Henrik B. Norås Bergel

Simultaneous Localization and Mapping Applied to an Unmanned Ground Vehicle

Design and Realization

Master's thesis in Cybernetics and Robotics
Supervisor: Jan Tommy Gravdahl
December 2020

Master's thesis

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

Henrik B. Norås Bergel

Simultaneous Localization and Mapping Applied to an Unmanned Ground Vehicle

Design and Realization

Master's thesis in Cybernetics and Robotics
Supervisor: Jan Tommy Gravdahl
December 2020

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

NTNU
Norges teknisk-naturvitenskapelige universitet

Fakultet for informasjonsteknologi, matematikk og elektroteknikk
Institutt for teknisk kybernetikk

MSc thesis assignment

Name of the candidate:	Henrik B. Norås Bergel
Subject:	Engineering Cybernetics
Title:	Simultaneous Localization and Mapping applied to an Unmanned Ground Vehicle: design and realization

Background:

This assignment is concerned with the design and building of a UGV and its use as a testbed for SLAM

Tasks:

1. Review necessary literature within the field of SLAM.
2. Design and build a UGV platform. The platform should be able to collect datasets based on IMU, encoders and lidar.
3. Propose a SLAM system based on GMapping. The SLAM system should account for a terrain like scenery and utilize the sensor data to produce a map for autonomous navigation.
4. Calibrate the odometry of the UGV and acquire two datasets: a benchmark dataset and a dataset from a terrain like scenery.
5. Test the SLAM system on the datasets and tune the algorithm for optimal performance.
6. Evaluate the performance of the SLAM system by comparing the two resulting maps estimated on the two different datasets and including the properties of the algorithm in the evaluation.
7. Investigate how the 2D output map estimates a slope.

To be handed in by: 20/12-2020
Co-supervisor: Joseph Piperakis. Kongsberg Defence & Aerospace

[illegible]

Jan Tommy Gravdahl
Professor, supervisor

Abstract

This Master's thesis presents the work of designing and building the Unmanned Ground Vehicle (UGV) platform, Phylax. It also investigates the possibility of using GMapping as the method in a Simultaneous Localization and Mapping (SLAM) system that estimates maps based on sensor data from Phylax. The existing SLAM methods all have different trade-offs in terms of complexity, cost and accuracy. GMapping has previously shown promising results in cost and accuracy in SLAM systems exploiting indoor environments. The contribution to the development of autonomous capabilities of a UGV by utilizing SLAM was the motivational factor.

The contribution of this thesis is the Phylax platform and an investigation of the performance of a developed SLAM system utilizing the SLAM method, GMapping. Phylax can both be controlled externally by a user, or operate fully autonomously. The UGV will acquire necessary Inertial Measurement Unit (IMU)-, odometry- and lidar data, which the SLAM system will use to estimate a map of the surrounding scenery. The software for Phylax was implemented in the framework Robot Operating System (ROS), using the programming languages C++ and Extensible Markup Language (XML).

The SLAM system was tested on two datasets that Phylax collected in an indoor environment. The first dataset was from a clean room only containing walls, and worked as the ground truth. The second dataset was acquired in an obstacle course built inside the same room. Prior to the experiment, the odometry in Phylax was calibrated by comparing the values in the ROS odometry topic with the actual traveled distance.

The results from the SLAM system are unambiguous. GMapping estimated accurate maps on the benchmark dataset. However, the maps from the obstacle course presented inadequate results. Relevant information was left out of the map, which made autonomous navigation, in that scenery, difficult. Phylax worked in a satisfactory way and the data acquisition was successful. However, there are possible errors in the middle motors that might affect the odometry calculations. Also, the transformation between coordinate frames may include inaccurate measurements for the translation. These errors could propagate into the system and cause GMapping to estimate a map based on erroneous data.

The thesis concluded that the UGV platform was successfully designed and built with the ability to collect datasets based upon the sensors: IMU, encoders and lidar. The SLAM system was also properly developed, but the use of GMapping as the SLAM method contributed to a low score on the level of autonomy (LOA) scale. Therefore, GMapping was concluded not to be suited as a SLAM method for autonomous navigation in terrain for a UGV.

Sammendrag

Denne oppgaven presenterer arbeidet med å designe og bygge det Ubemannede Kjøretøyet (UK), Phylax. Oppgaven undersøker også muligheten for å bruke GMapping som metode i et Simultan Lokalisering og Kartlegging's (SLOK) system. De eksisterende SLOK metodene har forskjellige kompromisser når det gjelder kompleksitet, kostnad og nøyaktighet. GMapping har tidligere vist lovende resultater i kostnad og nøyaktighet ved bruk i SLOK systemer som utforsker innendørs omgivelser. Å bidra i utviklingen av den autonome evnen til en UK ved å utnytte SLOK var en motiverende faktor for arbeidet.

Bidraget til denne oppgaven er Phylax-plattformen, samt en undersøkelse av ytelsen til et utviklet SLOK system som drar nytte av SLOK metoden GMapping. Phylax kan både bli kontrollert eksternt av en bruker, eller operere fullstendig autonomt. UK vil anskaffe nødvendig Inertiell Målingsenhets (IM) data, odometridata og lidardata som SLOK systemet kan bruke til å estimere et kart av omgivelsene med. Programvaren til Phylax ble implementert i rammeverket Robot Operativ System (ROS), ved bruk av programmeringsspråkene C++ og XML.

SLOK systemet ble testet på to datasett som Phylax samlet inn innendørs. Det første datasettet var basert på et rom kun bestående av vegger og et flatt gulv. Dette datasettet ble fungerende som et referansesett. Det andre datasettet ble samlet inn i det samme rommet, men nå også med en hinderløype. Før eksperimentet ble gjennomført, ble odometrien til Phylax kalibrert ved a sammenligne verdiene i et ROS emne med den faktiske kjørte distansen.

Resultatene fra SLOK systemet er entydige. GMapping estimerer nøyaktige kart basert på referansedatasettet. Kartene basert på datasettet fra hinderløypen gav imidlertid utilstrekkelige resultater. Relevant informasjon ble utelatt fra kartet, noe som gjorde autonom navigasjon i det gitte miljøet vanskelig. Phylax fungerte utmerket og datainnslamlingen var vellykket. Imidlertid er det mulige feil i de midterste motorene som kan påvirke odometriutregningene. Transformasjonene mellom koordinatsystemene kan også inneholde unøyaktige målinger av distansen mellom systemene. Disse feilene kan spre seg videre inn i systemet og føre til at GMapping estimerer kart basert på feil data.

Oppgaven konkluderte med at UK-plattformen ble designet og bygget med et suksessfullt resultat med mulighet for å samle inn data basert på sensorene: IM, odometri og lidar. SLOK systemet ble også riktig utviklet, men bruken av GMapping som SLOK metode bidro til en lav score på Autonomi Skalaen (AS). Derfor ble GMapping sett på som en utilstrekkelig SLOK metode for autonom navigering i terreng for en UK.

Preface

This thesis is the resulting report describing the work in TTK4900 Engineering Cybernetics - Master's Thesis and presents the final work of the Master of Science in Cybernetics and Robotics. The project has been in cooperation with Kongsberg Defence & Aerospace (KDA), which provided all the hardware components for the project. However, it is necessary to specify that the assembling of the components, the design of the platform and the software implementation for this project has been conducted by the author in cooperation with the supervisor from KDA. The work took place in the period from August 2020 to December 2020, and is based upon the project thesis from Spring 2020 by (Bergel (2020)). A visual representation of the resulting robot can be seen in the accompanying video.

Acknowledgments

I would like to thank my main supervisor Professor Jan Tommy Gravdahl from the Department of Engineering Cybernetics at NTNU for his assistance and support throughout this project. He made sure I was on track at all time and provided invaluable feedback for the project.

The work conducted in this thesis has been a cooperation between me and Joseph Piperakis. Mr. Piperakis has been an excellent supervisor and partner throughout this project. This project would not have been possible with out him. For this, I thank him a lot.

I would also like to thank Kongsberg Defence & Aerospace for providing any hardware necessary and an office place. The great environment gave me inspiration for learning and hard work.

Lastly, but not least, I would like to humbly thank my family and friends for unconditional support during my time at the Norwegian University of Science and Technology.

[illegible]

Henrik B. Norås Bergel, December 11, 2020

Contents

List of Tables

List of Figures

Abbreviations

UGV	=	Unmanned Ground Vehicle
SLAM	=	Simultaneous Localization and Mapping
IMU	=	Inertial Measurement Unit
ROS	=	Robot Operating System
XML	=	Extensible Markup Language
LOA	=	Levels of Autonomy
KDA	=	Kongsberg Defence & Aerospace
GPS	=	Global Positioning System
ALFUS	=	Autonomy Levels for Unmanned Systems
ESI	=	External System Independence
MC	=	Mission Complexity
EC	=	Environmental Complexity
EKF	=	Extended Kalman Filter
PF	=	Particle Filter
CPU	=	Central Processing Unit
RBPF	=	Rao-Blackwellized Particle Filter
FOV	=	Field of View
SIR	=	Sampling Importance Resampling
WD	=	Wheel Drive
2S	=	Two Cell
DC	=	Direct Current
CPR	=	Counts Per Revolution
USB	=	Universal Serial Bus
DoF	=	Degrees of Freedom
DMP	=	Digital Motion Processor
WIFI	=	Wireless Fidelity
I^2C	=	Inter-Integrated Circuit
HDMI	=	High Definition Multimedia Interface
4S	=	Four Cell
URDF	=	Unified Robot Description Format
PID	=	Proportional-Integral-Derivative
TCP	=	Transmission Control Protocol
SSH	=	Secure Shell
IP	=	Internet Protocol

Chapter 1

Introduction

1.1 Background

The use of robotic autonomous systems in the military has increased dramatically in modern time. Taking up more and more of the job for the human soldier, autonomous military robotics may help decreasing war casualties and the economic cost but also increase the effectiveness of the military operations. As Lin et al. (2008) writes, the distinct advantage that autonomous robots have on us, the *Homo Sapiens*, is the great cognitive capability. Being able to process a conflict picture that is getting both bigger and faster has been an important attribute for military to have.

In the military industry, autonomous UGV is starting to be an extensively used technology. The UGV works as a extension and replacement for the human soldier as it makes it possible to carry out highly dangerous missions without risking the soldiers life. The UGV will often face unknown environments and in order to be autonomous it needs to gather and process information about its surroundings. Another example is the potential of using autonomous UGVs for post-war demining (Forsvarets-forskningsinstitutt). This is a highly dangerous job where autonomous UGVs could extinguish this risk. However, a UGV has its limitations. Both the environment and situation in which the vehicle has to negotiate might be extremely complex. Huge datasets from the vehicle's sensors need to be processed in order to plan a path in the terrain the vehicle is facing.

Introducing highly advanced mapping and localization capabilities to the UGV could contribute to solve the addressed limits above. Combining appropriate hardware and software is necessary for the autonomous vehicles to perform well. One need to have well functioning algorithms as well as powerful computing system to perform complex perception functions in real time to have autonomous driving (Shi et al. (2017)). Implementing these autonomous features could increase the probability of mission success. An important waypoint on the path to autonomous

UGV operating in terrain is generating highly accurate maps of the environment.

1.2 Motivation

The motivation for this Master's thesis is to contribute to the development of the autonomous capabilities of a UGV by utilizing SLAM. There are numerous different SLAM methods that localize the agent and map the surrounding scenery. This map can be used when planning a path for autonomous driving. Each of these methods have different trade-offs in terms of complexity, accuracy of the map and loss of relevant information. Different SLAM methods estimates either 2D or 3D maps, where 3D maps includes more information of the environment but at the expense of computational time.

In modern time, several cars have the ability to drive autonomously on roads in civilized areas. Lidar, radar or computer vision are used for sensing the local environment. Global Positioning System (GPS) localizes the car and provides roadway maps (Jo et al. (2014)), but the GPS signals may not always be available and secure when operating in terrain or in a military operation. SLAM both maps the surroundings and localizes the agent within the map without the use of a GPS. A possible future scenario is using a drone to collect data over a large terrain where the UGV operates. This data can be transferred to the UGV, where a SLAM method estimates a map and localizes itself within the map. This would increase the capability of global path planning due to the comprehensive dataset. Hence, a successful application of SLAM is expected to be very valuable for autonomous navigation in terrain.

1.3 Objective and Scope

The goals in which this thesis was set to investigate was comprised into the following objectives:

- Review necessary literature within the field of SLAM.
- Design and build a UGV platform. The platform should be able to collect datasets based on IMU, encoders and lidar.
- Propose a SLAM system based on GMapping. The SLAM system should account for a terrain like scenery and utilize the sensor data to produce a map for autonomous navigation.
- Calibrate the odometry of the UGV and acquire two datasets: a benchmark dataset and a dataset from a terrain like scenery.
- Test the SLAM system on the datasets and tune the algorithm for optimal performance.
- Evaluate the performance of the SLAM system by comparing the two resulting maps estimated on the two different datasets, and including the properties of the algorithm in the evaluation.
- Investigate how the 2D output map estimates a slope.

Limitations

The UGV was built with an open chassis which put the electronics in an exposed position. Thus, to ensure that the risk of damaging critical components was limited the terrain was limited to an indoor obstacle course.

1.4 Levels of Autonomy - Military

Autonomy is in general not a binary property, a robot is not either autonomous or not. Therefore, we need to look into the levels of autonomy (LOA) for the military. The LOA is based on the system's capability to integrate sensing, communication, planning and execution. Autonomy Levels for Unmanned Systems (ALFUS) is a frame-work over three dimensions. The three dimensions for LOA in ALFUS are the External System Independence (ESI), the Mission Complexity (MC) and the Environmental Complexity (EC). Each of the axis has a set of metrics, and each of the scores for the three axis are computed for the resulting autonomy level. There are some options on how this final score is computed. Some options are weighted average and weighted minimum/maximum (Huang (2007)). Depending on how you calculate the score, you will get a result that tells you the LOA. The result will be on a scale from 0 - 10, where (Huang (2007)) has mapped the required scores and the different degrees of difficulty for the environments. Figure 1.1 presents this distribution of the scores.

The LOA in ALFUS is characterized through the missions that the UGV is required to perform or is capable of performing, in the kinds of environments, and with the levels of human interaction (Huang (2007)). In general, the two extremes on the LOA scale are fully manual or fully autonomous conditions (Vagia et al. (2016)).

A common mistake is to mix up the use of the words *autonomy* and *automation*. According to (Vagia et al. (2016)), every engineering system has a certain degree of autonomy associated with it. Autonomy in engineering is the ability to make a choice without any influence of the outside world and to change its initially programmed way of action. In (Parasuraman (2000)) automation is defined as *"... the execution by machine, usually a computer, of a function previously carried out by a human."*. The main difference is that with automation the system "only" performs a pre-defined task. On the other hand, an autonomous system has, depending on its LOA, the ability to make its own decisions without the supervision of an external system or human.

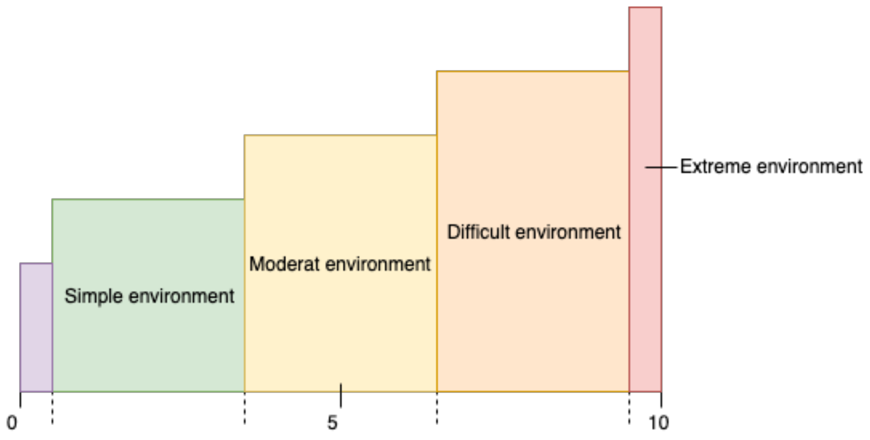


Figure 1.1: Level of autonomy scores for the different types of environments.

1.5 Literature review

In this section the concentration of literature will be around the implementations of SLAM, more precisely the GMapping algorithm. This section is included from the associated project thesis (Bergel (2020)), but with modifications.

In the early 1980's SLAM seriously started to be developed. A definition between the position and the structure of the environment was introduced by (Smith and Cheeseman (1986)). The period between 1986-2004 is known as the *classical age* in the history of SLAM. In this period of time the probabilistic formulations of SLAM were introduced, but also approaches based on extended Kalman filters (EKF), particle filters (PF) and maximum likelihood estimation were developed (Cadena et al. (2016)). A modern approach to SLAM is using nonlinear optimization to estimate the map and to solve a sparse graph of constraints that you apply to the optimization problem (Stachniss et al. (2016)).

The methods from the early stages had several constraints. The EKF method was proposed in the year 1990 and presented the idea of a stochastic map (Girard et al. (2019)). A state vector, computed by EKF, was used to comprise the position of the points of interests. A big drawback with this method is the size of the state vector as it linearly increases with the size of the map. Also, the computational complexity increases quadratically (Girard et al. (2019)). In (Bailey et al. (2006)), the authors examine the cause of the inconsistency of the EKF SLAM. By approximating the mean and variance, EKF SLAM can represent the state uncertainty. There are two problems with this. First, due to linearization the moments are approximated and may not match the true first and second moments. Second, EKF SLAM assumes a Gaussian probability distribution, while the true probability distribution is non-Gaussian. These factors may propagate and accumulate big errors when the SLAM probability distribution is projected in time. Inconsistency in the vehicle's heading may be one consequence (Bailey et al. (2006)).

These and other limitations led to other SLAM methods that were more advanced and computationally efficient. An algorithm that has emerged from this is the FastSLAM algorithm proposed by (Montemerlo et al. (2002)). FastSLAM is a filtering approach, where the map is estimated with EKF and the position is represented by the distribution of particles. The accuracy of the position was reduced with this approach, but the advantage with FastSLAM is the reduction of the algorithm complexity (Girard et al. (2019)).

In (Grisetti et al. (2007b)) the authors discuss the drawback with basic Rao-Blackwellized particle filter (RBPF) SLAM. The number of particles required to learn an accurate map is one of the main problem with this approach. GMapping is an algorithm that builds on this approach and it is based on (Grisetti et al. (2005)) and (Grisetti et al. (2007a)). From the observation likelihood of the most recent sensor information, odometry and scan matching, GMapping computes a proposal distribution. Having a more accurate proposal distribution leads to higher precision when drawing particles, which will result in a reduction in the number of required samples and lower computational time. The adaptive resampling helps to reduce the number of unnecessary resampling actions. Highly accurate maps are estimated in (Grisetti et al. (2007a)). Also, it is opensource and can be implemented in ROS. The output of GMapping is a 2D occupancy grid map.

Another SLAM method implemented in ROS is the Hector SLAM method. This method, along with Karto SLAM, was tested against GMapping in (Duchoň et al. (2019)). Hector SLAM seeks to find optimal alignment with the created map and laser scan endpoints through scan matching. IMU and odometry data are not needed in Hector SLAM, which decreases the complexity of the method. However, with rapid movement and lack of odometry the resulting map may include large errors. In the test between the algorithms, GMapping showed the best and most robust results with its use of odometry and a stochastic model.

The effect of tuning different parameters in the GMapping algorithm are exploited in (Abdelrasoul et al. (2016)). By studying how the computational time and map quality changes along with the parameters, they are able to present tuning guidelines. The algorithm runs on an pre-recorded dataset and is equal for all the runs. The results they present show that the number of particles can be as small as 5 with a resampling threshold at 0.5 at their dataset. It proves that GMapping manage to reduce the required number of particles needed to produce a satisfactory map. By only having 5 particles in the filter the Central Processing Unit (CPU) load and memory consumption decreased.

