIAL_CLASS UARTClass
47  50   #else
```

Figure 6.1: Changes made to the compiler flags in rosserial_arduino/src/ros_lib/ArduinoHardware.h to accommodate the MK20DX128 (Teensy 3.0) and MK20DX256 (Teensy 3.1). These changes were submitted as pull request 90 on GitHub, and merged into the ros-drivers:hydro-devel branch of the project [50].

The library then compiled and executed without issue. The Teensy was configured to receive messages of the ackermann_msgs/AckermannDriveStamped type, a specially developed message type for vehicles with steerable wheels and a non-zero turning radius.

Commanding the vehicle is done through the AckermannDriveStamped message, where the vehicle's forward velocity, acceleration, and jerk (change in acceleration) can be commanded, as well as the vehicle's steering angle (of a virtual center wheel) and rate of change

of steering angle. For this implementation and the sake of simplicity, the acceleration, jerk, and steering angle velocity are ignored. Commanding a non-zero forward velocity releases the brake and allows the cart to move forward, and a zero forward velocity results in the brake engaging. The steering angle is sent in degrees and accurately reflects the angle of the wheels, and drives the wheels to the commanded angle as fast as possible.

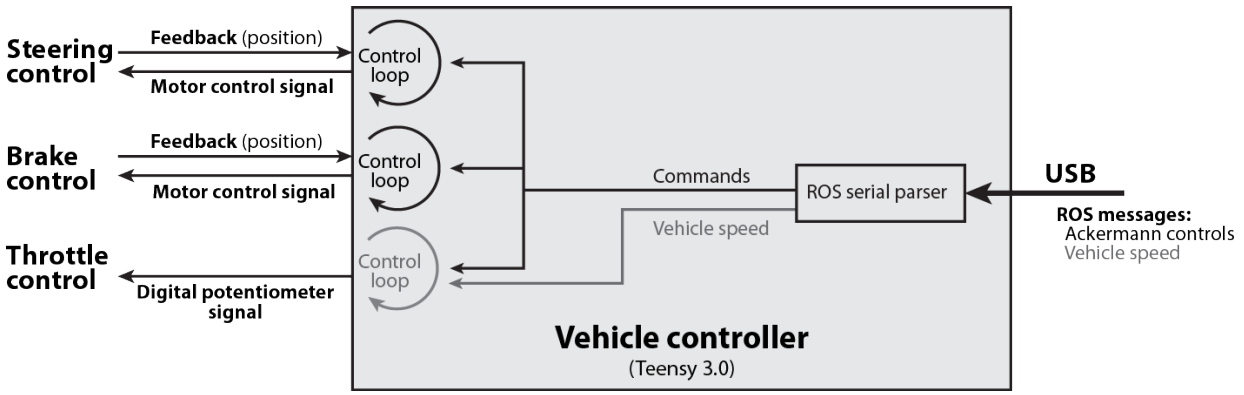


Figure 6.2: Vehicle controller software organization and inputs/outputs. The calculated vehicle speed and throttle control loop were not implemented in this iteration of the project due to time constraints.

To control the throttle, the digital potentiometer connected to the throttle was connected to the hardware SPI interface on the Teensy. Control of the digital potentiometer is achieved by writing the desired potentiometer value within a precalibrated range to the potentiometer based on the commanded speed in the ROS message. The specification for the Ackermann messages states that the commanded velocity should be in meters per second, but time constraints restricted the implementation of the throttle control to 0 to 100% of available throttle range scaled off of the commanded speed.

The Teensy was also connected to the Sabertooth motor controller on board, and supplied it with two PWM signals to control the speed and direction of the linear actuators for the steering and brakes. The potentiometers built in to the linear actuators were connected to analog inputs on the Teensy, and this positional feedback allows the actuators to be driven to a precalibrated

point or within a given range without the need for mounting additional sensors elsewhere on the vehicle.

6.2 Inertial Navigation

For the purposes of tracking motion independently of the vehicle's wheels, a Pololu MinIMU-9 v2 IMU module was incorporated into the system and interfaced with a Teensy 3.0 and the rosserial_arduino library to connect it to ROS. To acquire data from the sensor, the team started with the MinIMU9-AHRS example code provided by Pololu and accompanying L3G and LSM303 libraries to communicate with the gyroscope and accelerometer/magnetometer, respectively.

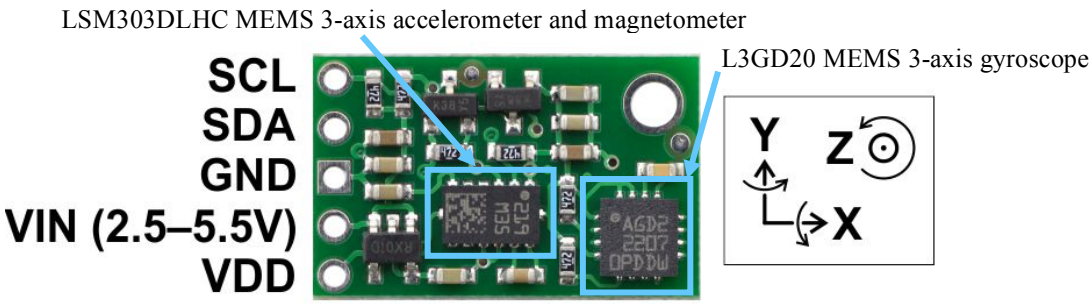


Figure 6.3: Pololu MinIMU-9 v2 IMU, with pinout and axes. The LSM303DLHC and L3GD20 are interfaced to the Teensy via I^2C.

The Teensy reads the gyro and accelerometer at a fixed 50Hz rate and the magnetometer at 10Hz. The gyro and magnetometer data processed to accumulate an angular orientation in the fashion of an Attitude and Heading Reference System (AHRS), involving integrating the gyroscope readings and compensating for drift using the magnetometer. The accelerometer data is read and stored, but not processed on the Teensy. The accelerometer, angular velocity, magnetometer, and AHRS data are published as separate topics within the same namespace over the rosserial connection due to the constraints on the size of the transmitting buffer, and assembled by a small ROS node into ROS IMU and magnetometer messages. The IMU message is then sent to MATLAB for processing.

