Bachelor's thesis

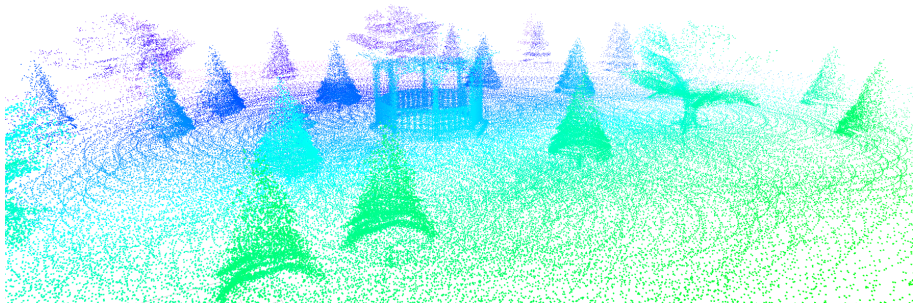
NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Eline Marie Håve, Sigrid Mellemseter and Cecilie Nikolaisen

ROS Simulated World for ATV with SLAM

ROS Simulert Verden for ATV med SLAM

Bachelor's thesis in Electrical Engineering - Automation and Robotics
Supervisor: Christian Fredrik Sætre
May 2022



NTNU
Kunnskap for en bedre verden

KONGSBERG

Eline Marie Håve, Sigrid Mellemseter and Cecilie Nikolaisen

ROS Simulated World for ATV with SLAM

ROS Simulert Verden for ATV med SLAM

Bachelor's thesis in Electrical Engineering - Automation and Robotics
Supervisor: Christian Fredrik Sætre
May 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of Science and Technology

KONGSBERG

Project title: ROS Simulated World For ATV With SLAM	
Authors: Eline Marie Håve Sigrid Mellemseter Cecilie Nikolaisen	**Project number:** E2202 **Due date:** 20.05.2022 **Grade:** [x] open [ ] closed
Study programme: Bachelor in Electrical Engineering	
Field of study: Automation and Robotics	
Supervisor internally: Christian Fredrik Sætre **Department:** Department of Engineering Cybernetics	
Client: Kongsberg Defence & Aerospace **Contact:** Roger Werner Laug, Henrik Bergel	

Abstract:
This thesis documents the bachelor assignment concerning a ROS simulated world for an autonomous ATV. The ATV is modelled in CAD software, and implemented in a Gazebo world. The ATV gathers data for a point cloud, which is used to generate a map and path with SLAM algorithms.

Sammendrag:
Denne rapporten dokumenterer bacheloroppgaven om en ROS-simulert verden for en autonom ATV. ATVen er modellert i CAD programvare, og implementert i en Gazebo verden. ATVen samler data til en punktsky som brukes til å generere et kart og sti ved hjelp av SLAM algoritmer

Keywords: Autonomous vehicles, localisation, mapping, modelling, point cloud, ROS, simulation	**Nøkkelord:** Autonome fartøy, lokalisering, kartlegging, modellering, punktsky, ROS, simulering

Abstract

This thesis documents the development of a simulated test platform for the Lone Wolf project. The thesis covers modelling, framework, implementation, simulation and visualisation. The thesis aims to enable further development of autonomous navigation in the Lone Wolf project.

The Lone Wolf ATV was modelled in CAD software and exported for use in ROS. The model of the ATV consists of a base link with joints forking out to the four wheels. It is simplified from the actual ATV to make joint definition less complicated and to enable simulation of movements, wheel spin and turning in the ROS environment. Two worlds are generated for the ATV to gather data from and map. One world is more complex than the other.

A Velodyne VLP-16 LiDAR is implemented to the model file in the ROS environment. The sensor is used to gather data and generate a point cloud. It is also implemented an IMU sensor in the model file, which is used to supplement the odometry of the LiDAR. Simultaneous navigation and mapping is implemented as a ROS package in order to generate a map. The algorithms uses the gathered point cloud from the LiDAR, to estimate the path and map, which is published and visualised.

The simulator works as intended and can be used as a test platform for SLAM algorithms. The ATV model gathers data in form of a point cloud, and this data along with data from the IMU is processed by a SLAM algorithm. Due to the intention of the project being testing and further development, most of the discussions in the thesis focuses on possible implementations and functionalities of the simulator.

Sammendrag

Denne rapporten dokumenterer utviklingen av en simulert testplattform for Lone Wolf-prosjektet. Rapporten dekker modellering, rammeverk, implementering, simulering og visualisering. Prosjektet har som mål å muliggjøre videreutvikling i Lone Wolf-prosjektet.

Firhjulingen Lone Wolf ble modellert i en CAD-programvare og eksportert for bruk i ROS. Modellen av firhjulingen består av en basislenke med fire ledd som går ut til hjulene. Modellen er forenklet fra den faktiske firhjulingen for å gjøre leddkonfigurasjonene mindre komplisert og for å muliggjøre simulering av bevegelser, hjulspinn og svinging i ROS-miljøet. To verdener er generert for forhjulingen å samle data fra og kartlegge. Den ene verdenen er mer kompleks enn den andre.

En Velodyne VLP-16 LiDAR ble implementert til modellfilen i ROS-miljøet. Sensoren brukes til å samle inn data og generere en punktsky. Det er også implementert en IMU-sensor i modellfilen, som brukes til å supplere odometrien til LiDARen. Samtidig navigasjon og kartlegging ble implementert som en ROS pakke for å generere kart. Algoritmen bruker punktskyen fra LiDARen til å estimere en sti og et kart. Denne dataen publiserer og kan dermed visualiseres.

Simulatoren fungerer etter hensikten og kan brukes som en testplattform for SLAM-algoritmer. Modellen av firhjulingen samler inn data i form av en punktksy og IMU data som behandles av en SLAM-algoritme. På grunn av at prosjektet er utviklet for testing og videreutvikling, fokuserer de fleste diskusjonene i rapporten på mulige implementeringer og funksjonaliteter til simulatoren.

Aknowledgments

With the completion of this project, the authors of this thesis completes the bachelor program in electrical engineering with specialisation in automation and robotics. We would like to thank our lecturers during this programme for providing us with the knowledge needed to complete this project and officially call ourselves engineers.

We would like to thank Ascend NTNU and its members for help and assistance during this project, and for providing us with knowledge within robotics and project management. We are particularly grateful for Dorde Veljkovic's help with solving our IT-problems. Furthermore we would express gratitude to Sander Furre for holding an introductory presentation to SLAM. We would also like to extend our appreciation to Henrik Bergel and Roger Werner Laug for their valuable insight and knowledge of the project.

Finally, we would like to thank our supervisor Christian Fredrik Sætre for his guidance and sound advice throughout the project.

Trondheim, May 20, 2022

Eline Marie Håve

Sigrid Mellemseter

Cecilie Nikolaisen

Contents

Figures

Tables

Listings

Chapter 1

Introduction

1.1 Background

Lone Wolf is a multi-disciplinary student project in Kongsberg Defence & Aerospace. The aim of the project is to design and build an autonomous all-terrain vehicle with implemented obstacle avoidance. The project started up in 2019 and has changed course over the years, from building an autonomous remote weapon station to designing and building an autonomous ATV. The Lone Wolf ATV is shown in figure 1.1.

Figure 1.1: Physical Lone Wolf ATV

The ATV is constructed to turn, gear, accelerate and break by computer control. Object detection has been implemented to assist if unforeseen obstacles are in the planned path. The saddle has been removed to implement framework for all other modifications such as remote steering and sensor placements. Figure 1.1 shows the modifications done to the ATV.

There are several challenges associated with making the ATV fully autonomous. Being able to test different methods of autonomous navigation and scenarios in fictional environments will help with the rate of development of the project. If multiple environments are created, it is possible to test in both simple and more complex environments in order to find an optimal method of autonomous navigation.

1.2 Requirements

One or more fictional environments should be available for testing autonomous navigation of the ATV both in simple and complex environments. To do this, a model of the ATV has to be made. The ATV is equipped with several sensors such as LiDAR, radar, IMU, Cameras and a GPS that can be included and used in the simulation for best possible autonomous navigation.

The ATV model should gather data in the form of a point cloud in the ROS simulator with the help of a LiDAR or stereo camera. This point cloud will be processed by SLAM algorithms and supplemented with different sensor data i.e. IMU-data and odometry. The map of the environment the SLAM-algorithm estimates should have adequate quality so the planning of the passable path for autonomous navigation can be seen through. The SLAM algorithm should be fitting in terms of complexity and result to continue working on in the Lone Wolf project 2022. The map should be three dimensional supplemented with other sensor data.

For the original project assignment document see appendix A and for a complete inventory list see appendix D.

1.3 Thesis Statement

Testing autonomous navigation is both challenging and time consuming, and there are a lot of unforeseen obstacles and challenges. In previous years, the summer students have not had an efficient way of testing autonomous mapping and navigation, with all testing having taken place in the terrain. A simulator that can gather data in form of a point cloud and generate maps by using SLAM algorithms enables a simpler way of testing autonomous navigation. With such a simulator, the students will be able to test autonomous mapping in both simple and more complex environments.

1.4 Problem Statements

Several problems have to be overcome due to the complexity of the project. This list summarises the most pertinent to the project.

- A similar model of the Lone Wolf ATV is needed for the simulator to be accurate.
- Simulated environments are needed to test SLAM algorithms.
- The model needs simulated sensors to gather data to visualise a point cloud.
- Point cloud data needs to be processed by a SLAM algorithm to generate maps.
- A test platform for autonomous navigation needs to be generated.
- Documentation of the project is crucial for further development and use of the simulator as a test platform.

1.5 Objectives

Some objectives to solve for the problems stated in the previous section were defined. These lay the foundation for the work and choices made for this thesis.

- By getting as accurate measurements of the ATV as possible, a representative model could be implemented in the simulator.

- Generating worlds in Gazebo makes environments for testing algorithms available.
- Implementation of the model and sensors in ROS enables gathering of data that can be used to visualise a point cloud.
- Building a SLAM package that is fed LiDAR data enables generation of maps based on point cloud data.
- Generation of maps through SLAM algorithms enables the possibility of autonomous navigation in later developments.
- Use of Git allows for version control, and storage of all software. This thesis works as a documentation of the development. All 3D modelling parts will be uploaded to the Git repository.

1.6 Report Structure

The report is divided into chapters based on the topics they discuss. This structure is implemented to make it more simple to read and reference for further development of the simulator. Each chapter contains its own theory, method, results, discussion and conclusion. Due to the nature of the project, some of the chapters will reference previous sections.

The first chapter includes the background of the project, requirements and the thesis statement. The issue of the project will be presented as well as requirements and a suggestion for solution. The chapter also contains a list of definition of terms and abbreviations that appear frequently in the thesis.

The second chapter has an overview of the research and development method used in this project, along with a HSE and risk assessment.

The third chapter introduces ROS, the robot development framework used for the simulator. This chapter includes a complete overview of the theoretical workings of ROS and basic features, as well as packages and essential applications such as Gazebo and RViz.

The fourth chapter contains all the aspects of modelling the ATV for the simulator. It presents all the modelled parts and discusses the choices made during the modelling process in order to make implementation to ROS as simple as possible.

In the fifth chapter, everything related to point clouds are described. The sensors and software needed to gather data and the methodology of how to visualise the data is presented.

The sixth chapter introduces the implementation of SLAM. The SLAM algorithms chosen is explained and discussed, as well as the methodology of integrating SLAM in the simulator software.

The seventh chapter describes structure and implementation of the model and world files to the simulator. The results section presents the complete simulator and its functionality.

The eight chapter describes Docker and the set up of a specific container for NVIDIA drivers. DockerFile practises are presented and discussed.

The final three chapters present the complete results, discussion and conclusions for the finalised project. Suggestions for further development of the simulator are also presented.

1.7 Definitions

Term	Definition
ATV	All-Terrain Vehicle
Autonomous	Freedom to govern itself
CAE	Computer-Aided Engineering
CAD	Computer-Aided Design
CAM	Computer-Aided Manufacturing
CoM	Centre of Mass
CPU	Central Processing Unit
DoF	Degrees of Freedom
ENU	East, North, Up coordinate system
GICP	Generalised Iterative Closest Point
GPS	Global Positioning System
GPU	Graphics Processing Unit
GTSAM	Georgia Tech Smoothing and Mapping
GUI	Graphical User Interface
HSE	Health, Safety and Environment
IMU	Inertial Measurement Units
KDA	Kongsberg Defence & Aerospace
LiDAR	Light Detection And Ranging
NDT	Normal Distributions Transform
Odometry	Estimated change in position and orientation
PCB	Printed Circuit Board
PCL	Point Cloud Library
Point Cloud	Set of data points in space
Pose	Position and orientation of all joints in a robot
ROS	Robot Operating System
SDF	Simulation Descriptive Format
SLAM	Simultaneous Localisation And Mapping
Voxel	Value on a regular grid in 3D space
XML	Extensible Markup Language

Table 1.1: Terms and Definitions

Chapter 2

Method

2.1 Development Method

During the preliminary phase of the project, the task manager platform Jira was taken into use [1]. The project contains a lot of different separate sections, that were worked on simultaneously. Jira has therefore been fundamental for keeping an overview of all the issues that are in progress and finished. An example of an issue in the agile board in Jira is shown in figure 2.1. Jira allows for the agile project management method Scrum, that is a management method where the work is carried out in short cycles called sprints. The sprint length for this project was set to two weeks, to fit the intervals of the bi-weekly reports and meetings with the group's supervisor.

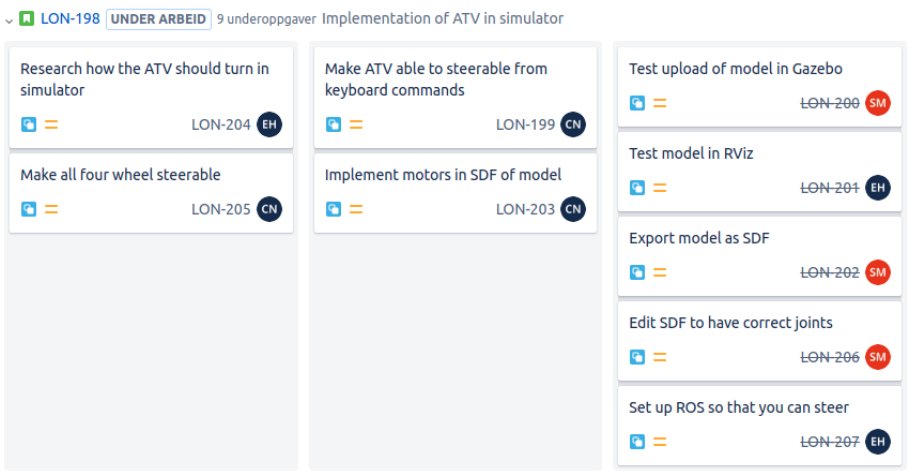


Figure 2.1: Agile Board of the Implementation Issue in Jira

Due to the project being a simulator, it is by nature software heavy. Several programming languages are utilised and most of the software has been developed by several project members simultaneously. To keep the software development agile, a Git repository was added. The use of Git allows version control of the software and collaboration within the team. When features were added or issues listed in Jira were resolved, the changes were committed to the main branch of the repository. A public Git repository provides a simple distribution of the software for further work and testing. The final commit of the repository is provided as a zip-file [2].

Microsoft Teams was used for file sharing within the group and to the supervisor. The use of MS Teams and Git ensured that each member is always up to date with the latest code or figures. Bi-weekly meetings and communication with the thesis supervisor were scheduled using Teams. Meetings with the client were scheduled and held using Skype for Business and Google Meet. LATEX was used as the main word processing program for writing this thesis. Like MS Teams and Git, this enables the group to work on the same file simultaneously.

2.2 Research Method

Developing a functioning test simulator for an autonomous vehicle requires several elements to function together. The group made an effort to divide the different parts into smaller, more manageable functions. This allowed for specifying the research on a more narrow topic, and dividing the topics among the group members. Because of the many element in this project, the research method varies for each topic. There was made an effort to use reliable and primary sources while gathering research material. Some of the topics in this project were completely new to the group members. This means that parts of the research covers the basics of these of the topics. The remaining research were built upon the knowledge already acquired from the Electrical Engineering programme at NTNU and the technical student organisations Ascend and Revolve.

2.3 HSE and Risk Assessment

A risk and HSE assesment were made during the preliminary project planning [1]. Seeing as this is a software project, there are not any risks tied to equipment or development method. The most significant risks in this project were related to the progress and quality, in addition to the client visit at Kongsberg Defence & Aerospace. All of these concerns have been assessed thoroughly in the preliminary project. The preliminary assessments have been sufficient and well managed as none of the risks have negatively impacted the quality or delayed the deadline for completion of the simulator.

Chapter 3

Robot Operating System

3.1 Introduction

Robot Operating System, ROS for short, is an open-source collaborative collection of software libraries and tools for making robot development easier [3]. Although the name states that it is an operating system, this is not accurate. ROS is a distributed framework of processes that works as a communication infrastructure that sends and receives information between different software programs. This infrastructure is described in detail in chapter 3.2.1.

Before ROS, the framework for robotics research had to be developed for each brand of robot [4]. The aim of the ROS project is to have a simple framework that lays a standard foundation for robot development and facilitate code reuse so that robotics research can focus on development of new technology, rather than reinventing the robotics framework for each new robot. The design of ROS enables simple distribution of software in addition to enabling independent decisions about implementation and development.

This chapter describes the functions of ROS 2, including communication infrastructure and file system. Project essential applications such as Gazebo and RViz along with libraries an packages used are presented. Finally the directory structure is presented and discussed along with the choice of ROS version and distribution.

3.2 Theoretical Framework

3.2.1 Communication Infrastructure

Nodes

Nodes are computation processes that allow them to communicate with one another using topics, services and the parameter server [5]. Nodes operate on a fine scale, this means that a robot control system usually consists of many nodes. The use of nodes in a control system reduces the complexity of the code compared to monolithic systems. They also provide an additional fault tolerance as crashes are isolated to individual nodes.

Topics

Topics are buses with graph resource names, over which nodes exchange messages [6]. Topics have a publish and subscribe functionality that can have several publishers and subscribers. Nodes are in general not aware of who they communicate with, they instead use the functionality that topics provide. Nodes that are interested in information subscribe to the relevant topic and nodes that generate data publishes this to the relevant topic. The types of information published to a topic can be vary considerably and can be user defined.

Services

The publish/ subscribe functionality of topics is a flexible way of communicating across an intricate system, but the many-to-many one-way model of transporting information is not appropriate in request and response interactions [7]. This interaction is instead done using services. A service is an action a node can take, that has a defined beginning and end, and will result in a single result. The service is defined by a pair of messages - one for the request and one for the response.

Messages

Messages are simple data structures comprised of typed fields [8]. Nodes uses messages to communicate with each other by publishing them to topics. Messages support primitive types such as integers, floating points and boolean in addition to arrays. Messages can also exchange a request and response message.

Parameter Server

A parameter serves is a multi-variate database shared between nodes [9]. This database is used by nodes to retrieve and store parameters while the program is running. This database is however not designed for high-performance, and is mostly used to store static or semi-static parameters. The parameters stored in this server is globally viewable, this lets the tools easily inspect and modify the state of the system if necessary.

3.2.2 File Structure

Workspace

A workspace is a directory used to organise a ROS project [10]. This directory contains at least three sub-directories called *build*, *install* and *src*. The *src* is a manually created file that contains the entire project's packages. The *build* and *install* folders are automatically created within the workspace when you use the ROS build function colcon in ROS 2. When you build a workspace several *setup.*sh* files will appear in the *install* file. If you source these files, the workspace will overlay on top of your ROS 2 environment and make it possible to access the packages.

Packages

ROS 2 software is organised into packages that can be consideres as a sort of container [11]. Packages make ROS 2 code shareable with others, because it allows others to easily build it and it is compact enough to be usable by other software. The contents of a ROS 2 package can vary considerably, but they should have enough

functionality for it to be reusable by others. The *atv_pkg* and *controller_pkg* folders that contain configuration, launch and setup files are examples of packages in this project. Packages are not required to contain nodes, they can just hold datasets as long as they are a useful component.

ROS 2 packages can be either CMake or Python packages, each of them having their own minimum required contents [11]. Both types of packages need to have a *package.xml* file that contains meta information about the package. CMake packages also needs to contain a *CMakeLists.txt* file that describes how the code is built within the package. Python packages have a *setup.py*, *setup.cfg* and <*package_name*> file in addition to the *package.xml* file. The *setup.py* file contains instructions for how to install the package. The *setup.cfg* is required when the package has executables so that the *ros2 run* command can find them. The <*package_name*> is a directory with the same name as your package, that the ROS 2 tools use to find you package.

Simulation Description Format

SDFormat is an XML format that describes objects and environments for robot simulators, control and visualisation [12]. This format is capable of describing all aspects of robots with links, joints, collision objects, visuals, and plugins. The format also describes environments with dynamic and static object, terrain, lighting and physics. This format is capable of running within a simulated world in the Gazebo physics engine, and allows communication from the controller to the links.

Launch Files

In order to start a system with multiple nodes, you can run the *ros2 run* command through the terminal in the specific order of the system. This can be tedious and time consuming, and allows for a large margin of error. Instead of starting the system node for node, the launch file system in ROS 2 can write a call for any number of nodes in a specific order and define any number of parameters on the parameter server [13]. Launch files describes the entire configuration of the system and starts the nodes at once when you run the command *ros2 launch* in the terminal. ROS 2 launch formats can be written using python, XML and YAML. Launch files can be launched from within other launch files as well. This allows for launching of

isolated parts of the system and makes the debugging easier. By using launch files the starting of an application will be tidier, and the user will save a lot of time.

3.2.3 Project-Essential Applications

The following descriptions are fundamental ROS 2 applications to the project.

Gazebo

Gazebo is an open source three-dimensional robotics simulation software [14]. It consists of a collection of libraries with support for sensor simulation, and actuator control and can be implemented with ROS 2 using the *gazebo_ros_pkgs* package. Gazebo utilises several high-performance physics engines that enables rendering of shadows, lighting and texture as well as accurate simulations of perception sensors such as LiDARs.

RViz

Rviz is a three-dimensional visualisation software tool for robots, sensors, and algorithms [15]. It enables the users to see the robot's perception of its world, real or simulated. The purpose of Rviz is to enable the user to visualise the state of the robot. It uses sensor data to create an accurate depiction of the environment around the robot.

rqt

rqt is a software that provides a graphical user interface (GUI) with access to various tools in the form of plugins [16]. This makes the information from the ROS 2 operations easier to break down, and more user-friendly. The package rqt_graph provides a GUI plugin for visualising the ROS computation graph. An example of this can be seen in figure 3.1.

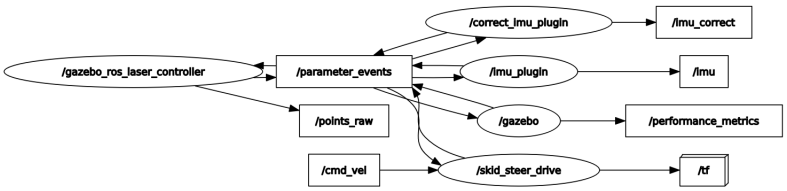


Figure 3.1: Example of rqt_graph

3.2.4 Project-Essential Packages

The following are ROS 2 packages that are fundamental to the project.

Gazebo Ros Package

The package *gazebo_ros_pkgs* includes wrappers, tools and additional API's for using ROS 2 with the Gazebo simulator [17]. The *gazebo_ros_api_plugin* plugin, located within the *gazebo_ros* package, initialises a ROS 2 node called "gazebo". It integrates the ROS 2 callback scheduler with Gazebo's internal scheduler to provide the ROS 2 interfaces. This ROS 2 API enables a user to manipulate the properties of the simulation environment over ROS, as well as spawn and introspect on the state of models in the environment. A secondary plugin named *gazebo_ros_paths_plugin* is available in the *gazebo_ros* package that simply allows Gazebo to find ROS 2 resources, i.e. resolving ROS 2 package path names.

The plugin *diff_drive_controller* is a controller for a differential drive wheel system [18]. This plugin is included in the *gazebo_ros_ pkgs*. The control is in the form of a velocity command, that is split and sent to two different joints. Odometry is computed from the feedback from the robot, and can be published.

Included in this package is another plugin called *skid_steer_drive*. This plugin has all of the same functions as *diff_drive_controller*, but it allows for specifying as many joints as needed. This allows steering four wheels. The plugin takes in multiple arguments, some of them can be seen in table 3.1.

Argument	Explanation
odometry_frame	Frame id of the odometry frame
update_rate	Number of iterations per second
max_wheel_torque	Maximum rotational force
covariance_x	Covariance in x-direction
covariance_y	Covariance in y-direction
covariance_yaw	Covariance in yaw
robot_base_frame	The frame id of the base link
wheel_separation	Distance between front/back wheels
wheel_diameter	Diameter of the wheel
publish_wheel_tf / publish_odom_tf	Publish transformations
left_joint	Front left wheel
right_joint	Front right wheel
left_joint	Back left wheel
right_joint	Back right wheel

Table 3.1: Arguments in Skid Steer Drive

The plugin for the simulated IMU is also included in *gazebo_ros_pkgs*. This plugin makes it possible to capture IMU data from the simulator and publish it on a topic with a *sensor_msgs/msg/Imu* message type.

Velodyne

The ROS 2 package *Velodyne* needs to be installed for the Velodyne VLP-16 LiDAR to work. The plugin *gazebo_ros_laser_controller* is included in this package and can be used to collect a point cloud from a simulated world.