The authors in (Wang et al. (2016)) examine the effect the quality of the laser scanner has on the loop closure correction in GMapping. Loop closing is detecting if the robot has returned to a previous visited location (Ho and Newman (2006)). They find that a less expensive laser gives a significantly lower loop closure performance than a more sophisticated laser. To be able to use a cheap laser and still perform well with loop closure, the authors present a Kalman filter based closure correction algorithm that corrects the state estimation. This algorithm will again help improve the loop closure performance.

GMapping shows not only promising results indoor but also outdoor. In (Weerasinghe et al. (2016)) they present a mapping and navigation procedure for long distance navigation. The authors in (Balasuriya et al. (2016)) present the use of GMapping in both an indoor and outdoor environment. ROS is used as platform where GMapping runs as a node on a small robot running on belts. The particle filter used is RBPF and the sensor is RPLIDAR 360. The outdoor terrain is sandy and a little rough with barricades, but the experiment shows that the GMapping works properly. Sensor fusion deals with the slippage of the tracks. Both of the papers presents real-time and good quality maps.

In (Brand et al. (2014)) they use SLAM GMapping to obtain a drift-free location and orientation estimate as well as to get a 2D map of the environment. The robot is supposed to operate in both indoor- and outdoor environment. By using stereo-vision the robot can produce a local obstacle map based on depth information from the cameras. This map is used for fast obstacle avoidance and local path planning. The 2.5D obstacle map needs to be projected into a 2D plane to work as an input for the GMapping algorithm. The combination of the depth information from the stereo vision and the SLAM algorithm the robot manage to navigate both indoor and outdoor. The RBPF SLAM algorithm performs loop closure in all cases.

The advantage of GMapping is twofold. First, GMapping shows promising results for both accuracy and complexity. The accurate calculation of the proposal distribution reduces the computational time substantially compared to, for example, EKF based methods. GMapping generates highly accurate and robust maps in the environments it has been tested. Second, GMapping estimates a 2D map which would struggle with mapping slopes and other 3D related features. If GMapping manages to include all the relevant 2D features in a terrain like scenery, there are other technologies that might supplement the SLAM system in creating a sense of space and still retain the real-time computational time.

1.6 Contribution

The contribution of this thesis is preliminary designing and constructing a robotic UGV platform with the capabilities of conducting a robust dataset in a terrain like environment. The contribution will also revolve around utilizing the existing SLAM method GMapping for autonomous navigation. The contribution of the Master's thesis can be summarized as the following:

- A literature review mapping SLAM approaches.
- Designing and building a UGV platform for data acquisition with the purpose of investigating autonomous navigation.
- A proposed SLAM system for mapping using lidar for perceiving the environment.
- Testing the SLAM system with GMapping on the acquired datasets.

1.7 Outline of Report

This thesis is divided into seven chapters. The introduction in **Chapter 1** presents the thesis definition, the SLAM literature review and the military levels of autonomy. In **Chapter 2** the theory behind the lidar and the theoretic foundation in developing SLAM is explained. The UGV platform is thoroughly explained in **Chapter 3**, before **Chapter 4** presents the methodology for developing the SLAM system. **Chapter 5** displays the experimental results. **Chapter 6** discusses the UGV platform and experimental results, while **Chapter 7** draws a thesis conclusion and recommend further work.

Chapter 2

Theory

This chapter presents the most relevant theory used in this thesis. Section 2.1 explains the mathematics behind the transformation between two coordinate frames. Section 2.2 describes the theory behind the lidar. Section 2.3 gives an introduction to SLAM, Rao-Blackwellized particle filter SLAM and GMapping, while Section 2.6 presents occupancy grid map. Lastly, Section 2.7 describes the framework of the implementation, ROS. Section 2.2 through 2.6, with the exception of Section 2.2.1, are included from the associated project thesis by (Bergel (2020)). There are, however, modifications made.

2.1 Rotation and Translation Between Coordinate Frames

A robot will contain several different joints where all of them have their own coordinate frame. It is convenient to represent the different frames in the same coordinate system. The notation for transforming a coordinate vector from one coordinate frame to another used in this project is equal to the notation used in (Fossen (2020)):

$$\mathbf{v}^{to} = \mathbf{R}^{to}_{from}\mathbf{v}^{from}. \tag{2.1}$$

Here, $\mathbf{R}^{to}_{from}$ is the 3×3 rotation matrix that is applied on the original coordinate vector $\mathbf{v}^{from} \in \mathbb{R}^3$ to obtain the new reference frame $\mathbf{v}^{to} \in \mathbb{R}^3$.

In (Euler (1776)), a theorem for relative orientation of two rigid bodies is stated.

Theorem 2.1.1 (Euler's Theorem on Rotation)**.** *Every change in the relative orientation of two rigid bodies or reference frames $\{\mathcal{A}\}$ and $\{\mathcal{B}\}$ can be produced by means of a simple rotation of $\{B\}$ in $\{A\}$.*

A simple rotation is a rotation along only one axis, and by using the Euler angles (roll (ϕ), pitch (θ) and yaw (ψ)) the principal rotation matrices

$$\mathbf{R}_{x,\phi} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & c\phi & -\mathrm{s}\phi \\ 0 & \mathrm{s}\phi & c\phi \end{bmatrix}, \mathbf{R}_{y,\theta} = \begin{bmatrix} \mathrm{c}\theta & 0 & \mathrm{s}\theta \\ 0 & 1 & 0 \\ -\mathrm{s}\theta & 0 & \mathrm{c}\theta \end{bmatrix}, \mathbf{R}_{z,\psi} = \begin{bmatrix} \mathrm{c}\psi & -\mathrm{s}\psi & 0 \\ \mathrm{s}\psi & \mathrm{c}\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

can be obtained, where $c = cos$ and $s = sin$ and the $\mathbf{R}_{\lambda,\beta}$ corresponds to a rotation β about the λ axis (Fossen (2020)). With zero rotation $\mathbf{R} = \mathbf{I}$. The rotation matrix $\mathbf{R}_{from}^{to}$ is usually described using the three principal matrices in Equation 2.2 about the z, y and x axis. According to (Fossen (2020)), this is because the rotation sequence, with the Euler angles, is mathematical equivalent to

$$\mathbf{R}_{from}^{to} := \mathbf{R}_{z,\psi}\mathbf{R}_{y,\theta}\mathbf{R}_{x,\phi}. \tag{2.3}$$

Along with the rotation, a physical distance between the frames might also be present. This distance is represented in a translation vector

$$\mathbf{t}_{from}^{to} = \begin{bmatrix} x_t \\ y_t \\ z_t \end{bmatrix} \tag{2.4}$$

where x, y and z represents the physical distance between the two frames in meters. By combining Equation 2.3 with 2.4 will yield a 4×4 transformation matrix

$$\mathbf{T}_{from}^{to} = \begin{bmatrix} \mathbf{R}_{from}^{to} & \mathbf{t}_{from}^{to} \\ \mathbf{0}_{1\times 3} & 1 \end{bmatrix} \tag{2.5}$$

transforming from one frame to another.

2.2 Lidar

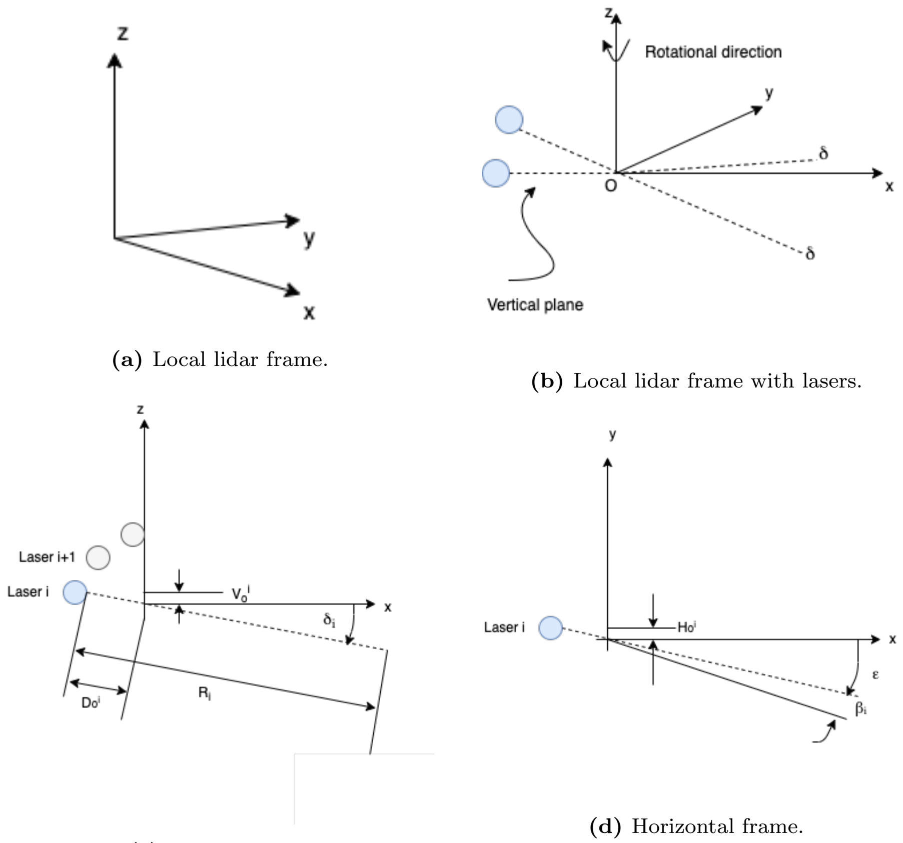
A common sensor used for collecting data for SLAM methods is the lidar. Lidar is an acronym for "light detection and ranging" and is used to measure the distance to an object. The differences in the distances can be used to create a map of the object or the environment the lidar is scanning. A typical lidar emits a pulsed light wave to the environment. When the wave hits an object it will reflect back and return to the sensor. By knowing the time-of-flight, the distance the light wave has traveled can be calculated. By repeating this process millions of times per second a real-time map of the environment is created (VelodyneLidar). The calculation of distance

$$d = \frac{1}{2}Ct_{wait} \tag{2.6}$$

are shown in Equation 2.6 where d = distance, C = speed of light and t_{wait} = time-of-flight. The lidar that will be presented in this section is the VLP-16 lidar.

2.2.1 Mathematical model of the VLP-16

The Velodyne VLP-16 lidar, also called scanner, has a total of 16 laser-detector pairs. To be able to compose a point cloud of the surroundings based on the scanners it is necessary to know the mathematical model of the lidar. Each laser-detector pair has a 30° field of view (FOV) and increments 2° horizontally over the FOV from −15° to 15° (Glennie et al. (2016)). Each scanner also scans vertically to obtain a 3D dataset.



(a) Local lidar frame.

(b) Local lidar frame with lasers.

(c) Vertical plane.

(d) Horizontal frame.

Figure 2.1: A representation of the scanner and laser frame with variables.

Figure 2.1 shows the geometry and the parameters of the lasers in the lidar. The coordinates (x, y, z) of a observed point in the frame of the scanner can be computed as

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \left(\mathrm{s^iR_i} + \mathrm{D_o^i}\right) \cdot \cos\left(\delta_\mathrm{i}\right) \cdot \left[\sin(\varepsilon) \cdot \cos\left(\beta_\mathrm{i}\right) - \cos(\varepsilon) \cdot \sin\left(\beta_\mathrm{i}\right)\right] \\ -\mathrm{H_o^i} \cdot \left[\cos(\varepsilon) \cdot \cos\left(\beta_\mathrm{i}\right) + \sin(\varepsilon) \cdot \sin\left(\beta_\mathrm{i}\right)\right] \\ \\ \left(\mathrm{s^iR_i} + \mathrm{D_o^i}\right) \cdot \cos\left(\delta_\mathrm{i}\right) \cdot \left[\cos(\varepsilon) \cdot \cos\left(\beta_\mathrm{i}\right) + \sin(\varepsilon) \cdot \sin\left(\beta_\mathrm{i}\right)\right] \\ +\mathrm{H_o^i} \cdot \left[\sin(\varepsilon) \cdot \cos\left(\beta_\mathrm{i}\right) - \cos(\varepsilon) \cdot \sin\left(\beta_\mathrm{i}\right)\right] \\ \\ \left(\mathrm{s^iR_i} + \mathrm{D_o^i}\right) \cdot \sin\left(\delta_i\right) + \mathrm{V_o^i} \end{bmatrix} \tag{2.7}$$

based on the geometry, where

$\mathrm{s^i}$	distance scale factor for laser i
$\mathrm{D_o^i}$	distance offset for laser i
δ_i	vertical rotation correction for laser i
β_i	horizontal rotation correction for laser i
$\mathrm{H_o^i}$	horizontal offset from scanner frame origin for laser i
$\mathrm{V_o^i}$	vertical offset from scanner frame origin for laser i
$\mathrm{R_i}$	raw distance measurement from laser i
ϵ	encoder angle measurement

To present the points in a global frame, a planar-based approach is used where the points are conditioned on a plane. A functional model describing this can be expressed as

$$\left\langle \overrightarrow{\mathbf{g}}_k, \begin{bmatrix} \overrightarrow{\mathbf{r}} \\ 1 \end{bmatrix} \right\rangle = 0 \tag{2.8}$$

where $\overrightarrow{\mathbf{g}}_k = \begin{bmatrix} g_1 & g_2 & g_3 & g_4 \end{bmatrix}^T$ are the unknown a priori parameters of plane k and $\overrightarrow{\mathbf{r}}$ is the vector of the lidar observations in a global coordinate frame (Glennie and Lichti (2010)). The direction cosines of plane k's normal vector are represented by g_1, g_2 and g_3 and need to satisfy the following unit length constraint

$$g_1^2 + g_2^2 + g_3^2 - 1 = 0. \tag{2.9}$$

g_4 is the negative orthogonal distance between the origin of the coordinate system and the plane k (Skaloud and Lichti (2006)). Equation 2.7 shows the coordinates in the local frame of the scanner. Since the functional model in Equation 2.8 requires the coordinates to be in the global frame a transformation from the local scanner frame l to the global frame g is needed. Following, is the rigid body transformation

$$\overrightarrow{\mathbf{r}}^g = \mathbf{T}_l^g \overrightarrow{\mathbf{r}}_j^l \tag{2.10}$$

of Equation 2.7. Here, the vector $\overrightarrow{\mathbf{r}}^l$ is given in Equation 2.7 for a point i. Each point may be scanned from different scan locations. Therefore, the index j represents the given scan location that the point were scanned from. Furthermore,

$\vec{\mathbf{r}}^g$ is the transformation matrix from a local scanner frame to the global lidar frame. Due to the fact that one point may be scanned from multiple scan locations and that the parameters and observations in Equation 2.8 are not separable, an adjustment model is needed. Therefore, the Gauss-Helmert adjustment model is applied to fit the parameters to the model (Skaloud and Lichti (2006)).

Gauss-Helmert Adjustment Model

The Gauss-Helmert adjustment model is usually applied to find the solution of an *overdetermined* system of equations. An overdetermined system is a system where there are more equations than unknowns. It is based on the combination of two principles: least squares and observation residuals. As there are two sets of unknown in the functional model (Equation 2.8), the boresight angles (seen in Figure 2.1) and the plane parameters, a linearization of the system is needed and will take the form

$$\mathbf{A}d\mathbf{x} + \mathbf{B}\mathbf{v} + \mathbf{w}_0 = 0. \tag{2.11}$$

$\mathbf{A}$ and $\mathbf{B}$ are the matrices containing the partial derivatives of the functional model (Equation 2.8) with respect to boresight angles and the plane parameters, respectively. $d\mathbf{x}$, $\mathbf{v}$ and $\mathbf{w}_0$ are the correction vector for approximation of parameter values, residual vector and misclosure vector, respectively. Furthermore, the constraint given in Equation 2.9 is linearized as

$$\mathbf{A}_c d\mathbf{p} + \mathbf{w}_c = \mathbf{v}_c \tag{2.12}$$

where $\mathbf{A}_c$ is the partial derivative of Equation 2.9 with respect to the plane parameters, $d\mathbf{p}$ is the correction vector for approximating the plane parameters and c is the number of constraints (Bae and Lichti (2007)).

Solution of the Adjustment Model

To solve the Gauss-Helmert adjustment model a combined least-square method is used. The method seeks to minimize the distance between points and their corresponding planes with a constraint condition. (Bae and Lichti (2007) proposed the cost function to be minimized as

$$\begin{aligned}\varphi = \mathbf{v}^T\mathbf{P}\mathbf{v} + \mathbf{v}_c^T\mathbf{P}_c\mathbf{v}_c + 2\mathbf{k}^T\left(\mathbf{A}d\mathbf{x} + \mathbf{B}\mathbf{v} + \mathbf{w}_0\right)\\ + 2\mathbf{k}_c^T\left(\mathbf{A}_c d\mathbf{p} + \mathbf{w}_c - \mathbf{v}_c\right).\end{aligned} \tag{2.13}$$

where the function is a combination of weighted squares of the residuals which are subject to constraints of the two models found in Equation 2.11 and Equation 2.12 (Skaloud and Lichti (2006)). $\mathbf{k}$ and $\mathbf{k}_c$ are the Lagrange multiplier vector, while $\mathbf{P}$ and $\mathbf{P}_C$ are the weight matrices for the observations and the constraints, respectively. From (Bae and Lichti (2007)), the abbreviated notation for the minimized cost function is

$$\mathbf{N}d\mathbf{x} + \mathbf{w} = \mathbf{0} \tag{2.14}$$

where

$$\mathbf{N} = \mathbf{A}^T \left(\mathbf{B}\mathbf{P}^{-1}\mathbf{B}^T\right)^{-1} \mathbf{A} + \begin{bmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_c^T \mathbf{P}_c \mathbf{A}_c \end{bmatrix} \tag{2.15}$$

and

$$\mathbf{w} = \mathbf{A}^T \left(\mathbf{B}\mathbf{P}^{-1}\mathbf{B}^T\right)^{-1} \mathbf{w}_0 + \begin{bmatrix} \mathbf{0} \\ \mathbf{A}_c^T \mathbf{P}_c \mathbf{w}_c \end{bmatrix}. \tag{2.16}$$

2.2.2 Weather Influence

When lidar is used outdoor different weather phenomena might influence the performance of the scanning. As an example, foggy weather conditions will break down the performance of state of the art lidars. The maximum viewing distance will be limited to only a fraction compared to clear weather capabilities (Bijelic et al. (2018)). Furthermore, rain may also influence the accuracy of the lidar were beams will be effected by rain drops in the air (Hasirlioglu et al. (2016)). Other weather conditions like, snow or sunshine, may also affect the accuracy of the information the lidar provides.

2.3 Simultaneous Localization and Mapping

According to (Cadena et al. (2016)), Simultaneous Localization and Mapping (SLAM) is a simultaneous estimation of the state of the robot and the construction of a model (map) of the surrounding environment. The basic instants of the robot's state are position and orientation, while the map represents the aspects of interest describing the environment the robot is operating in. To be able to construct a model of the environment the robot is equipped with sensors that perceives the surroundings (Cadena et al. (2016)).

With other words, SLAM seeks to recover a model of the surroundings and the robot state based on odometry and measurement data (Stachniss et al. (2016)). Mathematically, SLAM is usually described in a probabilistic terminology. Let t denote the time, while x_t denotes the position of the robot represented as a three-dimensional vector that includes a 2D coordinate and a single rotational value (Stachniss et al. (2016)). Then the sequence of locations for the robot is given as

$$X_T = \{x_0, x_1, x_2, \ldots, x_T\} \tag{2.17}$$

where T is terminal time and x_0 is initial location (Stachniss et al. (2016)). Furthermore, the relative information between t and $t-1$ is provided by the odometry, for example data from the robot's wheels. This relative motion of the robot can be described as a sequence

$$U_T = \{u_1, u_2, u_3 \ldots, u_T\} \tag{2.18}$$

where u_t is the odometry between two following locations (Stachniss et al. (2016)). Lastly, the map of the environment will be denoted by m. The location of the landmarks, objects and surfaces in the environment will be described by m. Information between the features in the environment m and the location of the robot x_t will be provided through robot measurements. The sequence of these measurements will be

$$Z_T = \{z_1, z_2, z_3, \ldots, z_T\} \tag{2.19}$$

where z_t is a measurement at a given time t (Stachniss et al. (2016)).

