Fall 2017

Ego-localization Navigation for Intelligent Vehicles using 360° LIDAR Sensor for Point Cloud Mapping

Tyler Naes

EGO-LOCALIZATION NAVIGATION FOR INTELLIGENT VEHICLES USING 360° LIDAR SENSOR FOR POINT CLOUD MAPPING

by

TYLER NAES

(Under the direction and supervision of Valentin Soloiu)

ABSTRACT

With its prospects of reducing vehicular accidents and traffic in highly populated urban areas by taking the human error out of driving, the future in automobiles is leaning towards autonomous navigation using intelligent vehicles. Autonomous navigation via Light Detection And Ranging (LIDAR) provides very accurate localization within a predefined, *a priori*, point cloud environment that is not possible with Global Positioning System (GPS) and video camera technology. Vehicles may be able to follow paths in the point cloud environment if the baseline paths it must follow are known in that environment by referencing objects detected in the point cloud environment and localizing its position to a high degree of accuracy. This investigation used a low cost two-dimensional (2-D) LIDAR to establish landmarks' coordinates in a point cloud environment, known as ego-points, and proceeded to navigate the environment mimicking a human driver while plotting its path in the point cloud environment. The vehicle then navigated the environment by referencing the ego-points and followed the recorded plotted baseline path. Results indicate that the intelligent vehicle was able to follow the baseline paths while having max normalized deviation away from the path of only 0.024 cm/cm; this deviation fell well within the established tolerance of navigating real world lane dimensions.

INDEX WORDS: Autonomous navigation, Lidar, Robot operating system, Intelligent vehicles, Point cloud, Localization

EGO-LOCALIZATION NAVIGATION FOR INTELLIGENT VEHICLES USING 360° LIDAR SENSOR FOR POINT CLOUD MAPPING

by

TYLER NAES

B.S., Georgia Southern University, 2015

A Thesis Submitted to the Graduate Faculty of Georgia Southern University in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

STATESBORO, GEORGIA

EGO-LOCALIZATION NAVIGATION FOR INTELLIGENT VEHICLES USING 360° LIDAR SENSOR FOR POINT CLOUD MAPPING

by

TYLER NAES

Major Professor:	Valentin Soloiu
Committee:	Fernando Rios-Gutierrez
	Biswanath Samanta

Electronic Version Approved:
December 2017

DEDICATION

To my father Michael and my brothers Jason, Mathew, Austin, and Ross.

In loving memory of Cindy and Reid.

ACKNOWLEDGEMENTS

I would like to thank Dr. Valentin Soloiu for allowing me to conduct my research in his lab and for supervising me. I also want to thank all my colleagues who helped me along the way: Tom, Bernard, Charvi, Martin, Remi, Jose, Aliyah and the entire Georgia Southern University mechanical engineering department faculty.

TABLE OF CONTENTS

Page

LIST OF TABLES

page

LIST OF FIGURES

page

CHAPTER 1. INTRODUCTION

1.1 The Need for Autonomous Navigation

Increasing the number of intelligent vehicles on the road is a major goal for not only automobile manufacturers but for society as a whole. Transitioning from manually driven automobiles will not only reduce the number of vehicular related injuries/deaths but it will also increase traffic flow in high vehicle density areas. Increasing traffic flow will reduce commute times for people on the road as well as reduce emissions that would be released from vehicles stuck in stop-and-go traffic.

Most vehicular accidents are due to human error from the driver of a vehicle. Modern day drivers are susceptible to numerous and various distractions while the driver is operating a vehicle. Texting while driving has been an issue for drivers because the driver has to take their eyes off the road which allows vehicular accident scenarios to occur. It is important to take the responsibility of driving away from humans and develop vehicles that will autonomously navigate an environment for the driver.

Some degree of autonomous navigation has been achieved in recently developed intelligent vehicles from companies like Google and Tesla. The intelligent vehicle models from Tesla are consumer purchasable however, their models still require the driver to maintain control of the steering wheel in case a scenario occurs where the vehicle is not able to navigate the environment safely. This is due in large part to algorithms not being fully developed to make decisions from real world road hazards that can suddenly occur like a human driver can. Human drivers are also able to assess the situations of other vehicles that have an effect on his/her own vehicles. For example, if a car in front of a driver's car suddenly has to put on the brakes due to an object coming into the lane, both cars can put on brakes almost simultaneously; intelligent vehicles would have

to wait until the car in front starts braking before it realizes it needs to slow down. This is why the intelligent vehicles must be able to communicate and interact with each other in order to enhance human safety on the road.

1.2 Urban Area Navigation

Urban environments are the areas that intelligent vehicles will have the most positive impact. Inner cities and their surrounding suburbs have the highest number of vehicles on the road and due to the number of commuters entering and leaving these areas, they are also the source of most stop-and-go traffic occurrences. Traffic in urban areas is due to the unsynchronized timings of when to start accelerating between vehicles. Typically, drivers who are waiting behind another vehicle must wait for the vehicle in front of him/her to build some distance between the two cars before the driver starts to accelerate also. The time difference between starting acceleration times between vehicles is compounded for each vehicle in a line. If vehicles were able to start accelerating at the same time then higher traffic flow would result with the reduction of the compounding time delays between accelerations.

In order for vehicles to move together, their paths must be planned ahead and then executed at the same time through vehicle to vehicle (V2V) communication. Vehicles will only be able to plan their paths if their environment is mapped out ahead of time for their reference and they know their exact position in that environment. Modern day autonomous vehicles do not know their exact position on the road; they only have a rough estimate from GPS and they use on board sensors collect data about their surrounding environment and react to that data in order to navigate autonomously. This navigation technique does not collaborate with other vehicles on the road so traffic jams will still easily occur.

1.3 Building Database for Future Navigation

Urban environments can be mapped out for intelligent vehicles by storing sensor data taken from various vehicles commuting through these environments. If modern day vehicles were fitted with sensors to build the environment, the paths that human drivers take while traveling through the environment can be recorded in such a manner that intelligent vehicles know to follow the recorded path in order to mimic human driving. One method for building an environment would be to use a multi-dimensional LIDAR scanner to build a point-cloud environment of permanently stationary objects in urban environments for intelligent vehicles to use as a reference to localize itself in that point cloud environment. These permanently stationary objects can range from objects as small as street signs, to objects as large as buildings. LIDAR sensing navigation can localize a vehicle very precisely due to its low variance in determining an objects range. As human drivers navigate urban areas, their point cloud environment data can be uploaded to a cloud based system to build a virtual environment for intelligent vehicles to navigate through, thus allowing full autonomous driving in urban areas. V2V communication for synchronized acceleration timings would then be feasible, resulting in higher flow of traffic in urban areas.

1.4 Hypothesis

If ego-points' coordinates are determined in an *a priori* environment using a 2-D LIDAR scanner while navigating a path, then a vehicle can autonomously follow that path by only referencing the ego-points in the *a priori* environment generated from data from the vehicle's local 2-D LIDAR scanner.

CHAPTER 2. LITERATURE REVIEW

2.1 Current Intelligence in Autonomous Vehicles

Autonomous vehicles are capable of reducing the number of driving accidents by operating on algorithms instead of being controlled by human drivers. These algorithms are based on information constantly streamed to the cars Electronic Control Unit (ECU) from locally instrumented sensors. Modern consumer vehicles have implemented various passive and active sensors to reduce on-road incidents; however, the future of autonomous driving requires a higher level of sophistication in sensor instrumentation than what is currently being used. The Society of Automotive Engineers (SAE) has outlined the six levels of automation in vehicles provided in Table 1 (NHTSA 2016).

Table 1: Society of Automotive Engineers Levels of Autonomy

SAE level of autonomy	Description of level capabilities.
Level 0	The human driver does everything
Level 1	An automated system on the vehicle can sometimes assist the human driver conduct some parts of the driving task;
Level 2	An automated system on the vehicle can actually conduct some parts of the driving task, while the human continues to monitor the driving environment and performs the rest of the driving task
Level 3	An automated system can both actually conduct some parts of the driving task and monitor the driving environment in some instances, but the human driver must be ready to take back control when the automated system requests
Level 4	An automated system can conduct the driving task and monitor the driving environment, and the human need not take back control, but the automated system can operate only in certain environments and under certain conditions
Level 5	The automated system can perform all driving tasks, under all conditions that a human driver could perform them

Modern day vehicles utilize some intelligent features to assist drivers in detecting avoidable objects and navigating lanes safely. Adaptive Cruise Control (ACC) helps drivers maintain a safe and relatively constant distance from other cars in front of them when in cruise

control. This is possible by using Radio Detection and Ranging (RADAR) sensors to detect the distance of objects in front of the vehicle while ACC adjusts vehicular speed to maintain a safe distance (Nkoro and Vershinin 2014). Lane detection cameras are implemented on some models to warn drivers when the car is navigating out of it's lane while steering the car safely into the middle of the lane (Nkoro and Vershinin 2014). GPS is routinely used in self-navigating vehicles. GPS allows the vehicle to accurately get its lattitudinal position within 9 meters and longitudinal position within 15 meters with a confidence interval of 95% (Grimes 2008). This is beneficial for general localization of vehicles and navigating road routes. GPS is, however, unreliable when a vehicle is in an environment that blocks communication with satellites such as tunnels and heavily forested areas. Inertial Measurement Units (IMU) allow the monitoring of vehicular rotations along all three-Dimensional (3-D) axes and translations by using onboard accelerometers and gyroscopes. Because the IMU is a local sensor, it is not vulnerable to the signal interference that GPS sensors encounter (Nkoro and Vershinin 2014).

Autonomous navigation in intelligent vehicles is only possible if the ECU in the intelligent vehicle has enough memory to store the algorithms that controls the vehicle based on sensor data. Today, ECUs in intelligent vehicles are efficient enough for companies like Tesla and Google who have already developed vehicles for autonomous navigation (Hirsch 2015, Guizzo 2011). In parallel with research on autonomous navigation methods, there is research currently being conducted on furthering this technology with self-parking intelligence in vehicles. Literature has been established on the best type of parking structures for self-driving cars as well as effective methods for managing self-parking for various vehicles (Tariq et al. 2016).

2.2 Simultaneous Localization and Mapping

2.2.1 Overview and History

A major issue for autonomous vehicles in an unknown environment is locating its position within that environment and navigating appropriately. In order for an intelligent vehicle to navigate an unknown environment, the vehicle must build a virtual model of its environment. This can be achieved by instrumenting the intelligent vehicle with local sensors that can extract data from its surroundings to find environmental landmarks. This data is then processed through various algorithms to build its virtual environment with the surrounding landmarks and determine its location in the environment relative to those landmarks. Since the intelligent vehicle is able to generate its environment and locate itself in that environment, it is then capable of navigating the unknown environment autonomously. However, because the landmarks have no absolute globally set positions, translation errors such as those shown in Figure 1 are inevitable.

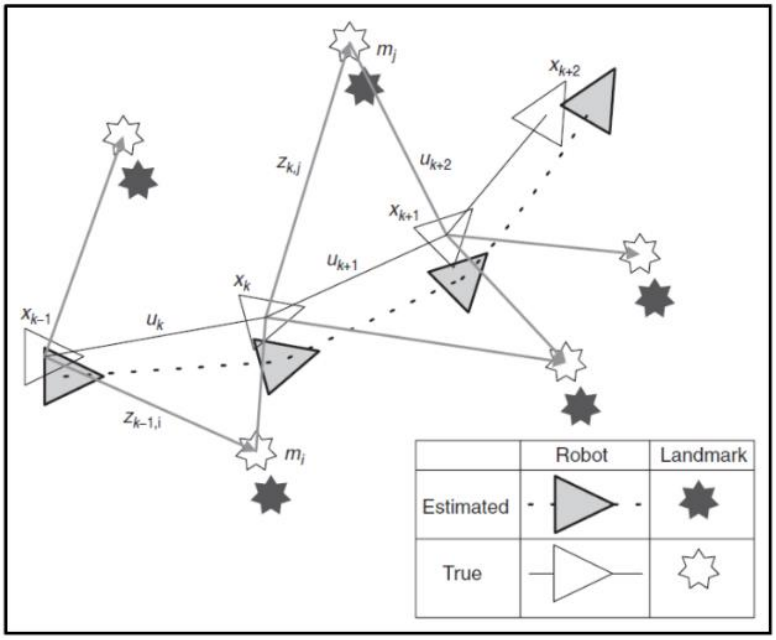

Figure 1: SLAM Process for Robot's Estimated and True Positions in an Environment (Durrant-Whyte and Bailey 2006)

The general process of mapping and localization in autonomous vehicles is provided in Figure 2. The data is also stored for localization and map building purposes for successive scans which can accumulate large amounts of computer memory.

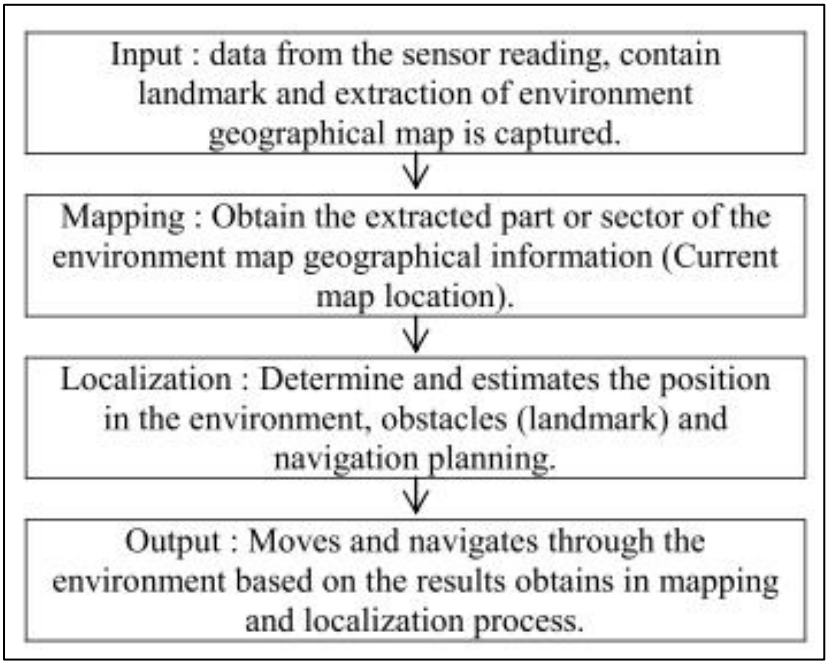

Figure 2: Robotic Process for Performing Autonomous Navigation (Khairuddin, Talib, and Haron 2015)

Simultaneous Localization and Mapping (SLAM) is the ability for an autonomous system to map out it's surrounding in an unknown environment and localize itself within that environment simultaneously (Durrant-Whyte and Bailey 2006, Davison 2003). The general process of performing a SLAM solution to a navigation problem is provided in Figure 3. However, modern SLAM algorithms require a closed loop process that makes changes to its mapping during successive scans of the environment. The general closed loop process of SLAM is provided in Figure 4.

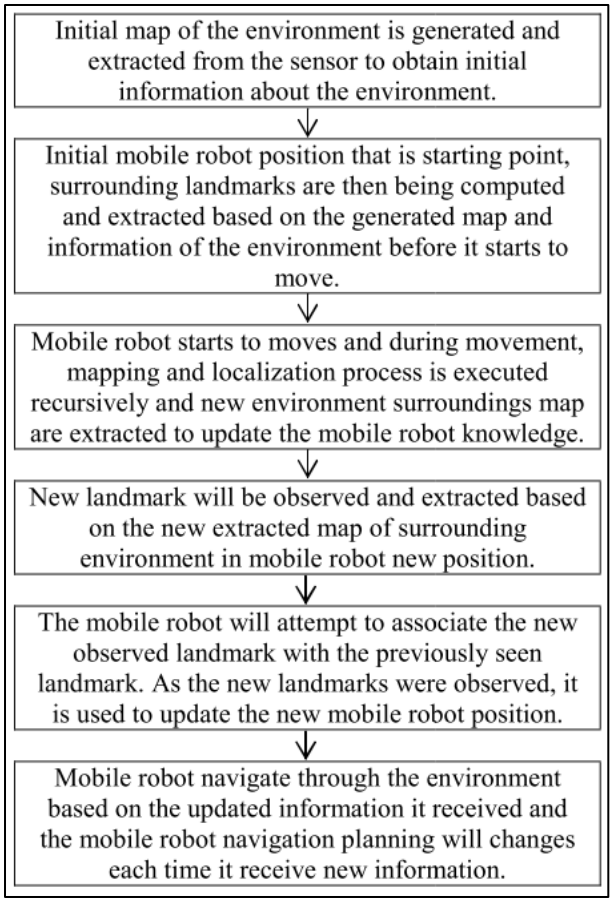

Figure 3: Process of Simultaneous Localization and Mapping Overview (Khairuddin, Talib, and Haron 2015)

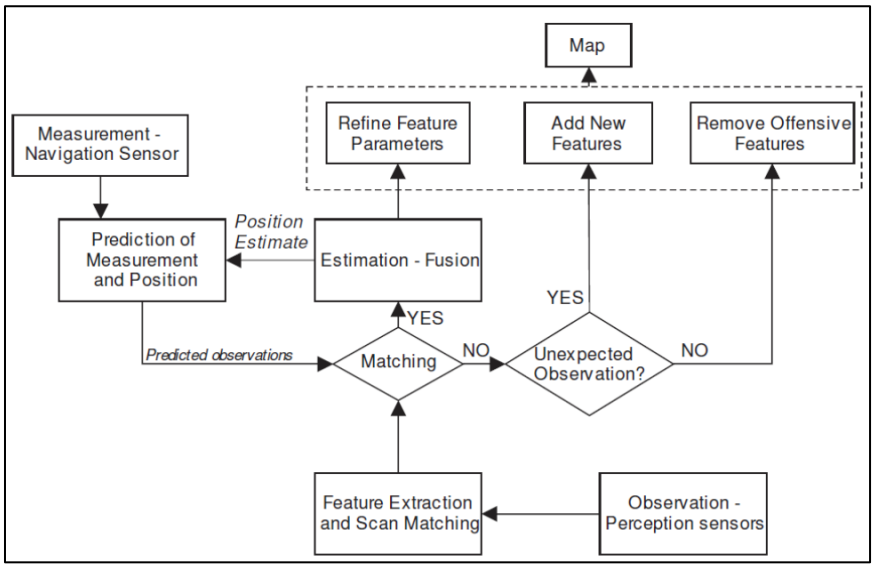

Figure 4: SLAM Process Block Diagram (Hidalgo and Bräunl 2015)

Numerous SLAM algorithms have been developed in order for vehicles to autonomously navigate new environments while also mapping out its surroundings (Khairuddin, Talib, and Haron 2015). Earliest literature on SLAM development in autonomous vehicles have provided valuable insight on the correlation of identified landmarks from each scan and the errors that come from them. A statistical component of SLAM technology determined that there must be high relationship correlations with landmarks in the environment and that these correlations become stronger through successive observations (Smith and Cheeseman 1986, Durrant-Whyte 1988). The landmarks in the environment have to be interconnected with each other due to the common error in vehicular localization within that environment as shown in later literature (Smith, Self, and Cheeseman 1987); However, this error would not occur if the landmarks have predefined global positions that are unchanging. The landmark correlation interconnections, shown in Figure 5, create a web-like spatial relationship between all landmarks in the environment that require one-to-one spatial relationships equal to the square of the number of landmarks (Durrant-Whyte and Bailey 2006).

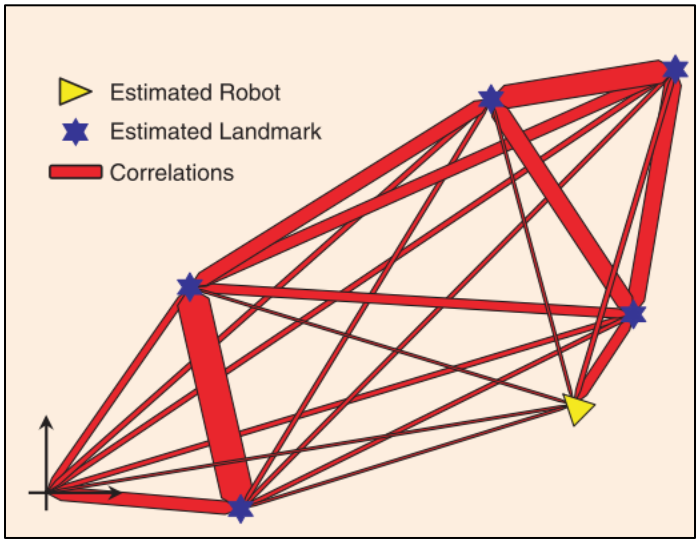

Figure 5: Correlations between Landmarks and a Robotic Vehicle

This means the amount of computing power required to update and store these correlations increases exponentially whenever another landmark is discovered/added to the environment. If there is a strong correlation between the spatial relationships then ideally an autonomous system would want to use the least amount possible of landmarks to localize itself to maximize computational efficiency. This investigation aims to show that an intelligent vehicle would be able to obtain its position and pose by referencing only 3 predefined landmarks which is the minimal number for position triangulation.

2.2.2 SLAM Computational Issues with Filtering

An issue with SLAM's computational efficiency led to research in SLAM with data filtering techniques such as Extended Kalman-Filter (EKF-SLAM) and Rao-Blackwellization (Fast-SLAM). EKF implementation in SLAM methodology was first introduced by Thrun in 1998 (Thrun, Burgard, and Fox 1998). Performing EKF-SLAM requires a high degree of computation power due to updating the landmarks and joint covariance matrix for every observation instant. Because each landmark has a correlation with all the other landmarks, the computation growth exponentially increases with every new landmark detected. Research in performing different variants of EKF-SLAM have improved the computational efficiency compared to normal EKF-SLAM (Guivant and Nebot 2001, J. Leonard, Jacob, and Feder 2000). However, EKF-SLAM is susceptible to data association errors when making observations. Sometimes incorrect landmark associations can be detected and this causes mapping error as shown in literature (Neira and Tardos 2001). Association errors can occur by detecting landmarks that are not there or missing a landmark that should be detected. Because no landmark has a global position, correlation convergence occurs at an error that is due to the unknown exact position of the robot when it started scanning; this error convergence is demonstrated in Figure 6 from results in previous literature.

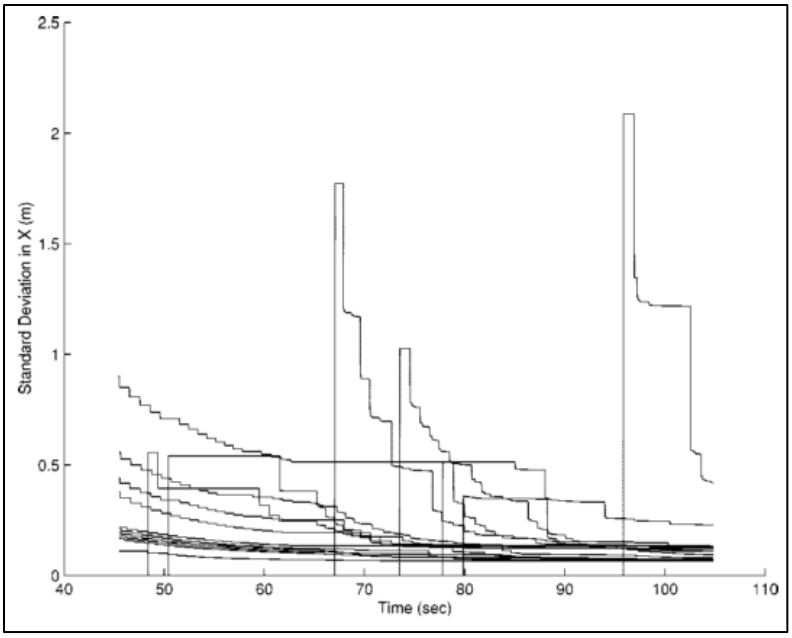

Figure 6: Typical EKF-SLAM Convergence of Landmark Uncertainty (Dissanayake et al. 2001)

This investigation aims to show that an intelligent vehicle is able to accurately navigate a path by dismissing invalid scans that would traditionally cause issue in EKF-SLAM. Correlation convergence will also be avoided due to using global coordinate systems for the autonomous vehicles position from the start.