Teleoperation Twist Keyboard

The *teleop_twist_keyboard* is a ROS 2 package for steering robots through a keyboard [19]. This package reads arguments from the keyboard and publishes the information as a *geometry_msgs/msg/Twist* message type to the *cmd_vel* topic. The message definition of this message type can be seen in appendix E. Some of the available actions for this package can be seen in table 3.2.

Argument	Action
u	Turn left while driving forward
i	Drive straight ahead
o	Turn right while driving forward
j	Turn left (in place)
k	Stop
l	Turn right (in place)
m	Turn left while backing up
,	Back up (straight)
.	Turn right while backing up
q/z	Increase/decrease max speeds by 10%
w/x	Increase/decrease only linear speed by 10%
e/c	Increase/decrease only angular speed by 10%

Table 3.2: Teleoperation Twist Keyboard Arguments

tf

The *tf* package lets the user keep track of multiple coordinate frames over time, and maintains the relationship between coordinate frames in a tree structure buffered in time [20]. The user can transform points, vectors, etc between any two coordinate frames at any desired point in time.

lidarslam_ros2

The *lidarslam_ros2* is a ROS 2 SLAM package where the front end operates with OpenMP-boosted GICP/NDT scan matching and the back end operates with graph-based SLAM [21].

LIO-SAM

LIO-SAM is a real-time lidar-intertial odometry package that transforms raw IMU data from the IMU frame to the LiDAR frame [22]. This package only works with a 9-axis IMU, which gives roll, pitch and yaw estimation.

li_slam_ros2

The *li_slam_ros2* package is a combination of two packages called *lidarslam_ros2* and *LIO-SAM* [23]. The package uses *LIO-SAM* for IMU composites and the SLAM algorithms from *lidarslam_ros2*.

3.2.5 Project-Essential Libraries

Point Cloud Library

Standard ROS 2 library for manipulation with point clouds, and is a standalone, large scale, open project for 2D/3D image and point cloud processing [24]. The library includes algorithms in registration, filtering, segmentation, and feature extraction. Tools for visualisation and manipulation with point clouds are also included.

g2o

The g2o is a pose graph optimisation library for graph-based nonlinear error functions [25]. It is the most used library for the pose graph optimisation, and offers well designed extendable interface which makes it easy to add a new definition of pose graph optimisation.

Eigen

Eigen is a open-source C++ library for linear algebra, and is a fast and well-suited library for tasks within heavy numerical computations, to simple vector arithmetic. The library includes modules for dense and sparse matrix representations, numerical solvers and transformation representation [26].

GTSAM

GTSAM is short for Georgia Tech Smoothing and Mapping (SAM), and is a C++ library that implements smoothing and mapping in robotics and computer applications. This includes SLAM [27].

3.3 Method and Equipment

In order to implement the different nodes, it is important that the topics have the right names. The publisher node publishes a message to a given topic, which the subscriber nodes listen to. This is visualised in figure 3.2. In some scenarios, in order to implement new packages, the topic names need to be changed to match.

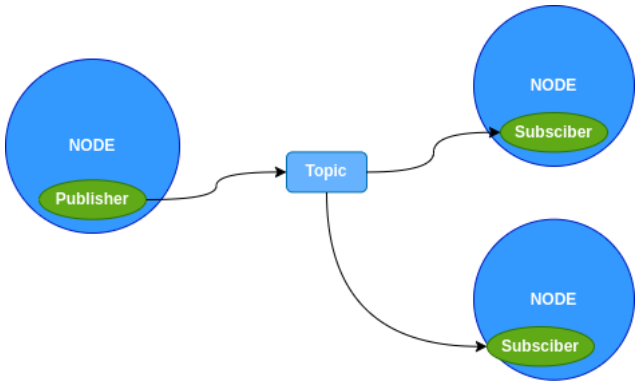


Figure 3.2: ROS 2 Graph [28]

In the project several packages are implemented, such as *velodyne*, *li_slam_ros2* and *teleop_twist_keyboard*. How these are implemented is described their respective chapters, see chapters 5.3.1, 6.3.1 and 7.3.4.

3.3.1 ATV Package

The ATV package is the package for launching the simulator, and is developed from scratch. This package includes the model and world files for the simulator, but also the launch file. The launch file, as mentioned in the theory section, launched the world and model in Gazebo.

After launching this package the data from the ATV's sensors is available for the other packages implemented through the ROS 2 framework.

3.3.2 ROS 2 Packages

Name	Description	Documentation
gazebo_ros_pkgs	ROS simulation gazebo	Open Robotics [29]
teleop_twist_keyboard	Teleoperation Twist Keyboard	Open Robotics [19]
lidarslam_ros2	LiDAR SLAM ROS 2 package	R. Sasaki [21]
LIO-SAM	LiDAR-inertial odometry package	T. Shan [22]
li_slam_ros2	LiDAR SLAM ROS 2 package	R. Sasaki [23]
ROS 2 binary packages	ROS 2 installation option	Open Robotics [30]
ROS 2 Foxy full desktop	ROS 2 distro and basic packages	Open Robotics [31]
Velodyne	Velodyne ROS 2 pakcage	Open Robotics [32]

Table 3.3: ROS Packages

3.4 Results and Empirical Findings

3.4.1 The Lonewolf Repository

The repository for this project is called is Lonewolf. Inside this repository the *build*, *install* and *src* folders are located. The *build* and *install* folders are automatically generated the first time the project is built. The *src* folder contains the integrated package *li_slam_ros2* and the self-developed *atv_pkg* package. The structure of the repository is shown in figure 3.3.

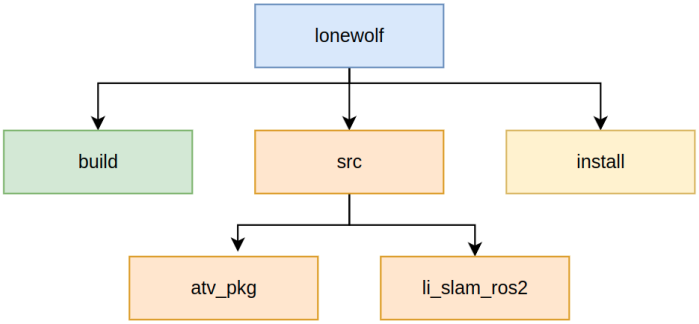


Figure 3.3: Lonewolf System Tree

3.4.2 ATV Package

An overview of the structure of the ATV package is shown in figure 3.4. The figure shows which layer the different layers the folders are located in, as well as the inventory of the folders. In the *atv_pkg* there are also three other files named *setup.py*, *package.xml* and *setup.cfg*, they are not included in the figure since they are automatically generated when creating a ROS 2 Python package. These files include installations of python modules, a manifestation of the package and configurations.

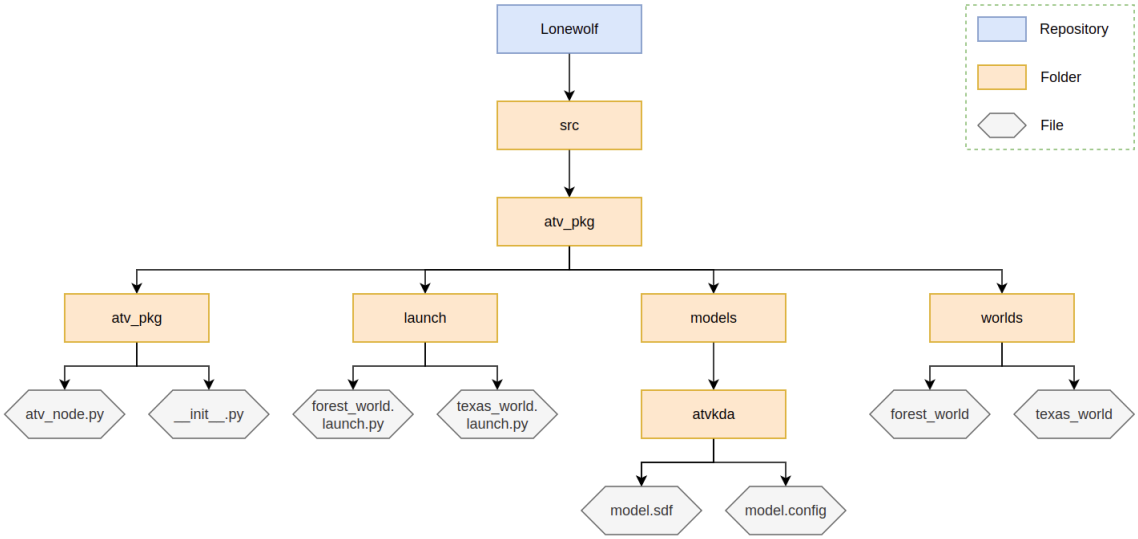


Figure 3.4: ATV Package Structure

3.4.3 System Flow

The whole system for the simulator is build upon the ROS 2 framework. The packages from the Lonewolf repository needs to be launched in order to start the simulator and the SLAM algorithms. To be able to navigate the ATV the Tele-operation Twist keyboard needs to be run as well. Simultaneously as the ATV is navigating, sensor data is gathered from the simulator. This data is fed in to the SLAM algorithms to generate modified data, which can be visualised in RViz. A graphical representation of the data flow is visualised in figure 3.5.

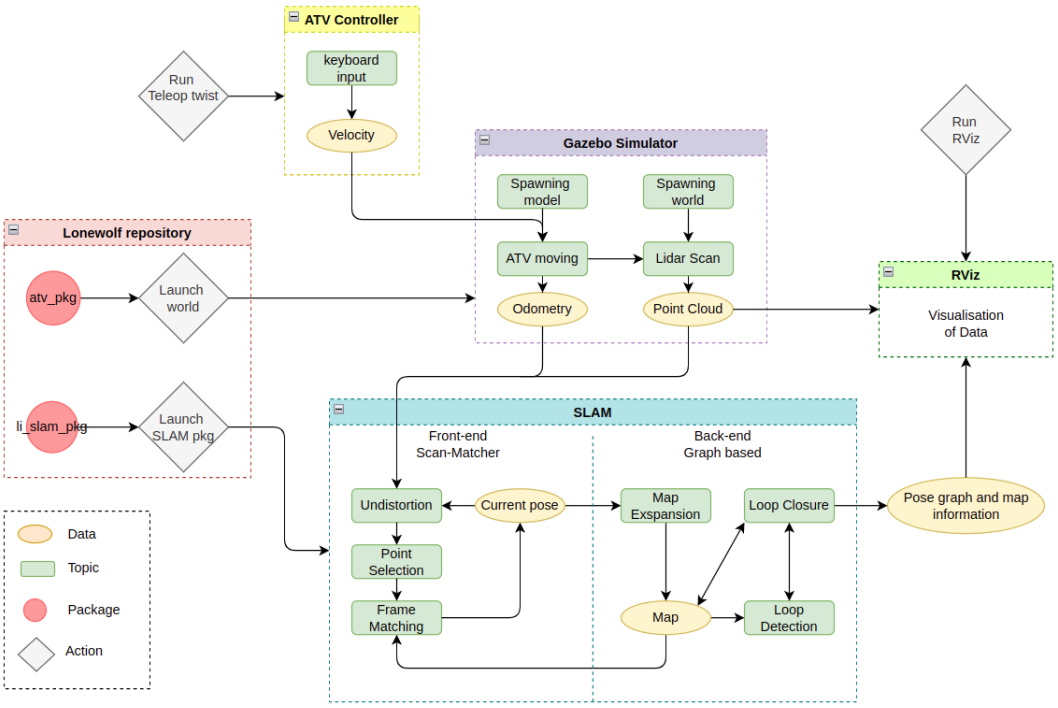


Figure 3.5: System Flow

3.5 Analysis and Discussion

3.5.1 Directory Structure

Many of the files in the lone wolf repository contains paths to models, worlds, meshes etc. A consequence of this is that it is crucial with a correct directory formation. If files are placed in the wrong directory, the simulator will not be able to find them. This results in various error messages and in worst case that the simulator won't launch. For that reason it is important to keep the structure and the directories tidy.

3.5.2 ROS Version and Distribution

The robotics industry has changed a lot since the ROS project was started. Although the ROS 1 project has had periodical improvements since the beginning, there has still been a need for a more adapted framework for the modern robotics community. The ROS 2 project leverages what is great about ROS 1 and has improvements for what missed. It is also worth mentioning that the ROS 1 project has an end of life

in year 2025 so ROS 2 might be a better framework for further development in the coming years. In addition the group members had more experience with ROS 2 in their work within aerial robotics prior to the project start.

Since the ROS 2 project is relatively new, there are still some functionalities that are not developed yet. Available packages were researched before a choice of framework was made. The conclusion of this research were that the needed packages had been included to the ROS 2 Foxy distribution. Taking all of these points into account, the ROS 2 Foxy distribution was chosen to be the framework in this project.

3.6 Chapter Conclusion

The setup of the Robot Operating System framework is successful, and the project is run entirely through ROS 2 Foxy terminal commands. The different ROS 2 packages and libraries is implemented, and works as intended. A tidy repository is set up to maintain an easy and manoeuvrable folder structure. The self-developed ATV package successfully launches the world and the model in Gazebo. Visualisations in RViz can be launched by terminal commands.

Chapter 4

3D model

4.1 Introduction

In order to have a functioning simulator for testing the Lone Wolf ATV, an model needs to be implemented in the ROS environment. The model needs to resemble the ATV both in looks and driving behaviour.

This chapter gives an overview of the modelling methods and choices made to model the ATV to the given requirements. The modelling of the individual parts will be explored in addition to a discussion on the choices made under the duration of the project.

4.2 Theoretical Framework

4.2.1 Bodies and Components

In Fusion 360, a body defined as any continuous 3D shape. They are physical object that exist in a component or in the global space [33]. A component refers to what is called a "part file". A component is a part that is capable of motion, and can serve as a container for design objects [34]. Components work as a organisation tool for designs, and the easy implementation of a component into other design files allow design reuse.

4.2.2 Joints

For the ATV to be able to turn and drive by wheel spin in the simulator, joints have to be defined between the wheels and body of the ATV. A joint allows the component to translate or rotate along or around the x, y or z axis. When modelling using a CAD software, there are several types of joints that can be defined between two components [35]. The three joint types compatible for SDF exportation are the revolute, slider and rigid joints. Revolute joints have one degree of freedom in rotational direction around the x, y or z axis. Slider, or prismatic, joints also have one degree of freedom but in the translational direction. This joint type makes the components slide along one another in the chosen axis. The rigid, or fixed, joint type fixes the two components to each other and provides no degrees of freedom. Rigid groups can be used to constrain multiple bodies to each other instead of creating multiple rigid joints [33]. This function locks the relative position of the selected components to one another. The rigid group is treated as a single object when it is moved or other joints are applied to the components.

4.3 Method and Equipment

4.3.1 Measurements

The project group travelled to Kongsberg Technology Park to get the specific measurements of the ATV. There has been done alterations on the ATV. The group focused on getting precise measurements of the additional parts KDA has made themselves. These measurements can be found in appendix B. The rest of the ATV is a standard model and its measurements can be collected in a data sheet which can be found in appendix C.

4.3.2 Modelling Software

In order to model the ATV accurately, a more precise modelling software than Gazebo was needed. This software has to create a model file that Gazebo and the ROS environment supports. Fusion 360 was chosen because it allows exportation of a model to a Gazebo friendly format by implementing a script and following specific rules [36]. Fusion 360 is a cloud-based 3D modelling, CAD, CAM, CAE, and PCB software platform for product design and manufacturing [37]. This modelling software has various functionalities such as precise modelling, animation and rendering. The software is free for students. It is also a big advantage that it is a cloud-based system, since there were multiple people working on the model simultaneously.

4.3.3 Modelling in Fusion 360

The modelling of the ATV was a complex process where the members of the group had responsibility for modelling the different parts before assembling everything together. Before modelling in 3D, the parts had to be drawn with the correct measurements as a 2D sketch. These sketches could then be transformed to a three dimensional part by using one of the functionalities in Fusion 360, such as the extrude and revolve tool. Each component was created as an isolated file. After all the components were completed, they were imported in to the main file and assembled.

All the bodies, apart from the wheels, were combined to one larger component. This was done to simplify the process of integrating the robot in Gazebo. This part is called the base link of the robot and can be seen in figure 4.1.

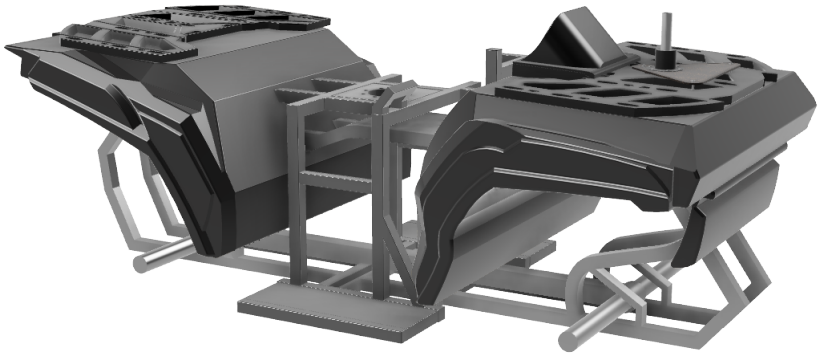


Figure 4.1: Base Link of the ATV Model

The joint functionality was utilised to connect the wheels to the base link. This joint type demands a parent link and a child link. For this model the base link is the parent and the wheel is the child. This causes that the wheels will be connected to the base link and that they rotate around a chosen axis. The joint between the base link and the wheel is shown with a blue circle and arrow in the middle of the wheel, in figure 4.2.

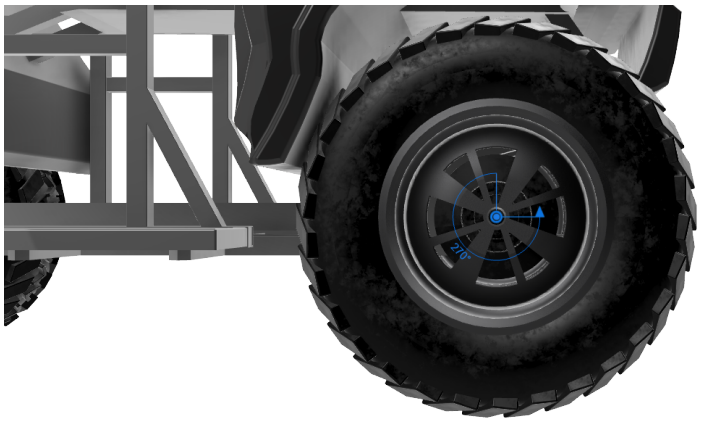


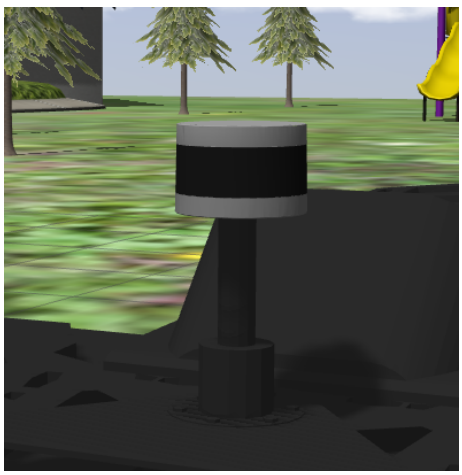
Figure 4.2: Wheel Joint

4.3.4 SDF Export

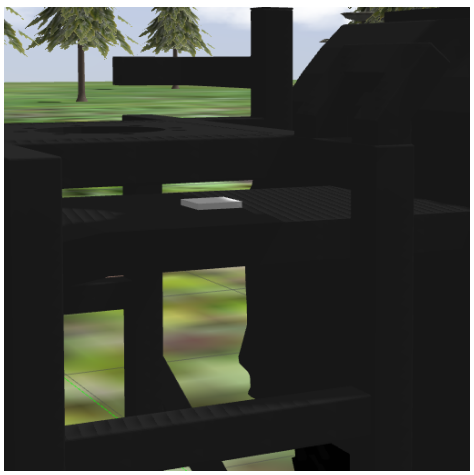
In order to export the ATV model from Fusion 360 to a world file in Gazebo, a script that can allows exportation in SD Format had to be followed. The ATV i modelled following a strict set of rules where the links have to be defined as rigid groups for the script to work. Joints have to be defined between components that belong to different rigid groups. Fixed, prismatic and revolute joint types are supported. Both links and joints have to follow a naming convention where links have to be defined as *EXPORT_link_name* and joints as *EXPORT_joint_name*. This is done to protect the structure of the model and avoid broken links when the model is uploaded in Gazebo. In this project, the entire body of the ATV is defined as a rigid body, from which all joints fork out from.

4.3.5 Sensor Implementation

The Velodyne VLP-16 LiDAR was implemented directly to the SDF file. There was already created a SDF model for this sensor that was available online. The IMU has a simple design since it is almost not visible on the ATV. It is design as a simple white box. Images of the sensors can be seen in figure 4.3.



(a) LiDAR Velodyne VLP-16



(b) IMU

Figure 4.3: Model of the Sensors

4.4 Results and Empirical Findings

4.4.1 Complete ATV Design

Parts modelled from measurements

Some of the parts of the ATV had well defined measurements taken from the ATV itself or from the part specifications. These include the tyres, rims LiDAR mount and frame as seen in figure 4.4. The frame is modelled completely hollow, just like the physical one is.

(a) LiDAR Mount (b) Frame

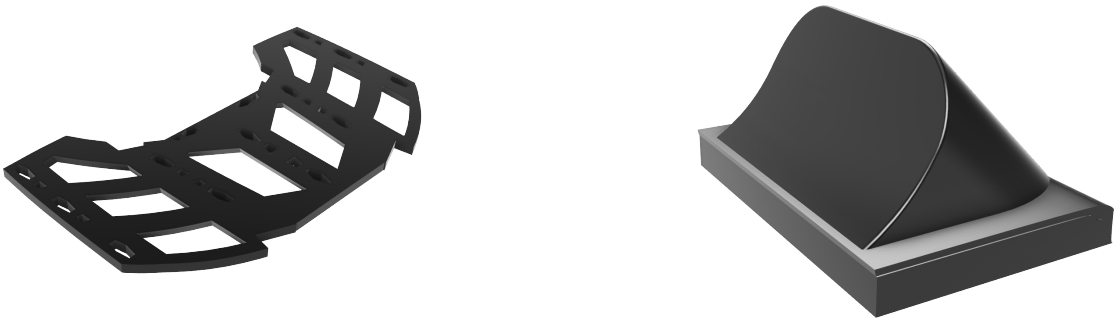
Figure 4.4: Model of Extra ATV Parts

Both front and rear wheels are modelled after the dimensions of the front tyres as seen in the data sheet in appendix C. Although the rear tyres are originally 5.2 cm wider than the front tyres, they were made to be the same width as the front tyres in order to make assembly and exportation to SDF simpler. All four wheels are therefore identical to the modelled front wheel in figure 4.5.

Figure 4.5: Modelled Front Wheel

Designed Parts

Some of the parts were designed after more vague measurements than the ones mentioned in the paragraph above. These parts were designed after pictures and measurements of the width, height and depth. The top plates and display in figure 4.6 are designed for the purpose of making the ATV look more similar to the physical one.



(a) Top Plate (b) Display

Figure 4.6: Designed Top Plate and Display

The chassis in figure 4.7 was designed after pictures of similar Can Am ATV models chassis' [38]. It was designed to fit the other parts, so that it is narrow enough for the frame to fit on top of it, and the wheels to fit on the sides of it without being too wide. Just like the frame in figure 4.4b, it is designed completely hollow.

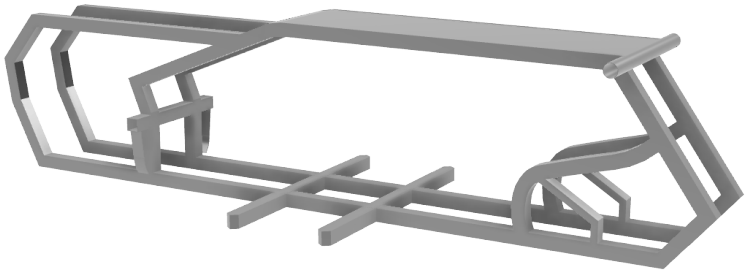


Figure 4.7: Designed ATV Chassis

The designed suspension for the model consists of a simple metal pole with two suspension rods perpendicular to the pole, as seen in figure 4.8. This is done to keep the implementation of joints between the suspension and wheels less complicated.

Figure 4.8: Designed ATV Suspension

The front and rear body parts in figure 4.9 were designed after measurements of the height, width and depth of the ATV. In addition to this, ground clearance and distance between the front and rear body were known. They were designed by sketching the side view of the ATV and extruding smaller parts varying length with the widest part at the bottom. The fillet tool was used to make it more similar to the physical vehicle.

(a) Rear Body (b) Front Body

Figure 4.9: Body Parts

The complete ATV with all the parts assembled can be seen in figure 4.10.

Figure 4.10: Complete ATV Model

4.4.2 SDF Export

In order to simplify the creation of the ATV model to use in ROS and get a similar looking model in the simulator, a script to export a SDF file directly from Fusion 360 was used [36]. A strict naming structure of links and joints had to be followed. The joints had to be either fixed, revolute or prismatic as well to avoid broken links when the file is uploaded to Gazebo. For simpler integration in Gazebo, all the links apart from the wheels were combined to the base link of the model. The link tree in Fusion 360 looks like figure 4.11. Fusion 360 automatically calculates the moment of inertia matrices and centre of mass of the model based on the material chosen for each part. Because this model is not complete and misses several parts from the physical ATV such as the motor, the moment of inertia and CoM was set manually after the model was exported to SDFormat. This, and how the model was implemented in the simulator in ROS is presented in chapter 7. The file structure after exporting the model can be seen in figure 4.12.

