Tabish Badar

Implementation of the autonomous functionalities on an electric vehicle platform for research and education

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Pakistan 20.May.2019

Thesis supervisor:

Prof. Arto Visala

Thesis advisor:

D.Sc. (Tech.) Mika Vainio

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Tabish Badar

Title: Implementation of the autonomous functionalities on an electric vehicle platform for research and education

Date: 20.May.2019 Language: English Number of pages:10+87

Department of Electrical Engineering and Automation

Professorship: Automation Technology Code: AS-84

Supervisor: Prof. Arto Visala

Advisor: D.Sc. (Tech.) Mika Vainio

Self-driving cars have recently captured the attention of researchers and car manufacturing markets. Depending upon the level of autonomy, the cars are made capable of traversing from one point to another autonomously. In order to achieve this, sophisticated sensors need to be utilized. A complex set of algorithms is required to use the sensors data in order to navigate the vehicle along the desired trajectory.

Polaris is an electric vehicle platform provided for research and education purposes at Aalto University. The primary focus of the thesis was to utilize all the sensors provided in Polaris to their full potential. So that, essential data from each sensor is made available to be further utilized either by a specific automation algorithm or by some mapping routine.

For any autonomous robotic system, the first step towards automation is localization. That is to determine the current position of the robot in a given environment. Different sensors mounted over the platform provide such measurements in different frames of reference. The thesis utilizes the GPS based localization solution combined with the LiDAR data and wheel odometry to perform autonomous tasks. Robot Operating System is used as the software development tool in thesis work.

Autonomous tasks include the determination of the global as well as the local trajectories. The endpoints of the global trajectories are dictated by the set of predefined GPS waypoints. This is called target-point navigation. A path needs to be planned that avoids all the obstacles. Based on the planned path, a set of velocity commands are issued by the embedded controller. The velocity commands are then fed to the actuators to move the vehicle along the planned trajectory.

Keywords: robot operating system, waypoint navigation, estimation and localization, sensor fusion and integration, autonomous control, self-driving cars

AALTO-YLIOPISTO
SÄHKOTEKNIIKAN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Tabish Badar

Työn nimi: Itsenäisten toimintojen toteuttaminen sähköajoneuvojen alustalla tutkimukseen ja koulutukseen

Päivämäärä: 20.May.2019 Kieli: Englanti Sivumäärä:10+87

Sähkötekniikan ja automaation laitos

Professuuri: Automaatiotekniikka Koodi: AS-84

Valvoja: Prof. Arto Visala

Ohjaaja: TkT Mika Vainio

Itsekseen ajamaan kykenevät autot ovat olleet tutkijoiden ja autonvalmistajien huomion keskipisteessä erityisesti viime vuosina. Autonomian tasosta riippuen ne ovat kyenneet liikkumaan, enemmän tai vähemmän, itsenäisesti paikasta toiseen ilman ihmisen jatkuvaa ohjaamista. Jotta tämä on saatu toteutettua, ajoneuvot on täytynyt varustaa monipuolisilla anturijärjestelmillä. Antureiden lisäksi tarvitaan luonnollisesti kehittyneitä algoritmeja käsittelemään ja hyödyntämään niiltä tulevaa dataa, niin että ajoneuvo saadaan liikkumaan turvallisesti haluttua reittiä pitkin.

Työssä käytetty Polaris Ranger on rinnakkain istuttava sähköistetty mönkijä (e-ATV), joka on hankittu Aalto yliopistoon tutkimus- ja koulutustarkoituksiin ja sittemmin kalustettu erilaisilla antureilla ja toimilaitteilla autonomisen ajamisen mahdollistamiseksi. Tämän diplomityön keskeinen tavoite on hyödyntää ajoneuvon antureiden tuottama anturitieto mahdollisimman hyvin erityisissä itsenäisen ajamisen mahdollistavissa algoritmeissa ja ympäristön kartoitusrutiineissa.

Toimiva autonominen robottijärjestelmä perustuu yleensä sen paikantamiseen työympäristössään. Erilaiset käytössä olevat anturit tuottavat tarvittavaa tietoa omissa koordinaatistoissaan. Työssä yhdistetään autonomisten tehtävien suorittamista varten GPS pohjainen paikkatieto ympäristön laserkeilaukseen (LiDAR) ja ajoneuvon renkailta ja ohjauksesta saataviin kuljetun matkan laskentaan tarvittaviin liiketietoihin maailmanlaajuisesti käytetyn robottien ohjelmistojen kehittämiseen ja avoimen koodin jakamiseen kehitetyn ROS (Robot Operating System) viitekehyksen avulla.

Yllä mainitut autonomiset tehtävät sisältävät tässä tapauksessa erityisesti ajoneuvon halutun liikeradan suunnittelun ja sen määrittämisen GPS-reittipisteiden avulla (target-point navigation). Suunnitellun polun pitää luonnollisesti kiertää kaikki tiedossa olevat esteet. Suunnitelman toteuttamiseksi sulautettu ohjain (embedded controller) antaa joukon nopeuskomentoja, jotka sitten syötetään moottoreihin ja ohjauslaitteisiin ajoneuvon liikuttamiseksi haluttua reittiä pitkin.

Avainsanat: ROS, reittipistenavigointi, estimointi ja paikannus, anturitietojen fuusiointi, itsenäinen ohjaus, itseajavat autot

Preface

Firstly, I would like to thank Professor Arto Visala, who provided me an opportunity to work on such a wonderful piece of equipment. Throughout the thesis work, I enjoyed working independently on the platform. This aided me a lot to grasp a thorough understanding of various systems of the vehicle under test. The professor was very kind and encouraged my methodologies to produce tangible results. The professor showed a great amount of confidence in my thesis process and allowed me to instruct a group of students on Project Work. Through this, I was able to contribute my bit to the research group by instructing younger students.

I am also very grateful towards my thesis advisor Mika Vainio for his guidance on this thesis, and for passing valuable suggestions to improve the structure and outlook of this thesis document. In particular, I wish to thank Heikki Hyyti and Andrei Sandru for their advisory conversations and practical help. Andrei was very supportive and helped me a lot in understanding the basic set up of the e-ATV.

Lastly, I would like to mention the unending and loving support of my parents, my wife, and my daughter. It is to them I owe all of my efforts.

Pakistan, 20.May.2019

Tabish Badar

Contents

List of Figures

List of ROS Configuration Files

Symbols, notations and abbreviations

Abbreviations

IMU	Inertial Measurement Unit
SLAM	Simultaneous Localization And Mapping
DOF	Degrees Of Freedom
EKF	Extended Kalman Filter
UKF	Unscented Kalman Filter
GPS	Global Positioning System
ROS	Robot Operating System
ABS	Automatic Braking System
SPAN	Synchronous Position, Attitude and Navigation
GNSS	Global Navigation Satellite System
DGNSS	Differential GNSS
RTCM	Radio Technical Commission for Maritime Services
e-ATV	Electric All-Terrain Vehicle
LiDAR	Light Detection and Ranging System
PLC	Programmable Logic Controller
LLA	Latitude, Longitude and Altitude
ECEF	Earth-Centered Earth-Fixed
UTM	Universal Transverse Mercator
MEMS	Micro-Electro-Mechanical Sensors
NLS	National Land Survey
WP	Waypoint
3D/2D	Three Dimensional/ Two Dimensional
fps	Frames per second
CAN	Controller Area Network

Symbols and notations

ϕ, θ, ψ	Roll, pitch and yaw angles respectively.
$\dot{\phi}, \dot{\theta}, \dot{\psi}$	Roll , pitch and yaw rates respectively.
x, y, z	Position along x-, y- and z- axis.
v_x, v_y, v_z	Linear velocities along x-, y- and z-axes respectively.
a_x, a_y, a_z	Linear acceleration along x-, y- and z-axes respectively.
R	Turn Radius
x_k	x-position at time instant k.
x_{k+1}	x-position at time instant k+1.

1 Introduction

1.1 Background

Autonomous vehicles, which have achieved outstanding developments in the past decade, are to become a reality on the roads in the near future. With tech giants such as Tesla, Google, Ford, and others investing largely in this area of technology, innovative challenges for the research community are still on the rise. Autonomous Systems Research Group at Aalto University wants likewise to contribute a fair share of research to this area.

Figure 1.1: Polaris: An electric vehicle platform inside Autonomous Systems Lab.

Electric All-Terrains-Vehicles (e-ATV) have found increasing use in forestry and surveillance. They are of great use to the rangers and forest personnel who have to work in an unfamiliar terrain [1]. In recent times, the importance for the development of e-ATVs, that are capable of traversing through unacquainted terrains in rough weather conditions, have gained significant importance. These are electric machines with zero emissions, less noise and are economical to operate. Thus far, humans have driven these vehicles, but the latest technology has made it possible to implement self-driving and autonomous capabilities in such vehicles.

Thus, this thesis aims at implementing autonomous capabilities for an electric ground vehicle platform shown in Figure 1.1. It is called Polaris Ranger [55], which is an e-ATV under development at Aalto University to become a self-driving ground vehicle. Polaris will serve both as research and as a teaching platform in the future. The platform is already equipped with a state-of-the-art set of sensors, actuators, and electric modules. These include Automatic Braking System (ABS), Programmable Logic Controller (PLC), embedded computer, wheel encoders, power steering control system, speed control system, Synchronous Position, Attitude and Navigation (SPAN) unit, Light Detection and Ranging (LiDAR) equipment along with an omnidirectional camera.

At Autonomous Systems Lab, many researchers and students have contributed a significant amount of work to this project so far. Firstly, ABS was installed [2]. This was to make sure the safety of the driver as well as to ensure the road safety of the vehicle in a high wheel slipping conditions. Subsequently, two low-level EPEC-5050 PLC based control units were installed on Polaris. The purpose of these sub-systems is to control the steering and forward motion of the vehicle. Besides, it keeps track of the counts the wheel has rotated around its center and what is the current steering angle. Software tools such as CODESYS were utilized in order to program PLCs.

Having set a tangible basic actuator system for the Polaris, the next task was to install all available sensors. A group of students achieved basic hardware integration [3]. They were also responsible for setting the Robot Operating System (ROS) as the software integration platform for the Polaris Ranger. ROS framework was selected because of its popularity among the researchers in the robotics community. The same group did testing and calibration of SPAN, LiDAR and omnidirectional camera. The capability to capture a panoramic image was also successfully tested.

With all sensors providing necessary data within the ROS framework, the subsequent task was to utilize this data for localization and mapping of Polaris in a dynamic environment. The initial setup was laid by the group of students in [4]. A perception platform for the Polaris was evaluated in ROS. The mapping of the environment was conducted effectively using LiDAR data only. Although, robot localization problem was not handled effectively, a (deterministic) kinematics model of the platform was implemented using the dead reckoning method.

Having set the appropriate background about the project, the key objectives set for this thesis work are discussed in the next section.

1.2 Objectives

In order to achieve the autonomous capability, one needs a model of the robot under test (Polaris Ranger in this case), sensors (and fusion algorithms) to localize it and provide situational awareness (to understand what it sees). Furthermore, it is required to set appropriate control and navigation parameters based on a priori info about the environment and mission objectives. Mission objectives define a path that the robot must traverse (autonomous navigation) while at the same time avoid obstacles (both static and dynamic) through fast reaction and longer-term re-planning (if needed).

Therefore, the prime objective of the thesis work was to utilize all the sensors onboard up to their full potential; so that, indispensable data from each sensor is made available. Thus, every possible useable data from each of the available sensors were added in order to achieve the above-mentioned tasks.

The first task was to equip the SPAN unit with the differential corrections provided by the National Land Survey of Finland [6]. Such a Differential GNSS (DGNSS) based operation is necessary in navigating the vehicles with minimum position errors over the ground. It is vital for the safe operation of autonomous ground vehicles. The rover - a moving GPS user such as SPAN installed on Polaris - receives the corrections through a wireless internet connection according to Networked Transport of RTCM via Internet Protocol (Ntrip) protocol. These corrections were then routed through a piece of hardware to the SPAN unit.

Relevant parts of the Robot Operating System, being the main decision-making or so to say the intelligence part of the project, were greatly improved during the thesis process. C/C++ and Python languages were used extensively for the programming of various portions of the ROS. For this purpose, a thorough understanding of multifarious concepts associated with ROS was first acquired.

The robot localization problem was handled within the ROS framework. A great deal of effort was put to test this part of the thesis. Tests were carried out during snowy weather to check the effects of the wheel slip on the odometric calculations. The subsequent objective was to implement the navigation package provided by ROS. Fused odometry data is thus fed to the navigation algorithm which is planning a viable path for the mobile robot to traverse.

Navigation in ROS is accomplished by gathering the odometry data, the map of the environment and a 3D point cloud (output of LiDAR) of the surroundings. It is responsible for planning the global and local paths for the mobile robot. The output of the navigation module is a set of velocity commands. Therefore, the next objective of the thesis was to implement a mission controller, which was responsible for setting the target waypoints for the vehicle to traverse and

provide them as goals to the path planning routines.

The next job was to implement the motion controller. It translates the velocity commands provided by the mission controller to the format useable by two low-level PLCs. These PLCs are then responsible for steering the vehicle in a specific direction at the command velocities. Finally, autonomous capabilities added to an existing set of code were tested on the actual hardware.

In addition, a group of students who are developing a simulation platform for Polaris using ROS and Gazebo [24] was also instructed during the thesis process. This has indeed added a new dimension to the overall project as a simulation platform will reduce the further efforts required in testing the added algorithms.

In the next section, the basic structure of this thesis is presented.

1.3 Structure

This document comprises information about the significant characteristics of a self-driving electric vehicle under test. The particulars about the essential advancements made to the existing autonomous ground vehicle platform are presented. These developments are typically in the form of software modifications to the existing set of code.

In Chapter 1, the objectives for the thesis and the main accomplishments made during the process are presented on a general level. The actual big picture of the accomplishments made during the process is presented in Chapter 2. In that chapter, a presentation of each individual element of the big picture is given along with a concise theoretical background of the topic.

A brief but relevant discussion about ROS is presented in Chapter 3. This chapter includes relevant information about ROS. It will be vital in understanding the related parts of the thesis. The information includes, for example, some basic ROS concepts and data conventions.

Chapter 4 is all about the sensor integration and sensor fusion. The chapter focuses on the set of sensors that is available in Polaris. It describes the data being provided by each sensor and how they are mixed together by means of ROS packages to form the odometry data. The odometry data is prepared with respect to some fixed or moving frame of reference. The chapter also refers to SLAM and its usage in the current settings.

Chapter 5 deals with the navigation and motion control of the self-driving cars. Navigation is carried out at both global as well as local level. The mission

controller is part of the control system that is managing the navigational part. It also deals with the management of the velocity commands. Velocity commands are provided to the motion controller. The motion controller is described along with the description of the functionality of the low-level controller.

Lastly, the results are presented in detail in Chapter 6. The results illustrate how Polaris is traversing autonomously through the set of predefined target waypoints. Data is presented in the form of real-time maps and paths of the Polaris during the actual run. At the end of this chapter, recommendations about future work are presented.

2 State-of-the-Art

A big picture to summarize the structure of this thesis is roughly captured in Figure 2.1. As an important remark, one could state that the *big picture* presented here, may not fully capture every state-of-the-art concept related to present top-of-the-line autonomous ground vehicles. However, it presents the current implementation of the autonomous capabilities in our Polaris Ranger e-ATV. The final system will most probably be more complex depending upon the future requirements. Nevertheless, all the basic ideas behind each module in the big picture are concisely presented with appropriate references in the following sections. But first, a little discourse on state estimation and filtering technique is presented in the next section.

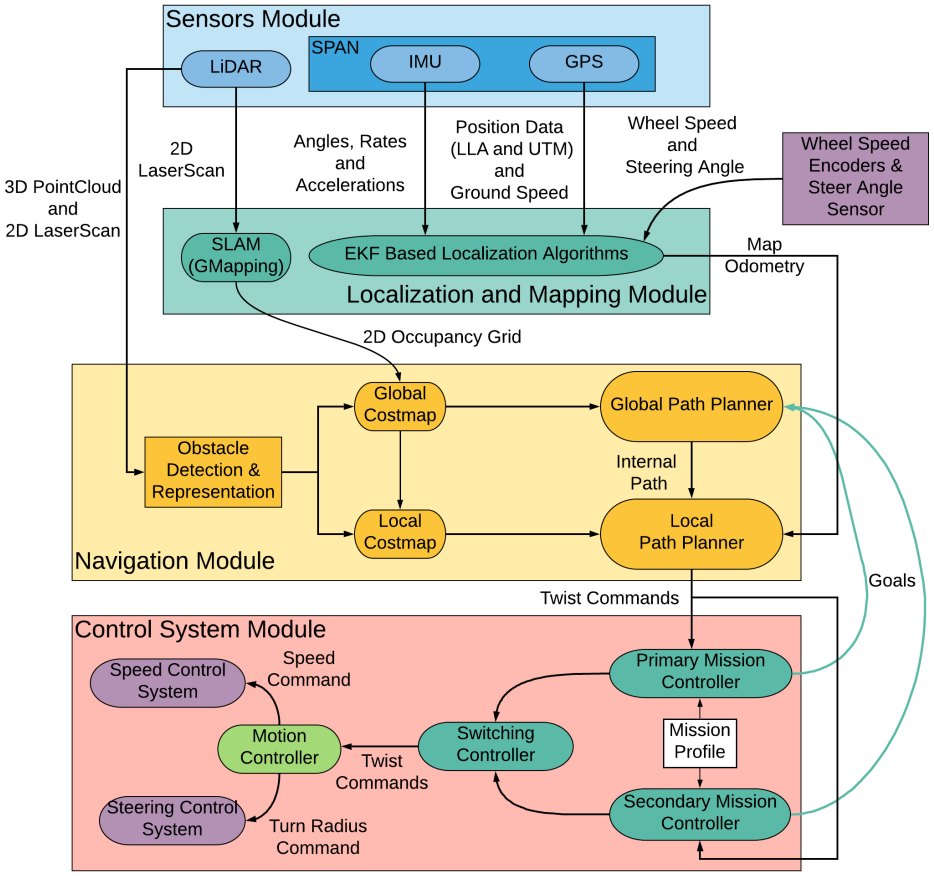

Figure 2.1: The Big Picture: The software flow of the autonomous functionalities implemented in Polaris.

2.1 State Estimation and Filtering Techniques

Each constituent module (block) in the big picture utilizes either one, or a combination, of the modern state estimation and filtering techniques. Therefore, it seems important to write a few lines about the most commonly used algorithms.

Kalman Filter, initially proposed by R. E. Kalman [7], is by far the most dependable and tried state estimation technique [10]. Also referred to as the Kalman-Bucy filter, it deals with state estimation given the dynamic model of the system is linearized and is corrupted by an additive source of the noise. It implements belief computation for continuous states [8]. At time instant t, the belief is represented by the mean and the covariance of the state. The posterior distribution of the state is Gaussian if the process, as well as the measurements, are assumed to be corrupted with additive Gaussian noise. It is an **optimal filtering** technique as it minimizes the covariances of the posterior distributions of the corresponding state over time.

Kalman-Bucy Filter [9] provides a mathematical basis to approximate solutions to nonlinear filtering problems. There exist various approximation methods to deal with nonlinear models, for example, based on Monte Carlo approximation, series expansions of processes and densities, Gaussian (process) approximation and many others [11]. Perhaps, **Extended Kalman Filter (EKF)** is the most commonly used (and simplest possible) extension to Kalman filtering which deals with the nonlinear system models [14]. EKF approximates the state transitions and measurements using linear Taylor series expansions. On the other hand, we get the **unscented Kalman-Bucy filter (UKF)** by selecting the Gaussian sigma-point type approximations of the drift functions (describing how noise would propagate) [12].

Non-Gaussian nature of the noise and nonlinearity of the system model renders optimal filtering techniques intractable. **Particle Filters** are a set of Monte Carlo algorithms used to provide solutions to the filtering problems when the processes are non-Gaussian and nonlinear [13]. **Sequential importance sampling (SIS)** is the most basic Monte Carlo method used in the particle filtering algorithm. SIS is based on *importance sampling*, which approximates a posterior distribution at a time step with a weighted set of samples (called particles), and recursively updates these particles to obtain an approximation to the posterior distribution at next time step. For each measurement, the importance factors (weights) are assigned to each particle. Thus, by incorporating the weights into the resampling step (drawing particles from posterior distribution at the next time step), the distribution of the particles is updated/corrected [15].

In the next sections, each module in the big picture is discussed separately with

a concise theoretical background.

2.2 Sensors Module

ROS is running on an **embedded computer**, which is a Linux based machine, placed inside a compartment built as a part of the e-ATV. The foremost part of building an autonomous robot is to provide the embedded computer with a set of data that is accurate and is provided at maximum possible data rate with a minimum delay. This real-time data includes the **orientation**, **position** and **velocity** information about the robot in some particular frame of reference. Such data in Polaris is coming from more than one sources, with each source measuring the orientation, position or velocity in its own frame of reference. Moreover, the embedded computer is receiving this data at different rates.

As an imperative precursor to robot's localization and mapping problems, it is important to discuss the Inertial Measurement Unit (IMU), Light Detection and Ranging (LiDAR), and Global Positioning System (GPS) units. As described in [16], IMU provides the information about orientation, angular velocities and linear accelerations of the robot. **Rate gyros** provide the angular velocities, while the **accelerometers** provide the information about how much force is applied to the machine in a specific direction. Both measurements are provided in the vehicle's body frame. However, such sensors are accurate up to a certain extent and are prone to noise, biases, and distortion with time. In addition, the orientation data, which include roll, pitch and yaw angles, is mostly estimated.

Optical radar or LiDAR is a device that constitutes a number of **laser range finders** [17]. A laser range finder consists of a transmitter that illuminates a target with laser light (at a certain frequency), and a receiver that detects the light that is reflected back from the target. Based on the time-of-flight, the distance to the target from the laser is calculated. A rotating (nodding) mirror is often placed in between the transmitter and the receiver to project the light in different directions. The time-of-flight is measured by either using a pulsed laser, modulated continuous wave or by measuring the phase shift of the reflected light. With multiple optical sensors or multiple laser beams, a 3D image of the environment can be realized using sophisticated scanning algorithms [8].

According to Andrew et. al. [40], **Global Navigation Satellite System** (GNSS) is the standard generic term for satellite navigation systems that provide autonomous geo-spatial positioning with global coverage. This term includes e.g. the GPS [37], GLONASS, Galileo, Beidou and other regional satellite systems that are used worldwide [25]. The advantage of having access to multiple satellite systems is accuracy, redundancy, and availability at all times. Synchronous

Position, Attitude and Navigation (SPAN) unit installed to Polaris can utilize signals from both GPS and GLONASS satellite systems simultaneously.

DGNSS system is an augmented GNSS system, which receives the position information from the ground-reference stations. Since the satellites and earth are constantly in motion with respect to one another, the position measurement from a normal GPS receiver is only accurate up to $\pm$ 7 meters. This location accuracy can be improved to a few centimeters if the correction information is supplied to the rover. A ground-reference station receives position information from a fixed GPS Antenna and keeps a record of the motion of the satellites over a long period of time. These records - position data, state covariances, observed satellites information, and time period of observations - are then broadcast over a communication network in the form of corrections.

2.3 Localization and Mapping Module

Robot localization is the procedure of defining where a mobile robot is positioned with respect to its surroundings. Localization is one of the most essential capabilities required from an autonomous robot because the information about the robot's own position is an indispensable precursor to creating decisions about future actions [19, 20, 21]. In a typical robot localization scenario, a **map of the environment** is available (known) and the robot is equipped with proprioceptive sensors (such as wheel encoders, gyroscopes, accelerometers, etc.) which monitor its own motion [21]. In practice, however, these sensors are noisy which makes the localization problem a bit more challenging. The localization becomes cumbersome in case the model (map) of the environment is incomplete [17].

Odometry is a generic term associated with the estimated trajectory (position) of the robot over time using motion sensors [21]. In wheel based odometry, measurement data from the wheel encoders (wheel speed) is used. In dead reckoning, the heading (yawing) measurements from a gyroscope or a magnetometer are also incorporated [17]. Wheel speeds (or transmission speeds) and steer angles are used to compute the linear and angular velocities of the robot. These velocities are then integrated to estimate the position and orientation of the robot in body coordinates. However, in order to estimate the pose (position and orientation) of the robot in the world (earth-fixed) frame, the (linear and angular) velocities must be transformed to earth-fixed coordinates before they are integrated. Another strategy is to transform the robot's pose in the world (earth-fixed) coordinates to the body coordinates and correct (update) the robot's pose in the body's frame of reference. This problem highlights the importance of sensor fusion using modern state estimation and filtering techniques.

Mapping of the surroundings of a robot involves registering (indexing) a list of objects in the environment and their locations [8]. Such listing of the environment as maps is either *feature-based* or *location-based*. In a feature-based map, each indexed (listed) feature is assigned a location in two-dimensional Cartesian coordinates. In location-based maps, the index corresponds to a specific location in the environment. Location-based maps are volumetric maps as they contain information about the occupied as well as the free space. Feature-based maps, however, only specify the shape of the surroundings at the locations where the objects were detected. Feature-based mapping makes it easier to adjust/correct the position of an object as soon as a new feature (object) is sensed in the environment.

Occupancy grid maps are location-based maps that make it easier for the robot to find an optimal (shortest) path through the unoccupied space. In such maps, a binary (0 meaning free space and 1 implying an occupied space in the map) occupancy value is assigned to every evenly spaced cell (a grid element) of the realized map [8]. In practice, each cell in the occupancy grid represents an individual Bayesian filter that keeps track of the probability of being occupied over time [21]. By treating each cell as a random variable, such mapping technique generates a consistent map from noisy and uncertain measurement data.

Thrun et. al in [8] describes **Simultaneous Localization and Mapping** (SLAM) as the problem associated with the localization of robot within an indefinite, static environment. This, in general, is a difficult problem (more difficult than localization), since the robot path and map of the environment are both unidentified. In the real world, the mapping between observations and landmarks (objects) is unknown; thus picking wrong data associations can have catastrophic consequences. Estimation techniques such as Kalman Filtering and Particle Filtering are mainly utilized in SLAM algorithms.

SLAM problem has two variants: online SLAM and full SLAM. The online variant of SLAM seeks to estimate the most recent pose of the robot and the map of the surroundings, whereas the full (global) version seeks to estimate the entire path and map of the environment [8]. EKF SLAM applies EKF to the online SLAM problem. It is a feature-based SLAM and constitutes an augmented state vector comprising pose coordinates of robot plus coordinates (in two dimensions) for each landmark. For pose data, the dynamics model is kinematics, while for landmarks a noise model is used.

The full SLAM has a very high computational cost and is seldom used. The online EKF SLAM has reduced computational complexity but needs sufficiently distinct landmarks and can diverge if nonlinearities (in system modeling) are large. Recently, Rao-Blackwellized particle filters have been introduced as an effective means to solve the SLAM problem [31]. It is also referred to as **Fas-**

tSLAM. This approach uses a particle filter in which each particle carries an individual map of the environment. Each map (within a particle) contains the pose information of each observed (registered) landmark along with the current pose of the robot. Each landmark is represented by an EKF. So, if there are **M** landmarks, each particle has to maintain **M** EKFs.