Landmark detection for intelligent vehicles can be extremely difficult when the landmark identity changes at different distances and viewpoints. An example of these type of landmarks would be direction arrows painted on lanes or road signs that change shades depending on the time of day and/or weather condition. Research has been performed that used cameras to confirm landmarks on roadways that appear differently at different angles (Ranganathan, Ilstrup, and Wu 2013). However, these difficulties can be bypassed altogether by detecting distinct physical objects with object detecting sensors such as a LIDAR scanner. SLAM also updates its landmarks in the virtual environment after every scan to build the strong correlation of spatial relationship for the next scan to reference. A major purpose for this investigation is to demonstrate that these landmarks do not have to constantly increase the correlation between each scan in order to navigate

an environment by only having to reference the same landmark spatial relationships from a baseline scan.

Nonlinearity is an issue with EKF-SLAM because it uses linear models to describe non-linear vehicular motions and landmark observations. This discrepancy in linearity leads to inconsistent localization and mapping solutions (Julier and Uhlmann 2001). To combat this problem, the SLAM filtering technique known as FAST-SLAM was developed. FastSLAM is a SLAM solution discovered by Montemerlo et al. that introduced particle filtering to represent the nonlinear process model which EKF-SLAM relied heavily on (Montemerlo et al. 2002). This novel approach was influenced by previous literature proposed by Murphy (Murphy 2000) and Thrun (Thrun, Burgard, and Fox 2000). The issue with FastSLAM on its own is that it has a high-dimension state-space that requires immense computation power thus making it unappealing for autonomous navigation. However, by applying a Rao-Blackwellization (R-B) filter, the sample space is reduced to a more manageable size. Where EKF-SLAM allows autonomous navigation based on its current position, FastSLAM navigates based on its current trajectory. During FastSLAM navigation, the autonomous vehicle is scanning for landmarks and based on its current vector trajectory makes adjustments to the vehicles movement through the control input iterations as represented in Figure 7 where m is the stored mapping locations for the landmarks, z, and the control inputs u, are applied to the vehicles current position, x.

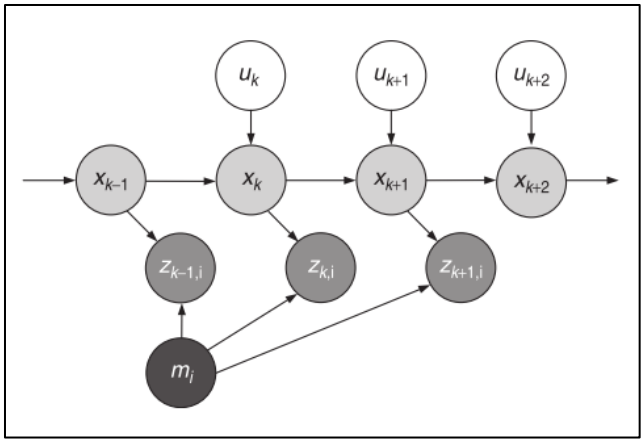

Figure 7: Representation for Navigation via SLAM Process

Two generations of FastSLAM have been generated and are labeled as version 1.0 and version 2.0. An issue with FastSLAM in both 1.0 and 2.0 versions is that it records all historical observation scans and does not forget previous scans which leads to degeneration. When resampling does deplete this history, map marginalization leads to inaccuracies (Bailey, Nieto, and Nebot 2006). This investigation will bypass degeneration by using preset global landmark positions (ego-points) as reference points that are not updated during navigation. Because no data for the ego-points' coordinates is ever deleted, map marginalization will also be avoided.

2.3 Light Detection and Ranging for Vehicular Localization

Intelligent vehicles have local sensors used specifically for localization in its surrounding environment. GPS fused with an inertial measurement unit (IMU) is generally used because roadways are already predefined in the world in terms of latitude and longitude so GPS can be utilized to localize the vehicle on these roadways. Localization from GPS is not precise enough for an intelligent vehicle to use on its own when navigating a lane. Research has shown that intelligent vehicles are able to localize itself in an environment by using object detecting sensors to establish its cartesian coordinates and pose in a global coordinate system. However, in order to perform this type of navigation, the autonomous vehicle requires an *a priori* type map (Wolcott and Eustice 2014). High accuracy navigation has been achieved via such object detecting sensors

in previous literature such as findings from Naranjo-Hernandez et al. that demonstrated localization accuracies as low as 0.5 meters when referencing roadside barriers (Naranjo-Hernandez et al. 2016).

Literature has shown that an intelligent vehicle was capable of using Sound Navigation and Ranging (SONAR) to detect the 'geometric beacon' and localize itself in the environment (Leonard and Durrant-Whyte 1991). The localization algorithm used in this literature is provided in Figure 8.

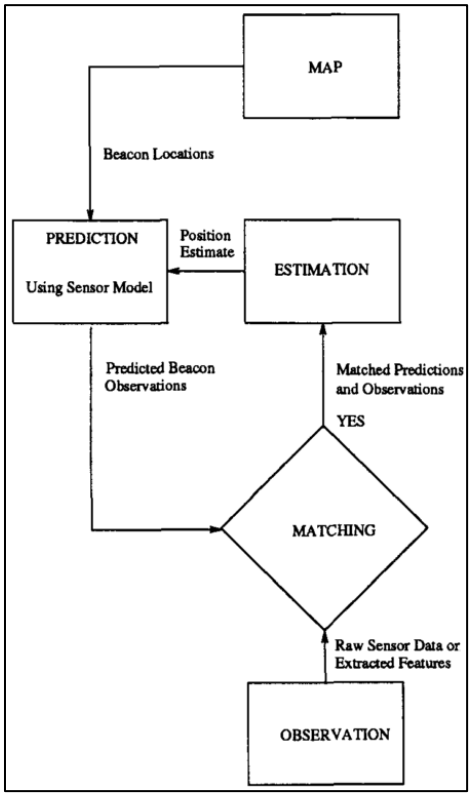

Figure 8: Localization Algorithm Matching Predefined Landmark Locations in Intelligent Vehicles (Leonard and Durrant-Whyte 1991)

Recent localization methods have utilized three-dimensional (3-D) LIDAR scanners such as the Velodyne HDL-32E to generate road intensity maps for navigational purposes (Levinson and Thrun 2010, Levinson, Montemerlo, and Thrun 2007, Wolcott and Eustice 2014, Hata and Wolf 2014). These 3-D LIDAR scanners are currently too expensive for practical use in

autonomous driving. Using 2-D LIDAR scanners would drastically reduce the cost during vehicle production so localization techniques using 2-D LIDAR scanners are currently being researched. The most common method of vehicular localization using 2-D LIDAR data is by referencing wall corners (Li and B Olson 2010, Chee and Yang 2013, Park et al. 2014, Weerakoon, Ishii, and Ali Forough Nassiraei 2015). These localization methods could be highly utilized for autonomous vehicles in urban areas by using building corners as landmarks. Recent literature from In et al. has converted 3-D point cloud mappings of urban areas to 2-D bird eye view maps to detect building corners in urban areas to acts as navigational landmarks (Jun-Hyuck, Sung-Hyuck, and Gyu-In 2016). Localization from overhead maps have also been used as a reference environment for 2-D scans in urban environments as well (Levinson, Montemerlo, and Thrun 2007). This investigation will utilize a 2-D LIDAR scanner to detect landmarks, label them as ego-points in a point cloud environment, and use them for autonomous navigation to demonstrate accurate navigation via a 2-D LIDAR scanner is feasible.

Too much memory would have to be allocated in an autonomous vehicles ECU in order to store landmarks for a large amount of area. Ideally, a vehicle would only need to reference a small portion of a virtual environment that contains landmarks in its vicinity to use as its a *priori* environment. This is feasible by pulling certain portions of the *a priori* map based on GPS location. This strategy was performed in previous literature where the environment had various maps correlated with corresponding GPS coordinates. The vehicle was then able localize itself by referencing it's GPS position and uploading the respective maps (Ranganathan, Ilstrup, and Wu 2013). This investigation will generate its own small scale *a priori* map that would contain the minimal amount of landmarks in order to perform autonomous navigation.

360° LIDAR scanners on intelligent vehicles can either be 2-D or 3-D. For 2-D LIDAR scanners, the LIDAR outputs a string of distances every rotation that correspond to all the individual laser points at all the angles from one scan. Plotting out the x and y cartesian coordinates for each LIDAR data point in 2-D scanners, such as the Slamtec RPLIDAR used in this investigation, are calculated using Equations 1 & 2 where $X_L(i)$ & $Y_L(i)$ are the x and y coordinates in a point cloud system for iteration, i, and $D(i)$ is the distance for that iteration (Dawood et al. 2016).

$$X_L(i) = D(i) \times cos\big(angle(i)\big) \tag{1}$$

$$Y_L(i) = D(i) \times sin\big(angle(i)\big) \tag{2}$$

2.4 Urban City Navigation

2.4.1 Common Issues with Urban Environment Complexity

Current navigation methods in intelligent vehicles rely heavily on GPS localization fused with IMU systems to get centimeter accuracy when localizing itself within its environment (Levinson, Montemerlo, and Thrun 2007). However, GPS localization has issues in urban environments due to being cut off from sufficient satellite signals from tall buildings and tunnels typically found in urban cities (Dawood et al. 2016). To deal with short term GPS signal losses in intelligent vehicles, incremental encoders and IMU sensors are fused together to track vehicle via dead-reckoning (DR); This however does not work for long term GPS signal connection loss due to drifting error accumulation via DR (Se, Lowe, and Little 2002, Dawood et al. 2011). Because of this issue, localization using object detecting sensors is the most practical method for navigating urban areas (Veronese et al. 2016, Wolcott and Eustice 2014). Researchers look to integrating SLAM technology to work alongside its GPS navigation systems in order to be able to navigate the complex urban environment when GPS signal acquisition fails. SLAM is able to localize itself

within a static environment however, urban environments have a mix of dynamic and static environmental landmarks. In order for SLAM to be implemented into autonomous navigation in vehicles, the vehicle must be able to distinguish dynamic variable (i.e. cars driving by) from static variables (i.e. buildings and curbs) (Levinson, Montemerlo, and Thrun 2007). Various SLAM techniques have worked on tracking dynamic objects such as cars in a static environment in order for vehicles to localize itself (Guivant, Nebot, and Baiker 2000, Hahnel, Schulz, and Burgard 2002, Chieh-Chih, Thorpe, and Thrun 2003).

The environment that SLAM is performed in can become more distinct for more accurate localization by performing offline environment scans over long periods of times (Levinson and Thrun 2010). The landmarks used in SLAM will have their distance correlation strengthened each scan as vehicles move through the environment for more accurate localization positioning (Durrant-Whyte and Bailey 2006). The high flow of traffic in urban areas allow the opportunity for generating a virtual point cloud environment for intelligent vehicles to navigate within. The landmark correlations will get strengthened for every intelligent vehicle that scans the environment.

2.4.2 Localization within an Urban Environment.

SLAM technology is useful in situations where the vehicle is in an unknown environment and must map out the environment in order to navigate it. However for modern day intelligent vehicles on the road, the vehicles no longer have to deal with unknown environment situations, especially in urban cities. Urban environments can be generated in a 3-D virtual model environment that can be used as a reference for vehicular localization. Literature has shown methods of creating these environments in real-time to rapidly expand 3-D virtual environment building (Kim et al. 2005, Takase et al. 2004, Takase et al. 2003).

Previous research has shown that an intelligent vehicle was able to navigate autonomously by overlaying its laser-scanned surroundings with a 3-D geographical information system (GIS) virtual environment (Dawood et al. 2016, Peng et al. 2009). Algorithms such as the Iterative Closest Point (ICP) developed by Zhang (Zhang 1994) have been able to take separate scan data from same position and determine rotation and translation transformations of these scans to have match up. These translation and rotation transformations can then be correlated to a vehicle's pose in the environment. Similar translation and rotation techniques can be applied to intelligent vehicles to determine its pose and location as demonstrated in this investigation.

Ideally, intelligent vehicles would navigate urban areas with similar arcs and speeds to that of a human driver for maximum ergonomically planned navigation routes. This can be achieved by plotting out cartesian coordinates in the *a priori* SLAM generated environment while a human driver is navigating inside it. These cartesian coordinates can act as reference points to navigate the same path as a human driver. This recording of vehicular position in an *a priori* environment using a 2-D LIDAR scanner is demonstrated in this investigation.

CHAPTER 3. METHODS

3.1 Methodology Overview

This study was conducted to analyze how well an intelligent vehicle could perform when only relying on landmark derived ego-points for navigation referencing. To perform this type of navigation, the individual tasks for the intelligent vehicle in this study were separated into three sections: zeroing the environment, creating baseline path, and autonomous ego-localization navigation.

The first task for the intelligent vehicle was to create the ego-localization environment and zero itself within that environment. First, the vehicle was placed inside the driving test area and it then proceeded to scan its environment using a 2-D LIDAR scanner to detect the wooden planks that act as landmarks for the intelligent vehicle. It then created a cartesian coordinate system with its current position at the origin and placed the x and y positions of each wooden plank it detected in the cartesian coordinate system. These x-y coordinates acted as ego-points inside the global point cloud system. Figure 9 illustrates the first task of zeroing the environment and labeling the landmark locations as ego-points in the *a priori* map.

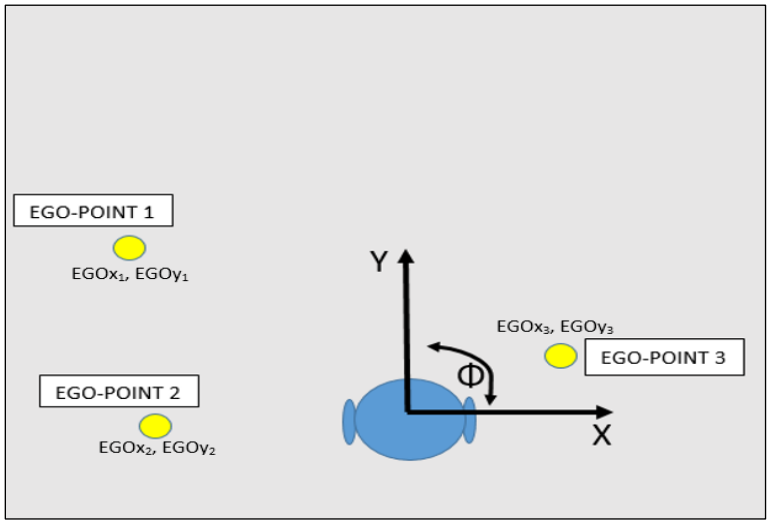

Figure 9: Zeroing the Environment and Building the *a priori* Map

The second task created a baseline path that the intelligent vehicle would have to follow autonomously. This was accomplished by hardcoding the wheel speeds of the intelligent vehicle for various speeds and durations to achieve a complex path. While the intelligent vehicle navigated inside the testing area, it used the LIDAR data to triangulate its position and current vehicular pose within the previously established cartesian coordinate system. The intelligent vehicle was able to record its path and vehicular pose by detecting the wooden planks and overlaying their positions to those in the *a priori* map generated from the first task. Once the intelligent vehicle determined its position and pose within the *a priori* map, it recorded its position and pose for later autonomous navigation trials. A visual representation of the baseline path recordings if provided in Figure 10. This positional recording of manual navigation was performed at a predetermined rate 2 Hz in this investigation.

Autonomous navigation was achieved in the third task by having the intelligent vehicle navigate to the recorded baseline run position points from the second task. This was possible by performing the same triangulation method in finding its current position & direction and adjusting the vehicles wheel speeds accordingly to get to the next required baseline point. The intelligent vehicle did this for all the recorded positions in the same order that they were recorded in to mimic autonomous path following. The test was declared over once the intelligent vehicle navigated to all the recorded positions from the baseline run. This autonomous navigation method is visualized in Figure 11.

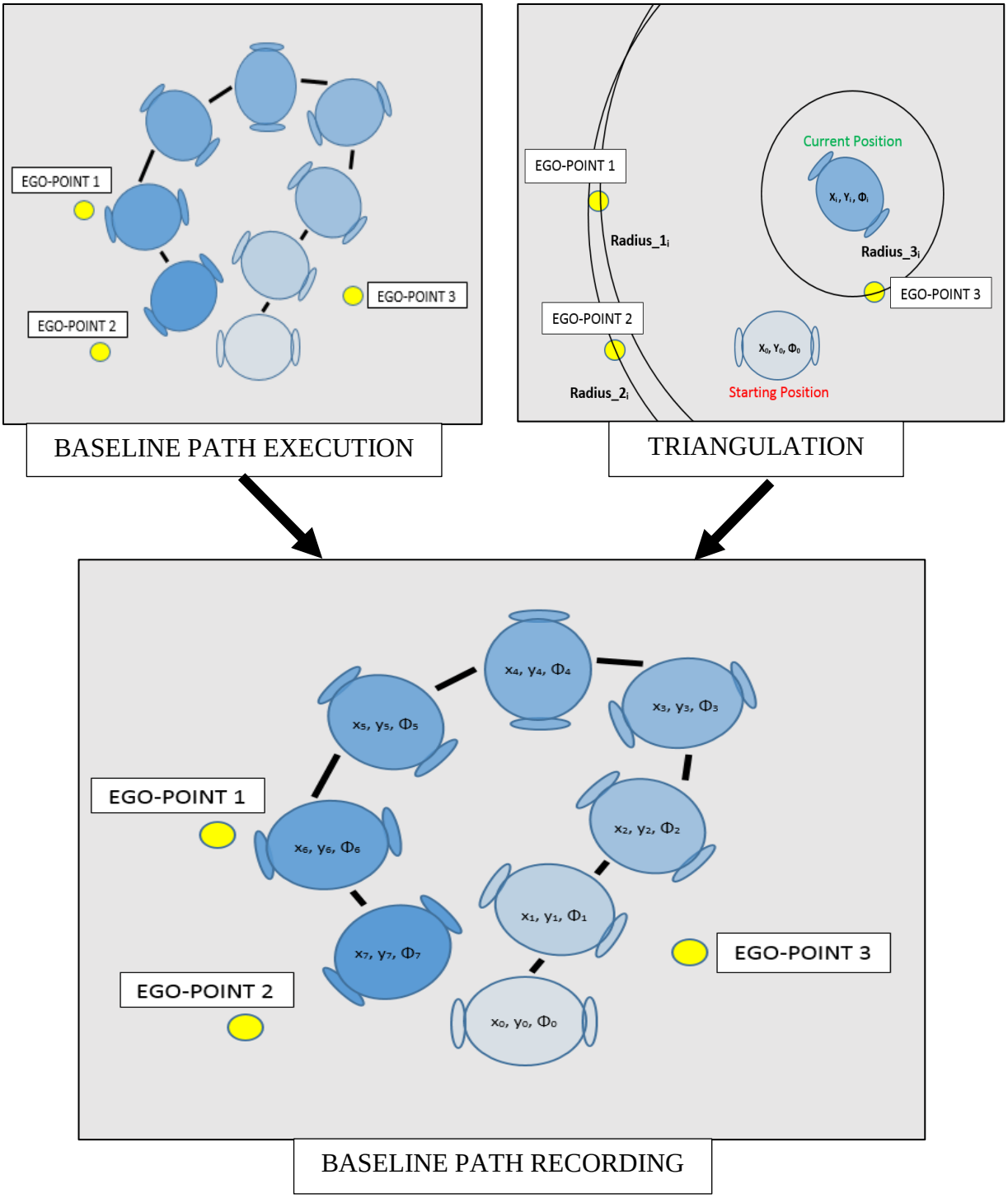

Figure 10: Recording the Baseline Path for Autonomous Navigation

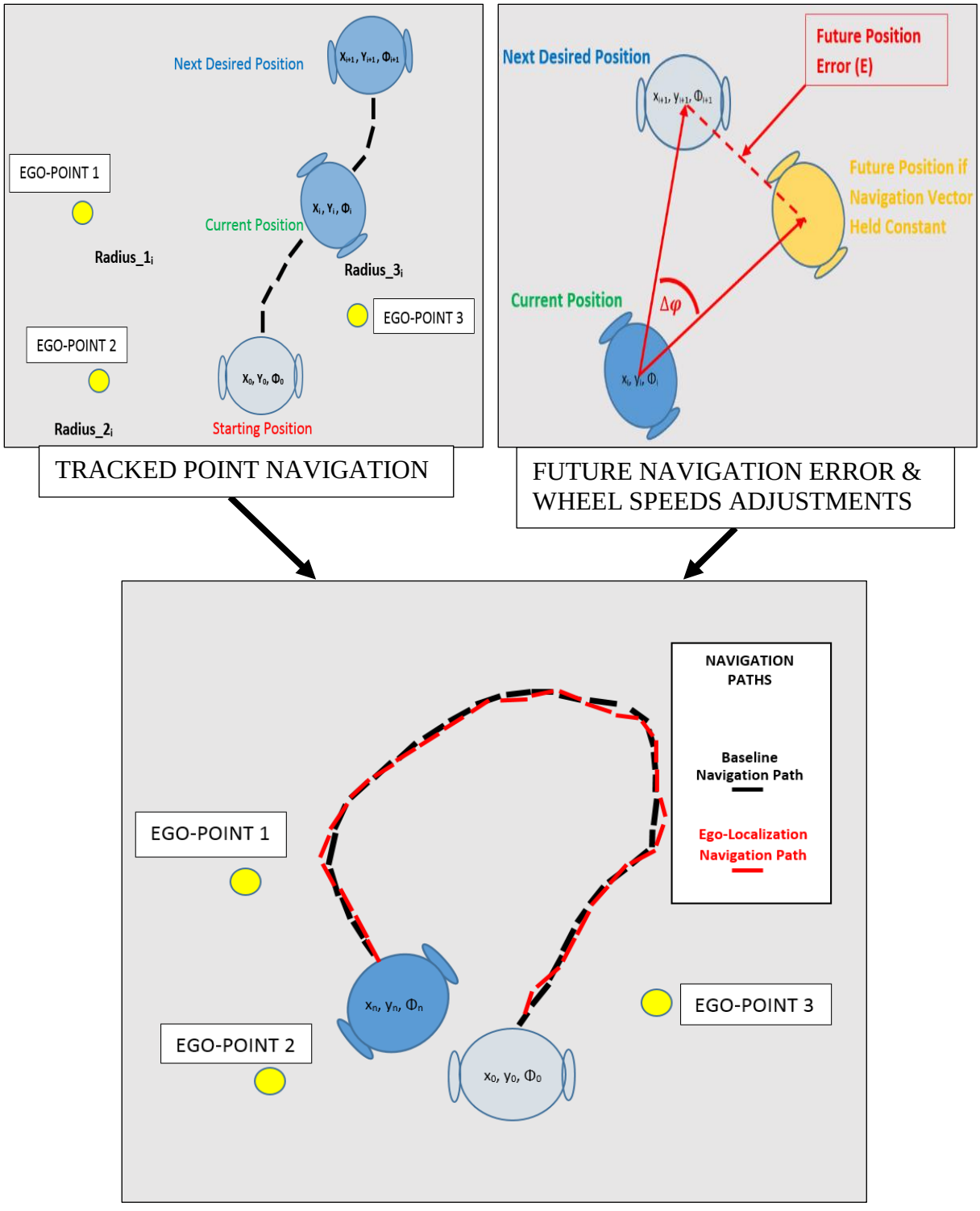

Figure 11: Ego-Localization Autonomous Navigation

3.2 Intelligent Vehicle Design

The small scale intelligent vehicle used in this investigation, displayed in Figure 12, was the Parallax Arlobot fitted with a Slamtec RPLIDAR, Arduino microcontroller, and an IBM ThinkPad laptop. The RPLIDAR is a 2-D 360° LIDAR scanner that has a maximum range of 8 meters and an angular resolution of 0.9 degrees. The Arduino microcontroller was utilized to control the Parallax motor drivers that controlled the vehicles wheel speeds. The laptop acted as the vehicles ECU to store autonomous navigation algorithms and extract sensor data from the LIDAR scanner. Mounts for the RPLIDAR were 3-D printed and designed to ensure that nothing on the intelligent vehicle would obstructs the robots field of view.