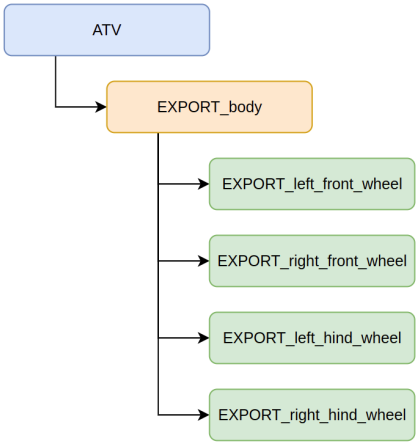


Figure 4.11: Part Configuration Tree in Fusion 360

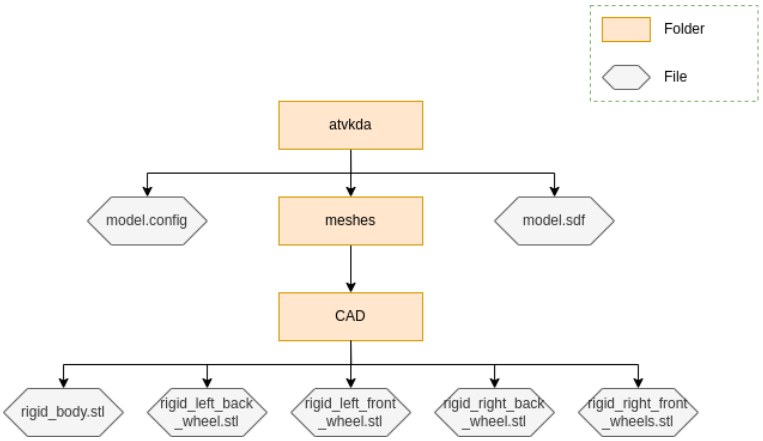


Figure 4.12: File Structure After Exporting

4.5 Analysis and Discussion

4.5.1 ATV Model

As can be seen from figure 4.10, the model is missing several parts from the actual ATV, like the motor and red tubing around the body. The aim of the modelling of the ATV has been to have a model that is similar in terms of looks and physics and that has the ability to move like a real life ATV. Given the time frame of the project, the focus has been on creating a model with correct measurements and joint configuration, rather than spending much time on implementing all the parts that were not pivotal for exportation to ROS. Had all the parts been implemented and modelled in Fusion 360, the exported SDF file would have had a more accurate moment of inertia and centre of mass.

4.5.2 ATV Parts

Chassis

Even after searching up the part number in online part dealers and reaching out to several Norwegian dealers, the measurements and pictures of the ATV's chassis were unattainable. Pictures of chassis' of similar Can Am ATV models were however attainable from online part dealers [38]. These pictures were all taken from an angle, so it was difficult to make a two-dimensional sketch based off of the side view from

the pictures. The chassis was therefore designed to look like the Can Am Outlander 1000 XT 12 chassis, and made to fit with the other parts of the ATV, such as the frame and wheels. If the simulator is further developed to include realistic load-bearing capacities of the ATV, the chassis will have to be remodelled to the correct specifications. These considerations have not been taken into account in this project, and the design of the chassis have no impact on the desired functionality of the simulator.

Wheels

At the first exportation of the model to SDF, the rear wheels were modelled after the specifications of the rear tyres meaning that the rear and front wheels had different width. While trying to implement right driving physics in Gazebo, the team ran into several problems with the wheels not being alike. It was difficult to get the ATV to drive in a straight line, because the centre of the rear and front wheels would not be in a completely straight line. The rear wheels were swapped out with the front wheels for that reason, making all four wheels identical. The original rear wheel model is included in the Fusion 360 file in the project's Github repository [2]. They can be implemented yet again for a more accurate simulation of the ATV's driving abilities.

Suspension

Although the ATV has a double A-arm front suspension and torsional trailing arm rear suspension, the design of the suspension has been kept simple. The designed suspension consists of a metal pole with two suspension rods perpendicular to the pole. Implementation of the actual suspension system would have been time consuming, and made it challenging to define the joints between the suspension and wheels. The joint definition would have been too complicated to allow simple development of driving behaviour in the simulator.

4.5.3 Assembly of Base Link Component

All the links, except the wheels and sensors, where combined into one link. This was done to make the SDF less complicated. By creating one rigid group the process of altering the inertia and mass centre got less complicated. This decision also made debugging the SDF much easier. All the links in the base link are static, which means that making one rigid group instead of multiple rigid joints between them makes no difference. One argument against this is that all the components in the base link get the same properties, such as visual material. With the time frame of the project in mind, the benefits of having a less complicated SDF was prioritised.

4.6 Chapter Conclusion

With the design and modelling choices presented in the results and discussion chapter, the model of the ATV satisfied the aim of having a model that resembles the physical ATV in terms of looks and driving behaviour. However there are still a lot of further development one can do to the model to improve the current simulations, as well as introduce new functionalities.

Chapter 5

Point Cloud

5.1 Introduction

A point cloud is essentially a huge collection of tiny individual points plotted in space. In this project the point cloud is gathered with a simulated VLP-16 LiDAR. This data is later fed in to a SLAM-algorithm that creates a map of the environment.

This chapter contains information about the Velodyne VLP-16 LiDAR, how the point cloud is collected and how it can be viewed in Rviz.

5.2 Theoretical Framework

Point clouds are made up of a multitude of points captured using a 3D laser scanner [39]. The scanner automatically combines the vertical and horizontal angles created by the laser bean to calculate a 3D x, y, z coordinate position for each point to produce a set of 3D coordinate measurements. The measurements often includes its colour value stored in RGB and intensity. The denser the points, the more detailed the representation, which allows smaller features and texture details to be more clearly and precisely defined.

There are two primary tools you can use to capture a point cloud: laser scanners and photogrammetry [39]. In this project a simulated VLP-16 LiDAR is used. This LiDAR is already in the Lone Wolf ATV inventory (see appendix D) and collects all the necessary data for this project.

Light detection and ranging (LiDAR) is a method that measures the distance to an object by illumintaing the object using an active laser "pulse", as can be seen in figure 5.1 [40]. The distance can then be calculated by

$$d = \frac{c \cdot t}{2} \tag{5.1}$$

where t is time of flight, c is speed of light in air and d is the distance to the object.

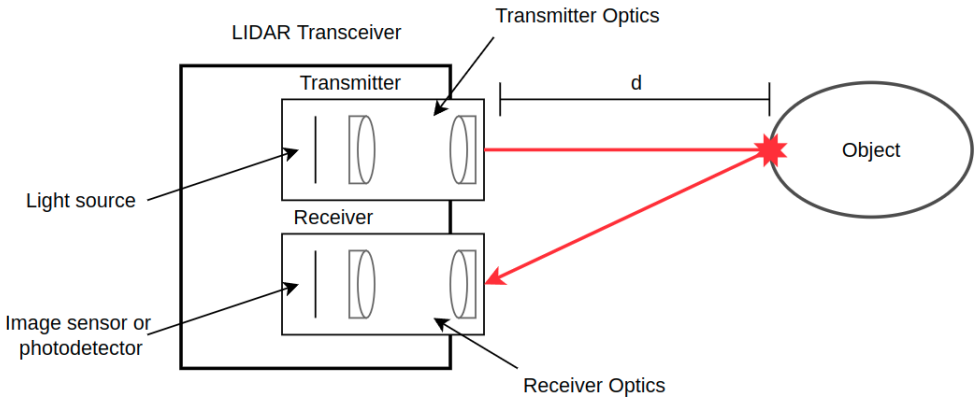


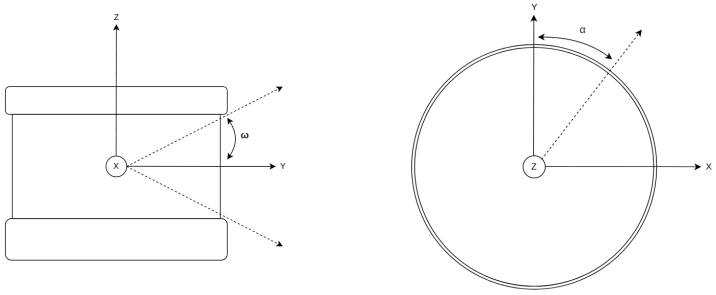
Figure 5.1: Example of How LiDAR Works [41]

Compared to other sensors, laser detection is more precise, and better suited for applications with high-speed moving vehicles. The laser sensor generally gives out 2D (x,y) or 3D (x,y,z) point cloud data. These point clouds provides high quality distance measurements, which works very efficiently for map construction.

The primary purpose of a point cloud is to create a 3D model [42]. In this project the purpose of the point cloud is to feed it in to the SLAM algorithm to make a map of the environment around the ATV. This map can then be used for autonomous driving.

5.2.1 Velodyne VLP-16 Specifications

The Velodyne VLP-16 creates 360° 3D images by using 16 laser/detector pairs [43]. The lasers are mounted in a housing that spins from 5 to 20 times a second. As a result, the scanner can acquire up to 300 000 points per second. More specifications can be found in table 5.1.



(a) Side View (b) Top View

Figure 5.2: VLP-16 Sensor Coordinate System [44]

Specification	Value
Channels	16
Maximum Measurements Range	50 m
Measurement Accuracy	± 3 cm
Field of View (Horizontal)	360 °
Angular Resolution (Horizontal)	0.1 °-0.4 °
Field of View (Vertical)	30 °(± 15 °)
Angular Resolution (Vertical)	2 °
Sample Rate (Adjustable)	5 Hz - 20 Hz

Table 5.1: Velodyne VLP-16 Technical Specification

5.3 Method and Equipment

5.3.1 Velodyne VLP-16 Specifications in SDF

The specifications of the Velodyne VLP-16 is included in the simulator. The code for the LiDAR is written directly in the *model.sdf*. This file can be seen in appendix G. The listing below shows how the specifications are included.

```
<scan>
  <horizontal>
    <samples>170</samples>           <!--Horizontal samples-->
    <resolution>1</resolution>
    <min_angle>-3.14159</min_angle>  <!--Min horizontal angle-->
    <max_angle>3.14159</max_angle>   <!--Max horizontal angle-->
  </horizontal>
  <vertical>
    <samples>16</samples>            <!--Vertical Samples-->
    <resolution>1</resolution>
    <min_angle>-0.261799</min_angle> <!--Min vertical angle-->
    <max_angle>0.261799</max_angle>  <!--Max vertical angle-->
  </vertical>
</scan>
<range>
  <min>0.5</min>                     <!--Min span in metres-->
  <max>50</max>                      <!--Max span in metres-->
  <resolution>0.001</resolution>
</range>
```

Listing 5.1: Velodyne VLP-16 Specifications in SDF

5.3.2 PointCloud2

To get a hold of the data the Velodyne VLP-16 produces, a plugin for Velodyne and ROS 2 is used. This plugin gathers range data from a simulated ray sensor, and returns results via publishing ROS 2 topic for point clouds. This is coded directly in the model file, which can be seen in appendix G.

The point cloud data from the LiDAR is gathered in the simulated world. Then

the data is published as sensor_msgs/PointCloud2 message on the topic /points_raw through the /gazebo_ros_laser_controller plugin as shown in figure 5.3.

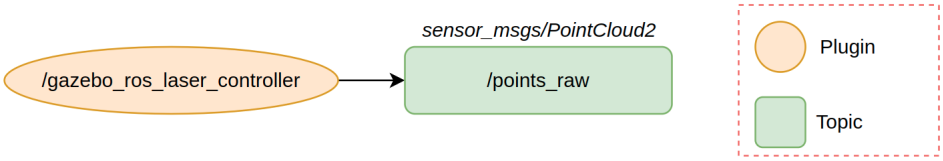
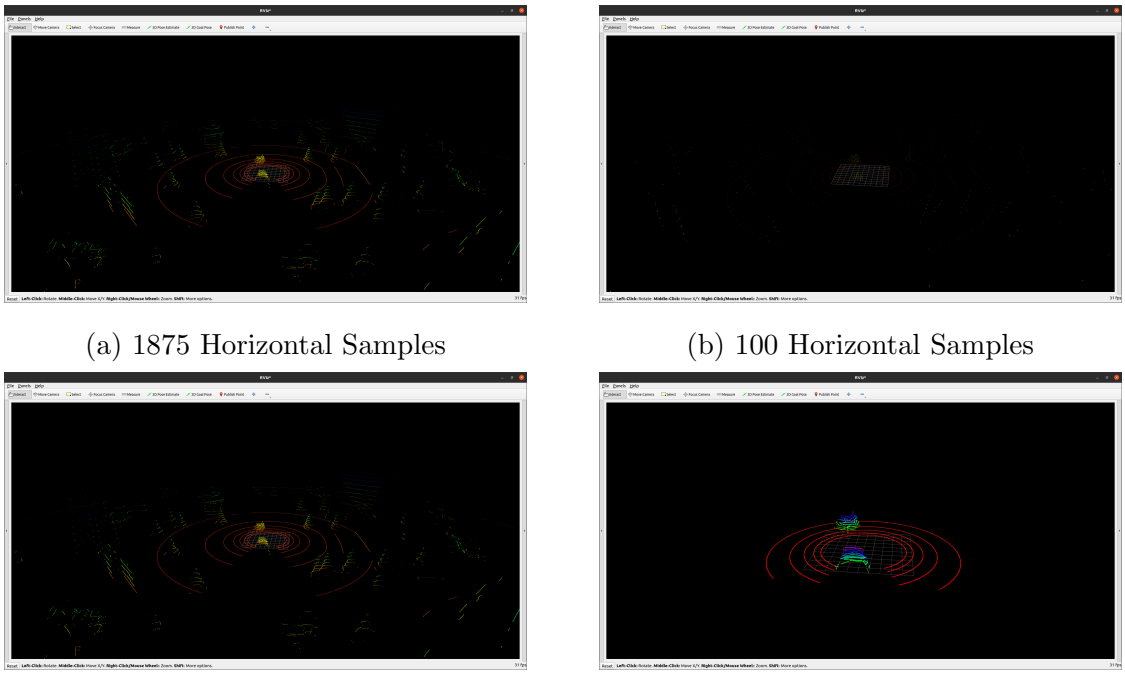


Figure 5.3: Production of the PointCloud2 Message

The message type sensor_msgs/PointCloud2 holds a collection of N-dimensional points, which may contain additional information such as normals and intensity [45]. The point data is stored as a binary large object (BLOB). The data may be organised 2D (image-like) or 1D (unordered).

In this project the point cloud is unordered, which means the data is stored in a 1D array, because the chosen SLAM algorithms doesn't demand an organised cloud. The definition of this message type can be found in appendix E.

After collecting the point cloud, it can be visualised in RViz. Using RViz gives an advantage when setting variables such as range and samples. Getting a live image of the cloud gives a better understanding of what the different parameters mean. An example of this is shown in figure 5.4.



(a) 1875 Horizontal Samples

(b) 100 Horizontal Samples

(c) Range 100

(d) Range 10

Figure 5.4: Visualising Point Cloud Data in RViz

5.4 Results and Empirical Findings

The purpose of the point cloud is to feed the gathered data continuously to a SLAM algorithm to generate maps. This means that the point cloud has to have a satisfactory quality, with the result that the maps are of a good quality. The parameter values in table 5.2 are sufficient values. The remaining values regarding the point cloud is decided by the LiDAR model.

Specification	Value
Horizontal samples	170
Vertical samples	16
Minimum range	0.5
Maximum range	50

Table 5.2: Sufficient Values for the Point Cloud

The result of the simulated LiDAR can be seen in figure 5.5. Figure 5.6 shows the position of the ATV when the point cloud was captured.

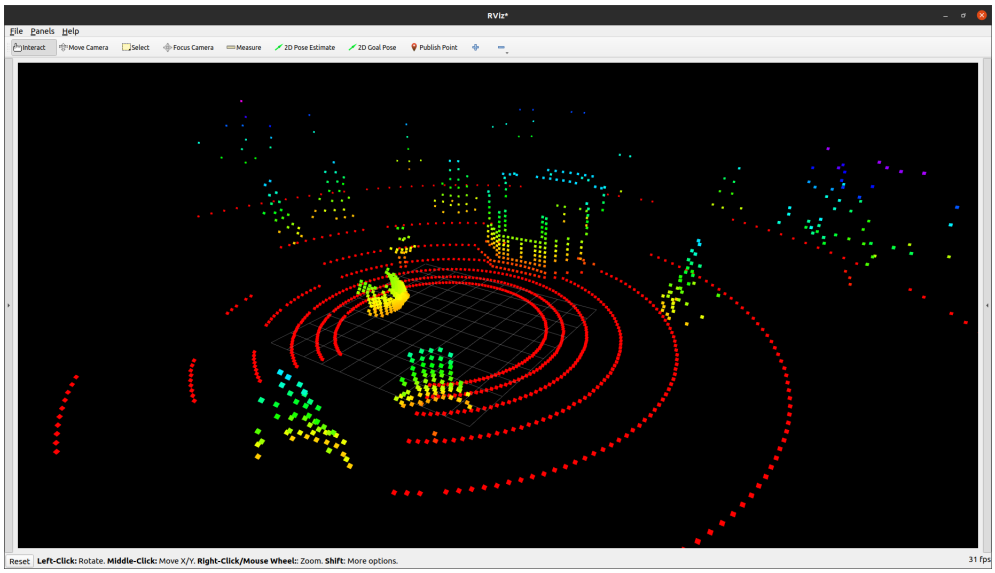


Figure 5.5: Point Cloud in RViz

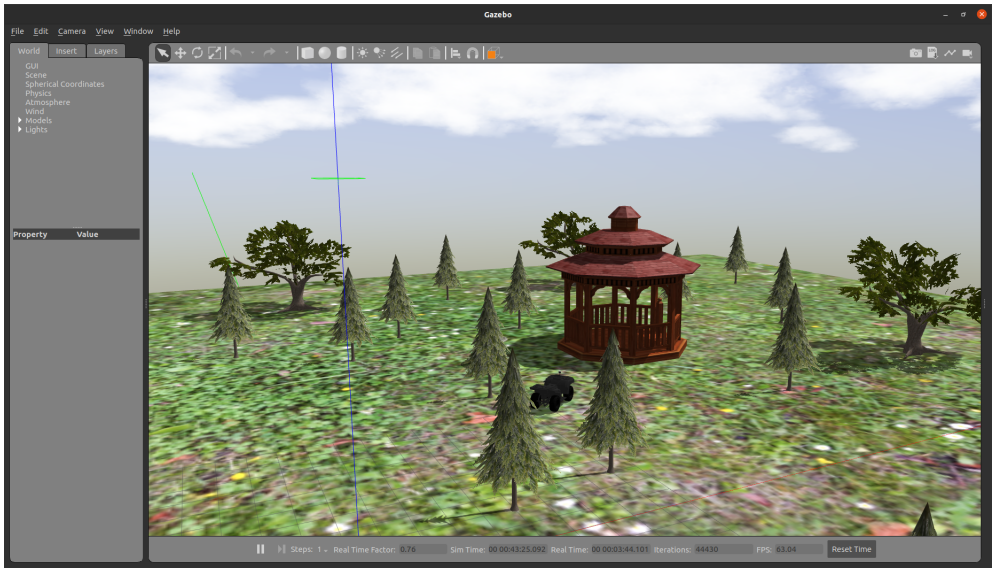


Figure 5.6: ATV Position When Gathering The Point Cloud Data

5.5 Analysis and Discussion

5.5.1 Ease of Implementation

Implementing the LiDAR VLP-16 and gathering the point cloud was very manageable. The code for this model was already available on Github and the plugin for the Velodyne laser was easy to understand. It is also simple to change the parameters.

5.5.2 Number of Samples

The point cloud consists of 170 horizontal and 16 vertical samples. The number of sampler needed, depends on the use-case of the point cloud and how detailed the map is required to be. For instance, obstacle avoidance doesn't require a detailed map of the surroundings. The algorithm for obstacle avoidance just needs to know if there is an object nearby that the ATV can crash in. But for other cases, where the user wants to capture shapes of buildings and surfaces, the point cloud needs to consist of more samples.

Taking 170 horizontal samples is a minimum regarding the quality of the map. This value is highly affected of the CPU on the computer the simulator is running on. 170 horizontal samples functions on a AMD Ryzen 7 4700U, but with a slower processing unit the simulator begins to lag. The point cloud in this simulator is used to generate maps for path planning and autonomous driving. 170 horizontal samples is enough to generate maps and paths with a satisfactory quality, but if these maps are adequate for autonomous driving needs to be tested in further development.

5.5.3 Range

The VLP-16 LiDAR has range up to 100 metres, but the range of the point cloud is set to 30 metres. The simulated world is so small that if the range is set to 100, the LiDAR maps almost the whole terrain before the ATV has started to drive. To test the SLAM algorithm, described in chapter 6, the ATV has to be able to drive and map simultaneously. In consequence the range had to be set lower. How short the

range can be depends on the environment. The LiDAR has to gather enough point cloud data for the SLAM algorithm to function properly. If the range is set to lower then 30 metres there is a great chance of that there isn't any objects near enough to scan. This can result in that the SLAM algorithms fails to locate the ATV.

5.5.4 3D vs 2D Point Cloud

Creating a three dimensional map demands a 3D LiDAR. Another option for gathering a point cloud is to use a 2D LiDAR to get a two dimensional map. 2D LiDARs will give out information about the floor plan [46]. If the point cloud data is used for navigation in a flat environment and the robot is not tall, a 2D LiDAR can be enough. However, when navigating in a forest or similar environments, there can be obstacles like hanging tree branches or pipes that the 2D LiDAR won't detect. In these cases 3D perception is necessary for detecting all obstacles. 3D LiDARs are also useful in other applications like terrain classification and segmentation.

5.5.5 Physical and Simulated LiDAR Compared

The project group has compared images of point clouds from a real LiDAR VLP-16 and the data the simulated one creates. The image in figure 5.7 shows a point cloud from a physical Velodyne VLP-16 LiDAR and figure 5.5 shows a point cloud from the simulated LiDAR. With the constrains of the GPU in mind, the simulated LiDAR in seems to imitate the physical one sufficiently.

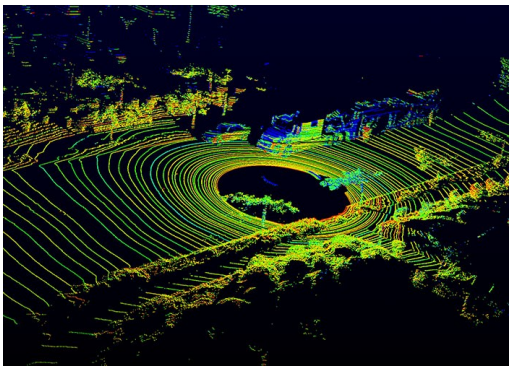


Figure 5.7: Point Cloud Crom a Physical Velodyne VLP-16 LiDAR [47]

5.6 Chapter Conclusion

A point cloud is generated from the simulated worlds. The simulated LiDAR represents the physical sensor well. The parameters of the LiDAR not decided by the the model specification, has been set through testing. Finding sufficient values for the point cloud contributes to better mapping when the SLAM algorithm is implemented. The point cloud seems to capture the environment around the ATV well.

Chapter 6

Simultaneous Localisation And Mapping

6.1 Introduction

SLAM is short for simultaneous localisation and mapping, sometimes also called synchronised localisation and mapping [40]. SLAM is the computational problem of constructing a map whilst keeping track of the location of the chosen mapping device. This process makes mobile mapping possible and allows map construction of large areas more efficiently.

This chapter presents SLAM and the different algorithms tested. For the pose estimation of the moving vehicle, the Scan-Matcher with NDT algorithm is used. Graph-based SLAM with GICP algorithm is implemented for the modified map generation.

6.2 Theoretical Framework

6.2.1 SLAM

There are many different approaches to SLAM, one of them called LiDAR SLAM. This approach uses the point clouds from the LiDAR to generate a map. The point cloud data generation can be read more about in chapter 5.2. Movement is estimated by matching point clouds, and is then used for localising the vehicle.

There are two types of technology components used to achieve SLAM, the first one being sensor processing [40]. This is front-end processing that is dependent on the sensors. The other type is pose-graph optimisation, including the back-end processing. Pose-graph optimisation is sensor-agnostic, meaning that the method or format of the data transmission is irrelevant.

The front-end of the process produces an intermediate representation of the sensor data. This consist of the point detection and tracking/matching as seen in figure 6.1. The back-end of the SLAM process computes an estimate given the represented sensor data. This is with map estimation.

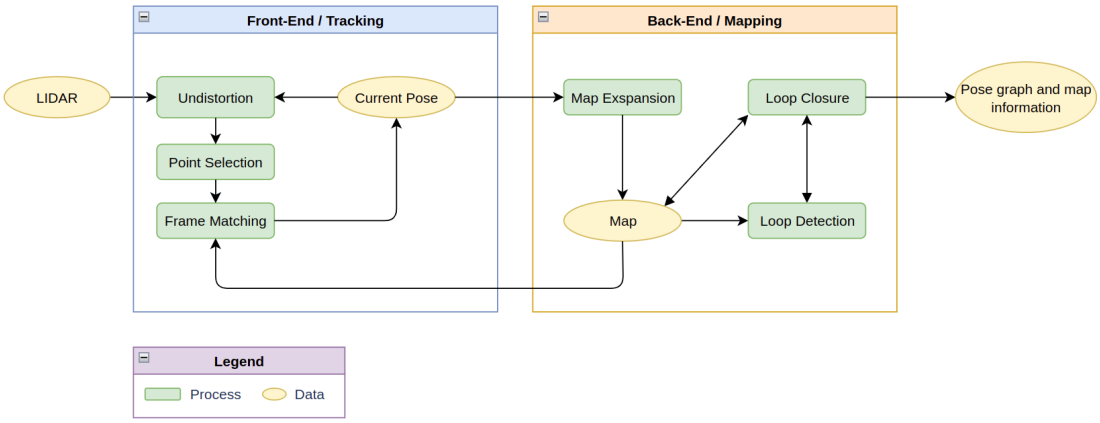


Figure 6.1: SLAM Processing Flow

6.2.2 Scan-Matcher

The *scanmatcher* package is an incremental laser scan registration tool, with the goal of finding the relative pose or transform between two positions where the scans were taken. The package allows to match scans between consecutive *sensor_msgs/msgs/PointCloud2* messages, and publish the estimated position of the laser as a *geometry_msgs/Pose* or a tf transform [48]. To see the definition of the message types, see appendix E. Scan-Matcher uses NDT algorithms to estimate the position.