2.3.1 System architecture

According to (Cadena et al. (2016)), a modern SLAM system consists of two main components: *front-end* and *back-end* (see Figure 2.2).

- **Front-end**: In this part of the system relevant data is extracted from the raw measurements provided by the sensors. Furthermore, it also associates each measurement to a specific landmark in the environment and builds the optimization problem.
- **Back-end**: The back end performs inference on the data provided by the front end and solves the optimization problem to estimate the map.

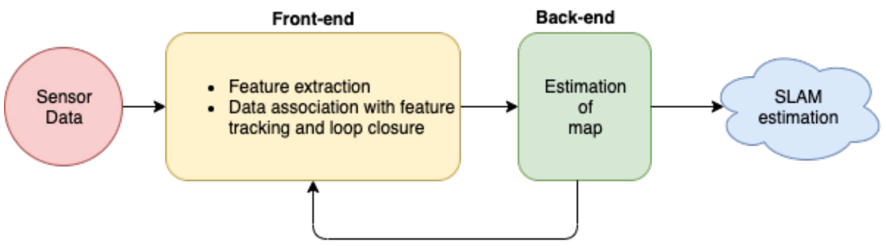


Figure 2.2: Overview of the anatomy of a modern SLAM system.

2.3.2 The two SLAM problems

The SLAM problem can be divided into two equally important problems; the *online* SLAM problem and the *full* SLAM problem. The full SLAM problem can be expressed as the joint posterior probability

$$p\left(X_T, m | Z_T, U_T\right) \tag{2.20}$$

over the sequence in Equation 2.17 and m, from the data in the sequences in Equation 2.18 and Equation 2.19 (Stachniss et al. (2016)).

The online SLAM problem differs from the previous problem by that it only seeks to recover the present robot location instead of the entire sequence (Equation 2.17). This problem can be defined as

$$p\left(x_t, m|Z_t, U_t\right). \tag{2.21}$$

Comparing the two SLAM problems; the full SLAM problem algorithms are usually batch, which means they process all data at the same time. On the other side, the algorithms for the online SLAM problem are usually incremental and can only process one data item at a time. These kind of algorithms are referred to as *filters* (Stachniss et al. (2016)).

There are three major paradigms of SLAM that are used to derive methods for solving the SLAM problem. The three methods are: *Extended Kalman Filter*(EKF), *Particle Filter*(PF) and graph-based (Stachniss et al. (2016)). EKF SLAM seeks to solve the online SLAM problem in Equation 2.21, while graph-based SLAM seeks to solve the full SLAM problem in Equation 2.20. PF SLAM, on the other hand, can be used to solve both full and online SLAM problems. This is because, for example in some PF algorithms, each particle has a sample of an entire path but it is only the recent pose that is used in the update equation (Stachniss et al. (2016)). The properties and potential pros and cons of the various methods are discussed in detail in Section 1.5.

The algorithms considered in this project are a version of the Rao-Blackwellized Particle Filter SLAM method, GMapping. The following sections will therefore only regard relevant theory based on this topic.

2.4 Rao-Blackwellized particle filter SLAM

Rao-Blackwellized particle filter (RBPF) SLAM is an online SLAM method (expressed in Equation 2.21) which seeks to estimate the *joint posterior probability*

$$p\left(x_{1:t}, m|z_{1:t}, u_{1:t-1}\right) \tag{2.22}$$

where $x_{1:t}$ is the potential trajectory of the robot, m is the potential map, $z_{1:t}$ is the observations and $u_{1:t-1}$ is the odometry measurements (Grisetti et al. (2007a)). In general, a particle filter is used to estimate the internal state in the system based on partial observations. In other words, the filter uses a set of particles, or samples, to estimate the posterior distribution from some stochastic process with noise. In the context of Equation 2.22, the particle filter seeks to estimate the trajectory and map, based on the sensor observations and odometry data. To make it more computationally efficient, a factorization of Equation 2.22 is introduced

$$p\left(x_{1:t}, m|z_{1:t}, u_{1:t-1}\right) = p\left(m|x_{1:t}, z_{1:t}\right) \cdot p\left(x_{1:t}|z_{1:t}, u_{1:t-1}\right)\ . \tag{2.23}$$

This factorization provides the opportunity to first only estimate the trajectory of the robot and then compute the map based on that trajectory. This may be referred to as the *Rao-Blackwellization* (Grisetti et al. (2007a)), and hence the name Rao-Blackwellized particle filter. The first term in Equation 2.23, $p\left(m|x_{1:t}, z_{1:t}\right)$, can be computed analytically since the trajectory $x_{1:t}$ and $z_{1:t}$ is given (Grisetti et al. (2005)). The second term , the posterior probability $p\left(x_{1:t}|z_{1:t}, u_{0:t}\right)$, is estimated using *a particle filter* where each particle represents a potential trajectory of the robot. An individual map is created for each sample and is updated based on the observations and trajectory for each particle. The sampling importance resampling (SIR) filter is one of the most common particle filters and is the filter that is used in this project together with Rao-Blackwellization (Grisetti et al. (2007a)).

2.4.1 Mapping with Rao-Blackwellized SIR filter

According to (Grisetti et al. (2007a)), the SIR algorithm consists of three steps: *sampling step*, *importance step* and *resampling step*. Mapping with a Rao-Blackwellized SIR filter will have the following four steps:

1. **Sampling** - In this first step, the next generation of samples are drawn from the so-called proposal distribution π (Stachniss et al. (2007)). In other words, the goal is to obtain $\left\{x_t^{(i)}\right\}$ from $\left\{x_{t-1}^{(i)}\right\}$ by sampling from π.

2. **Importance Weighting** - The proposal distribution is usually not equal to the target distribution (Stachniss et al. (2007)). To make up for this each particle is assigned an *importance weight* $w_t^{(i)}$. The importance weight principle that calculates each weight is given by

$$w_t^{(i)} = \frac{p\left(x_{1:t}^{(i)}|z_{1:t}, u_{1:t-1}\right)}{\pi\left(x_{1:t}^{(i)}|z_{1:t}, u_{1:t-1}\right)}. \tag{2.24}$$

3. **Resampling** - The goal for this step is to obtain the target distribution from the weighted proposal from the previous step. Resampling is done to avoid degeneracy (Koutsojannis and Sirmakessis (2009)). This is achieved by drawing particles based on their weights. (Stachniss et al. (2007)). Typically, particles with low $w_t^{(i)}$ get replaced by samples with high $w_t^{(i)}$ (Grisetti et al. (2005)).

4. **Map Estimating** - The last step of the process, is estimating the corresponding map

$$p\left(m^{(i)}|x_{1:t}^{(i)}, z_{1:t}\right) \tag{2.25}$$

for each particle.

Over time the trajectory sequence in Equation 2.17 will be become large, which leads to an inefficient algorithm. In (Grisetti et al. (2007a)), a recursive formulation is obtained to deal with this problem. The proposal distribution π is restricted to follow the assumption

$$\pi\left(x_{1:t}|z_{1:t}, u_{1:t-1}\right) = \pi\left(x_t|x_{1:t-1}, z_{1:t}, u_{1:t-1}\right) \cdot \pi\left(x_{1:t-1}|z_{1:t-1}, u_{1:t-2}\right) \tag{2.26}$$

to calculate the importance weights.

By combining Equation 2.24 and 2.26 the modified weights are computed. Following,

$$w_t^{(i)} = \frac{p\left(x_{1:t}^{(i)}|z_{1:t}, u_{1:t-1}\right)}{\pi\left(x_{1:t}^{(i)}|z_{1:t}, u_{1:t-1}\right)} \tag{2.27}$$

$$= \frac{\eta p\left(z_t|x_{1:t}^{(i)}, z_{1:t-1}\right) p\left(x_t^{(i)}|x_{t-1}^{(i)}, u_{t-1}\right)}{\pi\left(x_t^{(i)}|x_{1:t-1}^{(i)}, z_{1:t}, u_{1:t-1}\right)} \cdot \underbrace{\frac{p\left(x_{1:t-1}^{(i)}|z_{1:t-1}, u_{1:t-2}\right)}{\pi\left(x_{1:t-1}^{(i)}|z_{1:t-1}, u_{1:t-2}\right)}}_{w_{t-1}^{(l)}} \tag{2.28}$$

$$\propto \frac{p\left(z_t|m_{t-1}^{(i)}, x_t^{(i)}\right) p\left(x_t^{(i)}|x_{t-1}^{(i)}, u_{t-1}\right)}{\pi\left(x_t|x_{1:t-1}^{(i)}, z_{1:t}, u_{1:t-1}\right)} \cdot w_{t-1}^{(i)} \tag{2.29}$$

shows the mathematical representation of the importance weights, where the normalization factor $\eta = 1/p\left(z_t|z_{1:t-1}, u_{1:t-1}\right)$ comes from the Bayes' rule that is equal for all particles (Grisetti et al. (2007a)).

The SLAM method this thesis investigates is based upon RBPF with improved proposals and adaptive resampling. The next section will therefore consider the modified version.

2.5 Modified Rao-Blackwellized particle filter SLAM

This section will consider a modified version of the Rao-Blackwellized particle filter. The new version have an *improved proposal distribution* and an *adaptive resampling*. The following two sections will explain first the improved proposal before addressing the adaptive resampling.

2.5.1 Improved proposals

There is a close connection on how well the particle filters perform and how accurate the proposal distribution approximates the target distribution. In a perfect world drawing directly from the target distribution to obtain the next generation of particles would be possible. This is not possible when solving a SLAM problem, therefore different proposal distributions have been considered (Grisetti et al. (2007a)). In (Montemerlo et al. (2002)) a proposal distribution based on the odometry motion model is proposed, where the importance weights are calculated based on the observation model $p\left(z_t|m, x_t\right)$. The advantage with this distribution is that it is easy to compute for most robots, but it is only optimal if for instance the sensor information is significantly more precise than the motion estimate (Grisetti et al. (2007a)). Figure 2.3 shows the small area $L^{(i)}$ where the drawn samples includes the state space regions that have a high likelihood under the observation model. Because of this, the individual samples can differ a lot from each other. To overcome this problem, a high number of samples are needed to cover the high observation likelihood regions (Grisetti et al. (2007a)). Another approach is to use a smoothed likelihood function but this may lead to information being discarded. In a SLAM context this may result in less accurate maps (Grisetti et al. (2007a)).

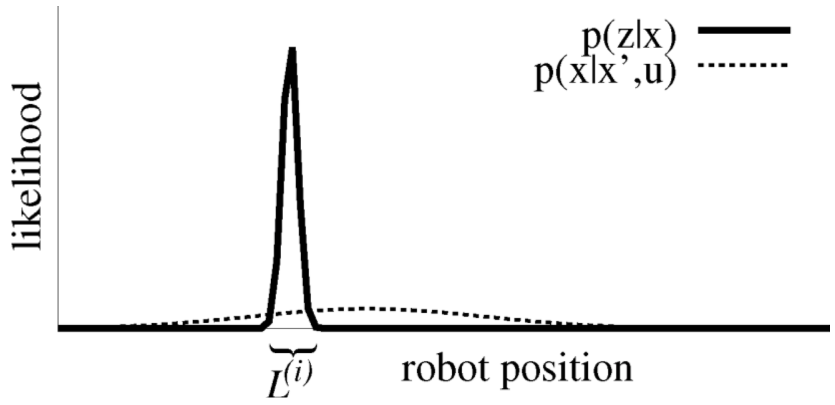


Figure 2.3: Interval $L^{(i)}$ shows where the two functions are dominated by the observation likelihood (Grisetti et al. (2007a)) © 2007 IEEE.

According to (Doucet (1998)),

$$p\left(x_t|m_{t-1}^{(i)}, x_{t-1}^{(i)}, z_t, u_{t-1}\right) = \frac{p\left(z_t|m_{t-1}^{(i)}, x_t\right) p\left(x_t|x_{t-1}^{(i)}, u_{t-1}\right)}{p\left(z_t|m_{t-1}^{(i)} x_{t-1}^{(i)}, u_{t-1}\right)} \tag{2.30}$$

is said to be the optimal proposal distribution. The recent sensor observation z_t is integrated into the proposal. By using Equation 2.30 when computing the importance weights the new weights will become the following

$$\begin{aligned} w_t^{(i)} &= w_{t-1}^{(i)} \frac{\eta p\left(z_t|m_{t-1}^{(i)}, x_t^{(i)}\right) p\left(x_t^{(i)}|x_{t-1}^{(i)}, u_{t-1}\right)}{p\left(x_t|m_{t-1}^{(i)}, x_{t-1}^{(i)}, z_t, u_{t-1}\right)} && (2.31) \\ &\propto w_{t-1}^{(i)} \frac{p\left(z_t|m_{t-1}^{(i)}, x_t^{(i)}\right) p\left(x_t^{(i)}|x_{t-1}^{(i)}, u_{t-1}\right)}{\frac{p\left(z_t|m_{t-1}^{(i)}, x_t\right) p\left(x_t|x_{t-1}^{(i)}, u_{t-1}\right)}{p\left(z_t|m_{t-1}^{(i)}, x_{t-1}^{(i)}, u_{t-1}\right)}} && (2.32) \\ &= w_{t-1}^{(i)} \cdot p\left(z_t|m_{t-1}^{(i)}, x_{t-1}^{(i)}, u_{t-1}\right) && (2.33) \\ &= w_{t-1}^{(i)} \cdot \int p\left(z_t|x'\right) p\left(x'|x_{t-1}^{(i)}, u_{t-1}\right) dx' && (2.34) \end{aligned}$$

This improved proposal is convenient when the sensor is very accurate which may cause a high peak at the likelihood function and makes one focus the sampling on the important parts of the observation likelihood (Grisetti et al. (2007a)). As mentioned before, when the likelihood function is peaked one needs a dense sampling to capture the small areas of the likelihood. An observation made in (Grisetti et al. (2007a)) is that the target distribution has a limited number of maxima, and usually only one.

GMapping is based on the previous approaches with the proposal distribution in Equation 2.30, the computation of the new importance weights and focusing the sampling around the maxima of the likelihood function. The main difference is that a scan-matcher is used in the beginning to find the meaningful areas (peaks) of the likelihood function so that the posterior in Equation 2.30 can be approximated around the maximum of the likelihood function (Grisetti et al. (2007a)). This reduces the number of required particles. However, the geometry and size of the environment decides the number of particles needed to generate a high-quality map (Abdelrasoul et al. (2016)). A Gaussian approximation $\mathcal{N}$ is computed to efficiently draw the next generation of samples. Furthermore, the meaningful areas are sampled and each sampled point is compared to the target distribution. Two Gaussian parameters $\mu_t^{(i)}$ and $\Sigma_t^{(i)}$ are determined for each particle i. They represent the mean and the variance, respectively. Following,

$$\mu_t^{(i)} = \frac{1}{\eta^{(i)}} \cdot \sum_{j=1}^{K} x_j \cdot p\left(z_t | m_{t-1}^{(i)}, x_j\right) \cdot p\left(x_j | x_{t-1}^{(i)}, u_{t-1}\right) \tag{2.35}$$

$$\Sigma_t^{(i)} = \frac{1}{\eta^{(i)}} \cdot \sum_{j=1}^{K} p\left(z_t | m_{t-1}^{(i)}, x_j\right) \cdot p\left(x_j | x_{t-1}^{(i)}, u_{t-1}\right) \cdot \left(x_j - \mu_t^{(i)}\right)\left(x_j - \mu_t^{(i)}\right)^T \tag{2.36}$$

shows how they are computed. K represents the number of the sampled points x_j in the interval $L^{(i)}$ (Grisetti et al. (2007a)). In this approach the normalization factor $\eta^{(i)}$ is computed in the following way

$$\eta^{(i)} = \sum_{j=1}^{K} p\left(z_t | m_{t-1}^{(i)}, x_j\right) \cdot p\left(x_j | x_{t-1}^{(i)}, u_{t-1}\right). \tag{2.37}$$

When the next generation particles are obtained, the importance weights are computed based on the proposal distribution

$$w_t^{(i)} = w_{t-1}^{(i)} \cdot p\left(z_t | m_{t-1}^{(i)}, x_{t-1}^{(i)}, u_{t-1}\right) \tag{2.38}$$

$$= w_{t-1}^{(i)} \cdot \int p\left(z_t | m_{t-1}^{(i)}, x'\right) \cdot p\left(x' | x_{t-1}^{(i)}, u_{t-1}\right) dx \tag{2.39}$$

$$\simeq w_{t-1}^{(i)} \cdot \sum_{j=1}^{K} p\left(z_t | m_{t-1}^{(i)}, x_j\right) \cdot p\left(x_j | x_{t-1}^{(i)}, u_{t-1}\right) \tag{2.40}$$

$$= w_{t-1}^{(i)} \cdot \eta^{(i)}. \tag{2.41}$$

To sum up this section, the computations above helps determine the Gaussian parameters of the proposal distribution for each particle individually. The proposal distribution allows an efficient sampling and takes both sensor observations and odometry reading into account. This results in a more accurate estimation (Grisetti et al. (2007a)).

2.5.2 Adaptive resampling

By looking back at subsection 2.4.1, the calculations above goes through the first 2 steps in the Rao-Blackwellized SIR filter algorithm: *sampling* and calculation of the *importance weights*. The next step that is modified in GMapping is the *resampling* step. As mentioned earlier; while resampling, particles with low importance weight $w^{(i)}$ is replaced with particles with high $w^{(i)}$. This may discard good samples that would have improved the performance. To deal with this problem a criterion is needed to decide whether or not to do the resampling (Grisetti et al. (2007a)). According to (Doucet (1998)), a quantity that can be used as a criterion for this is

$$N_{\text{eff}} = \frac{1}{\sum_{i=1}^{N} \left(\tilde{w}^{(i)}\right)^2} \tag{2.42}$$

where $\tilde{w}^{(i)}$ is the normalized weight of particle i. When N_{eff} drops below the threshold $N/2$, where N is number of particles, resampling will be done. N_{eff} also contributes to avoid "particles depletion" when having a small number of particles and frequently resampling (Abdelrasoul et al. (2016)).

2.5.3 GMapping in pseudocode

Algorithm 1 sums up GMapping in pseudocode according to (Grisetti et al. (2007a)). The algorithm is based on the RBPF SLAM method but with an improved proposal distribution and adaptive resampling, as explained in previous sections.