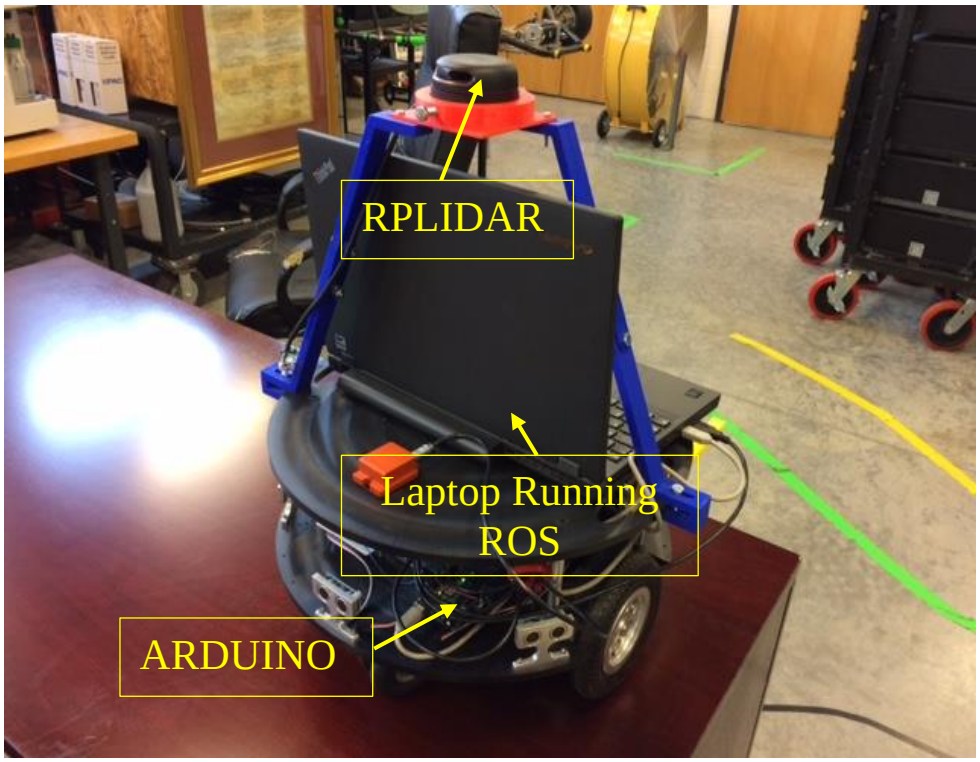

Figure 12: The Intelligent Vehicle used in this Investigation

3.3 Environment Design

One of the goals in this study was to provide the intelligent vehicle the ability to develop its own cartesian coordinate system for navigation. It also needed to be able to scan its environment

to plot out ego-points in the self-developed environment while driving a baseline path; therefore, the wooden stakes used to create the ego-points were not placed in pre-determined locations. The x and y positions of the wooden stakes in the vehicles generated *a priori* map were measured during the environment zeroing run. The stakes were placed in a manner to be sure that the robot will be able to see all three of them from any point within the drivable environment, i.e. at no point was there a time when two stakes lined up from the intelligent vehicles viewpoint.

The intelligent vehicle's scanning radius for the RPLIDAR was set to 1.7 meters; any LIDAR data points beyond this distance were discarded. To makes sure that the robot would not detect a wall of the room, the navigable area was outlined being 2.0 meters from any object or wall within the testing room.

3.4 Robot Operating System Design

Robot Operating System (ROS) is an operating system ideal for intelligent vehicles and it was used in this investigation. ROS operates by using callback functions which perform a task using data that is published to a *topic*. Data from sensors are published to these topics via *nodes* which are ROS based executable programming files. The callback functions will run for every *message* filled with data that is published to the topic and, once it finishes it will perform the callback function again for the next message that was published to the topic. This allows for data to be in queue in the topic if data is coming in faster than the callback function can handle. If there is no more data in the topic, the node running the callback functions will wait until data is published to the topic again.

3.4.1 Initializing the Environment

3.4.1.1 Environment Initialization Overview

In order for the autonomous vehicle to develop a baseline path, it first needed to develop a global cartesian based *a priori map* in which to navigate. During the zeroing run, the autonomous vehicle created the global cartesian coordinate system and placed its position at the origin facing at an angle of $\frac{\pi}{2}$ radians. The RPLIDAR on the intelligent vehicle then scanned its surroundings and detected the wooden stakes that act as ego-localization points. The RPLIDAR returned 360 data points with equal angular spacing between each data point every rotation. The x and y coordinates for each data point in the global cartesian coordinate system were calculated using Equations 1 & 2. Clusters of these data points were grouped together and their x and y values were averaged to get the x and y positions of each ego point. These coordinates were then placed within the self-generated *a priori* map; this process is visualized in Figure 13. The cartesian coordinate locations for each ego-point was then recorded in ROS by creating a .bag file. These stored ego-point coordinates were to be used later during baseline path trials and autonomous navigation trials. ROS nodes are programs that can run simultaneously in ROS and they can all be started using an ROS launch file. The launch file for the environment zeroing run and the nodes it starts are provided in Figure 14, where the red squares are hardware used, green squares and green arrows are objects created and yellow squares and arrows are loaded objects. The cloud objects are ROS topics where data is published and/or subscribed to.

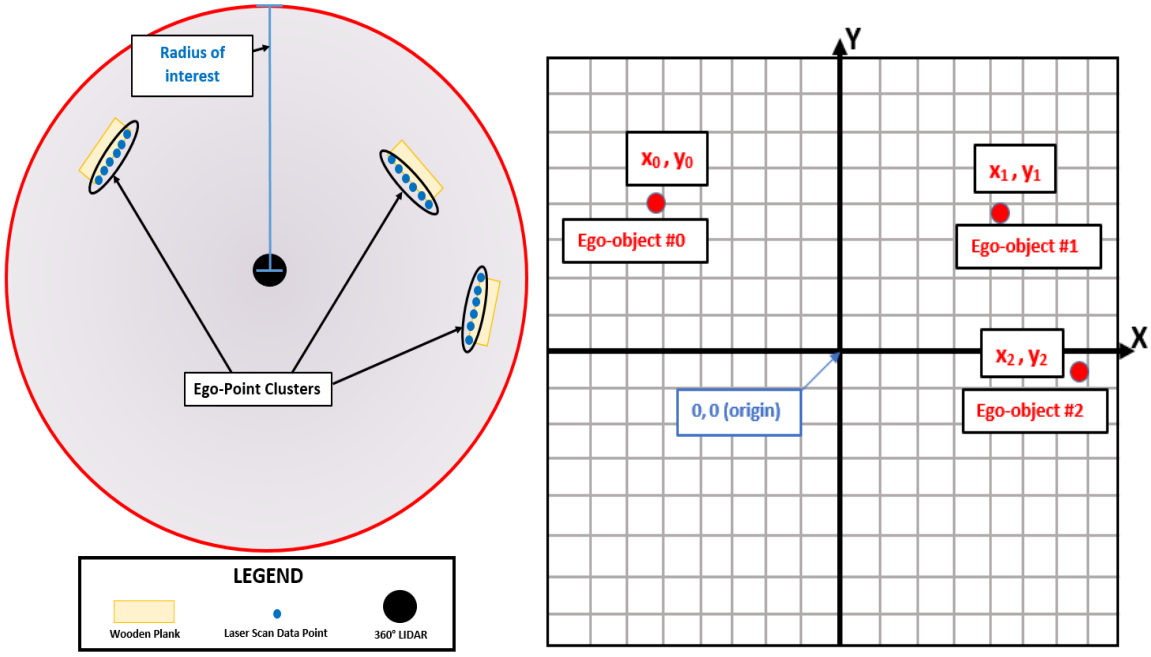

Figure 13: Grouping Landmarks and Plotting Their Ego-Point Locations during Environment Initialization Run

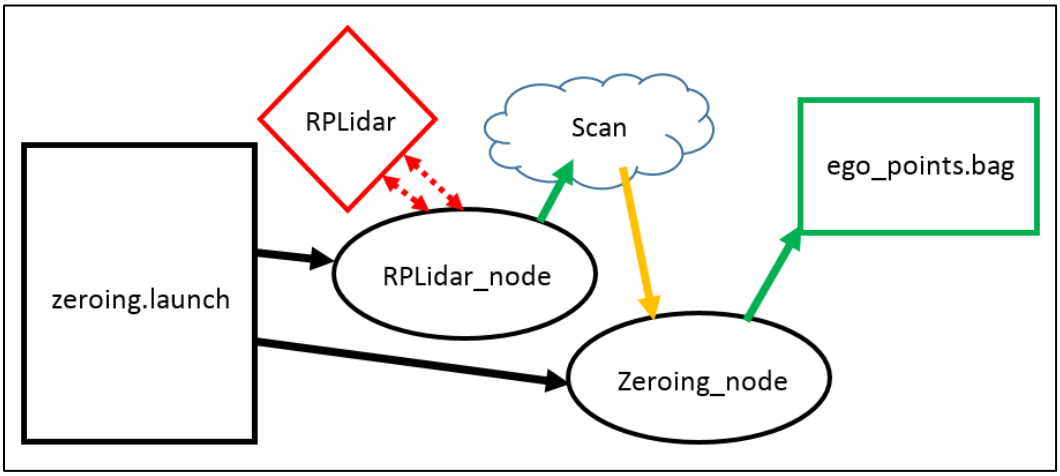

Figure 14: ROS Nodes, Topic, and Bag File Used/Created During Environment Zeroing Run

3.4.1.2 RPLidar_node

The *RPLidar_node* started the motor inside the RPLIDAR needed for 360° scanning. It also collected data and from each revolution of the RPLIDAR and published it to the topic *Scan*. The message type it used, *sensor_msgs/LaserScan.msg*, is provided in Figure 15. The only

information in this message type needed for this investigation was the minimum angle (angle_min), angle increment (angle_increment), and ranges array (ranges).

```
Header header            # timestamp in the header is the acquisition time of
                         # the first ray in the scan.
                         #
                         # in frame frame_id, angles are measured around
                         # the positive Z axis (counterclockwise, if Z is up)
                         # with zero angle being forward along the x axis

float32 angle_min        # start angle of the scan [rad]
float32 angle_max        # end angle of the scan [rad]
float32 angle_increment  # angular distance between measurements [rad]

float32 time_increment   # time between measurements [seconds] - if your scanner
                         # is moving, this will be used in interpolating position
                         # of 3d points
float32 scan_time        # time between scans [seconds]

float32 range_min        # minimum range value [m]
float32 range_max        # maximum range value [m]

float32[] ranges         # range data [m] (Note: values < range_min or > range_max should be discarded)
float32[] intensities    # intensity data [device-specific units].  If your
                         # device does not provide intensities, please leave
                         # the array empty.
```

Figure 15: sensor_msgs/LaserScan Message Type

For the RPLIDAR, the minimum angle was always $-\pi$ radians and it increased incrementally based on the angle increment value obtained from the scan (this value varied slightly due to the variance in the sampling rate from the RPLIDAR). The last angle with an attaching range value stopped just short of π radians due to $-\pi$ and π being the same angle. Each angle and the corresponding range value was determined for point cloud mapping using Equations 3 & 4 where i was the iteration value starting from 0 and ended when it matched the size of the ranges vector.

$$angle_i = angle\_\min + (i \times angle\_increment) \tag{3}$$

$$range_i = ranges[i] \tag{4}$$

3.4.1.3 Zeroing_node

This node initialized the *a priori* virtual environment in which autonomous navigation was later performed in. The node set the origin (0, 0) to the location where the autonomous vehicle was

at the time when the node was started. The scanned data from the *RPLidar_node* was then analyzed to group the range values at their respective angles when they indicated a detected object within a preset radius as shown in Figure 13; for this investigation, the preset radius was 1.7 meters. If a point was within 1.7 meters it was then to be sorted into detected landmark group. If the point was within a preset distance (grouping constant) of an already detected landmark, it was grouped with other points affiliated with that landmark. Otherwise a new landmark was created and the point was placed in that landmarks group. The grouping constant used in this investigation was 15 cm. This was performed for 3 consecutive scans to get sufficient values for averaging. Once the scans were finished, the x and y points for each point in each landmark group was averaged together to get an x and y position for the ego-point that represented its respective landmark. These ego-points were used for localization purposes during baseline run trials and autonomous navigation trials performed later in the investigation. These x and y coordinates for the ego-points in the *a priori* environment were stored in an ROS bag file called *ego_points.bag*.

3.4.2 Creating the Baseline Path

3.4.2.1 Baseline Trials Overview

A baseline path was created that the intelligent vehicle had to autonomously navigate in order to confirm that autonomous navigation via ego-point localization was achieved. This was performed by recording the intelligent vehicles position in the virtual *a priori* map while was navigating the baseline path. The baseline path that the intelligent vehicle navigated was created by controlling the wheel speeds and the durations at those speeds for the entire baseline run. This allowed implementation of different turn radii and number of straightaways in the baseline path. The baseline path was insured to be smooth tangential turns in order to mimic human driving prowess. Various ROS nodes were launched using the *drive_track.launch* launch file in order to

navigate and record the baseline path. The various nodes that started from *drive_track.launch* and the topics they were publishing/subscribing to are provided in Figure 16 along with the .bag files they loaded/created.

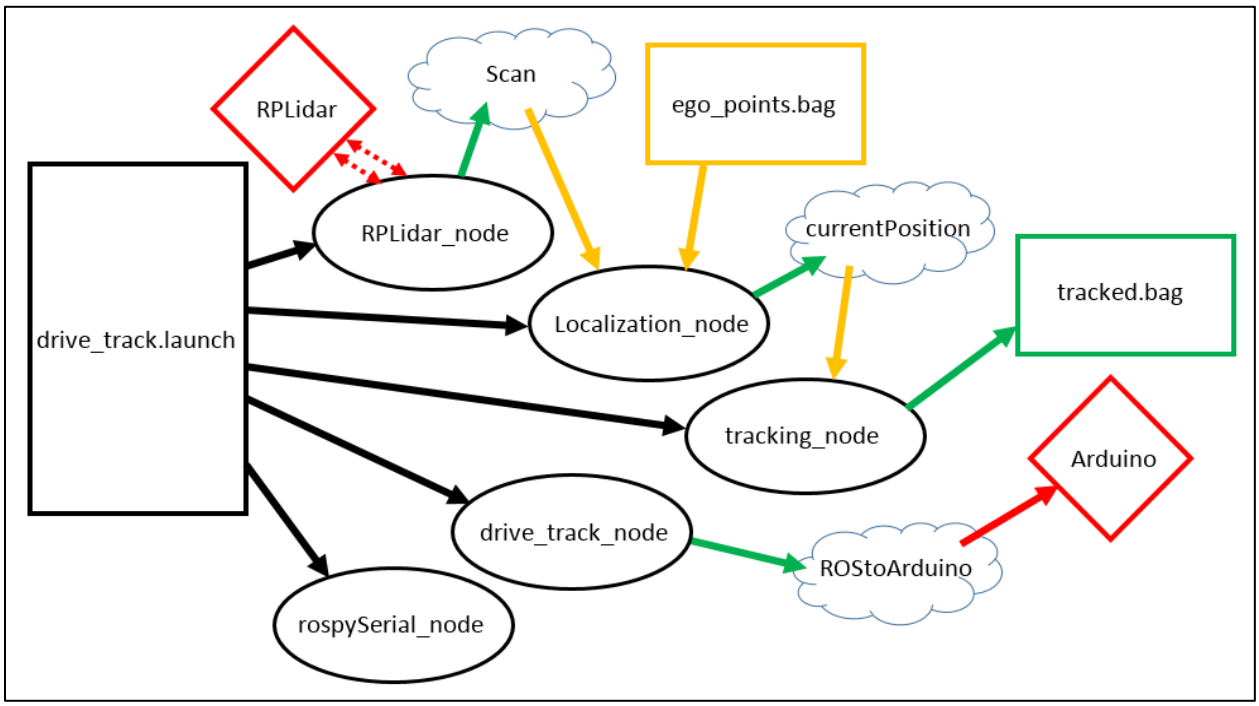

Figure 16: Flow of Information during Baseline Navigation Run

3.4.2.2 Localization_node

The vehicles current location in the *a priori* environment was constantly being updated using the *Localization_node*. This node loaded the cartesian coordinates for each ego-point from the *ego_points*.bag file and was subscribed to the *Scan* topic to work with the current LIDAR data. The *Localization_node* grouped and averaged LIDAR data points for detected landmarks using the same method from the *zeroing_node* to get cartesian coordinates for each ego-point from its current frame of reference. These local ego-points were then labeled with a number (either 0, 1, or 2) as performed in the zeroing node based on the order of when the ego-points were created during the grouping of data points portion of the node.

However, these local ego-point labels depended on the current pose of the autonomous vehicle in the *a priori* environment. In order confirm if the local ego-point labels matched those of the global ego-points from the environment zeroing run, the distances between the currently detected landmarks were compared with those from the *a priori* map using Equation 5; where J and K are the landmark identifying labels for the two landmarks having their distance measured, and their respective x and y coordinates are represented with ($\mathbf{x}_J$,$\mathbf{y}_J$) & ($\mathbf{x}_K$,$\mathbf{y}_K$).

$$\overleftrightarrow{J\_K} = \sqrt{\left(x_K - x_J\right)^2 + \left(y_K - y_J\right)^2} \tag{5}$$

If the distances matched up, within a preset tolerance of 10 cm, the landmark labels were confirmed to be correct. If the distances didn't match up, then the labels were rotated for each local ego-point and their distances were compared again. This labeling rotation was performed until the local ego-points were correctly identified. The rotation algorithm used in this investigation is displayed in Figure 17. Rotating and translating the currently detected landmark locations so that they overlay with their respective *a priori* landmark locations allowed the vehicles current pose and coordinates to be detected. Before the pose was determined the vehicle's current position coordinates had to be determine through triangulating its position from the landmarks.

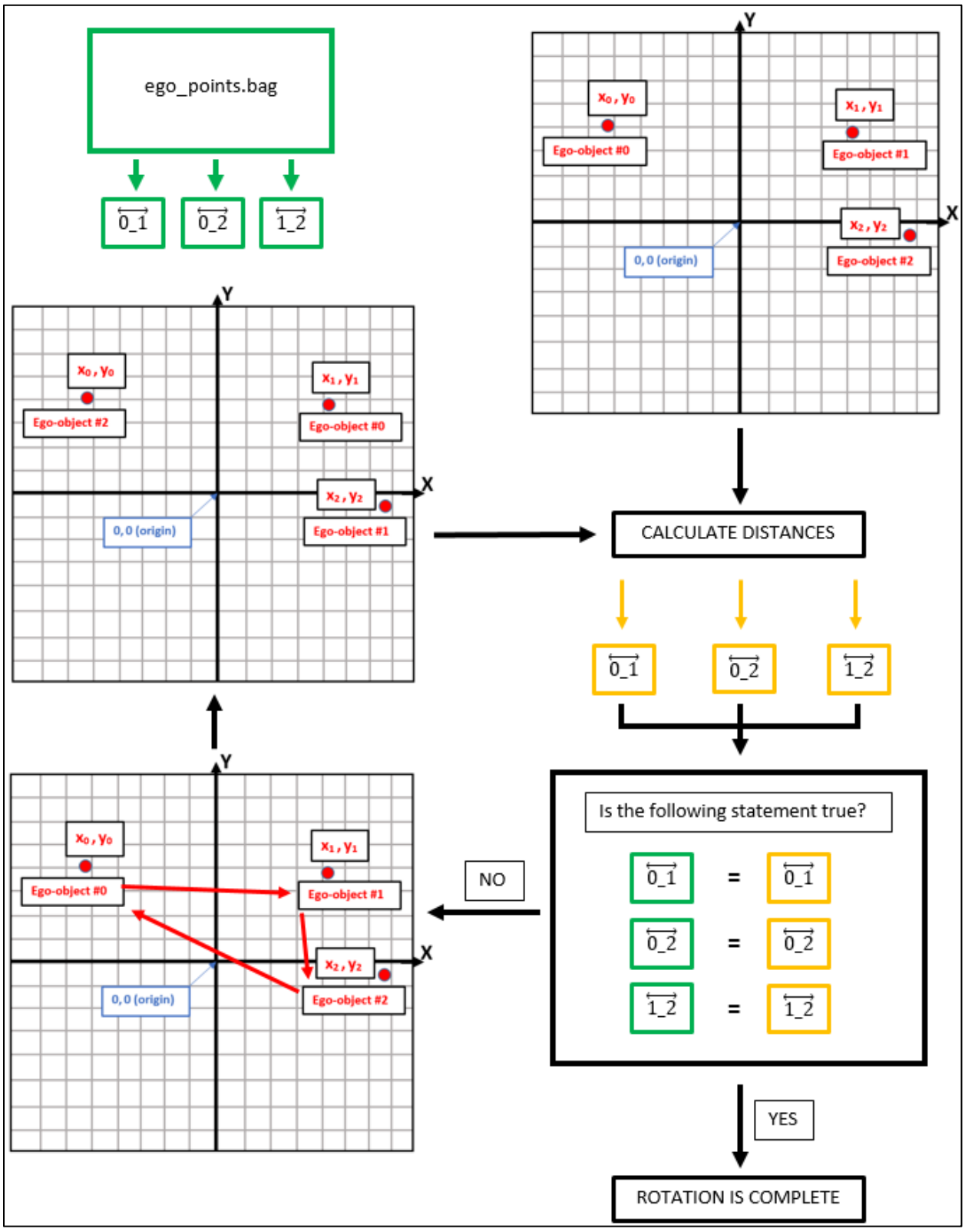

Figure 17: Rotation Algorithm for Ego-point Labeling

3.4.2.3 Localization via Triangulation

The intelligent vehicle localized itself in the *a* priori environment by triangulating its position using the local ego-points' distances and the global ego-points' coordinates. The x and y positions for each ego-point in the *a priori* map were denoted using a double bar above the x and y while the ego-points' local x and y positions were denoted with a single bar. For example, $\overline{\overline{x_0}}$ was the x position of ego-point #0 in the *a priori* map and $\overline{y_2}$ was the y position for ego-point #2 from the recently scanned data. To triangulate its position, the intelligent vehicle determined the distance, *d*, between ego-point #0 and ego-point #1. This was accomplished by calculating the delta x, *dx*, and delta y, *dy*, in the *a priori* map between the two ego-points using Equations 6 & 7; from there the distance between them, *d*, was calculated using Pythagorean theorem displayed in Equation 8.

$$dx = \overline{\overline{x_1}} - \overline{\overline{x_0}} \tag{6}$$

$$dy = \overline{\overline{y_1}} - \overline{\overline{y_0}} \tag{7}$$

$$d = \sqrt{(dx)^2 + (dy)^2} \tag{8}$$

The x and y coordinates for the midway-point (MP) between ego-points #0 and #1 needed to be determined next in the triangulation calculation. This MP is at the cross section of two lines: One line is going from ego-point #0 to ego-point #1 and the other line is going from the two intersecting points of the circles formed from the vehicles radii from ego-point #0 , $\overline{r_0}$, and from ego-point #1, $\overline{r_1}$. The distance from ego-point #0 to MP was also needed and was denoted as *A*. These variables and what they represented are presented in Figure 18 and their respective equations are provided in Equations 9 - 11.