Normal Distributions Transform Algorithm (NDT)

NDT algorithm is a registration algorithm that uses standard optimisation techniques applied to statistical models of three dimensional points to determine the most probable registration between two point clouds [49]. The idea is to represent the point cloud as a set of normal distributions. The surface constituted by the point cloud is divided into voxels, and projected into the normal distributions. The score calculation, average vector and covariance matrix is computed for each source point cloud. This is visualised in figure 6.2.

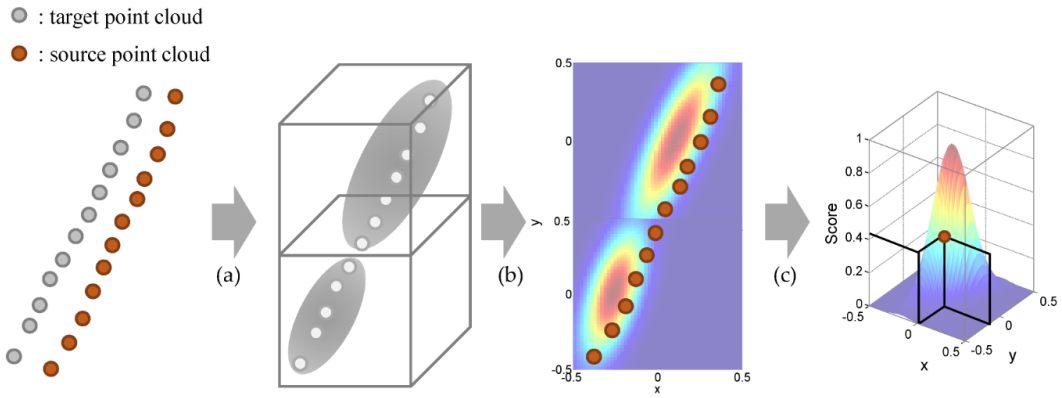


Figure 6.2: Process of Basic Normal Distribution Transform [50]

The first step in the NDT algorithm is called initialisation. Here the space occupied by the laser scan is subdivided into a grid of voxels. Each voxel is then assigned a portion of points. The voxel grid is a geometry type in 3D that is defined on a regular 3D grid, whereas the voxel can be thought of as the 3D counterpart to

the pixel in 2D. From each voxel a normal distribution, that locally models the probability of measuring a point $\vec{x}$, is computed. The location of $\vec{x}$ is generated by drawing from the distribution [51]. Drawing a sample from a distribution means observing a realisation of a random variable which has assigned that distribution to possible outcomes [52].

The second step of the NDT is called the "Matching" step. Here, a normal distribution function for each voxel is generated using space subdivision (dividing a space into a non-overlapping region). These distributions describes the clouds dispersion, but does not model the pieces accurately. The normal distribution can only locally model the points within the voxel, and does not always capture broader features. Each point in the model cloud is searched for the voxel that verifies this distribution.

The third and last step in the NDT is the transformation determination. In order to define the transformation, a cost function must be defined. The function is based on cloud intensity and the best estimation corresponds to the maximum intensity value of the image. The transformation is considered optimal at the maximum sum of normal distributions for all points with parameters 1. This sum is called transformation score, and is defined in equation 6.1.

$$Score(Trans) = \sum_i exp(\frac{-(u'_i - q^t_i)\Sigma_i^{-1}(u'_i - q_i))}{2}) \tag{6.1}$$

Equation 6.1: Cost Function [49]

$Trans$: Transformation (combining translation and rotation)
q_i : The average of the corresponding voxel
$\sum_i$: The covariance matrix of the normal distribution corresponding to point u'_i after applying NDT on the first scan.
u'_i : the point u_i mapped into the coordinate frame of the reference cloud according to the transformation $Trans$.

Finally, a nonlinear optimisation is performed to determine the transformation parameters.

6.2.3 Graph SLAM

A graph-based SLAM constructs a graph representation of the pose estimation problem and is an intuitive way to address the SLAM problem [53]. Solving the graph-based formulation involves the construction of a graph whose nodes represent robot poses or landmarks at a certain time stamp T.

The first step of the pose graph creation is to receive the robots movement. The transformation for the movement either comes from odometry measurements between sequential robot positions or are determined by aligning the observations acquired at the two robot locations. From the covariance and transformation, the edge or odometry edge for the graph can be created. The edges between two nodes represents a spatial constraint, that relates the two robot poses. The constraint consist in a probability distribution over the relative transformations between the two poses.

Summarily, the algorithm constructs a graph out of raw sensor measurements, where each node represents a robot position and a measurement acquired at that position.

After the graph is constructed, the problem is to find a configuration of the nodes that is consistent with the measurements. The graph is optimised by algorithms, either GICP or NDT. This results in the most likely position of all the nodes in the graph. This graph is called a pose graph, and the nodes represents the trajectory. An example of this can be seen in figure 6.3.

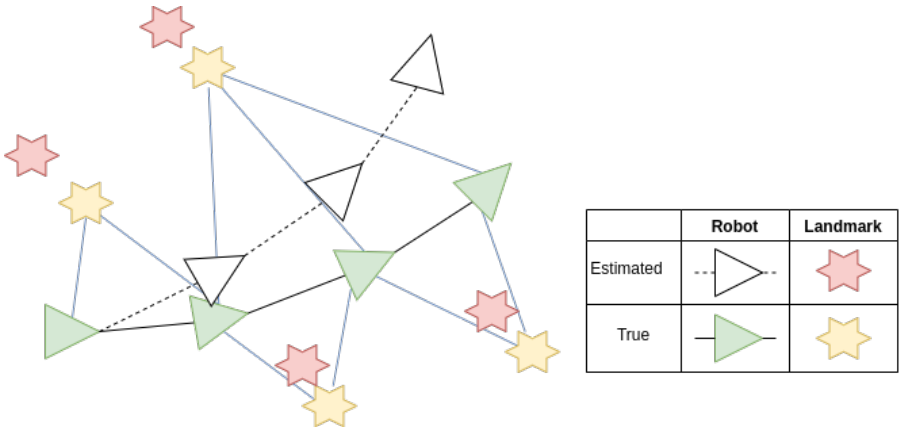


Figure 6.3: Graph-Diagram of the SLAM System

The graph-based SLAM is therefore decoupled into two tasks: construct the graph from raw measurements (graph construction) and determine the most likely configuration of the poses, given the edges of the graph (graph optimisation).

Loop Closure Creation

Loop closing happens when the robot reenters a known area after travelling for along time in a previously unknown region, and the algorithm seeks for matches of the current scan with the past measurements [53]. If the algorithm find matches between the current scan and the observation from a previous node, a new edge is added to the graph.

To determine which poses that overlaps, Dijkstra projection can be used. Dijkstra concatenate covariance and transformations along the minimum uncertainty path, which is selected based on the determinant of the covariance matrix [54]. By using the minimum uncertainty selection it is guaranteed that the algorithm will get from pose a to target b_i. The concatenation of covariance is done based on the following equations.

$$P_{a+b} = J_a P_a J_a^T + J_b P_b J_b^T \tag{6.2}$$

$$J_a = \begin{bmatrix} 1 & 0 & -xsin\theta - ycos\theta \\ 0 & 1 & xcos\theta - ysin\theta \\ 0 & 0 & 1 \end{bmatrix}, J_b = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{6.3}$$

where P_a and P_b is the accumulated covariance and additional covariance. J_a use the parameters from transformation $(x, y, \theta)_a$ and J_b from $(x, y, \theta)_b$.

After the generation of overlapping nodes is done, the potential pair needs to be tested by an registration algorithm. In order to avoid errors, the loop closure edges are grouped into groups based on their topological distance from each other. The inconsistent groups are deleted from the system after validation.

Generalised Iterative Closest Point (GICP)

Generalised ICP algorithms uses 3D point information to calculate point correspondences, distributions, and to perform the registration [55]. The generalised ICP algorithm deals with the iterative computation of the transformation. This approach is shown to be more robust to incorrect correspondences, and makes it easier to tune the maximum match distance parameter present in most variants of ICP. It maintains the speed and simplicity of ICP, and allow for the addition of outlier terms, measurement noise, and other probabilistic techniques to increase robustness. The ICP algorithm finds the transformation between a point cloud and the reference point cloud. A visualisation of this can be seen in figure 6.4. This is done by minimising the difference between two point clouds, and finding and applying the rotation and translation. The algorithm is therefore often used to reconstruct 3D surfaces from scans, to achieve optimal path planning and localising robots, etc.

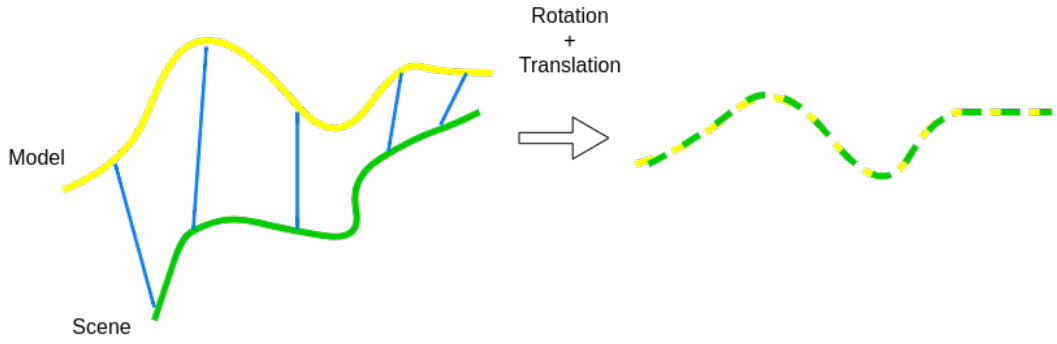
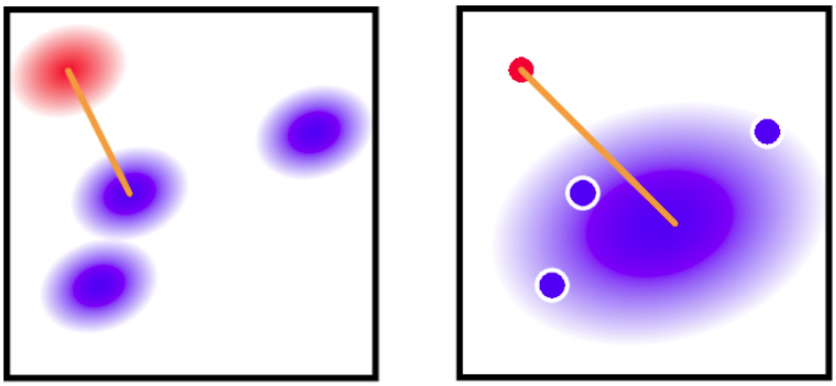


Figure 6.4: ICP Algorithm [56]

6.2.4 Difference Between GICP and NDT

The main difference between the GICP and the NDT algorithm is mostly how they measure the distance to a point [57]. The GICP algorithm measures the nearest distribution-to-distribution point, while the NDT algorithm uses voxel-based point-to-distribution. This is visualised in figure 6.5.

The NDT uses a collection of the points in a voxel to determine the distance, while the GICP uses the closest point. Figure 6.5 shows the correspondence models for distance calculation with GICP and NDT algorithms. The red circle indicate a source point and the blue circles indicate target points.



(a) GICP (b) NDT

Figure 6.5: Correspondece Models for Distance Calculation [57]

6.3 Method and Equipment

6.3.1 LiDAR SLAM ROS 2 Package

The *li_slam_ros2* package was cloned from repository on Github [23], and directly implemented in the *atv_pkg*. This package depends on multiple libraries. Eigen, PCL, g2o and GTSAM needs to be installed inside the Lonewolf repository. See section 3.2.5 for more info about the different libraries. To connect the *atv_pkg* and the *li_slam_ros2* package, the topic names needed to match. In order to visualise how the topics communicated, the ROS *rqt_graph* feature were used.

The *li_slam_ros2* package is mainly based on the *lidarslam_ros2* package, but is combined with the *LIO-SAM* package IMU composite method. This results in a combination between the Scan-Matcher front-end and Graph-Based algorithms from the *lidarslam_ros2*, with the IMU composite method from the *LIO-SAM* package.

Mathematical Calculations

The mathematical calculations for the different processes are computed by the support of the imported Eigen and PCL libraries. These libraries can be read more about in section 3.2.5 and 3.2.5

IMU Composite Method

The IMU composite method is a IMU preintegration method that corrects the odom data. The plugin takes the odometry from the Scan-Matcher and the IMU data from the IMU plugin, and summarise hundreds of inertial measurements into a single relative motion constraint. This is done with the support of the GTSAM library, which is described in section 3.2.5. The constraint is then sent back to the Scan-Matcher through the preintegrated_odom topic. The IMU preintegration flow is shown in figure 6.6.

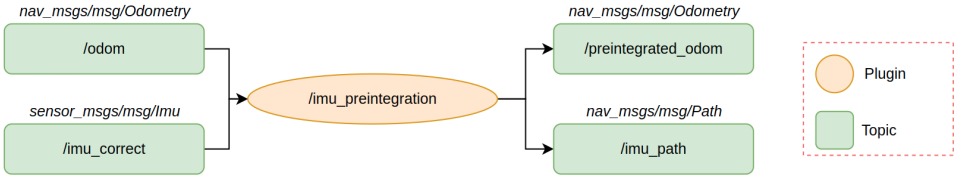


Figure 6.6: IMU Preintegration Flow

Scan-Matcher

The Scan-Matcher takes in two topics, *cloud_deskewed* and *preintegrated_odom*. The deskewed point cloud message comes from the Velodyne LiDAR, and is the raw point cloud assisted with data from the IMU sensor. The odometry message comes from the IMU preintegration. From this data the NDT algorithm estimates the pose, and finds the transform between the scans. The Scan-Matcher plugin then gives out the *odom*, *path*, *map* and *map_array* topics, visualised in figure 6.7. The pose is sent to graph-based slam through the sub-map included in the *map_array* topic.

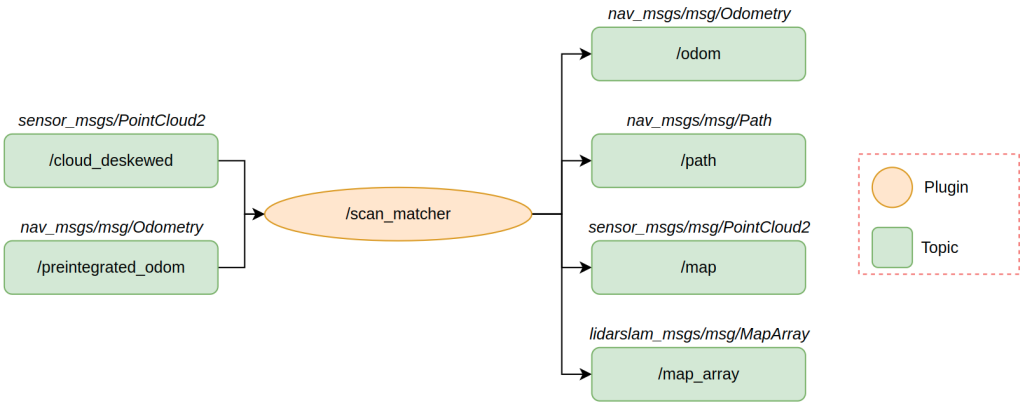


Figure 6.7: Scan-Matcher Process Flow

Graph-Based

The graph-based SLAM use the data from the scan-matcher to make modified data, see figure 6.8. The plugin takes in the *map_array* topic from the Scan-Matcher, and uses the GICP algorithm to calculate the point correspondences, which then results in a correction of the data. The correction results in a modified map, path and map array. This correction and optimisation is done using the g2o optimising algorithm, described in section 3.2.5.

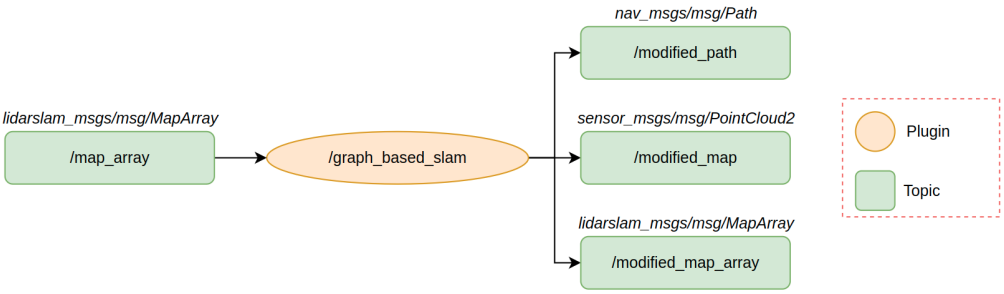


Figure 6.8: Graph-based SLAM Process Flow

6.3.2 SLAM in RViz

To visualise the map and path generated by the slam algorithms, RViz is used. The message types in this project are supported by RViz, such that they can easily be added to the program. Some of the available topics are map, modified map, modified path and path, which is visualised in figure 6.9.

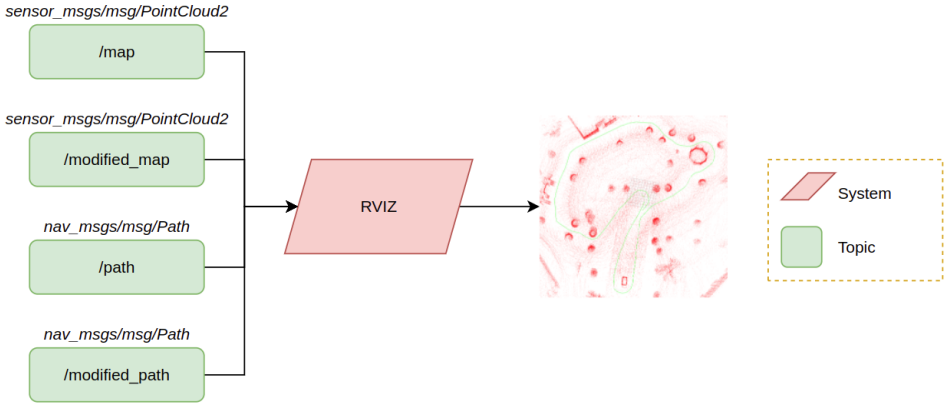


Figure 6.9: RViz Process Flow

6.4 Results and Empirical Findings

The purpose of the SLAM algorithms, is to use the point cloud gathered by the velodyne LiDAR to successfully generate a proper map. When running the *atv_pkg*, *li_slam_ros2* package and RViz, the generated data from the SLAM algorithms is visualised. Figure 6.10 is an example of the generated map and path after navigating in the Texas world.

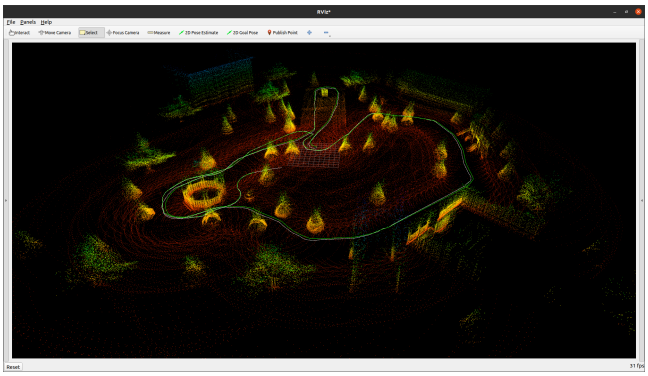
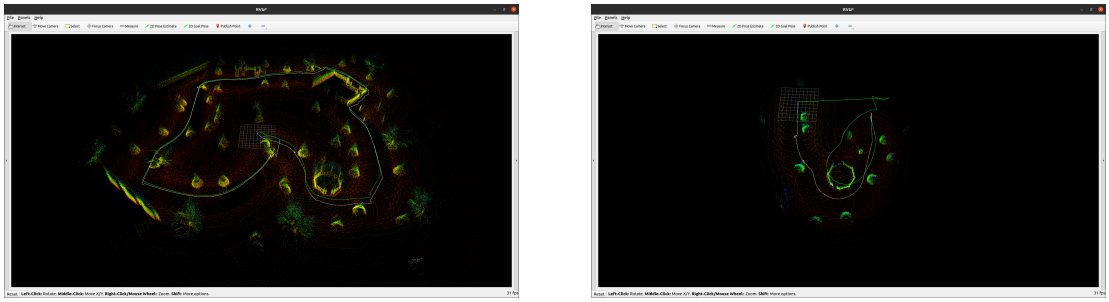


Figure 6.10: SLAM Map

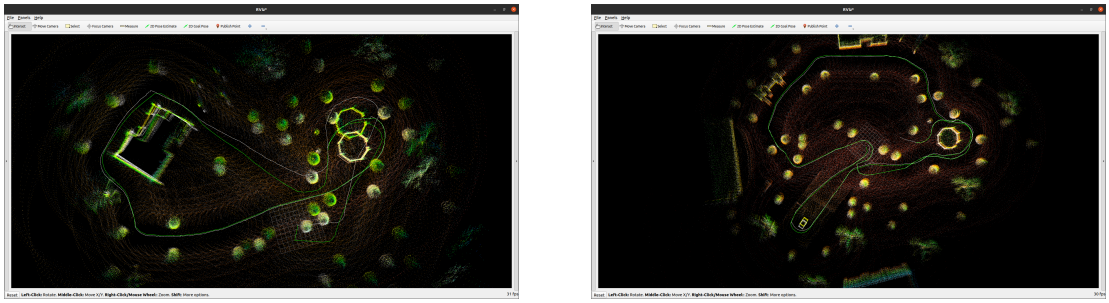
As described in section 5.5.3, the range on the Velodyne LiDAR can be changed. Alternating the range of the Velodyne LiDAR affects the quality of the generated maps. With range 30, the environment was mapped sufficiently, while with range 10, the environment was mapped poorly. Figure 6.11 shows the generated maps with the different ranges.



(a) Range 30 (b) Range 10

Figure 6.11: Visualising SLAM in RViz

The graph-based part of the *li_slam_ros2* package can be run on two different algorithms, the NDT and the GICP. Figure 6.12 a) shows the generated map while using the NDT as the registration algorithm. Figure 6.12 b) show the generated map while using the GICP as the registration algorithm.



(a) NDT (b) GICP

Figure 6.12: Graph-Based SLAM with Different Algorithms

After confirming that the simulator produces maps of sufficient quality with a horizontal ground plane, a ramp with an elevated plane was implemented. As shown in figure 6.13 the ATV mapped the ramp and elevated plane without any deviations in the quality.

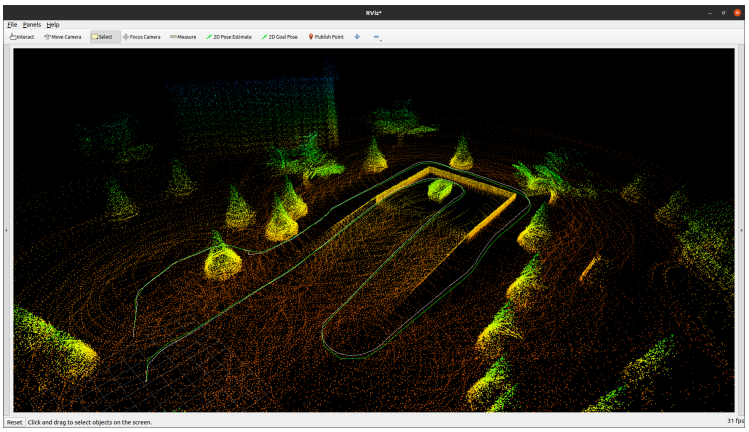


Figure 6.13: Map of Ramp and Elevated Plane

6.5 Analysis and Discussion

6.5.1 SLAM Package

The SLAM package has to be compatible with ROS 2 Foxy, since the entire project is built upon this framework. The SLAM algorithm is also required to make a three dimensional map from data gathered with a 3D LiDAR. These requirements is worked out through meetings with KDA. The data used to generate the map through SLAM needs to be supplemented by other sensor data. A package that fits these criteria is the *lidarslam_ros2* package made by Ryohei Sasaki [21].

After looking into the data flow and rqt graph it is discovered that the SLAM package does not include IMU data. This is caused by a bug in the package. As a result of this the *lidarslam_ros2* package is replaced with a LiDAR package that implements the IMU composite method 6.3.1. This package, *li_slam_ros2*, uses the same algorithms and works the same as the *lidarslam_ros2* package, but includes the correction stage of the IMU.

6.5.2 Scan-Matcher

The scan-matcher algorithm manages to work just by using the LiDAR scan data, but becomes more efficiently if it is supplemented with other sensor data. An example of this is to add the position data. By adding a guess for the current position of the ATV when a new scan message arrives, the process will be more efficient. The IMU is therefor implemented. As explained in the previous section 6.5.1, this also solved the multiple sensor input criteria.