```
samples = some integer
max = samples
If(~initialized)
     xOffSet = xOffset +x;
     yOffSet = xOffset +y;
     zOffSet = xOffset +z;
     samples-=1;
     if(samples<=0)
          initialized = 1;
          xOffSet = xOffset /max;
          yOffSet = xOffset /max;
          zOffSet = xOffset /max;
          zOffSet = xOffset -9.81
```

Figure 6.4: Pseudo code for MATLAB IMU initialization. This is executed for every message received until the end condition is met

Upon starting the MATLAB script, the IMU is assumed to be stationary such that the calibration routine reduces the effects of accelerometer bias errors. **Error! Reference source not ound.** shows how the accelerometer errors are mitigated by taking a running average of each accelerometer for a set number of samples. This gives an offset which is taken out of each accelerometer reading every time step. The orientation of the IMU is also taken into account such that the acceleration due to gravity is not corrupting the data. Equation 6.1,Equation 6.2, and Equation 6.3 describe how gravity compensation is computed in code. Equation 6.1describes the rotational matrix that describes the orientation in the IMU frame with respect to the earth frame. Equation 6.2 multiplies the result from Equation 6.1 to compute the acceleration of the IMU with respect to the earth frame. Equation 6.3 subtracts the acceleration due to gravity from

the result of Equation 6.2 to compute the sensed acceleration without the acceleration due to gravity. After calibration, the accelerometer data

Equation 6.1

$$z_{rotation} * y_{rotation} * x_{rotation} = R_{earth}^{IMU}$$

Equation 6.2

$$R_{earth}^{IMU} * Accel_{IMU} = Accel_{earth}$$

Equation 6.3

$$Accel_{without\ gravity} = Accel_{earth} - Gravity_{earth}$$

from the IMU is integrated with respect to time. Equation 6.4 shows how the position was integrated in code with the offset to account for accelerometer bias. Once the data was processed, the new data (accumulated position) is packaged in a ROS message and sent to the Kalman filter in ROS.

Equation 6.4

$$position_{(x,y,z)} = (acceleration_{without\ gravity(x,y,z)} - Offset_{(x,y,z)}) * dt + position_{(x,y,z)}$$

A simple program was also written in MATLAB that plots gyroscope data in real time. This was done to check the validity of the data, and to represent the data in an understandable format such as degrees. By rotating the IMU about one axis, the gyro output can be monitored on a display to ensure that it is sensing the rotations as expected. By combining an IMU with stereo odometry, more accurate position and orientation can be achieved for the system.

6.3 Stereo Vision and SLAM

The stereovision was implemented using ROS libraries, beginning with the image sensors. The first challenge was interfacing the Raspberry Pi cameras with ROS to efficiently

stream the camera data into ROS for processing. The Raspberry Pi Foundation is under an NDA with the SoC manufacturer (Broadcom), which prevents the public from accessing the source code of the binary "blob" containing the system drivers. The RPF has, however, published example source code to use the MMAL API calls used to control the camera and encoder, as well as for storing/streaming the acquired image data. An adaptation of the API examples for ROS was located on GitHub and installed on each Raspberry Pi system, wherein the images, after being encoded to JPEG, are broadcast as ROS image and camera_info messages for use by other ROS nodes.

Figure 6.5 shows how two Raspberry Pi cameras were mounted as stereo cameras. The cameras need to be calibrated to account for overlap, individual image distortions, and other possible errors. The calibration program was installed as a ROS package from Github. The

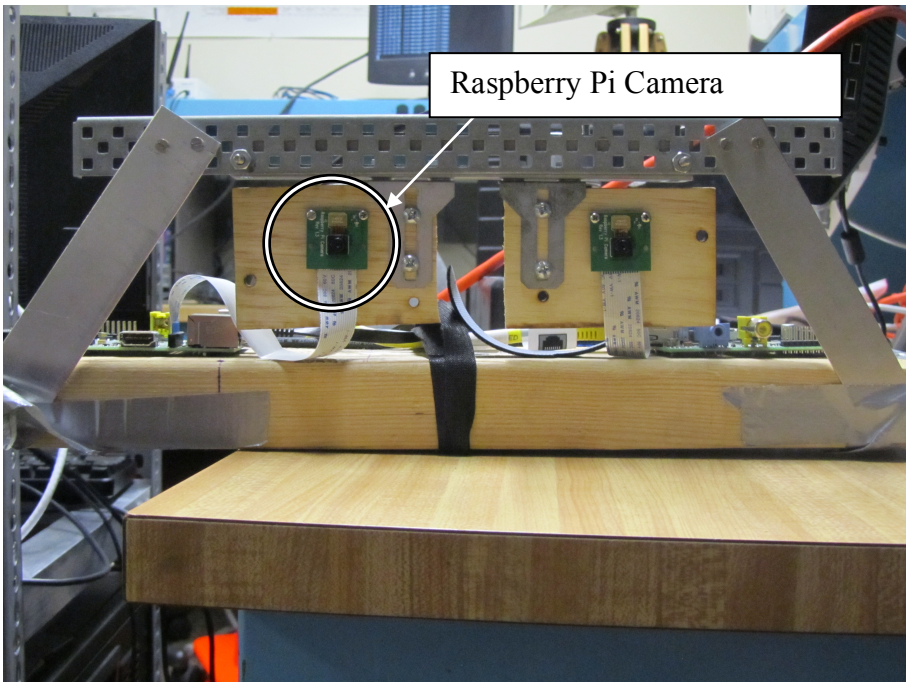


Figure 6.5: Shows two Raspberry Pi cameras side by side to function as stereo cameras.

program was easy to set up because of a small tutorial on the ROS wiki page, but getting the program to work proved more difficult.

The chief issue was that the program would appear to be idle and not performing any calibration, despite the functioning image message topics and no observable errors in the program's output. The issue was that the software comparing the images synchronizes the two image streams based on their respective timestamps, and the times on the Raspberry Pis were not the same. The Raspberry Pis lack the battery-backed RTC module found in a personal computer for the sake of price, and instead synchronize their system times over the Internet using the Network Time Protocol (NTP). The issue was resolved by connecting the Ethernet LAN of the system, including the Raspberry Pis, to WPI's network and configuring them to synchronize to the local NTP server hosted by WPI.

With the images acceptably synchronized in software, a calibration marker was printed from the internet. It consists of a black and white checkerboard pattern that the calibration

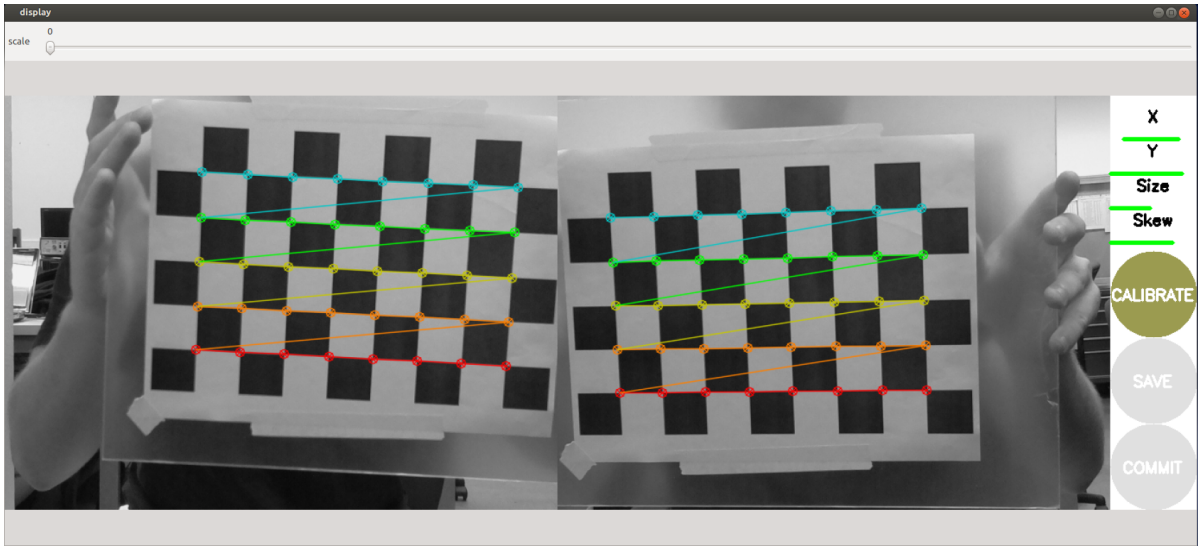


Figure 6.6: Stereo calibration software and grid. The corners of the grid are marked identified by the colored circles, and the color gradient is indicative of the orientation of the grid.

program uses to calibrate the cameras. Figure 6.6 shows the calibration program identifying the calibration board as well as the offsets between the two images. The software is provided with the size of the grid and size of a square in the grid, and evaluates the distortion of each grid to determine the camera's optical characteristics, as well as comparing the two images of the same board to establish their orientation relative to one another for stereovision. Once the cameras are calibrated, the images need to be rectified which consists of applying corrections to each camera frame. The corrections are computed one time during camera calibration. This process is done in a precompiled ROS package that computes rectified images, disparity images, and point clouds. This is the data that will be used as the input to SLAM. Given the system at hand, finding a SLAM algorithm was difficult because the software was often undocumented. As mentioned earlier, writing custom software was not an option due to the timetable of the project. Vslam was the first attempt at using a prebuilt SLAM program. It uses ROS, but Vslam was meant to use ROS Fuerte which was two versions behind the version of ROS the project team was using. The decided to devote some time to see if this program can be compiled from the source code, and those attempts were not successful. The team never got Vslam to compile under ROS Hydro because the code would need to be heavily modified to even compile under Hydro. Then the software would need to be tested to make sure that it works. The team decided to pursue another

SLAM program because Vslam appeared to be too risky and require too much time. The next approach was to try to use GMapping, another ROS package. This SLAM package needs data in 2D instead of 3D point clouds. ROS Hydro has a deprecated package, called pointcloud_to_laserscan, that converts a 3D point cloud and compresses it into 2D. Since pointcloud_to_laserscan was not updated for Hydro, the package needed to be downloaded and compiled from source. It did not compile at first, and some changes to the source code needed to

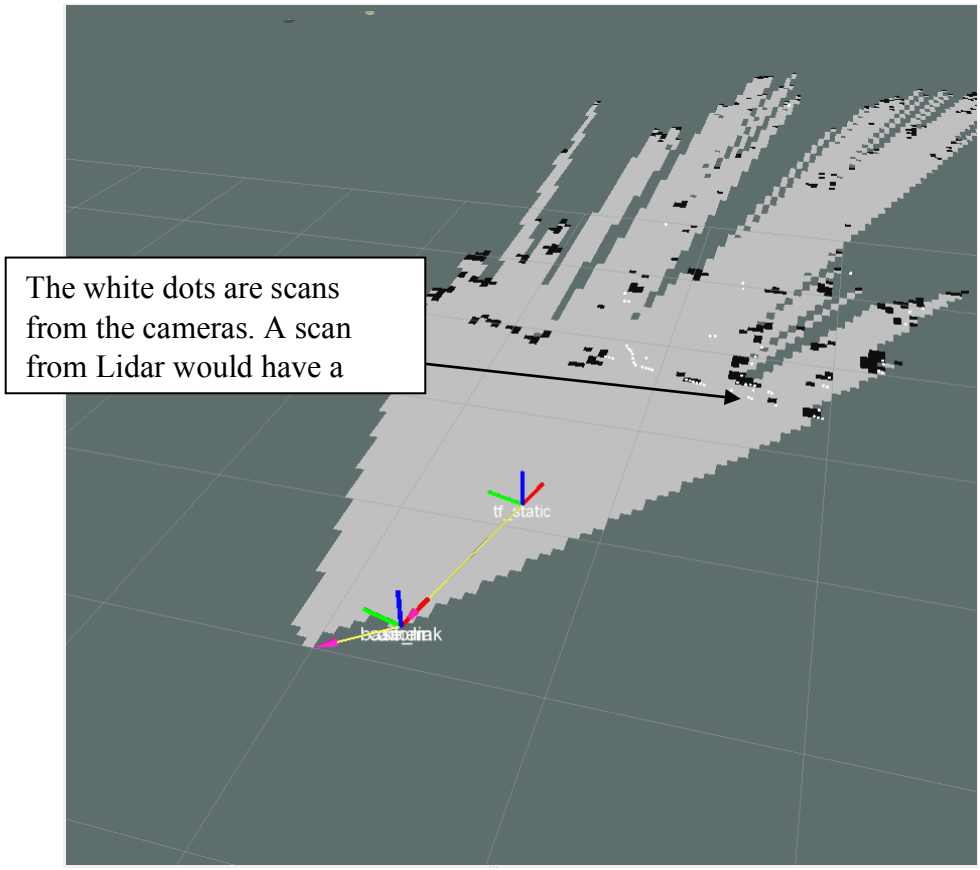


Figure 6.7: A map created by GMapping.

be made in order for it to compile on Hydro. The code worked for this plan, but it was not effective because the compressed point cloud had a very small field of view, about 20 degrees, compared to laser scans from Lidar which are often 180 degrees, and capable of being up to 360 degrees around. Figure 6.7 depicts the small field of view from the cameras. This caused issues in the GMapping software because it is intended to be used with Lidar. One solution was to make

the robot pan left and right before moving, but the issue is that the platform has a non-zero turn radius. This could have been resolved by mounting the cameras to a platform that could be rotated back and forth, which would require more hardware, as well as an appropriate controller and feedback into the system, and had a greater potential to fail or become stuck if not carefully implemented.

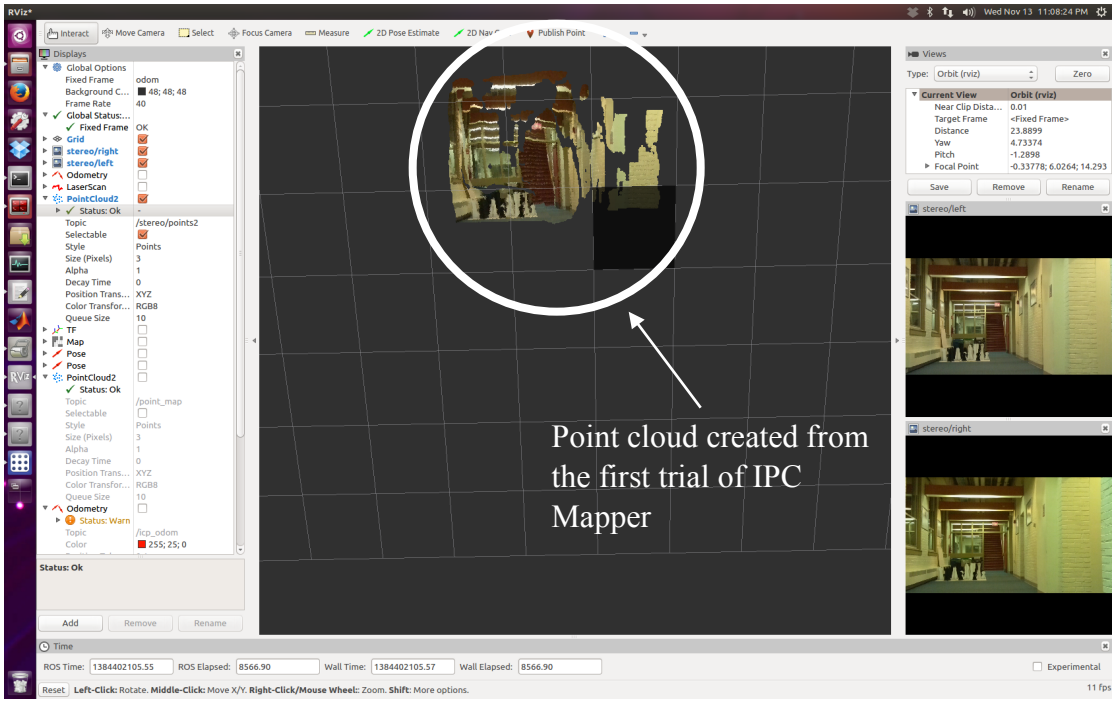


Figure 6.8: Image from ICP mapper before the test platform is moved

The team decided to try another approach that was written for a system that uses stereo vision, using the ROS package ethzasl_icp_mapping, and again the issue was that it was written for a version of ROS that is out of date.

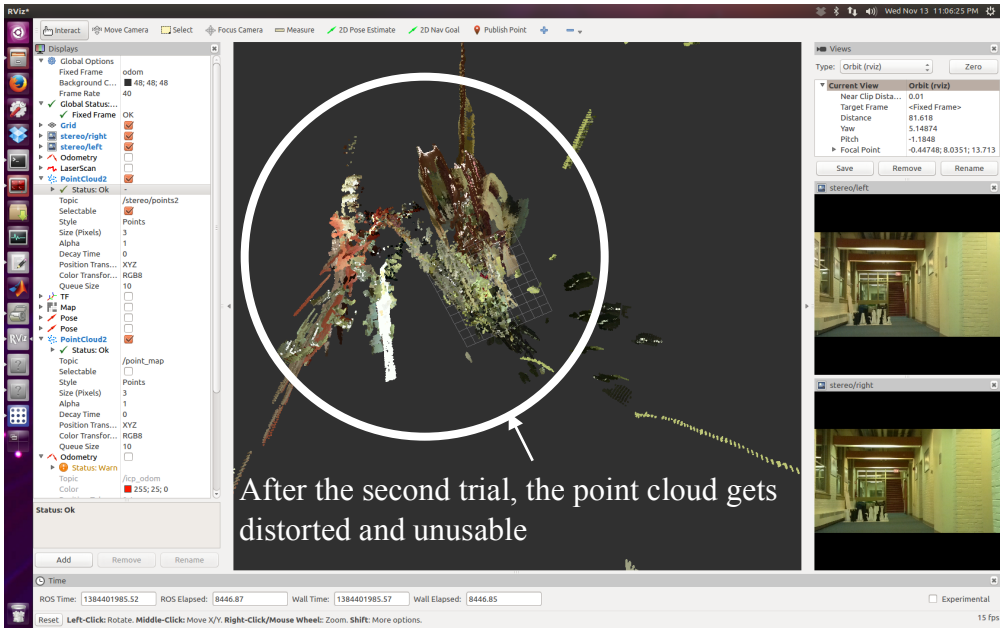


Figure 6.9: ICP mapper after the test platform has moved. The quality of this map and unreliability led the team to find a different SLAM package.

Despite that issue the team managed to compile the package. One issue discovered during testing was that the program would not map. It was correctly receiving all the needed data, but the program would still not map as needed. This problem was never solved because the program had poor error handling, and the code lacked documentation. The issue was just avoided because sometimes the program would seem to work. The team tried to determine the trigger for this behavior, but nothing was found. The code would be able to be tested periodically when the program was working, but aside from the first test the data was not good. The team determined that there needed to be less noise in the disparity images and that this program proved to be too unreliable to use because the results of the first test could not be replicated. Focus changed for a short time to try to improve the quality of the disparity images, and to create a depth image from

the disparity images so other SLAM programs can be tested. The team tried to improve the quality of the disparity image by changing parameters using the dynamic reconfigure GUI in

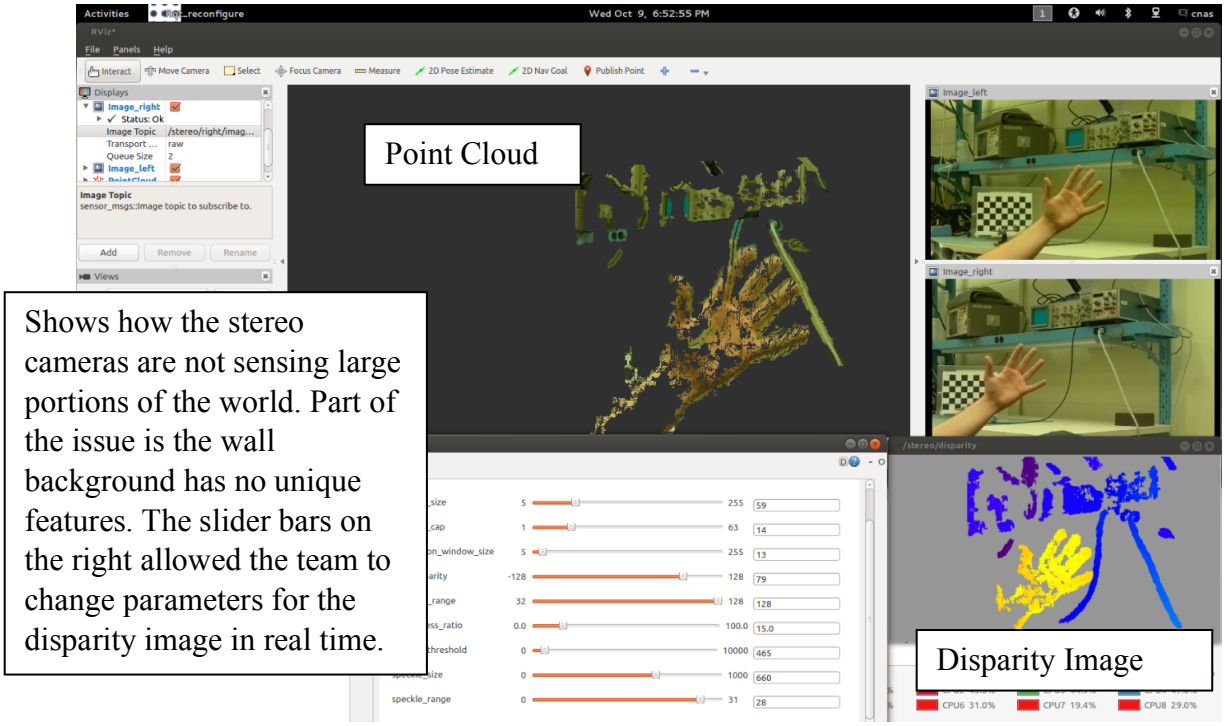


Figure 6.10: Shows how the stereo cameras are not sensing large portions of the world. Part of the issue is the wall background has no unique features.

ROS. This proved to be helpful for the most part, but it had some flaws. The GUI would sometimes crash, and it was often difficult to type values in by hand. Besides some of the mentioned flaws, this tool helped by allowing the team to change and test parameters in real time. The team did try to look for SLAM solutions at www.openslam.org and on the MATLAB file exchange. Some MATLAB programs were tested, but there was lack of documentation, and at the time the team did not have a good method of communication between ROS and

MATLAB. This was a problem that persisted for much of the project until the team stopped using IPC_Bridge and switched to a different communication method that uses ROS Java. The final SLAM program tested was rtabmap. It is written in C++ and implemented to be used in ROS. This code is able to perform loop closure in real time as well as perform visual odometry.

When using our own stereo system, the program did not seem to fully work because the program was throwing errors stating that loop closure could not be performed. By observing odometry data and the map, the team noticed that there was significant drift in the odometry data. The drift could be attributed to errors in IMU and noise from visual odometry. To prove that this code works, the Microsoft Kinect was used in place of the Raspberry Pi cameras because the code was originally written for the Kinect. Testing showed that with the Kinect the code was able to perform loop closure. Rtabmap was able to map and localize in an unknown environment using Kinect. Further research needs to be done with the custom stereo cameras to reduce noise in the images. If the custom stereo cameras can come close to the quality of the Kinect, and if the odometry quality improves then rtabmap will be viable solution on this system. Figure 6.12 show the flow of data from the stream of images all the way to the localized position. This position along the computed global map will be used to plan a path to the end goal for the system.