2.4 Navigation Module

Path planning is one of the crucial parts of a modern **navigation system**. It is usually a *geometric problem* and is associated with the guidance of the unmanned system to reach the target position from a particular location in a safe and optimum manner [36]. In other words, the main responsibility of the path planner is to acquire the least-cost and the shortest path from the initial location to the target location in a specific orientation. Obviously, any mobile robot on the ground takes into account the environment in order to generate a viable path. Paden et. al. [27] presented an excellent survey on the path planning techniques for urban vehicles.

The history of path planning goes back to 1956, in the Netherlands, when Edsger W. Dijkstra conceived an algorithm for finding these paths in an efficient and faster way. The logic behind is whatever path to be taken must be shortest from one point to another. Generally, it is a method to search the shortest path computed between two points on a graph so that the sum of the weights of the constituent edges is minimized. The applications of path planning are ubiquitous nowadays, from path planning of indoor robotics like a robotic vacuum cleaner to path planning in an outdoor environment as in the case of Polaris.

Moreover, Alonzo Kelly [21] states that the path planning must bear properties like soundness, completeness, and optimality. Soundness means that planned path must be feasible and admissible. In order to be complete, the planned path from the initial point to the final point must not contain any voids. Moreover, to be called optimal, the planned path must constitute the shortest distance between two points. Commonly, the path planner is *graph-based*. **Graph-based path planning** is a problem in which the objective is to find a path in the workspace (usually a map). In this setup, path planning is carried out in a configuration space. The configuration space is thus a planner readable object, which is built while moving the robot through the given environment (workspace) with obstacles.

Obstacle detection and representation is the process of using sensors' information (mainly laser scans from LiDAR) in order to detect the positions of the obstacles near the robot [21, 36]. This information is then used for creating and

updating a map of the environment. There are commonly two ways to represent the obstacles in the model (map) of the environment which are **obstacle map** and **cost map**. Obstacle map contains a set of discrete obstacle encoding various features of the obstacle such as pose, size, shape, and some motion attributes. In the cost maps, the map is a rasterized (divided into small finite regions or cells) cost field that stores a binary or continuous cost of traversal in each cell. To form a cost map, a common approach is to first construct a 3D volumetric representation of the LiDAR data, and derive a 2D cost representation from a **3D point cloud** [21]. The cost map is often defined over configuration space, as the cost of a configuration of a robot is naturally a configuration space quantity [21].

Obstacle avoidance is the process of adapting the precomputed trajectory (from global planner) while the robot is still moving in order to avoid unexpected obstacles that occlude the path. The desired path is found by performing a search for a minimum-cost path in the configuration space. According to Thrun et. al. [18], **graph search method** discretizes the configuration space of the vehicle into a graph. The search graph connects the states, where the states represent a finite collection of vehicle configurations (like position and orientation) and the edges represent transitions between states. The encoded spatial coordinates within the states are used to generate neighbors states and to access intersections with the obtacle or cost maps.

The graph search method solves the least cost problem between two states (edges) on a directed graph. A directed graph is one that is made up of the vertices connected by edges to a certain (directed) depth. Such a discrete representation of the graph structure has limitations. For example, the discretized state space makes completeness at the stake as it might miss to consider few vertices in the directed graph. Moreover, the feasibility of the path is often not inherently encoded. Several efficient algorithms for graph construction exists within the framework of geometric methods of path planning. For example, the vertical cell decomposition [28], generalized Voronoi diagrams [32], and visibility graphs [35].

On a directed graph, the **breadth-first search** method explores the neighboring vertices (nodes) before moving on to vertices at the next depth level. It corresponds to the wavefront expansion on a 2D grid, and considers the first-found solution as optimal if all edges have equal costs. The Dijkstra's search is a sorted variation of the breadth-first search, in which the first-found path is guaranteed to be optimal no matter the cell cost. Moreover, a search tree from a search graph is often constructed by the global path planner. Because in the real-time applications, computational cost and effective range of the sensors limit the motion planning of the robot on a larger scale (on the scale of kilo-

meters). **Depth-limited planning** restricts the planning horizon by generating a *limited* (depth level of) search tree. The real-time A* algorithm uses such a limited depth search process, as it propagates the total cost of path traversal from the neighboring states to the root (back up the tree) in an optimal manner [21].

However, in an expansive and dynamic environment, there is a continuous need for replanning the trajectory between the endpoints. This renders A* inefficient and impractical in such outdoor applications. D* (and its variant, D* Lite) algorithm is based on the **plan repair approach**, which constructs a new plan that is not so different from the previous plan. So, whenever a new feature is observed, the corresponding cost changes in the graph are incorporated [21]. Thus, the key feature of the D* algorithm is *incremental replanning* [19]. The incremental replanning has a lower computational cost than completely replanning as would be carried out in the A* algorithm [19]. Thus, the A* method is preferable for indoor robot applications whereas D* planner is suitable for dynamic outdoor field robots [21]. The current robotics community uses either of the Dijkstra's, A* and D* algorithms on a cost map for the global motion planning of the robot.

Waypoint navigation is dictated by a set of pre-determined **GNSS coordinates** called *target waypoints* or simply waypoints [41]. These waypoints are used to set the **goals** for the global motion planner. The global path is planned by one of the graphs based (graph or tree) search methods. The local path, however, is selected by either **Trajectory Rollout** or **Dynamic Window** approach [29, 30]. Both assume implicitly that the robot has a differential drive kinematics model [21]. The basic functionality of each local planner is to generate a trajectory that adapts to the dynamic environment by avoiding obstacles. Each trajectory is associated with an objective function, which includes goal heading, path heading, and obstacle clearance. The number of trajectories to be simulated is user-dependent and only one with the maximum objective function value is selected [23].

2.5 Control System Module

In our big picture, the control system module handles two **mission controllers** that read a file containing all the target waypoints of the trajectory to be traversed. These target waypoints are in *Longitude-Latitude* format. The waypoints are selected from one of the previous runs of the Polaris during the testing phase. These selected waypoints are stored in a text file, which in turn are used as a mission profile each time the software starts. Thus, the mission profile provides a set of goals for the global planner in the navigation module.

However, GNSS based positioning affects the accuracy of the selected target

waypoints [40]. Thus, in order to be accurate, each target waypoint should be selected through averaging of the measured positions of a single target point over 24 hours of satellite observations. The averaging of supposedly tens of waypoints is of course impractical. Consequently, this led to the use of a **goal cancellation policy** once the vehicle arrives in the ballpark of the target position. Such a policy dictates to cancel the previous goal once the vehicle reaches the target point within few meters, and sets the next target waypoint as the next goal for the vehicle to approach[1].

As a consequence of the goal cancellation policy, a single instance of the mission controller was incapable to continuously (smoothly) drive the vehicle from one goal to another. The reasoning will be provided in more detail in Section 5.2 in light of the discussion presented in Section 3.3. Therefore, a **double controller** - a primary plus secondary mission controller - strategy was utilized during the thesis process. Such configuration relies on running two identical controllers side-by-side. Analogous to a relay race, as soon as the primary controller completes its job, the control authority is *switched* to the secondary controller, and so on. Therefore, the smooth transitions from one target-point to the next can be achieved in a rather unconventional manner.

Switching controller is the algorithm responsible for the switching of the control action between the primary and the secondary controllers. It has an added functionality of the *dead man's switch*. As the name suggests, dead man's switch is a device used in (modern) railway systems to apply brakes (or stop sending commands) in case the driver of the train becomes unresponsive (for any medical reason). In this case, the algorithm will stop sending the commands issued from a particular controller once it realizes that the commands have been constant for a longer period. In other words, this chunk of code is receiving the command velocities from both primary and secondary controllers and is making the real-time decision of selecting the appropriate controller to issue the velocity commands to the Polaris' actuators. There are many ways to implement such an algorithm, and further research needs to be conducted for this topic in order to find the best possible solution. However, in this thesis work, mainly due to timing issues, a rather naïve strategy was implemented through which the main requirements of the project were *somehow, although not completely*, met.

Finally, as depicted in Figure 2.1, **motion controller** receives the velocity commands (twist commands) from the switching controller and converts them into speed and turn radius commands. Two PLC-based control units are installed on Polaris [3, 59]. One is dedicated to controlling the wheel speed (as an acceleration pedal controller) of the Polaris, while the other is controlling its power

[1] A somewhat comparable idea has also been put forward by [54] for their *Husky Outdoor GPS Waypoint Navigation project.*

steering mechanism. The acceleration pedal controller is essentially a PID controller (in autonomous drive mode) [3], which translates the speed command into the forward motion of the vehicle. In manual drive mode, however, the speed command is (obviously) generated by means of pressing the acceleration pedals.

The power steering controller is also implemented as a PID controller, which attempts to adjust the estimated curvature of motion of Polaris to match the turn radius command from the motion controller [3]. The steering angle of the front-wheel shaft link and the speed of the rear-wheel axle are sensed by dedicated wheel encoders. These wheel encoders measurements are sent to the embedded controller. The two PLCs communicate over the Controller Area Network (CAN) bus with each other and with the embedded computer. These measurements are utilized by the localization algorithms, which completes the feedback loop.

With the discussion about each module in the big picture, the presentation of the state-of-the-art involved in this thesis work is concluded. We will first talk about working with ROS in the next chapter, before moving on to present the implementation of autonomous capabilities in Polaris e-ATV.

3 Working with ROS

In this chapter, a brief description of the ROS (Robot Operating System) is presented. Many books [38, 39] and numerous on-line resources [42] are available, and they will naturally describe ROS in a more detailed way. Here, the main intention is to introduce the key concepts, which will help the reader to better understand the results of this thesis work.

3.1 Architecture and Philosophy

ROS is an open-source software development platform provided mainly for the researchers, teachers as well as for start-ups community associated with autonomous mobile robots. It is more like an environment used to test the algorithms rather than an actual operating system. ROS can be attributed as a middleware that comprises of a collection of software communicating with one another. Moreover, it provides exceptional modularity in the form of software drivers for a range of hardware, which are currently being used in the robotics industry.

The software can be written in either C or C++ or Python languages. Each individual collection of software in ROS is termed as a package. Philosophically, each package in ROS provides a unique or some special software features to the robot under test. In our case, for example, the mission controller portion is implemented as a package that handles a particular set of functionalities within ROS environment.

Moreover, the ROS provides device drivers that are instrumental in the basic integration of hardware to the main software. In our case, the ROS community had already developed device drivers for Novatel SPAN-IGM-S1 unit as well as for Velodyne's HDL-32E LiDAR. SPAN is the main GPS based navigation system in Polaris, while HDL-32E is the unit providing the necessary 3D point cloud to the main software.

One only needs to install the required ROS packages, manipulate the basic integration settings and the device is ready to be used by the program. On the downside, these packages are often written as a general purpose hardware integration tools and most times lack some more advanced features. This issue will be highlighted in the sensor integration chapter (Chapter 4).

ROS is responsible for process management and inter-process communications. Each package can have single or multiple nodes. A node can be regarded as a subset of tasks done by a package. These nodes communicate with one another on a local network using topics. Each ROS package has a set of topic subscribers

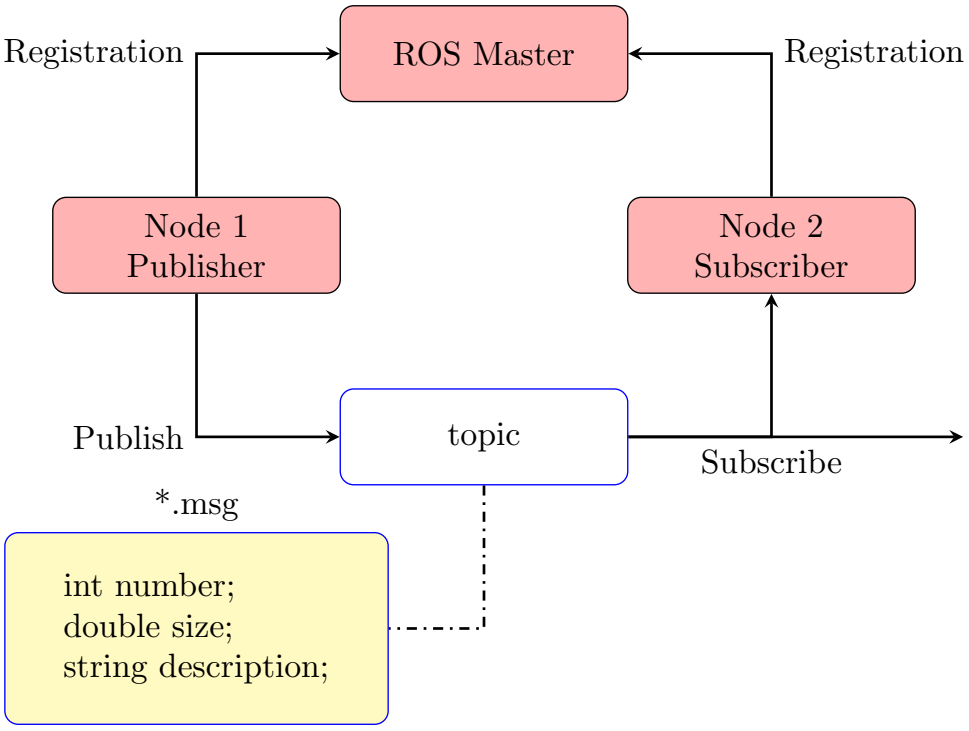

Figure 3.1: ROS Working Philosophy: A simple depiction about how ROS nodes acting as either subscriber or publisher or both communicate with one another and with ROS Master using topics.

as well as a set of topic publishers.

These topics communicate pieces of information within the ROS framework. Each package introduces a new node that adds a new set of topics to the existing ROS environment. These topics are just messages or chunk of information packed together in a ROS readable format. A typical situation is depicted in Figure 3.1, where two nodes are getting registered with the Master.

Topics have anonymous publish/subscribe semantics, which decouples the production of information from its consumption. In general, nodes are not aware of who they are communicating with. Instead, nodes that are interested in data subscribe to the relevant topic; nodes that generate data publish to the relevant topic. Master is the ROS Core initiated by the program, which handles all of the packages communicating within one framework via a local network. In Figure 3.1 one node is acting as a subscriber and the other is a publisher. The publisher is transmitting a topic, which is a message in ROS. There may be multiple subscribers to this topic that are utilizing the same information. There may be multiple publishers and subscribers to the topic.

3.2 Tools

ROS tools are responsible for the well-established ecosystem of ROS. ROS provides an excellent visualization tool called RViz for the program as shown in Figure 3.2. RViz helps to categorize multifarious topics in order to visualize the provided information.

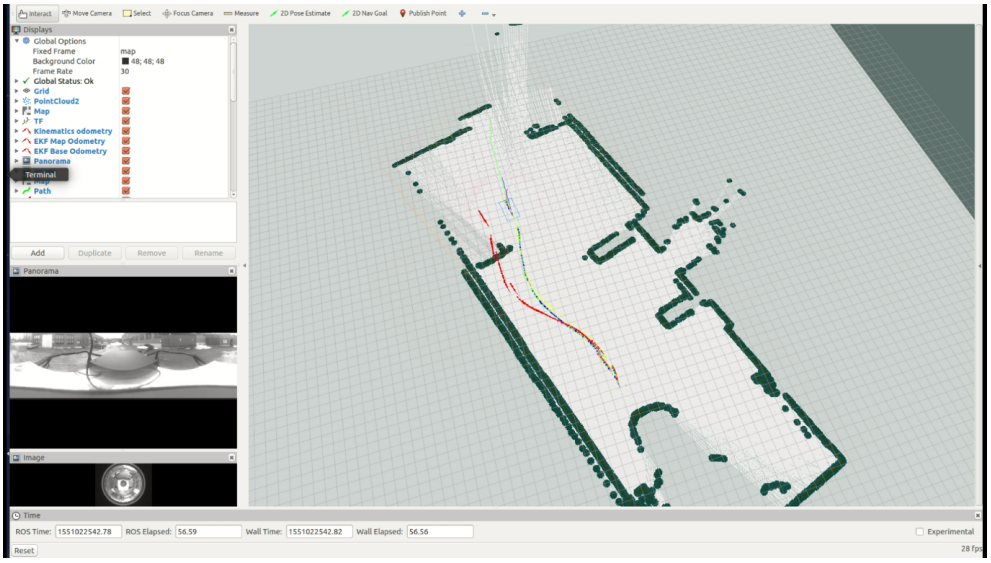

Figure 3.2: ROS RViz: A dynamic visualization environment to monitor the real-time behavior of topics. The arrangement of the layout is up to the user. Here, the list of topics is shown in the top left corner. In the bottom-left, one can see the output of the camera in a panoramic mode. The main window depicts the map of the environment along with the trajectory of the vehicle.

For example, the lines in Figure 3.2 are the paths predicted by different localization routines. Similarly, the map is provided by another topic, which is the output of the SLAM algorithm. The blackout area is the cost map issued by another package. Moreover, one can observe the panoramic output from a camera in the lower left corner, which is another topic.

Figure 3.3 shows the unified robot description format (*urdf* model) of Polaris being imported inside the Gazebo's *world* environment. The Gazebo simulator is one of the most important simulation tools provided for the ROS community [24]. It is a 3D simulator to simulate the rigid body dynamics of the robot. Moreover, the surroundings can also be simulated in Gazebo in order to evaluate the performance of the robot in any dynamic or static environment settings, as shown in Figure 3.5. Gazebo is an open-source software like ROS and is being used by the research community worldwide.

One can add up a number of topics to the simulation in order to simulate a variety

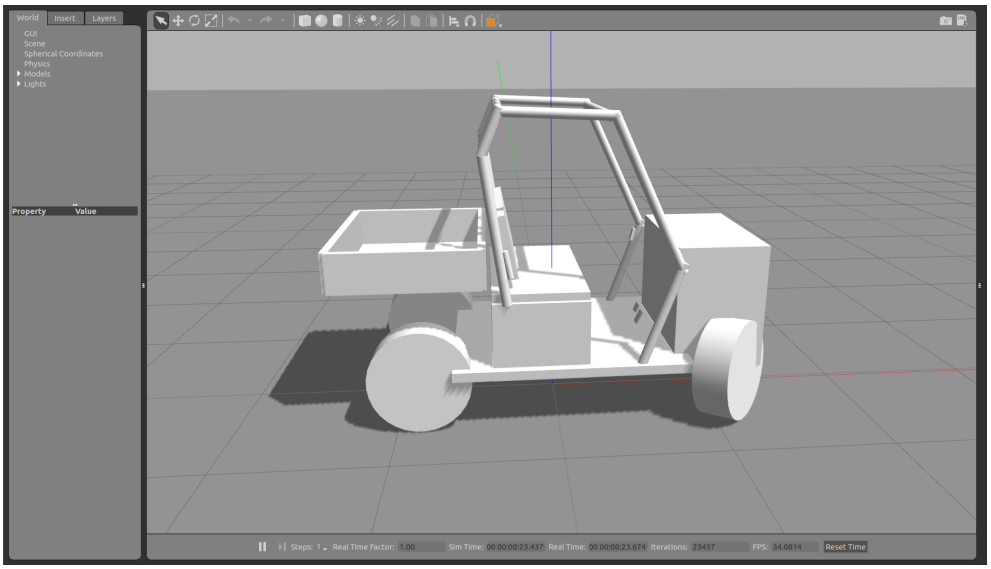

Figure 3.3: Gazebo: Polaris being imported into the Gazebo environment as a robot model. One can dynamically interact with the environment in Gazebo in order to provoke changes in the simulated environment.

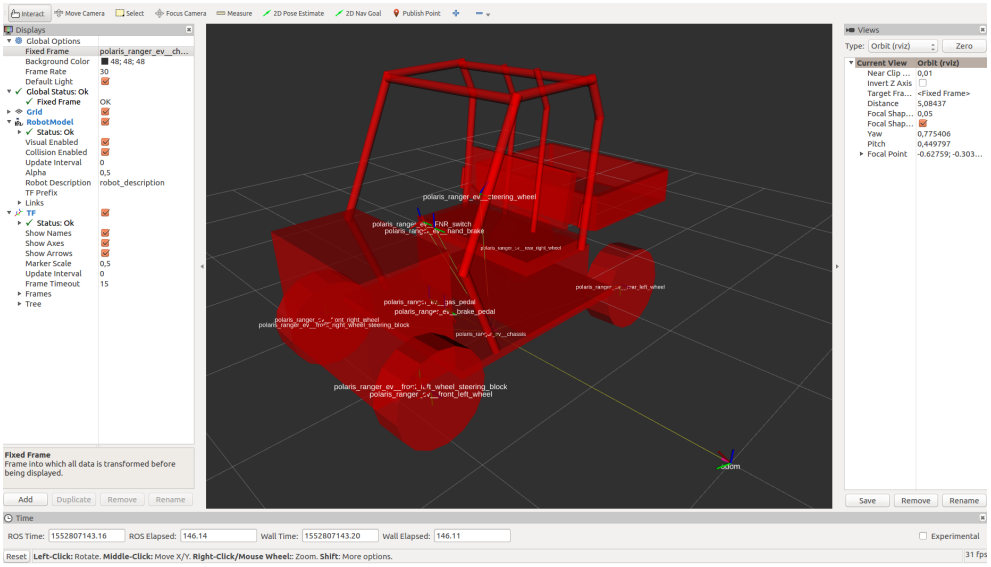

Figure 3.4: ROS RViz: Polaris model imported into the RViz environment. One can see various frames attached to different parts of the robot. These frames are linked with one another through a transformation tree.

of sensors. For example, the IMU/GPS will be emulated as nodes generating topics in the simulation. The robot model in a unified robot description format (*urdf* model) can be imported to RViz. Figure 3.4 shows the urdf model of Polaris imported as a robot model inside RViz.

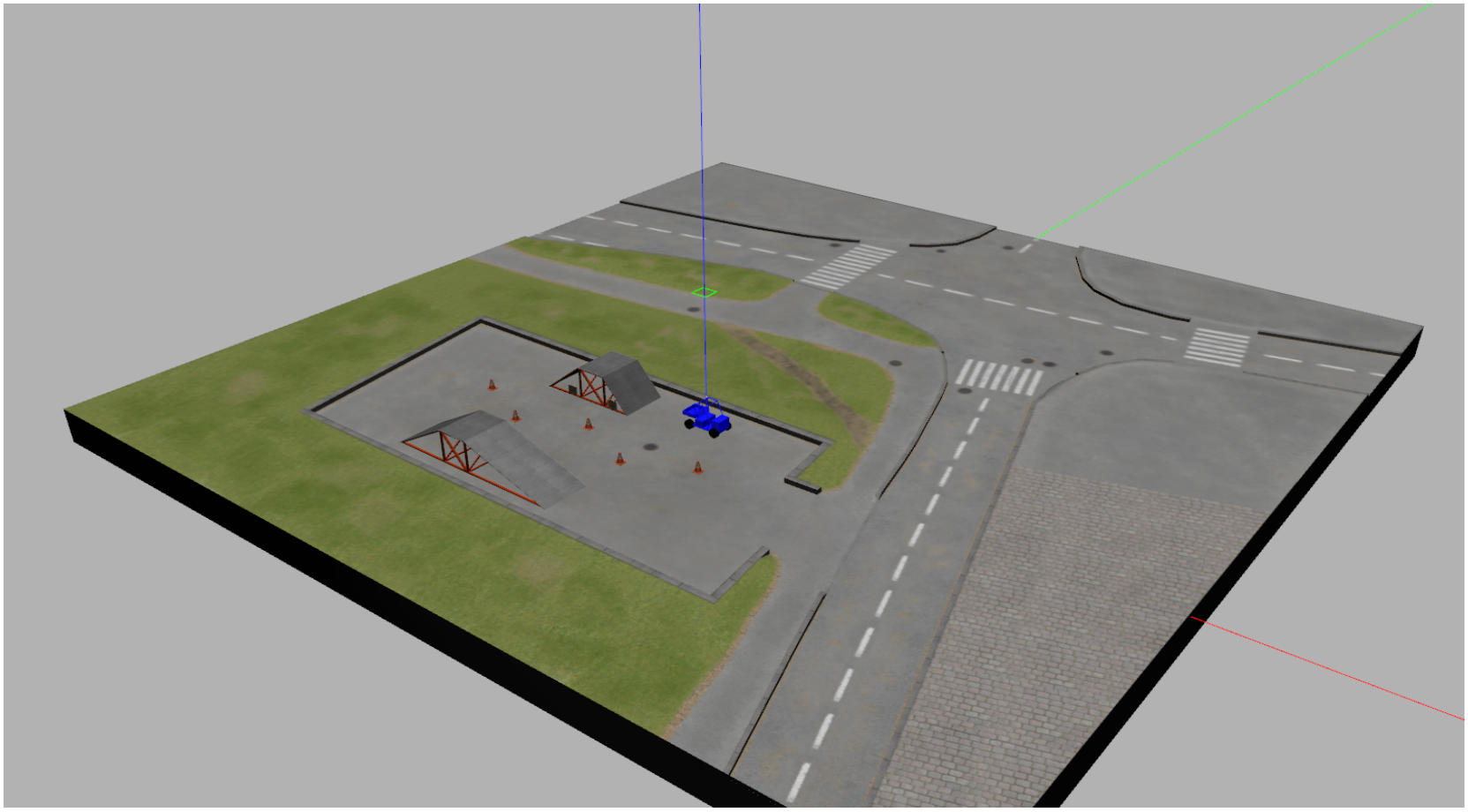

Figure 3.5: Robot environment simulated inside Gazebo. One can visualize the blue Polaris standing in front of a ramp. The simulated roads and other obstacles are also visible.

As discussed in Section 1.2, a group of students was instructed to build a simulation environment for autonomous Polaris Ranger e-ATV utilizing ROS and Gazebo. Lukas Wachter et. al. [5] managed to produce tangible results for their project work. Figures 3.3, 3.4 and 3.5 were managed by the same group. Thus, the possibility of building a simulation platform for the Polaris will enable us to rapidly test navigation and control algorithms using (somewhat) realistic scenarios.

3.3 Services and Actions

Figure 3.1 depicted the working philosophy within the ROS framework in the context of subscribers and publishers. ROS provides two more types of communication protocols between nodes, namely **services** and **actions**.

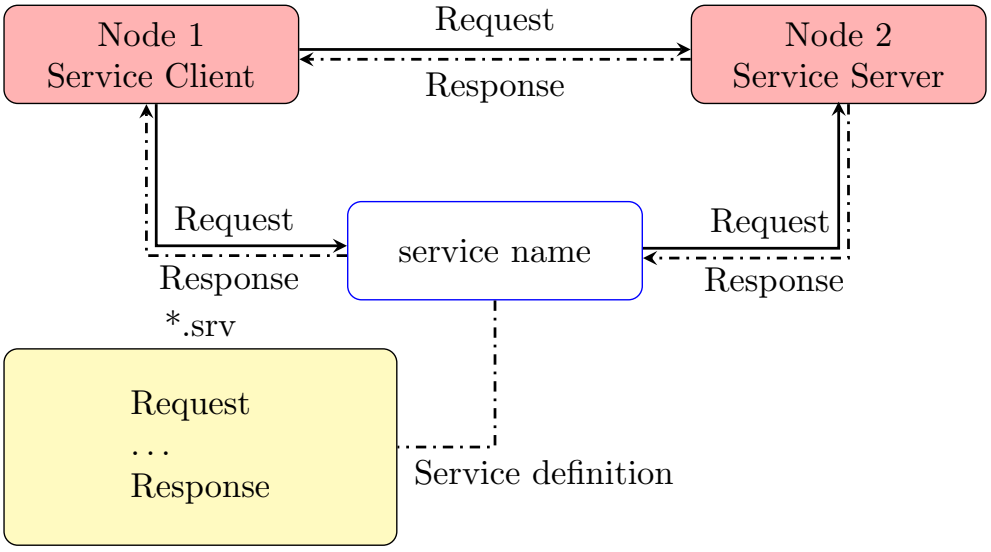

Figure 3.6: ROS Services: Communication between ROS service client and server.