6.5.3 NDT Algortihm

As mentioned, the Scan-Matcher uses the NDT algorithm for the scan matching. The PCL do have this algorithm implemented, but it is not optimal. Therefor, the *ndt_omp_ros2* repository is used instead. In this repository the algorithm is modified to be structural system and engineering friendly and multi-threaded. It can also run up to 10 times faster than its original version in PCL.

6.5.4 SLAM Algorithms for Back-End

For the back-end process of the slam, the graph-based slam is used. The package *li_slam_ros2* supports running two different registration algorithms for the back-end, GICP and NDT. When driving the same path two times in the simulator, the NDT algorithm generates a rather big drift in the modified output, and generates two gazebos, see figure 6.5. When the same scenario is tested with the GICP algorithm, the modified output does not suffer any significant drift, or generation of two gazebos.

This may happen due to scans not obtaining enough points for the normal distributions in the voxel. When the voxel size is small, it can only capture a few input points in one scanning, and there is only a few voxels having the normal distribution value.

6.5.5 Range

When testing with different ranges for the point cloud sampling, there is significantly better results with higher range. As shown in figure 6.11, when simulating with the range 10, the SLAM results in a weaker map. This is due to the algorithms not getting enough range to obtain the necessary points, which results in the SLAM algorithm struggling to localise the ATV. When simulating with range 30, the SLAM generates a map with higher quality. This is because with a higher range the LiDAR gathers more points, and the SLAM is therefor are able to map a bigger part of the area quicker.

6.6 Chapter Conclusion

The implementation of SLAM in the simulator works as intended. The SLAM package was implemented with Scan-Matcher and graph-based SLAM and the visualised map is of sufficient quality. To fulfil the criteria about supplemented sensor data a similar SLAM package is implemented. Different algorithms for the back-end processing is tested, which results in NDT being used for front-end while GICP for back-end. The map, path, and modified data gets published on topics, which can be visualised in RViz.

Chapter 7

Simulator in Gazebo

7.1 Introduction

A simulation program is used to simulate a real-life situation. The program creates a virtual version, that can be used for the purpose of testing or experiments. The requirements for the simulator given by the client is that the world had to be simulated in a ROS environment. The simulation program used in this project is Gazebo. As mentioned in chapter 3, Gazebo is a an open source three-dimensional robotics simulation software that is integrated in ROS 2.

This chapter gives an overview of the world file generated and the implementation of the 3D model of the ATV. The plugins used are also presented. The physics parameters of the model and worlds generated are discussed.

7.2 Theoretical Framework

7.2.1 World File

The world description file contains a description of all the elements that are used in a simulation. This includes robots, lights, sensors, and static objects. This file is formatted using SDF, and has a *.world* extension. This file is read by the Gazebo server (gzserver) to generate a world. Gazebo world files are written in XML, which can be generated and modified using a text editor.

The world file consist mainly of model declarations. A model can be a robot, sensor, static figure of the world (e.g. simple tree) or a manipulable object. As an example, the following declaration will create a simple tree:

```
<model name='tree_simple'>
    <static>1</static>
      <link name='link'>
        <pose frame=''>0 0 0.1 0 -0 0</pose>
        <collision name='collision_trunk'>
          <pose frame=''>0 0 2 0 -0 0</pose>
          <geometry>
            <cylinder>
              <radius>0.25</radius>
              <length>4</length>
            </cylinder>
          </geometry>
        </collision>
      <\link>
    <pose frame=''>49.5772894431 -0.301033806291 0 0 -0 0</pose>
</model>
```

Listing 7.1: Simple Tree

A set of attributes are associated with each models. These attributes describe the model's position and orientation and it is possible to compose models with them. The position and orientation is defined by the pose, $\langle pose frame\rangle$ 0 0 0 0 0 0 $\langle /pose\rangle$, where the first three numbers are the position in x,y and z coordinates, and the three last numbers are the orientation in Euler angles.

Canonical World File Layout

A world file can consist of many various components. In this paragraph a standard layout for the world file and some of the basic components are shown.

XML meta-data is placed at the beginning of the world file as can be seen in the listing below.

```
<?xml version="1.0"?>
<sdf version="1.5">
    <world name="default">
    ..
```

Listing 7.2: XML Meta-Data in World Files

The world file allows for setting global parameters. An example of this is setting the gravity for an environment that can be seen in the listing below.

```
<param : Global>
    <gravity>0.0 0.0 -9.8</gravity>
</param : Global>
```

Listing 7.3: Global Parameters in World Files

A robot needs a ground plane to be placed on. This can be included as shown in the listing below.

```
<model : GroundPlane>
    <id>ground1</id>
</model : GroundPlane>
```

Listing 7.4: Ground plane in World Files

Objects, such as trees and robots, can be included in the world as shown in the listing below. The object needs to be in SDF format.

```
<model name='tree_simple'>
    ....
</model>
```

Listing 7.5: Objects in World Files

To end a world file the two lines shown in the listing below needs to be written.

```
</world>
</sdf>
```

Listing 7.6: Objects in World Files

Coordinate Systems and Units

In the gazebo software the ENU coordinate system, with x in east direction, y in north direction, and z in upwards direction. This results in most models being designed upright along the z-axis and pointing along the positive x-axis. As mentioned in the previous section, the first three numbers are x, y and z coordinates while the last numbers indicates the objects orientation given by the Euler angles; roll, pitch and yaw.

7.2.2 Model File

A model file uses the same SDF format as world files, but should only contain a single $\langle model \rangle$ ... $\langle /model \rangle$. The purpose of these files is to facilitate model reuse, and simplify world files. Once a model file is generated it can be launched in the world using the launch file. In addition to the component introduced in this section, plugins are also a part of the model file. The implementation of plugins are presented in section 7.2.3.

Components of SDF models

The link element contains the physical properties for a defined body of the model. This can be a wheel, or a link in a joint chain. Each link might contain collision, visual and intertial elements. It is important to try to reduce the number of links, to reduce the complexity and increase the performance and stability. How the link element is written in the model file is shown in the listing below.

```
<link name="model">
    <inertial> .... <\inertial>
    <collision> .... <\collision>
    <visual> .... <\visual>
<\link>
```

Listing 7.7: Links in Model Files

A collision element encapsulates a geometry and is used for collision checking. How to include this in the model file is shown in the listing below.

```
<collision>
    <geometry> .... <\geometry>
<\collision>
```

Listing 7.8: Collision Element in Model Files

A visual element visualise parts of a link. This can be done by the geometry and material element. In the geometry element, meshes can be included using an $\langle/uri\rangle$ tag that describes where the meshes are located. How to use the visual element is shown in the listing below.

```
<visual>
    <geometry> .... <\geometry>
    <material> .... <\material>
<\visual>
```

Listing 7.9: Visual Element in Model Files

The inertial element describes the dynamic properties of the link, such as mass and rotational inertia matrix. How to include the inertial element is shown in the listing below.

```
<inertial>
    <mass> .... <\mass>
    <inertia> .... <\inertia>
<\inertial>
```

Listing 7.10: Inertial in Model Files

A joint connects two links. A relationship is established were one is the 'parent' and the other is the 'child'. This joint relationship can establish parameters such as rotation, joint limits and pose. How the joint element is included in a model file is shown in the listing below.

```
<joint>
    <parent> .... <\parent>
    <child> .... <\child>
    <pose> .... <\pose>
    <axis> .... <\axis>
<\joint>
```

Listing 7.11: Joints in Model Files

7.2.3 Plugins

A plugin is a shared library, often created by a third party. Plugins provides a simple and convenient mechanism to interface with Gazebo, and can be used to include functionalities such as sensor and steering of a robot. They can either be loaded through the command line, or specified in an SDF file as shown below.

```
<plugin name=' .... ' filename=' .... '>
  <ros>
    <namespace> .... </namespace>
    <argument> .... </argument>
  </ros>
  <frame_name> .... </frame_name>
  <output_type> .... </output_type>
</plugin>
```

Listing 7.12: Plugins in Model Files

Plugins specified in the command line are loaded first, then plugins specified in the SDF files are loaded. Some plugins are loaded by the server, such as plugins that affect physics properties, while other plugins are loaded by the graphical client to facilitate custom GUI generation. They can control almost any aspect of Gazebo, and are self-contained routines that are easily shared and can be inserted or removed from a running system.

There are six different types of plugins, where each type is managed by a different component of Gazebo. These are; world, model, sensor, system, visual and GUI. When the plugin is compiled as a shared library, the plugin can be attached to a world or a model in SD format.

7.3 Method and Equipment

7.3.1 World Files

According to the assignment, the ATV has to have a world fit for driving and navigation. This means that the world must contain a ground plane to drive on and objects to detect. The simulator has two worlds with different complexity, that can be used for testing. The worlds can be opened in Gazebo by navigating to the folder the world file is located in, and typing *gazebo world_name.world* in the terminal.

Forest World

The first world is called *forest.world*. This world was downloaded from Github from a repository called mrs_gazebo_common_recources [58]. The repository is developed by the Multi-robot Systems (MRS) group at Czech Techical University in Prague.

Texas World

The second world was created in Gazebo by inserting a ground plane and several models of unique objects. The models are a part of Gazebo's default model list, which is available by downloading Gazebo. This world is more complex because of the unique objects such as buildings, different trees and a ramp. The ramp makes it possible to drive uphill and test functionalities in upwards direction.

7.3.2 Model File

The model file is in SD format and was exported from the model in Fusion 360 via the SDFusion add-on. The file contains specifications of links and joints. The open-source sensor code was added directly to the model file. An overview of the items in the SDF of the model can be found in appendix F.

IMU Sensor in Model File

The IMU sensor is dependent on a plugin already included in the gazebo_ros_pkgs. For the sensor configuration, see appendix G. The IMU data is published to the topic *imu_correct* through the plugin *imu_plugin* as shown in figure 7.1. The correct part of *imu_corret* refers to the IMU being in the coordinate frame the SLAM algorithms requires.

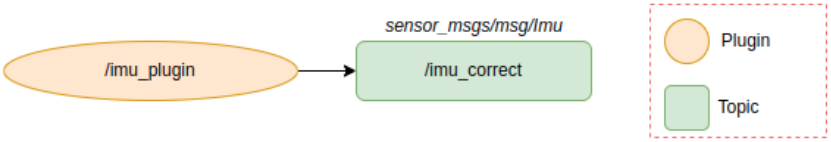


Figure 7.1: IMU Plugin

How the LiDAR Velodyne VLP-16 is included is described in chapter 5.

Physics

The SDFusion add-on for Fusion 360 didn't export the physics of the ATV correctly. This made it necessary to change the physics parameters. The mass, moment of inertia and friction elements were set by testing how the ATV was behaving while steering it. The chosen physics parameters can be seen in the *model.sdf* for the ATV in appendix G.

7.3.3 Implementation of ATV

Each world has its own respective launch file. These files have been created to spawn the ATV automatically inside the worlds. The launch files takes in the respective world file and the ATV package *atv_pkg*. When the launch files are run, they execute the process of first opening the world and then spawning the robot inside of it.

7.3.4 Implementation of Steering

In order to be able to drive and steer the ATV, the *teleop_twist_keyboard* package needs to be installed and run. As mentioned under 3.2.4, this package takes in keyboard commands through *teleop_twist_keyboard* plugin and publishes the information to the *cmd_vel* topic. This topic is used to send navigation commands to the *skid_steer_drive* plugin as seen in figure 7.2.

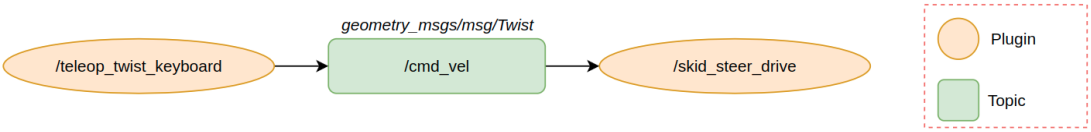


Figure 7.2: Topic and Plugins for Steering the ATV

7.4 Results and Empirical Findings

7.4.1 Model in SD Format

The ATV viewed in figure 7.3 is a result of the model file in SDF format. The ATV is a functioning ROS robot with controllable joints and sensors. The SDF that generated this model can be seen in appendix G.

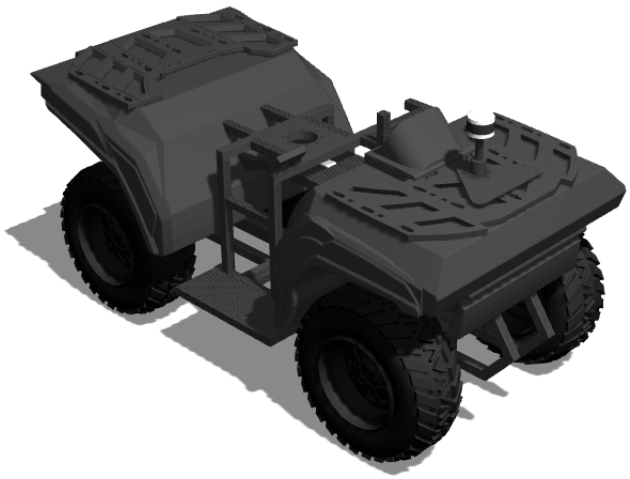


Figure 7.3: Model of the ATV in SDF Format

7.4.2 Worlds

Forest World

The forest world is a simple environment that consists of a ground plane and 500 identical trees. An image of the world can be seen in 7.4.

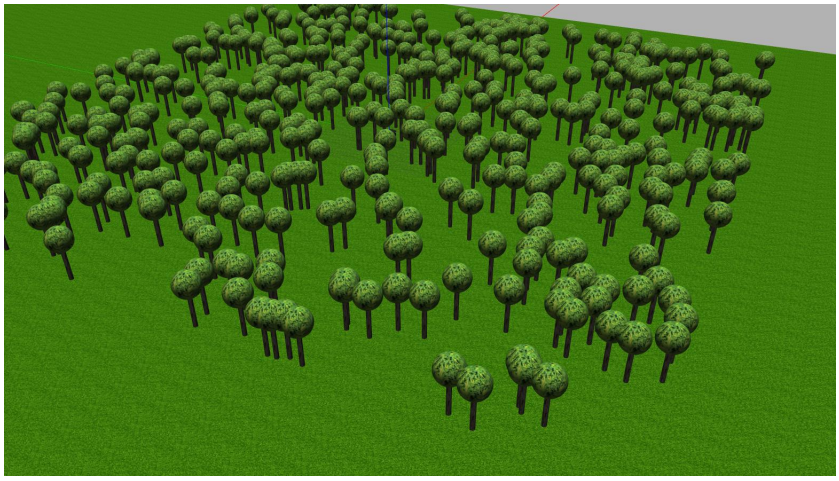


Figure 7.4: Forest World in Gazebo

Texas World

The Texas world consists of a ground plane, four different buildings, a play ground, trees, a gazebo, a ramp for driving in a higher plane and a water tower. An image of this world can be viewed in 7.5.

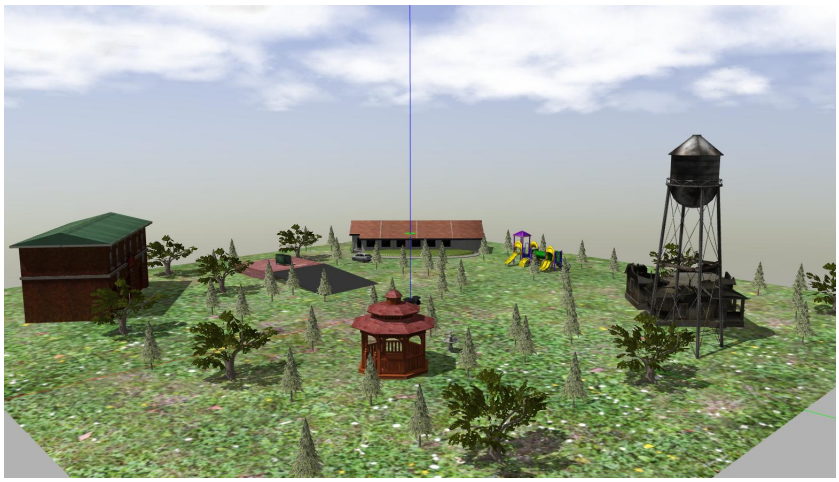


Figure 7.5: Texas World in Gazebo

7.5 Analysis and Discussion

The physics parameters were chosen by testing the steering abilities of the ATV. This resulted in incorrect physics compared to the ATV this model was made to imitate. Even though the physics of the model is different from the ATV, the ATV behaves well and more realistically when driving and turning with the chosen values than the correct values. This is probably because the model is more simple with less parts than the ATV actually has.

The IMU sensor code has integrated noise for making it more realistic. Some of the noise values has been altered after the sensor code was implemented to the model file. This was done to avoid too much drift in the IMU values, since the SLAM algorithm complain when the IMU has a large drift.

The Forest world consists of 500 identical trees. This makes the environment similar looking, regardless of where the ATV is located. A consequence of this is that it can make testing algorithms confusing because all the landmarks are identical. It can be hard to affirm that the real position of the ATV is continuous with the generated maps. The forest world works well with testing base functionalities of the algorithms and sensors, such as testing if the plugins work and that the algorithms actually generated maps.

The Texas world is more complex because all the elements, excluding the trees, are nonidentical. This solves the problem that the forest world gives. It is easier to see that the generated maps are correct in relation to the position of the ATV. In addition it is an advantage that this world enables testing of algorithms in various heights, since the physical ATV is most likely going to be tested in a terrain with height differences.

7.6 Chapter Conclusion

The simulator in Gazebo is a working test platform for SLAM algorithms. It consists of two worlds with different complexity and an ATV robot. The ATV is steerable through keyboards which makes navigation easy. The model of the ATV is not identical to ATV it is suppose to imitate, but a fair representation of it, since it is made for testing SLAM algorithms and not physical properties.

Chapter 8

Docker

8.1 Introduction

Docker is an open platform that enables a developer to separate their applications from their infrastructure and run it in an isolated environment that is called a container. This chapter describes the set up of a Docker container for developing the simulator in a standardised environment and the development practices for Docker are introduced and discussed. The container set up is targeted towards NVIDIA drivers, but are available for all machines.

8.2 Theoretical Framework

8.2.1 Containers

Containers package all code and dependencies together, this means that the dependencies does not need to be installed on the host machine [59]. Containers are shareable, meaning that several developers can be within one container and everyone that shares a container will have the same experience. Containers run as isolated processes in user space, several containers can therefore be run at the same time.

When developing within a container, an image is pulled from DockerHub. Docker images has everything needed to containerise an application, and have all code, configuration files, libraries and environment variables included [60]. When the image runs, a container is started. In order to build a tailored Docker image, the image needs to be built from a DockerFile. A Docker image consists of read-only layers stacked on top of each other. Each of the layers represent an instruction from the DockerFile [61]. Figure 8.1 shows how DockerFiles, images and containers are connected.

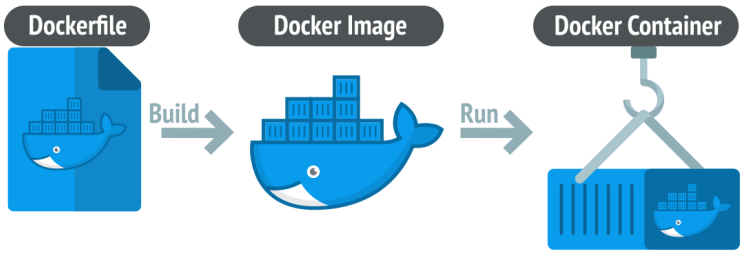


Figure 8.1: Structure of Docker Containers [62]

8.2.2 DockerFiles

A DockerFile is a text file that contains all commands needed to build an image [61]. Dockerfiles follow a strict format and rules so that Docker can automatically build the image by following the instructions in order. Some of the most important instructions are the *FROM*, *COPY*, *RUN* and *CMD* instructions, with each instruction creating a new layer. The *FROM* instruction in the DockerFile allows creation of a layer from an existing image. The listing below pulls the Ubuntu 20.04 image

and uses it as a parent image.

```
1 FROM Ubuntu:20.04
```

Listing 8.1: FROM instruction in DockerFiles

The *COPY* instruction copies new files or directories from the Docker client's current directive to the file-system of the container [61]. The *RUN* instruction executes any commands in a new layer and commit the results [63]. The resulting new image will be used for the next instruction in the DockerFile. The *CMD* instruction specifies what command to run and provide defaults for the container.

8.2.3 Multi-Stage Builds

Multi-stage builds allows several layers from existing images to build a tailored image [64]. When utilising multi-stage builds, the build stages needs to be named. Build stages are by default not named, and they are referred to by their integer number that starts from 0 with the first *FROM* instruction. By naming the build stages, the previous stage can be used as a new stage. This means the previous stage can be referred to by using the *FROM* directive.

8.2.4 Container Runtime

The container runtime is the container engine that is responsible for running containers on the host operating system [65]. It is the container runtimes that loads container images from a repository, isolates system resources for use of a container and manages the container lifecycle. By default, Docker does not add GPU devices to the containers runtime. By implementing the NVIDIA Container Runtime, the container runtime will be aware of the GPU and the GPU-accelerated application, along with its dependencies, will be wrapped into a single package [66].

8.2.5 Docker File System

The entire Docker file system is contained within the *.devcontainer* folder, where the files "*DockerFile*", "*devcontainer.json*" and "*docker-compose.yaml*" contain all

the instructions for setting up and entering the container. The entire folder tree can be viewed in figure 8.2. The *DockerFile* has already been discussed, and the two following are discussed below.

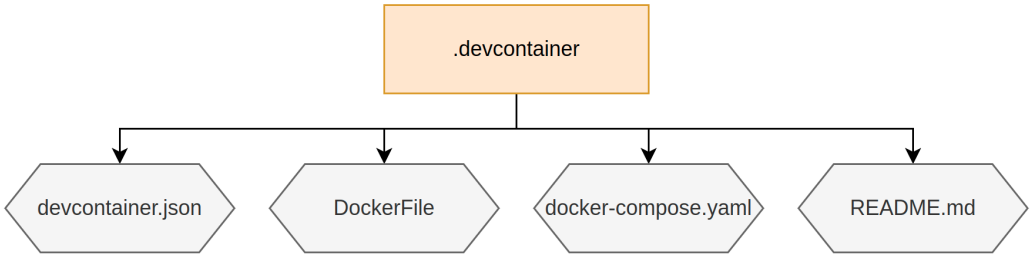


Figure 8.2: .devcontainer Folder Tree

The "*devcontainer.json*" file instructs VSCode on how to create or access a development container [67]. The file contains a path to the DockerFile and sets the context of what should be run within the container. With the context set to one level up, the parent folder the "*.devcontainer*" folder lies in, will be opened in the container. In this project, the development container will open the *Lonewolf* folder. The listing below shows the contents of the *devcontainer.json* file.

```
1 // Sets the run context to one level up instead of the .
     devcontainer folder
2 "context": "..",
3 // The 'dockerFile' property links to the DockerFile that is used
4 "dockerFile": "DockerFile",
```

Listing 8.2: Contents of devcontainer file

The "*docker-compose.yaml*" file defines networks, services and volumes for Docker applications [68]. Compose manages the entire lifecycle of the application by commands from the developer. Compose can for instance start, stop and rebuild services and view the status of running services.

8.3 Method and Equipment

8.3.1 Installation of Dependencies

This project requires a fair amount of dependencies in order to build and run the software. When developing software outside of a container on a computer with Ubuntu OS, dependencies are usually installed on the local computer with commands in the terminal. These commands can also be used when developing within a container, but it is easier to add all the dependencies in the DockerFile. This way the container is ready to run all software after the image is built.

The DockerFile is based on the *althack/ros2:foxy-gazebo-nvidia* image that allows running ROS 2 and Gazebo on NVIDIA drivers [69]. The image uses an Ubuntu 20.04 Docker image as parent image in the multi-stage build. The five build stages in the DockerFile are shown in the listing below.

```
1  FROM Ubuntu:20.04 AS base
2  # install ROS 2 Foxy
3
4  FROM base AS dev
5  # install dependencies
6
7  FROM dev AS full
8  # install full release of ROS 2 Foxy
9
10 FROM full AS gazebo
11 # install Gazebo
12
13 FROM gazebo AS gazebo-nvidia
14 #Expose the NVIDIA drivers and set NVIDIA Container Runtime
```

Listing 8.3: Multi-stage builds in the DockerFile

8.3.2 Setting Up the Container

To utilise the GPU of a NVIDIA machine, Docker needs to be installed by following the NVIDIA Docker installation guide [70]. The NVIDIA Container Runtime

is integrated by having the code lines in the listing below at the bottom of the DockerFile.

```
ENV NVIDIA_VISIBLE_DEVICES all
ENV NVIDIA_DRIVER_CAPABILITIES graphics,utility,compute
```

Listing 8.4: Setting NVIDIA Runtime in the DockerFile

To start developing within the container, the "Remote Containers" extension in VSCode is used. The *Add Development Container Configuration Files* extension is used to create the .devcontainer folder and the "*devcontainer.json*" file. The *Reopen in Container* extension opens the repository within the container.

8.4 Results and Empirical Findings

One is inside the container when the signature in the terminal is a root user like figure 8.3 below. The container is not accessed if the signature next to the cursor is the one set up by the local user. One can develop as usual within the container as one would outside of it.