Algorithm 1 Improved RBPF for Map Learning

```
Require:
 1: $\mathcal{S}_{t-1}$, the sample set of the previous time step
 2: $z_t$, the most recent sensor scan
 3: $u_{t-1}$, the most recent odometry measurement
 4:
Ensure:
 5: $\mathcal{S}_t$, the new sample set
 6: $\mathcal{S}_t = \{\}$
 7:

 8: for all $s_{t-1}^{(1)} \in \mathcal{S}_{t-1}$ do
 9:     $\left\langle x_{t-1}^{(i)}, w_{t-1}^{(i)}, m_{t-1}^{(i)} \right\rangle = s_{t-1}^{(i)}$
10:     $x_t'^{(i)} = x_{t-1}^{(i)} \oplus u_{t-1}$
11:     $\hat{x}_t^{(i)} = \text{argmax}_x\, p\left(x|m_{t-1}^{(i)}, z_t, x_t'^{(i)}\right)$
12:     if $\hat{x}_t^{(i)} = \text{failure}$ then
13:         $x_t^{(i)} \sim p\left(x_t|x_{t-1}^{(i)}, u_{t-1}\right)$
14:         $w_t^{(i)} = w_{t-1}^{(i)} \cdot p\left(z_t|m_{t-1}^{(i)}, x_t^{(i)}\right)$
15:     else
16:         for $k = 1, ..., K$ do
17:             $x_k \sim \left\{x_j || x_j - \hat{x}^{(i)}| < \Delta\right\}$
18:         end for

19:         $\mu_t^{(i)} = (0,0,0)^T$
20:         $\eta^{(i)} = 0$

21:         for all $x_j \in \{x_1, ..., x_K\}$ do
22:             $\mu_t^{(i)} = \mu_t^{(i)} + x_j \cdot p\left(z_t|m_{t-1}^{(i)}, x_j\right) \cdot p\left(x_t|x_{t-1}^{(i)}, u_{t-1}\right)$
23:             $\eta^{(i)} = \eta^{(i)} + p\left(z_t|m_{t-1}^{(i)}, x_j\right) \cdot p\left(x_t|x_{t-1}^{(i)}, u_{t-1}\right)$
24:         end for
```

```
25:             $\mu_t^{(i)} = \mu_t^{(i)} | \eta^{(i)}$
26:             $\Sigma_t^{(i)} = \mathbf{0}$
27:             for all $x_j \in \{x_1, ..., x_K\}$ do
28:                 $\Sigma_t^{(i)} = \Sigma_t^{(i)} + \left(x_j - \mu^{(i)}\right)\left(x_j - \mu^{(i)}\right)^T \cdot p\left(z_t | m_{t-1}^{(i)}, x_j\right) \cdot p\left(x_j | x_{t-1}^{(i)}, u_{t-1}\right)$
29:             end for
30:             $\Sigma_t^{(i)} = \Sigma_t^{(i)} / \eta^{(i)}$
31:             $x_t^{(i)} \sim \mathcal{N}\left(\mu_t^{(i)}, \Sigma_t^{(i)}\right)$
32:             $w_t^{(i)} = w_{t-1}^{(i)} \cdot \eta^{(i)}$
33:         end if
34:         $m_t^{(i)} = \text{integrateScan}(m_{t-1}^{(i)}, x_t^{(i)}, z_t)$
35:         $\mathcal{S}_t = \mathcal{S}_t \cup \left\{\left\langle x_t^{(i)}, w_t^{(i)}, m_t^{(i)} \right\rangle\right\}$
36:     end for
37:     $N_{\text{eff}} = 1 / \sum_{i=1}^{N} \left(\tilde{w}^{(i)}\right)^2$
38:     if $N_{eff} < T$ then
39:         $\mathcal{S}_t = \text{resample } \mathcal{S}_t$
40:     end if
```

The worst case scenario, regarding computational time, may occur when resampling. However, since the algorithm uses adaptive resampling only a few resampling steps are necessary (Grisetti et al. (2007a)). Also, different parameters have direct influence on the computational time. An example on parameters that can be tuned, are how often new observations are processed. A low frequency may result in a poor constructed map but decreases the CPU load (Abdelrasoul et al. (2016)).

2.6 Occupancy Grid Map

The output of GMapping is a 2D occupancy grid map. The idea is to have an evenly spaced field where binary variables represent if an obstacle is present or not at that location. Depending on the resolution of the map there is a fixed number of grid cells $\langle x, y \rangle$. Each grid cell has an occupancy value attached. These values measure the subjective belief if the robot can be moved to the center of that cell or not (Thrun and Bücken (1996)). The discrete grid values can be estimated differently but in GMapping it is estimated from $p\left(m^{(i)}|x_{1:t}^{(i)}, z_{1:t}\right)$ (also found in Equation 2.25).

2.7 Robot Operating System

Robot Operating System (ROS)[1] is used for robotic software development and contains different software frameworks to help achieve complex and robust robot behavior. Some of the useful functionalities ROS provides when building a robot application is message-passing, device drivers and hardware abstraction. When creating a network with ROS, the way to go is to use the message-passing functionality where you have publisher/subscriber connections between the different processes, also called nodes. These nodes perform a specific task e.g. lidar driver or navigation. ROS helps solving complex problems by connecting many different already existing nodes.

The following sections will explain the communication, logging capabilities, how to use it and other relevant tools that the experiment of this thesis will take advantage of.

2.7.1 ROS Communication

ROS organizes the internal communication by using a graph structure. Each process is represented by one node in the graph, while the edges between the nodes, termed *topics*, handles the information that needs to be exchanged between the nodes. In ROS terms, this is called *message-passing*. A node is set to both receive an input and distribute an output to the system. When a node receives information from a topic, it *subscribes* to that topic. On the other hand, if a node advertises information to a topic it *publishes* to that topic. A system may contain a number

[1] `https://www.ros.org/`

of shared static or semi-static parameters. ROS handles these parameters through a *parameter server* which is a database consisting of those parameters. An example of a message is found in Listing 2.1.

```
# This is a message to hold data from an IMU

Header header

geometry_msgs/Quaternion orientation
float64[9] orientation_covariance

geometry_msgs/Vector3 angular_velocity
float64[9] angular_velocity_covariance

geometry_msgs/Vector3 linear_acceleration
float64[9] linear_acceleration_covariance
```

Listing 2.1: ROS Message for IMU

The main role for the ROS Master is to help the nodes in the system locate each other so they can start communicating peer-to-peer. The Master keeps track of every published and subscribed topic and by using this register the Master sets up the nodes that need to communicate. Newly initialized nodes register itself to the ROS Master with information about its publish and subscribe topics. ROS has a decentralized architecture which is shown in Figure 2.4.

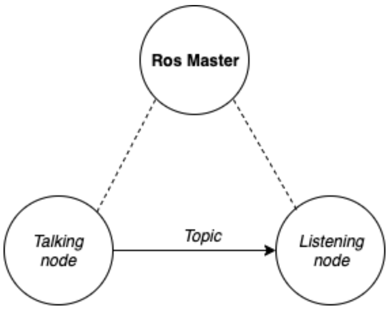


Figure 2.4: The Talking node publishing to the Listening node, while ROS Master is registering the communication.

2.7.2 ROS Launch

To easily launch multiple nodes at the same time ROS provides the package *roslaunch*. By gathering the nodes that is to be launched in a configuration file, or launch file, the launch package helps launching every specified node at the same time with one command. Additionally, it is possible to also set parameters in the parameter server for the ROS system by defining them in the launch file. The following command shows how launch files can be initiated.

```
$ roslaunch package_name file.launch
```

Listing 2.2: Launch command

2.7.3 ROS Topic

The ROS package *rostopic* is a useful tool to obtain control of the topics available and the data that the topic contains. This package offers different commands but the two functions this thesis uses is `rostopic list` and `rostopic echo`, which shows all the available topics and displays the messages on a specific topic, respectively. Following, are the two commands for initiating the tools

```
$ rostopic list
```

Listing 2.3: View published topics

and

```
$ rostopic echo /topic_name
```

Listing 2.4: Echo a published topic

2.7.4 Logging Data in ROS

A feature ROS provides is that ROS message data can be recorded and played back by using the ROS package *rosbag*. When a ROS system publishes topics, rosbag can log data from the topics into a file format called bags. This file can be played back in a manner that replicates the original data from the topic. To log data from a topic the following command is executed.

```
$ rosbag record /topic
```

By replacing `/topic` with `-a`, one can log every topic in the system. The below command plays back the file.

```
$ rosbag play file_name.bag
```

2.7.5 RVIZ

RVIZ is a 3D visualization tool in ROS. Here, topics can be visualized or the robot can be controlled. The following command lets the user view a specific topic

```
$ rosrun rviz rviz -f /topic_name
```

Chapter 3

Phylax

This chapter presents all the hardware and software of the robot. The robot will be referred to as *Phylax* and the whole platform as the *Phylax platform* or *ROS system*, from now on. Section 3.1 presents the chassis and motors used on Phylax. Section 3.2 provides the specifications of the motor controller and microcontroller, their functionality and the software implementation of the microcontroller. Finally, Section 3.3 describes the processing unit and the implemented software.

The work of building the Phylax platform, including both software and hardware, has been a project in cooperation between the author of this thesis and one of the supervisors, Joseph Piperakis. By looking at Figure 3.1, the reader will get an introduction to the architecture of the Phylax platform and Figure 3.2 shows the finished robot.

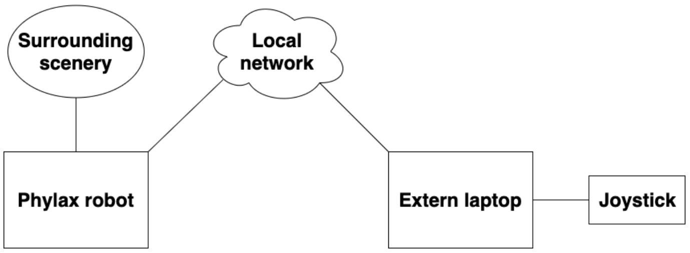


Figure 3.1: A primitive representation of how the Phylax platform is built up.

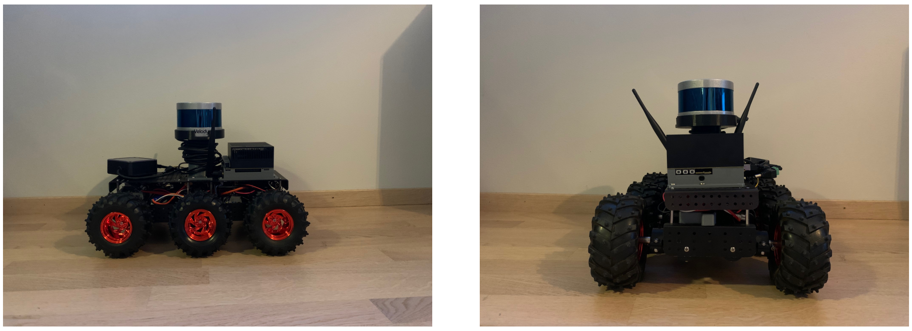


Figure 3.2: Phylax viewed from the front and side.

3.1 Chassis and Motors

The chassis, wheels and four of the motors are from DAGU Electronics and are easy to assemble. Phylax runs on a 6 Wheel Drive (WD) differential drive system where all the motors share the same power supply. The power supply is a two cell (2S) battery with 7.4V and is connected to a main discrete switch where the user can choose if the power should either be on or off. The switch also isolates the battery from the rest of the electronics to prevent damage. Each wheel has a rugged torsion suspension with rubber mounts that provides a flexible joint which makes it capable of driving on rough ground. Every wheel is attached to a motor shaft, that is connected to its corresponding gearmotor consisting of a high-power 6V brushed Direct Current (DC) motor combined with a 34:1 gearbox. In addition, each of the two middle motors are equipped with an integrated encoder embedded in the motor, which provides 1632.67 counts per revolution (CPR). Originally, the encoders only had a CPR of 48, but the gear ratio increases the CPR to approximately 1632 ($48 \times 34 = 1632$). Since these two motors replace the original middle motors, a hut that encapsulates the new motors needed to be designed and 3D-printed in order to connect the motors to the chassis and suspension (see the design in Appendix A). The chassis is made of steel and two compartments for battery housing. There are numerous holes where the necessary hardware can be fastened by screws both on the top and lower deck.

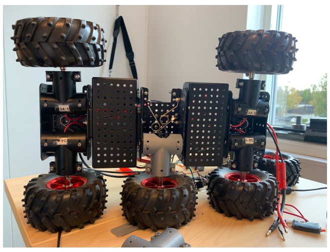


Figure 3.3: The bottom side of Phylax when one of the motors is removed due to reparations.

3.1.1 Assembling

First, the 6 motors were attached to the chassis at their correct places and the wheels were mounted on the output shafts of the motors. Each motor has at least two wires; one for battery voltage and one for grounding (+/- wires). In the middle of the chassis these wires were connected in a serial connection with the battery and the power switch (see Figure 3.4). In addition, the two motors with the encoders have four extra wires, where two wires supply the encoder with power and the other two provide the encoder output. These four wires were all connected to a microcontroller that both supplies power and reads the encoder output. For the wheels to behave as the user commands all six motors were connected to motor controllers that control the motors. These motor controllers are located in the back of Phylax, and control three motors each. In other words, one microcontroller controls the motors on the left side and the other one the right side. One motor controller includes a microcontroller, a filter and power controller. A detailed explanation of the microcontrollers can be found in Section 3.2.

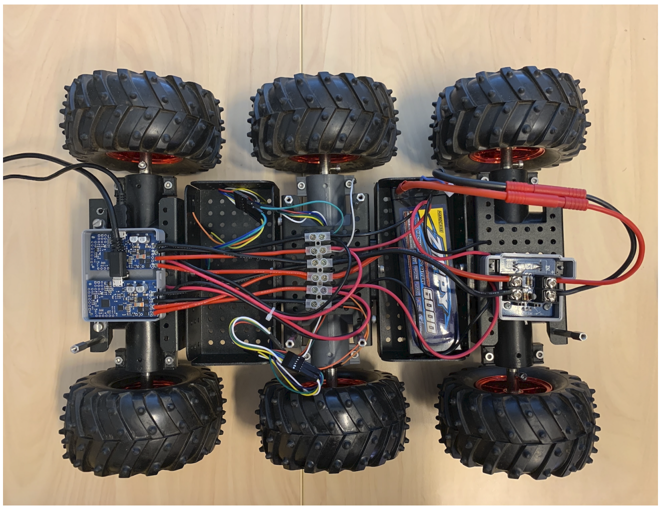


Figure 3.4: The top side of the bottom chassis including most of the electronics and one battery.

3.2 Microcontrollers

Phylax has two motor controllers that control the wheels on each side of the vehicle, and one microcontroller that distributes odometry and IMU data to the ROS system. This section will provide the specifications of the motor controller and microcontroller, their functionality and the software implementation of the microcontroller.

3.2.1 Motor Controller

As mentioned in the previous section, there are two motor controllers that control the motors on each side of the vehicle. These motor controllers are Pololu SMC G2 and are pre-programmed from the factory. However, in order to integrate the controllers into the Phylax platform the interface between the given commands and the motor controllers needed to be implemented. The motor controllers receive their commands through a Universal Serial Bus (USB) connection wired to the processing unit. Here, a running ROS node sends commands straight to the motor controllers. The software for the processing unit will be presented in Section 3.3.1.

3.2.2 Microcontroller for Odometry and IMU

To calculate and distribute the odometry and IMU data to the rest of the ROS system a Sparkfun ESP32 microcontroller is installed and will be referred to as ESP32 from now on. The IMU data is retrieved from the Sparkfun 9DoF IMU Breakout - ICM-20948 which is wired straight into the microcontroller. This is an IMU with 9 Degrees of Freedom (DoF); 3-axis gyroscope, 3-axis accelerometer and 3-axis magnetometer. In addition to the 9DoF the IMU also includes a Digital Motion Processor (DMP) that offloads the computation of motion sensing and calculates the quaternions. The IMU is placed in the center of Phylax to obtain minimum transformations to avoid potential errors that might propagate. The ESP32 is also wired to the encoders on both of the middle motors.

Software

The ESP32 is compatible with the Arduino platform which provides board definition and libraries. The software on the ESP32 is managed through the Arduino interface on a laptop and uploaded to the board through a USB cable. All the software is programmed in C++. Before any coding is done the correct library environment needs to be set. To make the IMU talk to the ESP32 the *Sparkfun 9DoF IMU Breakout* library is required. Figure 3.5 presents the subsystem that handles IMU and odometry data.

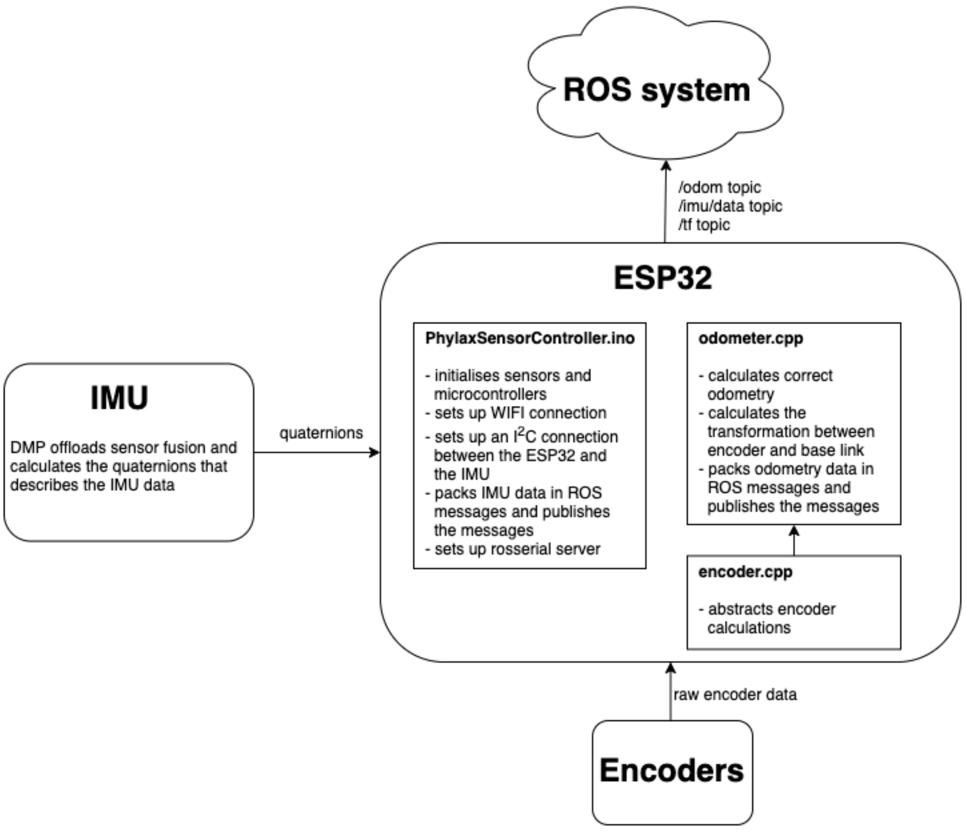

ROS system
/odom topic
/imu/data topic
/tf topic
ESP32
PhylaxSensorController.ino
- initialises sensors and microcontrollers
- sets up WIFI connection
- sets up an I^{2}C connection between the ESP32 and the IMU
- packs IMU data in ROS messages and publishes the messages
- sets up rosserial server
odometer.cpp
- calculates correct odometry
- calculates the transformation between encoder and base link
- packs odometry data in ROS messages and publishes the messages
encoder.cpp
- abstracts encoder calculations
IMU
DMP offloads sensor fusion and calculates the quaternions that describes the IMU data
quaternions
raw encoder data
Encoders

Figure 3.5: A representation of the subsystem with both software and hardware handling odometry and IMU data.

The DMP on the IMU processes the measurements and calculates the quaternions before the raw IMU data is offloaded to the ESP32. Here, the data is retrieved and processed in `PhylaxSensorController.ino` where the IMU data is packed in ROS messages and distributed to the ROS system under the topic */imu/data*. The file contains specific ROS implementations, WIFI setup and quaternion calculation implemented specific for this project. However, a few libraries from the sensor manufacturer are utilized and `PhylaxSensorController.ino` is therefore not included in the Appendices. It is important to state that the file is still unique for this project.

`PhylaxSensorController.ino` also initializes the sensors and sets up an Inter-Integrated Circuit (I^2C) connection between the ESP32 and the IMU. In addition, a Wireless Fidelity (WIFI) connection to the local network is created. This is due to the rosserial library implementation used on the ESP32 that does not support a USB connection, hence a WIFI connection is needed to be able to publish different ROS topics. Furthermore, the `encoder.cpp`, found in Appendix C.1, extracts the raw encoder data from the encoders, so that the algorithm in `odometer.cpp` can calculate the correct odometry, pack it in ROS messages and publish it to the ROS system under the */odom* topic. The odometer file also computes the transformation between the coordinate frame of the encoders and the coordinate from of Phylax, called *base_link*. This transformation is published under the */tf* topic. `odometer.cpp` is presented in Appendix C.2.

3.3 Processing Unit

The processing unit on Phylax is the Nvidia Jetson AGX Xavier and is mounted on the top of the chassis. The Xavier is equipped with a 512-core GPU and an 8-core ARM v8.2 64-bit CPU. The developer also possesses numerous module interfaces, but the interfaces used in this thesis is the Gigabit Ethernet, USB and High Definition Multimedia Interface (HDMI). It also supports neural network. The power source to provide the Xavier and other accessory with power is a four cell (4S) LiPO battery at 14.8V. To mount the Xavier on the chassis, a processing unit housing was designed and 3D-printed (see Appendix A) for optimal and robust installation. The width, length and height of the Xavier is $105mm \times 105mm \times 65mm$, respectively. Figure 3.6 shows the Xavier and housing.