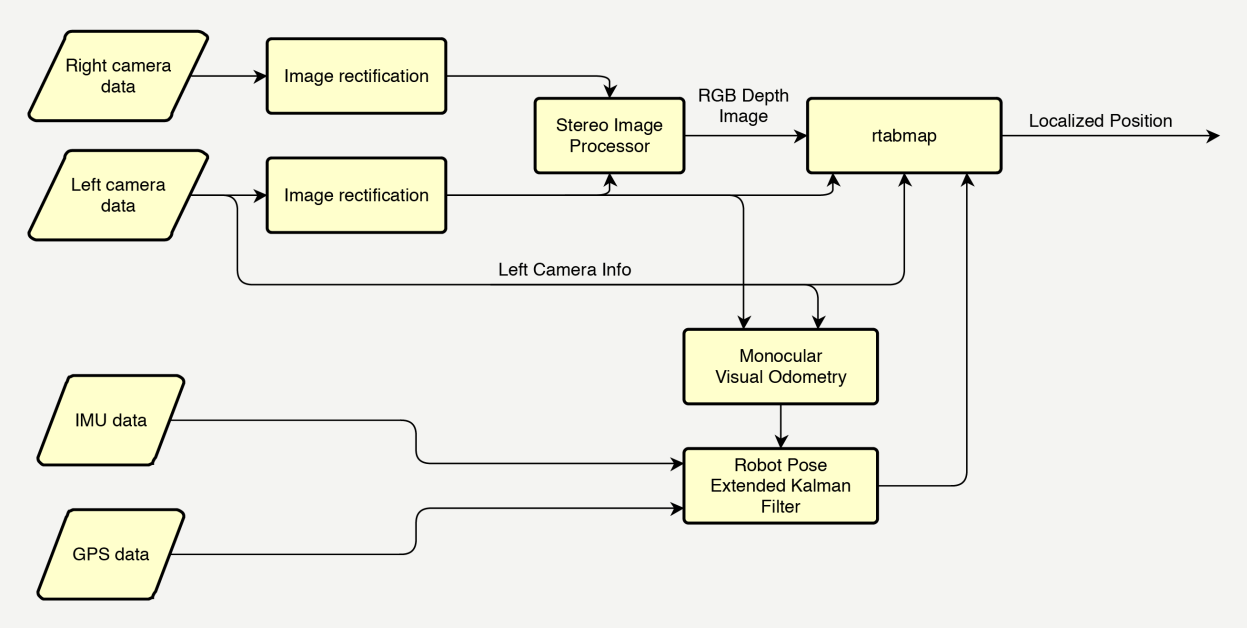


Figure 6.12: Program flow from when both Raspberry Pi cameras send information until a localized position is attained. This process runs every time step. The EKF requires one input at the given time step, but all three may also be active.

6.4 Path Planning

A* was intended to be used as to plan paths for the autonomous system. The known issue was that the vehicle uses Ackerman steering, and the path planner needed to account for this.

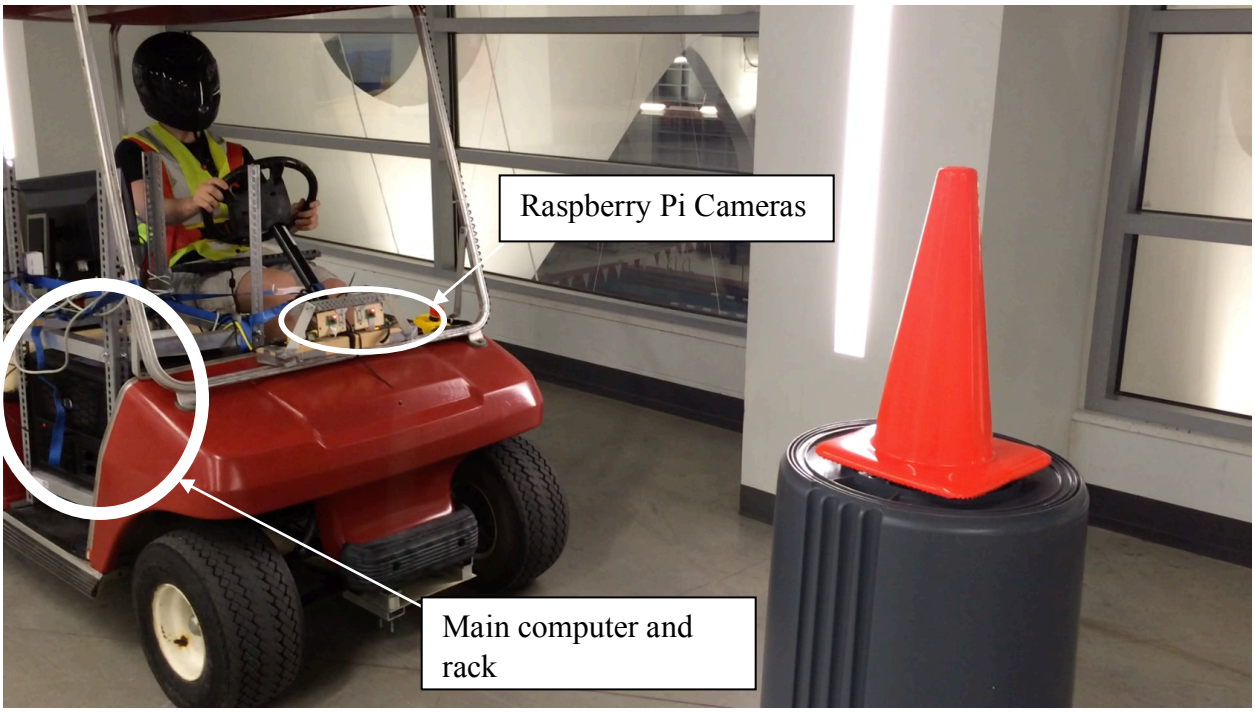


Figure 6.13: This is the golf stopped at the orange cone. The cameras are detecting it and MATLAB is sending commands to a teensy to apply the brakes.

This part of the project was given far less time than intended because of all the time that was spent working on SLAM. The ROS package sbpl_lattice_planner was downloaded and compiled from source because it was written for a previous version of ROS. The package compiled, but some of the tutorials did not work, and the documentation on this program was not very helpful. When the project finished, path planning was incomplete and still in the early stages. There does not seem to be a well-documented solution that has already been written, so new software will need to be written. The largest road blocks were time, and taking into account the motion constraints of the cart. In the end, some MATLAB code was written to control the cart from software. The program moved the cart forward until the largest orange blob occupied more than

a certain percent of the pixels in the image. This was tested by first lifting the rear wheels of the cart onto stands. One person moved the cone towards the cart until the brakes where applied. This approach was taken to minimize risk to people and damage to the cart. Once the program was tuned and all emergency stops were tested, the cart was taken off of the stands for more testing. After two more test runs without the cart on stands, it would consistently stop two to three feet before the cone as shown in Figure 6.13.

6.5 ROS MATLAB Communication

With using ROS to define how data is sent and received, a program was needed that can bridge MATLAB and ROS. The first attempt to connect MATLAB and ROS was through the use of a software package called IPC_Bridge. Figure 6.14 shows that IPC Bridge acts as a buffer between ROS and MATLAB to ensure that each receives the expected information. This package had been used by a team member in the past to transfer data between ROS and MATLAB and

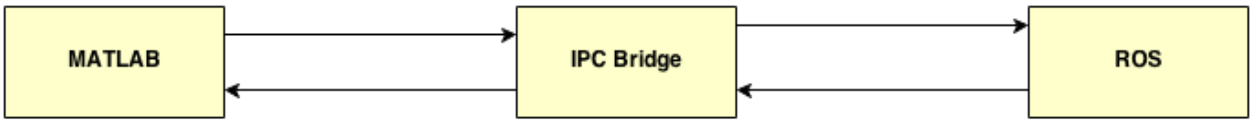


Figure 6.14: Flow diagram of how IPC Bridge transfers data between ROS and MATLAB. IPC Bridge is only able to handle one data exchange at a time which causes issues if MATLAB needs more than one data set.

thus was selected for use for the project. This program was downloaded and compiled from source. This process was more difficult than expected as there is minimal documentation on how to install this software. Additionally, when the team attempted to use IPC Bridge, a few complications arose. IPC_Bridge does not support all ROS messages requiring team members to write custom IPC message types or attempt to use similar message types. This made the code more complicated because data sent in similar messages had to then be put into the correct message type before the data could be processed. IPC_Bridge also used different Boost libraries than MATLAB which are used for smart memory management. The ROS message for images

uses Boost pointers, and when ROS sent or received an image from MATLAB, IPC_Bridge would crash. One possible solution would be to recompile ROS from source against the version of the Boost libraries shipped with MATLAB R2013a. This would additionally require all of the ROS packages installed via package management to be compiled from source as well. The team determined this was not a viable option because of the sheer number of ROS packages that were depended upon for this project, and thus downloading and building them from source would require a significant time investment. It was determined that this was not an efficient use of time for the project and thus was not a feasible option.

The team then tried to use the ROS-MATLAB bridge which uses a Java implementation of ROS to allow MATLAB code to be run as a ROS node. There were easy to follow directions online as well as a tutorial, which made this software very easy to learn. ROS-MATLAB bridge was implemented by following the installation directions at the repository. Once familiar with the ROS-MATLAB bridge, a small library was made using object oriented MATLAB. The goal of this library was to make the ROS-MATLAB bridge easier to use, more intuitive, as some of the function calls were long and confusing, and expandable. This side task was determined to be an effective use of time as it was simple and did not take much time to complete. At the moment, the library consists of a limited number of classes, but it was designed to be expandable, and it can be continued by another team at a later date.

6.6 Chapter Summary

A project of this magnitude requires copious amounts of software that must be working in harmony for smooth operation. The challenge may not always be writing code to solve a problem, but to make software that has been written by others working together. This project shows that getting software to work together is tough, but not impossible task. When using

prewritten software, it is very important to fully understand the system for which it was written. This became an issue with many visual SLAM programs as they were written for the Kinect, but showed lackluster performance when using the system implemented in this project.

While there were early struggles to connect ROS and MATLAB, a good solution was found and implemented. ROS-MATLAB bridge provides the infrastructure to write ROS nodes in MATLAB. This allowed the team to utilize many of MATLAB's toolboxes. Further research should be done to try to enable ROS-MATLAB bridge to support Ackermann messages as it will eliminate an intermediate step that adds complexity to the system.

The vehicle still needs a path planner that will consider the motion constraints of the system. A* could be used in the beginning, but depending on the size and dynamic nature of the environment a more efficient algorithm may need to be used.

The vehicle controller is functional but lacking features. The slow speed of the actuators on the cart meant a bang-bang style control loop for the brake and steering actuators could be implemented, as opposed to a more sophisticated algorithm that ramped up and down the speeds of each. Similarly, the throttle control simply scales the velocity to values output to the digital potentiometer, but is uncalibrated for the relation between commanded speed in meters per second and throttle values. This was due to an issue with the digital potentiometer interfaced to the throttle, in that the cart drove at the same speed regardless of the value. It also does not control the vehicle's acceleration, and it is left to the momentum of the cart to dictate how rapidly it accelerates and decelerates. Incorporating the current vehicle speed into a control loop will enable proper acceleration and velocity control.

7 Aerial Implementation

Two tasks needed to be addressed such that the aerial vehicles would be capable of serving their defined purpose in this project: (1) the navigation to waypoints along the planned path of the ground vehicle and (2) the identification of orange traffic cones. The ability to successfully perform these two tasks required the completion of a series of subtasks. These tasks and their required subtasks are presented in Section 6.2 and Section 6.3 below. However, the first challenge to be address before the two tasks could be attempted was the establishment of communications with the AR Drone via ROS. The method used for this project to interface the AR Drone with ROS is reviewed in Section 6.1.

7.1 Drone Communication with ROS

Before attempting to develop solutions for the required tasks of the aerial drone, the team was faced with the challenge of communicating with the drones and transmitting data to and from them. This communication needed to be done using Ubuntu 13.10 and ROS Hydro. These software versions were required for other portions of this project for compatibility with software being used to complete the various required tasks. All software must be of the same version in order to successfully integrate the separate components of this project.

One open source ROS package was identified which possessed the capability to communicate with the AR Drone under these constraints. This package, ardrone_autonomy, utilizes the AR Drone API and SDK to connect with and manipulate the various elements of the drone. Use of this package grants the capability to publish takeoff and land messages as well as desired command velocities and the capability to subscribe to data such as the image feeds from both cameras, raw data from the IMU, and the estimated velocities and accelerations calculated from this raw data among several other sources of data the drone provide. Navigation data which can be retrieved from the drone by utilizing an ardrone_autonomy subscriber can be seen in

Table 7.1: Data available through the ardrone_autonomy package from the AR Drones as seen in the documentation.

Legacy Navigation Data

Message	Description	Units
Header	ROS message header	NA
batteryPercent	The remaining charge of the drone's battery	Percentage
state	The drone's current state	0-Unknown, 1-Inited, 2-Landed, 3&7-Flying, 4-Hovering, 5-Test, 6-Taking off, 8-Landing, 9-Looping
rotX	Left/right tilt, rotation about the X axis of the drone	Degrees
rotY	Forward/backward tilt, rotation about the Y axis of the drone	Degrees
rotZ	Orientation, rotation about the Z axis of the drone	Degrees
magX, magY, magZ	Magnetometer readings	TBA
pressure	Pressure sensed by the drone's barometer	TBA
temp	Temperature sensed by the drone's sensor	TBA
wind_speed	Estimated wind speed	TBA
wind_angle	Estimated wind angle	TBA
wind_comp_angle	Estimated wind angle compensation	TBA
altd	Estimated altitude	mm
motor1..4	Motor PWM values	TBA
vx, vy, vz	Linear velocity	mm/s
ax, ay, az	Linear acceleration	g
tm	Timestamp of the data returned by the drone since drone's boot-up	µs

The package also contains launch files and executables which can be used to start the ardrone_autonomy node and initialize the sensors of the drones. In addition, this package provides capabilities to run the preprogrammed flight animations developed by Parrot.

It was decided to use this package based on team member's previous experience with it. The package provides the basic capabilities required for completion of the desired tasks of the drones in this project. Furthermore, the ardrone_autonomy package is seemingly the most popular method of interfacing with the AR Parrot drones through ROS. Many other ROS packages designed for control of the AR Drone use this package and many other universities conducting research using the AR Drone also use the ardrone_autonomy package.

The ardrone_autonomy package was cloned from its GitHub repository at the time of required use for the project and compiled from source. However, during this project the executable binaries for this package were made available through the package management system alongside the ROS core, and compiling from source is no longer needed on Linux systems. The team used the provided launch file, ardrone.launch, as a reference when creating a customized launch file to fit the needs of the drones for this project.

To leverage the tools available in MATLAB, the ardrone_autonomy package in ROS was utilized simply as a communication bridge for data transfer to and from the drone. In order to transfer the data retrieved by the ardrone_autonomy package into MATLAB, the team downloaded and attempted to use a software package called IPC Bridge. As previously stated, not all ROS message types were supported and thus the team was required to create files for IPC in order to send empty messages for takeoff and land commands for the drones. In addition, due to differences in the boost libraries, image messages could not be successfully sent between ROS and MATLAB. This was a major issue with respect to the completion of the required tasks of the