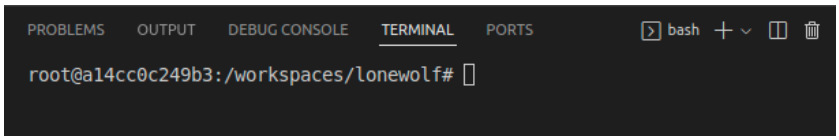


Figure 8.3: Terminal Within the Docker Container

8.5 Analysis and Discussion

8.5.1 Keeping the DockerFile Small

Most DockerFiles start from a parent image, and the Docker image for this project is no exception. One of the best practices for writing DockerFiles is to keep the image small. This can be done by using multi-stage builds and starting with an appropriate base image. The DockerFile currently uses multi-stage builds, but uses a generic Ubuntu image as base. If the DockerFile is set up using a ROS 2 Foxy for NVIDIA image as a base image, the ROS distribution and dependencies do not

need to be installed again each time the DockerFile is changed. This results in shorter build time. Although the that would be the best practice, the DockerFile uses a generic image in this project to allow for installation of dependencies at the appropriate stage in the multi-stage build of the image.

8.5.2 Launching Simulations

As of now, the simulation and visualisation in Gazebo and RViz can not be launched locally through the Docker container. To set this up, scripts that allows this needs to be programmed and integrated in the Docker set up. There is documentation on launching simulations from a container, but most of it is application specific and intended for ROS 1 distribution containers. The existing documentation and software could be pieced together to launch this project's simulations, but it is not done due to time constraints.

8.6 Chapter Conclusion

The Docker imaged developed in this project lays the foundation for further development in the ROS 2 Foxy distribution within containers. The container structure still needs advancements in order to use it for further development of the simulator.

Chapter 9

Results and Empirical Findings

9.1 Technical Design

The finished result of this project is an operative testing platform for testing different SLAM algorithms. A point cloud is gathered with a 3D LiDAR and processed with a SLAM algorithm which is supplemented with data from an IMU sensor. This results in a three dimensional map of the environment around the ATV which can later be used for autonomous navigation.

The user interface of the simulator can be viewed in figure 9.1. The interface consists of RViz (bottom left), which is used to visualise a map, Gazebo (right), which is used to view a world and the model, and the package *teleop_twist_keyboard* (top left), which is included for controlling the ATV. Having all of these applications open at the same time lets the user steer the ATV while seeing the generation of the map in near real time.

The simulator is made adaptable, which makes it easy to change which slam package to run. It is also made in such a way that it is easy to include new algorithms, worlds, sensors and models.

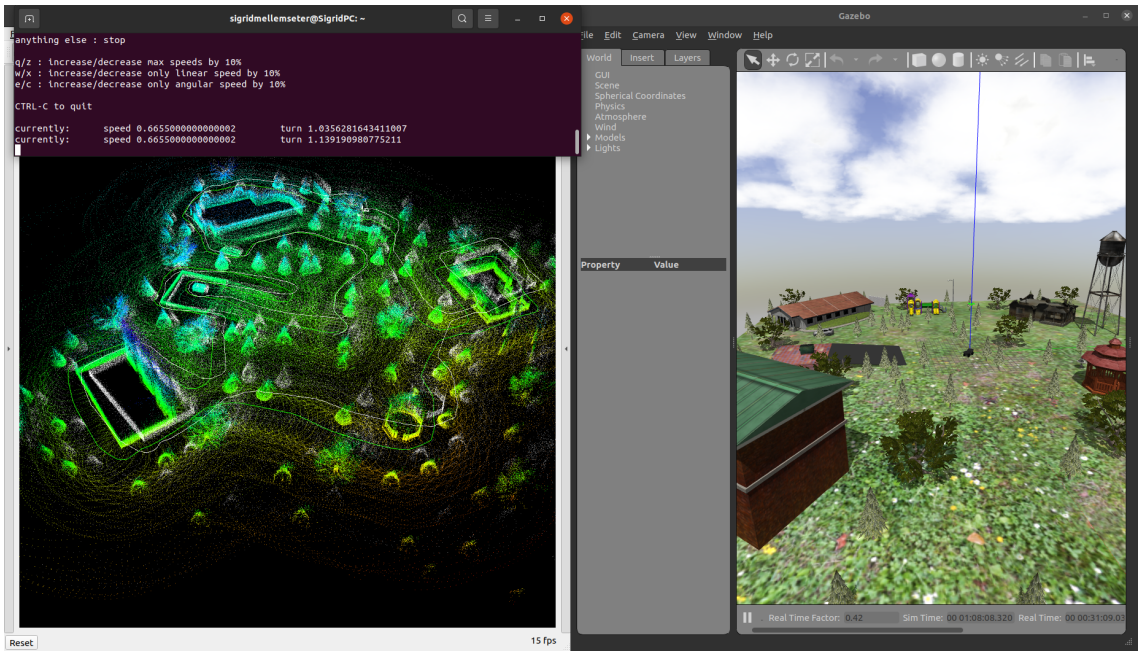


Figure 9.1: Simulator User Interface

Changing the parameters of the different features must be done in the source code. Most of the parameters of interest are in the SDF of the ATV model. This code is tidy and it should be easy to understand where to set the different parameters.

9.2 Deployment

The simulator is available by cloning the code from the *lonewolf* Github repository and following the instructions in the user manual. The repository is linked in reference [2] and user manual can be found in appendix H.

Chapter 10

Analysis and Discussion

10.1 Development Within Containers

When several people work on developing the same software, situations where the software works on one computer but not the other might occur. This can result in time consuming troubleshooting to resolve build and dependency issues. Developing within containers solves this issue because it isolates the application so that everyone that uses the container have the same experience. Had the team started development from within the containers right away, these troubleshooting sessions would probably have been fever and more spaced out. Containers also make the deployment of the application more simple as the developers know exactly how the application behaves within the container.

Although a Docker container has been set up for development from a NVIDIA machine, it was not set up for development from the personal laptops of the team. This was not prioritised at the start of the project, and the NVIDIA container was not set up until far into the project. Setting up this container to run smoothly is time consuming and calls for a lot of troubleshooting. For that reason, the team made a decision to prioritise other aspects of the project rather than setting up a container for everyone to develop from as well.

10.2 Future Development

10.2.1 ATV Model

Extended functionality of the simulator could be developed by implementing more of the ATV's characteristics. The physical ATV has an integrated electrical system with turning from a gear positioned where the handle bars normally are. This could be implemented in the simulator to achieve a more realistic resolution of the ATV's turning abilities. By replacing the *skid_steer_drive* plugin with another turning functionality, the ATV model can behave more like the Lone Wolf ATV.

As most combustion vehicles, the ATV does not have linear acceleration. This is not taken into consideration in the simulator due to the lack of an acceleration model to implement. This could be implemented by gathering data from the ATV and making a mathematical model to add into the SDF model of the ATV.

10.2.2 Communication Between ATV and Simulator

The simulator could be used as a "mission control" station for testing the actual ATV if communication between the simulator and ATV is set up. If the data sent from the ATV is implemented into the simulator framework, one could launch the data from the ATV the same way as the simulator, and view it in real-time. This would allow the operator to view and assess the quality of the data, and possibly kill the operation entirely in case of an unsafe situation.

10.2.3 Autonomous Navigation

The maps generated through SLAM algorithms can be used for autonomous navigation. This can be done in many ways. One of the options is to use the ROS 2 Navigation Stack with with 2D costmaps, since the 3D point cloud map can be converted to a 2D costmap [71]. The result of this is that the ATV will be able to navigate autonomously through both known and unknown environments. The most commonly used packages for localisation are the *nav2_amsl* package and

the *slam_toolbox*. Both of these packages publish the map to odometry coordinate transformation which is necessary for a robot to localise on a map. There are many packages and tutorials available online for this type of navigation.

Another option is to use the 3D point cloud map directly for navigation. Steve Macenski is the project leader of the NAV2 stack [72]. Macenski has developed the spatio-temporal voxel layer, which is new voxel layer leveraging modern 3D graphics tools to modernise navigation environmental representations. The *voxel_grid* ROS 2 package provides an implementation of an efficient 3D voxel grid.

Another alternative is to build a navigation package with the desired navigation and control algorithms. This would be the most time consuming choice, but would result in a bigger level of adaption and personalising.

10.2.4 Realistic World

To test the ATV in a more realistic environment, a complex world with terrain should be implemented. A possibility here is to model a ground plane with uneven terrain i Fusion 360, and convert it to SDF. After the model has been imported to the simulator, different models could be implemented to simulate the desired testing environment.

A more complex world, which is modelled to emulate the actual test environment for the real ATV, could be generated. Correct height maps can be ordered online of the desired area, to achieve a ground plane similar to the terrain. After uploading the height map, different models of the environment, such as trees, rocks and buildings, could be implemented to match the actual test site. With these adaptions the simulator would be an more optimal test platform, since the SLAM algorithms could be tested in an more similar environment.

Chapter 11

Conclusions

To sum up this project, conclusions will now be drawn. The system design came out as intended, with a clean and functioning file structure.

Two environments are available for testing autonomous navigation. In the model design, all parts were sufficiently modelled and successfully connected. The simulated ATV is equipped with multiple sensors from the ATV inventory list and it is possible to implement the remaining sensors.

An ATV model that gathers data in form of a point cloud with a 3D LiDAR has been made. Fitting SLAM algorithms in terms of complexity are tested. It is possible to continue development with the already included algorithms or implement new ones. The SLAM package is fully operational and able to generate a three dimensional map using the data from the point cloud, supplemented with IMU and odometry data. The generated maps have adequate quality for planning a passable path for autonomous navigation.

The ATV can be navigated through the simulator by commands from the keyboard. The entire simulator is operational in ROS 2 and available for further development. A functioning platform that enables testing of autonomous navigation has been developed.

Bibliography

[1] Eline Marie Håve, Sigrid Mellemseter, and Cecilie Nikolaisen. *ROS Simulated World For ATV With SLAM: Preliminary Projcect*. 2022.

[2] Eline Marie Håve, Sigrid Mellemseter, and Cecilie Nikolaisen. *Lonewolf*. 2022. URL: https://github.com/sigridmellemseter/lonewolf (visited on May 16, 2022).

[3] Open Robotics. *ROS/Introduction*. 2018. URL: http://wiki.ros.org/ROS/Introduction (visited on Mar. 2, 2022).

[4] Ricardo Tellez. *A History of ROS (Robot Operating System)*. 2019. URL: https://www.theconstructsim.com/history-ros/ (visited on Mar. 3, 2022).

[5] Open Robotics. *Nodes*. 2018. URL: http://wiki.ros.org/Nodes (visited on Feb. 28, 2022).

[6] Open Robotics. *Topics*. 2019. URL: http://wiki.ros.org/Topics (visited on Mar. 2, 2022).

[7] Open Robotics. *Services*. 2019. URL: http://wiki.ros.org/Services (visited on Mar. 2, 2022).

[8] Open Robotics. *Messages*. 2016. URL: http://wiki.ros.org/Messages (visited on Feb. 28, 2022).

[9] Open Robotics. *Parameter Server*. 2018. URL: http://wiki.ros.org/Parameter%5C%20Server (visited on Mar. 2, 2022).

[10] Open Robotics. *Creating a Workspace*. 2022. URL: https://docs.ros.org/en/foxy/Tutorials/Workspace/Creating-A-Workspace.html (visited on Mar. 2, 2022).

[11] Open Robotics. *Creating your first ROS 2 package*. 2022. URL: https://docs.ros.org/en/foxy/Tutorials/Creating-Your-First-ROS2-Package.html (visited on May 13, 2022).

[12] Open Source Robotics Foundation. *SDFormat*. 2020. URL: http://sdformat.org (visited on Mar. 24, 2022).

[13] Open Robotics. *Introducing ROS 2 launch*. 2022. URL: https://docs.ros.org/en/foxy/Tutorials/Launch/CLI-Intro.html?highlight=launch (visited on Apr. 6, 2022).

[14] Open Robotics. *About Gazebo*. 2022. URL: https://gazebosim.org/about (visited on May 9, 2022).

[15] Addison Sears-Collins. *What is the Difference Between RViz and Gazebo?* 2020. URL: https://automaticaddison.com/what-is-the-difference-between-rviz-and-gazebo/ (visited on Apr. 30, 2022).

[16] Open Robotics. *rqt_graph*. 2018. URL: http://wiki.ros.org/rqt_graph (visited on Mar. 3, 2022).

[17] Open Source Robotics Foundation. *ROS Communication*. 2014. URL: https://classic.gazebosim.org/tutorials?tut=ros_comm&cat=connect_ros (visited on Apr. 15, 2022).

[18] Open Robotics. *diff_drive_controller*. 2021. URL: https://wiki.ros.org/diff_drive_controller (visited on Apr. 16, 2022).

[19] Open Robotics. *teleop_twist_keyboard*. 2022-05-15. 2022. URL: http://wiki.ros.org/teleop_twist_keyboard.

[20] ROS.org. *tf - Package Summary*. 2021. URL: http://wiki.ros.org/tf (visited on May 3, 2022).

[21] Ryohei Sasaki. *lidarslam_ros2*. 2022. URL: https://github.com/rsasaki0109/lidarslam_ros2 (visited on May 11, 2022).

[22] Tixiao Shan. *LIO-SAM*. 2022-05-15. 2022. URL: https://github.com/TixiaoShan/LIO-SAM.

[23] Ryohei Sasaki. *li_slam_ros2*. 2022. URL: https://github.com/rsasaki0109/li_slam_ros2 (visited on May 11, 2022).

[24] PCL. *Point Cloud Library*. 2022-05-15. 2021. URL: https://pointclouds.org/.

[25] Rainer Kümmerle. *g2o*. 2022-05-15. 2021. URL: https://github.com/RainerKuemmerle/g2o.

[26] Wikipedia contributors. *Eigen (C++ library) — Wikipedia, The Free Encyclopedia*. [Online; accessed 15-May-2022]. 2021. URL: https://en.wikipedia.org/w/index.php?title=Eigen_(C%2B%2B_library)&oldid=1050274018.

[27] Georgia Tech BORG Lab. *GTSAM 4.1*. 2021. URL: https://gtsam.org/ (visited on May 16, 2022).

[28] Open Robotics. *Understanding ROS 2 nodes*. 2022. URL: https://docs.ros.org/en/foxy/Tutorials/Understanding-ROS2-Nodes.html (visited on Apr. 3, 2022).

[29] Open Robotics. *ROS Documentation*. 2022. URL: http://wiki.ros.org (visited on Mar. 2, 2022).

[30] Open Robotics. *Installation - Binary Packages*. 2022. URL: https://docs.ros.org/en/foxy/Installation.html (visited on Feb. 12, 2022).

[31] Open Robotics. *ROS 2 Documentation: Foxy*. 2022. URL: https://docs.ros.org/en/foxy/index.html (visited on Mar. 2, 2022).

[32] Open Robotics. *Velodyne*. 2022-05-15. 2022. URL: https://github.com/ros-drivers/velodyne.

[33] Autodesk. *Getting started with Fusion 360*. 2022. URL: https://help.autodesk.com/view/fusion360/ENU/?guid=GUID-1C665B4D-7BF7-4FDF-98B0-AA7EE12B5AC2 (visited on May 15, 2022).

[34] Aaron Magning. *Fusion 360 Components & Bodies for New Designers*. 2018. URL: https://www.autodesk.com/products/fusion-360/blog/components-bodies-for-new-designers/ (visited on May 15, 2022).

[35] Autodesk. *Assemblies and joints*. 2022. URL: https://help.autodesk.com/view/fusion360/ENU/courses/AP-ASSEMBLIES-AND-JOINTS (visited on May 11, 2022).

[36] Roboy. *SDFusion*. 2020. URL: https://github.com/Roboy/SDFusion (visited on Feb. 21, 2022).

[37] Autodesk. *Integrated CAD, CAM, CAE, and PCB software*. 2022. URL: https://www.autodesk.com/products/fusion-360/overview?term=1-YEAR&tab=subscription (visited on May 10, 2022).

[38] PSN. *CAN-AM OUTLANDER 1000 XT 12 FRAME*. 2022. URL: https://www.powersportsnation.com/can-am-outlander-1000-xt-12-frame-705205310-31737.html.

[39] GeoSLAM. *Point Clouds for Beginners*. 2022. URL: https://geoslam.com/point-clouds/ (visited on May 6, 2022).

[40] GeoSLAM. *What is SLAM (Simultaneous Localization and Mapping)*. 2022. URL: https://geoslam.com/what-is-slam/ (visited on May 4, 2022).

[41] Vedaant Varshney. *LiDAR: The Eyes of an Autonomous Vehicle*. 2019. URL: https://medium.com/swlh/lidar-the-eyes-of-an-autonomous-vehicle-82c6252d1101 (visited on May 6, 2022).

[42] Stetson Doggett. *What Are Point Clouds, And How Are They Used?* 2020. URL: https://www.dronegenuity.com/point-clouds/ (visited on May 6, 2022).

[43] Lieutenant Eric Younkin and Ensign Patrick Debroisse. *Velodyne VLP-16 Laser Scanner Acceptance*. 2016. URL: https://nauticalcharts.noaa.gov/learn/docs/mapping-shoreline-features-with-laser-scanners/velodyne-vlp16-acceptance-final.pdf (visited on May 6, 2022).

[44] Inc. Velodyne LiDAR. *VLP-16 User Manual*. 2019. URL: https://velodynelidar.com/wp-content/uploads/2019/12/63-9243-Rev-E-VLP-16-User-Manual.pdf (visited on May 6, 2022).

[45] Open Robotics. *sensor_msgs/PointCloud2 Message*. 2022. URL: http://docs.ros.org/en/noetic/api/sensor_msgs/html/msg/PointCloud2.html (visited on May 6, 2022).

[46] Shahab Khokhar. *How to Choose the Right LiDAR Sensor for Your Project*. 2020. URL: https://store.clearpathrobotics.com/blogs/blog/how-to-choose-a-LIDAR (visited on May 6, 2022).

[47] Yohei Kajiwara. *VLP16 — C++ Quick Example*. 2018. URL: https://medium.com/@yohei.kajiwara/vlp16-c-quick-example-35b9ceea2059 (visited on May 6, 2022).

[48] Open Robotics. *Laser_scan_matcher*. 2019. URL: http://wiki.ros.org/laser_scan_matcherhttps://gazebosim.org/about (visited on May 10, 2022).

[49] M. Attia and Y. Slama. *Normal Distribution Transform with Point Projection for 3D Point Cloud Registration*. 2017. URL: http://ipco-co.com/PET_Journal/vol_csp_2017/ID-87.pdf (visited on May 10, 2022).

[50] Sunnjiin Cho. Chansoo Kim. Jaehyun Park. Myoungho Sunwoo. Kihun JO. *Semantic Point Cloud Mapping of LiDAR Based on Probabilistic Uncertainty Modeling for Autonomous Driving*. 2020. URL: https://www.mdpi.com/1424-8220/20/20/5900/htm (visited on May 9, 2022).

[51] Martin Magnussn. *The Three-Dimensional Normal-Distributions Transform - Chapter 6*. 2020. URL: http://www.diva-portal.org/smash/get/diva2:276162/FULLTEXT02.pdf (visited on May 9, 2022).

[52] Rapaio. *When we take draws from a normal distribution what are we drawing? - stack exchange*. 2022. URL: https://stats.stackexchange.com/questions/361406/when-we-take-draws-from-a-normal-distribution-what-are-we-drawing (visited on May 10, 2022).

[53] G. Grisetti et al. *A Tutorial on Graph-Based SLAM*. 2010. URL: http://www2.informatik.uni-freiburg.de/~stachnis/pdf/grisetti10titsmag.pdf (visited on May 10, 2022).

[54] Edwin Olson. *Recognizing places using spectrally clustered local matches*. 2009. URL: https://april.eecs.umich.edu/media/pdfs/olson2009ras.pdf (visited on May 10, 2022).

[55] J. Servos and S.L. Waslander. *Multi-Channel Generalized-ICP: A robust framework for multi-channel scan registration*. 2017. URL: https://www.sciencedirect.com/science/article/pii/S0921889016306790 (visited on May 12, 2022).

[56] Rudolph Molero Sebastian Schere Sanjiv Singh and Lyle Chamberlain. *Navigation and Control for Micro Aerial Vehicles in GPS-Denied Environments*. 2022. URL: https://www.researchgate.net/figure/1-The-ICP-algorithm-alighns-two-corresponding-clouds-of-points-Model-and-Scene-by_fig2_266163414 (visited on May 10, 2022).

[57] K. Koide M. Yokozuka S. Oishi and A. Banno. *Voxelized GICP for Fast and Accurate 3D Point Cloud Registration*. 2021. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9560835&tag=1 (visited on May 12, 2022).

[58] Tomáš Báča. *Forest world*. 2021. URL: https://github.com/ctu-mrs/mrs_gazebo_common_resources/blob/master/worlds/forest.world (visited on May 4, 2022).

[59] Docker Inc. *What is a Container?* 2022. URL: https://www.docker.com/resources/what-container/ (visited on Apr. 11, 2022).

[60] Docker Inc. *Getting Started*. 2021. URL: https://docs.docker.com/get-started/ (visited on May 8, 2022).

[61] Docker Inc. *Best practices for writing Dockerfiles*. 2021. URL: https://docs.docker.com/develop/develop-images/dockerfile_best-practices/ (visited on May 8, 2022).

[62] Bobby Borisov. *What is a Docker Container: An Introductory Guide for Beginners*. 2022. URL: https://linuxiac.com/what-is-docker-container/ (visited on May 8, 2022).

[63] Docker Inc. *DockerFile Reference*. 2021. URL: https://docs.docker.com/engine/reference/builder/ (visited on May 8, 2022).

[64] Docker Inc. *Use Multi-Stage Builds*. 2021. URL: https://docs.docker.com/develop/develop-images/multistage-build/ (visited on May 8, 2022).

[65] Docker Inc. *Runtime options with Memory, CPUs, and GPUs*. 2021. URL: https://docs.docker.com/config/containers/resource_constraints/ (visited on May 6, 2022).

[66] NVIDIA Corporation. *NVIDIA Container Runtime - Deploy Containerized GPU Applications*. 2022. URL: https://developer.nvidia.com/nvidia-container-runtime (visited on May 6, 2022).

[67] Microsoft. *Create a development container*. 2022. URL: https://developer.nvidia.com/nvidia-container-runtime (visited on May 6, 2022).

[68] Docker Inc. *Overview of Docker Compose*. 2021. URL: https://docs.docker.com/compose/ (visited on May 8, 2022).

[69] Allison Thackston. *althack/ros2:foxy-gazebo-nvidia*. 2022. URL: https://hub.docker.com/layers/ros2/althack/ros2/foxy-gazebo-nvidia/images/sha256-16d1f63c6a54459dfaabc9920f0532622a2ee50356692f307e739972fab79403?context=explore (visited on Apr. 27, 2022).

[70] NVIDIA Corporation. *Installation Guide*. 2022. URL: https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/install-guide.html#docker (visited on Apr. 3, 2022).

[71] Automatic Addison. *Navigation and SLAM Using the ROS 2 Navigation Stack*. 2021. URL: https://automaticaddison.com/page/15/ (visited on May 3, 2022).

[72] Steve Macenski. *Spatio-Temporal Voxel Layer*. 2022. URL: https://github.com/SteveMacenski/spatio_temporal_voxel_layer (visited on May 3, 2022).

Appendices

Appendix A

Bachelor Assignment

NTNU
Institutt for elektroniske systemer
Institutt for elkraftteknikk
Institutt for teknisk kybernetikk

Oppgaveforslag bacheloroppgave elektroingeniør (BIELEKTRO) i Trondheim, vårsemester 2022

Navn bedrift: Kongsberg Defence & Aerospace	**Kontaktperson: Roger Werner Laug** **Epost: roger.werner.laug@kongsberg.com** **Telefon/mobil: 98220530**	
Tittel på oppgave: ***ROS simulert verden for ATV og finne optimal SLAM algoritme for generert punktsky***		
Hvilke studieretninger passer oppgaven for? (kryss av for alle aktuelle retninger; flervalg er mulig):	**Automatisering og robotikk**	x
	Elektronikk og sensorsystemer	
	Elkraft og bærekraftig energi	
Er oppgaven reservert for noen bestemte studenter? I så fall skriv navnene på studentene til høyre.	Eline Marie Håve, Cecile Nikolaisen, Sigrid Mellemseter	
Er dette en lukket oppgave? Dvs. at sluttrapporten ikke kan publiseres senere fordi den inneholder sensitiv informasjon.	[ ] ja [x] nei [ ] ikke enda bestemt	

Kort beskrivelse av oppgaven med problemstilling.

Lone Wolf er vårt årlige sommerprosjekt for studenter i Kongsberg Defence & Aerospace (KDA), Division Land Systems (DLS) der vi utforsker teknologier for fremtiden. Denne Bacheloroppgaven passer godt inn i utviklingen til sommerprosjektet for 2022, der studentene skal blant annet finne en farbar vei og navigere autonomt med vår ATV.

Fra tidligere er Lone Wolf ATV'en konstruert til å kunne svinge, gire, akselerere og bremse slik at den skal kunne kontrolleres fra en datamaskin. Det er tidligere implementert objekt gjenkjenning som et hjelpemiddel hvis det dukker opp uforutsette hinder i den planlagte ruten. Det er mange utfordringer å ta tak i og for å kunne prøve ut flere metoder og scenarioer ønsker vi å få laget en eller flere fiktive omgivelser for ATV'en i ROS simulator slik at man kan teste autonom navigasjon både i enkle og mer kompliserte omgivelser. For å gjøre dette må det lages en modell av vår ATV. Det er inkludert en del sensorer på ATV'en som LIDAR, radar, IMU, kameraer og GPS, som kan inkluderes og benyttes i simulering for best mulig autonom navigasjon.