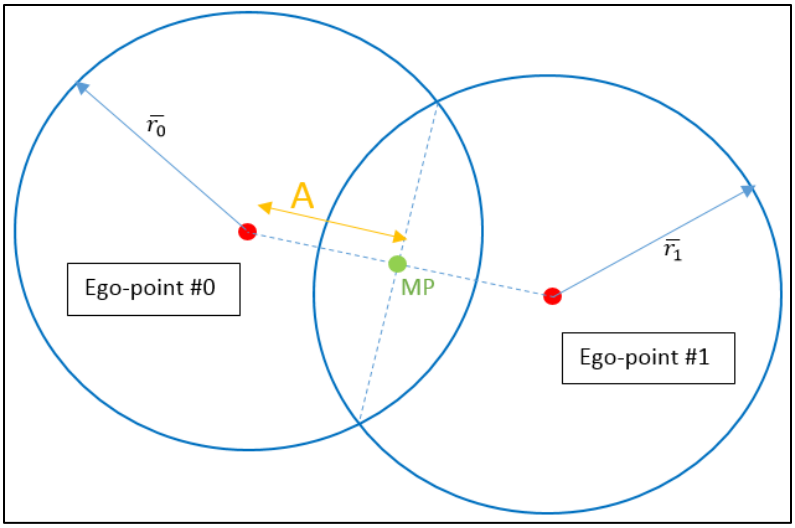

Figure 18: Calculated Variables between Ego-points #0 and #1 That Were Used during Triangulation

$$A = \frac{((\bar{r_0})^2 - (\bar{r_1})^2 + (d)^2)}{2 \times d} \tag{9}$$

$$MP_x = \overline{\overline{x_0}} + \left(\frac{dx \times A}{d}\right) \tag{10}$$

$$MP_y = \overline{\overline{y_0}} + \left(\frac{dy \times A}{d}\right) \tag{11}$$

The distance between MP and either one of the circle intersecting points, denoted as *h*, was calculated using Equation 12. Then the x and y offsets from the circle intersecting points, represented as r_x & r_y respectively, were calculated using Equations 13 & 14. A visual representation of these offsets is provided in Figure 19.

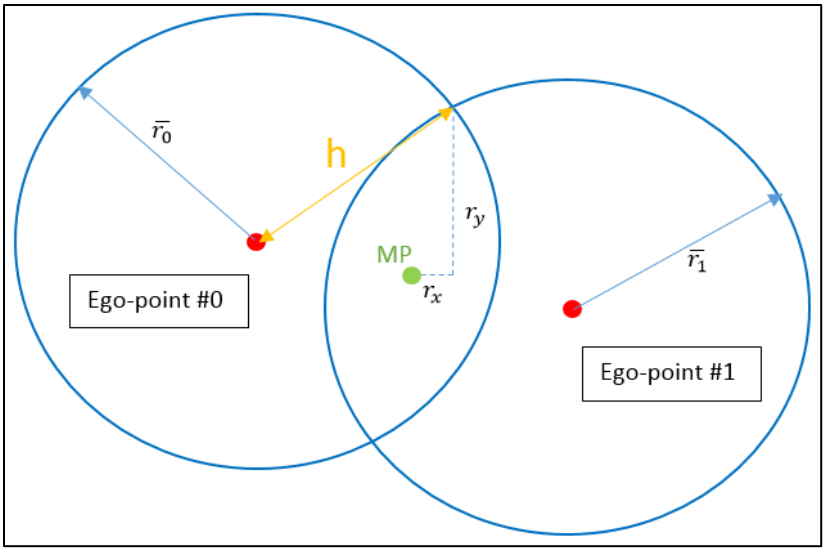

Figure 19: MP Offsets from Landmark Radii Intersecting Points

$$h = \sqrt{(\bar{r_0})^2 - (A)^2} \tag{12}$$

$$r_x = -dy \times \frac{h}{d} \tag{13}$$

$$r_y = dx \times \frac{h}{d} \tag{14}$$

At this point, the intelligent vehicles location was at one of the two intersecting points so the next step was to determine which of the two points it was at. The intersecting point it was at was found by testing both points to see if which one of their distances from ego-point #2 matched the radius to ego-point #2, $\bar{r_2}$, that the intelligent vehicle was reading during that scan. The intersecting points, denoted as *IP_1* and *IP_2*, had their cartesian coordinates calculated using Equations 15 - 18. Their respective distances to ego-point #2, *d1* and *d2*, were then calculated using the Pythagorean theorem equation as shown in Equations 19 & 20.

$$IP\_1_x = MP_x + r_x \tag{15}$$

$$IP\_1_y = MP_y + r_y \tag{16}$$

$$IP\_2_x = MP_x - r_x \tag{17}$$

$$IP\_2_y = MP_y - r_y \tag{18}$$

$$d1 = \sqrt{(IP\_1_x - \overline{\overline{x_2}})^2 + \left(IP\_1_y - \overline{\overline{y_2}}\right)^2} \tag{19}$$

$$d2 = \sqrt{(IP\_2_x - \overline{\overline{x_2}})^2 + \left(IP\_2_y - \overline{\overline{y_2}}\right)^2} \tag{20}$$

The x and y cartesian coordinates for the intelligent vehicle in the *a priori* map were then determined based off an if-then statement algorithm that set the vehicles x and y position, denoted as V_x and V_y, as the x and y coordinates for *IP_1* or *IP_2* depending on whether *d1* or *d2* was closer to $\overline{r_2}$. A tolerance of 10 cm was applied to the if-then statements in order to reject invalid data during testing. This if-then statement algorithm is presented in Figure 20.

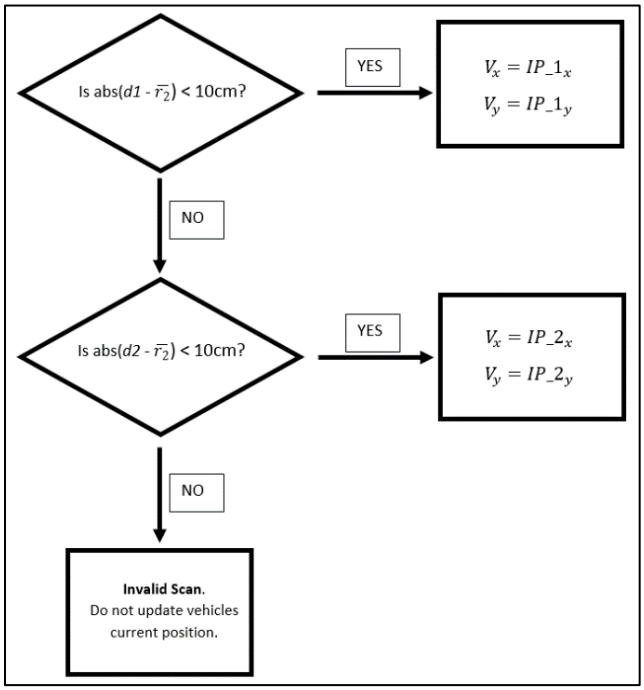

Figure 20: If-Then Statement for Determining the Vehicles Coordinates during Triangulation

The pose of the intelligent vehicle in the *a priori* map was determined once the coordinates for the intelligent vehicle were determined through triangulation. In the *a priori* map, the y-axis was in the pose direction of $\frac{\pi}{2}$ radians which is the direction the vehicle was facing when the

initialization run was performed. The vehicles pose was determined by analyzing the angles that the intelligent vehicle would be detecting each landmark at if it was facing in the y-axis direction based on cartesian coordinates of the ego-points and the vehicle. These angles were compared against the respective angles that the intelligent vehicle was actually detecting the ego-points at. The difference between the angles for each landmark was applied to the angle of $\frac{\pi}{2}$ radians to get the actual pose of the intelligent vehicle in the virtual environment.

The RPLIDAR provided data starting at a minimum angle of -π radians that increased incrementally up to $\pi - \alpha$ going in the counter-clockwise direction where α was the angle increment. This angle increment varied for each scanning revolution depending on the collection rate and revolutions per minute (RPM) of the RPLIDAR. The minimum angle was always opposite of the data communication wiring on the RPLIDAR as shown in Figure 21.

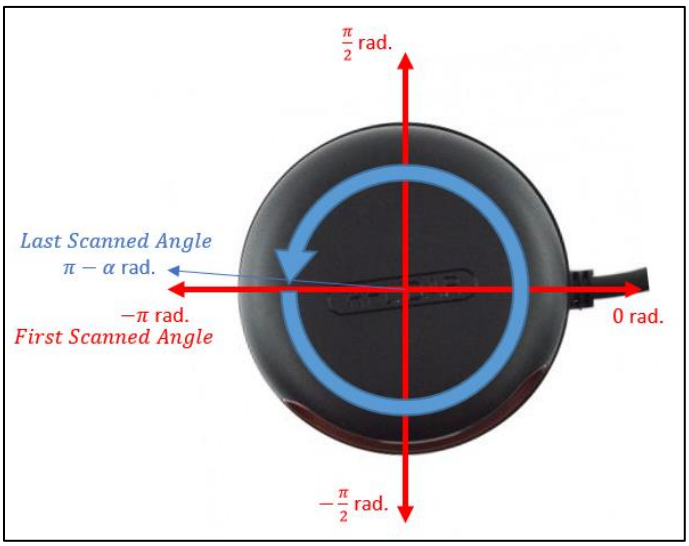

Figure 21: Scanning Angles in Relation to RPLIDAR Hardware

Due to the major jump in scanning angles on the axis separating the 2nd & 3rd quadrants, special considerations in the algorithm were considered when performing computations across the negative X-axis. The angle in the *a priori* map where the ego-point was in relation to the vehicles

cartesian coordinates was denoted as β_k, where k was the ego-point label (0, 1, or 2). A visual representation for determining β_k, using ego-point #0 as an example, is presented in Figure 22.

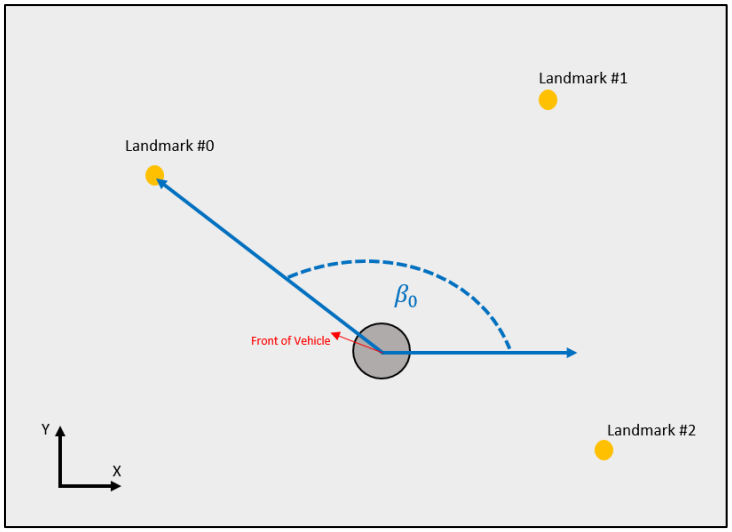

Figure 22: Angle Representation for β

The RPLIDAR was mounted on top of the intelligent vehicle and orientated in such a manner to ensure that any object directly in front of the vehicle was detected at an angle of $\frac{\pi}{2}$ radians. The angle at which an ego-point was detected according to the data from the RPLIDAR was denoted as θ_k where k indicated the ego-points label that was used by β for the same ego-point. Because the front of the vehicle was always $\frac{\pi}{2}$ radians, the x-axis for the RPLIDAR data was to the right of the vehicle; so θ_k was the angle between the axis extending from the right side of the vehicle and the vector toward ego-point #k. This is visually represented in Figure 23.

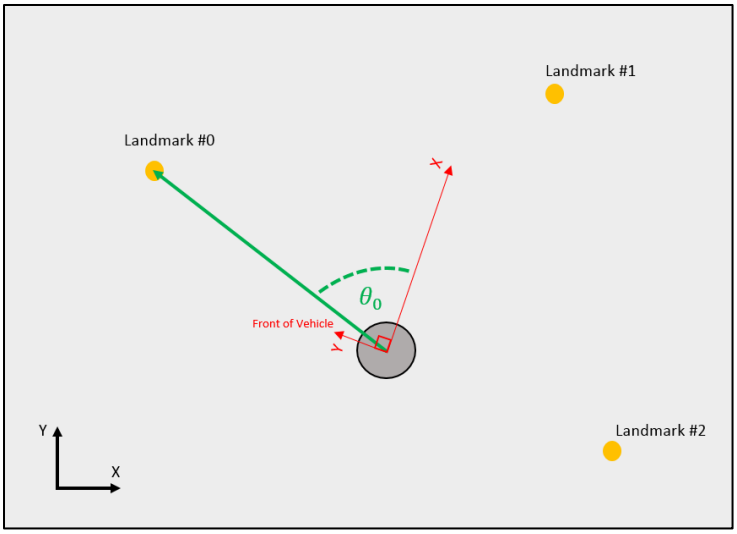

Figure 23: Angle Representation for θ

The calculation for β for each of the ego-points was independent from the intelligent vehicles pose because it only relied on the vehicles cartesian coordinates within the *a priori* map. Trigonometric calculations using inverse tangent only gave an angle in either the 1st or 4th quadrant; therefore if the ego-point was in the 2nd quadrant, the calculated β was increased by π (Equation 21) and if it fell within the 3rd quadrant then β was decreased by π (Equation 22). If the ego-point was within the 1st or 4th quadrant then β was not adjusted and Equation 23 was used.

$$\beta_k = \tan^{-1}\left(\frac{\overline{\overline{y_k}} - V_y}{\overline{\overline{x_k}} - V_x}\right) + \pi \tag{21}$$

$$\beta_k = \tan^{-1}\left(\frac{\overline{\overline{y_k}} - V_y}{\overline{\overline{x_k}} - V_x}\right) - \pi \tag{22}$$

$$\beta_k = \tan^{-1}\left(\frac{\overline{\overline{y_k}} - V_y}{\overline{\overline{x_k}} - V_x}\right) \tag{23}$$

The vehicles pose according to each ego-point was represented as φ_k where *k* corresponded to the ego-point that it was referencing its angular pose with. Different equations were required for calculating each φ_k depending on certain scenarios. The equation used to calculate each φ_k depended which quadrant in the *a priori* map the intelligent vehicle was in and which quadrant β_k

fell in. It also depended on the difference between angles β_k and θ_k. The various equations used are displayed in Equations 24 - 26.

$$\varphi_k = \frac{5\pi}{2} + \beta_k - \theta_k \tag{24}$$

$$\varphi_k = -\frac{3\pi}{2} + \beta_k - \theta_k \tag{25}$$

$$\varphi_k = \frac{\pi}{2} + \beta_k - \theta_k \tag{26}$$

Equation 24 was used in the following circumstances:

- β_k was in the 3rd quadrant *and* $\theta_k - \beta_k > \frac{3\pi}{2}$

Equation 25 was used under the following circumstances:

- β_k was in either the 1st, 2nd, or 4th quadrant *and* $\theta_k - \beta_k \leq -\frac{\pi}{2}$

Equation 26 was used under the following circumstances:

- β_k was in the 1st, 2nd, or 4th quadrant *and* $\theta_k - \beta_k > -\frac{\pi}{2}$

or

- β_k was in the 3rd quadrant *and* $\theta_k - \beta_k \leq \frac{3\pi}{2}$

The algorithm used in the *Localization_node* determined which quadrant β_k was in by checking if the vehicles x-coordinate was greater than or equal to the ego-points x-coordinate. If it was, the vehicles y-coordinate was compared to the ego-points y-coordinate to determine if β_k was in either the 2nd or 3rd quadrant. If it wasn't, then there was no need to determine which quadrant β_k fell in because both the 1st and 4th quadrants used the same equation. This algorithm is expressed as a flow chart in Figure 24 where the green diamond was the starting point and the red boxes were the ending equations for the intelligent vehicles pose in the *a priori* map.

The *Localization_node* then took the vehicles cartesian coordinates as well as its pose within the *a priori* map and published them to the *currentPosition* topic using a custom message type.

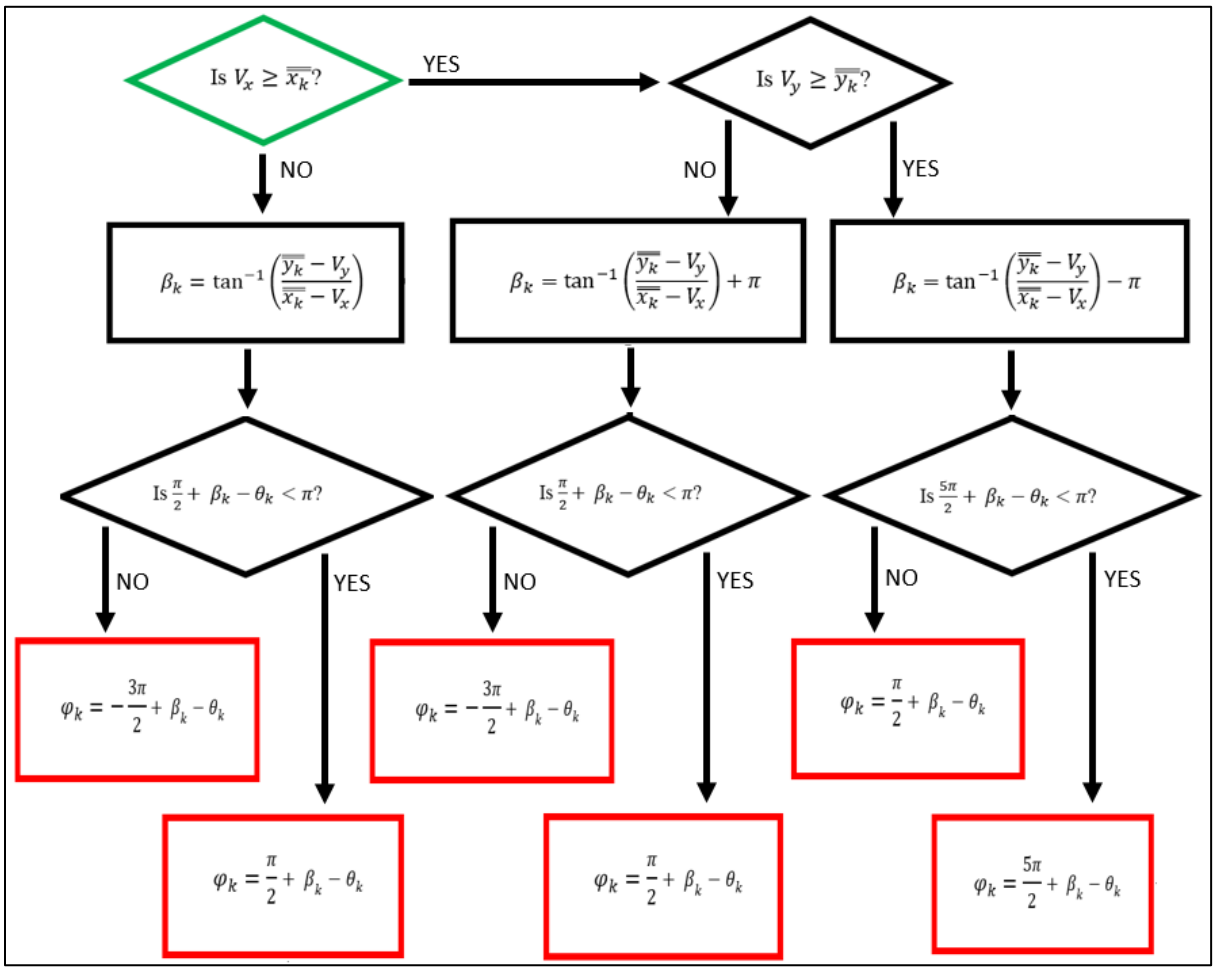

Figure 24: Algorithm Flow Chart for Vehicular Pose Determination

3.4.2.4 Tracking_node

The *tracking_node* was used to plot the vehicles current position during the baseline run. The *tracking_node* subscribed to the *currentPosition* topic and recorded the intelligent vehicles cartesian coordinates and the pose at those coordinates in the *tracked.bag* bag file. The *tracking_node* was also designed so that the recording rate of the vehicles coordinates could be adjusted so that the effects of different recording rates on deviation from baseline during

autonomous navigation could be investigated. Once the vehicle finished its baseline path navigation, the *tracked.bag* bag file was closed for future use during the autonomous navigation trials.

3.4.2.5 Drive_track_node

The *drive_track_node* controlled the wheel speeds on the intelligent vehicle and their durations at those speeds during the baseline navigation run. The wheels' motors were supplied power via two 12V batteries and the motor controller was controlled by an Arduino Mega. The Arduino communicated with ROS through a USB port so that it can be subscribed to the *ROStoArduino* topic. This topic held integer values that represented wheel speed values for the left and right wheel separately. The *drive_track_node* published to this topic using the *wheel_speeds.msg* custom message type. The Arduino applied these wheel speeds to their respective wheel on the intelligent vehicle instantly through a callback function whenever there was data published to the *ROStoArduino* topic.

The *drive_track_node* controlled duration by setting a *wait()* command between each publishing command. This allowed complete control over what type of baseline track was generated for the user. In terms of wheel speeds, negative numbers represented reverse direction and positive numbers represented forward direction. If the wheel speed was zero, it essentially acted as a brake. The Arduino was programmed to take these values and accurately control wheel speeds through the use of a built-in encoder in the Parallax robots' motors. The vehicles speed was maintained at 6.5 ± 0.2 inches/sec during the controlled baseline navigation runs. The experimental setup for the intelligent vehicle's starting point for the left turn baseline trials is displayed in Figure 25 and in Figure 26 for the right turn baseline trials.