As illustrated in Figure 3.6, a ROS service realizes communication between a service server and a client. The structure of the service is similar to the messages, only the service definition is found in *.srv files. Each service client fills a service request to modify the data structure of the *.srv file. Service server receives the request, and process it by calling the appropriate function. As soon as the server services the request, it responds back (via some predefined response) to the client.

Essentially, ROS Service provides a blocking call for processing a request [22], which makes it suitable for applications where short triggers or abrupt calculati-

ons are required. For example, in our motion controller, the turn radius is one of the ROS service parameters. Whenever it is required to modify the turn radius, a service client makes a call. Service server applies the commanded turn radius and gives feedback (as the response) to the service client as soon as the commanded turn radius is achieved. In some situations, however, it is often required by the user to cancel the execution of the task. An example of such a situation is for example, when the task is taking longer time to execute or when it is no longer required. Such functionality is provided by the ROS Actions library.

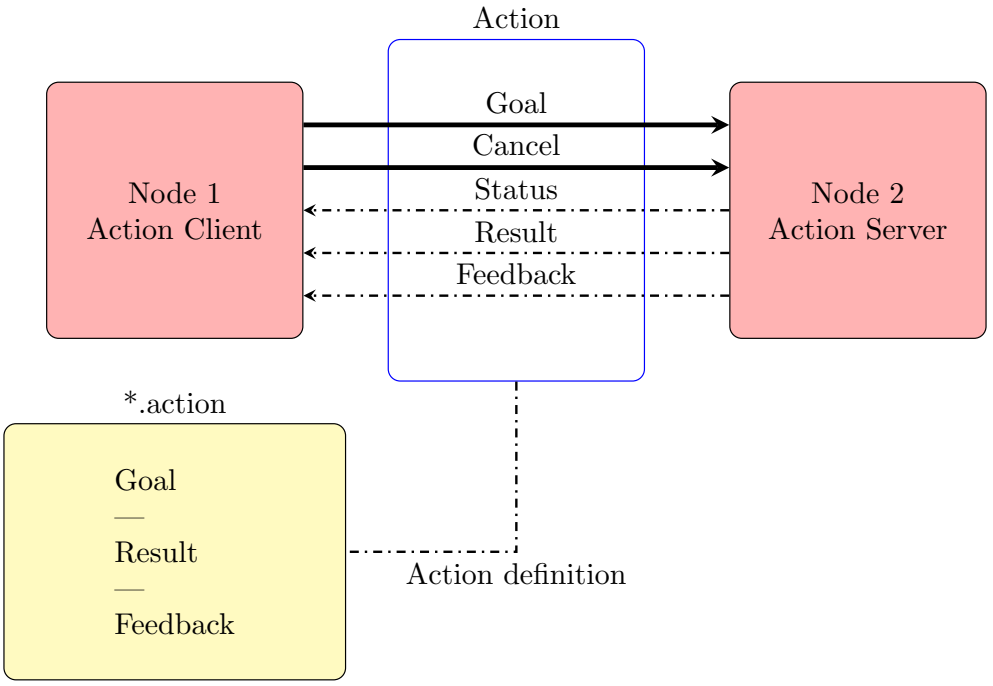

Figure 3.7: ROS Actions: Communication between ROS action client and server.

Figure 3.7 describes the working philosophy of ROS Actions. The communication protocol is somewhat similar to service calls along with the possibility to cancel the task or to receive feedback on the progress while the task is being executed. The action definition is placed inside a *.action file. It provides a better way to implement non-blocking, *preemptable* goal-oriented tasks [22].

In Figure 3.7 **Goal**, **Result** and **Feedback** are messages with which the server and the client communicate. The Goal message is sent by the user via an action client. The Feedback message informs the client about the progress of the task. The Result message is sent from the server upon completion of the goal. Common applications of ROS actions are in the implementation of navigation, grasping or motion execution algorithms. In the case of *move_base* package [51], the goal (where the robot should move in the world) is fed to the controller via

an action client. The client can check the progress of the *move_base* controller at any time instant. It can also cancel (reset) the goal sent to the action server.

3.4 Conventions

A thorough understanding of the coordinate systems and data conventions followed by the ROS community is an essential precursor to sensor integration in ROS. ROS follows REP-103 standards [44], which defines the standards of measurements and conventions. Figure 3.8 highlights the different coordinate systems attached to a hypothetical vehicle. The most elementary coordinate system is attached to the body of the vehicle.

To understand the body's frame, consider a person sitting in the driving position, the forward direction is positive x-direction. To the left side of the driver is the positive y-direction. The positive z-direction points upward. The orientation of the ground vehicle follows the right-hand rotation rule. The rule says that point the thumb in the positive x-axis, then the fingers will curl to show the positive roll angle direction. Following this rule, a positive roll is left side up; the positive pitch is nose-down and positive yaw angle corresponds to the counter-clockwise turn of the vehicle. The same goes with the angular velocities.

Accelerations convention followed by ROS is a bit tricky to understand. Naturally, forward motion corresponds to positive acceleration in the body's x-axis. When the vehicle is turning in a counter-clockwise direction, the left-side acceleration (acceleration along the y-axis) must be positive. In addition, for the flying vehicles, the upwards flight corresponds to positive acceleration along the z-axis.

The coordinate system attached to the *world frame* must have its origin attached to the center of the earth, or can be related to Earth-Centered-Earth-Fixed (ECEF) frame of reference. This is the reason why East-North-Up (ENU) is used for describing the world frame in ROS. Another such type of system is the North-East–Down (NED) coordinate system. NED coordinate system is mostly used for the flying vehicles. Figure 3.8 (left) depicts the relationship between ECEF and ENU coordinate system. ENU can be considered as a plane attached to the surface of the Earth. With East and North pointing towards the East and North of the Earth.

GNSS measurement instruments calculate the position of the vehicle in Latitude, Longitude, and Altitude (LLA) format. However, such position data is with reference to the satellites and must be transformed into a grid that is attached to Earth via the ECEF coordinate system. There are well-established algorithms

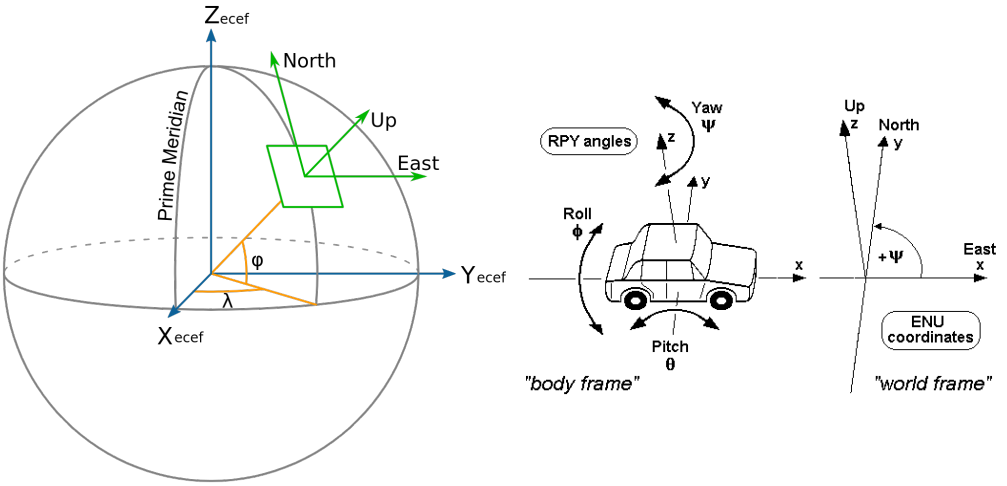

Figure 3.8: Left: Relationship between ENU and ECEF coordinate systems is shown. Right: Relationship between body frame of the car to the world frame in illustrated.

to do such transformations as described in [46]. One such transformation is from LLA to Universal Transverse Mercator (UTM) grid system. The UTM projection is defined as a navigational grid projected onto the Earth's curved surface. Thus, it aids in pinpointing the vehicle's location in the world frame. In our ROS settings, this also defines a *map frame* for the vehicle.

The three-dimensional coordinates system attached to the vehicle's body is related to the world (map) frame by means of the ENU coordinate system. The *base_link* coordinate is referred to as the center of the body's frame of reference. Figure 3.8 (right) shows the orientation of the car with respect to the ENU coordinate system. At zero heading, the x-axis in the body's frame is aligned with the East direction of the world frame. The positive y-axis, to the left side of the driver, must be aligned with the North of the world frame, and the Up in world frame is up in body frame.

Figure 3.9 depicts the various coordinate frames attached to a mobile robot. These frames are linked to one another according to ROS REP-105 standard [45]. *base_link* is attributed to the frame that is fixed to the mobile robot platform, and is supposedly attached to the center of gravity (c.g.) of the robot. *odom* is referred to as the odometry frame. This is a hypothetical frame and is updated as the robot moves from its starting position. *map* frame is associated with the map of the environment in which the robot is moving. This map is generally provided by the SLAM algorithm based on the LiDAR data. For outdoor robots, such as Polaris, the map frame is linked to the *world* frame via special

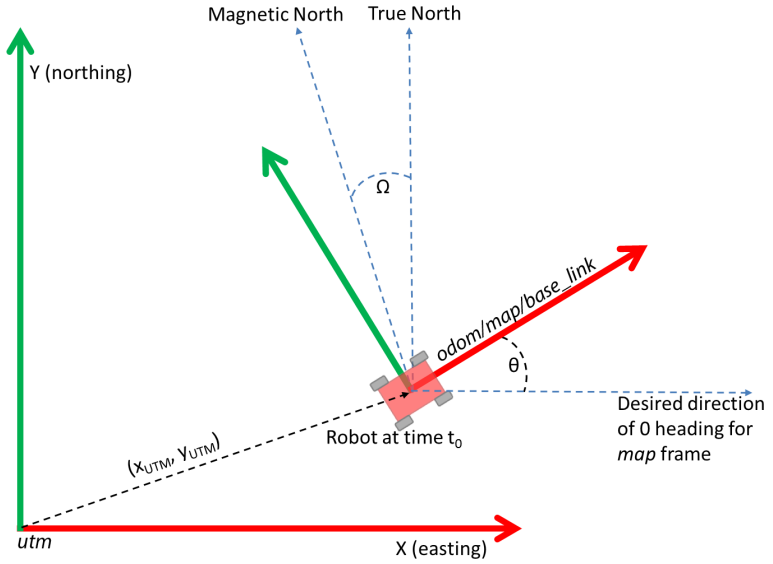

Figure 3.9: Various frames attached to the robot in relation to UTM frame of reference.

coordinate transformations. The nature of such transformations within the ROS framework is discussed in the next section.

3.5 Transformation System

ROS Transformation system keeps track of the coordinate frames attached to various parts of the robot with time [47]. It maintains a relationship between coordinate frames in a tree structure.

Figure 3.10 depicts the ROS transformation tree implemented in Polaris. Each coordinate frame is linked to others via this transformation tree. It is important to mention here that there can be only one parent for each node in a transformation tree. For example, *odom* can be originating from only one coordinate frame, in our case, from the *map* frame. Either these transformations can be provided by some ROS package or these transformations may be static.

The base link (described as *base_link* in the ROS framework) of Polaris is the center point of the rear axle. Basically, all odometric measurements are calculated with respect to this coordinate frame that is hypothetically attached to the center of the rear axle. All sensors (LiDAR, Camera, IMU, and GPS) are represented as coordinate systems attached to particular locations on Polaris. All transformations originating from the base_link are static transformations. These are representing the fixed placement of the sensors with respect to the base_link on Polaris. The *velodyne_tf* represents the frame depicts where the camera is

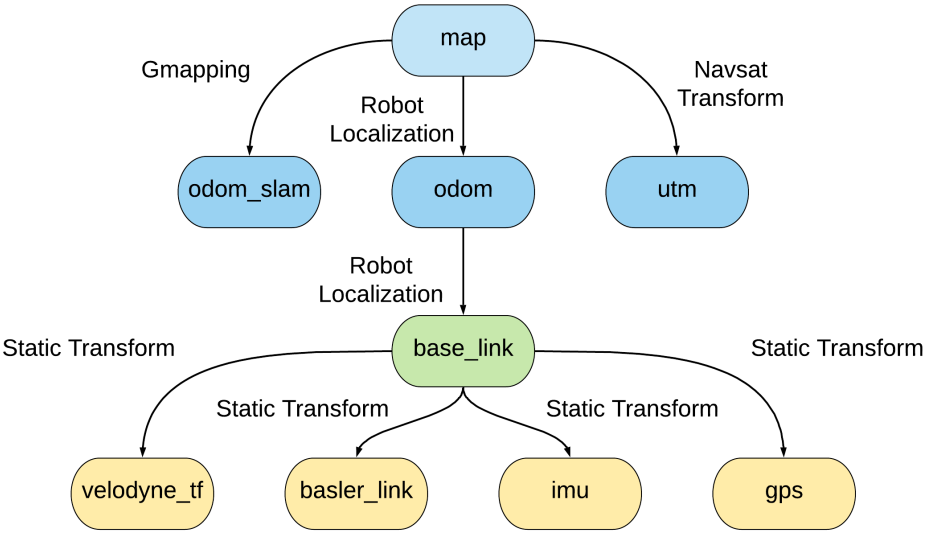

Figure 3.10: A working transformation tree implemented in ROS is redrawn to illustrate the relationship between various coordinate frames attached to the Polaris.

fixed on Polaris. *imu* is showing IMU's pose, and *gps* is localizing the orientation of the GPS unit on Polaris.

In the ROS framework, the transformations between *map*, *odom* and *base_link* coordinate frames are maintained by two separate instances of the *Robot Localization* algorithm. A part of this routine, *Navsat Transform*, provides the relationship between *map* and *utm* coordinates. While, the relation between *map* and *odom_slam* coordinates is provided by the *GMapping* algorithm.

In the next chapter, the implementation of each odometric coordinate frame will be discussed in detail.

4 Localization and Mapping

As said earlier, the main goal of the thesis work is to utilize each sensor to its full potential. This chapter begins with a short introduction of the individual sensors, as are depicted in Figure 4.1. It then enlists the available data from every sensor at the provided data rate. Furthermore, it presents the integration of the sensor data in Polaris. Finally, it highlights the fusion of the available data in ROS.

(a) Velodyne HDL-32E

(b) Basler Ace Series Camera

(c) SPAN-IGM-S1

(d) EPEC 5050 Control Unit

Figure 4.1: Electronics systems available in Polaris.

4.1 Electronics Systems

A brief explanation about each electronics system installed in the Polaris is provided below. The description is based on the device manuals as provided in [56, 57, 58, 59].

(a) The Velodyne HDL-32E is shown in Figure 4.1a. It utilizes 32 lasers aligned from +10° to -30° to provide a vertical field of view, and its rotating head captures a 360° horizontal field of view in real-time. It generates a point cloud of up to 700,000 points per second with a range of 100 meters and a typical accuracy of $\pm$ 2cm. The optimal update frequency of these point clouds is 10 Hz.

(b) Polaris is equipped with a Basler Ace Series Camera that is shown in Figure 4.1b. It is connected to the embedded computer via Gigabit Ethernet for configuration and image retrieval purposes. The camera has an omnidirectional lens attached to it that enables it to capture pictures of the whole 360° range of the surrounding area. Manual adjustment of the aperture size

and focus is done by rotating the knobs of the lens' housing. The field of view of the camera system is -60° to +15°. It would be beneficial for future research work, for example, for texturing of the point cloud (created by the LiDAR) and obstacle detection.

(c) Synchronous Position, Attitude, and Navigation (SPAN) technology brings the amalgamation of Global Navigation Satellite System (GNSS) based positioning and inertial navigation technologies. The absolute accuracy of GNSS positioning and the stability of the Inertial Measurement Unit (IMU) are tightly coupled. It can receive differential corrections to improve the positioning accuracy. Figure 4.1c shows the SPAN device which provides an exceptional 3D navigation solution that is stable and continuously available. It performs well in situations when satellite signals are blocked by the surroundings. Moreover, it uses both GPS and GLONASS satellite systems simultaneously.

(d) The e-ATV vehicle had been equipped with two EPEC 5050 controllers (shown in Figure 4.1d). These controllers take care of the low-level communication between base-level motion sensors and actuators, and the higher level embedded computer via CAN-bus. The automation software of each PLC is developed in the CODESYS environment. The software executes at 10 Hz.

4.2 Sensor Integration

4.2.1 List of Available Data Packets

Following is the list of all available data packets acquired from the SPAN, LiDAR and camera.

1. GPS position expressed in UTM coordinate system at 10 Hz via a serial channel at 115200 bps.
2. Integrated GPS position provided in the Latitude, Longitude, and Altitude (LLA format) at 50 Hz via USB bus at 115200 bps.
3. GPS Track over Ground Velocity and Course Angle at 10 Hz via a serial channel at 115200 bps.
4. Integrated North, East and Up Velocities at 50 Hz via USB bus at 115200 bps.
5. Corrected IMU angular rates and linear accelerations at 50 Hz via USB bus at 115200 bps.

6. Position, attitude and velocity covariances 1 Hz via a serial channel at 115200 bps.
7. Estimated orientation (Roll, Pitch, and Azimuth) at 50 Hz via USB bus at 115200 bps.
8. Wheel encoder data and steering angle measurements at 10 Hz via CAN bus provided by the two EPEC units.
9. LiDAR Data in the form of a 3D point cloud at 20 Hz via Ethernet channel.
10. 360° Camera data at 10 fps via Ethernet channel.
11. DGPS corrections from NLS Finland in RTCM Format at 1 Hz via a serial channel at 115200 bps.

Figure 4.2: Satellite image of the test site from Google Maps.

4.2.2 Test Drive Profile

Figure 4.2 depicts the satellite view of the area in which the driving tests were conducted. The red pointer is on the outside of the Autonomous System Lab.

It is chosen to be the starting as well as the finishing point for the test drives. The location was selected to avoid traffic in normal working hours. Moreover, around the top-left and bottom-left corners were closed spaces, where the loss of satellite signals was observed. This came up with few test drives during which the GPS was completely lost. However, only good data is shown here in order to highlight the fact that the SPAN data is integrated properly.

The position profile of a typical test drive is shown in Figure 4.3. The red cross is the starting point of the test drive and the green circle is the finishing point. Polaris followed right-hand driving rule. In order to check the 360° turn, a loop was made. The yaw angle and yaw rate profiles during this test are shown in Figure 4.4. The green curve shows the evolution of the yaw angle during this drive. After the initialization of SPAN, it started with the reading of -50°. Yaw rate also follows the correct sign conventions during the test drive as depicted in the red curve.

Figure 4.5 depicts the behavior of linear accelerations along each of the constituent body axes. An important thing to notice here is the noise levels in the acceleration data. Considering the specifications of the accelerometers, this may be due to induced vibrations into the system because of the non-standard fixture of the SPAN in Polaris as depicted in Figure 4.1.

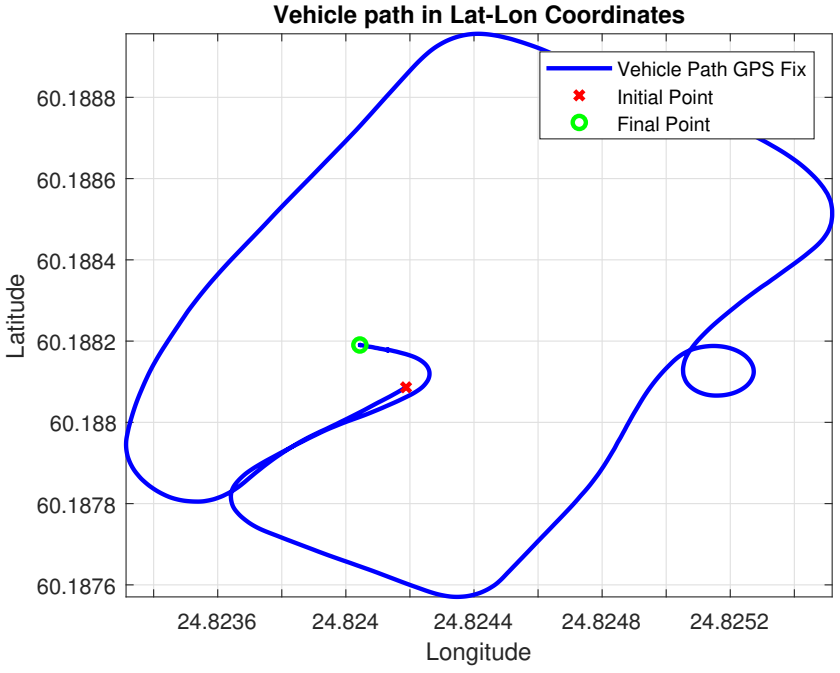

Figure 4.3: Position profile during the test drive.

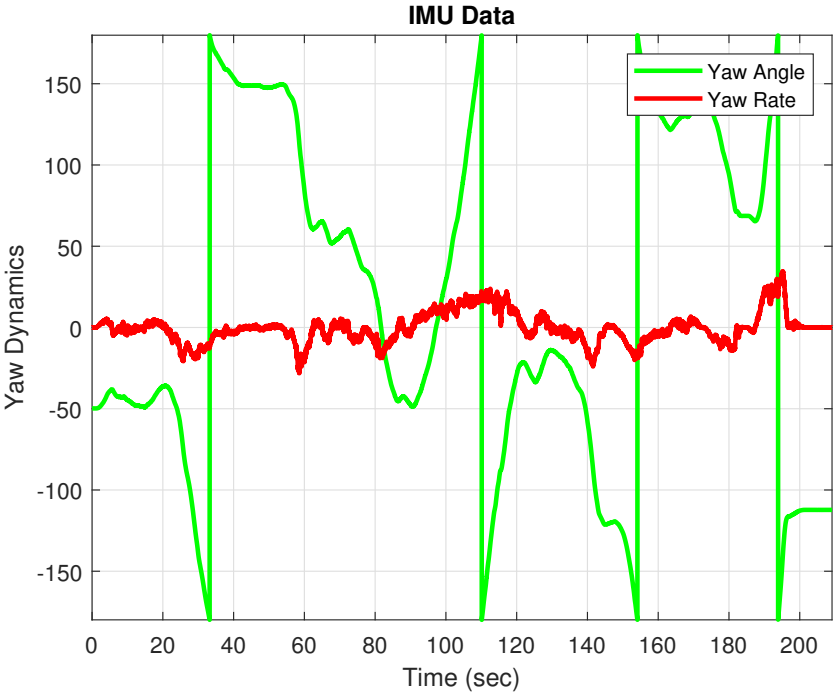

Figure 4.4: Yaw angle and yaw rate during the test drive.

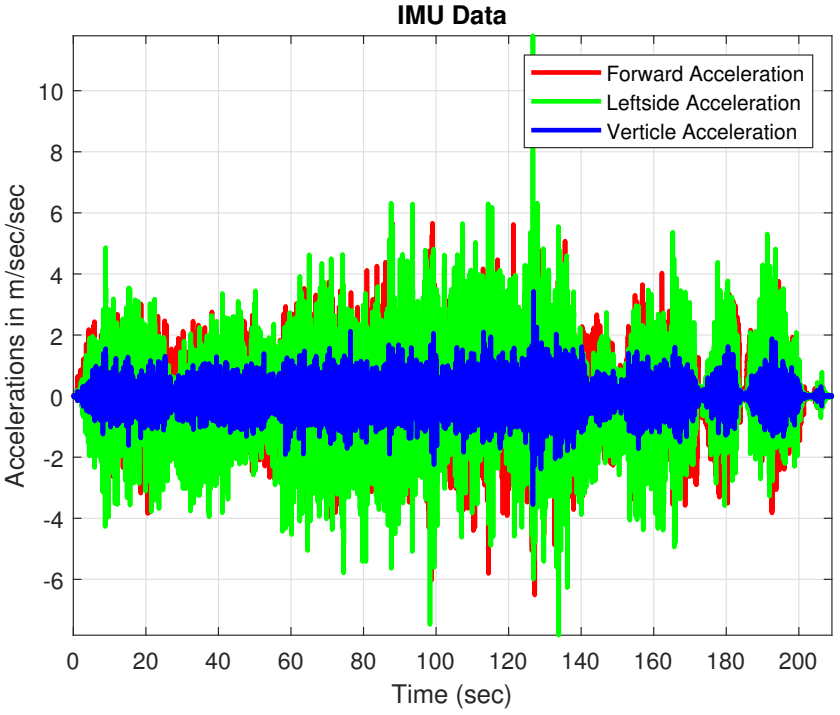

Figure 4.5: Acceleration data from the test drive.

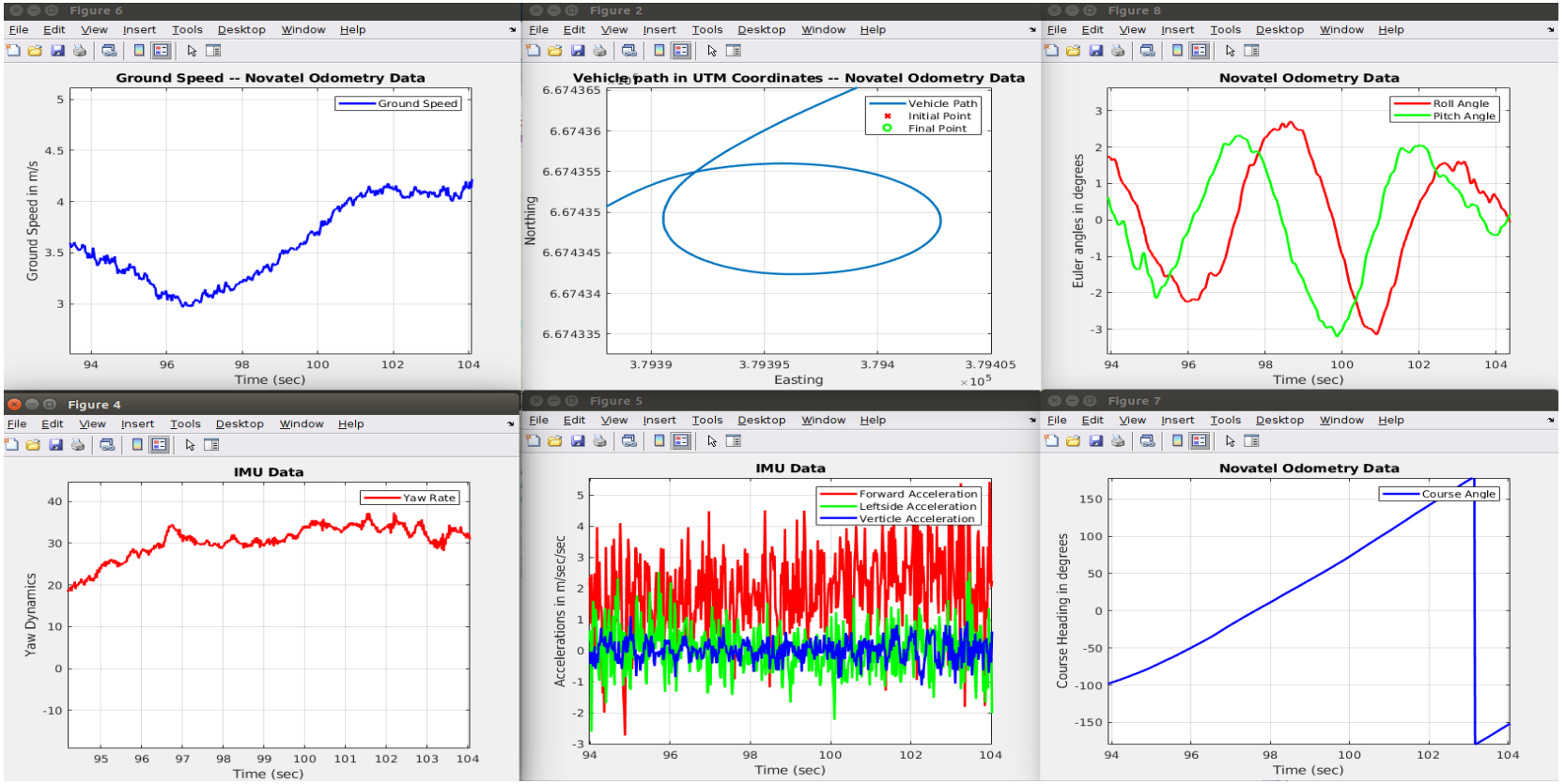

Figure 4.6: Top Left: Ground Speed. Top Middle: Vehicle Path in UTM Coordinates. Top Right: Roll and Pitch Angle. Bottom Left: Yaw Rate. Bottom Middle: Accelerations. Bottom Right: Course Angle. It depicts the relation between speed, turn radius and yaw rate.