Det ønskelig at ATV-modellen skal innhente data i form av en punktsky i ROS simulatoren ved hjelp av LIDAR eller stereo-kamera. Denne punktskyen skal prosesseres av en valgfri SLAM-algoritme og supplert med annen sensor data som for eksempel IMU-data og odometri. Kartet av omgivelsene som SLAM-algoritmen estimerer bør ha tilstrekkelig kvalitet slik at planlegging av farbar vei for autonom navigasjon kan gjennomføres. Det er interessant å finne en passende algoritme basert på kompleksitet og resultat som vi kan videreføre i Lone Wolf prosjektet 2022. En avveining er om kartet bør være i 3D eller 2D supplert med andre hjelpemidler som for eksempel objekt gjenkjenning.

Oppgaven som studentene ønsker å utdype innen dette feltet er relativt åpent, men KDA ønsker å påvirke deler av oppgaven underveis, for eksempel hvordan den simulerte verden skal se ut eller hvor rask algoritmen må være. Dersom det er behov for utstyr skal KDA være behjelpelig med det (begrenset av tilgjengelighet og kostnad). KDA har hatt et møte med studentene og de virker interessert i oppgaven.
https://www.kongsberg.com/no/careers/summer-jobs/lone-wolf/

Appendix B

Measurements of the ATV and 3D models

Part	Measurements (mm)	Remarks
Frame, height	570	
Frame, width (total)	645	The widest part of the frame
Frame, width (tail)	305	The width of the tail of the frame
Frame, ground clearance	295	From the ground to the lowest part of the frame
Frame length	685	Length of the main frame (without the extra parts)
Frame, square tube size	40	The size of the square tube used to build the frame
Gear, diameter	175	
Gear, circumference	550	
LIDAR mount, length	250	Length from the base line to the top of the triangle
LIDAR mount, width	410	Length of the baseline of the triangle
LIDAR mount, height (front)	35	Length of the front pillars the mount is attached to
LIDAR mount, height (back)	10	Length of the back pillar the mount is attached to
LIDAR mount stand, height	109	Height of the cylindrical mounting on the plate
LIDAR mount, thickness	5	Thickness of the plate

Table 1: Physical measurements of the attachments to the ATV

Figure 1: Simple 3D-model of the frame

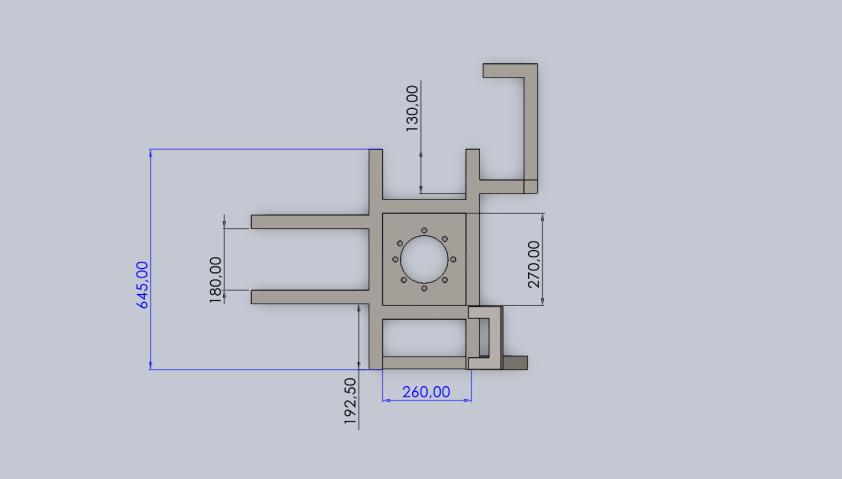


Figure 2: Simple 3D-model of the frame: top view

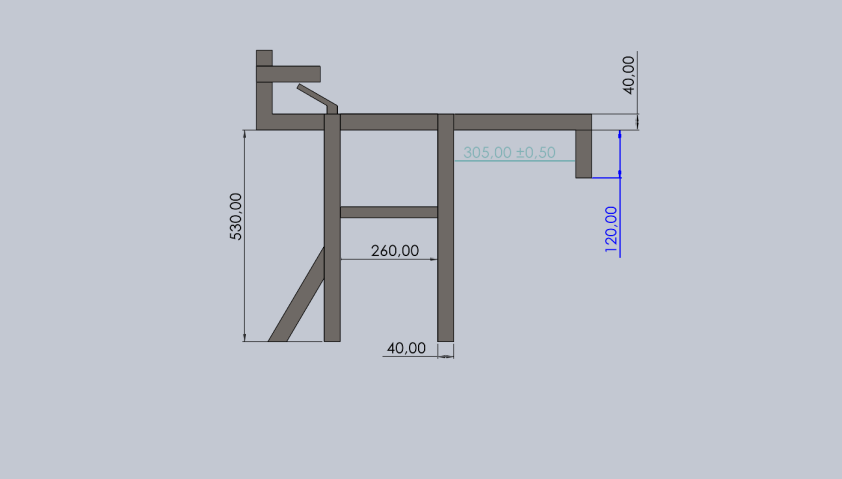


Figure 3: Simple 3D-model of the frame: side view

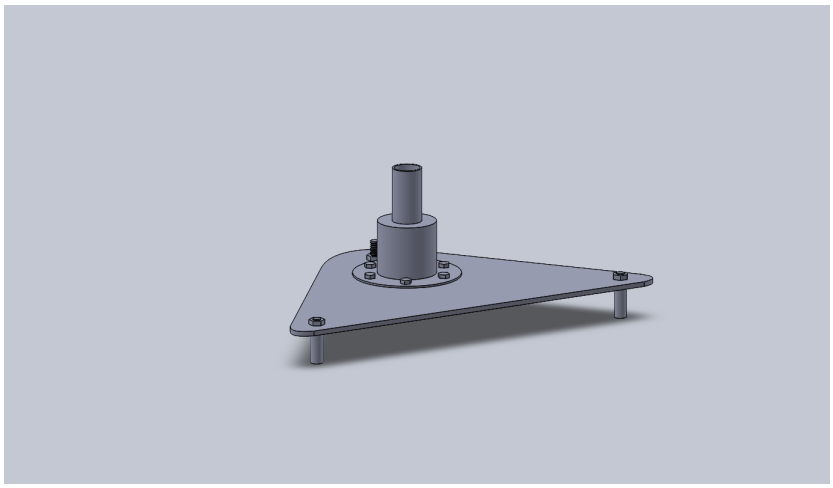


Figure 4: Simple 3D-model of the LIDAR mount

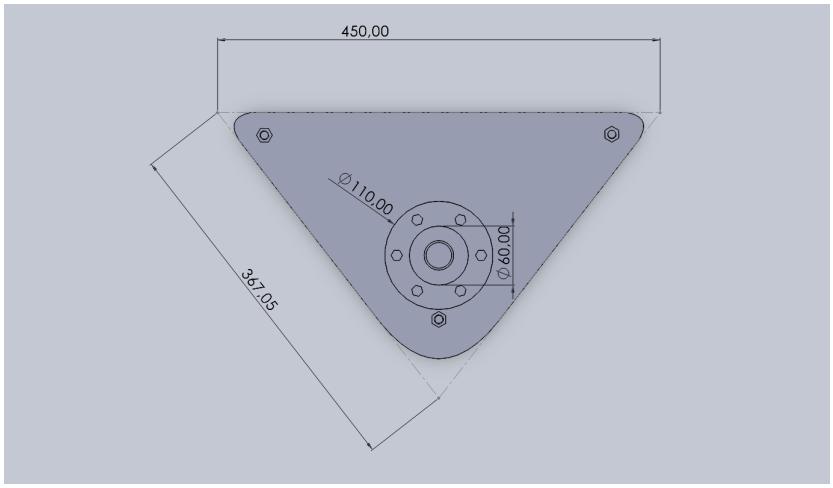


Figure 5: Simple 3D-model of the LIDAR mount, top view

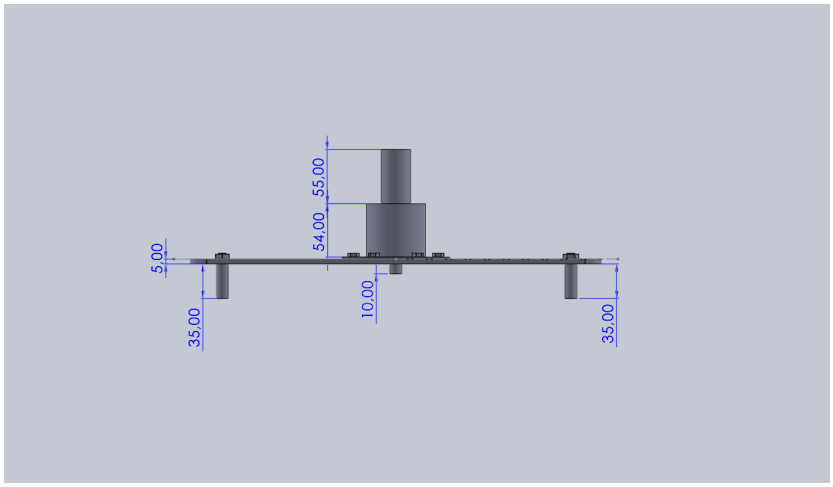


Figure 6: Simple 3D-model of the LIDAR mount, side view

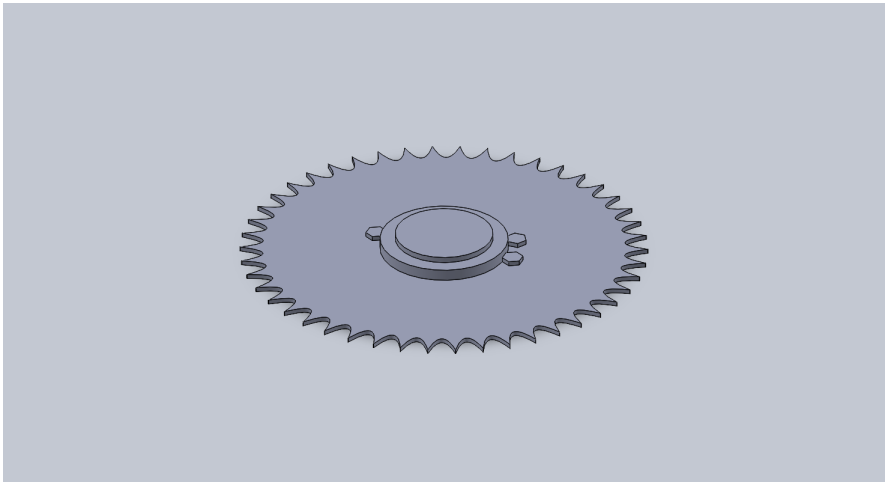


Figure 7: Simple 3D-model of the gear

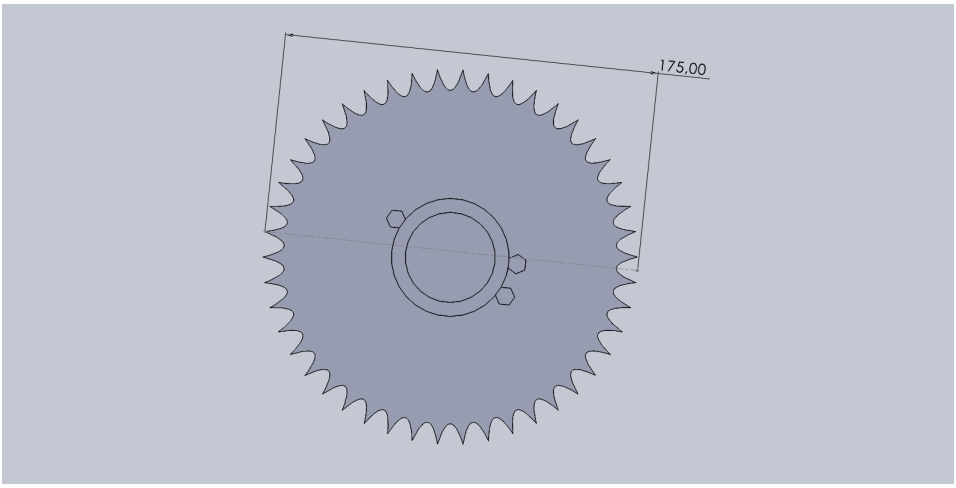


Figure 8: Simple 3D-model of the gear, top view

Appendix C

Data Sheet Can-Am Outlander XT

2018 OUTLANDER™ MAX XT™

HIGHLIGHTS

- Rotax® V-twin engine options
- Continuously Variable Transmission (CVT) with engine braking
- Tri-Mode Dynamic Power Steering (DPS™)
- 3,000-lb (1,361 kg) WARN† winch with roller fairlead
- Heavy-duty front and rear bumpers
- 14-in. cast-aluminum wheels
- 26-in. ITP† Terracross radial tires
- Painted plastics for premium look
- Handlebar wind deflectors
- Convertible Rack System (CRS) with LinQ™ system
- Dynamic passenger comfort ergonomics
- Multiposition passenger handgrips
- Raised floorboards
- 5.7-gal (21.4 L) water-resistant rear compartment
- 625-W magneto

Brushed Aluminum & Can-Am Red / 650 / 850 / 1000R
Intense Red / 650 / 850 / 1000R
Mossy Oak† Break-Up Country Camo† / 650 / 850

ENGINES

ENGINES	650	850	1000R
Type	62 hp, Rotax 649.6 cc V-twin, liquid cooled	78 hp, Rotax 854 cc V-twin, liquid cooled	89 hp, Rotax 976 cc V-twin, liquid cooled
Fuel Delivery System	Electronic Fuel Injection (EFI)		
Transmission	CVT, P / R / N / H / L, standard engine braking		
Drive Train	Selectable 2WD / 4WD with Visco-Lok‡ QE auto-locking front differential		
Power Steering	Tri-Mode Dynamic Power Steering (DPS)		

SUSPENSIONS

Front Suspension	Double A-arm 9 in. (22.9 cm) travel
Front Shocks	Oil
Rear Suspension	Torsional Trailing arm Independent (TTI) 9.3 in. (23.6 cm) travel
Rear Shocks	Oil

BRAKES

Front	Dual 214 mm ventilated disc brakes with hydraulic twin-piston calipers
Rear	Single 214 mm ventilated disc brake with hydraulic twin-piston caliper

TIRES / WHEELS

Front Tires	ITP Terracross 26 x 8 x 14 in. (66 x 20.3 x 35.6 cm)
Rear Tires	ITP Terracross 26 x 10 x 14 in. (66 x 25.4 x 35.6 cm)
Wheels	14-in. cast-aluminum

DIMENSIONS / CAPACITIES

L x W x H	94 x 46 x 53 in. (238.8 x 116.8 x 135 cm)
Wheelbase	59 in. (149.9 cm)
Ground Clearance	11 in. (27.9 cm)
Seat Height	34.5 in. (87.7 cm)
Engine / Dry Weight*	650 / 795 lb (361 kg) 850 / 821 lb (372 kg) 1000R / 840 lb (381 kg)
Rack Capacity	Front: 100 lb (45 kg) Rear: 200 lb (90 kg)
Storage Capacity	Rear: 5.7 gal (21.4 L)
Towing Capacity	1,300 lb (590 kg)
Fuel Capacity	5.4 gal (20.5 L)

FEATURES

Gauge	Multifunction Digital: Speedometer, tachometer (bar graph RPM, bottom bar numerical RPM), odometer, trip & hour meters, diagnostic center, gear position, engine hour meter, 4 x 4 indicator, temperature and engine lights, fuel gauge, clock	Instrumentation	Lighter type DC outlet in console, standard connector in the back (15-A)
		Lighting	230 W from twin 60-W projectors and dual 55-W reflectors with tail light / brake light
		Seat	Convertible Rack System (CRS)
Winch	3,000-lb (1,361 kg) WARN winch with roller fairlead	Anti-theft System	RF Digitally Encoded Security System (D.E.S.S.™)
Protection	Heavy-duty front & rear bumpers, handlebar wind deflectors		

WARRANTY

Factory	6-months limited warranty
Extended	Up to 36 months B.E.S.T. coverage

can-am BRP

Can-AmOffroad.com

Appendix D

Lone Wolf ATV Inventory List

Beskrivelse	Merke	Modell	Ekstra info
RTK GNSS receiver	Emlid	Reach RS+	
HDMI capture card	Magwell	USB Capture HDMI	
RS232 to USB cable			3 stk
USB 3.0 Hub	Sandstrøm		
VectorNav INU	Vector Nav	VN-300-DEV	
LIDAR	Velodyne	80-VLP-16-A	
AC2600 WIFI router	Netgear	Nighthawk X4S	
AC1200 WIFI router	TP-Link	Archer C1200	
Battery charger LIPO/PB	SKYRC	B6AC+ v2	Kan lade alle typer oppladbare batterier. Bruker modulære kabler for tilkobling.
Multimeter			2 stk.
DC/DC 12V/24V omformer		DC1224	35A
Playstation controller	PS	Dualshock 4	
Motorsykkel batteri	Biltema		
Bil batteri	Biltema		
Kobber belagt breadboard			
Coax capture card	Bosch	VIP-X1XF-E	
Video camera	RICOM	5mm 1-2.2 1/1.8"	2 stk.
Relay card		2 rele, 24V DC	
Motorcontroller card	Polulu	0J10613	2 stk.
Raspberry Pi		16 GB minne	2 stk.
Ethernet to USB			
Drone control	Cube	Pixhawk 2	
Camera interface card 3 camera	Leopard Imaging	LI-JTX1-MIPI-ADPT Rev1.2	Works with Jetson TX1
Camera interface card 6 camera	Leopard Imaging	LI-JTX1-MIPI-ADPT 6CAM V1.0	Works with Jetson TX1
Memory card reader USB			
Jetson TX2	Nvidia		
Linear motor controller programming tool		EM-236A Interface Unit	
4G antenna			2 stk.
Digital scale		Hook style 40 KG	
4g modem USB	D-link		
Digital Caliper			

Appendix E

ROS 2 Message Types

Message type	Content	Explenation
geometry_msgs/msg/PoseStamped		Pose with reference coordinate frame and timestamp
Header	header	Frame id and time stamp.
Pose	pose	Pose including position and orientation.
geometry_msgs/TransformStamped[ ]		Expresses a transform from coordinate frame header.frame_id to the coordinate frame child_frame_id
std_msgs/Header	header	Frame id and time stamp.
string	child_frame_id	Frame id of the child frame.
Transform	transform	Translation and rotation of the frame.
geometry_msgs/msg/Twist		Expresses velocity in free space broken into its linear and angular parts.
Vector3	linear	Linear velocity.
Vector3	angular	Angular velocity.
lidarslam_msgs/msg/MapArray		An array of maps
std_msgs/Header	header	Frame id and time stamp.
lidarslam_msgs/SubMap[ ]	submaps	List of submaps.
int8	cloud_coordinate	Local or global cloud_coordinate.
int8	LOCAL=0	Local cloud_coordinate.
int8	GLOBAL=1	Global cloud_coordinate.
lidarslam_msgs/msg/SubMap		An array of submaps
std_msgs/Header	header	Frame id and time stamp.
float64	distance	The distance the robot has travelled.
geometry_msgs/Pose	pose	The robots pose when the data was collected.
sensor_msgs/PointCloud2	cloud	Actual point data, size is (row_step*height).
nav_msgs/msg/Path		An array of poses that represents a Path for a robot to follow
Header	header	Frame id and time stamp.
geometry_msgs/PoseStamped[ ]		Array of poses.
nav_msgs/Odometry		Represents an estimate of a position and velocity in free space
std_msgs/Header	header	Frame id and time stamp.
string	child_frame_id	Frame id the pose points to. The twist is in this coordinate frame.
geometry_msgs/PoseWithCovariance	pose	Estimated pose that is typically relative to a fixed world frame.
geometry_msgs/TwistWithCovariance	twist	Estimated linear and angular velocity relative to child_frame_id.
sensor_msgs/msg/Imu		This is a message to hold data from an IMU (Inertial Measurement Unit)
Header	header	Frame id and time stamp.
geometry_msgs/Quaternion	orientation	Orienation expressed in x,y,z,w.
float64[9]	orientation_covariance	Row major about x, y, z axes.
geometry_msgs/Vector3	angular_velocity	Angular velocity expressed in x, y, z.
float64[9]	angular_velocity_covariance	Row major about x, y, z axes.
geometry_msgs/Vector3	linear_acceleration	Linear acceleration expressed in x, y, z.
float64[9]	linear_acceleration_covariance	Row major x, y z.
sensor_msgs/msg/PointCloud2		Holds a collection of N-dimensional points
Header	header	Time of sensor data acquisition, and the coordinate frame ID.
uint32	height	2D structure of the point cloud.
uint32	width	2D structure of the point cloud.
PointField[ ]	fields	Describes the channels and their layout in the binary data blob.
bool	is_bigendian	Is this data bigendian?
uint32	point_step	Length of a point in bytes.
uint32	row_step	Length of a row in bytes.
uint8[ ]	data	Actual point data, size is (row_step*height).
bool	is_dense	True if there are no invalid points.
tf2_msgs/msg/TFMessage		Array of transforms
geometry_msgs/TransformStamped[ ]	transforms	Array of transforms

Appendix F

Items in the ATV Model SDF

Item	Function	Description
base_link	Link	The base of the robot
rigid_left_front_wheel	Link	Front left wheel
rigid_right_front_wheel	Link	Front right wheel
rigid_left_back_wheel	Link	Back left wheel
rigid_right_back_wheel	Link	Back right wheel
velodyne	Link	Velodyne VLP-16 Lidar
imu	Link	IMU
rev_left_front_wheel	Revolute joint	Joint between left front wheel and base link
rev_right_front_wheel	Revolute joint	Joint between right front wheel and base link
rev_left_back_wheel	Revolute joint	Joint between left back wheel and base link
rev_right_back_wheel	Revolute joint	Joint between right back wheel and base link
lidar_joint	Fixed joint	Joint between LIDAR and base link
imu_joint	Fixed joint	Joint between IMU and base link
veldoyne-VLP16	Sensor	Velodyne VLP-16 Lidar Sensor
my_imu	Sensor	IMU sensor
skid_steer_drive	Plugin	Plugin for steering the robot
gazebo_ros_laser_controller	Plugin	Plugin for Velodyne VLP-16 LIDAR
imu_plugin	Plugin	Plugin for the IMU sensor