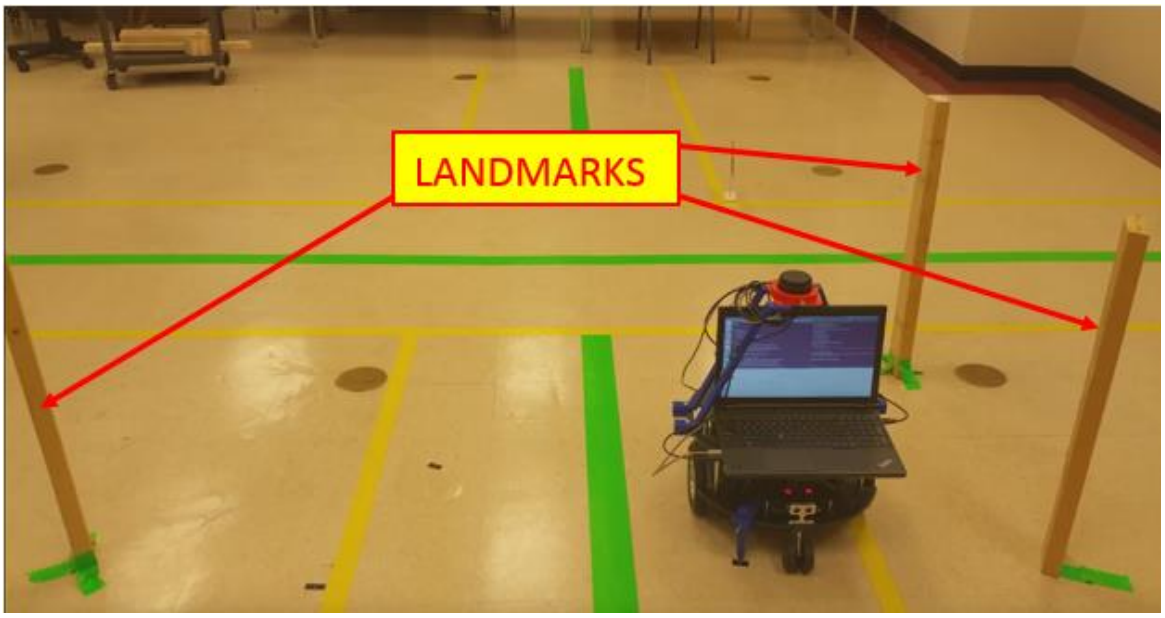

Figure 25: Experimental Setup for the Left Turn Baseline Trials

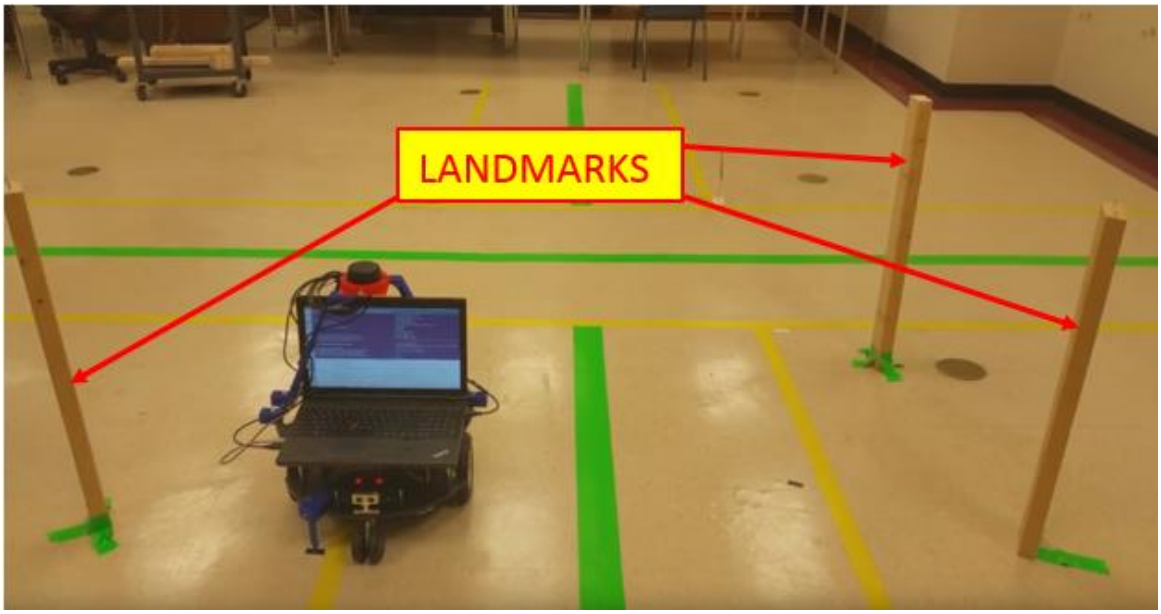

Figure 26: Experimental Setup for the Right Turn Baseline Trials

3.4.2.6 RospySerial_node

The *rospySerial_node* was used to open serial communication through the USB port between ROS and any hardware that was connected to it. The Arduino required this serial communication to subscribe/publish to ROS topics. In order to specify the port name that was used for Arduino communication on the laptop operating ROS, the Arduino was plugged into the laptop

through a USB cable and the port names were searched to find the port name for the Arduino. The port name was "ACM0" and this name was a required parameter when launching the *rospySerial_node*.

3.4.3 Navigating Autonomously with Ego-localized System

3.4.3.1 Autonomous Navigation via Ego-Localization Overview

Autonomous navigation was achieved during the third run of this investigation. The intelligent vehicle was placed at the origin of the baseline path from the previous run and the *navigation.launch* file was launched in ROS. The web of information transferred throughout the nodes from this launch file are displayed in Figure 27.

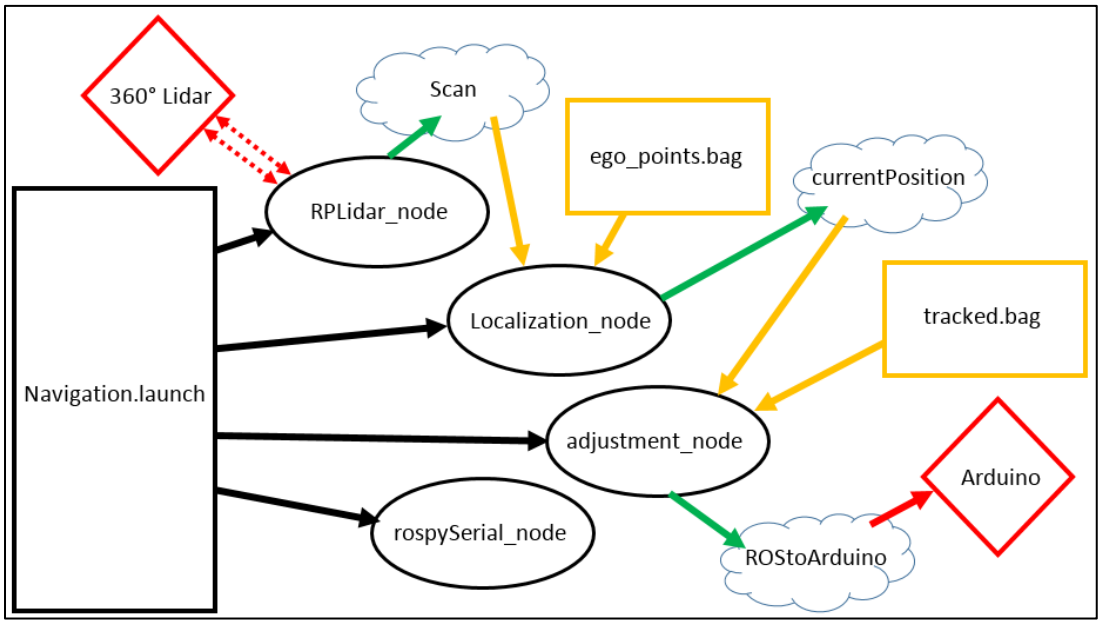

Figure 27: ROS Operations during Autonomous Navigation Run.

This launch file utilized some nodes that were used previously during the baseline run such as the *RPLidar_node*, *Localization_node*, and the *rospySerial_node*. The *adjustment_node* was used only during autonomous navigation so it was not used during the environment zeroing run or the baseline path generation run.

3.4.3.2 Adjustment_node

Autonomous navigation was achieved by taking the current position of the intelligent vehicle in the *a priori* map and determining where it needs to go next in order to accurately navigate the baseline path. The *adjustment_node* performed this operation by first loading the string of cartesian coordinates from the *tracked.bag* bag file and creating a vector for the x-coordinates, y-coordinates, and the φ values. This created an organized list of coordinates that the intelligent vehicle had to navigate to in the order they were listed in. Once the intelligent vehicle navigated to all the listed coordinates, it successfully navigated the baseline path.

The *adjustment_node* was subscribed to the *currentPosition* topic and its callback function contained the algorithms for determining the intelligent vehicles next commands during autonomous navigation. Inside this callback function, the vehicle compared its current position to the next position it needed to navigate to. If the current position was within a predefined acceptable radius of the next position, then the vehicle moved on to the next cartesian coordinates it needed to navigate to. If the autonomous navigation was not within the acceptable radius of the next desired point in the *a priori* map then it decided on whether it needed to make an adjustment in its current navigation path or if it needed to continue navigating at its current wheel speeds.

The vehicle determined its distance from the next "Target" coordinates by using the Pythagorean theorem between its current coordinates and its target coordinates. This equation is presented in Equation 27 where V_D was the vehicles distance from the next target location, T_x, T_y were the targets coordinates, and V_x, V_y were the vehicles current coordinates. The vehicle checked to see if its V_D was less than the preset radius from desired point value that is denoted as T_D; making it within the acceptable radius. If V_D was less than T_D, then next target point was generated from the vectors loaded from the *tracked.bag* file. If V_D was greater than T_D then the vehicle determined

whether it needed to make adjustments in its navigation vector to get to the next coordinates or not.

$$V_D = \sqrt{(T_x - V_x)^2 + \left(T_y - V_y\right)^2} \tag{27}$$

The vehicle was only able to smoothly navigate to the next point if it was headed in its general direction. This means that if the angle between its current vector trajectory and the vector to the next point from its current position was too large, then the vehicle rotated in place until the angle, denoted as $\Delta\varphi$, was within an acceptable range to make wheel adjustments. In order to calculate $\Delta\varphi$, the angle to the next target coordinates in the *a priori* map, φ_{nxt}, was calculated. The calculation of φ_{nxt} depended on which quadrant contained the next target coordinates from the intelligent vehicles frame of reference. The three equations used for calculating φ_{nxt} are provided in Equations 28 - 30 and the flowchart presented in Figure 28 shows the process of determining which equation was used.

$$\varphi_{nxt} = \tan^{-1}\left(\frac{T_y - V_y}{T_x - V_x}\right) + \pi \tag{28}$$

$$\varphi_{nxt} = \tan^{-1}\left(\frac{T_y - V_y}{T_x - V_x}\right) - \pi \tag{29}$$

$$\varphi_{nxt} = \tan^{-1}\left(\frac{T_y - V_y}{T_x - V_x}\right) \tag{30}$$

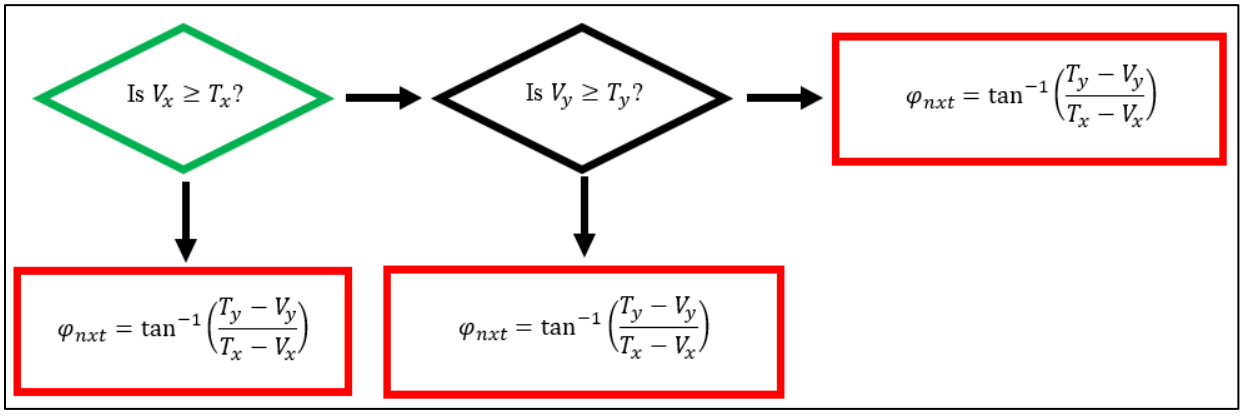

Figure 28: Determining φ_{nxt} during Autonomous Navigation

After φ_{nxt} was found, $\Delta\varphi$ was calculated. However, due to the jump in radians from π to $-\pi$ across the RPLIDARs negative x-axis, the vehicles current phi, φ_{cur}, and the angle to the next target coordinates in the *a priori* map, φ_{nxt}, had to be adjusted accordingly before calculating $\Delta\varphi$. Whether adjustment to φ_{cur}, φ_{nxt}, or both was needed or not depended on certain circumstances regarding their values. Instead of adjusting the values themselves, their adjustments, if any, were applied to the $\Delta\varphi$ calculations, Equations 31 - 33, using the algorithm provided in Figure 29 to determine which equation to use.

$$\Delta\varphi = \varphi_{nxt} - \varphi_{cur} + 2\pi \tag{31}$$

$$\Delta\varphi = \varphi_{nxt} - \varphi_{cur} - 2\pi \tag{32}$$

$$\Delta\varphi = \varphi_{nxt} - \varphi_{cur} \tag{33}$$

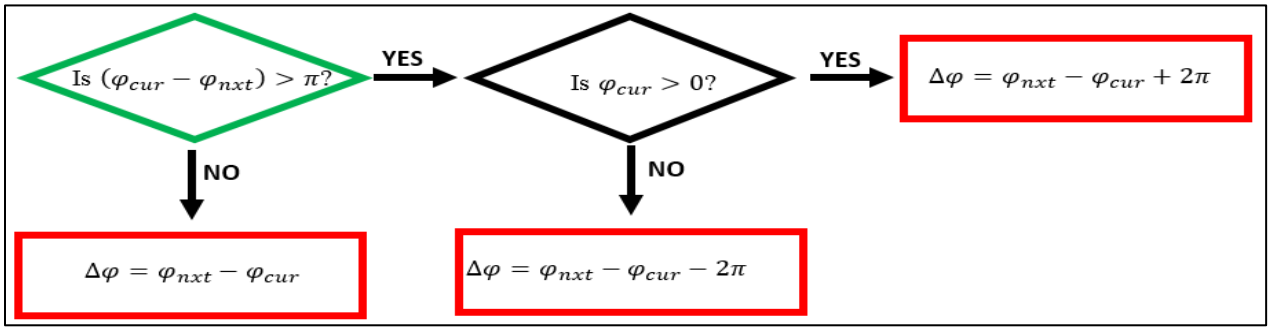

Figure 29: Algorithm for Determining Δφ

This $\Delta\varphi$ was required to be within predetermined angular tolerances from 0 radians in order to make wheel adjustments in the forward direction to reach the next target coordinates. If $\Delta\varphi$ was outside of these *rotation boundaries*, then the intelligent vehicle rotated in place toward the next target coordinates until $\Delta\varphi$ was within the *rotation boundaries*. This investigation set the rotation boundaries to be $0 \pm \frac{\pi}{6}$ radians. If $\Delta\varphi$ was less than $-\frac{\pi}{6}$, then the intelligent vehicle rotated counter-clockwise; it rotated clockwise if it was greater than $\frac{\pi}{6}$ radians. While rotating, the intelligent vehicle constantly calculated it's $\Delta\varphi$ to know when it fell within the *rotation boundaries*. If $\Delta\varphi$

fell within $0 \pm \frac{\pi}{6}$ radians then the intelligent vehicle proceeded to drive straight and determine how much adjustment to the wheel speeds, if any, was needed in order to navigate toward the target coordinates.

In order to determine if the intelligent vehicle needed to make wheel adjustments or not, the vehicle needed to determine how far off from the target it would have been if it had continued its navigation vector. This future offset distance, denoted as F_D, was calculated using Equation 34.

$$F_D = |V_D \times \sin(\Delta\varphi)| \tag{34}$$

If F_D, was less than the predetermined offset distance, denoted as F_{max}, then the intelligent vehicle continued its current trajectory by publishing the same forward direction value for both wheel speeds. If F_D was greater than F_{max}, then one wheel speed was increased and the other decreased. Which wheel that had its speed increased/decreased and by how much depended on the magnitude of F_{max} and whether $\Delta\varphi$ was positive or negative. F_{max} was set to 3 cm for the autonomous navigation trials in this investigation.

The algorithm used to make wheel adjustments during the autonomous navigation trials is provided in Figure 30. The value for the *rotation boundaries* was adjustable in the *adjustment_node* and was called *TURN* in the wheel adjustment algorithm. The functions *rotate_left(), rotate_right(), check_left_adjustment(),* and *check_right_adjustment()* were used to change the wheel speed values for the intelligent vehicles based on the situation in the wheel adjustment algorithm.

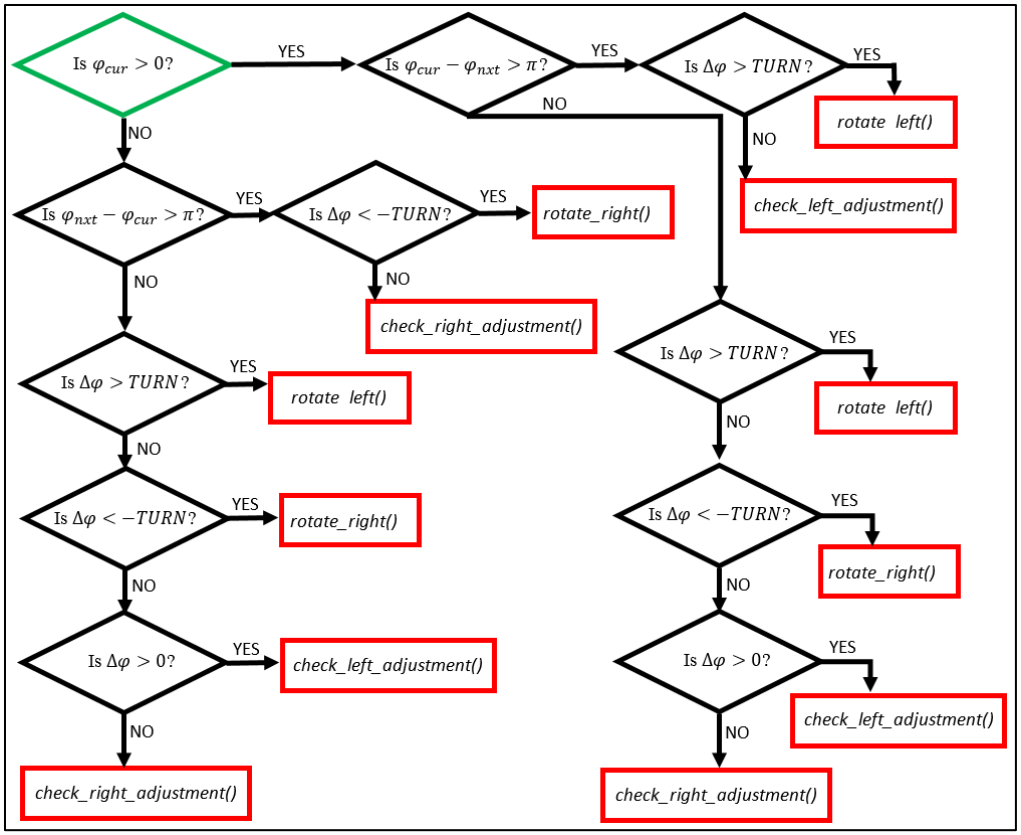

Figure 30: Wheel Adjustment Algorithm during Autonomous Navigation Trials

The function *rotate_left()* set the left wheel speed to be -50 and 50 for the right wheel speed. This rotated the intelligent vehicle in place going in the counter-clockwise direction. The *rotate_right()* function had the same effect in the clockwise direction by setting the left wheel speed to be 50 and -50 for the right wheel speed. If the intelligent vehicle did not need to rotate due to the $\Delta\varphi$ being within the *rotation boundaries* then it used either the *check_left_adjustment()* or the *check_right_adjustment()* function depending on it $\Delta\varphi$ was positive (*check_left_adjustment()*) or negative (*check_right_adjustment()*).

For these adjustment functions, the intelligent vehicle had to take into account both its future offset distance, F_D, and its distance away from the next target coordinates, V_D, when determining if it should make wheel speed adjustments or continue to drive straight. This was needed because if the intelligent vehicle made wheel corrections when it was relatively far away,

it could have over corrected making it have to make wheel corrections going in the opposite direction after the next RPLIDAR scan. This would have created unnecessary zig-zag patterns during the autonomous navigation trials. The intelligent vehicle therefor decided if it needed to go straight or make wheel corrections based on the ratio of F_D to V_D.

In order to keep the vehicles speed constant, the average of the left and right wheel speed always averaged a predetermined *normal* value of 50 whenever it navigated forward. This value of 50 translated into 6.5 ± 0.2 inches/sec of vehicular speed. The left wheel speed was calculated first using Equation 35 where C_w was the wheel adjustment constant; the equation used the "+" sign in the *check_left_adjustment()* function and it used the "-" sign in the *check_right_adjustment()* function. Once the left wheel speed value was found, the right wheel speed value was calculated using Equation 36. However, for the *check_left_adjustment()* function, the minimum value that the left wheel speed could be was set to 10. If the calculated left wheel speed was less than ten then the left wheel speed value was given a value of ten. For the *check_right_adjustment()* function the maximum left wheel speed was set to 90 so if the calculated went over that value, it was just given a value of 90.

$$left\ wheel\ speed = normal \pm \left(C_w \times \frac{F_D}{V_D}\right) \tag{35}$$

$$right\ wheel\ speed = normal + (normal - left\ wheel\ speed) \tag{36}$$

The intelligent vehicle continued this wheel speed adjustment process for every time when an ROS message was published to the *currentPosition* topic. The intelligent vehicle ended the autonomous navigation trial once it reached the final target point.

CHAPTER 4. RESULTS

4.1 Baseline Path Generation

Two baseline paths were generated for autonomous navigation testing. The first test generated a left turn path and the second generated a right turn path. For each test, the robots wheels were set at the same speed for two seconds traveling along the positive y-axis in the *a priori* map, then the wheels were set to make an arching path either left or right. This arching path continued until the *Localization_node* was publishing data that determined it had turned 90 degrees in the direction it was turning. The intelligent vehicle was then instructed to drive forward but keep adjusting its wheels so that it maintained a y-coordinate similar to the one that was published when the intelligent vehicle had finally turned 90 degrees. Maintaining this y-coordinate while driving forward created a square angle between the initial straight path, before turning started, and the straight path after it turned 90 degrees; this path mimicked a four-way intersection. Each baseline path generation was performed five times and their published current position data was averaged to generate a discretized baseline path for autonomous navigation. The baseline path and the resulting discrete points that were averaged from the paths are provided in Figure 31 for the left turning path and in Figure 32 for the right turning path.