drones. This was one of the major issues mentioned in within Chapter 5: Vehicle Software with caused the team to decide to search for another method of communication between ROS and MATLAB.

When the team implemented the ROS-MATLAB Bridge package, MATLAB classes were created for each of the required tasks of the drones. Using the ROS-MATLAB Bridge capabilities, publishers and subscribers were written to retrieve and send the necessary data to complete the desired drone tasks. Figure 7.1 shows the transfer of data from the drone through the various software being used and back.

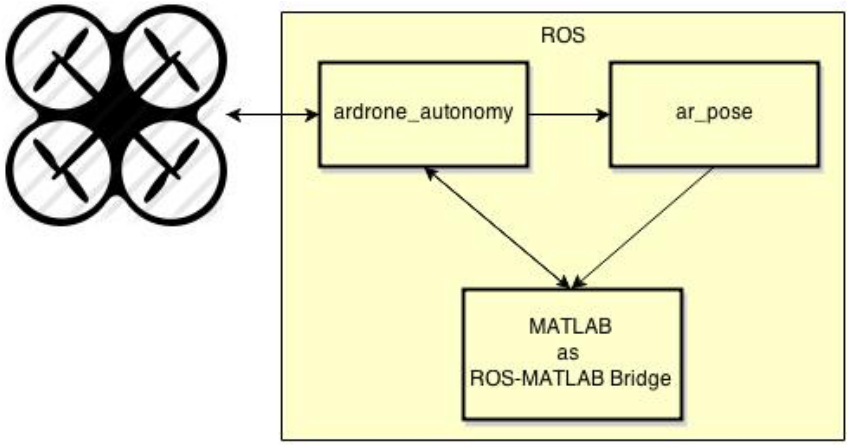


Figure 7.1: Data transfer from the drone to ROS, through ROS-MATLAB Bridge, to MATLAB and vice versa.

This method of data transfer and code structure simplified the organization of the code needed to complete these tasks as all data retrieved from the drones was available for use anywhere in the MATLAB class. In addition, publishing messages to the drone was simplified to as few as two steps: set variable and publish variable.

7.2 Aerial Navigation

Once communication to and from the drone was established and the capability to use MATLAB to perform the desired tasks was achieved, the team began working on the task of

navigation. The team began with simply setting up the basic movement of the drone. The first task was to achieve a basic hover as seen in Figure 7.2 below.

Figure 7.2: Drone at a standard hover position

This was accomplished by sending a takeoff command followed by a command velocities with all directions set to 0. From hover, command velocities were sent to have to drone fly in the desired directions or rotate to a desired heading. While the commanding of the drones to hover and fly at specified velocities is rather simple, the team found that battery life and the environment can cause very erratic behavior of the drone. Since the drones use ultrasonic sensors for achieving a desired height after takeoff, confined areas caused the ultrasonic pings to reflect and result in very sporadic drone behavior. It was also estimated that the building materials of the floor over which the drone is flying also caused odd behaviors. There were many instances where the drone was being tested in a small area in a room or on a non-solid floor of a building and launched itself into the ceiling or launched to a height much lower than the designated takeoff and hover height as can be seen in Figure 7.3.

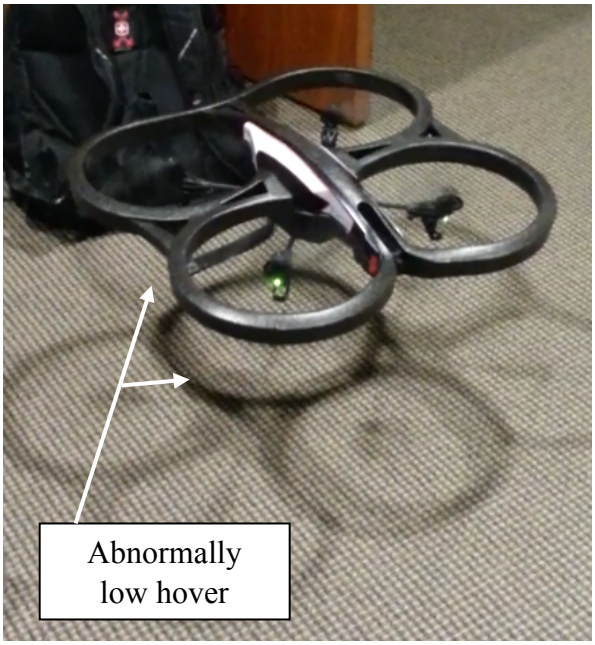


Figure 7.3: Abnormally low hover height caused by reflection of ultrasonic pings and non-solid surfaces.

When the drone was tested in an open area on solid ground, the sporadic behavior ceased. When the drone was tested with a battery lower than 75%, the drone began losing accuracy of flight. The drone would have difficulties taking off directly vertically and would drift significantly while hovering. An example of this low battery caused drift can be seen in Figure 7.4.

Figure 7.4: Low battery can cause significant drift and launch errors.

This was remedied when operated with a fully charged battery. It was observed that the drone could have around 4 minutes of flight time before losing accuracy. After achieving these basic movements and overcoming the various obstacles associated with them, the team moved to address the task of navigation.

However, physical testing of this was halted soon after these basic movements were accomplished. An issue occurred during testing with the communication to the drone. This resulted in a very consistent behavior of the drone taking off and then rolling to the right to fly indefinitely despite attempts to send other commands including command velocities as well as land commands. Figure 7.5 shows the drone in midflight while performing this behavior.

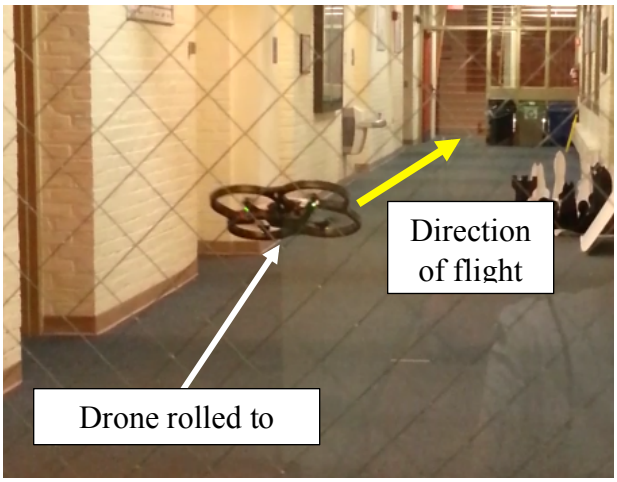


Figure 7.5: Drone rolled to right and flying uncontrollably.

This error occurred at a nondescript moment with no correlation to changes made by the team. The behavior was consistent through several troubleshooting attempts made by the team. The first attempt to troubleshoot the issue was to ensure the drone was being flown with a fully charged battery. The team charged two batteries to the full state as indicated by the Parrot AR Drone battery charger. Neither of these fully charged batteries changed the behavior of the drone. Next, a switch was made in drones being used for testing. However, testing with a different drone resulted in the same flight pattern. The team also tested drone control code written in both

MATLAB and Python in attempts to simply make the drone hover. Despite which code platform was being used to send and receive messages, the behavior of flying to the right was consistent. Two different launch files were run with each of these code platforms, but displayed no change in behavior. The team then physically held the drone on the ground and launched it while displaying the PWM values being sent to the motor of the rotors. This showed that the drone was being sent rotor speeds which caused the roll.

It is important to note, though, that when attempting to troubleshoot this problem using a python script, a land command was successfully sent to the drone when, and only when, rospy was send a shutdown command. It was written that when a rospy shutdown command was received, a land command was to be published to the drone. Upon termination of the script, the shutdown command was received and the land command was successfully sent to the drone. This shutdown command behavior in addition to the other possibilities discounted by the rest of the troubleshooting suggests that perhaps there is an issue with opening and closing the correct ports with the attempt to communicate to and end communication with those ports respectively. The team was unable to determine whether this was the cause of this behavior.

The team moved to address another element of navigation: the ability to sense the movement of drone and translate this into movement within the real world. Initially, the team attempted to do this by simply accumulating the data retrieved from the IMU. This data was returned from the ardrone_autonomy package as accelerations and as velocities in addition to rotations in each of the x, y, and z directions. The team wrote MATLAB programs to poll the velocities and timestamps of each message received from the drone. This data was used to calculate distance traveled in the respective distance of the velocities. However, due to the imperfections of the IMU, this generated a distance traveled while sitting still. It was attempted

to ignore velocities lower than a set level, but the imperfections of the IMU were inconsistent and at times read velocities of almost 0.5 meters per second at rest. It was written in the ROS launch file to recalibrate the IMU at startup. However, this did not entirely remedy the issue. The recalibration minimized the range of velocities which needed to be ignored to prevent movement from registering while the drone was actually at rest, but this range was relatively different with each calibration. For example, one calibration might determine that velocities ranging from -5 to 5 millimeters per second should be ignored where another would determine velocities from -200 to 200 millimeters per second. The team currently ignores velocities in the x direction in the range of -15 millimeters per second to 15 millimeters per second and velocities in the y direction in the range of -30 millimeters per second to 30 millimeters per second. This range is unfortunately not ideal as it results in a large amount of error when attempting to calculate distances travelled.

To assist with the error of the IMU, it was decided to use an additional ROS package called ar_pose. This package is a ROS wrapper for ARToolkit, augmented reality software from OpenCV. Information about the package is found on the ROS website. [51] The team downloaded ar_pose from in the ar_tools package on GitHub. Implementation of ar_pose required a launch file to be customized using the ar_pose_reverse.launch file. The ar_pose node launch

Using ar_pose, the drone was trained to recognize and calculate a pose from a specified vision target. This ar_pose allowed for a more accurate initial position to be calculated for the drone relative to the cart. Figure 7.6 shows the drone localizing to the black screen of a computer which the team was temporarily using as a vision target. The ar_pose package contains a training feature allowing for any vision target which meets the defined criteria to be used. These vision

targets were to be mounted on the cart such that the drones would take off and view the targets to get an initial pose relative to the cart then the drones would begin navigating to waypoints received from the vehicle. However, at the end of the time allotted for the project, the team had not reached a point where the drone localization to vision targets mounted on the cart was tested.

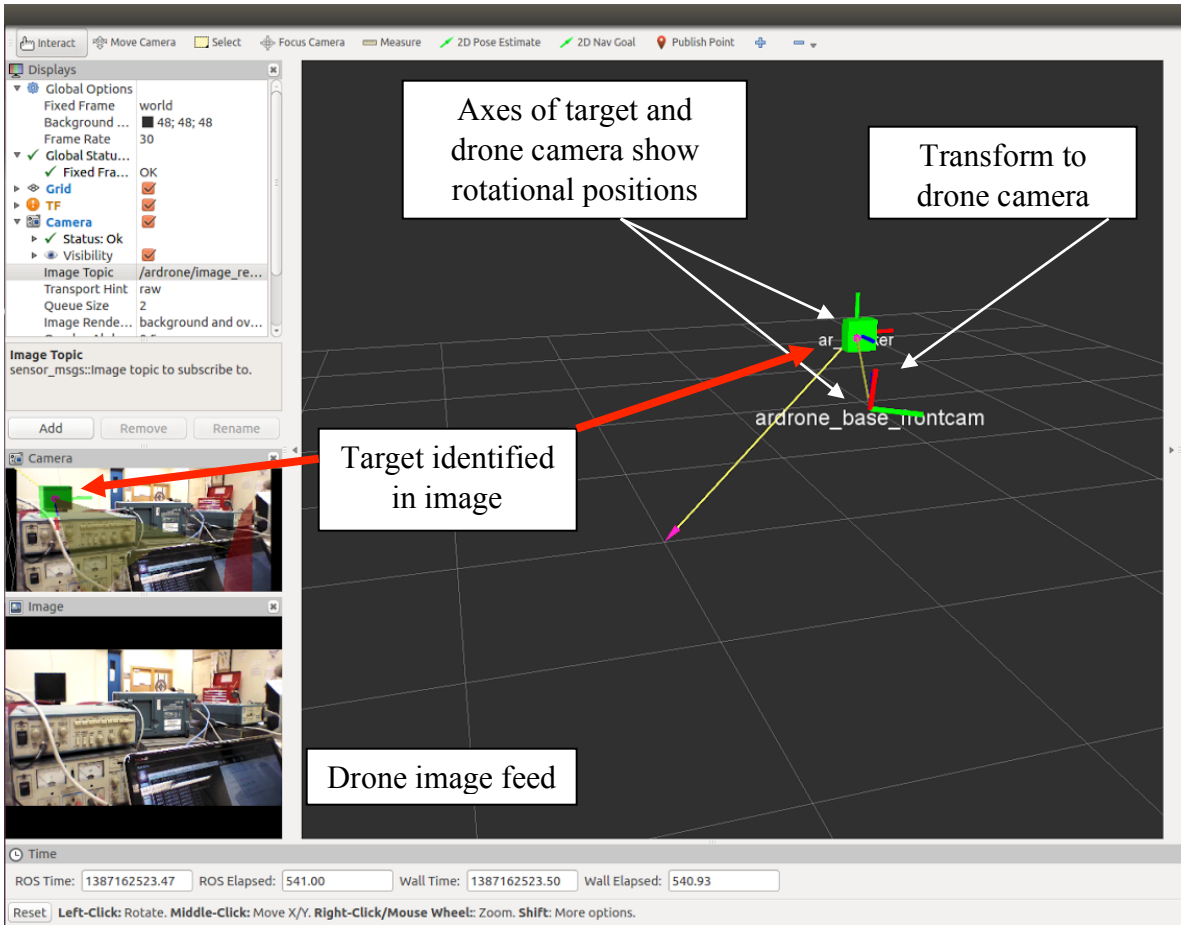


Figure 7.6: The drone is using ar_pose to localize to the black screen of the computer seen in the lower left hand corner of the image. A pose is returned giving the drone's position in relation to this screen.

With this added accuracy to the starting position, the previous method of distance accumulation using the given velocities was used. A defined range of velocities was ignored to compensate for error and noise from drift of the drone. The team decided to use the combination of these two methods due to the specific implementation of the drones for this project. The error

resulting from the distance accumulation calculations was determined to be within an acceptable range as it was less than or equal to the width of the ground vehicle. Since the drones are ultimately tasked with locating and determining the positions of traffic cones, a relative range of positional error is tolerable due to the way these cones and their locations are designed to be treated in the global map. For example, if a cone position is registered in the map with an error of 1 meter from the actual position, it will not cause a significant issue as the configuration space expanded around the cone is greater than or equal to the width of the cart and thus will be large enough to negate this error.

With this calculated initial position, a Proportion Differential (PD) control loop was written to perform accurate waypoint navigation. As can be seen in the Figure 7.7 below, a PD control loop operates on the error in the system. The PD loop uses two constants, Kp and Kd, to simulate the proportion and derivative terms of the PID control equation. These constants are multiplied by a term which represents the difference between the current and desired position and a term which represents the rate at which this difference changes. This is otherwise referred to as the error and the change in error, respectively. Specifically, this meant the PD control operated on the difference between the current position of the drone and the position of the desired waypoint.

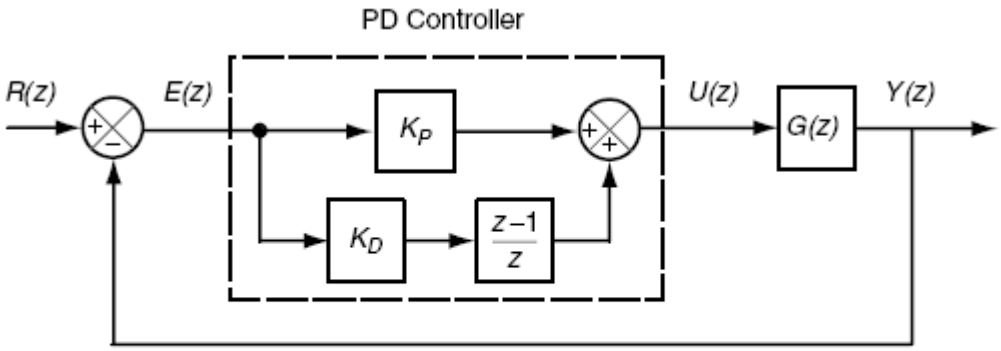


Figure 7.7: A PD control loop uses the error of a system to determine the appropriate movements to reach a desired position.

The results of the multiplication of the terms by the relative error factors are then added. This summation is used to determine the control signal sent to the drones to assist in navigating to the waypoints. Code is written to allow a range of error due to the lack of precision and inaccuracies of the drones and to allow for mild drift or mild overshoot from the target. If this allowance of error was not accounted for, the drone would never reach the position of the waypoint exactly and thus would never exit the PD control loop. An example of a PD loop with the stated error allowance is provided using pseudocode in Figure 7.8.