4.2.3 Lateral Dynamics Evaluation

The lateral dynamics are of main importance in a 3-DOF vehicle motion. This highlights the significance of performing a 360° loop during the test drive. The mathematical formula, $\mathbf{v} = \mathbf{R} \times \omega$ is put to use in this case. Here, $\mathbf{v}$ being the ground speed, $\mathbf{R}$ is the turn radius and ω is the instantaneous angular velocity. Here $\omega = \dot{\psi}$. This 360° is a counter-clockwise turn; thus, the yaw rate must be positive, and the yaw angle must be increasing.

The used ground speed, v, is provided by SPAN and during the test, the speed was kept near steady around 4 meters per second. The distance measured across the diameter of the loop is around 12 meters, which makes the radius around 6 meters. The site at which the test was conducted was not even, so the change in the roll and pitch angles were also observed.

The measured yaw rate is shown in the bottom left corner of Figure 4.6, and it is following the proper trend. For a counter-clockwise turn the left-side acceleration, which is the acceleration along the body's y-axis must be measured positive. This can be observed in the red curve presented in the bottom-middle section of Figure 4.3. One can see that the legend in this part of the figure shows the red curve as forwarding acceleration. This was due to the misalignment of the SPAN measurement axis with the body axis of Polaris. This test helped to perform the sensor corrections, which was applied to orientation, angular rates, and accelerations.

At time instant, $t = 100$ seconds, $R = 6$, $v \approx 3.6$ the $\omega \approx 36$ degrees per second, which is quite close to the measured yaw rate $\dot{\psi} = 40$ degrees per second. This concludes the discussion of sensor integration. In the next sections, various methodologies applied to address the localization problem of Polaris are presented along with results.

4.3 Kinematics Model of Polaris

A commonly used model for a four-wheeled robot (a car-like vehicle such as Polaris) is the bicycle model, as discussed in [19], is depicted in Figure 4.7. The bicycle has a rear wheel fixed to the body (middle of the rear axle) and the plane of the front wheel rotates about the vertical axis to steer the vehicle (along turn radius, *R*). Therefore, the position and heading angle of the vehicle is calculated with respect to the instantaneous center of curvature (ICC). Here, *L* is the longitudinal wheel separations (wheelbase) or the distance between mean (middle point) of the front and rear axles. The vehicle's velocity is only assumed to be in (positive or negative) x-direction, with no sideways velocity (velocity

along y-direction is zero).

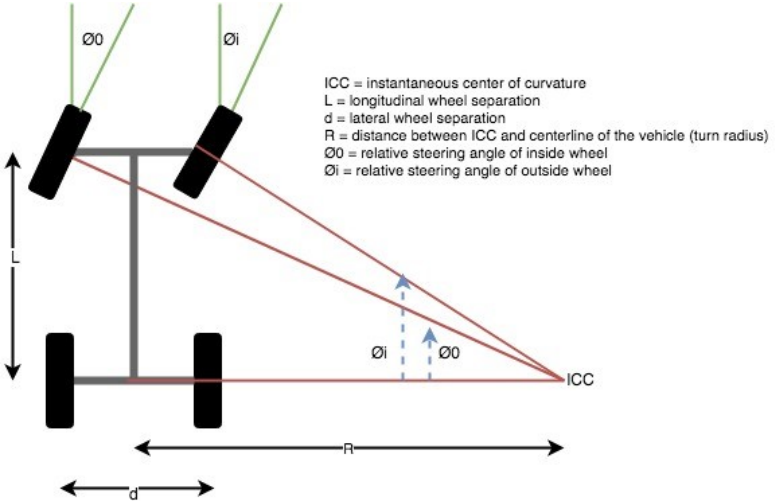

Figure 4.7: 4-Wheeled Robot: Depiction of various parameters.

When a four-wheeled vehicle goes around a corner the two steered wheels follow circular paths of different radius and therefore the angles of the steered wheels, illustrated by ϕ_i and ϕ_o in Figure 4.7, should be marginally altered. The need to trace out circles of different radii for the front wheels led to the invention of the commonly used Ackerman steering mechanism. Such a mechanism allowed the tires to follow the curve without slipping sideways. This results in lower wear and tear on the tires. A differential gearbox (between the motor and the driven wheels) enables the driven wheels to rotate at different speeds on corners [19].

In dead reckoning, the readings of encoders on the wheels, as well as the heading angle, are utilized to update the position and orientation of the robot over time [34]. As discussed in Section 2.5, PLC units transmit these measurements data over the CAN bus. The *atv_can* node listens to this CAN bus and publishes linear velocity of each wheel and steering angle of the vehicle as topics. The *polaris_kinematics_node* then uses this data to estimate the kinematics of Polaris in odom $\rightarrow$ base_link transformation frames. This node calculates the position and orientation of the vehicle as dictated by following a set of difference equations:

$$x_{k+1} = x_k + v \times \delta t \times \cos(\psi_k + \phi_k/2) \tag{4.1}$$

$$y_{k+1} = y_k + v \times \delta t \times \sin(\psi_k + \phi_k/2) \tag{4.2}$$

$$\psi_{k+1} = \psi_k + \tan(\phi_k) \times v \times (\delta t/L), \tag{4.3}$$

where x_k and y_k are position of the vehicle in xy-plane, ψ_k is the orientation (heading) of the vehicle, ϕ_k is the steering angle at time k, v is the calculated

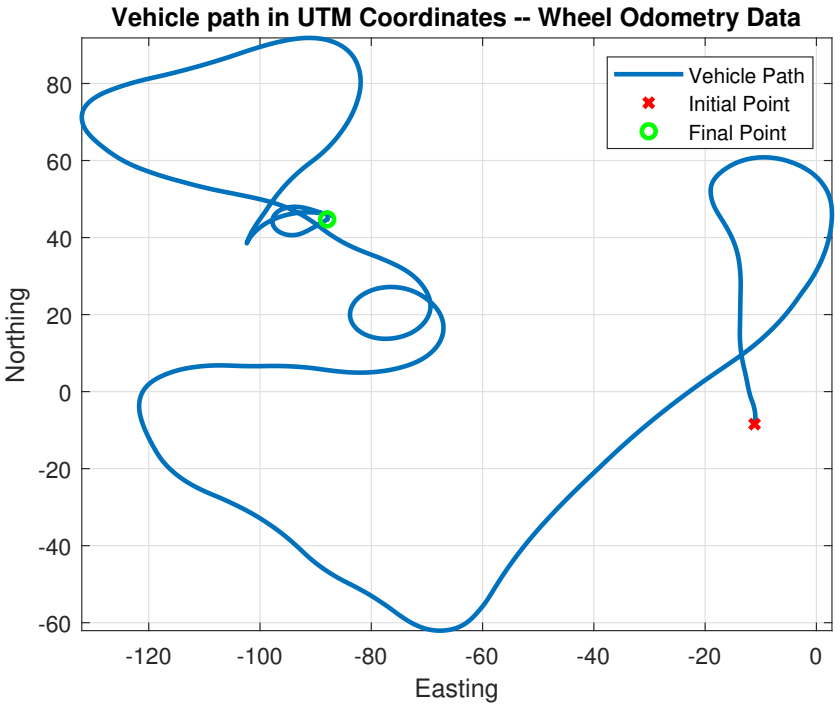

Figure 4.8: Dead Reckoning: Position profile.

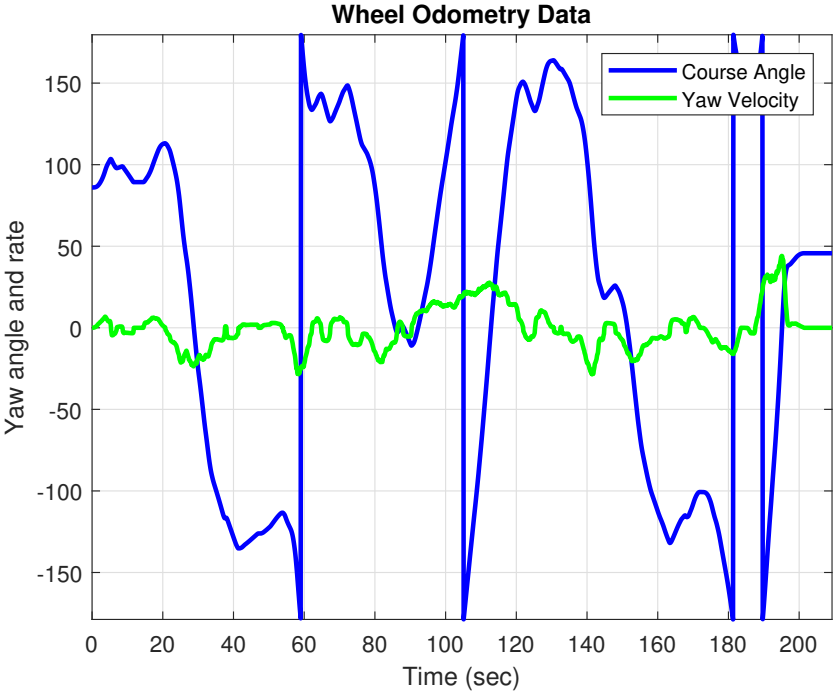

Figure 4.9: Dead Reckoning: Heading dynamics.

average velocity of all four wheels, $\delta t = 0.1$ is sampling time interval and L is the wheel base. $x(k+1)$, $y(k+1)$ and $\psi(k+1)$ are updated position and heading

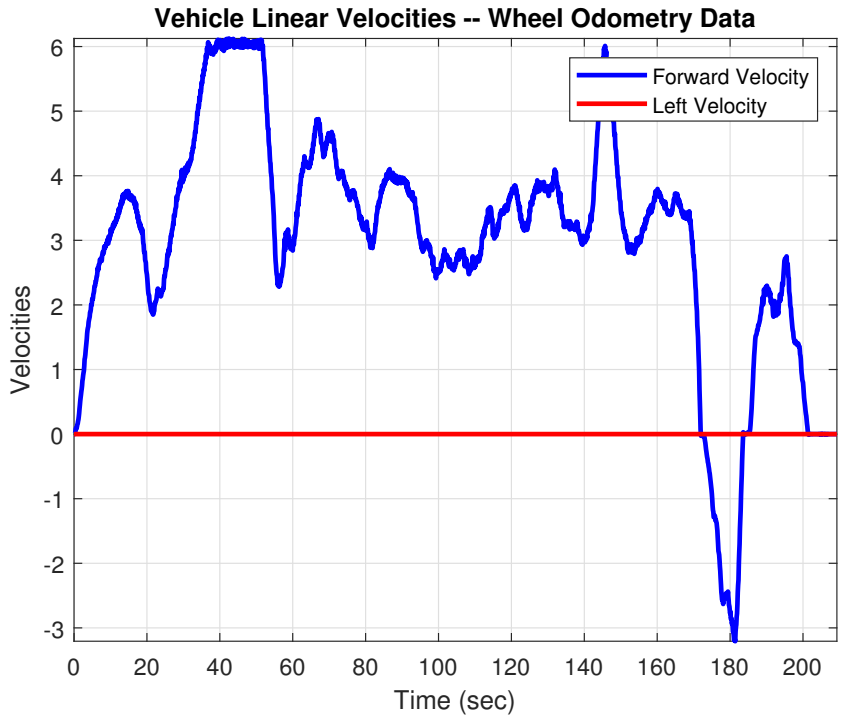

Figure 4.10: Wheel Odometry: Wheel speed.

of the vehicle respectively at time instant $k+1$. The rate of change of heading, $\dot{\psi} = v/R$, is called turn rate or yaw rate can be obtained by differentiating yaw angle with sampling step size δt. Figures 4.8, 4.9 and 4.10 illustrate position, yaw and velocity data during test drive respectively.

This model is called kinematic because it deals with the velocities of the vehicle and not the forces or torques that cause the velocity. However, a drawback of using such a model is that it is susceptible to various sources of errors and thus needs proper calibrations [33]. These errors are mostly induced by wheel slippage, noise from wheel encoder signals, erroneous initial values of measurements, measurement imprecision, and divergences in the system parameters. This highlights the importance of applying modern state estimation and filtering techniques to the robot localization problem.

4.4 EKF-based Vehicle State Estimation

Thomas et. al. [26] did a brilliant job in the implementation of an EKF-based state estimation algorithm for mobile robots within ROS framework [48]. This section begins with a brief discussion about the *robot_localization* package along with its *navsat_transform_node*. Two instances of EKF-based vehicle

state estimation are executed. First estimating the states of vehicle in map $\rightarrow$ base_link and second in odom $\rightarrow$ base_link transformation frames respectively. Figure 3.10 depicted the standard transformation tree implemented during this thesis work.

4.4.1 Robot Localization Package

Salient features of the *robot_localization* package are mentioned below:

(a) It is a general purpose state estimation package, which has no limit on the number of input data sources.

(b) State estimation is based on either EKF or UKF algorithms.

(c) State vector is 15-dimensional, which constitutes:

  (a) Positions: x, y and z coordinates.

  (b) Orientation: roll (ϕ), pitch (θ) and yaw (ψ) angles.

  (c) Linear Velocities: v_x, v_y and v_z.

  (d) Body Rates or Angular Velocities: $\dot{\phi}$, $\dot{\theta}$ and $\dot{\psi}$.

  (e) Linear Accelerations: a_x, a_y and a_z.

(d) It allows per-sensor input customization by setting the limits on the process and measurement covariances.

(e) Covariance is the key to sensor fusion. This is to say that if there are two sensors providing similar data, the *robot_localization* package will pick the one with lower covariance. In other words, data with less noise is used.

(f) It allows imposing a threshold onto the measurement update step of the EKF. This threshold is implemented as a regular Mahanalobis Distance (MD) of the new measurements. Here MD is computed as $D_M(x) = \sqrt{(x - T(x))^T S^{-1}(x)(x - T(x))}$, where x is the new measurement (state), $T(x)$ is the mean of the state vector and $S(x)$ is the covariance matrix associated with that particular state. If the MD of new measurement is greater than the threshold, it will be treated as noise and will be rejected.

(g) It provides a continuous estimation of the states. If there is a loss or jumps in the sensor data, the filter will continue to estimate the robot's state via an internal motion model.

(h) It fuses continuous sensor data e.g. wheel speeds and gyroscope data to produce locally accurate state estimates.

(i) It fuses continuous data with global pose estimates e.g. from SLAM and GPS to produce accurate and complete global state estimates.

(j) It provides the navsat transform node, which is a sensor pre-processing node. Navsat transform node allows users to easily transform geographic coordinates (mostly GPS based coordinates) into the robot's world frame.

4.4.2 Navsat Transform Node

Salient features of the *navsat_transform_node* are listed below:

(a) Idea is to convert GPS data to UTM coordinates.

(b) Use the initial UTM coordinates, EKF/UKF output, and IMU to generate a static transform, *utm*, from UTM grid to Polaris' *world* frame (which is *map* frame in Polaris).

(c) It then transforms all future GPS measurements using this map $\rightarrow$ utm transform as depicted in Figure 3.10.

(d) It takes GPS position data along with the information of the **yaw offset** and **magnetic declination** and transforms it into a frame that is consistent with Polaris starting pose in *map* frame.

Figure 4.11 depicts the calculation of the magnetic declination from an online source. Magnetic declination is the angle between true north and the horizontal trace of the local magnetic field as already depicted in Figure 3.9.

Mathematically, it is creating a rotation matrix given the initial orientation ($\phi = 0, \theta = 0$ and $\psi = -50°$) and position in UTM coordinates along with yaw offset ($90°$) and magnetic declination ($9°$). It then multiplies position vector at each time instant with that rotation matrix. Figure 4.12 shows the position profile during the test drive in UTM coordinates. Furthermore, Figure 4.13 illustrates output of the *navsat_transform_node*. It shows the initial orientation of the vehicle being transformed to origin of the odometry data. ROS Configuration File 1 refers to parameters set for the *navsat_transform_node*.

```
navsat_transform:
  frequency: 50 # The frequency at which it is checks for new
  /novatel/gps/fix
  delay: 3.0
  magnetic_declination_radians: -0.0877 # take negative of
  half of 9.17 +/- 0.44 degrees error according to source:
  https://www.ngdc.noaa.gov/geomag/calculators/magcalc.shtml
```

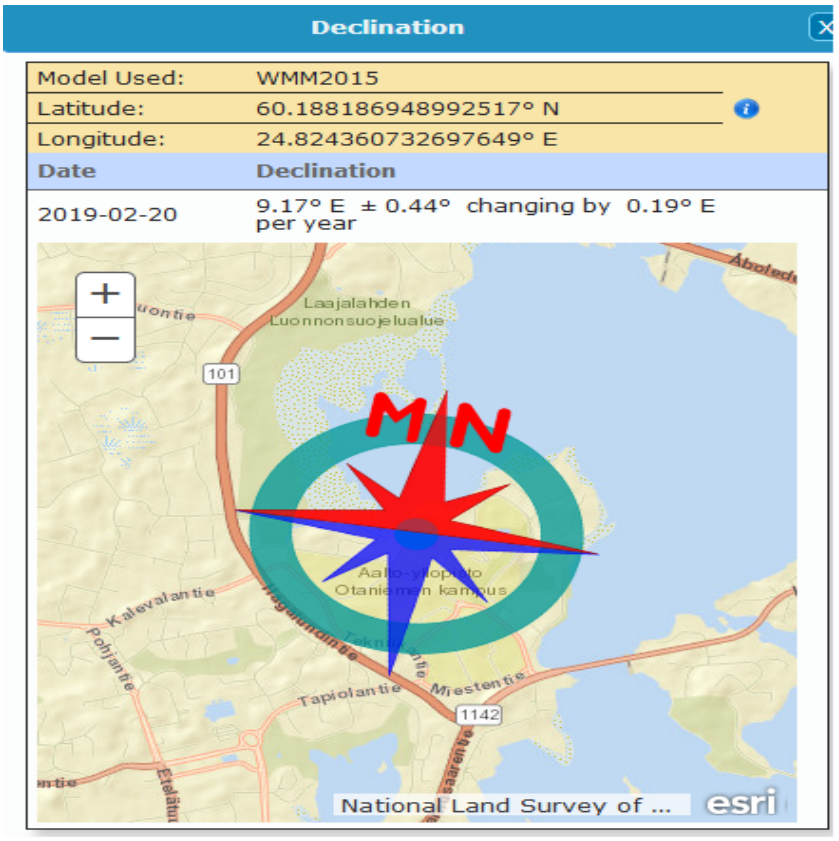

Figure 4.11: Magnetic declination calculations.

```
yaw_offset: -1.5707963 # Take negative to adjust to correct
sign, for both magnetic declination and yaw offset.
zero_altitude: false
broadcast_utm_transform: true
publish_filtered_gps: true
use_odometry_yaw: false
wait_for_datum: false
```

ROS Configuration File 1: navsat_params.yaml

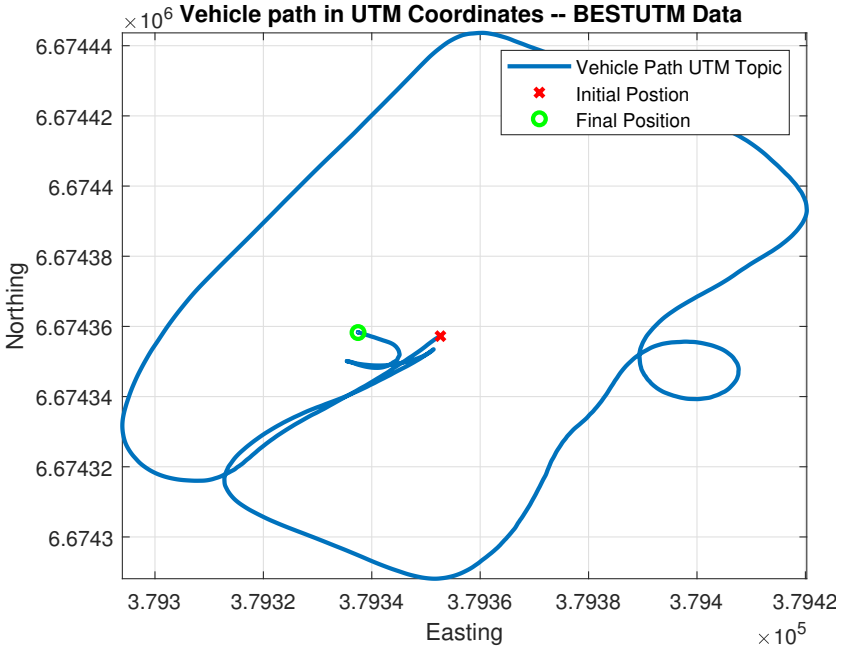

Figure 4.12: Position profile in UTM coordinates.

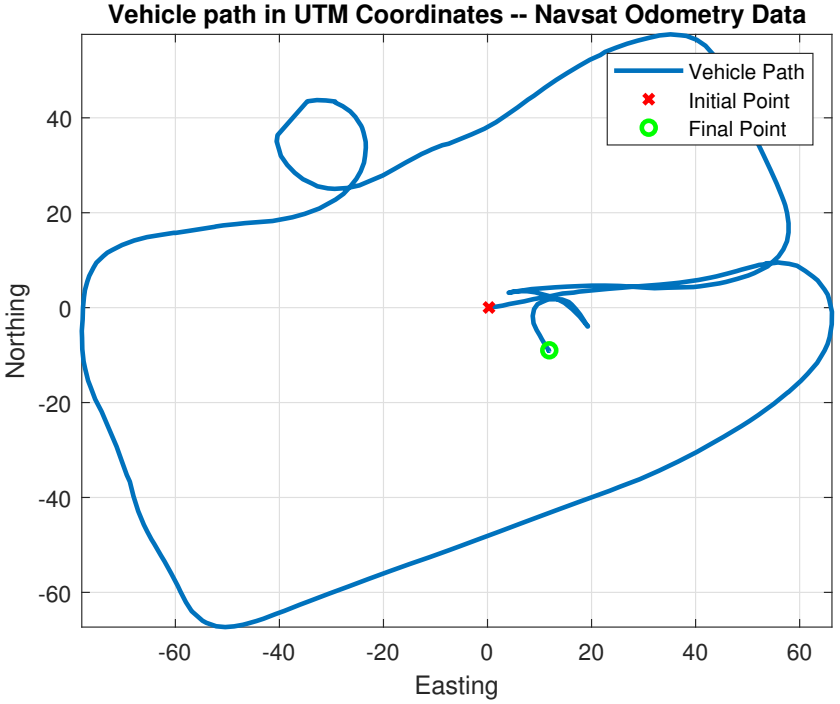

Figure 4.13: Output position profile of *navsat_transform_node*.

4.4.3 Top-level EKF: Map → Base_link Odometry

Following settings were used to fuse the data in map → base_link transformation frames using *robot_localization* package. Figures 4.14, 4.15 and 4.16 illustrate integrated position, yaw dynamics and velocity data respectively.

(a) Positions from *navsat_transform_node*.

(b) Roll and pitch angles from IMU.

(c) Linear velocities from wheel encoders.

(d) Linear velocities from GPS.

(e) Body rates from IMU.

(f) Linear accelerations from IMU.

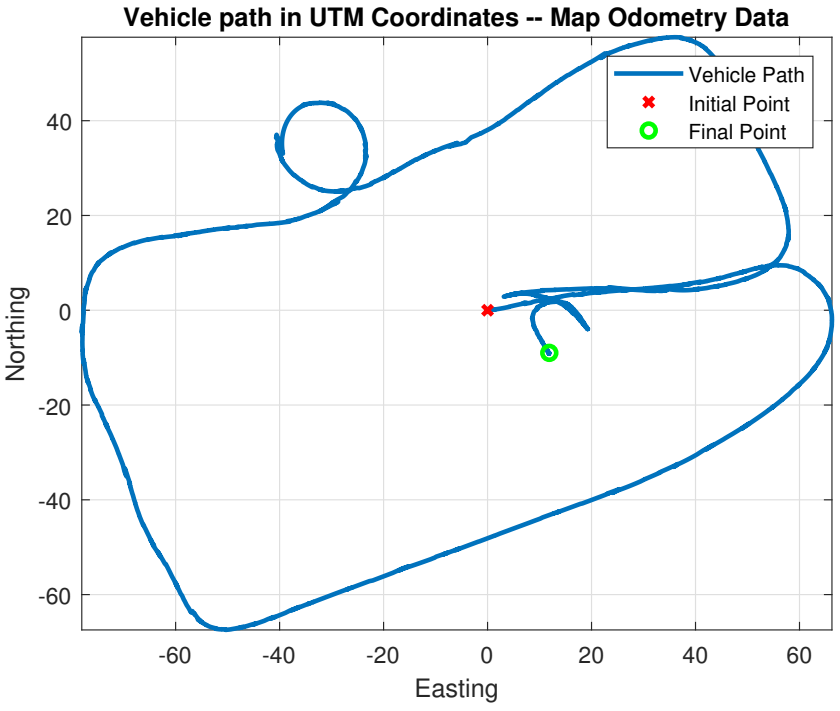

Figure 4.14: Map → base_link Odometry: Position data.

```
# For parameter descriptions, please refer to the template
parameter files for each node.
ekf_se_map:
  frequency: 100
```

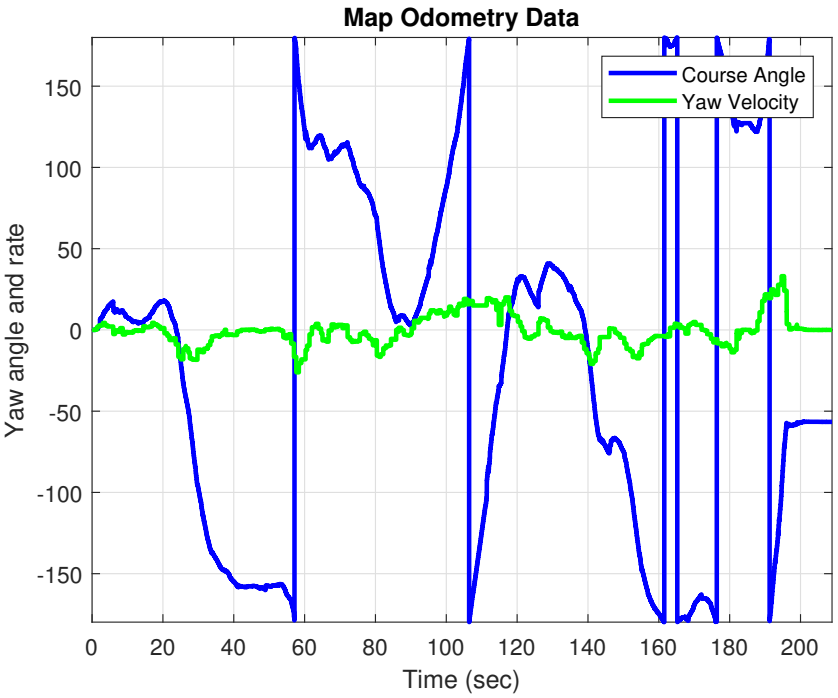

Figure 4.15: Map → base_link Odometry: Yaw dynamics.