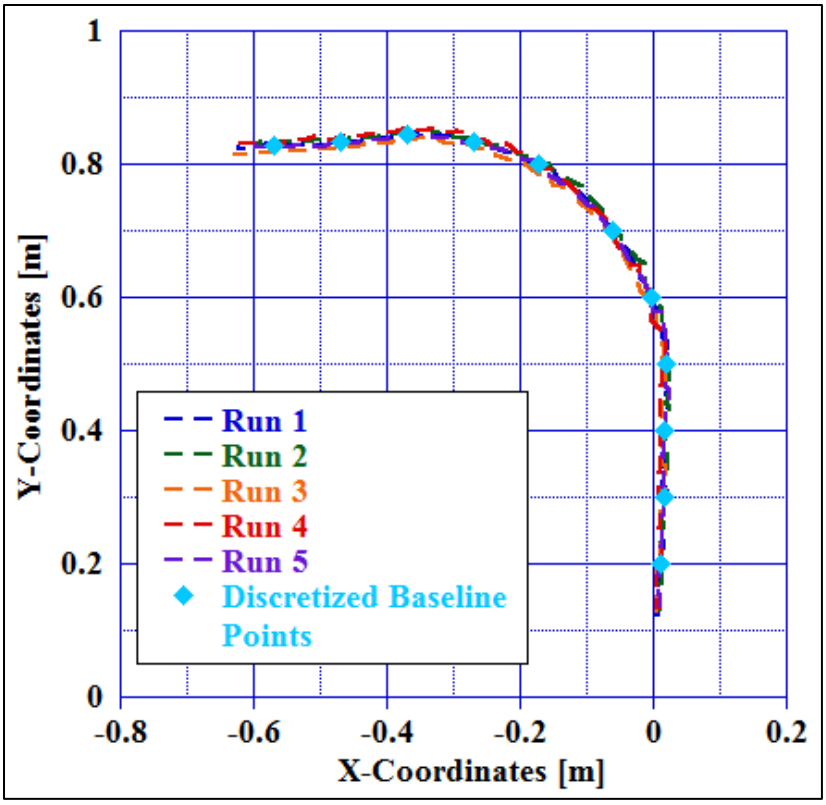

Figure 31: Turning Left Baseline Path Runs and Resulting Discretized Points

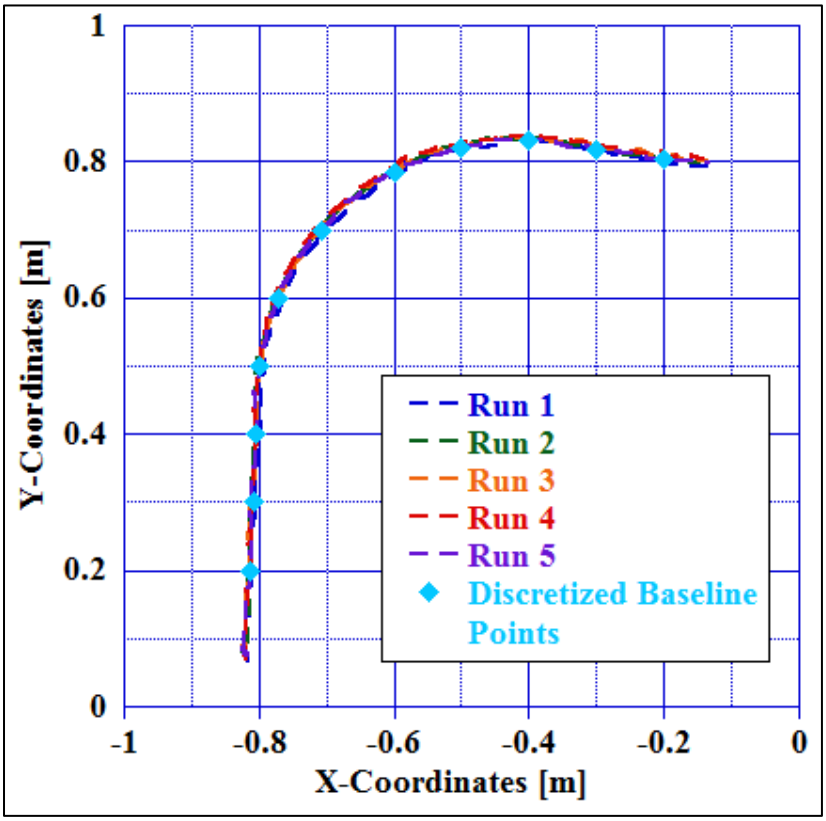

Figure 32: Turning Right Baseline Path Runs and Resulting Discretized Points

4.2 Autonomous Navigation Trials

The intelligent vehicle in this study performed autonomous navigation by navigating to the discretized baseline points for both the turning left and turning right baseline path runs. Autonomous navigation was performed three times for each baseline path and the vehicles cartesian coordinates and pose orientation was recorded throughout each run. The resulting tracked navigation paths are provided in Figure 33 for the left turn trials and in Figure 34 for the right turn trials.

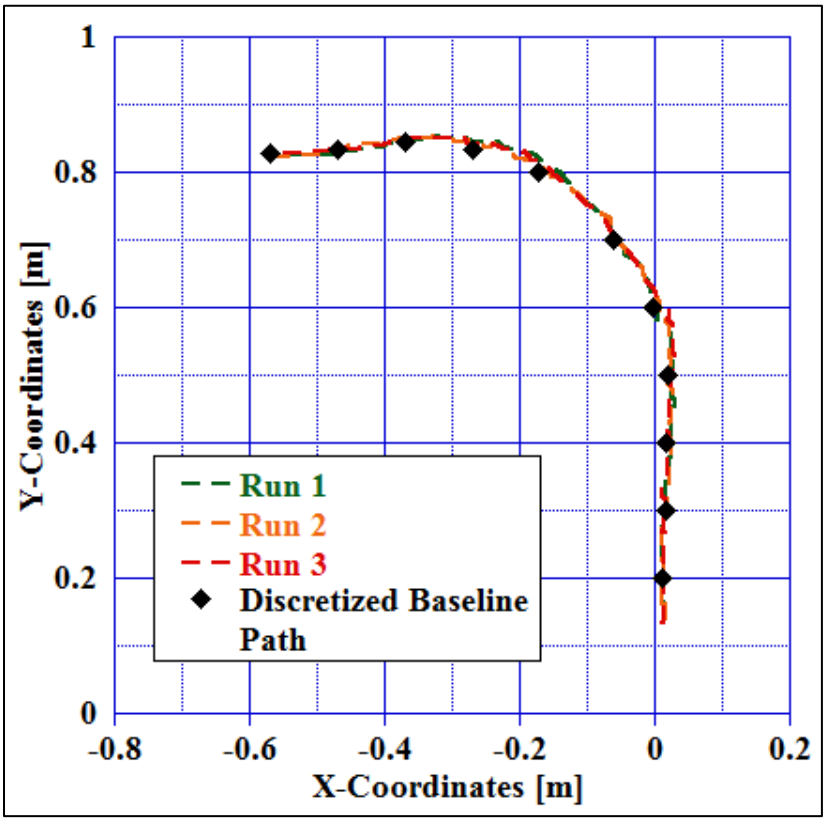

Figure 33: Turning Left Autonomous Navigation Recorded Pathings

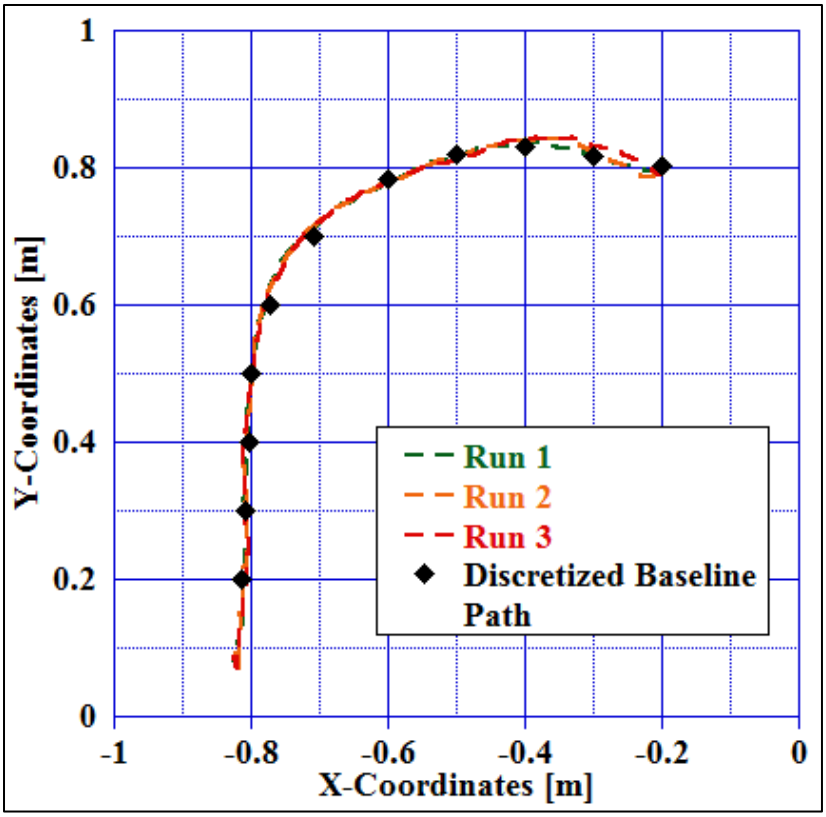

Figure 34: Turning Right Autonomous Navigation Recorded Pathings

4.3 Analysis of Results

The baseline paths and the trials were compared to determine how far the intelligent vehicle deviated from the baseline path when performing autonomous navigation for both turning paths. This was accomplished by interpolating the baselines x and y coordinates to match those of the trial coordinates. The x-coordinates were interpolated up to the point in the baseline path where there was a greater change in x-coordinates than the y-coordinates between the successive discretized baseline points. For the left turn trials, the x-coordinates were interpolated up to the 7^{th} baseline point and the y-coordinates were interpolated for all data after that up to the final 11^{th} point. For the right turn trials, the x-coordinates were interpolated up to the 6^{th} baseline point and the y-coordinates after that until the final 10^{th} point.

For the left turn trials, the absolute difference in the y-coordinates between the trials and the baselines were calculated up to the 7^{th} baseline point where the absolute difference in the x-

coordinates were calculated for the remainder of each trial. These linear differences between their readings and their baseline interpolated data points are considered errors in ego-localization and they are provided in Figure 35. For the right turn trials, the x-coordinate differences were calculated up to the 6th baseline point and then the y-coordinate differences were calculated to the final 10th baseline point; these ego-localization errors were plotted in Figure 36.

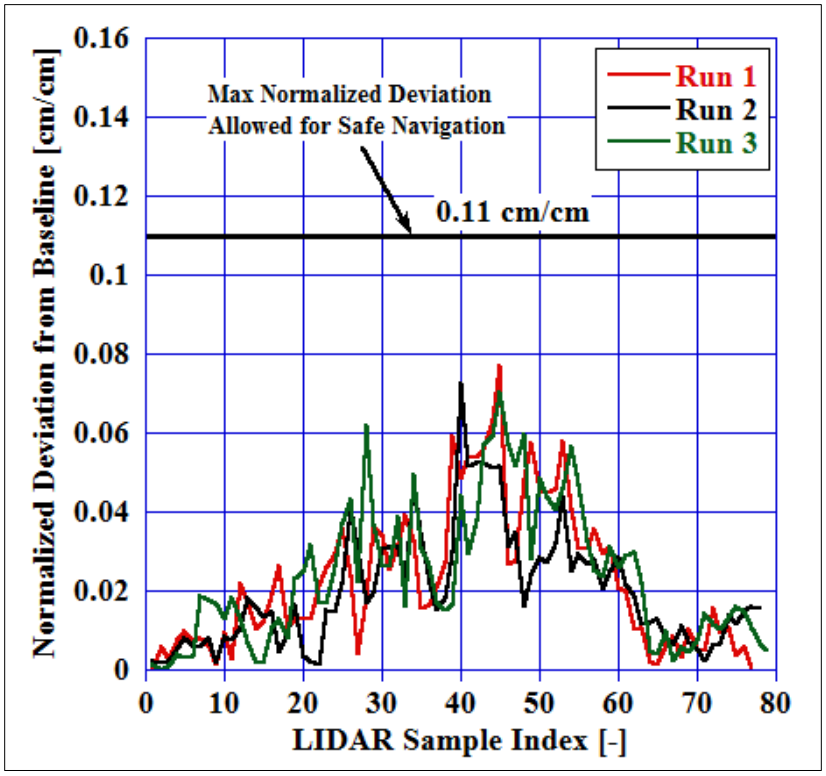

Figure 35: Normalized Ego-localization Deviations from Baseline Path during Left Turn Autonomous Navigation Trials

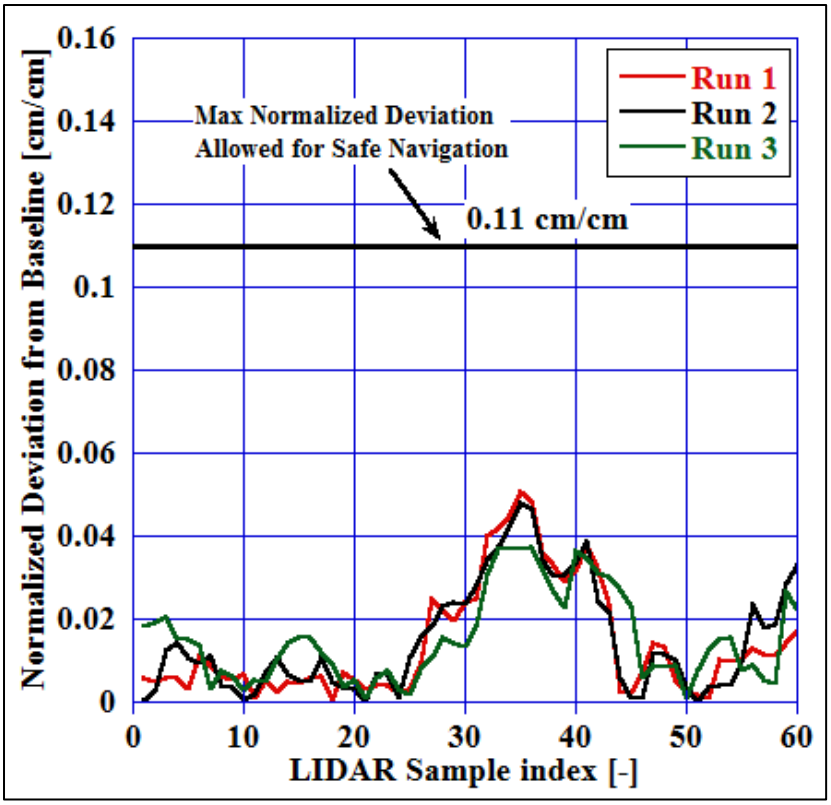

Figure 36: Normalized Ego-localization Deviations from Baseline Path during Right Turn Autonomous Navigation Trials

The baseline turning paths were generated from smooth tangential turns between two straight paths. These smooth transitional turns are ideal for comfortable driving and should be replicated as much as possible during autonomous navigation. The variance in the intelligent vehicles orientation during the autonomous navigation runs compared to the baseline runs are provided in Figures Figure 37 & Figure 38 for the left turn trials and right turn trials respectively. Statistical values for the ego-localization and orientation errors from the baseline runs are provided in Table 2.

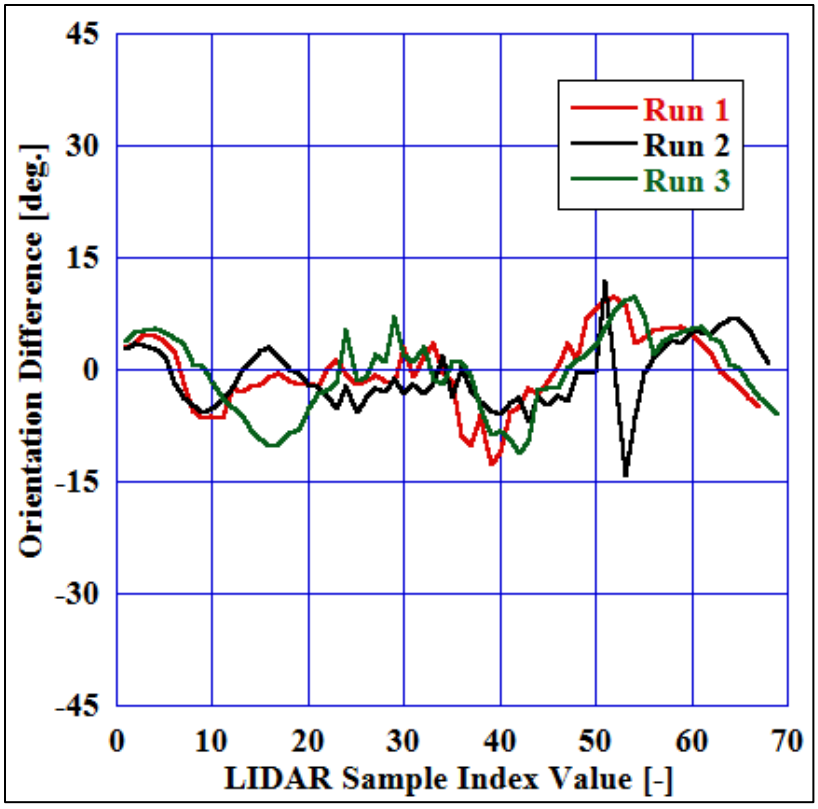

Figure 37: Orientation Errors during Left Turn Autonomous Navigation Trials

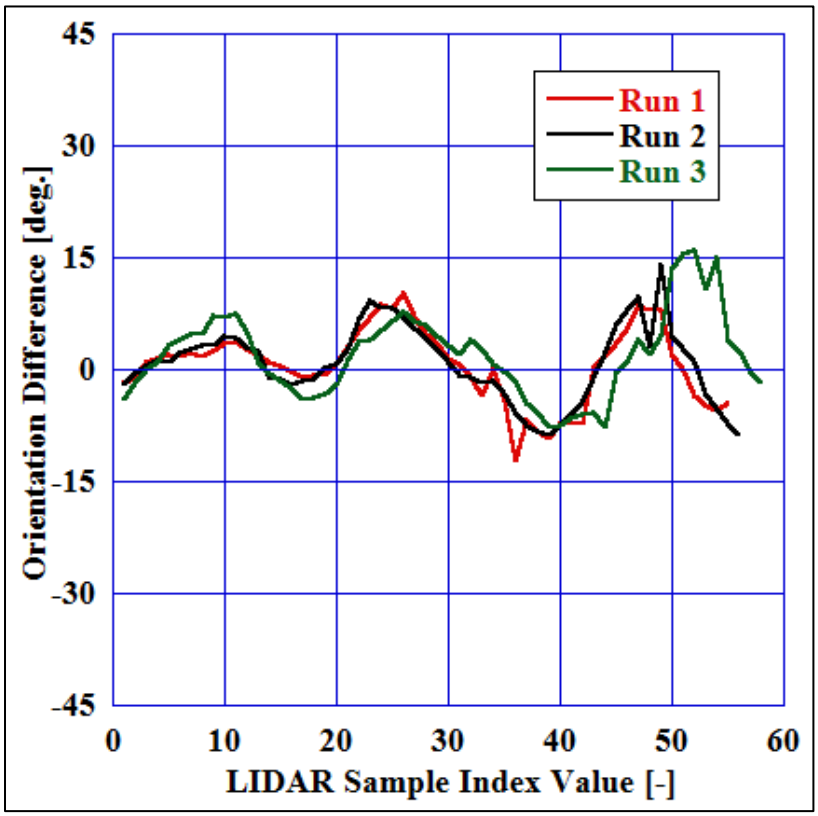

Figure 38: Orientation Errors during Right Turn Autonomous Navigation Trials

Table 2: Statistical Values for All Autonomous Navigation Trials

	Trial Number	Normalized Deviations [cm/cm]		Orientation Differences [deg.]	
		Mean	Standard Deviation	Mean	Standard Deviation
Turning Left	**1**	0.023	0.018	-0.237	4.77
	2	0.015	0.020	-0.802	4.20
	3	0.024	0.018	-0.377	5.30
Turning Right	**1**	0.014	0.013	0.615	4.92
	2	0.016	0.013	0.789	5.01
	3	0.018	0.013	1.98	5.61

4.4 Assessment of Results and Criteria for Success

Autonomous navigation failure occurs when the intelligent vehicle fails to stay within its lane when navigating. Roads have a minimal 12’ width on interstate highways and 10’ minimal width in urban areas. With the 10’ lane width, typical standard size cars have a gap distance between the side of the car and the lane line that is roughly 1/3 of the vehicles width if the vehicle was in the middle of the lane. Applying this concept to this investigation, the width of the intelligent vehicle was 45 cm giving a distance of 15 cm for the gap between the vehicles side and the driving lane. Meaning the intelligent vehicle stayed within the driving lane during the autonomous navigation trials if it deviated no more than 15 cm from its baseline path. However, vehicles riding along a lane line is unsafe so the vehicle must stay relatively close to the center of the lane while navigating in order to be considered safe. A factor of safety of 3 applied to the gap distance gives 5 cm which is the max distance the intelligent vehicle can deviate from the baseline path in order to consider its autonomous navigation via ego-localization as a success. This max distance was normalized with the width of the intelligent vehicle in order to determine the max normalized deviation that can occur which can scale to various vehicular widths. Dividing the max deviation distance of 5 cm by the 45 cm width of the intelligent vehicle resulted in a max

normalized deviation of the baseline path of 0.111 cm/cm. A diagram of these measurement is provided in Figure 39. If the intelligent vehicle results indicated that it would always deviate, when normalized to the vehicles width, no more than 0.111 cm/cm on a 100% confidence interval then the autonomous navigation trial was a success.

The max normalized deviation from the baseline path during the left turn trials was 0.077 cm/cm which occurred during the first trial. For the right turn trials, the max normalized deviation from the baseline path was 0.051 cm/cm which occurred during the first trial. This means that autonomous navigation was achieved using a factor of safety of 3. For both trials, the max deviation occurred during the turning phase where the intelligent vehicle had to make vector trajectory adjustments to remain near the baseline path as it was turning.

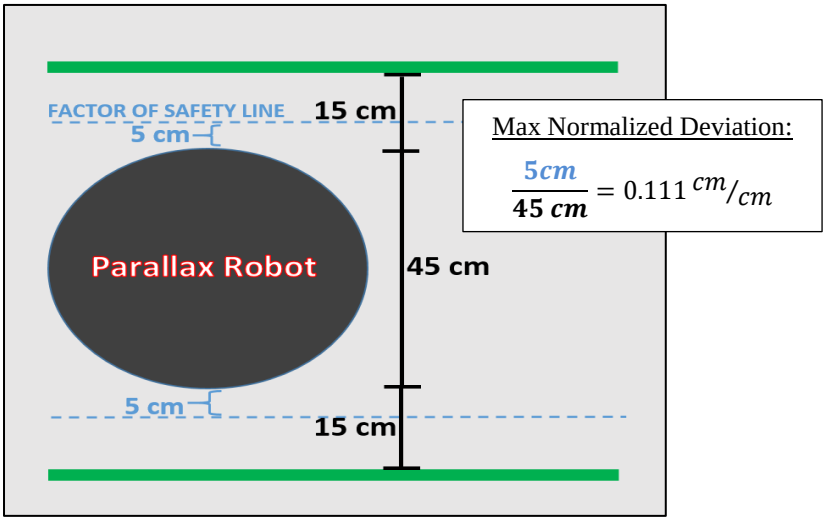

Figure 39: Measurements for Criteria of Success for Performing Autonomous Navigation

The intelligent vehicle was able to stay within a 26 degree angular frame from the angle it should have been at according to the baseline orientation data for the left turn trials. This frame was slightly narrower during the right turn trials with an angular frame window of 24 degrees. These angular windows demonstrate that the intelligent vehicle was consistently moving forward to the next baseline target point as demonstrated in the autonomous navigation paths in Figures Figure 33 & Figure 34.

CHAPTER 5. CONCLUSIONS

5.1 Hypothesis Confirmation

The hypothesis for this investigation was as followed: *If landmark coordinates are built in an a priori environment using a 2-D LIDAR sensor while navigating a path, then a vehicle can autonomously follow that path by only referencing the landmarks in the a priori environment from a local 2-D LIDAR sensor.*

Autonomous navigation was achieved by building ego-point coordinates inside an *a priori* environment and referencing those ego-point coordinates in order to ego-localize the position of the intelligent vehicle during autonomous navigation. This navigation technique was able to generate an infinite number of baseline tracks as long as the landmarks were visible to the RPLIDAR on the intelligent vehicle so that all ego-points could be found. These coordinates also needed to be stored in such a manner so that the intelligent vehicle could load the ego-points' coordinates to be used in its navigation algorithms.

5.2 Real World Implementation

Autonomous navigation via LIDAR scanners is only applicable in environments where there are an abundance of physical objects to detect. Urban areas are the most ideal environments because buildings, road signs, and traffic lights are all able to be detected by LIDAR scanners and all these physical objects are widespread in urban areas. Urban areas are also the most important areas for autonomous navigation due to the high traffic flow throughout the cities and neighboring highways. The baseline paths can be generated by mounting LIDAR scanners to vehicles and having human drivers continue to navigate the urban environments as they normally would. While the human driver is navigating the environment, landmarks can be detected from the LIDAR scanner and the data can be uploaded to a cloud based system keeping track of detected landmarks

so ego-points for that environment can be generated. These ego-points would be tied with GPS coordinates so that when autonomous navigation does need to be performed using ego-point coordinates, the vehicle will only load ego-points around its GPS location. Once enough data has been collected from drivers, autonomous navigation can be implemented onto vehicles and eventually every car on the road would have this capability as more intelligent vehicles hit the market.