```
float PDControlFunction (waypoint[x, y])
{
    //Time calculations
      currentTime = time from drone message;
      dt = currentTime - previousTime;

    //Positional error calculations
      currentPosition = output from distance function;
      error = waypointPosition - currentPosition ;
      if(lowRange<error<highRange) error = 0;

    //Change in error calculations
      errorRate = error - previousError;

    //Calculate control signal to be sent to drone
      signal = kp*error + kd*errorRate/dt;

    //Update variables
      previousTime = currentTime;
      previousError = error;
}
```

Figure 7.8: Example code for the proportional derivative control loop utilized for waypoint navigation.

These desired waypoints were received from a subscriber to the cart planned path. In order to reduce opportunity for error accumulation, it was decided to strafe to the waypoints rather than to yaw about to face the desired point. This meant the drone was always facing ahead or forward. Figure 7.9 displays the diagonal movement of the drone along the arrow to achieve the desired final position. Note how the drone does not change heading and faces ahead while moving along the diagonal.

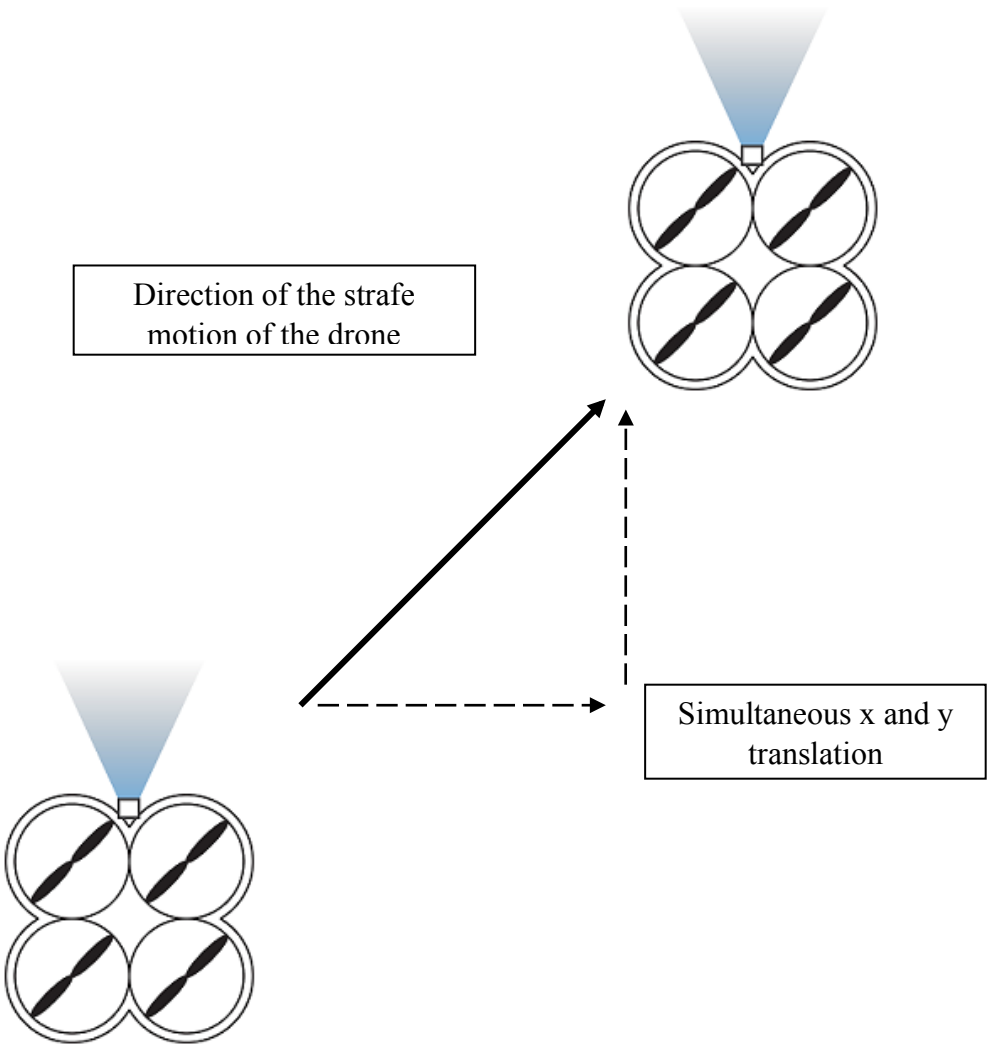


Figure 7.9: The drones are programmed to move via strafing. This means the drones will move diagonally to an end position and will always have the same heading.

The PD loop was run for each of the x and y positions simultaneously meaning the flight control decisions were being made to achieve the desired x and y positions at the same time and thus the drone would fly diagonally to the desired waypoints. The constant values, Kp and Kd for proportion and differentiation respectively, were never able to be tuned however, due to the aforementioned issue of the drone taking off and rolling to the right.

7.3 Cone Identification

The next task of the drones was the identification of the orange traffic cones. This challenged the team to be able to retrieve and display live video feed in MATLAB. This was

done by subscribing to the raw image data from the front camera of the drone. MATLAB receives the camera data as row vector of length [1x(image height in pixels * image length in pixels*3)]. The challenge was to resize the array to into a three dimensional array such as, [3x(image height in pixels x image length in pixels)]. Each dimension represents the pixel value for red, blue, and green. After reshaping the input data, the images were able to be reconstructed in MATLAB. The first attempt to stream video performed poorly because repeated calls were made to MATLAB's function for displaying images, which proved to be costly function in terms of memory and processing power. To overcome this, MATLAB Video Player was utilized which allowed MATLAB to more efficiently stream the video from ROS.

Once the live image data was being received and displayed in MATLAB, orange cones needed to be detected. This was performed using a customized color detection program, SimpleColorDetection, which the team downloaded from the MATLAB File Exchange website [reference]. With the help of the online documentation of the MATLAB toolboxes and with some example code from a MATLAB symposium, the team modified this program to detect the range of orange which matched the orange color of the cone. The code was modified to also display only blobs larger than a defined number of pixels. This was tested and successfully worked on still images. In order to detect blobs in the video feed from the drone, the team ran this blob detection code on the image data from individual frames of the video. However, a problem occurred while trying to run this program in real time. MATLAB ran out of memory and displayed a warning for a recursive function call. The issue stemmed from the fact that this was set up as a listener. As a result, more data came in than was able to be processed. To resolve this issue, a flag system was implemented. While the function was running on one frame of data, the flag was set high and thus would not attempt to process the new video frame data being

received. On completion, the flag was set low. This caused some frames to be dropped, but the effect was negligible. The processed image data was displayed using the MATLAB Video Player. This meant the video player displayed a black image with any orange blobs currently in view. Figure 7.10 shows a cone in the view of the drone. Any part of the image which does not contain the desired range of orange is not shown and thus is black.

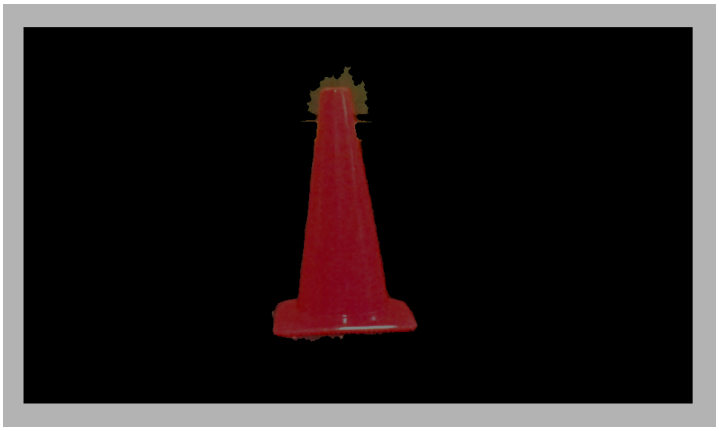


Figure 7.10: The image data processed by the custom blob detection function displays only orange blobs currently within the view of the camera.

While the team was completing the custom blob detection, a custom feature detection toolbox was discovered on the MATLAB File Exchange. This program, Cascade Training GUI, was a machine learning method of feature detection. The user inputs images of the desired feature into the GUI along with pictures which did not contain the desired feature, referred to as negatives. The program would then train using these images to search for the desired feature. The team tested the built in face detection and was able to successfully detect faces using the image feed of the drones. Based on this, the team decided this may be a more reliable and flexible option for the project and thus attempted to train it to detect traffic cones.

It was attempted to train the GUI with around 50 pictures of each category initially, but the function was unable to correctly detect any cones. Gradually, more cone pictures and more negatives were added in attempts for more accurate detection. The time required to run the

trainer increased significantly when more pictures were added or when the settings were changed in attempts to increase the accuracy. However, despite the increase in pictures and change in settings, the trainer was never able to accurately and consistently detect cones. The team input over 200 pictures containing one or more orange traffic cone and over 400 pictures contain no traffic cones or similarly shaped orange objects. The time required to run the trainer was around 30 minutes and after reading comments on the File Exchange page for the GUI and discovering other people were using over a thousand pictures for each category, it was decided that it was not an efficient use of time to continue the attempts to get the trainer functioning.

7.4 Experimental Results

At the end of the allotted time for this project, the team was unable to determine a percentage of completeness of the aerial platform due the unknown issue with the communication of data. This issue could have a simple solution which the team has been unable to identify and thus the aerial implementation could be over 80% complete or the issue could be an extremely significant impediment and result in a need to use different drones for the aerial platform leaving the project less than 25% complete. Despite this issue, the aerial platform is able to perform many of the desired tasks defined by the team in the proposed project design section 4.2.

The ability to retrieve and send data to and from the drone was achieved using ROS. This enables autonomous control of the drones. In addition, data was successfully transferred between ROS and MATLAB thus enabling the team to use the various MATLAB toolboxes to process and make decisions using this data. Orange blobs are successfully detected in live video feed from the drones meaning the drones can detect hazards. Code was written to enable waypoint navigation of the drone along the path of the vehicle. However, this code was unable to be tested.

The team also did not address the task of switching communication to a different drone once a cone location had been determined. As a result of these incomplete tasks, the team developed recommendations for any future work that is to be completed on this project. These recommendations are located in Section 7.1.

7.5 Chapter summary

The aerial platform utilized ROS and MATLAB to perform the required tasks of this project. Various software packages were implemented to assist with the communication with the drone, with the localization of the drone, and with the detection of the cones using the camera of the drone. Accuracy issues during navigation were remedied using visual target tracking to gather an initial position and using a proportional derivative control algorithm. Final results of waypoint navigation were unable to be collected due to a communication issue with the drones for which a solution was never found. Cones were successfully identified in the drone camera feed through the combination of color and blob detection. Orange blobs larger than a specified size are detected and displayed. The location of the cone is saved and able to be published to the global map. Future work for accomplishing the final tasks and improving on the accomplished tasks of the drone is defined in Chapter 8.1.

8 Conclusions and Future Work

The process of implementing an aerial platform into an autonomous vehicle project introduced many additional challenges for getting the system to collaborate successfully. The drones faced some of the same issues as were faced with the ground vehicle such as accurately recording movement of the robot. The use of multiple sensors results in more accurate estimates. Additionally, the implementation of control algorithms allows for the sensor data to be used to reach desired positions. It is important to consider the purpose of the algorithm and what is attempting to be accomplished when selecting the algorithm. The team observed that a less computationally expensive algorithm can be used if the behavior of the robot does not need to be extremely precise, such as the drones.

Similar to the ground vehicle, the team determined that more time should be given such that custom code can be written for the drones to ensure efficient completion of the desired tasks. Since the drones were capable of flying out of the box, the challenge was only the development of code to enable to drones to perform the desired tasks. Much of the project time was spent researching preexisting ROS packages and MATLAB toolboxes which could be utilized in the completion of the tasks. After the software was selected, the next challenge was debugging and integrating. The time spent on overcoming these challenges would have been better spent simply developing custom control for the drones. However, MATLAB provides well developed image processing tools and thus for simply detecting orange traffic cones, the use of the preexisting MATLAB toolboxes is likely more efficient than developing custom image processing code.

With the amount of time spent finding and adapting open source, prewritten code and the results observed from the use of this code, it was determined by the team that in order for a large scale, custom system such as the one developed in for this project, custom code needs to be generated to match to elements of the system such as what vehicles are used, what sensors these

vehicles have, how accurate these sensors are, where these vehicles are operating, and ultimately the desired tasks these vehicles are to accomplish.

8.1 Future Work

One recommendation for future teams attempting to reproduce this system or further its capabilities is to investigate other ROS packages to perform the tasks of ardrone_autonomy. The formatting of the navigational data by this package created a need for additional software to be run so that the data could be used for the designated tasks of the project. If another ROS package presented the navigational data in a standard format, the additional python script would not be needed and thus the aerial platform would require less processing power.

For additional efficiency of the performance of the drone tasks, future teams should consider drones with the capability for precision flight. This, in addition to a more accurate IMU, would eliminate the need for the image target tracking to achieve an initial position. The elimination of this would reduce the time and processing power required to operate the aerial platform. More precise navigation would reduce error accumulation during flight of the drones and more accurate IMU measurements would result in more accurate locations of the cones and thus a more accurate global map.

In addition to more precise drone flight, a more precise method of identifying cones would be desirable in the future. A recommendation would be to utilize feature detection, such as was attempted with the Cascade Training GUI, so that cones could be identified based on more than just the clustering of orange pixels. With the implementation of feature detection, objects other than orange cones could be used as example hazards. This flexibility would likely be very desirable in the event that this project were to be expanded for a purpose beyond simply research.

However, if a future team desired to simply continue with the current aerial platform, the following methods are recommended to improve the current performance. To resolve the communication issues, it is suggested to look into the TCP or UDP ports being used by the drones, by ROS, and by MATLAB. If these ports are not being properly opened and closed, it is likely there is data loss or a severing of data transfer thus resulting in the observed behavior. Once this communication issue is remedied, it is recommended to improve upon navigational accuracy. To do this, a long term resting average of the IMU data should be gathered. This average will then determine a much more accurate range of velocities which should be ignored when calculating distances. If processing power and time is not an issue, the drone IMU data can be run through a Kalman Filter like that being used by the cart for position estimation. Finally, it is suggested to attempt to use the Cascade Training GUI for cone detection. Over 800 cone pictures should be input and over 1000 negatives should be used. The cones should all be of similar aspect ratio and size in the cone images. In the event that these pictures result in a detector which still cannot accurately identify cones, it is recommend that research be done to find another method of feature detection as blob detection could be easily fooled.

With the implementation of these recommendations, the aerial platform should be able to accurately perform the desired tasks of this project. It is estimated that, pending the discovery of a solution for the communication issue, the completion of the aerial tasks could be completed in three to ten months if continued from the current state.

There is also room for continued improvement on the ground vehicle control system. One aspect of the ground vehicle control system that could be improved in later iterations of this project is the speed of the steering actuation. In the current implementation, it takes roughly seven seconds to transition from a full right turn, to a full left turn. This is too slow to operate in

a higher speed environment, as the increasing speed of the vehicle necessitates a faster steering reaction time in the presence of an obstacle.

Similarly to the steering system, the braking system could also benefit from a faster form of actuation. In a higher speed system, the time it takes to fully apply the brakes in the current design may be too slow for emergency braking. If the system was to be exposed to a more high risk, high speed environment, the need for emergency braking must be reevaluated, and modified if it is deemed unsafe. Another aspect of the braking control system that could be looked into in future projects would be adding force control to the actuation system. In the current design, the system uses the actuator's position for feedback instead of the force applied to the braking system components. By combining measured deceleration of the vehicle from the IMU or other odometry data with sensing the current drawn by the actuator or calibrating the relation of the brakes' displacement to rate of deceleration, the accuracy of the brake application could greatly improve.

The current design of the throttle controller does not have any feedback about vehicle speed or acceleration in relation to throttle position. If this were to be explored further in a future project, the team should look into creating a PID loop in the throttle control software to take into account the current speed, current throttle position, and desired speed and acceleration to make a decision about the future throttle, and possibly brake, position. For full compatibility with the ROS messages used in the control of the vehicle's motion, the acceleration and change in acceleration would also need to be factored in to this control loop.

Currently, the throttle interface still has a few unresolved bugs in its operation. For an undiscovered reason, the digital potentiometer used to control the throttle speed has a minimum resistance much higher than specified on the datasheet, preventing throttle input from going to

zero. It additionally does not have as wide a range of functional values as the analog potentiometer in the throttle. This problem may have to do with a faulty component, but the cause could not be identified during the course of this project due to time constraints.

On the software side of the project, the quality of the software needs to be improved for this project in order to allow the code base to be maintainable. Much of the current code was derived from open source libraries and projects in an attempt to move the project along. The issue was that this was also one of the major roadblocks in the project. Many of these libraries are undocumented or nearly so, and the project team was unfamiliar with their inner workings. For this project to have a long-term success, it needs to be developed in an iterative process. Code needs to be developed for the specific hardware, and the code needs to thoroughly documented while also using consistent coding standards throughout. This will have the effect of making it easier for others to become acquainted and comfortable with the code base very quickly. Based on other projects of this magnitude, there should be a minimum of three iterations to make an autonomous outdoor vehicle. Additionally, future teams will need to take on smaller parts of the overall project to allow time for writing and testing code. This is crucial because a reliable code base is needed so the system can eventually serve as a test bed for new projects and applications.

9 Bibliography

[1] Bill Gates, "A Robot in Every Home.," *Scientific American*, vol. 296, no. 1, pp. 58-65, Jan 2007.

[2] A. Curry. (2009, Aug) The Next Wave Futures. [Online]. http://thenextwavefutures.wordpress.com/2009/08/02/the-end-of-moores-law/

[3] P. LeBeau. (2013, Aug) Entrepreneur. [Online]. http://www.entrepreneur.com/article/228127

[4] D. Mead. (2013, Sept) Motherboard. [Online]. http://motherboard.vice.com/blog/nissans-autonomous-car-is-road-legal-in-japan

[5] S. Shankland. (2013, Spet) CNET. [Online]. http://news.cnet.com/8301-11386_3-57595738-76/how-googles-robo-cars-mean-the-end-of-driving-as-we-know-it/

[6] U.S. Department of Transportation Research and Innovative Technology Administration. (2014) ITS DOT [Online]. http://www.its.dot.gov/connected_vehicle/connected_vehicle.htm

[7] S. Shankland. (2013, Sept) CNET. [Online]. http://news.cnet.com/8301-11386_3-57595739-76/platooning-the-future-of-freeways-is-lining-up/

[8] California PATH. (1997) Berkley. [Online]. http://www.path.berkeley.edu/PATH/Publications/Media/FactSheet/VPlatooning.pdf

[9] DARPA. (2007) DARPA. [Online]. http://archive.darpa.mil/grandchallenge/overview.html

[10] Oshkosh Defense. (2014) Oshkosh Defense. [Online]. http://oshkoshdefense.com/technology-1/unmanned-ground-vehicle/#overview

[11] J. Markoff. (2010, Oct) New York Times. [Online]. http://www.nytimes.com/2010/10/10/science/10google.html?pagewanted=all

[12] Erico Guizzo. (2011, Oct) IEEE Spectrum. [Online]. http://spectrum.ieee.org/automaton/robotics/artificial-intelligence/how-google-self-driving-car-works

[13] Shane McGlaun. (2012, Aug) Daily Tech. [Online]. http://www.dailytech.com/Google+SelfDriving+Cars+Log+300000+AccidentFree+Miles/article25382.htm

[14] VisLab. (2014) VisLab. [Online]. http://vislab.it/pdf/VisLab_Presentation.pdf

[15] VisLab. (2011) VisLab. [Online]. http://viac.vislab.it/?page_id=11

[16] Staff Writer. (2014, March) Military Factory. [Online]. http://www.militaryfactory.com/aircraft/ww1-scout-aircraft.asp

[17] A. J. P. Taylor, *Jane's Book of Remotely Piloted Vehicles*.

[18] William Wagner, "Lightning Bugs and other Reconnaissance Drones; The can-do story of Ryan's unmanned spy planes," *Armed Forces Journal International*, 1982.

[19] Strike Weapons and Unmanned Aviation Public Affairs Office. (2013, September) Department of the Navy. [Online]. http://www.navy.mil/navydata/fact_display.asp?cid=1100&tid=2100&ct=1

[20] P Singer, *Wired for War: The robotics revolution and conflict in the 21st century*. New York: Penguin Group, 2009.

[21] Don Reisinger. (2014, February) Drones help prevent rhino, elephant poaching. [Online]. http://www.cnet.com/news/drones-help-prevent-rhino-elephant-poaching/

[22] Jillian Berman. (2013, March) 12 Companies Cashing in on Drones. [Online]. http://www.huffingtonpost.com/2013/03/11/companies-making-drones_n_2849569.html#slide=2196942

[23] Pololu. Pololu - MinIMU-9 v2 Gyro, Accelerometer, and Compass (L3GD20 and LSM303DLHC Carrier). [Online]. http://www.pololu.com/product/1268

[24] SparkFun Electronics. 9 Degrees of Freedom - Sensor Stick - SEN-10724. [Online]. https://www.sparkfun.com/products/10724

[25] Honeywell. 6 Degrees of Freedom Inertial Measurement Unit, 6 - D Motion Variant. [Online]. http://sensing.honeywell.com/index.php?ci_id=145138

[26] National Instruments. (2014) 3D Imaging with NI LabVIEW. [Online]. http://www.ni.com/white-paper/14103/en/#toc1

[27] Ken Denmead. (2013, May) MAKE Blog. [Online]. http://makezine.com/2013/05/24/new-project-diy-3d-laser-scanner-using-arduino/

[28] Lee Hutchinson. (2013, April) Ars Technica. [Online]. http://arstechnica.com/science/2013/04/how-nasa-brought-the-monstrous-f-1-moon-rocket-back-to-life/

[29] Barak Freedman, Alexander Shpunt, Meir Machline, and Yoel Arieli, "Depth mapping using projected patterns," Application 8,493,496, April 2, 2008.

[30] HOKUYO AUTOMATIC CO.,LTD. (2009, August) HOKUYO AUTOMATIC CO.,LTD. [Online]. http://www.hokuyo-aut.jp/02sensor/07scanner/download/products/urg-04lx-ug01/data/URG-04LX_UG01_spec_en.pdf

[31] IM Purchasing. IM Purchasing Partner Portal. [Online]. https://impurchasing.mysick.com/partnerportal/ProductCatalog/DataSheet.aspx?ProductID=33754

[32] IM Purchasing. IM Purchasing Partner Portal. [Online]. https://impurchasing.mysick.com/partnerportal/ProductCatalog/DataSheet.aspx?ProductID=45447

[33] Velodyne Lidar. (2010, March) Velodyne Lidar. [Online]. http://velodynelidar.com/lidar/products/brochure/HDL-64E%20S2%20datasheet_2010_lowres.pdf

[34] HOKUYO AUTOMATIC CO.,LTD. (2012) HOKUYO AUTOMATIC CO.,LTD. [Online]. http://www.hokuyo-aut.jp/02sensor/07scanner/download/products/urg-04lx-ug01/

[35] Velodyne. Velodyne Lidar. [Online]. http://velodynelidar.com/lidar/hdlpressroom/photogallery.aspx

[36] Velodyne Lidar. Index of /lidar/doc/CD HDL Product Information/Velodyne HDL-INFO CD v 1.8/Images. [Online]. http://velodyne.com/lidar/doc/CD%20HDL%20Product%20Information/Velodyne%20HDL-INFO%20CD%20v%201.8/Images/

[37] Mercedes-Benz. Mercedes-Benz Intelligent Drive. [Online]. http://www5.mercedes-benz.com/en/innovation/mercedes-benz-intelligent-drive-driver-assistance-systems-safety-comfort/

[38] Bruce Siciliano, Oussama Khatib, and Frans Groen, "Springer Tracts in Advanced Robotics," vol. 38, 2008.

[39] R. Faragher, "Understanding the Basis of the Kalman Filter Via a Simple and Intuitive Derivation," *Signal Processing Magazine*, vol. 29, no. 5, pp. 128-132, 2012.

[40] Greg Welch and Gary Bishop. (2001) An Introduction to the Kalman Filter. [Online]. http://www.cs.unc.edu/~tracker/media/pdf/SIGGRAPH2001_CoursePack_08.pdf

[41] N. Sariff and N. Buniyamin, "An Overview of Autonomous Mobile Robot Path," *Research and Development*, pp. 183-188, 2006.

[42] StackOverflow. (2014) A* Search Algorithm. [Online]. http://stackoverflow.com/questions/5849667/a-search-algorithm

[43] Andrew Davison. Lecture 2: Robot Motion. [Online]. http://www.doc.ic.ac.uk/~ajd/Robotics/RoboticsResources/lecture2.pdf

[44] Admin. (2012, Oct) Petrol Smell. [Online]. http://petrolsmell.com/2012/10/27/evolution-of-braking/

[45] n. Sclater, *Mechanisms and Mechanical Devices Sourcebook, 4th Edition*.: McGraw-Hill, 2007.

[46] wiseGeek. (2014) wiseGeek. [Online]. http://www.wisegeek.org/what-is-an-actuator.htm

[47] Karim Nice. (2014) How Car Steering Works. [Online]. http://auto.howstuffworks.com/steering2.htm

[48] Ajay Arora. (2011, Dec.) Progressive Automations. [Online]. http://www.progressiveautomations.com/progressive-linear-actuators-introduction-part-a-40.aspx

[49] Progressive Automations. (2014) Progressive Automations. [Online]. http://www.progressiveautomations.com/actuator-linear-actuator-with-potentiometer-14p-linear-actuator-

with-potentiometer-stroke-size-force-150-lbs-speed-040sec-p-75.aspx

[50] rosserial_arduino support for Teensy 3.0 and 3.1 • ros-drivers/rosserial • GitHub. [Online]. https://github.com/ros-drivers/rosserial/pull/90

[51] ROS. (2014, January) ar_pose. [Online]. http://wiki.ros.org/ar_pose

[52] VisLab. (2014) VisLab. [Online]. http://vislab.it/whos-vislab/

[53] DARPA. (2013) DARPA. [Online]. http://www.darpa.mil/About.aspx

[54] DARPA. (2012, Apr) DARPA. [Online]. https://www.fbo.gov/utils/view?id=74d674ab011d5954c7a46b9c21597f30

[55] DDRE. (2008, Jan) DARPA. [Online]. http://archive.darpa.mil/grandchallenge/docs/DDRE_Prize_Report_FY07.pdf

[56] K. Maxey. (2013, Apr) Engineering.com. [Online]. http://www.engineering.com/DesignerEdge/DesignerEdgeArticles/ArticleID/5624/DARPA-FANG-Challenge--1M-to-the-winners.aspx

[57] Tartan Racing. (2007) Tartan Racing. [Online]. http://www.tartanracing.org/gallery.html

[58] McMaster Carr. (2014) McMaster Carr. [Online]. mcmastercarr.com

Appendix A: Purchase List

Central Computing Platform	Quantity	Price	Total		
Motherboard: LGA1150 (CPU-compatible socket)	1	$199.99	$199.99		ASUS Z87-PRO Intel motherboard
CPU: Intel i7-4770K, 3.5 GHz quad-core	1	$339.99	$339.99		Intel Core i7-4770K Haswell 3.5GHz LGA 1150 84W Quad-Core (BX80646I74770K)
GPU: NVIDIA GeForce GTX 770	1	$380.00	$380.00		GIGABYTE GeForce GTX 770 2GB 256-bit GDDR5 450W (GV-N770OC-2GD)
RAM: 4x8GB, DDR3-2133 (240-pin)	1	$314.99	$314.99		G.SKILL Ripjaws Z Series 4x8GB, DDR3-2133 (F3-17000CL11Q-32GBZLD)
SSD 240GB	1	$170.00	$170.00		Intel 335 Series Jay Crest SSDSC2CT240A4K5 2.5" 240GB SATA III
PSU (1050W)	1	$200.00	$200.00		SeaSonic X-SERIES X-1050 1050W
Case	1	$52.00	$52.00		120 qt coleman xtreme cooler
Case fans	6	$15.00	$90.00		COUGAR CF-V12H Vortex Hydro-Dynamic-Bearing
CPU Cooler	1	$50.00	$50.00		XIGMATEK Dark Knight II SD1283 Night Hawk Edition
Case	1	99.99	99.99		Cooler Master HAF XB EVO - High Air Flow Test Bench and LAN Box Desktop Computer Case with ATX Motherboard Support
GPU	1	249.99	249.99		EVGA SuperClocked 02G-P4-2765-KR GeForce GTX 760 2GB 256-bit GDDR5 PCI Express 3.0 SLI Support w/ EVGA ACX Cooler Video Card
Power Supply	1	149.99	149.99		SeaSonic SS-750KM3 750W ATX12V V2.3/EPS 12V V2.91 SLI Ready 80 PLUS GOLD Certified Full Modular Active PFC Power Supply New 4th Gen CPU Certified Haswell Ready
16GB RAM	1	164.99	164.99		G.SKILL Ripjaws X Series 16GB (4 x 4GB) 240-Pin DDR3 SDRAM DDR3 1600 (PC3 12800) Desktop Memory Model F3-12800CL9Q-16GBXL
Motherboard	1	169.99	169.99		ASUS P9X79 LE
CPU Cooler	1	36.99	36.99		XIGMATEK Gaia SD1283 120mm Long Life Bearing CPU Cooler LGA1150 Haswell Compatible
CPU and SSD Combo	1	699.98	699.98		Intel Core i7-4930K Ivy Bridge-E 3.4GHz LGA 2011 130W Six-

					Core Desktop Processor BX80633i74930K
				$3,368.89	**subtotal**
Vision System					
Raspberry cameras	5	$20.00	$100.00		
Wired/wireless router, 4-port Gig-E, Wireless 802.11n	1	$70.00	$70.00		NETGEAR WNDRMAC-100NAS Wireless Gigabit Open Source Router
8-port Unmanaged Gigabit Ethernet switch	1	$39.00	$39.00		D-Link GO-SW-8GE Unmanaged 10/100/1000Mbps 8-Port Gigabit Metal Desktop Switch
16GB SD Card	2	$20.00	$40.00		PNY Pro-Elite 16GB Secure Digital High-Capacity
IMU	1	39.95	39.95		MiniIMU-9 v2 Gyro, Accelerometer, and Compass (L3GD20 and LSM303DLHC)
				$288.95	**subtotal**
Control System					
Teensy 3.0	4	$25.00	$100.00		
8" Linear actuator	1	$399.99	$399.99		
Actuator Mounting Bracket	1	$34.99	$34.99		
Motor Controller	1	$189.99	$189.99		
12V batteries	1	$90.00	$90.00		
Anderson connector with 120A breaker	1	$50.00	$50.00		
Anderson connectors	2	$20.00	$40.00		
4" Linear actuator	1	$110.00	$110.00		
Maxbotix MB1030 LV-MaxSonar-EZ3	2	$26.95	$53.90		
Maxbotix MB1010 LV-MaxSonar-EZ1	1	$24.95	$24.95		
Actuator Mounting Bracket	1	$12.95	$12.95		
Pi Case	2	$40.11	$80.22		
Tie Rod End	2	$6.89	$13.78		

Tie Rod	1	$29.43	$29.43		
				$1,230.20	**subtotal**
Safety System					
Helmets	2	$54.95	$109.90		
High Visibility Vest	7	$7.45	$52.15		
Cones	10	$12.97	$129.70		
Seatbelt	1	$29.99	$29.99		Racing Seat Belt 4 Point Harness - Red
Safety Light	1	$16.97	$16.97		Amber Emergency Hazard Warning LED Mini Bar Strobe Light w/ Magnetic Base
Fire Extinguisher	1	$49.99	$49.99		Kidde FA110 Multi Purpose Fire Extinguisher 1A10BC
Gloves	1	$15.99	$15.99		
Emergency Shutoff Switch	3	$44.60	$133.80		
				$538.49	**subtotal**
Assorted					
Jack	1	39.95	39.95		
Jack Stands	2	24.99	49.98		
WD-40	2	6.23	12.46		
Extension Cord	2	16.77	33.54		
Power Strip	1	20.32	20.32		
Gloves	2	16	32		
Scale	1	44.99	44.99		
Micro USB Cable	12	5.49	65.88		
				$299.12	**subtotal**
Golf Cart					
1995 Club Car	1	$1,500.00	$1,500.00		
				$1,500.00	**subtotal**

Appendix B: MATLAB Classes for ROS-MATLAB Bridge