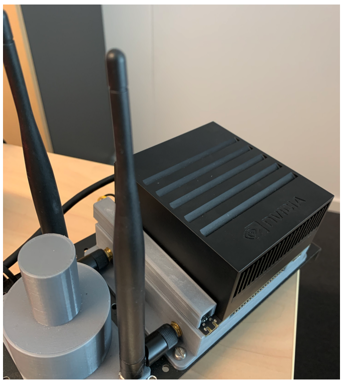


Figure 3.6: Nvidia Jetson AGX Xavier with antennas on the chassis. The gray part underneath the Xavier is the 3D-printed housing.

The processing unit is responsible for the majority of the ROS software through the `Phylax` repository. It also runs the software for the motor controllers, which is a ROS node running `PololuController`. This section will thoroughly explain the contribution of the processing unit to the Phylax platform by analyzing its software. To specify, the `Phylax` repository is implemented by the author and supervisor with inspiration from opensource examples and will therefore be explained, while

`PololuController` is purely Mr. Piperakis work and will therefore not be described in detail. The ROS opensource examples were used as a basis for writing the software.

3.3.1 Software

An analysis of the Phylax repository will now be presented to highlight the influence of this software on the Phylax platform. The implemented ROS packages `phylax_description`, `phylax_control`, `phylax_base` and `phylax_navigation` will be the center of attention during this section.

Description of Phylax

The algorithms for obtaining autonomous driving, for example SLAM, needs information about the state of the robot and the environment that the robot operates in. Therefore, a model of the robot with the necessary parameters was constructed in *Unified Robot Description Format* (URDF) (see Appendix D.2, `phylax.urdf.xacro`), which is located in the `phylax_description` package. The URDF is an XML format for representing the robot model in ROS. An XML macro language, called `xacro`, was utilized to make the XML files more generic.

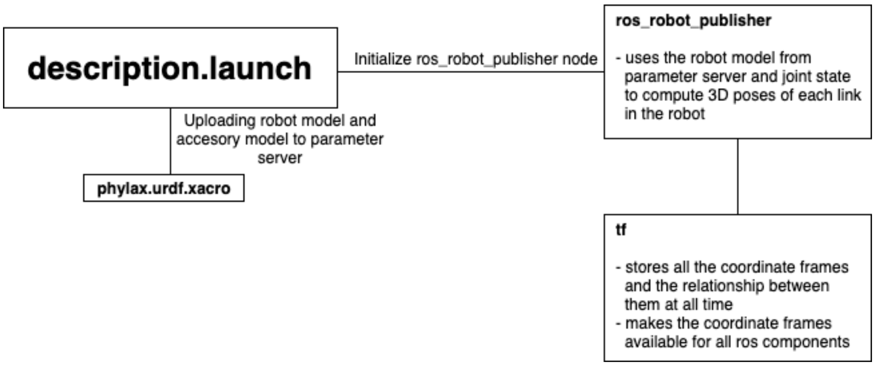

description.launch
Initialize ros_robot_publisher node
Uploading robot model and accesory model to parameter server
phylax.urdf.xacro
ros_robot_publisher
- uses the robot model from parameter server and joint state to compute 3D poses of each link in the robot
tf
- stores all the coordinate frames and the relationship between them at all time
- makes the coordinate frames available for all ros components

Figure 3.7: A representation of description.launch

By executing the description launch file, the URDF model of the robot is parsed and its definitions are uploaded to the ROS parameter server. The launch file also initializes a ROS node called *robot_state_publisher*, which publishes the state of the robot to a transformation library. The robot_state_publisher constructs a kinematic model of the robot internally, and by using the position of the joints it computes and publishes the forward kinematics of the robot under the ros topic *tf*. The joint states are retrieved from a separate node called *joint_states*. Here, all the different coordinate frames of Phylax and the relationship between them are stored and become available for all the ROS components at any time. The different frames are defined in Table 3.1.

Name of frame	**Symbol**	**Description**
base_link	b	main frame located at the bottom of the chassis
odom_right	o_r	local odometer frame for right center wheel
odom_left	o_l	local odometer frame for left center wheel
imu_link	i	local IMU frame
velodyne	v	local lidar frame

Table 3.1: Definition of frames on Phylax.

To visualize the different frames on the robot model, ROS Rviz is used. By creating the model of the robot and retrieving the robot model from the description files the different links can be presented and viewed in the following figures. The red axis represents the x-axis, the green axis represents the y-axis and the blue axis represents the z-axis. All the frames follow the right-hand rule (see Figure 3.8 to 3.11). The relationships between the frames are calculated in the *tf* node by using the transformation matrix 2.5 given in Section 2.1. In this project every frame needs to be transformed to frame b, see Figure 3.8. The two frames i (Figure 3.10) and v (Figure 3.11) only need to be translated with no rotation. Hence, $\mathbf{R}^{to}_{from} = \mathbf{I}$ which yields

$$\mathbf{T}^{to}_{from} = \begin{bmatrix} \mathbf{I} & \mathbf{t}^{to}_{from} \\ \mathbf{0} & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & x_t \\ 0 & 1 & 0 & y_t \\ 0 & 0 & 1 & z_t \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.1}$$

where x_t, y_t and z_t are replaced with the actual distance between the two frames. As an example, the transformation matrix from v to b would become

$$\mathbf{T}^{b}_{v} = \begin{bmatrix} \mathbf{I} & \mathbf{t}^{b}_{v} \\ \mathbf{0} & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -0.2 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.2}$$

where the translation is -0.2 meters on the z-axis.

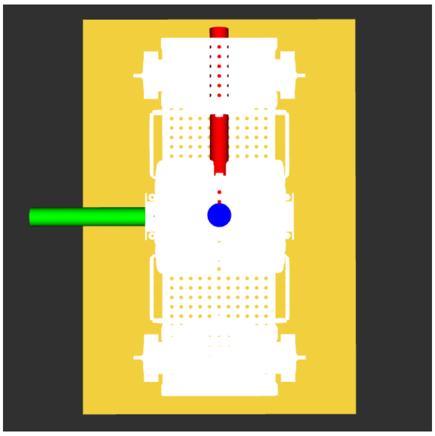
The two remaining frames o_r and o_l, however, need to be both translated and rotated. The frames are found in Figure 3.9. The rotation is a simple rotation around the the y-axis. Hence, Equation 2.3 from Section 2.1 becomes

$$\mathbf{R}^{to}_{from} = \mathbf{I}\mathbf{R}_{y,\theta}\mathbf{I} = \mathbf{R}_{y,\theta} \tag{3.3}$$

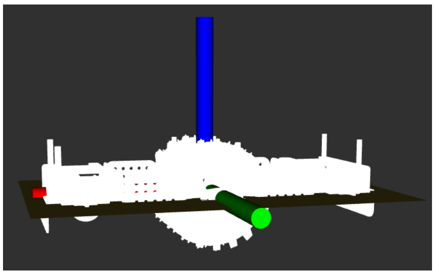
which yields

$$\mathbf{T}^{to}_{from} = \begin{bmatrix} c\theta & 0 & s\theta & x_t \\ 0 & 1 & 0 & y_t \\ -s\theta & 0 & c\theta & z_t \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.4}$$

where x_t, y_t and z_t are replaced with the actual distance between the two frames and θ is the rotation angle.

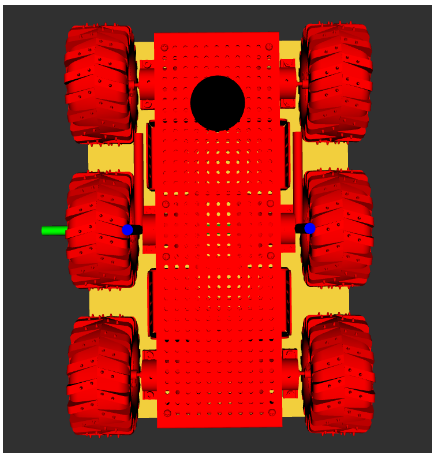


(a) The b-system viewed from the top.

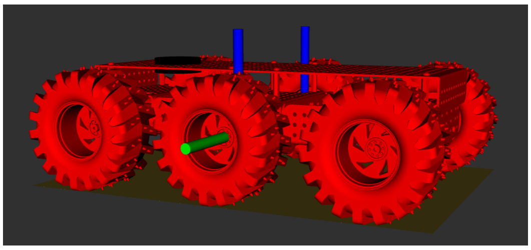


(b) The b-system viewed from the side.

Figure 3.8: Visualization of the *base_link* frame.

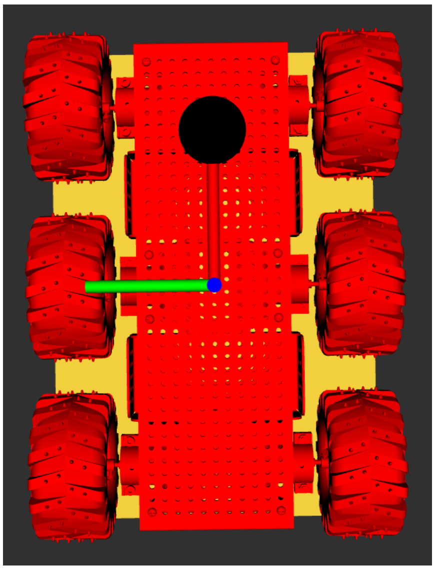


(a) The o_r-system and the o_l-system viewed from the top.



(b) The o_r-system and the o_l-system viewed from the side.

Figure 3.9: Visualization of the two odometer frames.


(a) The *i*-system viewed from the top.

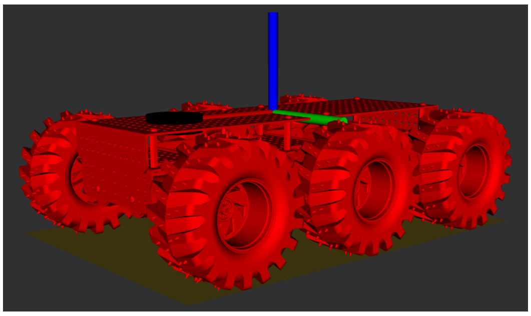
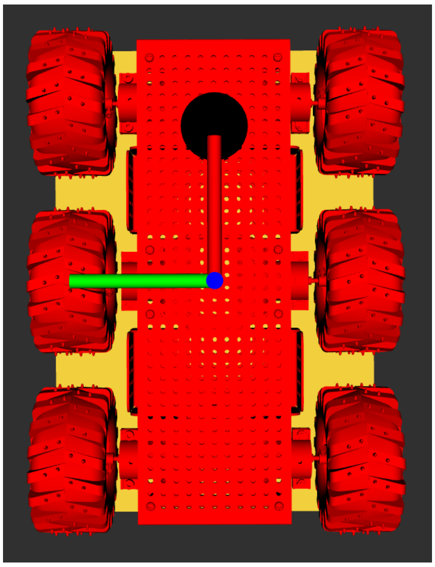

(b) The *i*-system viewed from the side.

Figure 3.10: Visualization of the *imu_link* frame.


(a) The *v*-system viewed from the top.

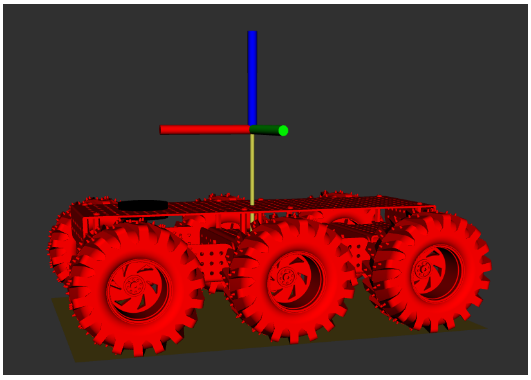

(b) The *v*-system viewed from the side.

Figure 3.11: Visualization of the *velodyne* frame.

Control of Phylax

The software for controlling Phylax is implemented in the ROS package `phylax_control`. The main launch files in this package, `teleop.launch` and `control.launch`, can be found in Appendix E.1 and E.2, respectively. They have two different areas of responsibility, where the first one abstracts the joystick or keyboard control and the other one implements a differential drive with a Proportinal-Integral-Derivative (PID) controller. The `teleop.launch` reads the profile of the joystick from the `teleop_ps4.yaml`, found in Appendix section E.3. The *joy_node* abstracts the driver of the joystick and publishes the states of the various controls found on the joystick and distributes it to the ROS system. Furthermore, the *teleop_twist_joy* node is responsible for converting the joystick messages into twist messages which can be fed straight into the differential drive node. The twist message contains linear and angular velocity commands based on the joystick commands. In addition, the *interactive_marker_twist_server* node is included in the launch file in order to allow the user to control Phylax through rviz. The teleop launch file, visualized in Figure 3.12, is only used when manual driving is required. For example, when gathering data.

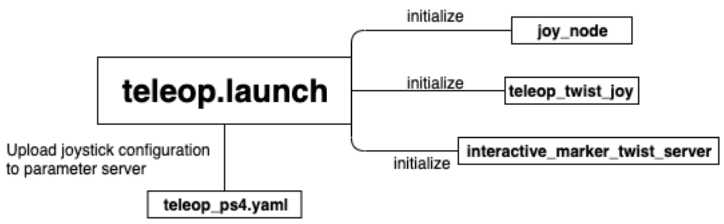


Figure 3.12: A representation of teleop.launch

A multiplexer is further implemented in the `phylax_control` package for all input controllers, which is visualized in Figure 3.13. The multiplexer node *twist_mux* takes in several inputs and prioritizes and selects the correct output to the motor controllers when needed in autonomous driving. This node retrieves multiplexer parameters from the `twist_mux.yaml` (see Appendix E.6) and uploads it to the parameter server. The PID controllers are started through the node *controller_manager*. The chosen controllers, which in this case are the joint state controller and the differential drive controller, including their parameters are defined in `control.yaml` (see Appendix E.4). In addition, the *robot_localization* node is initialized to provide nonlinear state estimation through sensor fusion (EKF). The parameters for this node can be found in `robot_localization.yaml`, Appendix E.5.

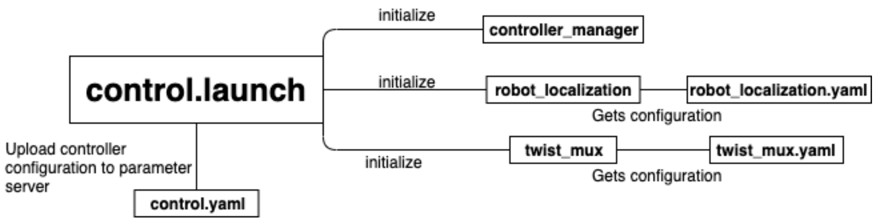


Figure 3.13: A representation of control.launch

Base of Phylax

The `phylax_base` package contains the control of the hardware and is implemented with two launch files for the user to choose. For a fully autonomous drive one should use `phylax.launch`, while `phylax_simple.launch` is a more basic implementation that does not include any controllers for the robot. These are located in Appendix section F.1 and section F.2, respectively. This section will present the two launch files and their area of use.

The `phylax.launch` includes `description.launch` and `control.launch` which are described previously in this section (Section 3.3.1). The former launch file publishes transformations between the various coordinate frames found on the robot, while the latter launch file introduces a differential drive controller (PID) on the left wheels and right wheels. This becomes a closed loop system where the motor sensors are read and compensated for accordingly. Also found in `phylax.launch` are `VLP16_points.launch` (supplied by the lidar manufacturer) and `point2laser.launch`. The latter launch file is in the *phylax_description* package and was primarily implemented to do the conversion between 3D and 2D scans and publish the lidar transformation (see Appendix section F.3).

Furthermore, the node *phylax_node* is initialized with the purpose of publishing and updating the states of the robot (see `phylax_base.cpp` and `phylax_hardware.cpp`, found in section F.4 and section F.5 respectively). The former spawns a thread that reads the position values from the robot joints position. The latter file defines the joints present on the vehicle and the way that they are read and updated. Two more ROS nodes, *rosserial_server* and *rosserial_python*, are introduced. These nodes contain an implementation of a host-side *rosserial* connection, which is a protocol for wrapping standard ROS serialized messages and multiplexing multiple topics over a serial port or network socket. The *rosserial_server* contains the implementation and handles setup, publishing and subscribing for a connected rosserial device, where *rosserial_python* node works as an aid for handling subscriptions. In order to be able to send messages between boards connected through a serial interface one needs to serialize and deserialize the messages. The *pololu_driver* package receives drive messages and converts them into a form required by the motor controllers. Figure 3.14 shows the architecture of `phylax.launch`. It is important to mention this launch file was not finished, due to the fact that the differential drive controller was not debugged yet.

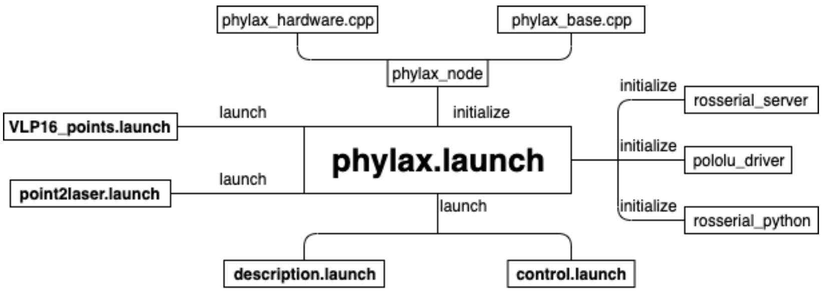


Figure 3.14: A representation of phylax.launch

An alternative is the simpler launch file `phylax_simple.launch`, which only implements differential drive. Similarly to `phylax.launch` the two nodes *rosserial_server* and *rosserial_python* are initialized and `VLP16_points.launch` and `point2laser.launch` are launched. The *simple_joy_node* takes joy messages from the joystick, implements the differential drive and produces messages suitable for the motor controller (see Appendix section F.6).

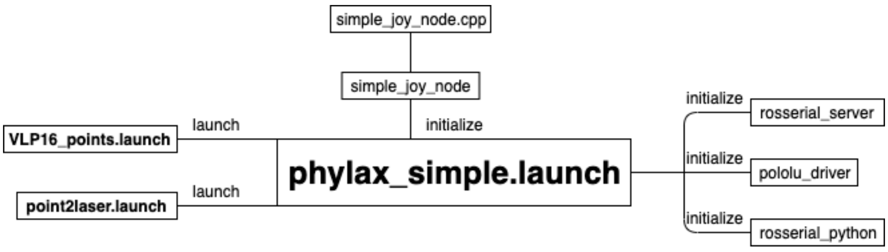


Figure 3.15: A representation of phylax_simple.launch

Navigation of Phylax

Navigation in Phylax is implemented using the standard ROS navigation package, hereby called `phylax_navigation`. A ROS implementation of GMapping with a custom configuration is utilized, see Appendix section G.1. The launch file takes in the already processed and transformed lidar and odometry data and estimates a map of the surroundings, see Figure 3.16. The theory behind GMapping is presented in Section 2.5.

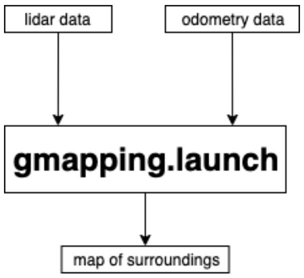


Figure 3.16: A representation of gmapping.launch

Chapter 4

Method

This chapter establishes the methodology for developing Phylax and the SLAM system. Section 4.1 describes different design choices and the debugging of the software. Section 4.2 gives an introduction to the SLAM system and the experiment where the datasets were conducted.

4.1 Phylax

This section presents different design choices and the debugging of the software.

4.1.1 Design choices

In both the hardware and software part of the Phylax platform different design choices were made to increase the performance of the platform and avoid non-technical problems during the process. The factors that were heaviest weighted were price, delivery time and documentation. Choices made for both hardware and software are discussed below.

Hardware

The ESP32, motor controllers, IMU and motors/encoders chosen for this project are all off-shelf products that are not designed for rugged use. These products were selected due to low price, fast delivery, comprehensive documentation and ready made software examples. In addition, the ESP32 was chosen due to the fact that the rosserial implementation on the ESP32 only supports Transmission Control Protocol (TCP) and it was the smallest available board with an extremely fast processor. An important part of the microcontrollers job is to publish data from the encoders and IMU to the rest of the system. Hence, a WIFI connection is needed. The ESP32 also worked as a power supply for the encoders. In theory the IMU could get its power from the same battery as the rest of the motor. The motors require significant and rapid changes of current from the battery. This would have created a lot of noise and spikes in the encoder counting and would have resulted in significant errors.