5.3 Future Research

Ideally, autonomous navigation via LIDAR information in urban environments would reference building corners as landmarks because they are highly distinguishable in LIDAR data. Also, a building corner represents an absolute point and would not involve averaging a group of LIDAR data points to determine ego-point locations which can lead to slightly dynamic coordinate positions in an *a priori* map. Also, an intelligent vehicle could build a baseline path by following lane lines which would involve implementation of lane detecting cameras. These cameras can also be used to detect stop signs so navigating an intersection in an urban environment would be realistic.

REFERENCES

Bailey, T., J. Nieto, and E. Nebot. 2006. "Consistency of the FastSLAM algorithm." Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006., 15-19 May 2006.

Chee, Y.W., and T.L. Yang. 2013. Vision and LIDAR Feature Extraction. Ithaca, NY, USA: Cornell University.

Chieh-Chih, Wang, C. Thorpe, and S. Thrun. 2003. "Online simultaneous localization and mapping with detection and tracking of moving objects: theory and results from a ground vehicle in crowded urban areas." 2003 IEEE International Conference on Robotics and Automation (Cat. No.03CH37422), 14-19 Sept. 2003.

Davison, Andrew J. 2003. "Real-Time Simultaneous Localisation and Mapping with a Single Camera." Proceedings of the Ninth IEEE International Conference on Computer Vision - Volume 2.

Dawood, M., C. Cappelle, M. E. El Najjar, M. Khalil, and D. Pomorski. 2011. "Vehicle geo-localization based on IMM-UKF data fusion using a GPS receiver, a video camera and a 3D city model." 2011 IEEE Intelligent Vehicles Symposium (IV), 5-9 June 2011.

Dawood, Maya, Cindy Cappelle, Maan E. El Najjar, Mohamad Khalil, Bachar El Hassan, Denis Pomorski, and Jing Peng. 2016. "Virtual 3D city model as a priori information source for vehicle localization system." *Transportation Research Part C: Emerging Technologies* 63:1-22. doi: http://dx.doi.org/10.1016/j.trc.2015.12.003.

Dissanayake, M. W. M. G., P. Newman, S. Clark, H. F. Durrant-Whyte, and M. Csorba. 2001. "A solution to the simultaneous localization and map building (SLAM) problem." *IEEE Transactions on Robotics and Automation* 17 (3):229-241. doi: 10.1109/70.938381.

Durrant-Whyte, H., and T. Bailey. 2006. "Simultaneous localization and mapping: part I." *IEEE Robotics & Automation Magazine* 13 (2):99-110. doi: 10.1109/MRA.2006.1638022.

Durrant-Whyte, H. F. 1988. "Uncertain geometry in robotics." *IEEE Journal on Robotics and Automation* 4 (1):23-31. doi: 10.1109/56.768.

Grimes, J. G. 2008. Global Positioning System Standard Positioning Service Performance Standard. edited by Department of Defense.

Guivant, J. E., and E. M. Nebot. 2001. "Optimization of the simultaneous localization and map-building algorithm for real-time implementation." *IEEE Transactions on Robotics and Automation* 17 (3):242-257. doi: 10.1109/70.938382.

Guivant, Jose, Eduardo Nebot, and Stephan Baiker. 2000. *Autonomous Navigation and Map building Using Laser Range Sensors in Outdoor Applications*. Vol. 17.

Guizzo, Erico. 2011. How Google's Self-Driving Car Works. *IEEE Spectrum*. Accessed Sept. 6, 2017.

Hahnel, D., D. Schulz, and W. Burgard. 2002. "Map building with mobile robots in populated environments." IEEE/RSJ International Conference on Intelligent Robots and Systems, 2002.

Hata, A., and D. Wolf. 2014. "Road marking detection using LIDAR reflective intensity data and its application to vehicle localization." 17th International IEEE Conference on Intelligent Transportation Systems (ITSC), 8-11 Oct. 2014.

Hidalgo, F., and T. Bräunl. 2015. "Review of underwater SLAM techniques." 2015 6th International Conference on Automation, Robotics and Applications (ICARA), 17-19 Feb. 2015.

Hirsch, Jerry. 2015. "Elon Musk: Model S not a car but a 'sophisticated computer on wheels'." *Los Angeles Times*, March 19, 2015. Accessed Sept. 6, 2017.

J. Leonard, John, Hans Jacob, and S. Feder. 2000. *A Computationally Efficient Method for Large-Scale Concurrent Mapping and Localization*.

Julier, S. J., and J. K. Uhlmann. 2001. "A counter example to the theory of simultaneous localization and map building." Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164), 2001.

Jun-Hyuck, Im, Im Sung-Hyuck, and Jee Gyu-In. 2016. "Vertical Corner Feature Based Precise Vehicle Localization Using 3D LIDAR in Urban Area." *Sensors (14248220)* 16 (8):1268-1290. doi: 10.3390/s16081268.

Khairuddin, A. R., M. S. Talib, and H. Haron. 2015. "Review on simultaneous localization and mapping (SLAM)." 2015 IEEE International Conference on Control System, Computing and Engineering (ICCSCE), 27-29 Nov. 2015.

Kim, Sung-Soo, Kyong-Ho Kim, Seong-Ho Lee, and Jong-Hun Lee. 2005. *Video navigation system using the geographic hypermedia*.

Leonard, J. J., and H. F. Durrant-Whyte. 1991. "Mobile robot localization by tracking geometric beacons." *IEEE Transactions on Robotics and Automation* 7 (3):376-382. doi: 10.1109/70.88147.

Levinson, J., and S. Thrun. 2010. "Robust vehicle localization in urban environments using probabilistic maps." 2010 IEEE International Conference on Robotics and Automation, 3-7 May 2010.

Levinson, Jesse, Michael Montemerlo, and Sebastian Thrun. 2007. *Map-Based Precision Vehicle Localization in Urban Environments*.

Li, Yangming, and Edwin B Olson. 2010. *Extracting general-purpose features from LIDAR data*.

Montemerlo, Michael, Sebastian Thrun, Daphne Koller, and Ben Wegbreit. 2002. *FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem*.

Murphy, Kevin. 2000. *Bayesian Map Learning in Dynamic Environments*.

Naranjo-Hernandez, J. E., F. Jimenez-Alonso, M. Clavijo-Jimenez, and O. Gomez-Casado. 2016. "Reference Element Guidance of Autonomous Vehicles Using 3D Laser Scanner." *DYNA*.

Neira, J., and J. D. Tardos. 2001. "Data association in stochastic mapping using the joint compatibility test." *IEEE Transactions on Robotics and Automation* 17 (6):890-897. doi: 10.1109/70.976019.

NHTSA. 2016. Federal Automated Vehicles Policy. edited by U.S. Department of Transportation.

Nkoro, A. B., and Y. A. Vershinin. 2014. "Current and future trends in applications of Intelligent Transport Systems on cars and infrastructure." 17th International IEEE Conference on Intelligent Transportation Systems (ITSC), 8-11 Oct. 2014.

Park, Yoonsu, Seok Min Yun, Chee Sun Won, Kyungeun Cho, Kyhyun Um, and Sungdae Sim. 2014. "Calibration between Color Camera and 3D LIDAR Instruments with a Polygonal Planar Board." *Sensors* 14:5333-5353. doi: 10.3390/s140305333.

Peng, J., M. E. El Najjar, C. Cappelle, D. Pomorski, F. Charpillet, and A. Deeb. 2009. "A novel geo-localisation method using GPS, 3D-GIS and laser scanner for intelligent vehicle navigation in urban areas." 2009 International Conference on Advanced Robotics, 22-26 June 2009.

Ranganathan, A., D. Ilstrup, and T. Wu. 2013. "Light-weight localization for vehicles using road markings." 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems, 3-7 Nov. 2013.

Se, Stephen, David Lowe, and Jim Little. 2002. "Mobile Robot Localization and Mapping with Uncertainty using Scale-Invariant Visual Landmarks." *The International Journal of Robotics Research* 21 (8):735-758. doi: 10.1177/027836402761412467.

Smith, R., M. Self, and P. Cheeseman. 1987. "Estimating uncertain spatial relationships in robotics." Proceedings. 1987 IEEE International Conference on Robotics and Automation, Mar 1987.

Smith, Randall C., and Peter Cheeseman. 1986. "On the Representation and Estimation of Spatial Uncertainty." *The International Journal of Robotics Research* 5 (4):56-68. doi: doi:10.1177/027836498600500404.

Takase, Y., N. Sho, A. Sone, and K. Shimiya. 2003. "Automatic Generation of 3D City Models and Related Applications." WG V/6 Visualization and Animation of Reality-based 3D Models, Tarasp-Vulpera, Engadin, Switzerland.

Takase, Y., A. Sone, T. Hatanaka, M. Shiroki, and T. Masumi. 2004. "A Development of 3D Urban Information System on Web." WG V/6 Processing and Visualization Using High-Resolution Images, Pitsanulok, Thailand.

Tariq, S., Choi Hyunsoo, C. M. Wasiq, and Park Heemin. 2016. "Controlled parking for self-driving cars." 2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC), 9-12 Oct. 2016.

Thrun, Sebastian, Wolfram Burgard, and Dieter Fox. 1998. "A Probabilistic Approach to Concurrent Mapping and Localization for Mobile Robots." *Machine Learning* 31 (1):29-53. doi: 10.1023/a:1007436523611.

Thrun, Sebastian, Wolfram Burgard, and Dieter Fox. 2000. *A Real-Time Algorithm for Mobile Robot Mapping With Applications to Multi-Robot and 3D Mapping*. Vol. 1.

Veronese, L. d. P., J. Guivant, F. A. A. Cheein, T. Oliveira-Santos, F. Mutz, E. de Aguiar, C. Badue, and A. F. De Souza. 2016. "A light-weight yet accurate localization system for autonomous cars in large-scale and complex environments." 2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC), 1-4 Nov. 2016.

Weerakoon, Tharindu, Kazuo Ishii, and Amir Ali Forough Nassiraei. 2015. *Geometric Feature Extraction from 2D Laser Range Data for Mobile Robot Navigation*.

Wolcott, R. W., and R. M. Eustice. 2014. "Visual localization within LIDAR maps for automated urban driving." 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, 14-18 Sept. 2014.

Zhang, Zhengyou. 1994. "Iterative point matching for registration of free-form curves and surfaces." *Int. J. Comput. Vision* 13 (2):119-152. doi: 10.1007/bf01427149.

APPENDICES

APPENDIX A

ZEROING NODE IN ROS

Below is the c++ file that was used in ROS to create the *zeroing_node*. This node was used to initialize the environment by detecting landmarks from the LIDAR data and storing their generated ego-point coordinates in the *a priori* map. The ego-points' coordinates were stored in an ROS .bag file for future baseline generation and autonomous navigation trials.

```
#include "localization.h"
#include "points.h"
#include "orientation.h"

float reject;
float grouping_constant;
int i;
int j;
int ego_size;
int size;
int scan_count;
int m;
int n;
int o;
int k;
int kk;
int ll;
float angle_increment;
float angle_min;
float angle;
float test_x;
float test_y;
float abs_diff_x;
float abs_diff_y;
float x;
float y;
float x_0, y_0, r_0, x_1, y_1, r_1, x_2, y_2, r_2;
float yaw, yaw_sum, yaw_final;
bool grouped;


float calcx(float range, float angle){
        float x = range * cos(angle);
        return x;}

float calcy(float range, float angle){
        float y = range * sin(angle);
        return y;}

Points p1;

rosbag::Bag bag;


int main(int argc, char** argv){
        sleep(1);

        reject = 1.7;

        grouping_constant = 0.15;

        ros::init(argc, argv, "zeroing_node");

        ros::NodeHandle nh;

        bag.open("egoPoints.bag", rosbag::bagmode::Write);

        naes::ego_points msg;

        ros::Subscriber sub;

        ros::Rate r(1);

        vector<float> ego_points;

        vector<Localization> ego;

        sub = nh.subscribe<sensor_msgs::LaserScan>("/scan",10, &Points::ScanCallback, &p1);

        while(ros::ok()){
                scan_count = 0;
                while(scan_count < 3){
                        ego_points.clear();
                        ego_points = p1.getvector();
                        ego_size = ego_points.size();
                        if(ego_size == 0){
                        }
                        else{
```

```
                        angle_increment = p1.get_inc();
                        angle_min = p1.get_min();
                        for(i=0; i < ego_size; i++){
                                if(ego_points[i] < reject){
                                        angle = angle_min + (i * angle_increment);
                                        x = calcx(ego_points[i], angle);
                                        y = calcy(ego_points[i], angle);

                                        if(ego.size() == 0){
                                                ego.resize(1);
                                                ego[0].pushx(x);
                                                ego[0].pushy(y);
                                        }
                                        else{// must go through all the current objects.
                                                size = ego.size();
                                                grouped = false;
                                                for(j=0; j<size; j++){
                                                        test_x = ego[j].get_localization_x();
                                                        test_y = ego[j].get_localization_y();
                                                        abs_diff_x = abs(x-test_x);
                                                        abs_diff_y = abs(y-test_y);
                                                        if((abs_diff_x <= grouping_constant)&&(abs_diff_y <= grouping_constant)){
                                                                ego[j].pushx(x);
                                                                ego[j].pushy(y);
                                                                grouped = true;
                                                        }
                                                        else{ //it does not belong to this group, do nothing and check the next one;
                                                        }
                                                }
                                                if(grouped == false){ //need to make a new object in ego vector.
                                                        ego.resize(size+1);
                                                        ego[size].pushx(x);
                                                        ego[size].pushy(y);
                                                }
                                                else{ //already grouped from the loop. do nothing finished with ego_point[i];
                                                }
                                        }
                                }
                        }
                        scan_count = scan_count + 1;
                        cout << "number of ego points is: " << ego.size() << endl;
                }
                r.sleep();
                ros::spinOnce();
        }// end of scan_count while loop

        ros::shutdown();
        o = ego.size();
        if(o == 3){
                vector<float> toAverageX;
                vector<float> toAverageY;
                for(k = 0; k < o; k++){
                        cout << "size of x object: " << ego[k].get_x_size() << endl;
                        cout << "size of y object: " << ego[k].get_y_size() << endl;
                }

                for(k = 0; k < o; k++){
                        toAverageX = ego[k].get_x_vector();

                        toAverageY = ego[k].get_y_vector();
                        m = toAverageX.size();
                        n = toAverageY.size();
                        float xsum = 0.0;
                        float ysum = 0.0;
                        for(kk = 0; kk < m; kk++){
                                xsum = xsum + toAverageX[kk];
                        }
                        for(ll = 0; ll < n; ll++){
                                ysum = ysum + toAverageY[ll];
                        }
                        float x_final = xsum / m;
                        float y_final = ysum / n;
                        ego[k].clearxego();
                        ego[k].clearyego();
                        ego[k].pushy(y_final);


                        ego[k].pushx(x_final);
                        toAverageX.clear();
                        toAverageY.clear();
                        toAverageX.resize(1);
                        toAverageY.resize(1);
                }

                x_0 = ego[0].get_localization_x();
                y_0 = ego[0].get_localization_y();
                x_1 = ego[1].get_localization_x();
                y_1 = ego[1].get_localization_y();
                x_2 = ego[2].get_localization_x();
                y_2 = ego[2].get_localization_y();
                msg.x_0 = x_0;
                msg.y_0 = y_0;
                msg.x_1 = x_1;
                msg.y_1 = y_1;
                msg.x_2 = x_2;
                msg.y_2 = y_2;
                bag.write("ego_points_topic", ros::Time::now(), msg);
                bag.close();
        }
        else{
                cout << "ERROR WITH ZEROING, THE NUMBER OF EGO POINTS WAS NOT THREE" << endl;}
    }// end of ros::ok()
    return 0;
}
```