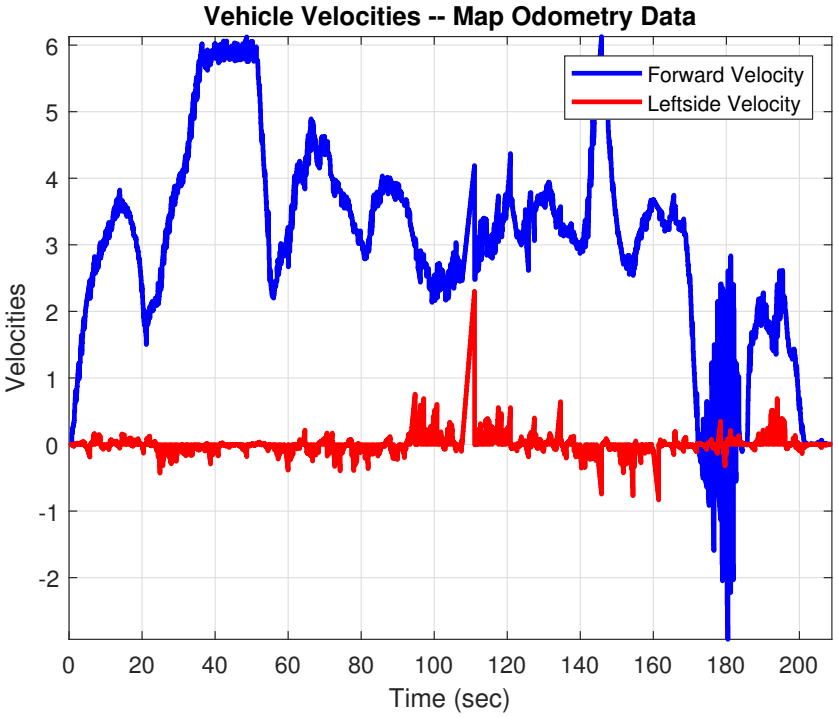

Figure 4.16: Map → base_link Odometry: Velocity dynamics.

```
  sensor_timeout: 0.1
  two_d_mode: false
  transform_time_offset: 0.0
  transform_timeout: 0.0
  print_diagnostics: true
  debug: false

  map_frame: map
  odom_frame: odom
  base_link_frame: base_link
  world_frame: map

# ---------------------------------------------------------
# GPS odometry: Must come from Navsat Handler.
  odom0: odometry/gps
  odom0_config: [true, true, true,
                 false, false, false,
                 false, false, false,
                 false, false, false,
                 false, false, false]
  odom0_queue_size: 10
  odom0_nodelay: true
  odom0_differential: false
  odom0_relative: false # Position data must not be relative.

#
-----------------------------------------------------------------
# Wheel Odometry.
  odom1: /polaris_kinematics/odom_wheel
  odom1_config: [false, false, false,
                 false, false, false,
                 true, true, true,
                 false, false, false,
                 false, false, false]
  odom1_queue_size: 10
  odom1_nodelay: true
  odom1_differential: false
  odom1_relative: false # Position data must not be relative.

# ---------------------------------------------------------
# GPS Odometry: From BESTVEL Data.
  odom2: /novatel/odom_gps
```

```
  odom2_config: [false, false, false,
                 false, false, false,
                 true,  true,  true,
                 false, false, false,
                 false, false, false]  # Include positions
                 feedback from Navsat Node.
  odom2_queue_size: 10
  odom2_nodelay: true
  odom2_differential: false
  odom2_relative: false # Position data must not be relative.


# ---------------------------------------------------------
# Position Data from AMCL Pose Node.

#  pose0: /amcl_pose
#  pose0_config: [true, true, true,
#                 true, true, true,
#                 false,  false,  false,
#                 false, false, false,
#                 false, false, false]
#  pose0_queue_size: 10
#  pose0_nodelay: true
#  pose0_differential: false
#  pose0_relative: false

# --------------------------------------
# IMU configure:

  imu0: /novatel/imu
  imu0_config: [false, false, false,
                true, true, false,
                false, false, false,
                true, true, true,
                true, true, true] # Include rates and
                accelerations measured in Correctrd IMU
                Handler.
  imu0_nodelay: true # Disable Nagle's Algorithm as data is
  coming at high speed.
  imu0_differential: false
  imu0_relative: false # Measured with respect to the initial
  states, especially yaw angle
```

```
  imu0_queue_size: 10
  imu0_linear_acceleration_rejection_threshold: 20.0
  imu0_remove_gravitational_acceleration: false # Corrected
  data removes the gravitational acceleration.

  use_control: false
```

ROS Configuration File 2: ekf_map_params.yaml

ROS Configuration File 2 refers to parameters set for the robot localization node for estimating vehicle state in map → base_link frame.

4.4.4 Low-level EKF: Odom → Base_link Odometry

The following settings were used to fuse the data in odom → base_link frame using the second instance of the *robot_localization* package. The difference is that this instance of EKF-based state estimation uses *odom* as the world frame. Moreover, this EKF is not having any position data updates. So, the pose estimation is totally carried out by integrating velocities and orientation. Figures 4.17, 4.18 and 4.19 illustrate integrated position, yaw dynamics and speed data respectively.

(a) Integrated Positions from EKF (no position corrections).

(b) Roll and pitch angle from IMU.

(c) Linear velocities from wheel encoders.

(d) Linear velocities from GPS.

(e) Body rates from IMU.

(f) Linear accelerations from IMU.

```
# For parameter descriptions, please refer to the template
parameter files for each node.
ekf_se_odom:
  frequency: 100
  sensor_timeout: 0.1
  two_d_mode: false
```

```
  transform_time_offset: 0.0
  transform_timeout: 0.0
  print_diagnostics: true
  debug: false

  map_frame: map
  odom_frame: odom
  base_link_frame: base_link
  world_frame: odom

#
----------------------------------------------------------------------
# Wheel Odometry.
  odom0: /polaris_kinematics/odom_wheel
  odom0_config: [false, false, false,
                 false, false, false,
                 true,  true,  true,
                 false, false, false,
                 false, false, false] # Include velocities
                 from wheel odometry topic.
  odom0_queue_size: 10
```

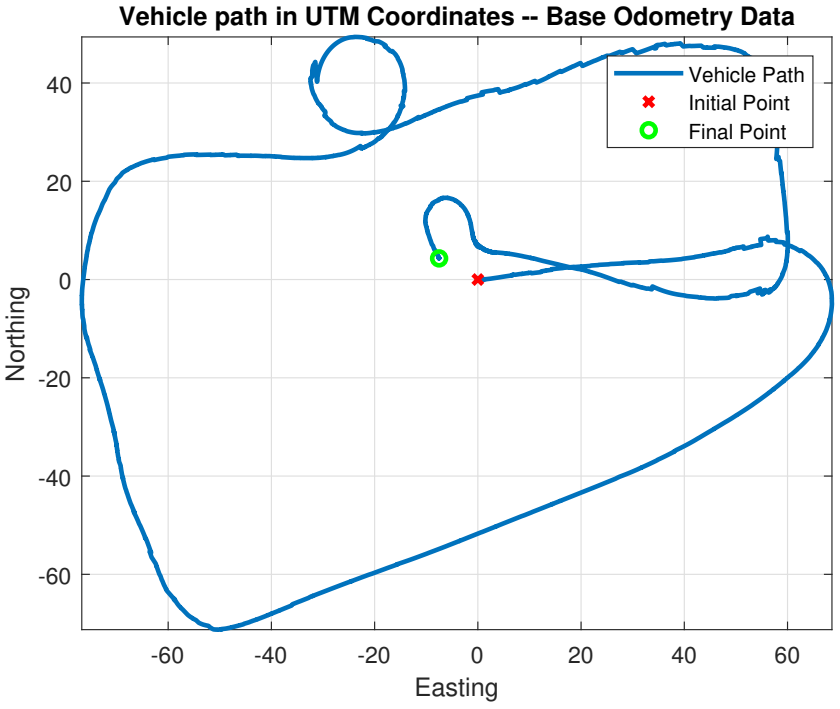

Figure 4.17: Odom → base_link Odometry: Position data.

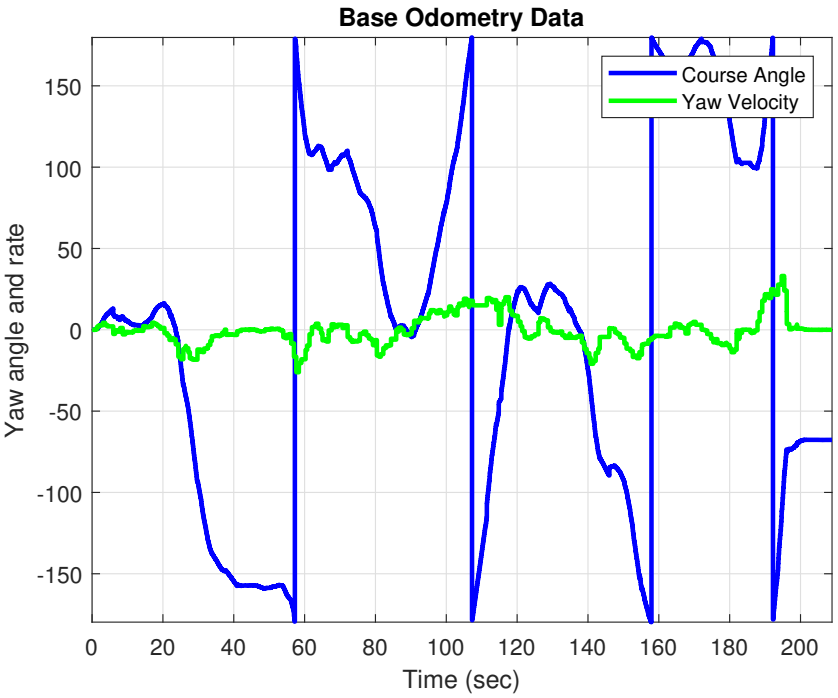

Figure 4.18: Odom → base_link Odometry: Yaw dynamics.

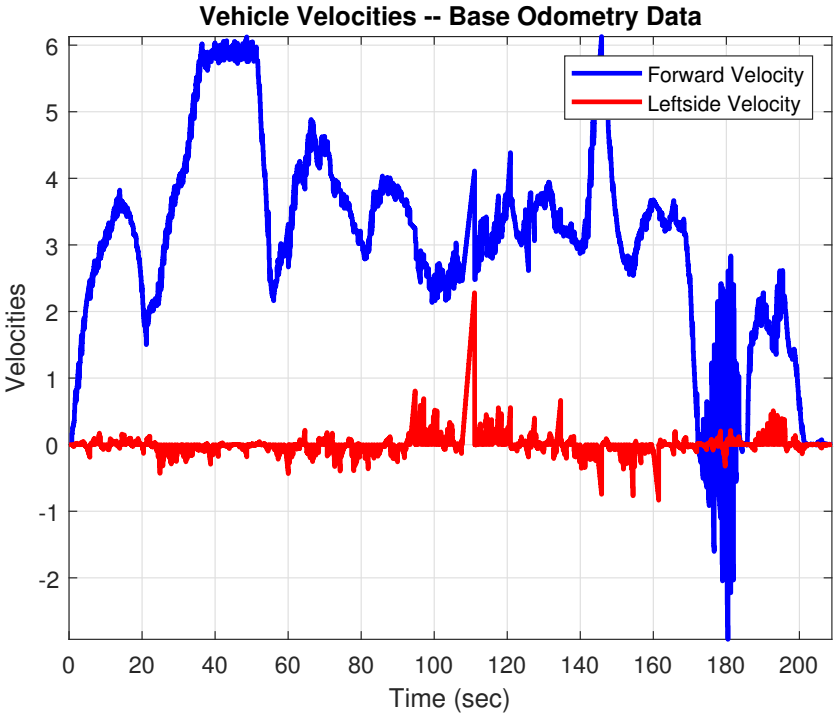

Figure 4.19: Odom → base_link Odometry: Velocity dynamics.

```
  odom0_nodelay: true
  odom0_differential: false
  odom0_relative: false # Position data must not be relative.

#
---------------------------------------------------------------------
# GPS Odometry: From BESTVEL Data.
  odom1: /novatel/odom_gps
  odom1_config: [false, false, false,
                 false, false, false,
                 true,  true,  true,
                 false, false, false,
                 false, false, false]  # Include velocities
                 from BESTVEL topic.
  odom1_queue_size: 10
  odom1_nodelay: true
  odom1_differential: false
  odom1_relative: false # Position data must not be relative.

# ----------------------------------------
# IMU configure:

  imu0: /novatel/imu
  imu0_config: [false, false, false,
                true, true, false,
                false, false, false,
                true, true, true,
                true, true, true] # Include rates and linear
                accelerations made in Correctrd IMU Handler.
  imu0_nodelay: true # Disable Nagle's Algorithm as data is
  coming at high speed.
  imu0_differential: false
  imu0_relative: false # Measured with respect to the initial
  states especially yaw angle
  imu0_queue_size: 10
  imu0_linear_acceleration_rejection_threshold: 20.0
  imu0_remove_gravitational_acceleration: false # Corrected
  data removes the gravitational acceleration.

  use_control: false
```

ROS Configuration File 3: ekf_odom_params.yaml

ROS Configuration File 3 refers to parameters set for another instance of the package for estimating vehicle states in odom → base_link frame.

4.5 Data Analysis

Within the ROS framework, *odometry* is a type of message that includes all the necessary information about the robot motion like position in UTM coordinates, linear and angular velocities. Moreover, the header information of such sentence includes fields such as *frame_id* and *child_frame*. This is of great importance during the sensor fusion step as the robot localization package must have the information about the parent frame and child frame as were depicted in Figure 3.10.

Figure 4.20 shows the positions observed by each of the above-mentioned sources of odometry data. In this figure, the **Map EKF Output** is the position profile in the odometry data of map → base_link transformation frame. It overlaps the position profile of **Navsat Node Output** except during the 360° loop. This is due to the slippage of the vehicle on the hard snow during the turning of the Polaris. It is supported by rapid jumps in speed profiles, which is illustrated by **Map Odometry** shown in red in Figure 4.21.

Figure 4.20 also highlights the importance of setting magnetic declination and yaw offset for navsat transform node. Based on the start (red cross) and finish (green circle) points of the test drive as depicted in Figures 4.14, 4.17 and 4.8; initially vehicle moves with positive Easting from the origin. Furthermore, observe the magenta and red (hidden beneath) in the initial leg of the position data. These two trajectories match quite well until around (-60 Easting, +20 Northing) clockwise turning point. Since, the EKF estimating odom → base_link position does not have the position correction, it was expected.

Another significant point to note is the position profile of different odometry data during the reverse motion of Polaris. Figure 4.21 illustrates the negative ground speeds of all profiles around 160 to 190 seconds. It also illustrates a drawback in using GPS ground speed as it is always positive as it is based on satellite data. It is blind to the direction of the motion of the vehicle. Track over Ground Angle provides an excellent alternative to using ground speed with a correct sign; however, this angle is only accurate when the vehicle is moving with speed of at least 5 meters per second. It also explains the oscillations in **Map Odometry** and **Baselink Odometry** speed profiles around the same period.

The shape of the hook near the origin captures the reverse motion. The position data is near perfect for the map EKF output as well as for the *navsat_transform_node's* position output, as compared to base EKF output. Lastly, the position output based on the kinematic dead reckoning model is far away from the exact trajectory. This was due to the heavy slippage of wheels and an initial non-zero angle of the steering wheel.

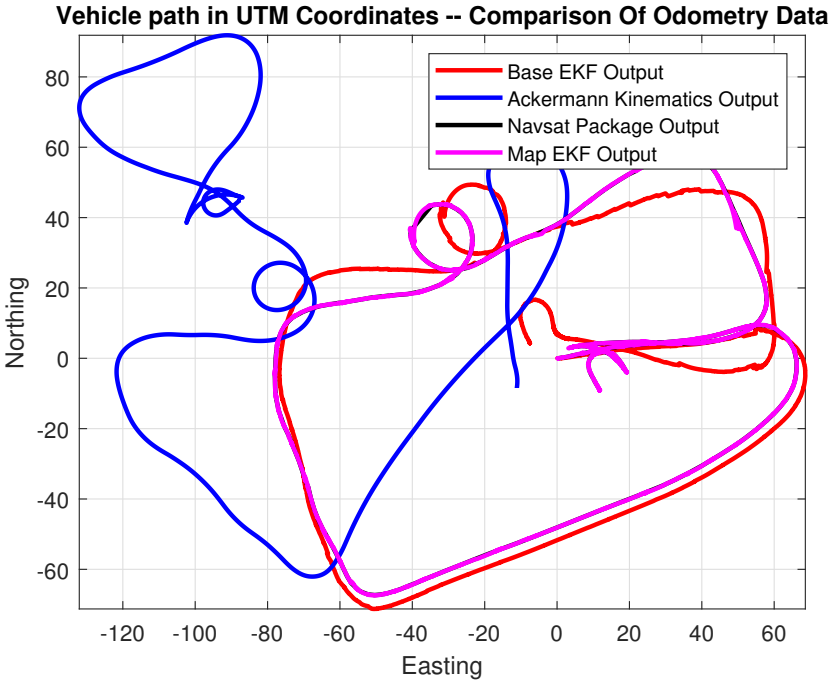

Figure 4.20: Odometry comparison.

In Figure 4.21, one can observe the correct speed sign during the reverse motion of Polaris. This encouraged to use the speed from wheel encoders instead of the GPS ground speed. The position profile of odom $\rightarrow$ base_link odometry is depicted in Figure 4.22. One can observe the gap in the return path, which is clearly due to the slippage of wheels. Moreover, the speed profile was near perfect.

Integrated yaw dynamics from both instances of EKF were shown in Figures 4.15 and 4.18 in Sections 4.4.3 and 4.4.4. It should be observed that both instances of EKFs were not provided with any yaw angle measurement update. This is because the definition of yaw angle may vary from sensor to sensor. In other words, yaw angle (or, simply yaw) may have different meanings. For example, SPAN is providing yaw as an inertial azimuthal angle calculated from IMU gyros and SPAN filters. A magnetometer can also provide the yaw angle in the form of a heading angle with respect to Earth's True North. Course over Ground Angle or Track Angle calculated by GNSS solution can also be used as the yaw angle.

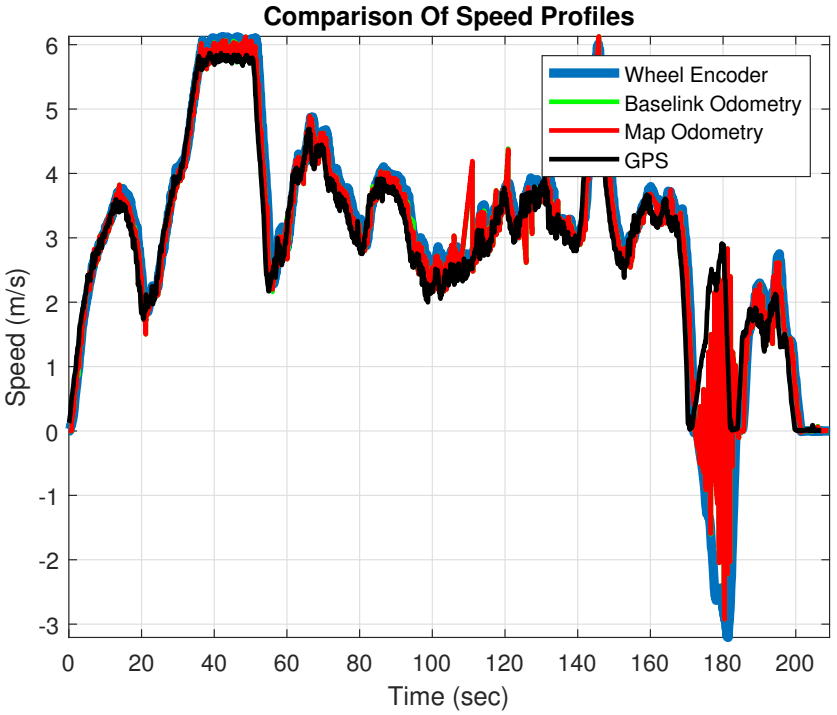

Figure 4.21: Speed profiles comparison.

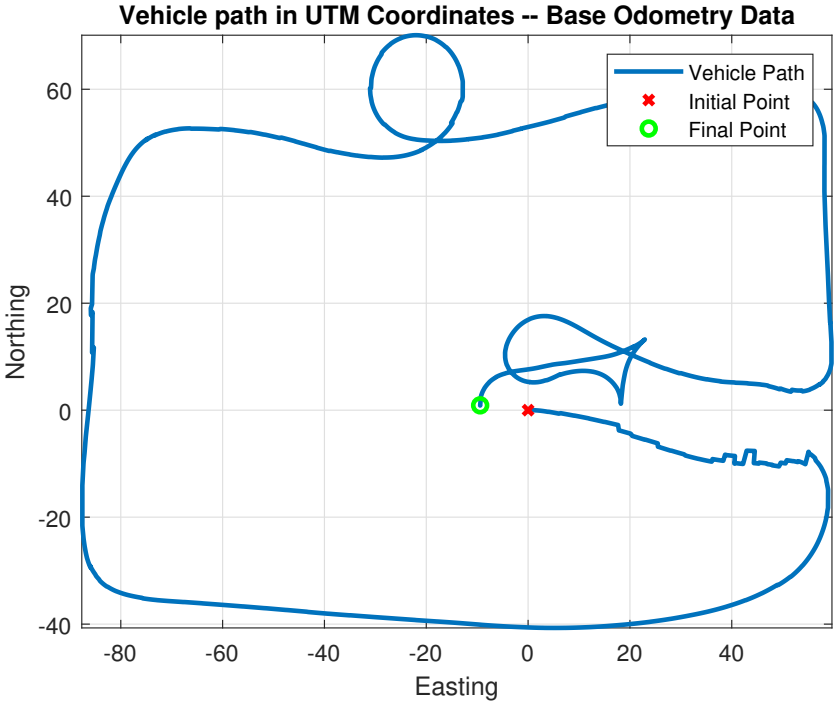

Figure 4.22: Odom → base_link Odometry: Pose data when wheel speed was used.

Another issue that needs to be discussed is the delays in different speed measurements. Since the outputs from both EKFs are expected to be delayed, the GPS ground speed (shown black in Figure 4.21) is updated with less delay when compared to speed from wheel encoders (shown blue in Figure 4.21). The main reason for that may be the number of packages that are involved in the calculation of ground speed based on wheel encoders data. The PLCs are computing the counts of wheel encoders and are transmitting this information via CAN-Bus to another package that is a ROS service. It is responsible for listening to the packet and transmitting this information over the ROS mainframe. The execution frequency of PLCs computers is only 10Hz (way too slow), which may be injecting extra delays into the overall computational cycle.

4.6 GMapping

GMapping is a highly efficient Rao-Blackwellized particle filer to learn grid maps from LiDAR data [31]. Figure 4.24 shows the 2D occupancy grid map obtained by combining the laser data from LiDAR and the map → base_link transformation. It is generated by means of the *gmapping* package in ROS [49]. In our settings, the map_frame is *map*, base_frame is *base_link*. Howe-

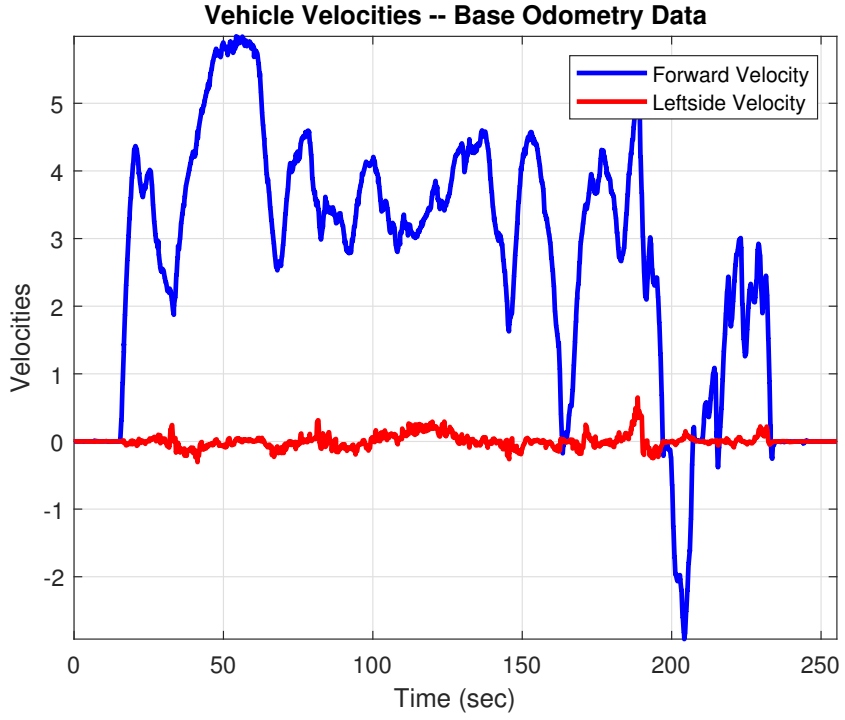

Figure 4.23: Odom → base_link Odometry: Velocity data when wheel speed was used.

ver, it is providing the map → odom_slam transformation as depicted in Figure 3.10, since the odom → base_link transformation is already provided by the *robot_localization* package as implemented in Section 4.4.3.

```
<?xml version="1.0" ?>

<launch>
<arg name="scan_topic" default="/lidar2scan/scan" />
<arg name="map_topic" default="/map" />
<group ns="gmapping">
  <node pkg="gmapping" type="slam_gmapping"
  name="slam_gmapping" output="log">
  <!-- Setting Frames -->
  <param name="map_frame" value="map" />
  <param name="odom_frame" value="odom_slam" />
  <param name="base_frame" value="base_link" />
  <!-- Process 1 out of every this many scans (set it to a
  higher number to skip more scans)  -->
  <param name="throttle_scans" value="1"/>
  <param name="map_update_interval" value="0.5" /> <!--It was
  1.5 -->
  <!-- The maximum usable range of the laser. A beam is
  cropped to this value.  -->
  <param name="maxUrange" value="40" />
  <!-- The maximum range of the sensor -->
  <!--If regions with no obstacles within the range of the
  sensor should
  appear as free space in the map.-->
  <param name="maxRange" value="80" />
  <param name="temporalUpdate" value="0.05" /> <!--It was
  -1-->
  <param name="linearUpdate" value="5.0" />
  <param name="angularUpdate" value="1.0" />
  <param name="particles" value="50" />
  <param name="minimumScore" value="100" />
  <param name="stt" value="0.01" />
  <param name="srr" value="0.01" />
  <param name="str" value="0.01" />
  <param name="srt" value="0.01" />
  <param name="delta" value="0.1" />
  <remap from="scan" to="$(arg scan_topic)"/>
  <remap from="map" to="$(arg map_topic)"/>
  </node>
```

```
</group>
</launch>
```

ROS Configuration File 4: gmapping.launch

It can be observed that the trajectory in the grid map is one that is obtained from map $\rightarrow$ base_link odometry data. The update rate of the map was set to 0.5 seconds. Launch File 4 depicts the settings of the *gmapping* package.