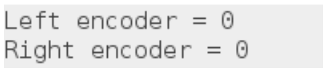
On the other side of the price scale is the VLP16 lidar and the Xavier. These are two very sophisticated components. Using the VLP16 lidar ensured a proper quality on the dataset. With correct use this lidar can gather a trustful and robust dataset. The Xavier provides large computational power that will be important for the future development of the platform. In this project the generous amount of computational power has not been in focus or needed, but the fact that it also supports neural network is considered very relevant for having the opportunity to add advanced features.

Software

The foundation of the software platform is built upon the first version of ROS. The reason ROS was chosen as the framework for this thesis was its already existing SLAM implementations and use-case area of real-time application. Besides the fact that ROS is meant for robotic software, having access to informative documentation and tutorials was considered valuable. ROS also provides numerous opensource tools and packages for the user to include in the software. Another strong argument for using ROS is the support for debugging. ROS topic offers necessary support for debugging the system.

4.1.2 Debugging

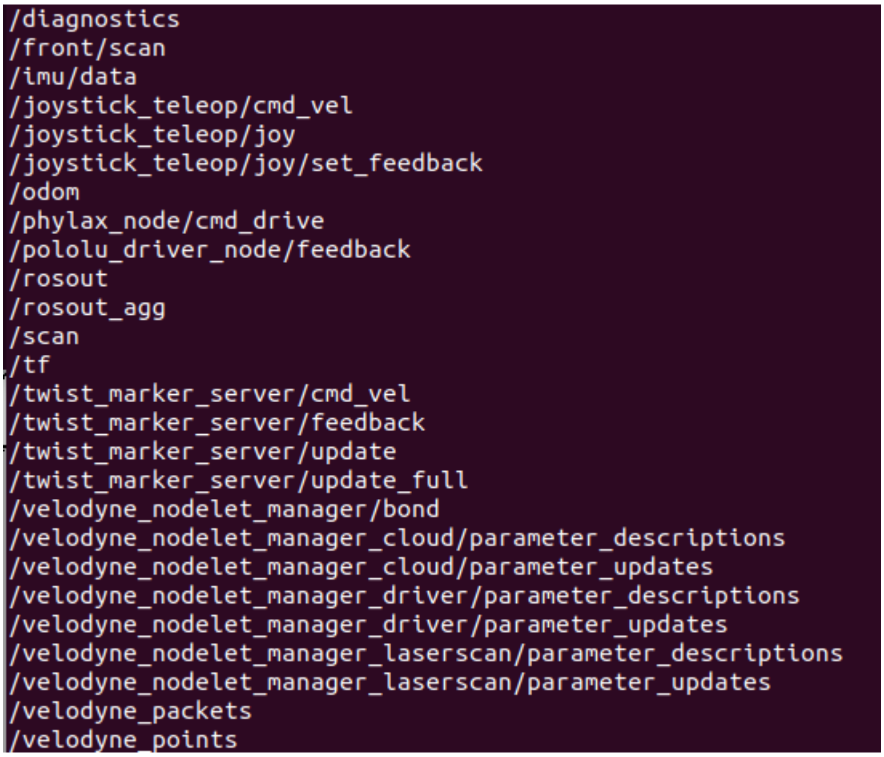
Debugging on the Arduino platform is mainly done by print-statements which appear on a monitor. The most challenging part of the implementation on the ESP32 was getting the odometry correct. Having print statements that present the left and right encoder count was useful for debugging the commands from the joystick and making sure the encoders behaved as expected. When driving on a straight line the encoder counts were expected to count the same number on both the left and right side. Figure 4.1 presents the print statements in the Arduino monitor. In addition, the calculation of odometry was implemented on the Arduino platform. To be able to trust the odometry calculations the path distance, seen at line 68 in Appendix section C.2, was printed (see line 70). The path distance takes into account the movement along the x-axis, y-axis and the orientation.



```
Left encoder = 0
Right encoder = 0
```

Figure 4.1: Presentation of encoder counts for debugging purposes.

The odometry topic, like all other ROS topics, can be echoed by taking advantage of the rostopic package, explained in Section 2.7.3. By executing the command in Listing 2.3, a list of all the available topics in the system is presented. This is useful when testing if nodes being implemented actually publish the topics they are expected to publish. Figure 4.2 presents a list of topics published on the Phylax platform using the `phylax_simple.launch`.



```
/diagnostics
/front/scan
/imu/data
/joystick_teleop/cmd_vel
/joystick_teleop/joy
/joystick_teleop/joy/set_feedback
/odom
/phylax_node/cmd_drive
/pololu_driver_node/feedback
/rosout
/rosout_agg
/scan
/tf
/twist_marker_server/cmd_vel
/twist_marker_server/feedback
/twist_marker_server/update
/twist_marker_server/update_full
/velodyne_nodelet_manager/bond
/velodyne_nodelet_manager_cloud/parameter_descriptions
/velodyne_nodelet_manager_cloud/parameter_updates
/velodyne_nodelet_manager_driver/parameter_descriptions
/velodyne_nodelet_manager_driver/parameter_updates
/velodyne_nodelet_manager_laserscan/parameter_descriptions
/velodyne_nodelet_manager_laserscan/parameter_updates
/velodyne_packets
/velodyne_points
```

Figure 4.2: List of topics published.

Furthermore, running the command in Listing 2.4 will echo the data on a specific topic. Again using odometry as an example, echoing the topic will both help for debugging and calibration of the odometry. To confirm that the odometry implementation and the encoders work properly so the odometry message published to the system is correct a final calibration test needs to be conducted. The first part of the test is putting a measuring scale on the ground next to Phylax and driving the vehicle a certain distance on the scale, see Figure 4.3.

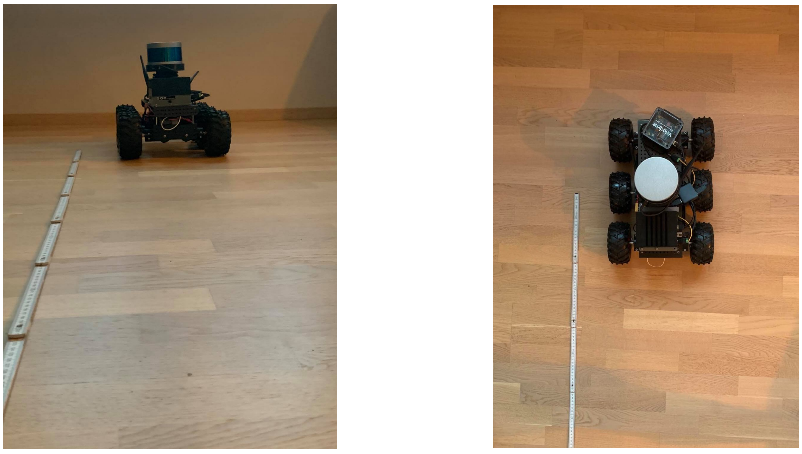
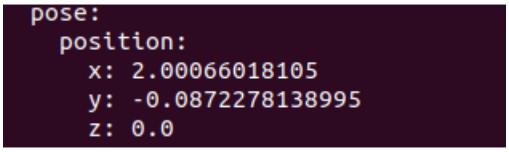


Figure 4.3: Calibration test of odometry.

If the traveled distance align with the output on the odometry topic the test is successful and the odometry can be concluded to be correct. Figure 4.4 shows the final odometry message when the vehicle was located 2 meters in front of and 0.09 meter to the left of the original start point.



```
pose:
  position:
    x: 2.00066018105
    y: -0.0872278138995
    z: 0.0
```

Figure 4.4: Screenshot of the message from the odometry topic.

The joystick commands and the vehicle's behavior are expected to be in analogy. The method used for debugging this part is comparing the physical behavior with the joystick input. In Appendix E.3 each button on the joystick was given a variable name based on the what happened when the particular button was pressed. The same method was used when confirming that the data in the messages sent to the motor controller driver were correct. Comparing line 52 and 53 in Appendix section F.6, line 52 includes a negative sign in the equation. By systematically tracking the response of Phylax while giving certain commands from the joystick this bug was discovered. By not having a negative sign the vehicle would drive in circle. This is because from the motors perspective they were going in the same direction but when operating on opposite sides this was not the actual result.

4.2 The Experiment

This section describes the SLAM system and the conducted experiment, which is purely the work of the author.

4.2.1 System Architecture of the Phylax SLAM System

The Phylax SLAM system is designed to be used in conjunction with the lidar and encoders of Phylax. The system generates a 2D map of the surrounding scenery. The architecture of the system is presented in Figure 4.5 comprising of: the lidar and the encoders, Phylax that includes the Xavier and ESP32 that process and publish data, an external laptop where the topics are logged and played and GMapping node that generates a 2D obstacle map as an outcome. Note that the system only utilizes odometry and lidar, while IMU is not needed for this experiment. The laptop used for the experiment runs on Ubuntu 18.04. The software and hardware used in this system are explained in Chapter 3.

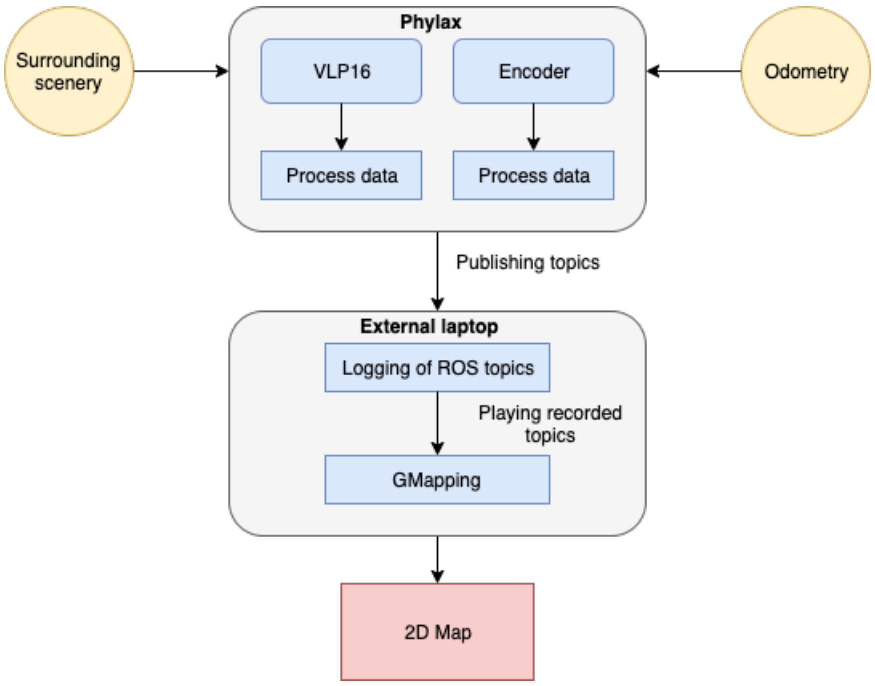


Figure 4.5: System architecture.

4.2.2 Obstacle Course Experiment

The obstacle course experiment was conducted in order to test the capabilities of the Phylax SLAM system in estimating a map to be used for autonomous navigation in terrain. The testing was performed on two different courses: one clean room inside with nothing but walls and a self-made obstacle course in the same room. The lighting conditions were daylight and no direct sunlight pointing directly at Phylax or the lidar. The test data was collected in ROS bags on the external laptop, while the runs of the Phylax SLAM system were conducted playing back the rosbags. The setup for the two environments are further explained in the next section.

4.2.3 Environmental Setup

The dataset from the clean room represented the ground truth for the estimated map and were used for tuning the algorithm, while the second dataset from the obstacle course were used as the test for GMapping's capabilities. The two next sections will present the surroundings Phylax operated in.

Clean Room

The ground truth dataset was conducted in a square room with a smooth and straight ground. The source of light was a window that provided sufficient daylight. Figure 4.6 shows the outlook of the room with corresponding dimensions. It is worth noticing that the bottom long wall was made out of sheets which did not act as a straight and smooth wall, like the other three walls. The simplicity of the environment was chosen due to the fact that GMapping works well in that kind of environment so the ground truth would be trustworthy. Also, tuning the algorithm on a known working dataset is a big advantage.

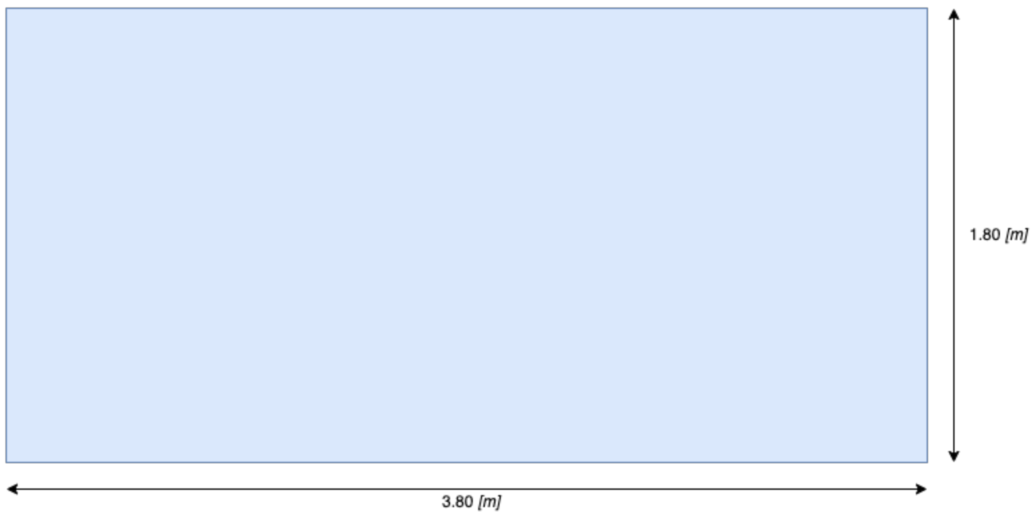


Figure 4.6: Overview of the clean room.

Obstacle Course Room

Before the main dataset could be collected an obstacle course in the same room needed to be designed. To represent different shapes that may occur in the terrain a box, plants, a candlestick and an incline was placed inside the room. The obstacle course is represented in Figure 4.7, the sizes of the obstacles are listed in Table 4.1 and actual figures of the course are found in Figure 4.8.

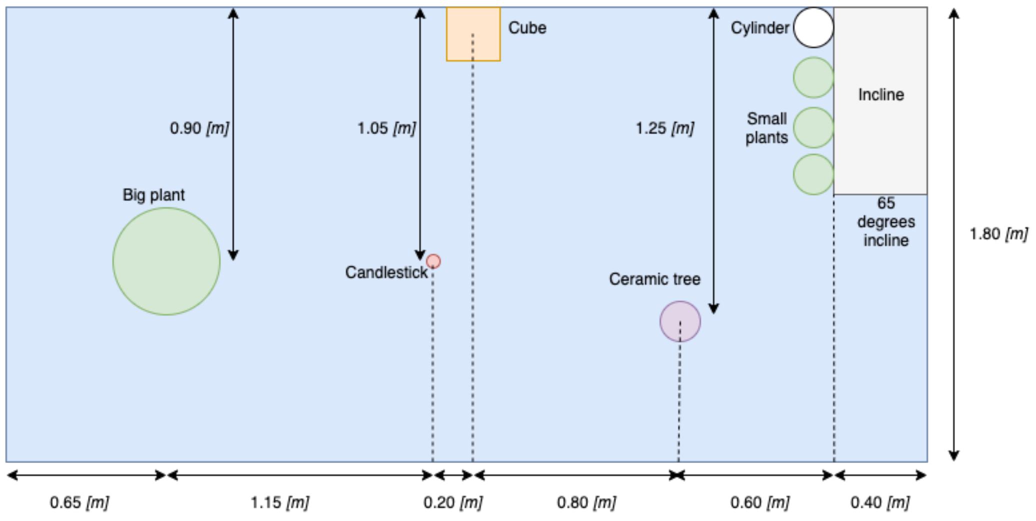


Figure 4.7: Overview of the obstacle course.

The idea behind the course is to include arbitrary shapes and sizes to detect how GMapping responds to a more complex environment. Since GMapping generates a 2D map it was of interest to investigate how GMapping would handle some sort of slope or incline. Therefore, a slope was included in the course with an incline of 65°.

Obstacle	**Size** [m] ***(what is measured)***
Big plant	0.5 *(diameter)*
Candlestick	0.01 *(diameter)*
Cube	0.27 x 0.27 x 0.27 *(height x width x length)*
Ceramic tree	0.2 *(diameter)*
Cylinder	0.2 *(diameter)*
Small plants	0.2 *(diameter)*
Incline	1.15 x 0.5 *(width x length)*

Table 4.1: Measurements of obstacles.

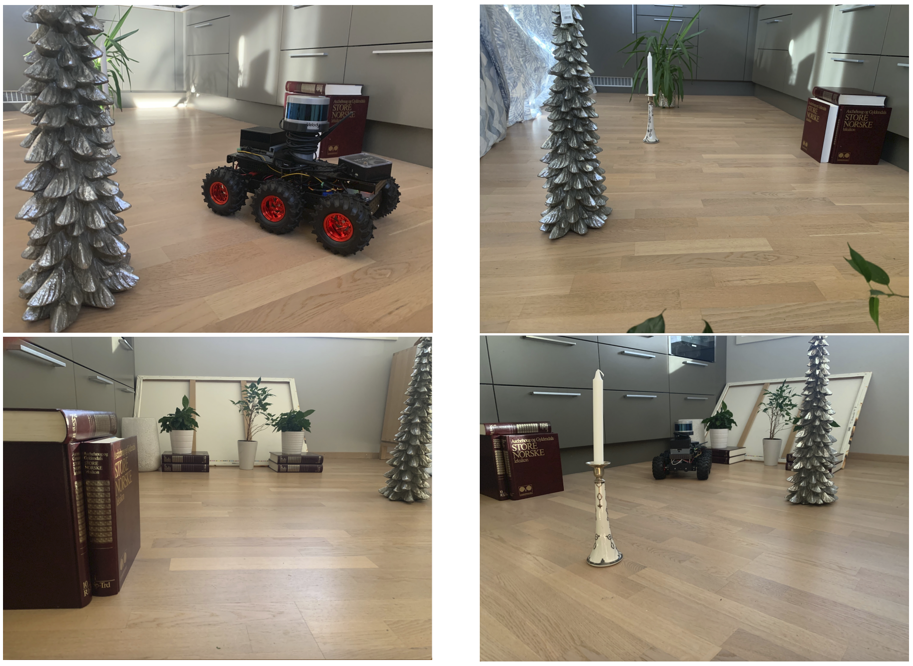
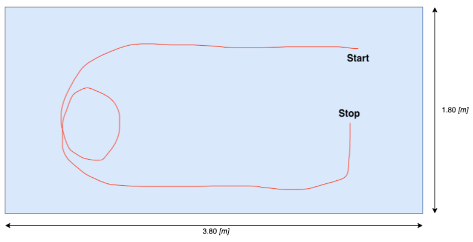


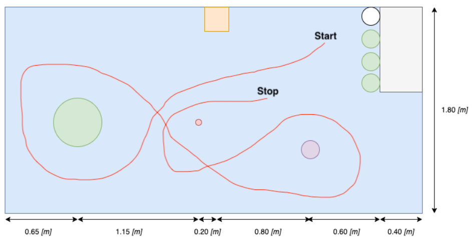
Figure 4.8: Actual obstacle course.

Trajectories

There were two guidelines that dictated where the trajectory of the vehicle should be. First of all, making sure every part of the course was scanned was the number one priority. Including loop closures, or making sure the vehicle returns to a previously visited location, was important due to testing GMappings capabilities when scanning surroundings twice. For each of the courses the vehicle drove about 1 meter before stopping for 2 seconds to make sure the environment was properly scanned. Below are the trajectories in their corresponding surroundings, see Figure 4.9.



(a) Clean room.



(b) Obstacle course.

Figure 4.9: Trajectories for the two datasets taken starting in the upper right corner of the trajectory.