```
classdef Node < handle
    %UNTITLED Summary of this class goes here
    %   Detailed explanation goes here

    properties
        publisher = java.util.HashMap;
        subscriber = java.util.HashMap;
        node  = [];
        Master_URI = 'http://localhost:11311';

    end

    methods
        % Constructor
        % param name -String
        function obj =  Node(name)
            jmb_init();
            obj.node = jmb_init_node(name,obj.Master_URI);

        end

        % Add a publisher to the desired node
        % param topic -The topic to publish to, String
        % param type -The type of ROS messege to be published, String
        function [] = addPublisher(o,topic,type)
            o.publisher.put(topic,o.node.newPublisher(topic,type));
        end

        % Add a subscriber to the desired node
        % param topic -The topic to subscriber to, String
        % param type -The type of ROS messege that is subsribed to, String
        function [] = addSubscriber(o,topic,type,callback)
            sub = edu.ucsd.SubscriberAdapter(o.node,topic,type);
            set(sub,'OnNewMessageCallBack',callback);
            o.subscriber.put(topic,sub);
        end

        % Finds publisher that corresponds to the given topic and messege
        % param topic -The topic to publish to, String
        % param msg  - The messege to be published, Variable type
        function [] = Publish(o,topic,msg)
            o.FindPublisher(topic).publish(msg);
        end

        % Shuts down specified publisher
        % param topic -The topic for the publisher to shutdown, String
        % This function has not been tested
        function [] = KillPublisher(o,topic)
            o.FindPublisher(topic).shutdown();
        end
        % shuts down the node
        function [] = KillNode(o)
           o.node.shutdown();
        end
```

```
        % Finds desired publisher
        % param topic -The topic of the desired publisher, String
        function [pub] = FindPublisher(o,topic)
            pub = o.publisher.get(topic);
        end

        % Finds the desiredSubscriber
        % param topic -The topic of the desired subscriber, String
        function [sub] = FindSubscriber(o,topic)
            sub = o.subscriber.get(topic);
        end
    end

end



classdef IMU < Node
    %TODO: subclass this.