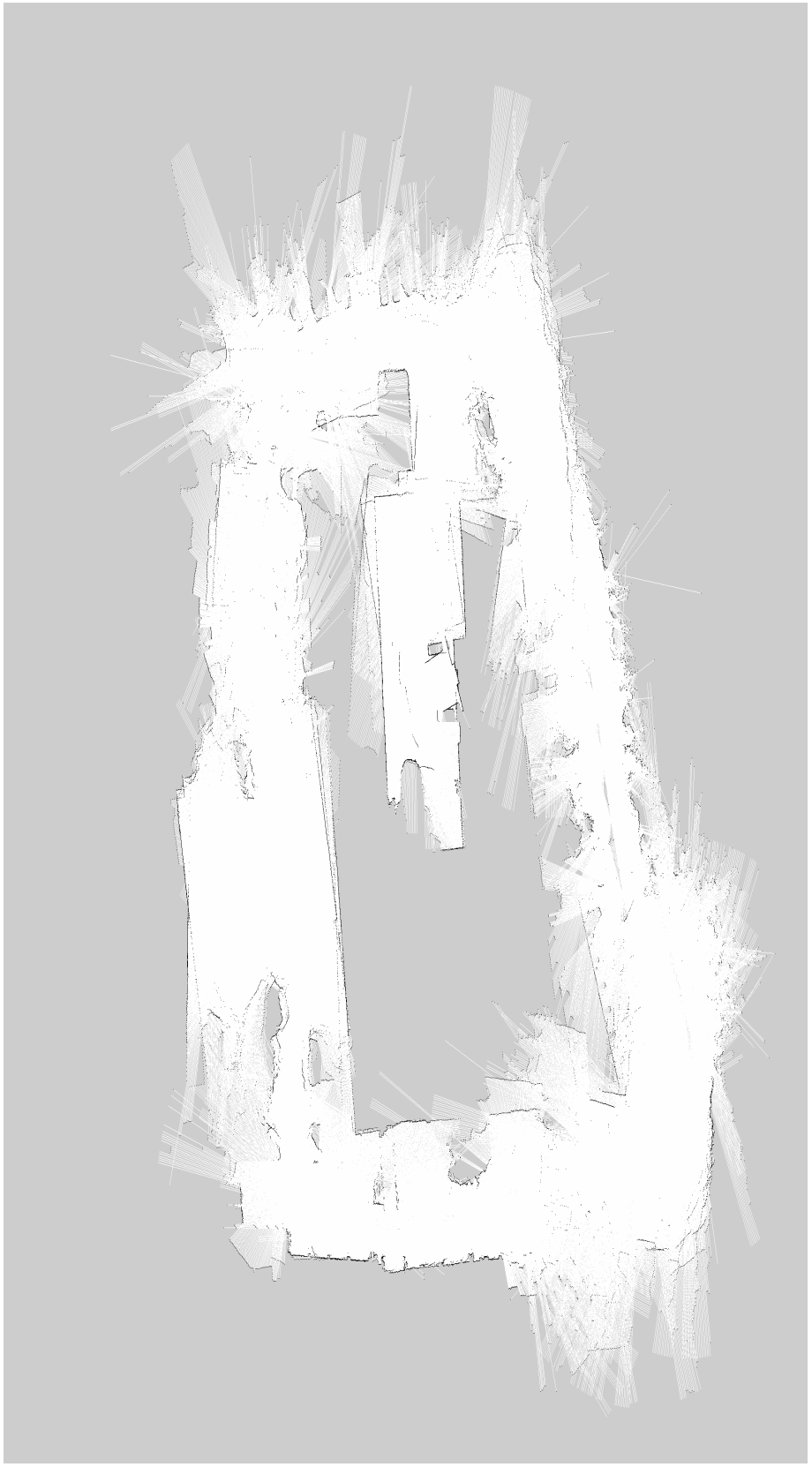

Figure 4.24: 2D Occupancy Grid Map.

5 Navigation and Control

In this chapter, the implementation of autonomous functionalities in Polaris is discussed. The implementation of the navigation module and control system module are presented with relevant background and results. The ROS configuration files are attached along with the discussion, and not in a separate appendix to the thesis document. As the intention is to inform the readers about the important ROS parameters. With the important theory covered in Chapter 2, mostly the implementation part is covered in this chapter.

5.1 Navigation Module Implementation

The implementation of the navigation module was highlighted in Figure 2.1 earlier in Section 2.4. As said earlier, it is the responsibility of the path planning algorithm to define a sequence of schemes to move the mobile robot from one position to another while avoiding at the same time all obstacles along its path. In ROS, the implementation of the global motion planning of the robot is based on the graphs based methods. Firstly, a cost map is generalized from the occupancy grid map (in 2D). The 2D occupancy grid map, as was shown in Figure 4.24, is provided as a topic by the *gmapping* package to the navigation module in the ROS setup.

The generation of the cost map in the navigation module is managed by the *costmap_2d* package [50]. With this map, a trajectory traversing through the cells with the lowest costs can be generated with avoiding obstacles. Figure 5.1 shows the cost map which indicates the free space for the e-ATV to travel through in a test. The cost map limits the free space by the blue-colored region which is corresponding to the walls and obstacles recognized by the LiDAR. There are two implementations of the cost maps in the Polaris. The global cost map is used for the global motion planning while the local cost map is used for the local navigation.

The blue rectangular box in the middle of the Figure 5.1 represents the footprint of the Polaris Ranger as per its specifications provided in the datasheet [55]. It is important as the cost assigned to the cells will depend on the characteristics of the robot. These characteristics were defined in the ROS Configuration File 5. Since the Polaris is of larger size, parameters such as *obstacle_range*, *raytrace_range* and *inflation_radius* were set accordingly. The inflation is the process of propagating cost outwards from each occupied cell to the inflation radius as shown by the region colored in cyan in Figure 5.1. The value of this parameter was set lower to allow the vehicle to pass through closed spaces wit-

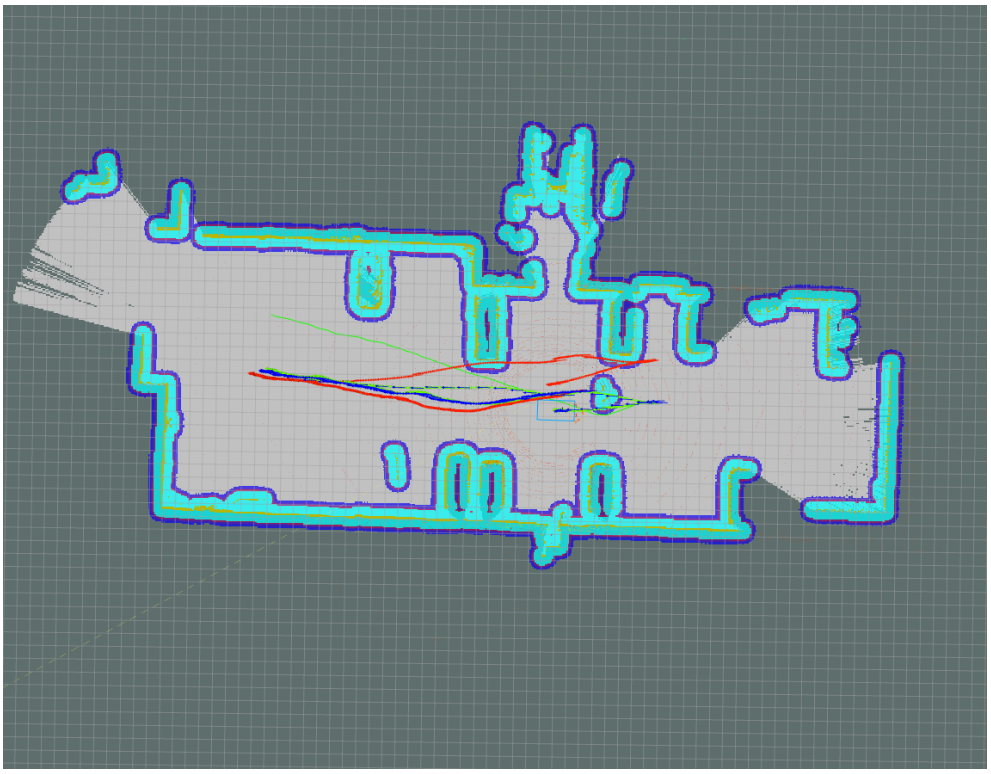

Figure 5.1: A 2D cost map generated by in ROS.

hout halting. However, in the future, an optimal value should be selected via thorough testing to address the safety concerns of the Polaris during a completely autonomous drive.

```
origin_z: 0.0
z_resolution: 1 # The z resolution of the map in meters/cell.
z_voxels: 2  # The number of voxels to in each vertical
column, the height of the grid is z resolution * z voxels.

obstacle_range: 1.0 # The default maximum distance from the
robot at which an obstacle will be inserted into the cost map.
raytrace_range: 3.0 # The default range in meters at which to
raytrace out obstacles from the map using sensor data
inflation_radius: 1.0 # controls how far away the zero cost
point is from the obstacle
cost_scaling_factor: 1 # slope of the cost decay curve with
respect to distance from the object. lower makes robot stay
further from obstacles

#---standard polaris footprint---
```

```
#---(in meters)---
footprint:  [ [2.15, -0.75], [-0.65, -0.75], [-0.65, 0.75],
[2.15, 0.75] ]
transform_tolerance: 0.2
map_type: costmap
publish_voxel_map: false
observation_sources: laser_scan_sensor

laser_scan_sensor: {sensor_frame: laser, data_type: LaserScan,
topic: scan, marking: true, clearing: true}
```

ROS Configuration File 5: common_costmap_params.yaml

For the autonomous test drive, the *observation_sources* parameter was limited to utilize only 2D laser scan data. However, the 3D point cloud data obtained from the LiDAR could also be used in conjunction with the laser scan data. ROS Configuration File 6 defines the parameters for the global cost map. The frequency at which the map is updated is 20 Hz, while the publishing frequency was set to 5 Hz. The aim was to reduce the computational complexity involved in displaying the cost map and to put more resources in computing the cost map. The *static_map* parameter was set to false, as it is only useful for the off-line ROS runs when the map of the environment is available beforehand.

```
global_costmap:
  global_frame: map
  robot_base_frame: base_link
  update_frequency: 20.0
  publish_frequency: 5.0
  width: 100.0
  height: 100.0
  static_map: false #true
  rolling_window: true
```

ROS Configuration File 6: global_costmap_params.yaml

Local cost map parameters are set in ROS Configuration File 7. The *rolling_window* parameter specifies that the Polaris will always be in the center of both global and local cost maps. For both cost maps, the *global_frame* is set to the *map* frame and the *robot_base_frame* is set to *base_link* frame. The map $\rightarrow$ base_link transformation is provided by the *robot_localization* package as

discussed in Section 4.4.3. Through such configurations, the obstacles detection and avoidance takes place within the area defined by the *height* and *width* (parameters) of the (global as well as local) cost maps. The selection of the height and width of the global cost map is dictated by how far the target waypoints are selected, otherwise, the global planner will not work. A rectangular global cost map might be a good idea to implement in the future.

```
local_costmap:
  global_frame: map # odom
  robot_base_frame: base_link
  update_frequency: 5.0
  publish_frequency: 10.0
  static_map: false
  rolling_window: true
  width: 30.0
  height: 30.0
  resolution: 0.05 # The resolution of map is meters per cell.
```

ROS Configuration File 7: local_costmap_params.yaml

The *move_base* package is responsible for moving a robot to the desired positions [51]. The *move_base* node links together a local and global planner to accomplish their navigation tasks. It may optionally perform recovery behaviors when the robot perceives being stuck. Local planner chooses appropriate velocity commands for the e-ATV to traverse the current segment of the global path. It combines the odometry in map $\rightarrow$ base_link frame with both global and local cost maps. It can recompute the path on the go to keep the Polaris avoiding obstacles and allowing it to reach its destination. The *base_local_planner* package provides implementations of either Trajectory Roll-Out, or Dynamic Window Approach (DWA) algorithm [52, 29, 30].

The idea of the DWA algorithm is based on Monte Carlo planner with cost evaluation which is illustrated in Figure 5.2. However, the basic idea for both local trajectory planners is similar. Descriptions about the local planner parameters are included in the ROS Configuration File 8. The salient features of the local planner are listed as follows:

(a) Define the Polaris' *control space* in *map* frame. The trajectory is computed as a set of three variables which comprises of x-position, y-position, and yaw angle (a 3-DOF motion simulator).

(b) For each velocity measurement, simulate future trajectories from the Pola-

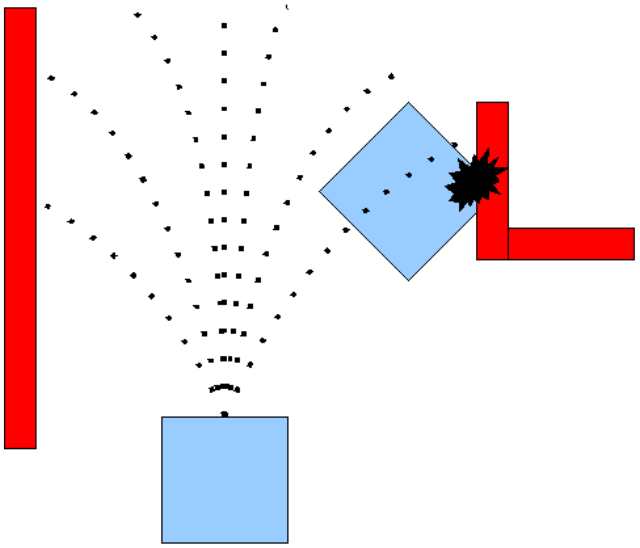

Figure 5.2: Simulation of trajectories by the local planner [52].

ris' current state at 10 Hz. These trajectories are simulated to predict what would happen if the measured (linear and angular) velocities are applied for the duration of 5 seconds. These velocities are applied within limits.

(c) Evaluate the cost of traversing each simulated trajectory after every 0.02 seconds. The cost evaluation is based on the closeness to obstacles (*occdist_scale*), vicinity to the local path (*pdist_scale*), and the proximity to the goal (*gdist_scale*).

(d) Discard those (simulated) trajectories that might involve a collision with obstacles.

(e) Pick the highest-scoring trajectory and send the associated velocities as twist commands to steer the Polaris towards the target position.

(f) After sensing a new (linear or angular) velocity measurement, clear the trajectories which were computed in the previous step, and repeat the whole process again.

```
TrajectoryPlannerROS:
  global_frame_id: map # Should be set to the same frame as
  the local costmap's global frame.
  publish_cost_grid_pc: true # Whether or not to publish the
  cost grid that the planner will use when planning. Default
  is false.
  max_vel_x: 5.0
```

```
 min_vel_x: 0.15 # Allowing velocities too low will stop the
 obstacle avoidance because low velocities won't actually be
 high enough to move the robot
 max_vel_theta : 1.0
 min_vel_theta : -1.0
 min_in_place_rotational_vel: 0.4
 escape_vel: -0.2
 acc_lim_th: 3.2
 acc_lim_x: 2.5
 acc_lim_y: 2.5
 holonomic_robot: false

# Goal Tolerance Parameters
 yaw_goal_tolerance: 0.1  # in rads
 xy_goal_tolerance: 0.2  # in meters
 latch_xy_goal_tolerance: false

 # Forward Simulation Parameters
 sim_time: 5.0   # setting time of each simulation that it
 must evaluate. Higher will create longer curves but too low
 can limit performance (<2)
 sim_granularity: 0.02 # the step size to take between points
 on a trajectory, or how frequent should the points on this
 trajectory should be examined
 angular_sim_granularity: 0.02
 vx_samples: 10  # how many samples of x velocity are taken
 for simulated trajectories
 vtheta_samples: 30 # how many samples of theta velocity are
 taken for simulated trajectories
 controller_frequency: 10.0  # how often the planning
 algorithm is performed (hz)

 # Trajectory scoring parameters
 meter_scoring: true # Whether the gdist_scale and
 pdist_scale parameters should assume that goal_distance and
 path_distance are expressed in units of meters or cells.
 Cells are assumed by default (false).
 occdist_scale:  0.1 # The weighting for how much the
 controller should attempt to avoid obstacles. default 0.01
 pdist_scale: 0.5  # The weighting for how much the
 controller should stay close to the path it was given .
 default 0.6
```

```
gdist_scale: 1.0 # The weighting for how much the controller
should attempt to reach its local goal, also controls speed
default 0.8
heading_lookahead: 5.0 # was 0.325  # How far to look ahead
in meters when scoring different in-place-rotation
trajectories
heading_scoring: true  # Whether to score based on the
robot's heading to the path or its distance from the path.
default false
heading_scoring_timestep: 5.0 # was 0.8   # How far to look
ahead in time in seconds along the simulated trajectory when
using heading scoring (double, default: 0.8)

dwa: true # Whether to use the Dynamic Window Approach
(DWA)_ or whether to use Trajectory Rollout
simple_attractor: false
publish_cost_grid_pc: true

# Oscillation Prevention Parameters
oscillation_reset_dist: 0.25 # How far the robot must travel
in meters before oscillation flags are reset (double,
default: 0.05)
escape_reset_dist: 0.1
escape_reset_theta: 0.1
```

ROS Configuration File 8: base_local_planner_params.yaml

The global planner is implemented as a *navfn* package within the navigation module [53]. It is a simplified A* type. The *navfn* package performs a Dijkstra's search through the available working area in order to find the best path. It begins at the robot's origin and radiates outward, checking each non-obstacle cell in turn until the goal is reached. Essentially, it uses the *breadth-first* search to check the cells of the cost map for the optimal path.

The optimal path of the *navfn* planner is based on its *potential*. The potential is the relative cost of a path based on the distance from the goal and from the existing path itself. The *navfn* algorithm updates each cell's potential in the potential map, as it checks that cell. Through such a technique, it can step back through the potential field to find the best possible path. The potential is determined by the cost of traversing a cell and the distance away that the next cell is from the previous cell. In our ROS settings, the default parameter values

for the *navfn* package were used.

5.2 Control System Module Implementation

It is worthwhile to discuss a few points about the selection of double (practically similar) instead of a single controller to move the vehicle through the set of waypoints. Firstly, it is important to mention here that the *move_base* is an action server (ROS actions were discussed in Section 3.3). It listens to the goals set by the mission controller (action client) and performs appropriate movements in order to achieve the goal. A single instance of *move_base* node was enough to make the vehicle move from one waypoint to the next. However, it added a significant amount of delays in the control loop whenever a goal has to be published; updated or canceled.

Furthermore, *move_base* being an action server keeps on chasing the target unless it achieves all the set of parameters. These parameters include the final position with respect to the target as well as the commanded orientation. On the other hand, in reality, we are not actually aiming to achieve the target position within the accuracy of few centimeters or even meters. Since the footprint of Polaris is quite large as compared to other robots, it is desirable to switch to the next goal as soon as the e-ATV is in the ballpark of the established target position. Therefore, once the vehicle achieves the goal within a user-defined tolerance limit, the preemption of the *move_base* goal-oriented tasks is required. Such task preemption in the *hard real-time embedded systems* has timing consequences [22]. Therefore, to meet the timing requirements of the control loop, the double controller strategy seemed to be a better idea to implement.

Henceforth, the implementation of the double controller is such that each mission controller (primary and secondary) calls an instance of *move_base* action server. As the name suggests, the mission controller will read the target waypoints. It will convert them into goals in the appropriate transformation frames. At first (for instance), the primary mission controller will send a goal to the *move_base* action server as an action client. As soon as the Polaris reaches the goal within a tolerance limit, the control authority is switched to the secondary mission controller. Through switching the control authority, the latencies involved in the goal cancellation and updating phases were avoided. By design, the waypoints for each controller are set in a way that as soon as one finishes its job, the other controller already sets the next target waypoint as its goal.

This also highlights the importance of implementing a switching controller that decides when to direct the correct commands from the active controller to the base level actuator control algorithm (motion controller in our case). In addition,

a security feature in the form a software dead man's switch is also included in the switching controller's design. Further, the constituent parts of the control system module are discussed.

5.2.1 Mission Controller

Figure 5.3 depicts the implementation of mission controller algorithm in ROS. The salient features of *mission_controller* package are listed below.

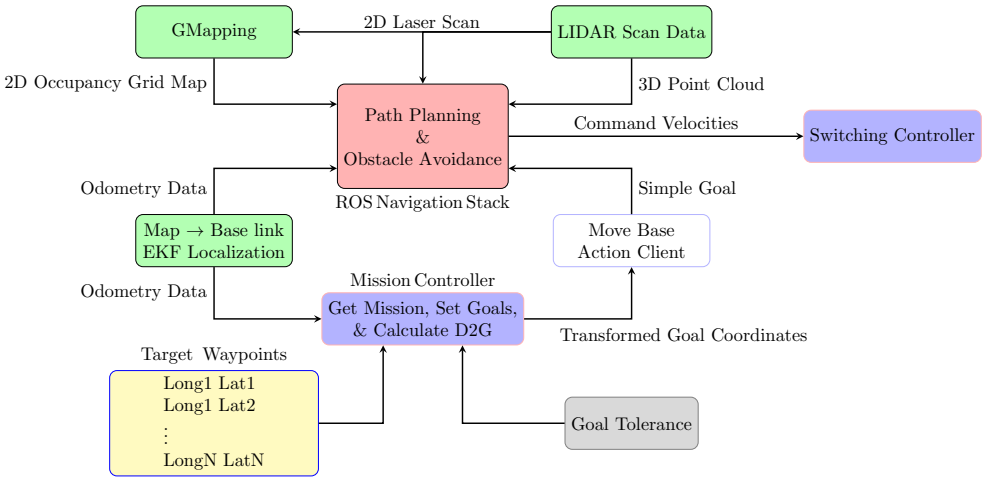

Figure 5.3: Functionality of Primary (or Secondary) Mission Controller.

(a) The idea is to run two instances of *move_base* package, each with a different set of target waypoints.

(b) The *primary_mission_controller* node issues the goals from target waypoint 1 to 2, then 3 to 4, and so on till second-last to last point.

(c) The *secondary_mission_controller* node issues goals from target waypoint 2 to 3, then 4 to 5, and so on till third-last to second-last point.

(d) Main input is the integrated odometry in map → base_link frame obtained from an instance of *robot_localization* node.

(e) Reads the target waypoints in Long-Lat format from a text file.

(f) Transforms each target waypoint using map → utm transformation.

(g) Sends the transformed goal to the navigation module for path planning and obstacle avoidance.

(h) ROS Configuration File 9 shows the parameter settings for both instances of the *move_base* controller.

```
shutdown_costmaps: false

controller_frequency: 20.0
controller_patience: 15.0

planner_frequency: 20.0
planner_patience: 5.0

oscillation_timeout: 0.0
oscillation_distance: 0.5

recovery_behavior_enabled: true
clearing_rotation_allowed: false # was true initially
```

ROS Configuration File 9: move_base.yaml

As discussed earlier, the criterion for achieving the target is based on the *goal tolerance* set for both mission controllers. The primary mission controller is calculating the *Distance to Go (D2G)* to the goal using the distance formula, $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. As soon as the goal tolerance is met, primary mission controller issues a *controller_done* signal to the secondary mission controller. After the secondary mission controller gets the signal, it takes over the control authority to move the Polaris to the next goal. In the meanwhile, the primary mission controller cancels the last goal and calculates the new goal to be sent later.

As discussed earlier, the goal achievement with a precise position and orientation was never intended. So, there was no need for applying the clearing rotations (defined by the *clearing_rotations_allowed* parameter) to the Polaris once it is in the ballpark of a target waypoint. Moreover, the frequency at which the control loop was executed (defined by the *controller_frequency* parameter) was 20 Hz. Since the power steering control and speed control loops were operating (inside the PLCs) at 10 Hz, sending the velocity commands at a higher rate would be futile. The recovery behavior (defined by the *recovery_behavior_enabled* parameter) was enabled but was ineffective as the ability of the reverse motion was not implemented in the autonomous driving mode.

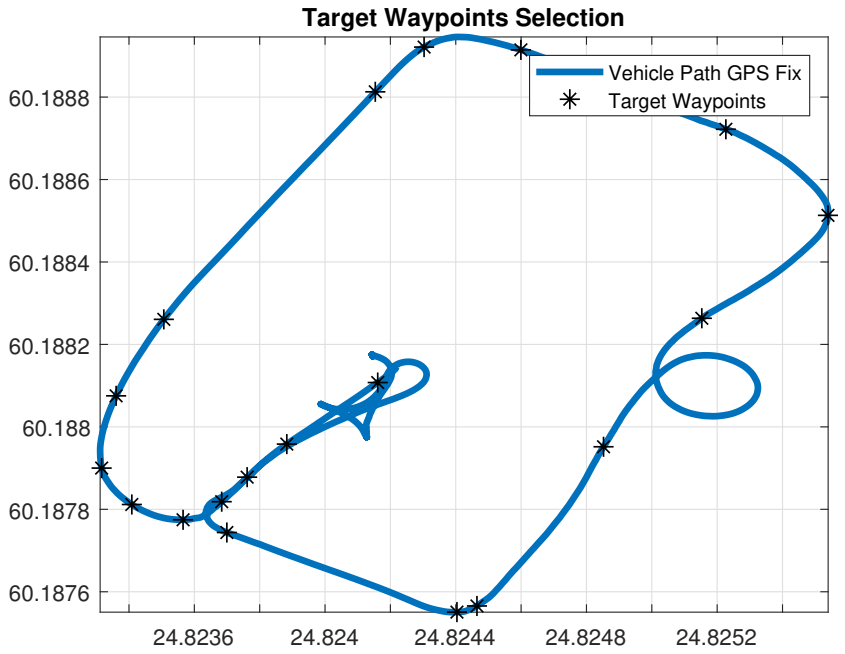

Figure 5.4: The selection of target waypoints from the test drive.

5.2.2 Mission Profile

The selection of target waypoints from a test drive of Polaris is depicted in Figure 5.4. This is called the **mission profile**, which shows each selected waypoint as a black star. These waypoints are carefully selected in order to traverse the vehicle from the start point to the finish point. Note that for the closed areas and turning points, the distance between two consecutive target waypoints was kept as small as possible. For straight and level drive, the distances between waypoints were kept longer. Moreover, the first and the last waypoints were kept similar in order to close the loop so to speak.