4.2.4 Collecting the Datasets

The next step is gathering the datasets for the SLAM system to use for generating maps. To begin with, the joystick needs to be wired to the external laptop, while Phylax and the router need to be powered up. It is important to make sure every device is connected to the same network: *Lone_Wolf_Local*. As mentioned, there are two `phylax_base` launch files that are for the user to choose. In this experiment the `phylax_simple.launch` is used. The biggest argument for this is that `phylax.launch` is unfinished and the extra performance is not needed in this experiment.

Since all the devices are on the same local network Phylax can be accessed through the external laptop using Secure Shell (SSH) using the following command

```
$ ssh joseppi@arthur
```

where `joseppi` is the username used on the Xavier and `arthur` is the Internet Protocol (IP) hostname used for simplicity instead of the actual IP address. All the software on Phylax and the laptop are located in a catkin workspace which is used for building and modifying catkin packages, also known as ROS packages in this project. Therefore, every time a bash terminal is opened, either on the external laptop or Xavier, the catkin workspace should be entered and made sure it is overlayed with the setup script. Given that the software is built, this is done by executing the two following commands

```
$ cd catkin_workspace
```

and

```
$ source devel/setup.bash
```

The next step is launching the `phylax_simple.launch` file on Phylax. This happens in the terminal that has access to Phylax by running

```
$ roslaunch phylax_base phylax_simple.launch
```

In the `.bashrc` file, which is a file that gets executed every time a bash terminal is opened where variables and scripts the user want executed is put, Phylax is defined as the ROS master. So when `phylax_simple.launch` is launched, the Xavier also initializes and becomes the ROS master for the entire system. Furthermore, on the external laptop the launch file for controlling Phylax is executed.

```
$ roslaunch phylax_control teleop.launch
```

On the external laptop navigate to the directory where the rosbags should be saved and execute the command

```
$ rosbag record -a
```

Now, Phylax is ready to collect data. After the necessary data is collected the processes are terminated and Phylax is powered off. In this project the rosbags are played back and mapped afterwards to easily tune GMapping, but the mapping could also be happening simultaneously when scanning the environment. To play back the bags and estimate the map the three following commands are needed. First, since the ROS master on Phylax is terminated the `.bashrc` file needs to be updated so the external laptop works as the master. Then the GMapping node is launched which also starts the laptop as the ROS master by running

```
$ roslaunch phylax_navigation gmapping.launch
```

Second,

```
$ rosbag play <your bag>
```

plays back the rosbags and publishes the topics, so the GMapping node can estimate the map. When the bag is done playing and the map is estimated

```
$ rosrun map_server map_saver -f <name of map>
```

saves the map in a `.png` file. This process can be repeated as the algorithm gets tuned by the user. The rosbags contains all the topics in the system but GMapping only uses odometry, transformations and lidar data to generate the map. The theory behind the estimation is explained in detail in Section 2.3 through 2.5.

Tuning of GMapping

In Appendix B all the parameters available for tuning along with their representative default values are found. The first map will be generated based on the default values before the two parameters: `particles` and `resamplingThreshold` will be tuned for optimized performance. After optimizing those two parameters the rest of the parameters are tuned based on systematic trying and failing. For every map generated the parameter values are registered and a comment on the quality of the map is written down. Some experimental tuning will also be conducted on the obstacle course dataset.

Chapter 5

Results

This chapter presents the estimated maps from the two datasets. Section 5.1 and Section 5.2 presents the maps from the clean room and the obstacle course, respectively. The default parameters are defined in Appendix B, while the tuned parameters are defined in Table 5.1.

Parameter Value	**Value**
`map_update_interval`	0.5
`resampleThreshold`	0.1
`lsigma`	0.0075
`linearUpdate`	0.1
`srr`	0.01
`srt`	0.02
`str`	0.01
`stt`	0.02
`delta`	0.04

Table 5.1: Tuned parameters and their value.

5.1 Clean Room

This section presents the maps from the clean room. All the maps are estimated on the same dataset.

5.1.1 Default Parameters

In Figure 5.1, the map generated from the first trial test is shown. One can see how the default parameters yield relatively good results although clearly with some possibilities for increasing the resolution which is set to $0.05[m]$ per grid cell. The uneven features on the right hand side show how the curtain, which in fact does represent an uneven wall, is captured by the mapping algorithm.

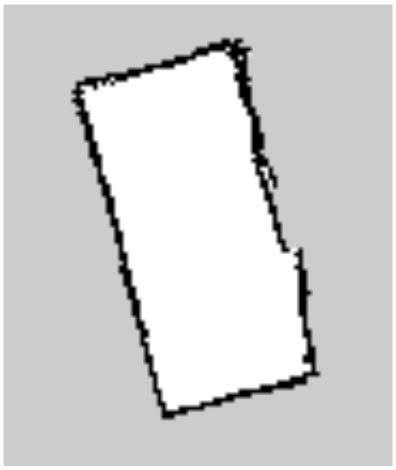
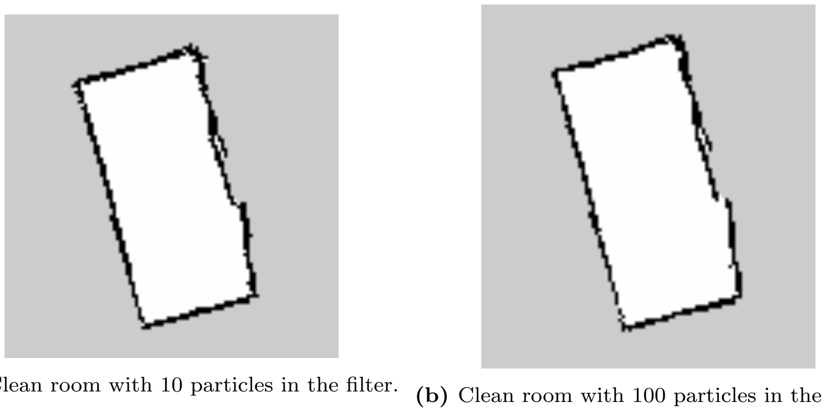


Figure 5.1: Resulting map with default parameters for GMapping (30 particles).

5.1.2 Tuned Parameters

Difference in Number of Particles

Figure 5.2a and 5.2b shows the maps generated where the number of particles are set to 10 and 100, respectively. One can see that the number of particles yield no significant impact on the map. However, the CPU load increases more than 100%.



(a) Clean room with 10 particles in the filter. **(b)** Clean room with 100 particles in the filter.

Figure 5.2: Resulting map showing the difference in number of particles used in the particle filter in GMapping. All other parameters are set to default value.

Number of Particles	**CPU load (%)**
10	15
100	36

Table 5.2: CPU load for GMapping with different numbers of particles.

Optimized Tuning of Parameters

In Figure 5.3 the estimated map with tuned parameters is presented. The number of particles parameter was set to default value (30 particles), where the parameters that were tuned are shown in Table 5.1. One can see that the parameter for resolution (delta) is decreased to $0.04[m]$ per grid cell which affect the resulting map positively. All relevant information are correct and accurate rendered.

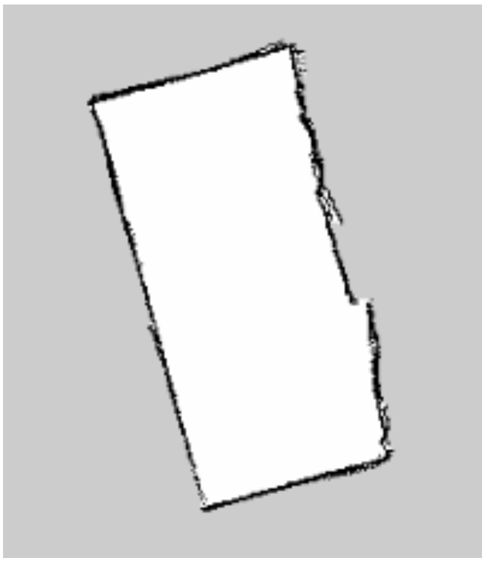


Figure 5.3: Resulting map with parameters tuned.

5.2 Obstacle Course

This section presents the maps from the obstacle course. All the maps are estimated on the same dataset.

5.2.1 Default Parameters

As shown in Figure 5.4, the default parameters yield a relatively inadequate result. The walls on the left- and right hand side are duplicated. The curtain is still captured in the bottom wall but the wall is more arbitrary compared to the maps from the clean room. Also, relevant information is missing. The box, candlestick and the small plants are almost or completely left out of the map.

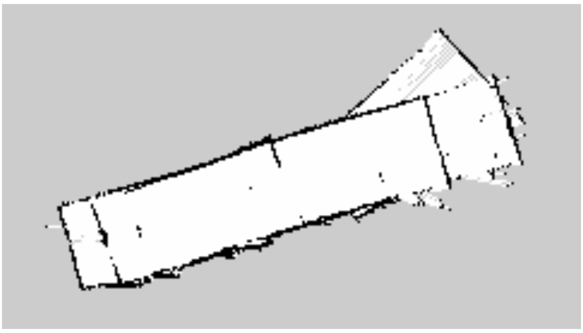


Figure 5.4: Resulting map with default parameters for GMapping.

5.2.2 Tuned Parameters

Low Number of Particles

In Figure 5.5, the resulting tuned map with only 10 particles is presented. One can see that the wall on the right hand side with the incline is presented in a better, but inadequate way. The wall on the left hand side is still duplicated and the same information is left out and poorly reconstructed.

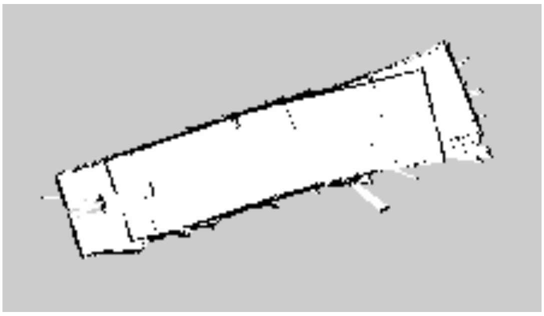


Figure 5.5: Obstacle course with 10 particles where the remaining parameters are tuned according to Table 5.1.

High Number of Particles

In Figure 5.6, the estimated map from the obstacle course with a high number of particles is presented. The parameter was tuned to 100 particles to include more information in the estimation process. One can see that the wall on the left hand side has improved significantly from Figure 5.5, while the wall on the right hand side remain inadequate for autonomous navigation. There are less information about obstacles included in the map with 100 particles.

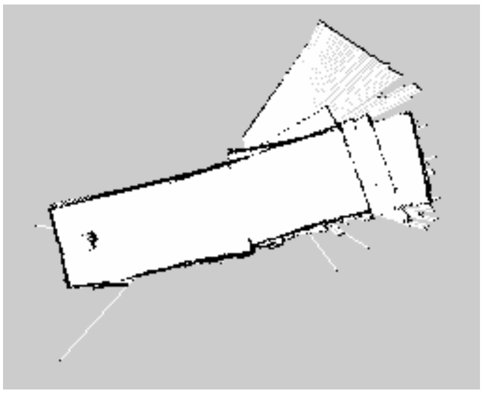


Figure 5.6: Obstacle course with 100 particles where the remaining parameters are tuned according to Table 5.1.

Optimized Tuning of Parameters

Figure 5.7 shows the final estimated map from the obstacle course with the tuned parameters. The parameters are tuned in the way so the map is updated and the scans are processed frequently to contain and collect all information from the dataset. The resampling is done relatively rare compared to when using the default parameter. As one can see, the walls and corners are both straight and correctly represented. The wall on the left hand side is no longer duplicated. The right hand side wall is still duplicated, but this might be the algorithms way of representing 3D features (the slope). The bottom wall is also more proper represented. However, most of the features are left out of the map. Only the big plant and ceramic tree are included.

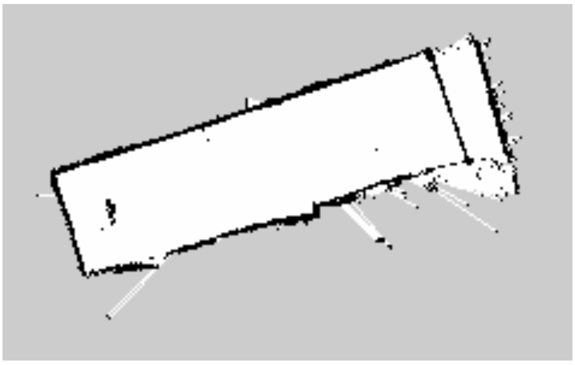


Figure 5.7: Resulting map with parameters tuned.

Chapter 6

Discussion

6.1 Phylax

This section discusses the design and building process of Phylax. In addition, the effect of the different parameters in GMapping will be discussed.

Hardware

As mentioned in Chapter 4, the design choices were based on performance, delivery time and cost. Throughout the project hardware like the encoders, ESP32 and cables all broke at least one time each because it got burnt or took a hit. It would have been a big risk if delivery time was not investigated before choosing off-shelf hardware, but due to fast delivery the broken hardware did not slow down the project.

In this thesis the processing unit on Phylax only ran a few ROS nodes. The mapping was done on the external laptop so the need for the Xavier for this project was close to zero. This is because the ROS software that the Xavier ran could have been executed directly on the ESP32 and distributed to the ROS system through its WIFI connection. Since `phylax_simple.launch` was preferred over `phylax.launch` the different controllers were not in use. Controlling the vehicle using the joystick was therefore harder because it was super responsive and the velocity acted discrete; almost full speed or no speed. Carrying the weight from the Xavier and lidar made the vehicle unbalanced when there were several starts and stops collecting data. Hence, removing the Xavier and running the software from the ESP32 in this project would have created more balance in the driving performance and ensure that the lidar operated in a more steady environment. When all of this is said, the Nvidia Xavier was chosen due to its high performance level and its compatibility to new and powerful technology, like neural networks, and creates robustness and potential for the Phylax platform to achieve a higher score on the level of autonomy scale.

The two middle motors were replaced with two new ones with encoders for registering odometry. When debugging the odometry it was registered that the two motors operated with a slightly different speed when given the same command from the joystick. This was discovered when the vehicle took a slight turn when it was set to go on a straight line. This is most likely due to different friction in the two motors. The cumulative distance in x-direction was always correct, while the cumulative distance in the y-direction was less reliable. The majority of the times the y-value was correct, but once in a while it contained small errors. Also, there is an uncertainty in the accuracy of the transformation, due to the fact that distances for translations might include small errors. These errors might propagate to the estimation of the map.

The use of WIFI to connect the different parts of the Phylax platform sets a range limit on the system. The joystick is wired to the external laptop and the joystick commands travel over the WIFI to the vehicle, so the range of the system equals the range of the network. For this project the range has not been of any trouble but for future use, if the user wants to take datasets outside and in real terrain this might cause some problems. A quick solution to this is to wire the joystick straight into the Xavier, as the Xavier does not supports bluetooth. However, this solution is only compatible in a limited academic environment, so an alternative solution to increase range and secure communication should be investigated.

Software

Designing the software platform using ROS turned out to be convenient. The numerous documentation, tutorials and examples were of great help for both designing and debugging the system. An example for debugging was the use of the rostopic package where topics could be listed and echoed. Also, tools for handling the description and transformations of the robot simplified the implementation and made it more robust. Line 4 in Appendix F.3 is an example of how short and compact a static transformation can be implemented.

In this project, version one of ROS is used even though ROS2 has been launched. The reason for choosing the first version of ROS was, as described in Section 4.1.1, the good access to documentation and examples that ROS2 could not provide on the same level. Apart from this, ROS2 offers features and modifications that might have increased the performance of Phylax. ROS2 is, different from ROS, a decentralized system where the ROS master is no longer needed. Each node has the ability to discover each other and distribute information to the system them self. In other words, the system becomes fully distributed which is more convenient. In addition, the first version of ROS can not guarantee real-time performance which ROS2 would have been able to do.

6.2 The Experiment

In this section utilizing GMapping as the SLAM method in the SLAM system will be discussed.

6.2.1 Initial Remarks

Before addressing the experiment results, the problems with one of the long walls in all of the resulting maps needs to be mentioned. As pointed out in Section 4.2.2, one wall is made out of sheets that represents a rough surface. Hence, the resulting maps only have three out of four straight walls. It is important to emphasize that this is not an algorithm problem but a direct effect of the surrounding scenery. The algorithm performed loop closure on both of the datasets which gives a good indication that the pose estimation is reliable.

6.2.2 GMapping

The estimated maps from the clean room, or benchmark dataset, shows good and reliable maps for autonomous navigation. All the different maps are generated on the same dataset with different algorithm parameters. Analyzing Figure 5.2a and 5.2b shows the differences in using a low and high number of particles for this scenery. The difference is negligible which implies that the scan matching technique in the GMapping algorithm for that scenery is properly reducing the number of particles required for maintaining an accurate map. With a lower number of particles, less information is processed in the algorithm, and hence less computational power is needed (see Table 5.2). This is an important factor to consider in autonomous navigation since the map of the surrounding scenery is needed in real-time.

The geometry of the environment is extremely simple so the need of particles would most likely be higher in a more complex world. However, Figure 5.5 and 5.6 shows a change in the estimated map when increasing the number of particles. The geometry of the room became more accurate by increasing the number of particles, with an exception of the right hand side. Normally, an increase in the number of particles would include more information in the mapping process, and hence estimate features more accurate. There might be several reasons this is not the case in this project, but the discussed error in odometry and the rapid response of the robot while collecting data may impact the resulting map. Comparing Figure 5.5 and 5.7 from the obstacle course presents a decrease in accuracy with a lower number of particles. The algorithm had, most likely, too few particles in the filter for reduction by scan matching.

Furthermore, the resampling threshold is tightly linked to the number of particles. If resampling is performed frequently with a small number of particles, particle depletion will occur. This results in highly erroneous maps where walls and corners tend to duplicate. Analyzing Figure 5.4 and 5.7, shows a duplicated short wall on the left side. In addition, the short wall on the right side tends to be duplicated but the slope is also a factor for this error and will be discussed later. In Figure 5.4 the resampling threshold is 0.5 compared to 0.1 in Figure 5.7, with the same number of particles. It is likely to assume that the resampling happened too frequently in the prior case, so the particles available were not sufficient to represent all the different robot poses.

As Phylax moves around in the environment new scans are processed. Linear update decides how far Phylax translates before a new scan is processed. Decreasing the linear update parameter, a larger amount of information from the environment is processed by the algorithm, but the downside is an increase of CPU load. Figure 5.7 with linear update at 0.1 shows a better result estimating the relevant features compared to Figure 5.4 with linear update at 1. The features that are estimated in the maps are more accurate presented with a low value for linear update.

The process of running back the rosbags and mapping simultaneously was convenient. The rosbag played its content and published the data on different topics, while the GMapping node subscribed the topics needed for mapping. If the rosbag contained data from a 180 second long experiment, it took 180 second to play back the bag and map it. This shows a close to real-time property of GMapping which is important for a military UGV. In this project the collection of data and mapping happened in two steps, but the mapping could have been conducted in parallel with the data acquisition due to its fast computational time.

6.2.3 Closing Remarks

The implementation of GMapping in this thesis is built on the existing ROS package. The performance of the ROS package has been shown in several studies, and the correctness of the implementation in this work is verified through the ability to estimate the clean room correctly. Based on this it was expected that the results for the obstacle course should be more satisfactory. However, there are elements in the conduction of the experiment that should be considered being altered to improve the results. First, a possible user error when driving Phylax may have occurred by driving inconsistently, causing rapid changes for the lidar when acquiring data. Second, the lidar contains several parameters that may be tuned for better performance. A structured approach should be applied to see which effect each of these parameters may have on the estimation of the environment.

It is fair to say that GMapping would have contributed to a low score on the LOA scale for navigating in terrain. The quality of the final map contains too little information about the environment. All of the features, except the big plant, are almost or completely left out. GMapping has difficulties deciding whether the slope on the right side in Figure 5.7 is accessible or not, which results in a duplication of the wall. Since the output of GMapping is a 2D map, a possible solution could be to supplement the SLAM system with an object detection method for detecting slopes, but this will be difficult due to the large uncertainty in how GMapping responds to slopes. In addition, the parameters discussed are tuned in a way that increases the CPU load, with an exception of the number of particles that remained with the default value. With a bigger and more complex dataset a more selective approach to the CPU load might be needed, which may affect the results negatively.