                        % get the progress bar working again
    %UNTITLED Summary of this class goes here
    %   Detailed explanation goes here

    properties
        init = 0;
        samplesMax = 100;
        samples = 100;

%         publisher = [];
%         subscriber = [];

%         node = []
%         MASTER_URI = 'http://localhost:11311';
        data = [];
        currTime = 0;
        prevTime = 0;
        dt = 0;
        %offsets
        xOff = 0;
        yOff = 0;
        zOff = 0;
        hProgBar = [];



        pub_pose = zeros(1,3);

    end

    methods
        function obj = IMU(nodeName)
            obj = obj@Node(nodeName);
%             jmb_init();
            obj.hProgBar = waitbar(0,sprintf('Calculating Accleromter Drifts:
%d%% complete',0));
            set(obj.hProgBar,'Visible','off');
```

```
%             obj.node = jmb_init_node(nodeName, obj.MASTER_URI);
%             obj.publisher =
obj.node.newPublisher('/odom_old','nav_msgs/Odometry');
%             obj.subscriber =
edu.ucsd.SubscriberAdapter(obj.node,'/cart/imu','sensor_msgs/Imu');
%             set(obj.subscriber,'OnNewMessageCallBack',@obj.imuCallback);

%% new method

obj.addSubscriber('/cart/imu','sensor_msgs/Imu',@obj.imuCallback);
            obj.addPublisher('/odom','nav_msgs/Odometry');

        end

        function [] = initialize(o,linacc, orientation)
            % drift compensation
            %
            %             for I = 1:o.samples
            %
waitbar(I/o.samples,o.hProgBar,sprintf('Calculating Accleromter Drifts: %d%%
complete',(I/o.samples)*100));
            %                 xOffAve(I) = linacc.x;
            %                 yOffAve(I) = linacc.y;
            %                 zOffAve(I) = linacc.z;
            %                 if(I==o.samples)
            %                     o.xOff = sum(xOffAve)/o.samples;
            %                     o.yOff = sum(yOffAve)/o.samples;
            %                     o.zOff = sum(zOffAve)/o.samples;
            %                     o.zOff = zOff + -9.81;
            %                     close(hProgBar);
            %                 end
            %             end
            %
            %             o.init = 1;
            if(o.samples<1)
                o.xOff = o.xOff/o.samplesMax;
                o.yOff = o.yOff/o.samplesMax;
                o.zOff =o.zOff/o.samplesMax;
                o.zOff = o.zOff + -9.81;
                close(o.hProgBar);
                o.init = 1;
            else
%
waitbar(o.samples/o.samplesMax,o.hProgBar,sprintf('Calculating Accleromter
Drifts: %d%% complete',(o.samples/o.samplesMax)*100));
                o.xOff=o.xOff+linacc.x;
                o.yOff=o.yOff+linacc.y;
                o.zOff=o.zOff+linacc.z;
                o.samples = o.samples-1;
                display(o.samples);
            end


        end
        % might be : function [] = imuCallback(o,handle,evt)
        function [] = imuCallback(o,handle,evt)
            o.run(evt);
```

```
        end

        function [newAcc] = gravityComp(o,acc,orientation)
            % convert from Quaternion to Euler
            [euler.x euler.y euler.z] = q_getEulerAngles([orientation.x
orientation.y orientation.z orientation.w],[1 2 3]);
            % X rotation matrix
            xRot = [1 0 0;
                0 cos(euler.x) -sin(euler.x);
                0 sin(euler.x) cos(euler.x)];
            % Y Rotation matrix
            yRot = [cos(euler.y) 0 sin(euler.y);
                0 1 0;
                -sin(euler.y) 0 cos(euler.y)];
            % Z Rotation Matrix
            zRot = [cos(euler.z) -sin(euler.z) 0;
                sin(euler.z) cos(euler.z) 0;
                0 0 1];
            % Full 3 dof rotation matrix
            toInertialFrame = zRot*yRot*xRot;
            % gravity with respect to earth
            ge = [0;0;9.8];% reverse sign for real imu
            % gravity with respect to IMU
            gc = toInertialFrame'*ge;

            linx = acc.x;
            liny = acc.y;
            linz = acc.z;
            gi = [linx; liny; linz];
            % add the offset to zero sensed gravity
            newAcc = gi+gc;

        end

        function [] = run(o,evt)
            msg= org.ros.message.nav_msgs.Odometry();
            orientation = evt.getSource.orientation;
            linacc = evt.getSource.linear_acceleration;


            if(~o.init)
                set(o.hProgBar,'Visible','on');
                o.initialize(linacc,orientation);
            else

                % if the imu message is connected


                %                 imu_msg=imu_cmd.read();
                %                 linacc = imu_msg.linear_acceleration;
                %                 orientation = imu_msg.orientation;
                %% Gravity Compensation

                newAcc = o.gravityComp(linacc,orientation);

                linx = newAcc(1)+(o.xOff*-1);
                liny = newAcc(2)+(o.yOff*-1);
```

```
                linz = newAcc(3)+(o.zOff*-1);


                %% Euler Angle Plot
                % I think the gi variable here needs to be changed for this
section
                % to be used
                %         [euler.x euler.y euler.z] =
q_getEulerAngles([orientation.x orientation.y orientation.z orientation.w],[1
2 3]);
                % Keep a circular indexed array of euler angles for
evaulation
                % purposes
                %         gx = circshift(gx,1);
                %         gx(1) = euler.x;
                %         gy = circshift(gy,1);
                %         gy(1) = euler.y;
                %         gz = circshift(gz,1);
                %         gz(1) = euler.z;
                %
                %     if(gi<=100)
                %         gx(gi) = orientation.x;
                %         gy(gi) = orientation.y;
                %         gz(gi) = orientation.z;
                %         gi= gi+1;
                %     else
                %
                %     end



                %     plot(gx.*180/pi,'g');
                %     hold on
                %
                %     plot(gy.*180/pi,'r');
                %     plot(gz.*180/pi,'b');
                %     ylim([-200 200]);
                %     hold off

                %     plot(orientation.y*180/pi,'r');
                %     plot(orientation.z*180/pi,'b');
                %     legend on;

                %     legend('x','y','z');
                %% Integrate Accelerations

                if(o.currTime==0 && o.prevTime==0)
                    % The first messege will have a dt of 0.
                    o.dt=0;
                    o.currTime = evt.getSource.header.stamp.toSeconds;

                else
                    % Calculate dt
                    o.prevTime = o.currTime;
                    o.currTime = evt.getSource.header.stamp.toSeconds;
                    o.dt = o.currTime-o.prevTime;
```

```
                    % Accumulate position
                    o.pub_pose(1,1) = o.pub_pose(1,1)+(linx*o.dt^2);
                    o.pub_pose(1,2) = o.pub_pose(1,2)+(liny*o.dt^2);
                    o.pub_pose(1,3) = o.pub_pose(1,3)+(linz*o.dt^2);

                    % Accumulate Velocity
                    pub_twist(1,1) = linx*o.dt;
                    pub_twist(1,2) = liny*o.dt;
                    pub_twist(1,3) = linz*o.dt;

                    % The be
                    % update measurements
                    %             prevState(3) = linx;
                    %             prevState(6) = liny;
                    %             prevState(9) = linz;
                    %
                    %             % propagate the state
                    %             currState = IMU_F(dt,prevState);
                    %
                    %             prevState = currState;
                    %
                    %             pub_pose = [currState(1) currState(4)
currState(7)];
                    %             pub_twist = [currState(2) currState(5)
currState(8)];

                    % Put positions and orientations as a quaternion into a
                    % poseStamped ROS messege
                    msg.header = evt.getSource.header;
                    msg.pose.pose.position.x = o.pub_pose(1,1);
                    msg.pose.pose.position.y = o.pub_pose(1,2);
                    msg.pose.pose.position.z = o.pub_pose(1,3);

                    msg.pose.pose.orientation = evt.getSource.orientation;
                    msg.pose.covariance = [0.0392,0,0,0,0,0,
0,0.0392,0,0,0,0, 0,0,0.0392 ,0,0,0,   0,0,0,0.00122,0,0, 0,0,0,0,0.00122,0,
0,0,0,0,0, 0.00122];

                    msg.twist.twist.linear.x = pub_twist(1,1);
                    msg.twist.twist.linear.y = pub_twist(1,2);
                    msg.twist.twist.linear.z = pub_twist(1,3);

                    msg.twist.twist.angular.x =
evt.getSource.angular_velocity.x;
                    msg.twist.twist.angular.y =
evt.getSource.angular_velocity.y;
                    msg.twist.twist.angular.z =
evt.getSource.angular_velocity.z;
                    msg.twist.covariance = [0.0392,0,0,0,0,0,
0,0.0392,0,0,0,0, 0,0,0.0392 ,0,0,0,   0,0,0,0.00122,0,0, 0,0,0,0,0.00122,0,
0,0,0,0,0, 0.00122];

                    o.Publish('/odom',msg);


                    % might not need below line becuase the messege type does
not
```

```
                    % have covariance. It will be set in ROS.
                    %pose_msg_pub.covariance = zeros(1,36);

                    %             pose_msg_pub.pose.position;
                    %             pose_msg_pub.pose.orientation;



                    %% Plot 3D Position

                    %         posex = linx*dt^2 + pose(1,1);
                    %         posey = liny*dt^2+pose(1,2);
                    %         posez = linz*dt^2+pose(1,3);


                    %         circshift(pose,1);
                    %         pose(1,1) = posex;
                    %         pose(1,2) = posey;
                    %         pose(1,3) = posez;
                    %
                    %         plot3(pose(1,1),pose(1,2),pose(1,3),'-o');
                    %         xlim([-1 1]);
                    %         ylim([-1 1]);
                    %         zlim([-1 1]);

                end

            end


        end

    end

end
```

Appendix C: Drone Launch File

```
<launch>
<!-- ardrone_autonomy Launch -->
      <node         name="ardrone_driver"         pkg="ardrone_autonomy"
type="ardrone_driver" output="screen" clear_params="true">
          <param name="outdoor" value="1" />
          <param name="max_bitrate" value="4000" />
          <param name="bitrate" value="4000" />
          <param name="navdata_demo" value="0" />
          <param name="flight_without_shell" value="0" />
          <param name="altitude_max" value="3000" />
        <param name="altitude_min" value="50" />
        <param name="euler_angle_max" value="0.21" />
        <param name="control_vz_max" value="700" />
          <param name="control_yaw" value="1.75" />
          <param name="detect_type" value="10" />
          <param name="enemy_colors" value="3" />
          <param name="detections_select_h" value="32" />
          <param name="detections_select_v_hsync" value="128" />
          <param name="enemy_without_shell" value="0" />
          <param name="do_imu_caliberation" value="true" />
          <param name="tf_prefix" value="mydrone" />
        <!-- Covariance Values (3x3 matrices reshaped to 1x9)-->
        <rosparam                                                file="$(find
ardrone_autonomy)/data/camera_info/ardrone_front.yaml"/>
        <rosparam                                                file="$(find
ardrone_autonomy)/data/camera_info/ardrone_bottom.yaml"/>
        <rosparam  param="cov/imu_la">[0.1,  0.0,  0.0,  0.0,  0.1,  0.0,
0.0, 0.0, 0.1]</rosparam>
        <rosparam  param="cov/imu_av">[1.0,  0.0,  0.0,  0.0,  1.0,  0.0,
0.0, 0.0, 1.0]</rosparam>
        <rosparam  param="cov/imu_or">[1.0,  0.0,  0.0,  0.0,  1.0,  0.0,
0.0, 0.0, 100000.0]</rosparam>
        <remap from="/ardrone/navdata/imu" to="raw_imu"/>
      </node>

  <!--ar_pose Launch
    <node pkg="rviz" type="rviz" name="rviz"
      args="-d $(find ar_pose)/launch/live_reverse.rviz"/>
    <node            pkg="tf"            type="static_transform_publisher"
name="world_to_marker"
      args="0 1 1 -1.57 3.14 -1.57 world ar_marker 10" />
    <node      ns="ardrone"      pkg="image_proc"      type="image_proc"
name="image_proc"/>

    <node  name="ar_pose"  pkg="ar_pose"  type="ar_single"  respawn="false"
output="screen">
      <param              name="marker_pattern"              type="string"
value="data/data/patt.hiro"/>
      <param name="marker_width" type="double" value="152.4"/>
      <param name="marker_center_x" type="double" value="0.0"/>
      <param name="marker_center_y" type="double" value="0.0"/>
      <param name="threshold" type="int" value="100"/>
      <param name="use_history" type="bool" value="true"/>
      <param name="reverse_transform" type="bool" value="true"/>
    </node>
```

```
       hector_pose Launch
        <launch>
        <node    name="PoseEstimationNode"    pkg=    "hector_pose_estimation"
type="pose_estimation"/>
        <node     pkg="nodelet"     type="nodelet"     name="captain_nodlete"
args="manager"/>
        <node pkg="nodelet" type="nodelet" name="pose_estimation_nodelet"
              args="load        hector_pose_estimation/pose_estimation_nodelet
manager">
        </node>-->
  </launch>
```

Appendix D: Python Drone Controller

```
#!/usr/bin/env python
import roslib; roslib.load_manifest('ardrone')
roslib.load_manifest('ardrone_autonomy');
import rospy;
import time;
from geometry_msgs.msg._Pose import Pose
from std_msgs.msg import Empty                  # for land/takeoff/emergency
from geometry_msgs.msg import Twist             # for sending commands to the
drone
from ardrone_autonomy.msg import Navdata        #for receiving navdata feedback
from drone_status import DroneStatus
from geometry_msgs.msg import PoseStamped