5.2.3 Switching Controller

Switching controller block as depicted in Figures 5.3 and 5.5 is implemented as a node within *mission_controller* package in ROS. The functionality of this node is listed below.

(a) Reads twist (linear and angular velocity) commands from each mission controller as shown in Figure 5.5.

(b) Checks for the valid (non-zero as well as varying) twist commands along with the *controller_done* signal from both mission controllers.

(c) Routes only valid twist commands to the motion controller.

(d) Issues goal cancellation requests to the inactive (primary or secondary) *move_base* package.

(e) As the **dead man's switch**, its task is to make sure that if either the primary or secondary mission controller becomes unresponsive, it will not publish the velocity commands to the motion controller at all. In other words, it should avoid the situation when a constant command is being provided by any mission controller for an unexpectedly long period of time.

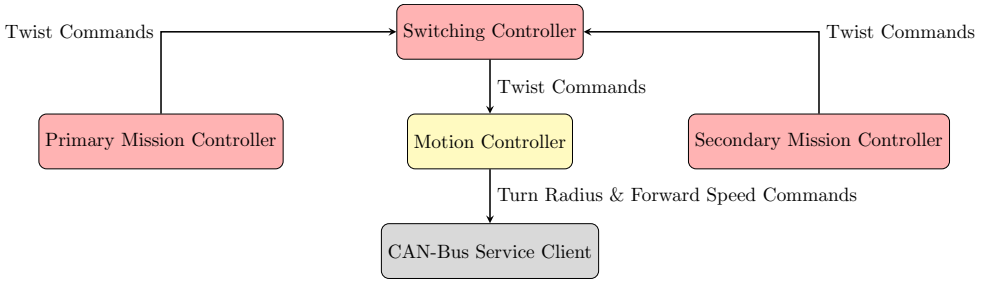

Figure 5.5: Motion Controller Function.

5.2.4 Motion Controller

As depicted in Figure 5.5, *motion_controller* is a dedicated ROS package and its salient features are listed below.

(a) Subscribes to the velocity commands from the *switching_controller* node.

(b) Translates the received velocity commands into **turn radius** and **speed** commands.

(c) Acts as a **ROS Service Client** to send the turn radius and speed commands to the PLCs via CAN Bus.

The scale factors are such that speed command is provided in millimeters per second in the range of [-32767, 32768]. It is important to notice here that the speed command may be issued with a negative sign to make use of the reverse direction of motion. However, this feature was not tested due to project timing constraints. But, after special modifications to the motion controller algorithm reverse driving features can be realized in the future. According to the datasheet of the Polaris Ranger [55], it can achieve the maximum velocity of about 11.1 meters per second.

On the other hand, the steering angle command is issued in terms of the turn radius in the range of [0, 65535]. The zero turn radius value means rotation to the extreme left, and 65535 turn radius value to the extreme right the front wheel may steer. Hence, the nominal command for maintaining the zero angle steering position is 32678, and it must be published continuously to the CAN-Bus node by the motion controller. The turn radius is generated by using the formula $v = R \times \omega$. Here, v is the linear x-velocity and ω is the angular z-velocity, in our case $\omega = \dot{\psi}$ is the yaw rate, and R is the turn radius. According to the datasheet of Polaris [55], the maximum turn radius it may acquire is 3.81 meters.

Having set all the basic autonomous functionalities on the e-ATV, we move on to discuss the self-driving tests performed with Polaris within the ROS framework.

6 Autonomous Drive

In this section, results for a self-driving test conducted with Polaris with the settings described in previous chapters are shown.

6.1 Description of Self-driving Test

The autonomous driving capability of Polaris was tested. However, the test was limited to following at most two initial target waypoints. This was due to the road conditions present during the test. Moreover, the objective was to evaluate the performance of the designed control scheme. It was a partial success as the mission controller was working fine along with the motion controller. The navigation module was also working properly. Nevertheless, the only issue left unproven was the switching controller.

Figure 6.6 highlights the initial settings in ROS as soon as the software starts the actual execution. The green line is the global path planned for Polaris seen by the primary mission controller. It is calculated by the first instance of *move_base* package responsible for moving the vehicle from WP 1 to WP 2. The cost map is also well-defined based on the laser scan of the environment. The panoramic image is also displayed in the bottom-left corner of the snapshot. The footprint of Polaris is shown as the blue rectangle. The two rectangles shown on the right as boundaries of the cost map and the one on the left can be easily recognized as cars standing by and considered naturally as obstacles.

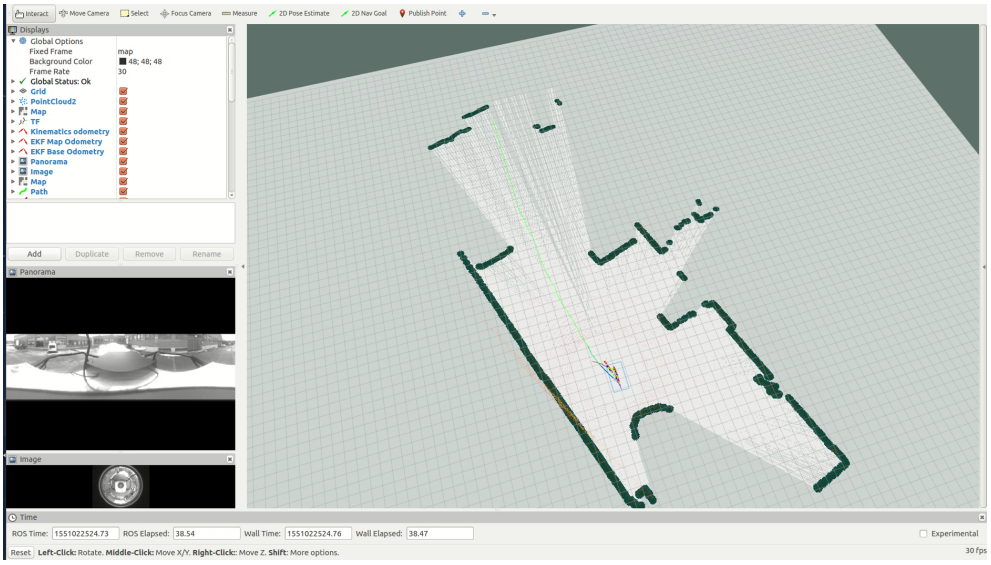

Figure 6.6: Self-driving Test (RViz Snapshot): Initial settings.

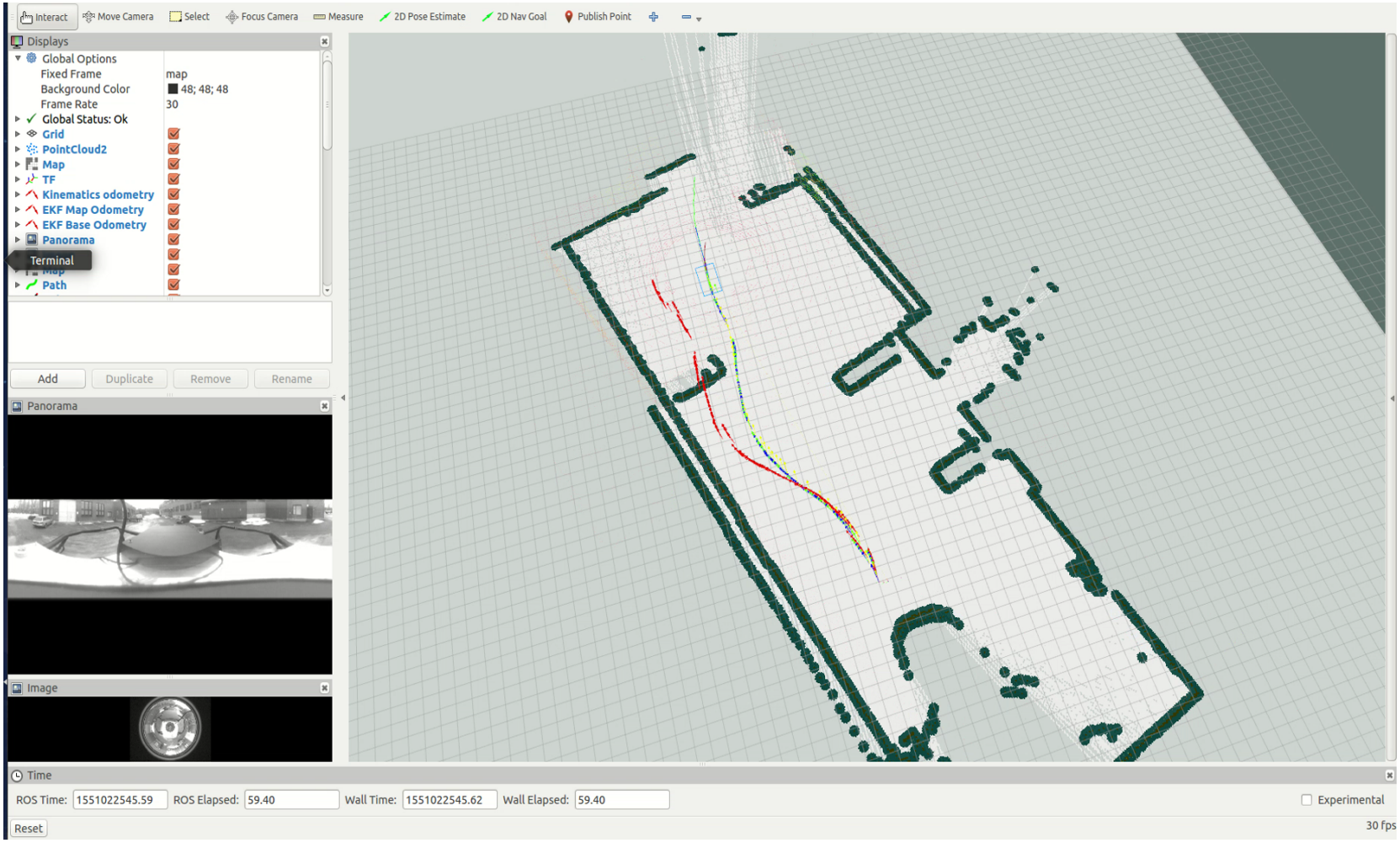

Figure 6.7: Self-driving Test (RViz Snapshot): Path (calculated by primary mission controller) followed by Polaris.

Figure 6.7 (shown on previous page) depicts the traversing of Polaris autonomously through the planned path dictated by global path planner. The thick green, blue and yellow (shown with arrowheads) trajectories (overlying on top of each other) are those calculated by odom → base_link odometry, map → base_link odometry and *navsat_transform_node* respectively. The red path is, however calculated by the robot's kinematic model, and is going way off from the actual path. Thin brown and blue (a bit longer than brown) trajectories are the path planned by the local planners.

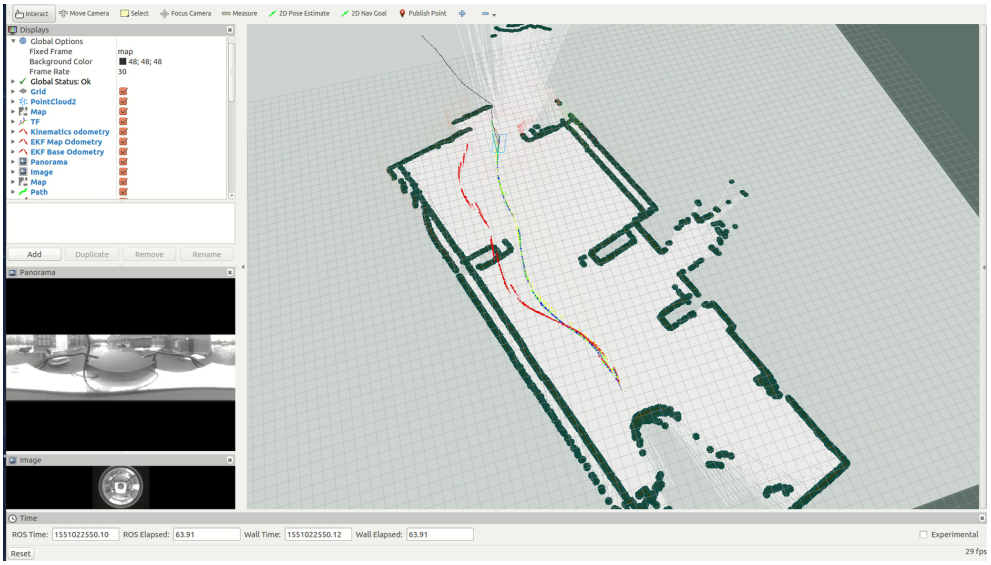

Figure 6.8: Self-driving Test (RViz Snapshot): Path calculated by secondary mission controller (thin black line).

Figure 6.8 depicts the start of a global path planned by the secondary mission controller shown by a thin black line around the top of the cost map. At this point, the command velocities generated by the secondary mission controller must have taken authority. The switching controller must have canceled all the goals set by the primary mission controller. Moreover, the second instance of *move_base* controller (instantiated by secondary mission controller) must have been in charge of vehicle motion from the second target waypoint to the next one.

However, this could not be achieved due to a programming bug in the switching controller, which was set to sense the zero velocity commands from the first instance of *move_base*. This resulted in the continuation of the utilization of commands generated by the primary mission controller even beyond the point when the first waypoint was hit successfully. The goal tolerance was set to 5 meters for both instances of the mission controller. The D2G was calculated continuously by using distance formula in transformed UTM coordinates.

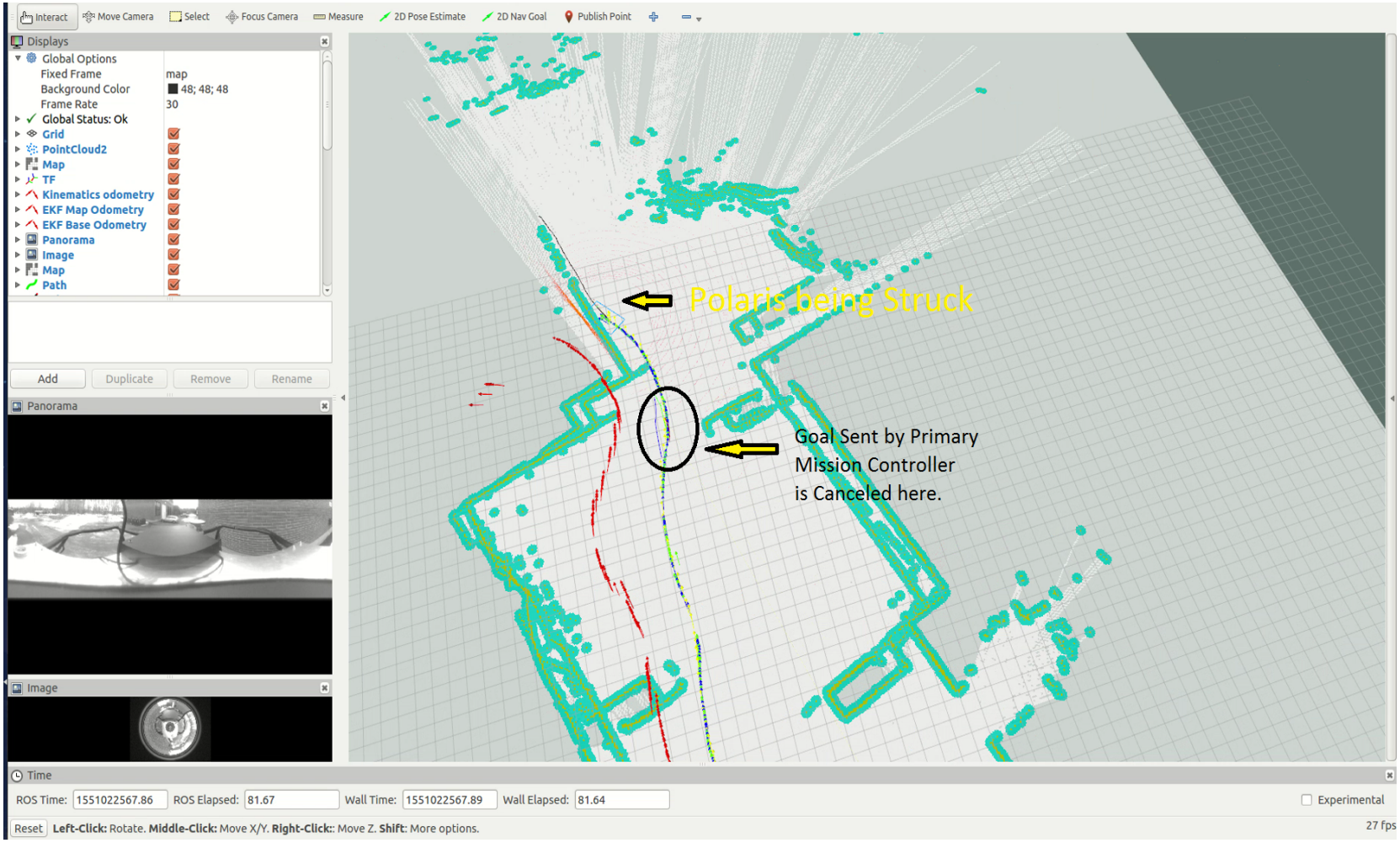

Figure 6.9: Self-driving Test (RViz Snapshot): Path followed by Polaris; *erroneous results*.

As soon as the D2G levels goes below 5 meters, a request to cancel the set goal to move base action server is sent via the switching controller. The reason for goal cancellation, as discussed earlier in Section 5.2, was the fact that the footprint of Polaris was considered to be too large. Also, the move base is an action server and keeps on tracking its goal unless the status of the server shows a success. The tolerance setting for the goal achievements can be parametrized. However, the cancellation of the goal was considered a more robust technique as it was never truly intended to achieve the actual goal position with the accuracy of a few centimeters. Figure 6.9 highlights the Polaris getting stuck into an obstacle (the wall of the building nearby). Furthermore, the thin green line (set by the global planner of the primary mission controller) is cut along with the thin blue line (of the long term local planner) as shown inside the black oval region highlighted in the same figure.

These findings were supported by the data analysis of the autonomous test drive, as explained in the next section.

6.2 Data Analysis

Figure 6.10 highlights the path followed by Polaris during the autonomous drive. Few mission waypoints are depicted as black stars, along with the starting point (red star) and the end point of the trajectory (green circle).

One can observe that the first target waypoint was hit with a desirable level of accuracy. The zig-zag trajectory before culmination to the final point was self-induced. At the time of the test, the reason for not following the path was not known and the Polaris was reversed manually in order to allow *move_base* package to re-plan the local paths to be traversed. But, each time the Polaris got stuck into the wall as if it would be following a fixed velocity command. This was later proven to be true when the command velocities generated by each instance of move base were compared with the output of the switching controller.

The routing of the velocity commands were depicted earlier in Section 5.2.3. Figure 6.10 illustrated the generation of commands by the move base instantiated by the primary mission controller. Figure 6.12 depicts the commands generated by *move_base* node instantiated by secondary mission controller and Figure 6.11 shows the commands routed to the motion controller by the switching controller.

One can observe that through-out the self-driving test only commands generated by the primary mission controller were routed to the motion controller. Furthermore, the secondary mission controller was trying to get the vehicle back on

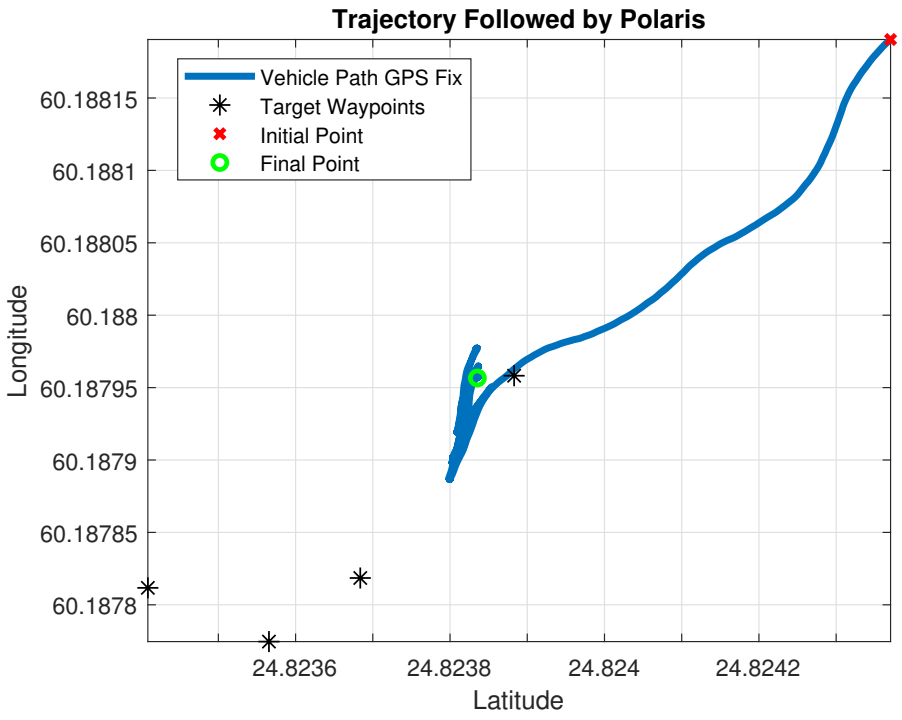

Figure 6.10: Autonomous test drive trajectory shown in GPS coordinates.

track by issuing correct yaw velocity commands while Polaris was stuck into the wall. However, the switching controller was not behaving correctly.

Figure 6.14 illustrates the output of mentioned localization routines during the autonomous drive. Here, the outputs from map → base_link odometry and that from navsat_transform_node are overlaid over each other. In this drive, the results of kinematic model looks more promising than the one from odom → base_link localization routine.

Figure 6.15 illustrates the speed profiles from the described localization routines. One can see that Polaris took around 40 seconds to traverse the trajectory autonomously till it got stuck into the wall. The manual drive switch installed on Polaris was switched on at this moment leaving the vehicle at stop from time 40 seconds till time 60 seconds. It was then reversed from time 60 seconds onwards shown by oscillations in the map → base_link and odom → base_link odometry. This was explained in Section 4.5, and the reason was that we are using GPS ground velocity as the forward speed and it is always positive even when Polaris is moving backwards.

At this point, Figure 6.13 becomes more intriguing as the command velocities are still changing even when the target point is achieved and the global as well as the local paths planned by the instance of *move_base* authorized by the primary

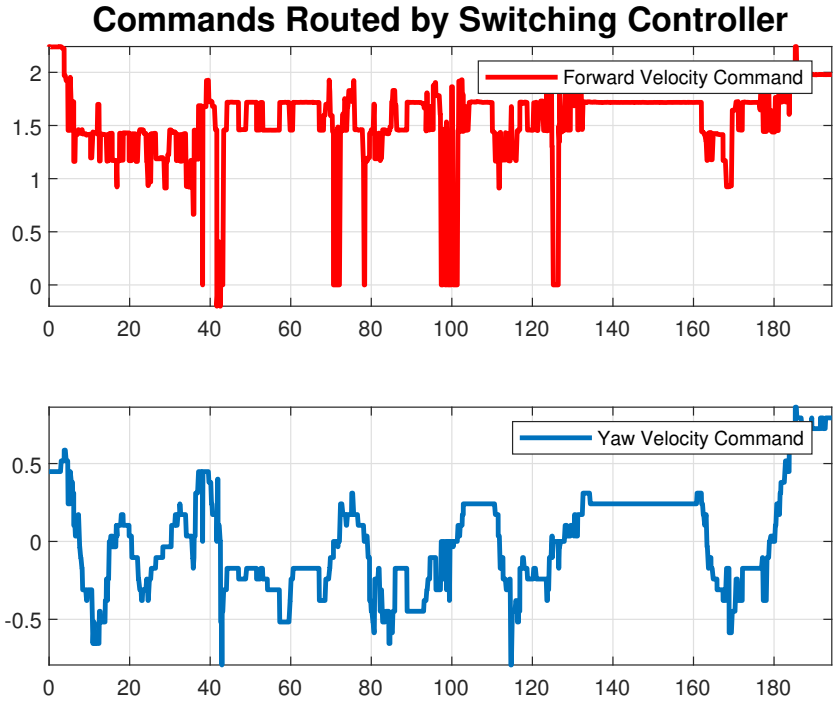

Figure 6.11: Command velocities routed by switching controller.

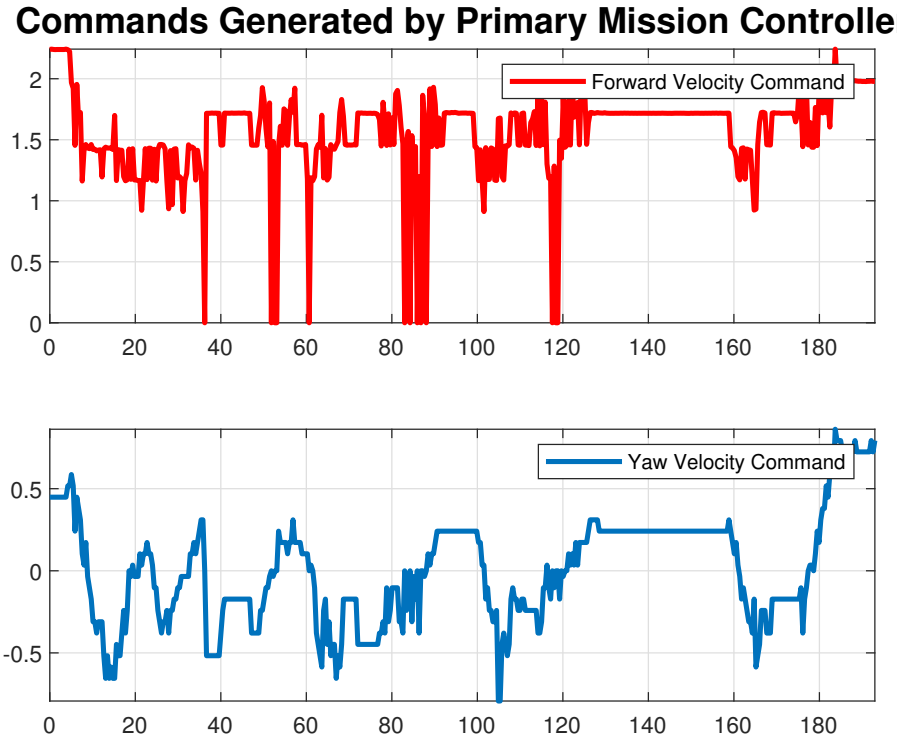

Figure 6.12: Command velocities generated by primary mission controller.