Appendix G

SDF of ATV Model

```
<?xml version="1.0" ?>
<sdf version="1.6">
  <model name="atvkda">
    <pose>0 0 0.35 0 -0 0</pose>
  <!-- ******* Base Link *******   -->
    <link name="base_link">
      <self_collide>false</self_collide>
      <pose frame="">-0.5509772858698141 -0.35921503762573787
0.4691281110478071 0.0 -0.0 0.0</pose>
      <inertial>
        <pose frame="">0.0 0.0 0.0 0 0 0</pose>
        <mass>30</mass>
        <inertia>
          <ixx>1</ixx>
          <ixy>1</ixy>
          <ixz>1</ixz>
          <iyy>1</iyy>
          <iyz>1</iyz>
          <izz>1</izz>
        </inertia>
      </inertial>
      <collision name="base_link_collision">
        <geometry>
          <mesh>
            <uri>model://atvkda/meshes/CAD/rigid_body.stl</uri>
            <scale>0.001 0.001 0.001</scale>
          </mesh>
        </geometry>
      </collision>
      <visual name="base_link_visual">
        <geometry>
          <mesh>
            <uri>model://atvkda/meshes/CAD/rigid_body.stl</uri>
            <scale>0.001 0.001 0.001</scale>
          </mesh>
        </geometry>
        <material>
          <ambient>0.1 0.1 0.1 1</ambient>
          <diffuse>0.1 0.1 0.1 1</diffuse>
          <specular>1.5 1.5 1.5 1</specular>
          <emissive>0.05 0.05 0.05 1</emissive>
        </material>
      </visual>
    </link>

  <!-- *******Left Front Wheel *******   -->
    <link name="rigid_left_front_wheel">
      <self_collide>false</self_collide>
      <pose frame="">0.149313570651654 0.10837677345109908
-0.0027835333140371534 0.0 -0.0 0.0</pose>
      <inertial>
        <pose frame="">0.0 0.0 0.0 0 0 0</pose>
        <mass>5</mass>
        <inertia>
          <ixx>0.0125</ixx>
          <ixy>0</ixy>
          <ixz>0.0</ixz>
          <iyy>0.0125</iyy>
          <iyz>0.0</iyz>
```

```
          <izz>0.05</izz>
        </inertia>
      </inertial>
      <collision name="rigid_left_front_wheel_collision">
        <geometry>
          <mesh>
            <uri>model://atvkda/meshes/CAD/rigid_left_front_wheel.stl</uri>
            <scale>0.001 0.001 0.001</scale>
          </mesh>
        </geometry>
        <surface>
          <friction>
            <ode>
              <mu>0.6</mu>
              <mu2></mu2>
              <slip1>0.001</slip1>
              <slip2>0.001</slip2>
            </ode>
          </friction>
          <contact>
            <ode>
              <soft_cfm>0</soft_cfm>
              <soft_erp>0.2</soft_erp>
              <kp>1e+13</kp>
              <kd>1</kd>
              <max_vel>0.01</max_vel>
              <min_depth>0.5</min_depth>
            </ode>
          </contact>
        </surface>
      </collision>
      <visual name="rigid_left_front_wheel_visual">
        <geometry>
          <mesh>
            <uri>model://atvkda/meshes/CAD/rigid_left_front_wheel.stl</uri>
            <scale>0.001 0.001 0.001</scale>
          </mesh>
        </geometry>
        <material>
          <ambient>0.01 0.01 0.01 1</ambient>
          <diffuse>0.1 0.1 0.1 1</diffuse>
          <specular>1.5 1.5 1.5 1</specular>
          <emissive>0.001 0.001 0.001 1</emissive>
        </material>
      </visual>
    </link>

  <!-- ******* Right Front Wheel *******  -->
    <link name="rigid_right_front_wheels">
      <self_collide>false</self_collide>
      <pose frame="">0.14931385014328322 -0.7921332073751959
-0.0027817838858620993 0.0 -0.0 0.0</pose>
      <inertial>
        <pose frame="">0.0 0.0 0.0 0 0 0</pose>
        <mass>5</mass>
        <inertia>
          <ixx>0.0125</ixx>
          <ixy>0</ixy>
          <ixz>0.0</ixz>
          <iyy>0.0125</iyy>
```

```
          <iyz>0.0</iyz>
          <izz>0.05</izz>
        </inertia>
      </inertial>
      <collision name="rigid_right_front_wheels_collision">
        <geometry>
          <mesh>
            <uri>model://atvkda/meshes/CAD/rigid_right_front_wheels.stl</uri>
            <scale>0.001 0.001 0.001</scale>
          </mesh>
        </geometry>
        <surface>
          <friction>
            <ode>
              <mu>0.6</mu>
              <mu2></mu2>
              <slip1>0.001</slip1>
              <slip2>0.001</slip2>
            </ode>
          </friction>
          <contact>
            <ode>
              <soft_cfm>0</soft_cfm>
              <soft_erp>0.2</soft_erp>
              <kp>1e+13</kp>
              <kd>1</kd>
              <max_vel>0.01</max_vel>
              <min_depth>0.5</min_depth>
            </ode>
          </contact>
        </surface>
      </collision>
      <visual name="rigid_right_front_wheels_visual">
        <geometry>
          <mesh>
            <uri>model://atvkda/meshes/CAD/rigid_right_front_wheels.stl</uri>
            <scale>0.001 0.001 0.001</scale>
          </mesh>
        </geometry>
        <material>
          <ambient>0.01 0.01 0.01 1</ambient>
          <diffuse>0.1 0.1 0.1 1</diffuse>
          <specular>1.5 1.5 1.5 1</specular>
          <emissive>0.001 0.001 0.001 1</emissive>
        </material>
      </visual>
    </link>

  <!-- ******* Left Back Wheel *******   -->
    <link name="rigid_left_back_wheel">
      <self_collide>false</self_collide>
      <pose frame="">-1.3496864291871955 0.10837677345106926
-0.0033596523819218326 0.0 -0.0 0.0</pose>
      <inertial>
        <pose frame="">0.0 0.0 0.0 0 0 0</pose>
        <mass>5</mass>
        <inertia>
          <ixx>0.0125</ixx>
          <ixy>0</ixy>
          <ixz>0.0</ixz>
```

```
          <iyy>0.0125</iyy>
          <iyz>0.0</iyz>
          <izz>0.05</izz>
        </inertia>
      </inertial>
      <collision name="rigid_left_back_wheel_collision">
        <geometry>
          <mesh>
            <uri>model://atvkda/meshes/CAD/rigid_left_back_wheel.stl</uri>
            <scale>0.001 0.001 0.001</scale>
          </mesh>
        </geometry>
        <surface>
          <friction>
            <ode>
              <mu>0.6</mu>
              <mu2></mu2>
              <slip1>0.001</slip1>
              <slip2>0.001</slip2>
            </ode>
          </friction>
          <contact>
            <ode>
              <soft_cfm>0</soft_cfm>
              <soft_erp>0.2</soft_erp>
              <kp>1e+13</kp>
              <kd>1</kd>
              <max_vel>0.01</max_vel>
              <min_depth>0.5</min_depth>
            </ode>
          </contact>
        </surface>
      </collision>
      <visual name="rigid_left_back_wheel_visual">
        <geometry>
          <mesh>
            <uri>model://atvkda/meshes/CAD/rigid_left_back_wheel.stl</uri>
            <scale>0.001 0.001 0.001</scale>
          </mesh>
        </geometry>
        <material>
          <ambient>0.01 0.01 0.01 1</ambient>
          <diffuse>0.1 0.1 0.1 1</diffuse>
          <specular>1.5 1.5 1.5 1</specular>
          <emissive>0.001 0.001 0.001 1</emissive>
        </material>
      </visual>
    </link>

  <!-- ******* Right Back Wheel *******   -->
    <link name="rigid_right_back_wheel">
      <self_collide>false</self_collide>
      <pose frame="">-1.3496861498881583 -0.792133207375183
-0.003357902953795265 0.0 -0.0 0.0</pose>
      <inertial>
        <pose frame="">0.0 0.0 0.0 0 0 0</pose>
        <mass>5</mass>
        <inertia>
          <ixx>0.0125</ixx>
          <ixy>0</ixy>
```

```
          <ixz>0.0</ixz>
          <iyy>0.0125</iyy>
          <iyz>0.0</iyz>
          <izz>0.05</izz>
        </inertia>
      </inertial>
      <collision name="rigid_right_back_wheel_collision">
        <geometry>
          <mesh>
            <uri>model://atvkda/meshes/CAD/rigid_right_back_wheel.stl</uri>
            <scale>0.001 0.001 0.001</scale>
          </mesh>
        </geometry>
        <surface>
          <friction>
            <ode>
              <mu>0.6</mu>
              <mu2></mu2>
              <slip1>0.001</slip1>
              <slip2>0.001</slip2>
            </ode>
          </friction>
          <contact>
            <ode>
              <soft_cfm>0</soft_cfm>
              <soft_erp>0.2</soft_erp>
              <kp>1e+13</kp>
              <kd>1</kd>
              <max_vel>0.01</max_vel>
              <min_depth>0.5</min_depth>
            </ode>
          </contact>
        </surface>
      </collision>
      <visual name="rigid_right_back_wheel_visual">
        <geometry>
          <mesh>
            <uri>model://atvkda/meshes/CAD/rigid_right_back_wheel.stl</uri>
            <scale>0.001 0.001 0.001</scale>
          </mesh>
        </geometry>
        <material>
          <ambient>0.01 0.01 0.01 1</ambient>
          <diffuse>0.1 0.1 0.1 1</diffuse>
          <specular>1.5 1.5 1.5 1</specular>
          <emissive>0.001 0.001 0.001 1</emissive>
        </material>
      </visual>
    </link>

  <!-- ******* LIDAR *******   -->
    <link name='velodyne'>
      <pose frame=''>0.235 -0.34 0.85933 0 0 0</pose>
      <inertial>
        <pose frame='map'>0 0 0 0 0 0</pose>
        <mass>0.2</mass>
        <inertia>
          <ixx>0.01</ixx>
          <ixy>0.01</ixy>
          <ixz>0.01</ixz>
```

```
          <iyy>0.01</iyy>
          <iyz>-0.01</iyz>
          <izz>0.01</izz>
        </inertia>
      </inertial>
      <collision name='base_footprint_collision_1'>
        <pose frame=''>0 0 0 0 0 0</pose>
        <geometry>
          <cylinder>
            <length>0.0717</length>
            <radius>0.0516</radius>
          </cylinder>
        </geometry>
        <surface>
          <contact>
          <ode/>
          </contact>
          <friction>
          <ode/>
          <torsional>
            <ode/>
          </torsional>
          </friction>
          <bounce/>
        </surface>
        <max_contacts>10</max_contacts>
      </collision>
      <visual name='base_footprint_visual_1'>
        <pose frame=''>0 0 0 0 0 0</pose>
        <geometry>
          <mesh>
          <scale>1 1 1</scale>
          <uri>model://velodyne_VLP16/meshes/VLP16_base_1.dae</uri>
          </mesh>
        </geometry>
        <material>
          <script>
          <uri>__default__</uri>
          <name>__default__</name>
          </script>
        </material>
      </visual>
      <visual name='base_footprint_visual_2'>
        <pose frame=''>0 0 0 0 0 0</pose>
        <geometry>
          <mesh>
            <scale>1 1 1</scale>
            <uri>model://velodyne_VLP16/meshes/VLP16_base_2.dae</uri>
          </mesh>
        </geometry>
        <material>
          <script>
            <uri>__default__</uri>
            <name>__default__</name>
          </script>
        </material>
      </visual>
      <visual name='base_footprint_visual_3'>
        <pose frame=''>0 0 0 0 0 0</pose>
        <geometry>
```

```
355           <mesh>
356             <scale>1 1 1</scale>
357             <uri>model://velodyne_VLP16/meshes/VLP16_scan.dae</uri>
358           </mesh>
359         </geometry>
360         <material>
361           <script>
362             <uri>__default__</uri>
363             <name>__default__</name>
364           </script>
365         </material>
366       </visual>
367       <sensor name='velodyne-VLP16' type='ray'>
368         <visualize>0</visualize>
369         <update_rate>10</update_rate>
370         <ray>
371           <scan>
372             <horizontal>
373               <samples>170</samples>
374               <resolution>1</resolution>
375               <min_angle>-3.14159</min_angle>
376               <max_angle>3.14159</max_angle>
377             </horizontal>
378             <vertical>
379               <samples>16</samples>
380               <resolution>1</resolution>
381               <min_angle>-0.261799</min_angle>
382               <max_angle>0.261799</max_angle>
383             </vertical>
384           </scan>
385           <range>
386             <min>0.5</min>
387             <max>50</max>
388             <resolution>0.001</resolution>
389           </range>
390           <noise>
391             <type>gaussian</type>
392             <mean>0</mean>
393             <stddev>0</stddev>
394           </noise>
395         </ray>
396         <plugin name='gazebo_ros_laser_controller'
    filename='libgazebo_ros_velodyne_laser.so'>
397           <ros>
398             <namespace></namespace>
399             <argument>~/out:=points_raw</argument>
400           </ros>
401           <frame_name>velodyne</frame_name>
402           <min_range>0.5</min_range>
403           <max_range>50</max_range>
404           <gaussian_noise>0.008</gaussian_noise>
405           <robotNamespace></robotNamespace>
406           <organize_cloud>false</organize_cloud>
407
408           <output_type>sensor_msgs/PointCloud2</output_type>
409         </plugin>
410         <pose frame=''>0 0 0 0 0 0</pose>
411       </sensor>
412       <gravity>0</gravity>
413       <self_collide>0</self_collide>
```

```
      <kinematic>0</kinematic>
    </link>

  <!-- ******* IMU *******  -->
    <link name="imu">
      <pose frame=''>-0.55 -0.34 0.465 0 0 0</pose>
      <inertial>
        <mass>0.1</mass>
        <inertia>
          <ixx>0.01</ixx>
          <ixy>0.01</ixy>
          <ixz>0.01</ixz>
          <iyy>0.01</iyy>
          <iyz>-0.01</iyz>
          <izz>0.01</izz>
        </inertia>
      </inertial>
      <collision name="imu_collision">
        <pose frame=''>0 0 0 0 0 0 </pose>
        <geometry>
          <box>
            <size> 0.05 0.05 0.01 </size>
          </box>
        </geometry>
      </collision>
      <visual name="imu_visual">
        <geometry>
          <box>
            <size> 0.05 0.05 0.01 </size>
          </box>
        </geometry>
      </visual>

      <sensor name="my_imu" type="imu">
        <imu>
        <angular_velocity>
        <x>
          <noise type="gaussian">
          <mean>0.0</mean>
          <stddev>2e-4</stddev>
          <bias_mean>0.0000075</bias_mean>
          <bias_stddev>0.0000008</bias_stddev>
          </noise>
        </x>
        <y>
          <noise type="gaussian">
          <mean>0.0</mean>
          <stddev>2e-4</stddev>
          <bias_mean>0.0000075</bias_mean>
          <bias_stddev>0.0000008</bias_stddev>
          </noise>
        </y>
        <z>
          <noise type="gaussian">
          <mean>0.0</mean>
          <stddev>2e-4</stddev>
          <bias_mean>0.0000075</bias_mean>
          <bias_stddev>0.0000008</bias_stddev>
          </noise>
        </z>
```

```
        </angular_velocity>
        <linear_acceleration>
        <x>
          <noise type="gaussian">
          <mean>0.0</mean>
          <stddev>1.7e-2</stddev>
          <bias_mean>0.00001</bias_mean>
          <bias_stddev>0.00001</bias_stddev>
          </noise>
        </x>
        <y>
          <noise type="gaussian">
          <mean>0.0</mean>
          <stddev>1.7e-2</stddev>
          <bias_mean>0.00001</bias_mean>
          <bias_stddev>0.00001</bias_stddev>
          </noise>
        </y>
        <z>
          <noise type="gaussian">
          <mean>0.0</mean>
          <stddev>1.7e-2</stddev>
          <bias_mean>0.00001</bias_mean>
          <bias_stddev>0.00001</bias_stddev>
          </noise>
        </z>
        </linear_acceleration>
        </imu>
        <always_on>true</always_on>
        <update_rate>30</update_rate>
        <plugin name="imu_plugin" filename="libgazebo_ros_imu_sensor.so">
          <ros>
            <namespace></namespace>
            <argument>~/out:=imu_correct</argument>
          </ros>
          <frame_name>map</frame_name>
        </plugin>
      </sensor>
      <gravity>0</gravity>
      <self_collide>0</self_collide>
      <kinematic>0</kinematic>
    </link>

  <!-- *******Joints *******  -->
    <joint name="imu_joint" type="fixed">
          <parent>base_link</parent>
          <child>imu</child>
          <pose>>0 0 0 0 0 0</pose>
      </joint>

    <joint name="lidar_joint" type="fixed">
          <parent>base_link</parent>
          <child>velodyne</child>
          <pose>>0 0 0 0 0 0</pose>
      </joint>

    <joint name="rev_right_back_wheel" type="revolute">
      <parent>base_link</parent>
      <child>rigid_right_back_wheel</child>
      <pose frame="">0 0 0 0 0 0</pose>
```

```
      <axis>
        <xyz>0 1 0</xyz>
        <use_parent_model_frame>0</use_parent_model_frame>
      </axis>
    </joint>
    <joint name="rev_left_back_wheel" type="revolute">
      <parent>base_link</parent>
      <child>rigid_left_back_wheel</child>
      <pose frame="">0 0 0 0 0 0</pose>
      <axis>
        <xyz>0 1 0</xyz>
        <use_parent_model_frame>0</use_parent_model_frame>
      </axis>
    </joint>
    <joint name="rev_left_front_wheel" type="revolute">
      <parent>base_link</parent>
      <child>rigid_left_front_wheel</child>
      <pose frame="">0 0 0 0 0 0</pose>
      <axis>
        <xyz>0 1 0</xyz>
        <use_parent_model_frame>0</use_parent_model_frame>
      </axis>
    </joint>
    <joint name="rev_right_front_wheel" type="revolute">
      <parent>base_link</parent>
      <child>rigid_right_front_wheels</child>
      <pose frame="">0 0 0 0 0 0</pose>
      <axis>
        <xyz>0 1 0</xyz>
        <use_parent_model_frame>0</use_parent_model_frame>
      </axis>
    </joint>


  <!-- ******* Plugin for steering the ATV *******  -->

    <plugin name='skid_steer_drive' filename='libgazebo_ros_diff_drive.so'>
      <ros>
        <namespace>  </namespace>
      </ros>

      <update_rate>30</update_rate>

      <num_wheel_pairs> 2 </num_wheel_pairs>

      <left_joint>rev_left_front_wheel</left_joint>
      <right_joint>rev_right_front_wheel</right_joint>
      <wheel_separation>0.914</wheel_separation>
      <wheel_diameter>0.694622</wheel_diameter>

      <left_joint>rev_left_back_wheel</left_joint>
      <right_joint>rev_right_back_wheel</right_joint>
      <wheel_separation>0.914</wheel_separation>
      <wheel_diameter>0.694622</wheel_diameter>

      <max_wheel_torque>100</max_wheel_torque>
      <max_wheel_acceleration>100</max_wheel_acceleration>

      <publish_odom>false</publish_odom>
      <publish_odom_tf>true</publish_odom_tf>
```

```
      <publish_wheel_tf>true</publish_wheel_tf>

      <odometry_topic>odom</odometry_topic>
      <odometry_frame>odom</odometry_frame>
      <robot_base_frame>base_link</robot_base_frame>
    </plugin>

  </model>
</sdf>

```

Appendix H

User Manual

H.1 Introduction

This user manual provides a detailed description of how to launch and use the simulator of the Lone Wolf ATV. ROS 2 Foxy with Ubuntu 20.04 and required packages specified on the equipment section should be installed.

H.2 Equipment

The applications and libraries needed to build and launch the simulator is described in table H.1 below.

Name	Description
Ceres-Solver	C++ library for modelling and optimisation
Gazebo	3D robotics simulation software
GTSAM	C++ sensor fusion library
G2O	Pose graph optimisation library
PC with Ubuntu 20.04 or newer	Necessary for running ROS 2
RViz	Robotics visualisation software
ROS Foxy full desktop	ROS 2 full distro
ros-foxy-gazebo-pkgs	ROS simulation in Gazebo
ros-foxy-ros2-control	Controls framework for ROS 2
ros-foxy-ros2-controllers	Controls framework for ROS 2
ros-foxy-teleop-twist-keyboard	Teleoperation twis keyboard
ros-foxy-velodyne	Velodyne VLP-16 package

Table H.1: Equipment for running the simulator

H.3 Create a ROS 2 Workspace

Create a ROS 2 workspace and clone the repository by typing these commands in the terminal line.

```
$ source opt/ros/foxy/setup.bash
$ mkdir ~/ros2_ws/src
$ cd  ~/ros2_ws/src
$ git clone git@github.com:sigridmellemseter/lonewolf.git
$ cd ~/ros2_ws
$ colcon build
```

Copy the models from the folder *gazemodels* and place them in */.gazebo/models/*. This can be done by writing the following in the terminal.

```
$ cp -r lonewolf/gazebomodels/. ~/.gazebo/models/
```

H.4 Launching the Simulations

Open a terminal and write the following.

```
$ source opt/ros/foxy/setup.bash
$ cd ~/ros2_ws
$ source install/setup.bash
$ ros2 launch atv_pkg texasworld.launch.py
```

You should now see the ATV and the world in Gazebo.

Steering the ATV

To steer the ATV, these commands have to be run in the terminal.

```
$ source opt/ros/foxy/setup.bash
$ ros2 run teleop_twist_keyboard teleop_twist_keyboard
```

The Teleoperation Twist Keyboard takes in the arguments in table H.2 for steering the ATV in Gazebo.

Argument	Action
u	Turn left while driving forward
i	Drive straight ahead
o	Turn right while driving forward
j	Turn left (in place)
k	Stop
l	Turn right (in place)
m	Turn left while backing up
,	Back up (straight)
.	Turn right while backing up
q/z	Increase/decrease max speeds by 10%
w/x	Increase/decrease only linear speed by 10%
e/c	Increase/decrease only angular speed by 10%

Table H.2: Teleoperation Twist Keyboard Arguments

How to View the Point Cloud in RViz2

While the simulator is running, open a new terminal window and run rviz2.

```
$ source opt/ros/foxy/setup.bash
$ rviz2
```

In rviz2 you have to define the correct frame. Write "velodyne" as the Fixed Frame under Global Options. Now you have to add the PointCloud2 in RViz.

- Click "Add" in the bottom left corner
- Choose "By Topic"
- Choose "PointCloud2" under /point_raq

Now you should see the point cloud in rviz. To get a better visual choose the Style "Flat Squares", Size (m) to 0.03 and set Color Transformer to "AxisColor"

Launching SLAM

While the simulator is running, open a new terminal and write the following.

```
$ source opt/ros/foxy/setup.bash
$ cd ~/ros2_ws
$ source install/setup.bash
$ ros2 launch scanmatcher lio_bigloop.launch.py
```

You can now view the generated maps in RViz2:

- Open RViz2 as described above
- Set the fixed frame to "map"
- Add the topics "/map", "/path", "/modified_map" and "/modified_path"

To see the data more clearly, you can choose the following styles:

- /map: Flat Squares, Size (m) 0.03, Color Transformer to AxisColor
- /path: Color White
- /modified_map: Flat Squares, Size (m) 0.03, Color Transformer to FlatColor
- /modified_path: Color green

H.5 Open ROS 2 in Container

If you don't have a PC with operating system Ubuntu 20.04 or newer, you can set up development from a Docker container. If you are new to Ubuntu/Linux, or don't already have any other specific plan on how you want to set up development with docker, we recommend you set up the Docker container this way.

Firstly, you need to install a Docker version that is compatible with your operating system. If you are using Ubuntu, remember to add Docker to the admin group so you don't have to add "sudo" before every Docker command. After adding docker to the admin group, remember to reboot for the change to take effect.

The second step is to clone the repository and open it in VSCode with these commands:

```
$ git clone git@github.com:sigridmellemseter/lonewolf.git
$ code lonewolf
```

In VSCode, download the extension "Remote-Containers". To open the container in VSCode press CTRL+SHIFT+P and search for Remote-Containers: Reopen in Container. The image should now start building. This might take a while. After the build is complete, you can start to develop the simulator as if ROS 2 Foxy is installed on Ubuntu 20.04.

Appendix I

Poster

NTNU
Norwegian University of Science and Technology

ROS Simulated World For ATV With SLAM

Department of Engineering Cybernetics

Eline Marie Håve *Sigrid Mellemseter* *Cecilie Nikolaisen*

KONGSBERG

Abstract

This poster summarises the bachelor assignment concerning a ROS simulated world for an autonomous ATV. The ATV is modelled in CAD software, and implemented in a Gazebo world. The ATV gathers data for a point cloud, which is used to generate a map and path with SLAM algorithms.

Simulator

The simulator in Gazebo is a test platform for SLAM algorithms. It consists of two worlds with different complexity and an ATV robot. The ATV is steerable through keyboard commands. The model of the ATV is a fair representation of the Lone Wolf ATV.

Model

The ATV model consists of a base link with joints forking out to the four wheels. It was modelled by following a strict naming and joint definition structure to allow exportation to simulation descriptive format (SDF). This format is capable of running within a simulated world in the Gazebo physics engine. The SDF includes the sensors Velodyne VLP-16 LiDAR and IMU.

Figure 1:ATV Model

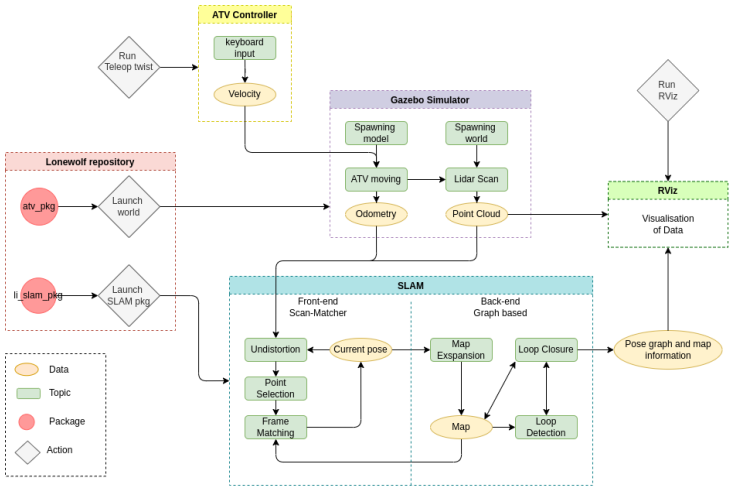


Figure 2:System Architecture

ROS

ROS 2 was chosen as the framework for the simulator, and different ROS packages and libraries are implemented. An overview of the ROS architecture is visualised in figure 2.

Point Cloud

A point cloud is generated from the simulated worlds. The simulated LiDAR represents the physical sensor well. Finding sufficient values for the point cloud contributes to better mapping when the SLAM algorithm is implemented.

Specification	Value
Horizontal samples	170
Vertical samples	16
Minimum range	0.5
Maximum range	50

Table 1:Sufficient Values for the Point Cloud

SLAM

The *li_slam_ros2* package is implemented with Scan-Matcher and Graph-Based SLAM as the sensor-processing and pose-graph optimisation. Different algorithms for the back-end processing are tested, which results in NDT being used for front-end while GICP for back-end. The map, path, and modified data is published on topics, which can be visualised in RViz.

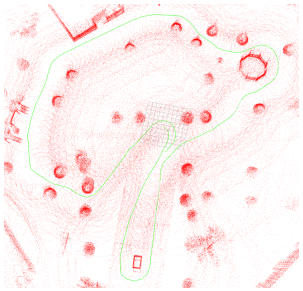


Figure 3:Mapping and Localisation

Conclusion & Future Work

The system design works as intended, with a clean and functioning file structure, meaning that the following key conclusions can be drawn:

- All parts of the ATV are sufficiently modelled and successfully connected. Some adjustments are needed for a correct representation of joints and physics.
- The ATV navigates in the simulator through keyboard commands. The entire simulator is operational in ROS 2 and available for further development.
- The simulated LiDAR seems to simulate the physical sensor well, and captures the environment around the ATV.
- The SLAM package is fully operational and able to generate a map using the data from the point cloud. The point cloud data is supplemented with IMU data to achieve sufficient map quality.

A few suggestions for future development of the project:

- A more realistic model of the ATV and it's test environment.
- Implementation of the NAV2 stack for autonomous navigation.
- Setting up communication within the physical ATV and the simulator.

References

- Eline Marie Håve, Sigrid Mellemseter and Cecilie Nikolaisen. *ROS Simulated World for ATV With SLAM* 2022
- Eline Marie Håve, Sigrid Mellemseter and Cecilie Nikolaisen. *Lonewolf Repository* `https://github.com/sigridmellemseter/lonewolf` 2022

Håve, Mellemseter and Nikolaisen

ROS Simulated World for ATV with SLAM

NTNU
Kunnskap for en bedre verden

KONGSBERG