# Some Constants
COMMAND_PERIOD = 100 #ms


class BasicDroneController(object):
    def __init__(self):
        self.status = Navdata()
        self.prevStatus = Navdata()
#         self.dt = 0.0
#         self.currvX = 0.0
#         self.currvY = 0.0
#         self.currvZ = 0.0
#         self.pose = PoseStamped()
#         self.currpX = 0.0
#         self.currpY = 0.0
#         self.currpZ = 0.0
#         self.currqX = 0.0
#         self.currqY = 0.0
#         self.currqZ = 0.0
#         self.currqW = 0.0
#         self.waypX = 0.0
#         self.waypY = 0.0
#         self.waypZ = 0.0
#         self.wayqX = 0.0
#         self.wayqY= 0.0
#         self.wayqZ = 0.0
#         self.wayqW = 0.0
#         self.dronePose  = rospy.Subscriber('pose', PoseStamped,
self.ReceivePose)
#         self.waypoint   = rospy.Subscriber('/waypoints', PoseStamped,
self.ReceiveWaypoint)
        self.subNavdata = rospy.Subscriber('/ardrone/navdata', Navdata,
self.ReceiveNavdata)
        self.pubLand    = rospy.Publisher('/ardrone/land',Empty)
        self.pubTakeoff = rospy.Publisher('/ardrone/takeoff', Empty)
        self.pubReset   = rospy.Publisher('/ardrone/reset',Empty)
        self.pubCommand = rospy.Publisher('/cmd_vel',Twist)
        self.command    = Twist()
        self.commandTimer =
rospy.Timer(rospy.Duration(COMMAND_PERIOD/1000.0), self.SendCommand)
        rospy.on_shutdown(self.SendLand)
        pass
```

```
    def ReceiveNavdata(self,data):
        if False: self.prevStatus = Navdata()
        self.prevStatus = self.status
        self.status = data

        self.currvX = data.vx
        self.currvY = data.vy
        self.currvZ = data.vz
        #time between packets in milliseconds
        self.dt = (data.tm - self.prevStatus.tm)/1000#(data.header.stamp.secs
- self.prevStatus.header.stamp.secs)*1000 + \
#                  (data.header.stamp.nsecs -
self.prevStatus.header.stamp.nsecs)/1000000.0

        pass

#     def ReceivePose(self, data):
#         if False: self.pose = PoseStamped()
#         self.pose = data
#         self.currpX = data.Pose.Point.x
#         self.currpY = data.Pose.Point.y
#         self.currpZ = data.Pose.Point.z
#         self.currqX = data.Pose.Quaternion.x
#         self.currqY = data.Pose.Quaternion.y
#         self.currqZ = data.Pose.Quaternion.z
#         self.currqW = data.Pose.Quaternion.w
#         print "pose x:", self.xPt, "pose y:", self.yPt, "pose z", self.zPt
#         print "quat x:", self.qtX, "quat y:", self.qtY, "quat z", self.qtZ
#         print "quat w:", self.qtW
#         pass
#
#     def ReceiveWaypoint(self, data):
#         if False: self.pose = PoseStamped()
#         self.pose = data
#         self.waypX = data.Pose.Point.x
#         self.waypY = data.Pose.Point.y
#         self.waypZ = data.Pose.Point.z
#         self.wayqX = data.Pose.Quaternion.x
#         self.wayqY = data.Pose.Quaternion.y
#         self.wayqZ = data.Pose.Quaternion.z
#         self.wayqW = data.Pose.Quaternion.w
#         print "pose x:", self.xPt, "pose y:", self.yPt, "pose z", self.zPt
#         print "quat x:", self.qtX, "quat y:", self.qtY, "quat z", self.qtZ
#         print "quat w:", self.qtW
#         pass

    def SendTakeoff(self):
        self.pubTakeoff.publish(Empty())
#         if self.status is DroneStatus.Landed:
#             self.pubTakeoff.publish(Empty())
#             pass
        pass


    def SendLand(self):
```

```
        self.pubLand.publish(Empty())
        pass

    def SendEmergency(self):
        self.pubReset.publish(Empty())
        pass


    def SetCommand(self,roll, pitch, z_velocity, yaw_velocity):
        self.command.linear.x  = pitch
        self.command.linear.y  = roll
        self.command.linear.z  = z_velocity
        self.command.angular.z = yaw_velocity
        pass

    def SendCommand(self,event=''):
        if self.status is DroneStatus.Flying or self.status is
DroneStatus.GotoHover or self.status is DroneStatus.Hovering:
            self.pubCommand.publish(self.command)
            pass
        pass
    #end class


if __name__ == '__main__':
    try:
        rospy.init_node('drone_controller')
        bdc = BasicDroneController()
        global data
        data = 0
        x = 0
        while bdc.command.angular.z is not 0:
            bdc.SetCommand(0, 0, 0, 0)

        while not rospy.is_shutdown():

            bdc.SendTakeoff()
            pass
        bdc.SendLand()


#             while bdc.status is DroneStatus.Landed or not bdc.status is
DroneStatus.TakingOff:
#                 bdc.SendTakeoff()
#                 print 'status', bdc.status.state
#                 #if bdc.status.state is 6 : break
#                 pass
#             print 'out of while'
#
#             while 1: #bdc.status is DroneStatus.Flying or bdc.status is
DroneStatus.Hovering:
#                 bdc.SendLand()
#                 pass
#             print 'outside'
#             bdc.SendLand()

    except rospy.ROSInterruptException: pass
```

Appendix E: Python Drone Navigation Data Extraction

```
#!/usr/bin/env python
import roslib; roslib.load_manifest('ardrone')
roslib.load_manifest('ardrone_autonomy');
import rospy;

from geometry_msgs.msg import TwistStamped          # for sending commands to
the drone
from ardrone_autonomy.msg import Navdata      #for receiving navdata feedback
from drone_controller import BasicDroneController


class NavDataConverter(object):
    def __init__(self):
        self.pubVels  = rospy.Publisher('/curr_vel', TwistStamped)
        self.currVel  = TwistStamped()
    pass

    def SendVels(self):
        self.currVel.linear.x = bdc.currvX
        self.currVel.linear.y = bdc.currvY
        self.currVel.linear.z = bdc.currvZ
        self.pubVels.publish(self.currVel)
        pass


if __name__ == '__main__':
    try:
        rospy.init_node('NavData')
        bdc = BasicDroneController()
        ndc = NavDataConverter()
        while not rospy.is_shutdown():
            ndc.SendVels()
            print 'sending vels'
            #print ndc.currVel.linear.x
            pass
    except rospy.ROSInterruptException:
        bdc.SendLand()
        pass
```

Appendix F: Python Drone Flight Controller

```
#!/usr/bin/env python
import roslib; roslib.load_manifest('ardrone')
roslib.load_manifest('ardrone_autonomy');
import rospy;
import time;
from std_msgs.msg import Empty                  # for land/takeoff/emergency
from geometry_msgs.msg import Twist             # for sending commands to the
drone
from drone_status import DroneStatus
from drone_controller import BasicDroneController

class DroneFlightControl(object):
    def __init__(self, navdata_subscriber):
        self.navdata_sub = navdata_subscriber
        self.first = True
        self.xDist = 0
        self.yDist = 0

    def getDist(self):
        #integrate x distance
        if -15 < bdc.currvX < 15:
            currvX = 0
            pass
        else:
            currvX = bdc.currvX/1000
            pass
        self.xDist += currvX*(bdc.dt/1000000)

        #integrate y distance
        if -30 < bdc.currvY < 30:
            currvY = 0
            pass
        else: currvY = bdc.currvY/1000
        self.yDist += currvY*(bdc.dt/1000000)

        pass

    def fakeNav(self, goalX, goalY):
        dfc.getDist()
        xerror = goalX - self.xDist #forward = +; backward = - distance
        yerror = goalY - self.yDist #left = +; right = - distance
        if xerror < -0.25: #overshoot
            pitch = -0.1 #backward
            pass
        elif 0.25 < xerror: #undershoot
            pitch = 0.1 #forward
            pass
        else:
            pitch = 0
            #print 'within acceptable error'
            pass
        if yerror < -0.25:
            roll = 0.1 #left
            pass
        elif 0.25 < yerror:
```

```
            roll = -0.1 #right
            pass
        else:
            roll = 0
            #print 'such y fail'
            pass

        bdc.SetCommand(roll, pitch)
        if xerror == 0 and yerror == 0:
            bdc.SendLand()
            time.sleep(5)
            pass

        print 'xerror =', xerror
        print
        print 'yerror =', yerror
        print
        pass

    def linearize(self):
        droneMax = 1.0
        droneMin = 0.0
        ctrlMax = 10
        ctrlMin = -10
        ctrlMax  =  (ctrlMax  -  ctrlMin)*(droneMax  -  droneMin)/(droneMax  -
droneMin)
        return

    def PD(self, kp, kd, goalX, goalY):
        dfc.getDist()
        xerror = goalX - self.xDist #forward = +; backward = - distance
        yerror = goalY - self.yDist #left = +; right = - distance

        if dfc.first is True:
            xprevError = 0
            yprevError = 0
            dt = 1
            dfc.first = False
            pass
        else:
            dt = bdc.dt
            xprevError = xerror
            yprevError = yerror
            pass


        if -0.25 < xerror < 0.25: xerror = 0
        if -0.25 < yerror < 0.25: yerror = 0

        xderivative = (xerror - xprevError)/dt
        yderivative = (yerror - yprevError)/dt
        xPval = kp * xerror
        yPval = kp * yerror
        xDval = kd * xderivative
        yDval = kd * yderivative
        xprevError = xerror
        yprevError = yerror
```

```
        xupdate = xPval + xDval
        yupdate = yPval + yDval

        print 'xerror', xerror, 'xupdate',xupdate
        print
        print 'yerror', yerror, 'yupdate', yupdate
        print
        pass

if __name__ == '__main__':
    try:
        rospy.init_node('toDest')
        bdc = BasicDroneController()
        dfc = DroneFlightControl(bdc)
        r = rospy.Rate(0.5) # 10hz
        first = True
        x = 0
        y = 0

        while bdc.status is DroneStatus.Landed or bdc.status is
DroneStatus.TakingOff:
            bdc.SendTakeoff()

            pass

        while not rospy.is_shutdown():
            #bdc.SendTakeoff()
            dfc.fakeNav(2, 1)
            #dfc.PD(0.5, 0.5, 2, 1)

            bdc.SendCommand()

            pass

#     start_time = time.time()
#     while 1:
#         if (time.time() - start_time) % 0.5 == 0:
#             PID(0, 1, 2, 3, 4)
#             print time.time()
#             pass
#         pass
#     pass


#         rospy.Timer(rospy.Duration(0.01), counter())
#         threading.Timer(0.01, counter)
#         print count
#         if count % 100 == 0:
#             PID(0, 1, 2, 3, 4)
#             print count
#             pass
#         pass

    except rospy.ROSInterruptException:
        bdc.SendLand()
        pass
```

Appendix G: MATLAB Drone Navigation

```
classdef droneVels < Node

    properties
        processing = 0;
        first = 0;
        xDist = 0;
        yDist = 0;
        vx = 0;
        vy = 0;
        xupdate = 0;
        yupdate = 0;
        xprevError = 0;
        yprevError = 0;
        currTime = 0;
        dt = 0;

    end

    methods
        function obj = droneVels(nodeName)
            obj = obj@Node(nodeName);

obj.addSubscriber('/curr_vel','geometry_msgs/TwistStamped',@obj.droneVelCallb
ack);
            obj.addPublisher('/cmd_vel','geometry_msgs/Twist');
            obj.addPublisher('/ardrone/takeoff', 'std_msgs/Empty');

        end
        %% Initialization
        function [] = initialize(o, cmdVel)
            cmdVel.linear.x = 0;
            cmdVel.linear.y = 0;
            cmdVel.linear.z = 0;
            cmdVel.angular.x = 0;
            cmdVel.angular.y = 0;
            cmdVel.angular.z = 0;
        end

        %% Velocity Callback
        function [] = droneVelCallback(o,handle,evt)
            if (~o.processing)
                disp('processing');
                o.processing = 1;
                prevTime = o.currTime;
                o.vx = evt.getSource.Twist.linear.x;
                o.vy = evt.getSource.Twist.linear.y;
                o.currTime = evt.getSource.header.stamp.nsecs;
                o.dt = o.currTime - prevTime;
                o.run(evt);
                o.processing = 0;
            end
            disp('end processing');

        end
```

```
 %% Run
function [] = run(o,evt)
    disp('run');
    empty = org.ros.message.std_msgs.Empty();
    cmdVel = org.ros.message.geometry_msgs.Twist();
    if(~o.first)
        o.xprevError = 0;
        o.yprevError = 0;
        o.dt = 0.00001;
        o.initialize(cmdVel);
        o.droneTakeoff(empty);
        o.first = 1;
    end

%         o.getDistance();
%         o.PD(2, 1);
%         cmdVel.twist.linear.x = o.xupdate;
%         cmdVel.twist.linear.y = o.yupdate;
    o.Publish('/cmd_vel',cmdVel);
    disp('about to land');
    o.droneLand(empty);

end
%% Takeoff
function [] = droneTakeoff(o, empty)
    disp('Taking off');
    tic;
    while (toc<3)
        o.Publish('/ardrone/takeoff', empty);
    end
end

%% Land
function [] = droneLand(o, empty)
    disp('Landing');
    tic;
    while(toc<5)
        o.Publish('/ardrone/land', empty);
    end
end

%% Distance Accumulation
function [] = getDistance(o)
    %integrate x distance
    if -15 < vx < 15
        vx = 0;
    else
        vx = o.vx/1000;
    end
    o.xDist = o.xDist + vx*(dt/1000000);

    %integrate y distance
    if -30 < vy < 30
        vy = 0;
    else
        vy = o.vy/1000;
    end
```

```
            o.yDist = o.yDist + vy*(dt/1000000);
        end

        %% PD Control
        function [] = PD(o, kp, kd, goalX, goalY)
            xerror = goalX - o.xDist; %forward = +; backward = - distance
            yerror = goalY - o.yDist; %left = +; right = - distance
            dt = o.dt;

            if -0.25 < xerror < 0.25
                xerror = 0;
            end
            if -0.25 < yerror < 0.25
                yerror = 0;
            end

            xderivative = (xerror - o.xprevError)/o.dt;
            yderivative = (yerror - o.yprevError)/o.dt;
            xPval = kp * xerror;
            yPval = kp * yerror;
            xDval = kd * xderivative;
            yDval = kd * yderivative;
            o.xprevError = xerror;
            o.yprevError = yerror;
            o.xupdate = xPval + xDval;
            o.yupdate = yPval + yDval;
            o.linearize();
        end

        %% Linearize Control Signal
        function[] = linearize(o)
            droneMax = 1.0;
            droneMin = 0.0;
            ctrlMax = 10;
            ctrlMin = -10;
            lin = (ctrlMax - ctrlMin)*(droneMax - droneMin)/(droneMax -
droneMin);
            o.xupdate = o.xupdate*lin;
            o.yupdate = o.yupdate*lin;
        end

    end
end
```

Appendix H: MATLAB Drone Blob Detection

```
classdef droneImage < Node
    properties
        first = 0;
        imdata;
        height;
        width;

    end

    methods
        function obj = droneImage(nodeName)
            obj = obj@Node(nodeName);

            % new method

obj.addSubscriber('/ardrone/image/front/image_raw','sensor_msgs/Image',@obj.d
roneImageCallback);

        end
        %% Initialization
        function [] = initialize(o)

        end

        %% Image Callback
        function [] = droneImgCallback(o,handle,evt)
            o.imdata = evt.getSource.data;
            o.height = evt.getSource.height;
            o.width = evt.getSource.width;
            o.run(evt);

        end

        %% Run
        function [] = run(o,evt)
            img = org.ros.message.sensor_msgs.Image();

            if(~o.first)

                o.initialize(img);

                o.first = 1;
            end
        end
    end
end
```