APPENDIX B

LOCALIZATION NODE HEADER FILE IN ROS

The *Localization_node* in ROS used the header file provided below to detect landmarks, generate local ego-points for the landmarks and match the ego-points in the *a priori* environment. It also localized the vehicle and determined its pose in the *a priori* environment. The vehicle's coordinates and pose were then published to the *currentPosition* topic to be used for various navigational purposes. The main code is contained in the *scanCallback()* callback function.

```
#pragma once
#include "Naes.h"
#include "localization.h"

#ifndef SCANNED_H
#define SCANNED_H

class Scanned {
        public:

                Scanned(){
                        /*PUBLICATIONS*/
                        currentPositionPub = nh.advertise<naes::current_position>("/currentPosition", 1);
                        stopPub = nh.advertise<naes::stop>("stop", 1);

                        /*SUBSCRIPTIONS*/
                        scanSub = nh.subscribe<sensor_msgs::LaserScan>("/scan", 2, &Scanned::scanCallback, this);}


                void scanCallback(const sensor_msgs::LaserScan::ConstPtr& scan){
                        ego_points.clear();
                        ego.clear();
                        ego.resize(0);
                        set_ego_size(scan->ranges.size());
                        set_min_angle(scan->angle_min);
                        set_angle_increment(scan->angle_increment);
                        set_ego_points(scan->ranges);

                        for(i=0; i < ego_size; i++){
                                if(ego_points[i] < reject){ // Needs to be put in a group.
                                        angle = angle_min + (i * angle_increment);
                                        x = calcx(ego_points[i], angle);
                                        y = calcy(ego_points[i], angle);
                                        if(ego.size() == 0){ //This is the first group
                                                ego.resize(1);
                                                ego[0].pushx(x);
                                                ego[0].pushy(y);}
                                        else{
                                                size = ego.size();
                                                grouped = false;
                                                for(j=0; j<size; j++){
                                                        test_x = ego[j].get_localization_x();
                                                        test_y = ego[j].get_localization_y();
                                                        abs_diff_x = abs(x-test_x);
                                                        abs_diff_y = abs(y-test_y);
                                                        if((abs_diff_x <= grouping_constant)&&(abs_diff_y <= grouping_constant)){
                                                                ego[j].pushx(x);
                                                                ego[j].pushy(y);
                                                                grouped = true;}
                                                }
                                                if(grouped == false){ //need to make a new object in ego vector.
                                                        ego.resize(size+1);
                                                        ego[size].pushx(x);
                                                        ego[size].pushy(y);}
                                        }
                                }
                        }
                        o = ego.size();
                        if(o == 3){
                                toAverageX.clear();
                                toAverageY.clear();
                                for(k = 0; k < o; k++){
                                        toAverageX = ego[k].get_x_vector();
                                        toAverageY = ego[k].get_y_vector();
                                        m = toAverageX.size();
                                        n = toAverageY.size();
                                        xsum = 0.0;
                                        ysum = 0.0;
                                        for(kk = 0; kk < m; kk++){
                                                xsum = xsum + toAverageX[kk];}
                                        for(ll = 0; ll < n; ll++){
                                                ysum = ysum + toAverageY[ll];}
                                        x_final = xsum / m;
                                        y_final = ysum / n;
                                        phi_final = calc_average_phi(x_final, y_final);
                                        ego[k].clearxego();
                                        ego[k].clearyego();
```

```
                ego[k].clearphiego();
                ego[k].pushy(y_final);
                ego[k].pushx(x_final);
                ego[k].pushphi(phi_final);
                toAverageX.clear();
                toAverageY.clear();
                toAverageX.resize(1);
                toAverageY.resize(1);
        }
        //These are to now be rotated to appropriate ego_positions.
        x_0 = ego[0].get_localization_x();
        y_0 = ego[0].get_localization_y();
        x_1 = ego[1].get_localization_x();
        y_1 = ego[1].get_localization_y();
        x_2 = ego[2].get_localization_x();
        y_2 = ego[2].get_localization_y();

        /***************
        ROTATION SECTION
        ***************/

        rotation = false;
        bad_rotation = false;
        rotate_count = 0;
        bad_reading_count = 0;
        while(rotation == false){
                bad_reading_count = 0;
                phi_0 = calc_phi(x_0, y_0);
                phi_1 = calc_phi(x_1, y_1);
                phi_2 = calc_phi(x_2, y_2);
                local_dist01 = sqrt(((x_1 - x_0)*(x_1 - x_0))+((y_1 - y_0)*(y_1 - y_0)));
                local_dist02 = sqrt(((x_2 - x_0)*(x_2 - x_0))+((y_2 - y_0)*(y_2 - y_0)));
                local_dist12 = sqrt(((x_2 - x_1)*(x_2 - x_1))+((y_2 - y_1)*(y_2 - y_1)));
                delta_dist01 = abs(dist01 - local_dist01);
                delta_dist02 = abs(dist02 - local_dist02);
                delta_dist12 = abs(dist12 - local_dist12);
                if(rotate_count > 5){
                        rotation = true;
                        cout << "TOO MANY ROTATIONS!!!!!!!" << endl;
                        stop.stop = "stop";
                        stopPub.publish(stop);}

                if((abs(dist01 - local_dist01)) >= offset){
                        bad_reading_count = bad_reading_count + 1;}
                if((abs(dist02 - local_dist02)) >= offset){
                        bad_reading_count = bad_reading_count + 1;}
                if((abs(dist12 - local_dist12)) >= offset){
                        bad_reading_count = bad_reading_count + 1;}

                if(bad_reading_count > 1){
                        rotate();
                        rotate_count = rotate_count + 1;}

                if(bad_reading_count == 1){
                        bad_rotation = true;
                        rotation = true;}

                if(bad_reading_count == 0){
                        rotation = true;}

        }

        if(bad_rotation == false){      //NEED TO PERFORM TRIANGULATION AND PUBLISH HERE.
                r_0 = calc_radius(x_0, y_0);
                r_1 = calc_radius(x_1, y_1);
                r_2 = calc_radius(x_2, y_2);
/*****************************************************************************************************************************************************/

                dx = x__1 - x__0;
                dy = y__1 - y__0;

                //* Determine the straight?line distance between the centers. */
                d = sqrt((dy*dy) + (dx*dx));

                /* Check for solvability. need to add a buffer */
                if (d > (r_0 + r_1 + 0.05)){
                        /* no solution. circles do not intersect. */
                        cout << "no solution, circles do not intersect" << endl;}


                else if (d < abs(r_0 - r_1)){
                        /* no solution. one circle is contained in the other */
                        cout << "not solution, one circle is contained in the other" << endl;}
                else{ //SOLVABILITY ELSE STATEMENT
                        /* 'point 2' is the point where the line through the circle
                        * intersection points crosses the line between the circle
                        * centers.
                        */

                        /* Determine the distance from point 0 to point 2. */
                        a = ((r_0*r_0) - (r_1*r_1) + (d*d)) / (2.0 * d) ;

                        /* Determine the coordinates of point 2. */
                        point2_x = x__0 + (dx * a/d);
                        point2_y = y__0 + (dy * a/d);

                        /* Determine the distance from point 2 to either of the
                        * intersection points.
                        */
                        h = sqrt((r_0*r_0) - (a*a));

                        /* Now determine the offsets of the intersection points from
                        * point 2.
                        */
                        rx = -dy * (h/d);
                        ry = dx * (h/d);

                        /* Determine the absolute intersection points. */
                        intersectionPoint1_x = point2_x + rx;
                        intersectionPoint2_x = point2_x - rx;
                        intersectionPoint1_y = point2_y + ry;
                        intersectionPoint2_y = point2_y - ry;
```

```
/* Lets determine if circle 3 intersects at either of the above intersection points.
*/
dx = intersectionPoint1_x - x__2;
dy = intersectionPoint1_y - y__2;
d1 = sqrt((dy*dy) + (dx*dx));
dx = intersectionPoint2_x - x__2;
dy = intersectionPoint2_y - y__2;
d2 = sqrt((dy*dy) + (dx*dx));

//HERE THE NEXX AND NEWY ARE FOUND

if(abs(d1 - r_2) < 0.1) {
        newx = intersectionPoint1_x;
        newy = intersectionPoint1_y;}
else if(abs(d2 - r_2) < 0.1) {
        newx = intersectionPoint2_x;
        newy = intersectionPoint2_y;}
else {
        cout << "INTERSECTION Circle1 AND Circle2 AND Circle3: NONE" << endl;
}


//HERE THE POSE IS FOUND
if(newx >= x__0) {
        if(newy >= y__0) {//beta is in third quadrant
                beta0 = atan((y__0-newy)/(x__0-newx)) - (M_PI);
                if( ((M_PI)/2) + M_PI + beta0 + (M_PI - phi_0) < M_PI) {
                        phi0 = ((M_PI)/2) + M_PI + beta0 + (M_PI - phi_0);}
                else{
                        phi0 = ((M_PI)/2) + (M_PI + beta0) + (-M_PI - phi_0);}
        }
        else {//beta is in second quadrant
                beta0 = (M_PI)+ atan((y__0-newy)/(x__0-newx));
                if((beta0+((M_PI/2)-phi_0)) < M_PI) {
                        phi0 = beta0+(M_PI/2)-phi_0;}
                else {
                        phi0 = -M_PI - (M_PI - beta0) + (M_PI/2)-phi_0;}
        }
}
else {//beta is in first or fourth quadrant
        beta0 = atan((y__0-newy)/(x__0-newx));
        if(((-M_PI)/2) + beta0 + (M_PI - phi_0) < M_PI) {
                phi0 = ((-M_PI)/2) + beta0 + (M_PI - phi_0);}
        else {
                phi0 = ((-M_PI)/2) + beta0 + (-M_PI - phi_0);}
}
if(newx >= x__1) {
        if(newy >= y__1) {//beta is in third quadrant
                beta1 = atan((y__1-newy)/(x__1-newx)) - (M_PI);
                if( ((M_PI)/2) + M_PI + beta1 + (M_PI - phi_1) < M_PI) {
                        phi1 = ((M_PI)/2) + M_PI + beta1 + (M_PI - phi_1);}
                else{
                        phi1 = ((M_PI)/2) + (M_PI + beta1) + (-M_PI - phi_1);}
        }
        else {//beta is in second quadrant
                beta1 = (M_PI)+ atan((y__1-newy)/(x__1-newx));
                if((beta1+((M_PI/2)-phi_1)) < M_PI) {
                        phi1 = beta1+(M_PI/2)-phi_1;}
                else {
                        phi1 = -M_PI - (M_PI - beta1) + (M_PI/2)-phi_1;}
        }
}
else {//beta is in first or fourth quadrant
        beta1 = atan((y__1-newy)/(x__1-newx));
        if( ((-M_PI)/2) + beta1 + (M_PI - phi_1) < M_PI) {
                phi1 = ((-M_PI)/2) + beta1 + (M_PI - phi_1);}
        else {
                phi1 = ((-M_PI)/2) + beta1 + (-M_PI - phi_1);}
}
if(newx >= x__2) {
        if(newy >= y__2) {//beta is in third quadrant
                beta2 = atan((y__2-newy)/(x__2-newx)) - (M_PI);
                if( ((M_PI)/2) + M_PI + beta2 + (M_PI - phi_2) < M_PI) {
                        phi2 = ((M_PI)/2) + M_PI + beta2 + (M_PI - phi_2);}
                else{
                        phi2 = ((M_PI)/2) + (M_PI + beta2) + (-M_PI - phi_2);}
        }
        else {//beta is in second quadrant
```

```
                beta2 = (M_PI)+ atan((y__2-newy)/(x__2-newx));
                if((beta2+((M_PI/2)-phi_2)) < M_PI) {
                        phi2 = beta2+(M_PI/2)-phi_2;}
                else {
                        phi2 = -M_PI - (M_PI - beta2) + (M_PI/2)-phi_2;}
        }
}
else {//beta is in first or fourth quadrant
        beta2 = atan((y__2-newy)/(x__2-newx));
        if( ((-M_PI)/2) + beta2 + (M_PI - phi_2) < M_PI) {
                phi2 = ((-M_PI)/2) + beta2 + (M_PI - phi_2);}
        else {
                phi2 = ((-M_PI)/2) + beta2 + (-M_PI - phi_2);}
}
if( (abs(phi0 - phi1) > 1) || (abs(phi1 -phi2) > 1) ) { //crossing over the pi mark
        if(phi0 < 0) {
                //cout << "phi0 crossed over the pi boundary. added 2*PI" << endl;
                phi0 = phi0 +(2* M_PI);}
        if(phi1 < 0) {
                //cout << "phi1 crossed over the pi boundary. added 2*PI" << endl;
                phi1 = phi1 +(2* M_PI);}
        if(phi2 < 0) {
                //cout << "phi2 crossed over the pi boundary. added 2*PI" << endl;
                phi2 = phi2 +(2* M_PI);}
        phiavg = (phi0 + phi1 + phi2) / 3;
        if(phiavg >= M_PI) {
                phiavg = phiavg - (2* M_PI);}
        newphi = phiavg;
}
else {
        newphi = (phi0 + phi1 + phi2) / 3;}
coordinates.x = newx;
coordinates.y = newy;
coordinates.phi = newphi;
currentPositionPub.publish(coordinates);
```

```
                        } //end of solvability else statement
                } //end of if(bad_rotation == false). current position should be published by now.
                else{
//                      cout << "bad_rotation == true so there is no published current position..." << endl;
                }
        } //end of if(o == 3);
        else{
                cout << "Number of Ego-Points detectedwas not 3, it was: " << o <<endl;}
}//End of scanCallback


//Mutator Functions

void set_ego_size(int value){
        ego_size = value;
        ego_points.resize(value);}
void set_min_angle(float value){
        angle_min = value;}
void set_angle_increment(float value){
        angle_increment = value;}
void set_ego_points(vector<float> points){
        ego_points = points;}
void set_reject_value(float value){
        reject = value;}
void set_grouping_constant_value(float value){
        grouping_constant = value;}

float calcx(float rangevalue, float anglevalue){
        float xvalue = rangevalue * cos(anglevalue);
        return xvalue;}

float calcy(float rangevalue, float anglevalue){
        float yvalue = rangevalue * sin(anglevalue);
        return yvalue;}

float calc_average_phi(float xx, float yy){
        if(xx < 0 && yy > 0){// need to add pi
                average_phi = atan( (yy/xx) ) + M_PI;}
        else if( xx < 0 && yy <= 0){// need to subtract pi
                average_phi = atan( (yy/xx) ) - M_PI;}
        else{// regular tan inverse
                average_phi = atan( (yy/xx) );}
        return average_phi;
}


//Accessor Functions
float get_current_phi(){
        return newphi;}

/***************
rotation section
***************/
void set_global_ego_points(float xx0, float yy0, float xx1, float yy1, float xx2, float yy2){
        x__0 = xx0;
        y__0 = yy0;
        x__1 = xx1;
        y__1 = yy1;
        x__2 = xx2;
        y__2 = yy2;

        dist01 = sqrt(((x__1 - x__0)*(x__1 - x__0))+((y__1 - y__0)*(y__1 - y__0)));
        dist02 = sqrt(((x__2 - x__0)*(x__2 - x__0))+((y__2 - y__0)*(y__2 - y__0)));
        dist12 = sqrt(((x__2 - x__1)*(x__2 - x__1))+((y__2 - y__1)*(y__2 - y__1)));}

void set_offset(float value){
        offset = value;}

void rotate(){
        grabx0 = x_0;
        graby0 = y_0;
        grabx1 = x_1;
        graby1 = y_1;
        grabx2 = x_2;
        graby2 = y_2;

        x_0 = grabx1;
        y_0 = graby1;
```

```
        x_1 = grabx2;
        y_1 = graby2;
        x_2 = grabx0;
        y_2 = graby0;}

float calc_phi(float xx, float yy){
        if(xx < 0 && yy > 0){// need to add pi
                phi = atan( (yy/xx) ) + M_PI;}
        else if( xx < 0 && yy <= 0){// need to subtract pi
                phi = atan( (yy/xx) ) - M_PI;}
        else{// regular tan inverse
                phi = atan( (yy/xx) );}
        return phi;
}

float calc_radius(float xx, float yy){
        radius = sqrt( (xx * xx) + (yy * yy) );
        return radius;}

/***************
triangulation section
***************/
//no functions to put here
```

```
        private:
                ros::NodeHandle nh;
                ros::Publisher currentPositionPub;
                ros::Publisher stopPub;
                ros::Subscriber scanSub;
                naes::stop stop;
                naes::current_position coordinates;

                //Member Variables
                /************************
                SCAN SECTION
                ************************/
                vector<Localization> ego;
                vector<float> ego_points, toAverageX, toAverageY;
                float angle_increment, angle_min, reject, angle, grouping_constant;
                float x, y, test_x, test_y, abs_diff_x, abs_diff_y;
                float xsum, ysum, x_final, y_final, phi_final, average_phi;
                float x_0, y_0, phi_0, x_1, y_1, phi_1, x_2, y_2, phi_2;
                float x__0, y__0, x__1, y__1, x__2, y__2;
                int ego_size, size, scan_count;
                int i, j, m, n, o, k, kk, ll;
                bool grouped;

                /************************
                ROTATION SECTION
                ************************/
                float dist01, dist02, dist12, local_dist01, local_dist02, local_dist12, delta_dist01, delta_dist02, delta_dist12;;
                float offset, phi, radius;
                float grabx0, graby0, grabx1, graby1, grabx2, graby2;
                float r_0, r_1, r_2;
                int rotate_count, bad_reading_count;
                bool rotation, bad_rotation;

                /********************
                TRIANGULATION SECTION
                ********************/
                float newx, newy, newphi, phiavg;
                float beta0, beta1, beta2, phi0, phi1, phi2;
                float a, dx, dy, d, h, rx, ry, d1, d2;
                float point2_x, point2_y;
                float intersectionPoint1_x, intersectionPoint2_x, intersectionPoint1_y, intersectionPoint2_y;

};
//Scanned::~Scanned(){}


#endif
```

APPENDIX C

ADJUSTMENT NODE HEADER FILE IN ROS

The *adjustment_node* was used after the discretized baseline path was generated from the baseline runs. This node obtained its current location from the *currentPosition* topic and ran a callback function that made wheel adjustments in order to get to the next baseline target point in the baseline path. Once the point was reached, it moved on to making wheel adjustments to get to the next baseline path target point. This process continued until all baseline path points were autonomously navigated to. The header file that contained the callback function is provided below:

```
#pragma once
#include "Naes.h"

#ifndef ADJUSTMENT_H
#define ADJUSTMENT_H

class Adjustment {
        public:

                Adjustment(){
                        /*PUBLICATIONS*/
                        motorsPub = nh.advertise<naes::wheel_speeds>("ROStoArduino", 10);
                        stopPub = nh.advertise<naes::stop>("stop", 1);

                        /*SUBSCRIPTIONS*/
                        currentLocationSub = nh.subscribe<naes::current_position>("/currentPosition", 2, &Adjustment::adjustmentCallback, this);}


                void publish_brakes(){
                        motorsPub.publish(brakes);}
                void publish_forward(){
                        motorsPub.publish(forward);
                        left_speed = normal;
                        right_speed = normal;}
                void publish_rotate_left(){
                        motorsPub.publish(rotate_left);
                        static_turn = true;}
                void publish_rotate_right(){
                        motorsPub.publish(rotate_right);
                        static_turn = true;}
                void come_out_of_static_turn(){
                        motorsPub.publish(forward);
                        left_speed = normal;
                        right_speed = normal;
                        static_turn = false;}
                void publish_custom(){
                        custom.left_wheel = left_speed;
                        custom.right_wheel = right_speed;
                        motorsPub.publish(custom);}
                void make_left_adjustment(){
                        if(count < wheel_change){
                                count = count + 1;}
                        else{
                                count = 0;
                                left_speed = left_speed - adjustment_constant;
                                right_speed = right_speed + adjustment_constant;
                                publish_custom();}
                }
                void make_right_adjustment(){
                        if(count < wheel_change){
                                count = count + 1;}
                        else{
                                count = 0;
                                left_speed = left_speed + adjustment_constant;
                                right_speed = right_speed - adjustment_constant;
                                publish_custom();}
                }
                void publish_origin_turn(){
                        left_speed = 20;
                        right_speed = -20;
                        publish_custom();}
```

```
void check_left_adjustment(){
        ratio = future_offset / distance;
        left_speed = normal - (ratio * adjustment_constant);
        if(left_speed < 10){
                left_speed = 10;}
        right_speed = normal + (normal - left_speed);
        publish_custom();}

void check_right_adjustment(){
        ratio = future_offset / distance;
        cout << "ADJUSTING RIGHT...RATIO IS: " << ratio << endl;
        left_speed = normal + (ratio * adjustment_constant);
                if(left_speed > 90){
                        left_speed = 90;}
        right_speed = normal + (normal - left_speed);
        publish_custom();}


void calculate_delta_phi_and_offset(){
        if( (current_phi > 0) && ((current_phi - phi_to_next) > M_PI)) { //going across border going left
                adjusted_current_phi = current_phi;
                adjusted_phi_to_next = phi_to_next + (2*M_PI);}
        else if( (current_phi <= 0) && ((phi_to_next - current_phi) > M_PI) ){ //going across border going right
                adjusted_phi_to_next = phi_to_next;
                adjusted_current_phi = current_phi + (2*M_PI);}
        else { //not going across border
                adjusted_phi_to_next = phi_to_next;
                adjusted_current_phi = current_phi;}
        delta_phi = adjusted_phi_to_next - adjusted_current_phi;
        future_offset = abs( distance * sin(delta_phi));}


void currentPositionCallback(const naes::current_position::ConstPtr& msg1) {
        current_x1 = msg1->x;
        current_y1 = msg1->y;
        current_phi1 = msg1->phi;}

float get_current_phi1(){
        return current_phi1;}

/* SET FUNCTIONS FOR NODE CONSTANTS */
void set_max_offset(float offsetvalue){
        max_offset = offsetvalue;}
void set_adjustment_constant(int adjustmentconstval){
        adjustment_constant = adjustmentconstval;}
void set_desired_radius(float radiusvalue){
        desired_radius = radiusvalue;}
void set_turn_value(float turnvalue){
        turn = turnvalue;}
void set_iteration_to_zero(){
        iteration = 0;
        count = 0;}
void set_static_turn_to_false(){
        static_turn = false;}
void set_origin_state(bool value){
        origin = value;}
void set_origin_rotate_value(float rvalue){
        origin_rotate_value = rvalue;}
void set_wheel_change_value(int wvalue){
        wheel_change = wvalue;}
void set_normal_wheel_value(int normwheelval){
        normal = normwheelval;
        forward.left_wheel = normwheelval;
        forward.right_wheel = normwheelval;
        rotate_left.left_wheel = -normwheelval;
        rotate_left.right_wheel = normwheelval;
        rotate_right.left_wheel = normwheelval;
        rotate_right.right_wheel = -normwheelval;
        brakes.left_wheel = 0;
        brakes.right_wheel = 0;
        check = 34;
        check_count = 0;}

/* PUSHING BAG FILE INFORMATIONS */
void pushx(float x){
        plotted_xs.push_back(x);}
void pushy(float y){
        plotted_ys.push_back(y);}


void pushphi(float angle){
        plotted_phis.push_back(angle);}
void set_navigation_size(){
        size = plotted_xs.size();}


void adjustmentCallback(const naes::current_position::ConstPtr& msg){
                current_x = msg->x;
                current_y = msg->y;
                current_phi = msg->phi;
        if( iteration < size ){
                next_x = plotted_xs[iteration];
                next_y = plotted_ys[iteration];
                next_phi = plotted_phis[iteration];

                distance = sqrt( (next_x - current_x)*(next_x - current_x)) + ( (next_y - current_y)*(next_y - current_y)));
                if( (next_x < current_x) && (next_y > current_y) ){                     //second quadrant
                        phi_to_next = atan((next_y - current_y)/(next_x - current_x)) + M_PI;}
                else if( (next_x < current_x) && (next_y <= current_y) ){               //third quadrant
                        phi_to_next = atan((next_y - current_y)/(next_x - current_x)) - M_PI;}
                else {                                                                  //first or fourth quadrant
                        phi_to_next = atan((next_y - current_y)/(next_x - current_x));}
                if( distance <= desired_radius){        //It has reached it's desired destination, move to next desired point
                        if(check != 0){
                                check_count = 1;
                                check = 0;
                                iteration = iteration + 1;}
                        else{
                        check_count = check_count + 1;
                        iteration = iteration + 1;}
                }
                else{
                        calculate_delta_phi_and_offset();
                        if( current_phi > 0 ) {
                                if( (current_phi - phi_to_next) > M_PI) { //go accross border going left
                                        if(delta_phi > turn) { //need to rotate left
                                                if(check != 1){
                                                        check_count = 1;
                                                        check = 1;
                                                        publish_rotate_left();}
```

```
                                        else{
                                                check_count = check_count + 1;
                                                publish_rotate_left();}
                                        }
                                        else { //continue path or make wheel adjustments
                                                if(static_turn == true) { //If it just came out of rotation state, go straight
                                                        cout << "RESULT 2(static turn) CALLED" << endl;
                                                        come_out_of_static_turn();}
                                                else{
                                                        if(check != 3){
                                                                check_count = 1;
                                                                check = 3;
                                                                check_left_adjustment();}
                                                        else{
                                                                check_count = check_count + 1;
                                                                check_left_adjustment();}
                                                }
                                        }
                                }
                                else { //doesnt need to go across the border
                                        if(delta_phi > turn) { //need to rotate left
                                                if(check != 4){
                                                        check_count = 1;
                                                        check = 4;
                                                        publish_rotate_left();}
                                                else{
                                                        check_count = check_count + 1;
                                                        publish_rotate_left();}
                                        }



                                        else if(delta_phi < (-1*turn)) { //need to rotate right
                                                if(check != 5){
                                                        check_count = 1;
                                                        check = 5;
                                                        publish_rotate_right();}
                                                else{
                                                        check_count = check_count + 1;
                                                        publish_rotate_right();}
                                        }



                                        else { //continue path or make wheel adjustments
                                                if(static_turn == true) {
                                                        come_out_of_static_turn();}
                                                else{
                                                        if(delta_phi > 0){//need to check adjustment left
                                                                if(check != 7){
                                                                        check_count = 1;
                                                                        check = 7;
                                                                        check_left_adjustment();}
                                                                else{
                                                                        check_count = check_count + 1;
                                                                        check_left_adjustment();}
                                                        }
                                                        else{
                                                                if(check != 8){
                                                                        check_count = 1;
                                                                        check = 8;
                                                                        check_right_adjustment();}
                                                                else{
                                                                        check_count = check_count + 1;
                                                                        check_right_adjustment();}
                                                        }
                                                }

                                        }
                                }
                        }
                        else { //current_phi <= 0
                                if( (phi_to_next - current_phi) > M_PI) { //go accross border going right
                                        if(delta_phi < (-1*turn)) { //need to rotate right
                                                if(check != 9){
                                                        check_count = 1;
```

```
                                                        check = 9;
                                                        publish_rotate_right();}
                                                else{
                                                        check_count = check_count + 1;
                                                        publish_rotate_right();}
                                        }



                                        else { //continue path or make wheel adjustments
                                                if(static_turn == true) {
                                                        come_out_of_static_turn();}
                                                else{
                                                        if(check != 11){
                                                                check_count = 1;
                                                                check = 11;
                                                                check_right_adjustment();}
                                                        else{
                                                                check_count = check_count + 1;
                                                                check_right_adjustment();}
                                                }
                                        }
                                }
                                else { //doesnt need to go across the border
                                        if(delta_phi > turn) { //need to rotate left
                                                if(check != 12){
                                                        check_count = 1;
                                                        check = 12;
                                                        publish_rotate_left();}
                                                else{
                                                        check_count = check_count +1;
                                                        publish_rotate_left();}

                                        }
                                        else if(delta_phi < (-1*turn)) { //need to rotate right
                                                if(check != 13){
                                                        check_count = 1;
                                                        check = 13;
                                                        publish_rotate_right();}
                                                else{
                                                        check_count = check_count +1;
                                                        publish_rotate_right();}
```

```
                                                else { //continue path or make wheel adjustments
                                                        if(static_turn == true) {
                                                                come_out_of_static_turn();}
                                                        else{
                                                                if(delta_phi > 0){
                                                                        if(check != 15){
                                                                                check_count = 1;
                                                                                check = 15;
                                                                                check_left_adjustment();}
                                                                        else{
                                                                                check_count = check_count +1;
                                                                                check_left_adjustment();}
                                                                }
                                                                else{
                                                                        if(check != 16){
                                                                                check_count = 1;
                                                                                check = 16;
                                                                                check_right_adjustment();}
                                                                        else{
                                                                                check_count = check_count +1;
                                                                                check_right_adjustment();}
                                                                }
                                                        }
                                                }
                                        }
                                }
                        }
                        else{ //at the last point
                                if(origin == false){
                                        motorsPub.publish(brakes);
                                        cout << "THE ROBOT HAS FINISHED NAVIGATION!!!" << endl;
                                        stopmsg.stop = "stop";
                                        stopPub.publish(stopmsg);}
                                if(origin == true){
                                        origin_check = abs(current_phi - (M_PI/2));
                                        cout << "origin_check is: " << origin_check << endl;
                                        if(origin_check >= origin_rotate_value){
                                                publish_origin_turn();}
                                        else{
                                                motorsPub.publish(brakes);
                                                stopmsg.stop = "stop";
                                                stopPub.publish(stopmsg);}
                                }
                        }
                }

        private:
                ros::NodeHandle nh;
                ros::Publisher motorsPub;
                ros::Publisher stopPub;
                ros::Subscriber currentLocationSub;
                naes::wheel_speeds motors, brakes, forward, rotate_left, rotate_right, custom;
                naes::stop stopmsg;

                //Member Variables
                vector<float> plotted_xs;
                vector<float> plotted_ys;
                vector<float> plotted_phis;
                float current_x, current_y, current_phi, next_x, next_y, next_phi;
                float max_offset;
                float desired_radius;
                float turn;
                float distance;
                float phi_to_next;
                float adjusted_current_phi, adjusted_phi_to_next, delta_phi, future_offset;
                float current_x1, current_y1, current_phi1;
                float origin_rotate_value;
                float origin_check;
                float ratio;
                int check, check_count;
                int adjustment_constant;
                int iteration;
                int normal;
                int size;
                int left_speed, right_speed;
                int wheel_change;
                int count;


                bool origin;
                bool static_turn;

};
#endif
```

APPENDIX D

ARDUINO CODE FOR MOTOR CONTROLLER

The Arduino microcontroller used in this investigation controlled the wheel speeds of the intelligent vehicle. The Arduino contained a header file that allowed it to run ROS callback functions through serial communication with ROS. The Arduino was subscribed to the *ROStoArduino* topic and ran a callback function that extracted the published wheel values from the topic and applied them to the electric motors on the intelligent vehicle. The Arduino code is provided below:

```
#include <Servo.h>
#include <ros.h>
#include <std_msgs/Empty.h>
#include <std_msgs/String.h>
#include <motor_driver/Motor_speeds.h>

Servo leftmotor;
Servo rightmotor;

ros::NodeHandle  nh;

void messageCb( const motor_driver::Motor_speeds& msg){
                                  //void mesageCb( const message_type any_name)
  String A = msg.data;
  int comma_index = A.indexOf(',');
  int length =A.length();
  String left_motor_speed = A.substring(0, comma_index);
  String right_motor_speed = A.substring(comma_index+1, length);
  int L = left_motor_speed.toInt();
  int R = right_motor_speed.toInt();
  leftmotor.write(L);
  rightmotor.write(R);

}

ros::Subscriber<motor_driver::Motor_speeds> sub("ROStoArduino", &messageCb );

void setup()
{
  leftmotor.attach(6);
  rightmotor.attach(10);
  nh.initNode();

}

void loop()
{
  nh.subscribe(sub);
  nh.spinOnce();
}
```