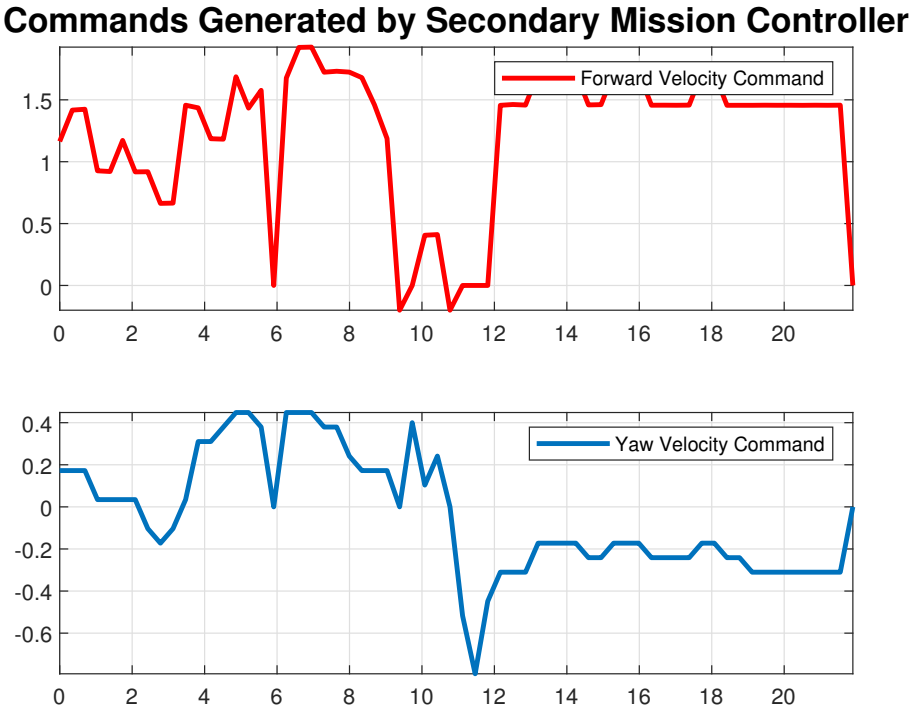

Figure 6.13: Command velocities generated by secondary mission controller.

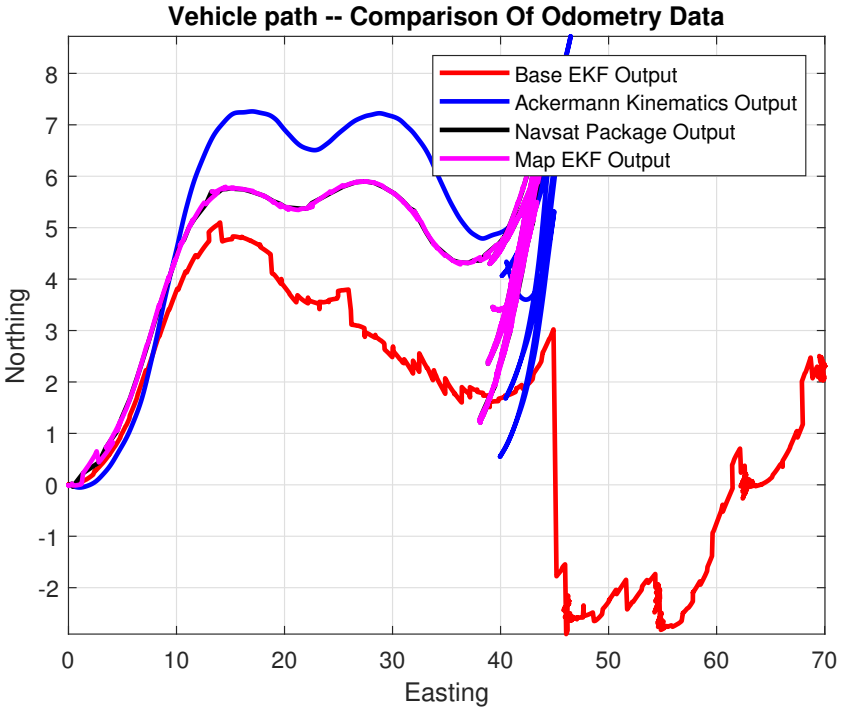

Figure 6.14: Comparison of odometries generated by various localization sources.

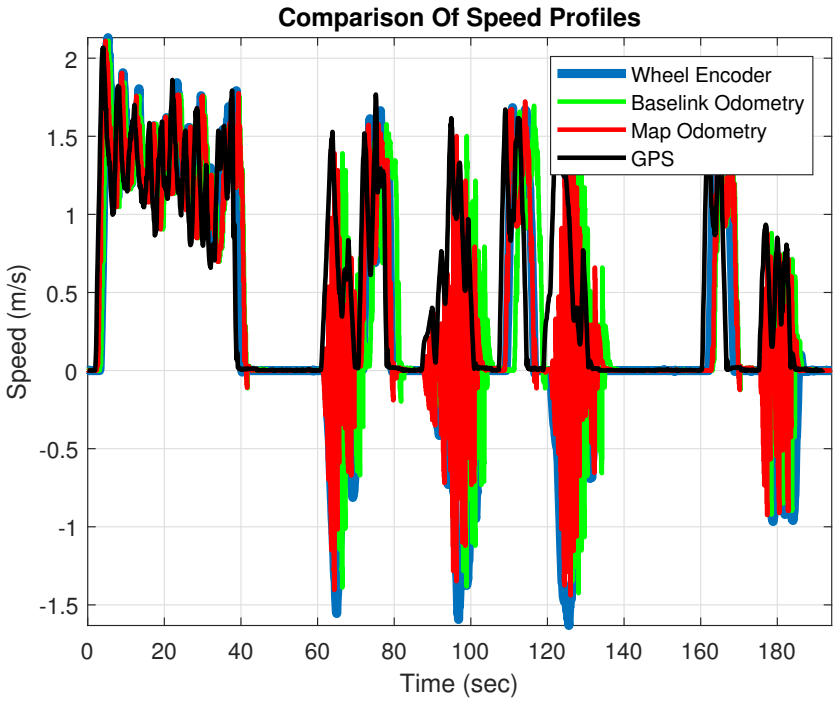

Figure 6.15: Comparison of speed profiles generated by various localization sources.

mission controller have been stopped. Moreover, the duration of the activity in both the primary mission controller and in the localization nodes is the same. However, the secondary mission controller has a short duration (around 20 seconds) as it was initiated a bit later when controller_done signal was issued by the primary mission controller.

7 Conclusions

The principal objective of the thesis work was to furnish the Polaris - an e-ATV - with the autonomous capabilities utilizing the crucial data from the available sensors. The vital data is in the form of the Polaris' pose information, a point cloud of its surroundings, the wheel speeds, and the steering angle as it moves in the environment. This data was supplied with minimum transport delays, and at the maximum update frequency to the embedded computer.

ROS being the decision-making or intelligence part of the system is installed on the computer. ROS is an open-source firmware that equips the programmer with a multitude of software features. The drivers for the Novatel's SPAN, Velodyne's HDL-32E, and Basler's Camera were already contributed to the research community by ROS developers. The Novatel's SPAN drivers were considerably improved to acquire angular rates and acceleration data in accordance with the ROS data conventions. Moreover, to improve the positioning accuracy of the SPAN, the differential corrections provided by the National Land Survey of Finland were also integrated into the system. Furthermore, various ROS topics encapsulating the required information about the vehicle's position, orientation, linear and angular velocities, accelerations, world frame, and child frame were produced. These augmentations served as an important precursor to the Polaris' localization and mapping in the world.

Assuming a 3DOF vehicle model, a dead reckoning method could have addressed the robot's localization problem. Such a model estimates the position - in x- and y-coordinates - and heading angle of the vehicle using the wheel speed and steering angle. However, this model was inaccurate in keeping track of the vehicle's states - the x- and y- positions and heading angle in the *odom (odometry) frame* - with time due to the slipping of the wheels and steering actuation delays. This explains the inclusion of the rates and accelerations data from the IMU; plus position and velocity data from the GPS unit. The IMU gives its data in the body's frame, while the GPS receiver provides the location of the rover in a satellite-based coordinate system.

The rover's position provided in the Latitude Longitude Altitude (LLA) format has to be converted into a *world frame*. In ROS, Universal Traverse Mercator (UTM) coordinate system is adapted to show the robot's position in the world frame. Subsequently, the pose of the robot is then transformed into the *map frame* by multiplying the robot's current pose with a rotation matrix. This rotation matrix contains the info about the robot's initial pose. Hence, the map frame is a coordinate frame whose origin is attached to the starting position of the robot in the world frame. Similarly, a *base link frame* that defines the robot's origin in the body frame, is hypothetically attached to the midpoint of the rear

axle.

As the sensor measurements are nonlinear and prone to errors, two separate instances of the Extended Kalman Filter (EKF) were employed for filtering and nonlinear state estimation through sensor fusion. Each EKF keeps track of the 15-dimensional state of the vehicle moving in 3D space. The state vector contains the vehicle's positions, linear and angular velocities, Euler angles, and body accelerations in 3D coordinates. The top-level EKF keeps track of the robot's motion in map frame with respect to the base link. While the low-level EKF estimates the state vector in odom frame moving with respect to the base link. The top-level EKF is provided with the position corrections from the GPS receiver. The low-level EKF implements what the regular dead reckoning model does, but using the rates and accelerations data from IMU.

The data analysis carried out in Section 4.5 highlights the significance of the EKF-based vehicle's states estimation. Position, orientation and velocity profiles from both EKFs - fusing data from IMU, GPS, and wheel encoders - were compared in that section. The performance of both EKFs was above par as compared to the simple dead reckoning method. The GPS Ground Speed was used as the forward velocity parameter in both EKFs. This had an advantage in terms of measurement accuracy and data update rate. However, during reverse motion of the Polaris, this caused oscillations in the position and speed profiles due to its nature of being always positive. Nevertheless, the wheel speed is a good alternative to the GPS Ground Speed in situations where the GPS is completely lost. Hence, the wheel speed was also incorporated into the sensor fusion step but with a higher - hardcoded - covariance value. Plenty of efforts were put to test this part of the thesis during harsh Nordic weather conditions.

With the localization problem of the Polaris well-addressed, the mapping of the environment was carried out by utilizing a highly efficient Rao-Blackwellized particle filter. The *GMapping* package was responsible for producing a 2D occupancy grid of the environment. It utilizes the laser scan data from the LiDAR and map to base link transformation provided by the top-level EKF. This occupancy grid is then generalized by the navigation module into global and local cost maps. The cost map parameters were set in accordance with the dimensions of the Polaris and the maximum possible distance between two consecutive waypoints.

Target waypoints were selected from the position profile of a manual test drive. Together these waypoints (WPs) act as a mission profile. Each waypoint was transformed into the map frame and supplied to the global motion planner as goals. Over the global cost map, the global planner applies a simplified A* search algorithm, such as Dijkstra's, to find the optimal path. This optimal path minimizes the cost of traversing the vehicle to the target position. This opti-

mal path is provided to the local planner which is a Monte Carlo planner with cost evaluation. The local planner simulates trajectories of the vehicle using the instantaneous linear and angular velocities of the vehicle. The velocities that generate the most effective trajectory are sent as twist commands to the base-level actuator controller.

A package named *mission_controller* was created whose main responsibilities were: to become a client of an instance of the *move_base* package, read the mission profile, transform the waypoints into the map frame, send these transformed waypoints as goals to the global motion planner node within the *move_base* package. Initially, the clearing rotations were set to true with a goal tolerance of 1 meter. However, the Polaris's inability to do the reverse motion in autonomous mode hampered its ability to perform clearing rotation at the target waypoint. Moreover, Polaris has a larger footprint that renders clearing rotations useless. In addition, the target waypoints are themselves prone to position inaccuracies (up to $\pm$ 7 meters in the absence of differential corrections). Overall, this led to the adaption of a goal cancellation policy. Through such a policy, the goal was canceled as soon as the Distance-to-Go (D2G) to the waypoint drops below 5 meters. On the downside, this strategy is based on preemption of move base tasks; and thus, added significant delays in traversing the vehicle from the current waypoint to the next.

To address this problem, two mission controllers were implemented with identical codes and different set of waypoints. The primary mission controller will be responsible for moving the Polaris from WP1 to WP2. And the secondary mission controller will autonomously navigate the vehicle from WP2 to WP3, and so forth. Through such topology, a smooth self-driving was expected through all the waypoints. To decide when to switch the control authority, a *switching_controller* was implemented as a node within the mission controller package. The switching controller was also equipped with a dead man's switch, which checks the validity of the velocity commands from each instance of the move base node. It also checks if the update rate of these commands falls below a certain frequency.

With the configurations discussed above, the autonomous driving capability of Polaris was tested. Unfortunately, the switching controller failed to switch the control authority to the secondary as soon as the D2G calculated by the primary dropped below 5 meters. This was verified by the data analysis of the autonomous test drive, as carried out in Section 6.2. The data analysis also revealed that the secondary was producing adequate twist commands. But, these commands were not routed to the motion controller by the switching controller. Due to the project's timing constraints, this issue was left unattended.

Another package called *motion_controller* was created to receive and convert

the twist commands into turning radius and speed commands. These commands were then routed to the PLCs to control the wheel actuators via CAN-bus. Again, to meet the project's timing requirements, the reverse motion capability was not tested on the Polaris. The PLC program was operated at a lower rate (10 Hz) and hence, proved futile in producing smooth steering and forward motions.

With the above-mentioned discussion, we finish extracting the conclusions from the entire thesis work. A few future work recommendations drawn from the conclusion are presented in the next section.

7.1 Future Work Recommendations

As far as the hardware is concerned, the fixture of SPAN to the Polaris chassis needs to be standardized. It should be tested whether the current fixture is inducing oscillations into the accelerometers. Besides, a mobile carrier reader needs to be installed in order to acquire differential corrections in real-time.

Differential GPS will definitely give an edge to the precise localization of Polaris. Positioning will be robust and less prone to position and velocity inaccuracies. A mobile network connection will give access to the wireless internet when the Polaris is being driven outside. Afterward, receiving GPS corrections via an internet connection from NLS and then routing it to SPAN would be straightforward. Moreover, SPAN needed alignment each time it was powered up. This adds a considerable amount of time to the initialization procedures. This issue definitely needs to be sorted out in the future. In addition, special attention needs to be paid to increase the operation frequency of the PLC programs.

The double controller scheme needs to be verified systematically. For this reason, the switching controller code needs to be thoroughly debugged. The dead man's switch part also needs to be improved. This will allow the detection of the constant (incorrect) commands from the active controller. The issue with the goal update in ROS Action Server also needs more debugging.

For now, the localization is based on the LiDAR and IMU data. In the future, vision-based odometry could be added to the algorithm set. In case the GPS signals are lost, it would allow trespassing through the unidentified terrain. Moreover, some type of vision-based obstacle avoidance would be valuable. For path planning, D* and D* Lite algorithms need to be implemented and tested.

Although the main idea behind the project is to build a platform for researchers and students, the lack of an accurate system model of Polaris hampers the research on relevant autonomous capabilities. This highlights the significance of system modeling. For advanced research, Computational Fluid Dynamics (CFD)

and Finite Element Methods (FEM) analysis could provide a new dimension to the project as far as system modeling is concerned.

That would assist in utilizing advanced control algorithms for self-driving vehicles. Advanced controller architectures such as a Model-based Controller or Sliding-Mode Controller, etc. would be possible to realize. System identification techniques could be applied to the vehicle as well. That would assist in quantifying the relationship between inputs (such as command velocities) and the outputs (such as rate of steering or forward speed). Such heuristic models could become the foundation of a more sophisticated control system architecture.

A simulation platform for the autonomous ground vehicle in Gazebo has already being realized. It will allow the testing and evaluation of the advanced navigation and control algorithms in a simulated environment, which should speed up this Polaris Ranger (e-ATV) based project. However, extensive testing is required to verify if the simulation results match up with the actual test runs on the vehicle.

Command and Control Station (CCS) for real-time parameter tracking would help us to achieve our long-term objective, which is to be, hopefully not in so distant future, fully capable of performing completely autonomous operations with our Polaris Ranger in dynamic environments. Real-time telemetry would naturally allow the user to monitor the state of the vehicle online from a greater distance, and remotely control of the Polaris in case of some serious problems. Because there is always a possibility for Polaris to get stuck in rough weather conditions, CCS might well be used as an advanced security awareness system. In case the vehicle is about to get beaten by the elements, it could adjust the whole mission profile in real-time and perform some safety measures if available.

As a final remark, the future work recommendations are based on the literature survey carried out in the state-of-the-art section (Chapter 2). Nevertheless, the future research path is naturally highly dependent on the funds being provided for this project.

Let's hope for the best.

References

[1] Paul Wolfram and Nic Lutsey, *Electric vehicles: Literature review of technology costs and carbon emissions*, International Council on Clean Transportation, 15 July 2016.

[2] Nicola Geromel, *ATV Braking System: Introduction and System Modeling*, Masters Thesis Report Submitted to Arto Visala, 2014. Available https://aaltodoc.aalto.fi/handle/123456789/13923.

[3] Andrei Sandru, Eva Koppali, Lassi Kaariainen and Martin Granholm, *Teaching and Research Platform for Autonomous Vehicles*, PT33: Final Project Report submitted to Arto Visala and Heikki Hyyti, Project Work Course, 2016, AEE Department, Aalto University.

[4] Ville Kukkonen, Amin Modabberian, Pietu Roisko, Pyry Viita-aho and Ilmari Vikström, *Perception Platform for Autonomous Vehicles*, Project Number 16: Final Project Report submitted to Arto Visala and Andrei Sandru, Project Work Course, 2017, AEE Department, Aalto University.

[5] Lukas Wachter, Onur Sari, Panu Pellonpaa and Karhu Valtteri , *Building Simulation Platform for Polaris e-ATV in ROS using Gazebo*, Project Number 12: Final Project Report submitted to Mika Vainio and Tabish Badar, Project Work Course, 2019, AEE Department, Aalto University.

[6] National Land Survey of Finland, *Maps and Spatial data: DGNSS Service*. Refer to website: https://www.maanmittauslaitos.fi/en/maps-and-spatial-data/positioning-services/dgnss-service.

[7] Rudolph E. Kalman, *A New Approach to Linear Filtering and Prediction Problems*, Transactions of the ASME pp. 35-45, USA, 1960.

[8] Sebastian Thrun, Wolfram Burgard and Dieter Fox *Probabilistic Robotics, First Edition*, Intelligent Robotics and Autonomous Agents Series, 2001.

[9] Rudolph E. Kalman and R. S. Bucy, *New Results in Linear Filtering and Prediction Theory*, Transactions of the ASME pp. 95-108, USA, 1961.

[10] Simo Sarka and Arno Solin, *Applied Stochastic Differential Equations*, Institute of Mathematical Statistics Textbooks, Cambridge University Press, Feb 2019.

[11] D. Crisan and B. L. Rozovshii, *The Oxford Handbook of Nonlinear Filtering*, Oxford University Press, Oxford and New-York, USA, 2011.

[12] Simo Sarka and Arno Solin, *On Unscented Kalman Filtering for State Estimation of Continous-time Nonlinear Systems*, IEEE Transactions on Automatic Control, 52(9), pp 1631-1641, 2007.

[13] Simo Sarka, *Bayesian Filtering and Smoothing*, Institute of Mathematical Statistics Textbooks, Vol. 3. Cambridge University Press, Cambridge, 2013.

[14] Yaakov Bar-Shalom, X. Rong Li and Thisgalingam Kirubarajan, *Estimation with Applications to Tracking and Navigation: Theory, Algorithms and Software*, John Wiley and Sons, Inc., 2001.

[15] M. S. Arulampalam, S. Maskell, N. Gordon and T. Clapp, *A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking*, IEEE Transactions on Signal Processing. 50(2):174-188.

[16] Shmuel Merhav, *Aerospace Sensor Systems and Applications*, Springer-Verlag New York, Inc., First Edition, 1996.

[17] Roland Siegwart and Illah R. Nourbaksh, *Introduction to Autonomous Mobile Robots*, A Bradford Book, The MIT Press, Cambridge, MA, USA 2004.

[18] Howie Choset, Kevin M. Lynch, Seth Hutchinson, George A. Kantor, Wolfram Burgard, Lydia E. Kavraki and Sebastian Thrun, *Principles of Robot Motion: Theory, Algorithms, and Implementation*, The MIT Press, 2007.

[19] Peter Corke, *Robotics, Vision and Control*, Springer Tracts in Advanced Robotics, Volume 73, 2011.

[20] Robin R. Murphy, *Introduction to AI Robotics*, MIT Press, Cambridge, Massachusetts, USA, 2000.

[21] Alonzo Kelly, *Mobile Robotics: Mathematics, Models and Methods*, Cambridge University Press, USA, 2013.

[22] Phillip A. Laplante and Seppo J. Ovaska, *Real-Time Systems Design and Analysis: Tools for the Practitioner, Fourth Edition*, the Institute of Electrical and Electronics Engineers, Inc. Published 2012 by John Wiley & Sons, Inc.

[23] Davide Brugali, Luca Gherardi, A. Biziak, Andrea Luzzana and Alexey Zakharov, *A Reuse-Oriented Development Process for Component-Based Robotic Systems*, SIMPAR 2012, LNAI 7628, pp. 361–374, 2012.

[24] Gazebo, *Gazebo: Simulations Made Easy*. Please see: gazebosim.org.

[25] United Nations Office For Outer Space Affairs, *Current and Planned Global and Regional Navigation Satellite Systems and Satellite-based Augmentations Systems*, International Committee on Global avigation Satellite Systems Provider's Forum, 2016.

[26] Thomas Moore and Daniel Stouch, *A Generalized Extended Kalman Filter Implementation for the Robot Operating System*, Intelligent Autonomous Systems 13, 2016.

[27] Brian Paden, Michal Cáp, Sze Zheng Yong, Dmitry Yershov and Emilio Frazzoli, *A Survey of Motion Planning Techniques for Self-driving Urban Vehicles*, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge MA, USA, 2016. Available https://ieeexplore.ieee.org/document/7490340.

[28] B. Chazelle, *Approximation and decomposition of shapes*, Advances in Robotics, Vol. 1, pp. 145-185, 1987.

[29] B. P. Gerkey, *Planning and Control in Unstructured Terrain*, Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008).

[30] Dieter Fox, Wolfram Burgard and Sebastian Thrun, *Dynamic Window Approach to Collision Avoidance*, IEEE Robotics & Automation Magazine (Volume: 4 , Issue: 1 , Mar 1997).

[31] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard, *Improved Techniques for Grid Mapping with Rao-Blackwellized Particle Filters*, IEEE Transactions on Robotics, Volume 23, pages 34-46, 2007.

[32] O. Takahashi and R. J. Schilling, *Motion planning in a plane using generalized voronoi diagrams*, Transactions on Robotics and Automation, Vol. 5, pp. 143-150, 1989.

[33] Phillip J. McKerrow and D. Ratner, *Caliberating a 4-wheeled Mobile Robot*, Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems, October 2002.

[34] Bong-Su Cho, Woo-sung Moon, Woo-Jin Seo and Kwang-Ryul Baek, *A dead reckoning localization system for mobile robots using inertial sensors and wheel revolution encoding*, Journal of Mechanical Science and Technology, vol. 25, no. 11, pp. 2907–2917, 2011. DOI 10.1007/s12206-011-0805-1. Available: https://link.springer.com/article/10.1007/s12206-011-0805-1.

[35] J. C. Latombe, *Robot Motion Planning*, Springer Science and Business Media, Vol. 124, 2012.

[36] Edited by Shuzi Sam Ge and Frank L. Lewis *Autonomous Mobile Robots: Sensing, Control, Decision Making and Applications*, Taylor and Francis Group, CRC Press, 2006.

[37] David M. Bevly and Stewart Cobb, *Global Positioning Systems, Inertial Navigation, and Integration*, Norwood, MA, 2010.

[38] Anil Mahtani, Luis Sanchez, Enrique Fernandez and Aaron Martinez, *Effective Robotics Programming with ROS, Third Edition*, Packt Publishing Ltd., UK, 2016.

[39] Carol Fairchild and Dr. Thomas L. Harman, *ROS Robotics by Examples, Second Edition*, Packt Publishing Ltd., UK, 2017.

[40] Mohinder S. Grewal, Lawrence R. Weill and Angus P. Andrews, *GNSS for Vehicle Control*, John Wiley and Sons Inc., Second Edition, 2007.

[41] M.H.A. Hamid, A.H. Adom, N.A. Rahim and M.H.F. Rahiman , *Navigation of mobile robot using Global Positioning System (GPS) and obstacle avoidance system with commanded loop daisy chaining application method*, 5th International Colloquium on Signal Processing & Its Applications, 2009.

[42] Dominic Jud, Martin Wermelinger, Marko Bjelonic, Péter Fankhauser and Marco Hutter, *Programming for Robotics - ROS*, HG G1, Lecture Notes in Robotics System Lab, 2019. Available http://www.rsl.ethz.ch/education-students/lectures/ros.html

[43] F. Dellaert, D. Fox, W. Burgard and S. Thrun, *Monte Carlo localization for mobile robots*, IEEE International Conference on Robotics and Automation, vol.2, 1999. Available http://www.ing.unibs.it/~cassinis/ Dida/2005/robb/lezioni/Articoli,%20ecc./dellaert_frank_1999_2.pdf

[44] Tully Foote and Mike Purvis, *Standard Units of Measure and Coordinate Conventions*, ROS REP-103 standard, created October, 2010. Available http://www.ros.org/reps/rep-0103.html.

[45] Wim Meeussen, *Coordinate Frames for Mobile Platforms*, ROS REP-105 standard, created October 2010. Available http://www.ros.org/reps/rep-0105.html.

[46] Charles F. F. Karney, *Algorithms for Geodesics*, SRI International, 201 Washington Rd, Princeton, NJ 08543-5300, USA, Dated September 2011.

[47] Tully Foote, Eitan Marder-Eppstein and Wim Meeussen, *ROS Documentation on **tf2** Package*. Available http://wiki.ros.org/tf2.

[48] Tom Moore, *ROS Documentation on **robot_localization** Package*. Available http://wiki.ros.org/robot_localization.

[49] Vincent Rabaud, *ROS Documentation on **gmapping** Package*. Available http://wiki.ros.org/gmapping.

[50] Eitan Marder-Eppstein, David V. Lu and Dave Hershberger, *ROS Documentation on **costmap_2d** Package*. Available http://wiki.ros.org/costmap_2d.

[51] Michael Ferguson, David V. Lu and Aaron Hoy, *ROS Documentation on **move_base** Package*. Available http://wiki.ros.org/move_base.

[52] Eitan Marder-Eppstein, *ROS Documentation on **dwa_local_planner** Package*. Available http://wiki.ros.org/dwa_local_planner.

[53] Kurt Konolige and Eitan Marder-Eppstein, *ROS Documentation on **dwa_local_planner** Package*. Available http://wiki.ros.org/navfn?distro=melodic.

[54] ClearPath Robotics Inc., *Husky Outdoor GPS Waypoint Navigation*. See https://www.clearpathrobotics.com/assets/guides/husky/HuskyGPSWaypointNav.html.

[55] Polaris Inc., *Device Description of Polaris Ranger EV*. Refer to: https://ranger.polaris.com/en-us/ranger-ev/.

[56] Velodyne LiDAR Inc., *Device Manual For Velodyne HDL 36E*. Available https://velodynelidar.com/hdl-32e.html.

[57] Baseler Inc., *Device Manual For Basler Ace acA1600-20gm*. Available: https://www.baslerweb.com/en/products/cameras/area-scan-cameras/ace/aca1600-20gm/.

[58] NovAtel Inc., *Device Manual For SPAN-IGM-S1*. Available: http://www.novatel.com/products/span-gnss-inertial-systems/span-combined-systems/span-igm-s1/.

[59] EPEC Oy., *Device Manual For EPEC-5050 Control Unit*. Available: https://epec.fi/products/control-system-products-5050/.