Chapter 7

Conclusion

In this thesis, the UGV platform Phylax has been designed and built. Phylax is capable of conducting datasets through its three sensors: IMU, encoders and lidar. In addition, a GMapping based SLAM system utilizing the datasets has been developed to map the surrounding environment of the UGV. Phylax can operate in fully autonomous mode with differential drive controllers and a multiplexer. The differential drive controller had a bug so the simpler version was used in this thesis. The platform consists of the vehicle, a local network, an external laptop and a joystick. The joystick sends commands through the network to Phylax, which exploits the environment and perceives through sensors. The sensor data are recorded on the external laptop. When the data acquisition is completed the SLAM system estimates a map of the environment based on the recorded dataset.

The performance of the SLAM system was tested indoor in two environments: clean room and obstacle course. The dataset from the clean room was used for tuning the algorithm and worked as a ground truth for GMapping. The obstacle course was designed to include terrain features. The results from the benchmark dataset verifies that GMapping is properly implemented and tuned. The estimated maps are adequate and usable for autonomous navigation, in that scenery. The results from the obstacle course, however, raises a concern for the use of GMapping as a SLAM method for autonomous navigation in terrain. The algorithm struggles to map several objects, and does not decide whether the slope is accessible or not in the resulting occupancy grid map. Even though the algorithm was tried tuned for the specific dataset, the resulting map did not provide all information needed for autonomous navigation in the given scenery. Also, the tuning resulted in parameters effecting the CPU load negatively. However, there are possible errors from the odometry and transformations that may propagate into the estimation of the maps. Also, the user error and the uncertainty if the parameters for the lidar was properly tuned are elements that could have had an impact on the final result.

To conclude this Master's thesis, the UGV platform is successfully designed and built with the ability to acquire datasets based on the sensors: IMU, encoders and lidar. The SLAM system was also properly developed, but GMapping estimated 2D maps of poor quality from the obstacle course. The GMapping method would have contributed to a low score on the LOA scale in the given scenery and undoubtedly in terrain. Although several potential improvements to the experiment have been identified, the results indicate that GMapping can be rejected as a SLAM method for autonomous navigation in terrain for a UGV.

7.1 Further Work

The following is the proposed further work for improving the Phylax platform and the SLAM system:

First, the possible odometry and transformation offsets should be corrected to exclude the induced insecurity. This would lead to a more robust and correct dataset for GMapping to estimate on. Second, the differential drive controller should be debugged so Phylax can operate in full autonomous mode. This would be an additional way of testing the SLAM system to see if Phylax can autonomously navigate based on the estimated map. Final, and perhaps the most important, investigate alternative SLAM methods. The Google Cartographer SLAM method that estimates both 2D and 3D maps could be a promising method. Google Cartographer has shown promising results in indoor environments, and the fact that it estimates 3D maps and is an opensource implementation makes the algorithm interesting.

Bibliography

Abdelrasoul, Y., Saman, A.B.S.H., Sebastian, P., 2016. A quantitative study of tuning ROS gmapping parameters and their effect on performing indoor 2D SLAM, in: 2016 2nd IEEE International Symposium on Robotics and Manufacturing Automation (ROMA), Ipoh, Malaysia. pp. 1–6. doi:10.1109/ROMA.2016.7847825.

Bae, K.H., Lichti, D.D., 2007. On-Site Self-Calibration Using Planar Features For Terrestrial Laser Scanners, in: ISPRS Workshop on Laser Scanning 2007 and SilviLaser 2007, Espoo, Finland. pp. 14–19. URL: https://foto.aalto.fi/ls2007/final_papers/Bae_2007.pdf.

Bailey, T., Nieto, J., Guivant, J., Stevens, M., Nebot, E., 2006. Consistency of the EKF-SLAM Algorithm, in: 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems, Beijing, China. pp. 3562–3568. doi:10.1109/IROS.2006.281644.

Balasuriya, B., Chathuranga, B., Jayasundara, B., Napagoda, N., Kumarawadu, S., Chandima, D., Jayasekara, A., 2016. Outdoor robot navigation using Gmapping based SLAM algorithm, in: 2016 Moratuwa Engineering Research Conference (MERCon), Moratuwa, Sri Lanka. pp. 403–408. doi:10.1109/MERCon.2016.7480175.

Bergel, H.B.N., 2020. Simultaneous Localization and Mapping applied on UGV. Technical Report. Norwegian University of Science and Technology.

Bijelic, M., Gruber, T., Ritter, W., 2018. A Benchmark for Lidar Sensors in Fog: Is Detection Breaking Down?, in: 2018 IEEE Intelligent Vehicles Symposium (IV), Changsu, China. pp. 760–767. doi:10.1109/IVS.2018.8500543.

Brand, C., Schuster, M.J., Hirschmüller, H., Suppa, M., 2014. Stereo-vision based obstacle mapping for indoor/outdoor SLAM, in: 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, Chicago, USA. pp. 1846–1853. doi:10.1109/IROS.2014.6942805.

Cadena, C., Carlone, L., Carrillo, H., Latif, Y., Scaramuzza, D., Neira, J., Reid, I., Leonard, J.J., 2016. Past, Present, and Future of Simultaneous Localization and Mapping: Toward the Robust-Perception Age. IEEE Transactions on Robotics 32, 1309–1332. doi:10.1109/TRO.2016.2624754.

Doucet, A., 1998. On sequential simulation-based methods for bayesian filtering. Technical Report. University of Cambridge.

Duchoň, F., Hažík, J., Rodina, J., Tölgyessy, M., Dekan, M., Sojka, A., 2019. Verification of SLAM Methods Implemented in ROS 6, 10579–10584. URL: http://www.jmest.org/wp-content/uploads/JMESTN42353033.pdf.

Euler, L., 1776. Novi Commentarii Academiae Scientairum Imperialis Petropolitan. volume XX.

Forsvarets-forskningsinstitutt, . Autonom minerydding. URL: https://www.ffi.no/forskning/prosjekter/autonom-minerydding.

Fossen, T.I., 2020. Handbook of Marine Craft Hydrodynamics And Motion Control. 2 ed., John Wiley & Sons Ltd., Chichester.

Girard, R., Mavromatis, S., Sequeira, J., Belanger, N., Anoufa, G., 2019. A Vision-Based Assistance Key Differenciator for Helicopters Automonous Scalable Missions, in: Althoefer, K., Konstantinova, J., Zhang, K. (Eds.), Towards Autonomous Robotic Systems, Springer International Publishing, Cham. pp. 202–210. doi:10.1007/978-3-030-25332-5_18.

Glennie, C., Lichti, D.D., 2010. Static Calibration and Analysis of the Velodyne HDL-64E S2 for High Accuracy Mobile Scanning. Remote Sensing 2, 1610–1624. doi:10.3390/rs2061610. number: 6 Publisher: Molecular Diversity Preservation International.

Glennie, C.L., Kusari, A., Facchin, A., 2016. Calibration And Stability Analysis Of The VLP-16 Laser Scanner, in: ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, Lausanne, Switzerland. pp. 55–60. doi:10.5194/isprs-archives-XL-3-W4-55-2016.

Grisetti, G., Stachniss, C., Burgard, W., 2005. Improving Grid-based SLAM with Rao-Blackwellized Particle Filters by Adaptive Proposals and Selective Resampling, in: Proceedings of the 2005 IEEE International Conference on Robotics and Automation, Barcelona, Spain. pp. 2432–2437. doi:10.1109/ROBOT.2005.1570477.

Grisetti, G., Stachniss, C., Burgard, W., 2007a. Improved Techniques for Grid Mapping With Rao-Blackwellized Particle Filters. IEEE Transactions on Robotics 23, 34–46. doi:10.1109/TRO.2006.889486.

Grisetti, G., Tipaldi, G.D., Stachniss, C., Burgard, W., Nardi, D., 2007b. Fast and accurate SLAM with Rao–Blackwellized particle filters. Robotics and Autonomous Systems 55, 30–38. doi:10.1016/j.robot.2006.06.007.

Hasirlioglu, S., Kamann, A., Doric, I., Brandmeier, T., 2016. Test methodology for rain influence on automotive surround sensors, in: 2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC), Rio de Janeiro, Brazil. pp. 2242–2247. doi:10.1109/ITSC.2016.7795918.

Ho, K.L., Newman, P., 2006. Loop closure detection in SLAM by combining visual and spatial appearance. Robotics and Autonomous Systems 54, 740–749. doi:10.1016/j.robot.2006.04.016.

Huang, H.M., 2007. Autonomy levels for unmanned systems (ALFUS) framework: safety and application issues, in: PerMIS '07: Proceedings of the 2007 Workshop on Performance Metrics for Intelligent Systems, Washington D.C., USA. pp. 48–53. doi:10.1145/1660877.1660883.

Jo, K., Kim, J., Kim, D., Jang, C., Sunwoo, M., 2014. Development of Autonomous Car—Part I: Distributed System Architecture and Development Process. IEEE Transactions on Industrial Electronics 61, 7131–7140. doi:10.1109/TIE.2014.2321342.

Koutsojannis, C., Sirmakessis, S., 2009. Tools and Applications with Artificial Intelligence. Springer Science & Business Media, Berlin.

Lin, P., Bekey, G., Abney, K., 2008. Autonomous Military Robotics: Risk, Ethics, and Design. Technical Report. US Department of Navy, Office of Naval Research. URL: https://apps.dtic.mil/sti/pdfs/ADA534697.pdf.

Montemerlo, M., Thrun, S., Koller, D., Wegbreit, B., 2002. FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem, in: AAAI-02, Edmonton, Canada. pp. 593–598. URL: https://www.aaai.org/Papers/AAAI/2002/AAAI02-089.pdf.

Parasuraman, R., 2000. Designing automation for human use: empirical studies and quantitative models. Ergonomics 43, 931–951. doi:10.1080/001401300409125.

ROS, 2019. gmapping - ROS Wiki. URL: http://wiki.ros.org/gmapping.

Shi, W., Alawieh, M.B., Li, X., Yu, H., 2017. Algorithm and hardware implementation for visual perception system in autonomous vehicle: A survey. Integration 59, 148–156. doi:10.1016/j.vlsi.2017.07.007.

Skaloud, J., Lichti, D., 2006. Rigorous approach to bore-sight self-calibration in airborne laser scanning. ISPRS Journal of Photogrammetry and Remote Sensing 61, 47–59. doi:10.1016/j.isprsjprs.2006.07.003.

Smith, R.C., Cheeseman, P., 1986. On the Representation and Estimation of Spatial Uncertainty. The International Journal of Robotics Research 5, 56–68. doi:10.1177/027836498600500404.

Stachniss, C., Grisetti, G., Burgard, W., Roy, N., 2007. Analyzing gaussian proposal distributions for mapping with rao-blackwellized particle filters, in: 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems, San Diego, USA. pp. 3485–3490. doi:10.1109/IROS.2007.4399005.

Stachniss, C., Leonard, J.J., Thrun, S., 2016. Simultaneous Localization and Mapping, in: Siciliano, B., Khatib, O. (Eds.), Springer Handbook of Robotics. Springer International Publishing, Cham, Switzerland. Springer Handbooks, pp. 1153–1175. URL: https://doi.org/10.1007/978-3-319-32552-1_46.

Thrun, S., Bücken, A., 1996. Integrating Grid-Based and Topological Maps for Mobile Robot Navigation, in: Proceedings of the Thirteenth National Conference on Artificial Intelligence AAAI, Portland, USA. URL: https://roboticsclub.org/redmine/projects/colony/repository/revisions/1945/raw/branches/scout/SLAM/Integrating%20Grid%20Based%20Approach.pdf.

Vagia, M., Transeth, A.A., Fjerdingen, S.A., 2016. A literature review on the levels of automation during the years. What are the different taxonomies that have been proposed? Applied Ergonomics 53, 190–202. doi:10.1016/j.apergo.2015.09.013.

VelodyneLidar, . What is Lidar? URL: https://velodynelidar.com/what-is-lidar/.

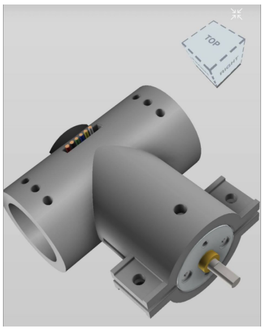
Wang, P., Chen, Z., Zhang, Q., Sun, J., 2016. A loop closure improvement method of Gmapping for low cost and resolution laser scanner. IFAC-PapersOnLine 49, 168–173. doi:10.1016/j.ifacol.2016.07.569.

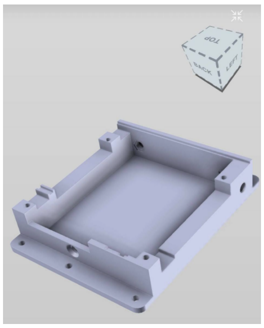
Weerasinghe, K.K.D.K.U., Silva, L.C.J., Basnayake, B.M.S.S., Sandanayaka, S.D.M., Kumarawadu, S.P., Chandima, D.P., Jayasekara, A.G.B.P., 2016. Mapping and path planning for long distance autonomous navigation using multisensory data, in: 2016 Electrical Engineering Conference (EECon), Colombo, Sri Lanka. pp. 1–6. doi:10.1109/EECon.2016.7830926.

Appendices

Appendix A

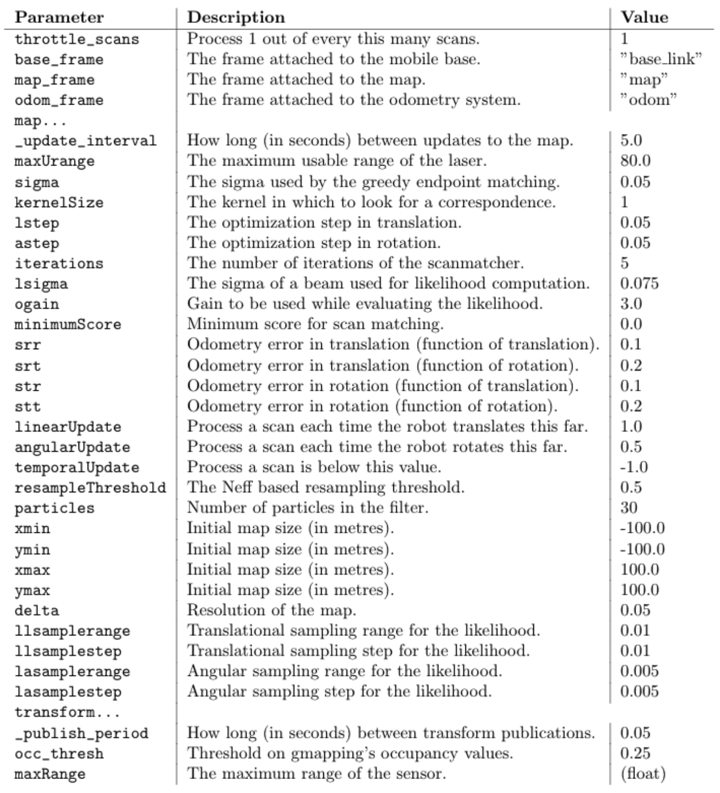
3D Design


(a) Design of the 3D printed motor hot.


(b) Design of the 3D printed Xavier mounting.

Appendix B

Default Parameter Values for GMapping



Parameter	Description	Value
`throttle_scans`	Process 1 out of every this many scans.	1
`base_frame`	The frame attached to the mobile base.	"base_link"
`map_frame`	The frame attached to the map.	"map"
`odom_frame`	The frame attached to the odometry system.	"odom"
`map...`		
`_update_interval`	How long (in seconds) between updates to the map.	5.0
`maxUrange`	The maximum usable range of the laser.	80.0
`sigma`	The sigma used by the greedy endpoint matching.	0.05
`kernelSize`	The kernel in which to look for a correspondence.	1
`lstep`	The optimization step in translation.	0.05
`astep`	The optimization step in rotation.	0.05
`iterations`	The number of iterations of the scanmatcher.	5
`lsigma`	The sigma of a beam used for likelihood computation.	0.075
`ogain`	Gain to be used while evaluating the likelihood.	3.0
`minimumScore`	Minimum score for scan matching.	0.0
`srr`	Odometry error in translation (function of translation).	0.1
`srt`	Odometry error in translation (function of rotation).	0.2
`str`	Odometry error in rotation (function of translation).	0.1
`stt`	Odometry error in rotation (function of rotation).	0.2
`linearUpdate`	Process a scan each time the robot translates this far.	1.0
`angularUpdate`	Process a scan each time the robot rotates this far.	0.5
`temporalUpdate`	Process a scan is below this value.	-1.0
`resampleThreshold`	The Neff based resampling threshold.	0.5
`particles`	Number of particles in the filter.	30
`xmin`	Initial map size (in metres).	-100.0
`ymin`	Initial map size (in metres).	-100.0
`xmax`	Initial map size (in metres).	100.0
`ymax`	Initial map size (in metres).	100.0
`delta`	Resolution of the map.	0.05
`llsamplerange`	Translational sampling range for the likelihood.	0.01
`llsamplestep`	Translational sampling step for the likelihood.	0.01
`lasamplerange`	Angular sampling range for the likelihood.	0.005
`lasamplestep`	Angular sampling step for the likelihood.	0.005
`transform...`		
`_publish_period`	How long (in seconds) between transform publications.	0.05
`occ_thresh`	Threshold on gmapping's occupancy values.	0.25
`maxRange`	The maximum range of the sensor.	(float)

Figure B.1: Default values for the parameters used in GMapping ROS (2019).

Appendix C

ESP32 Software

C.1 encoder.cpp

```
#include "encoder.h"

// Set encoder variables
PololuEncoder::PololuEncoder(int gearing, int cpr, float radius):
  _gears(gearing),
  _cpr(cpr),
  _wheelRadius(radius),
  _dPhiL(0),
  _dPhiR(0),
  _th(0),
  _x(0),
  _y(0),
  _encoder1CountPrev(0),
  _encoder2CountPrev(0),
  _pathDistance(0)
{
  _left_encoder = new ESP32Encoder();
  _right_encoder = new ESP32Encoder();
}

void PololuEncoder::init( )
{
  // Enable the weak pull up resistors
  ESP32Encoder::useInternalWeakPullResistors=UP;

  _left_encoder->attachFullQuad(19, 18);
  _right_encoder->attachFullQuad(17, 16);

  //Clear encoder count
  _left_encoder->clearCount();
  _right_encoder->clearCount();

  _enc2rev = 1.0/1632.0;
  _enc2rad = _enc2rev * 2 * PI;
  _enc2wheel = _enc2rad * _wheelRadius;
}

// Get count from encoders
void PololuEncoder::update(uint32_t &encoder1Count, uint32_t&
    encoder2Count)
{
  encoder1Count = _left_encoder->getCount();
  encoder2Count = _right_encoder->getCount();
}
```

C.2 odometer.cpp

```

#include "odometer.h"
#include <tf/transform_broadcaster.h>
#include <tf/tf.h>
#include <std_msgs/Int32.h>
#include <sensor_msgs/Imu.h>
#include <sensor_msgs/Temperature.h>
#include <nav_msgs/Odometry.h>
#include <geometry_msgs/TransformStamped.h>

char base_link[] = "base_link";
char odom[]      = "odom";

nav_msgs::Odometry odom_msg;
ros::Publisher odometry_pub("/odom", &odom_msg);

geometry_msgs::TransformStamped t;
tf::TransformBroadcaster odom_broadcaster;

Odometer::Odometer(PololuEncoder *encoder):_encoder(encoder){}

void Odometer::init(ros::NodeHandle &nh, float track)
{
  _track = track;
  //Initialize pololu encoder(s)
  _encoder->init();

  // Advertise topics
  nh.advertise(odometry_pub);
  odom_broadcaster.init(nh);
}

void Odometer::updateEncoder()
{
  uint32_t encoder1Count;
  uint32_t encoder2Count;

  _encoder->update(encoder1Count, encoder2Count);

  int32_t dEncoder1 = (encoder1Count - _encoder1CountPrev);
  int32_t dEncoder2 = (encoder2Count - _encoder2CountPrev);

  // Update the angle increment in radians
  float dphi1 = ((float)dEncoder1 * _encoder->_enc2rad);
  float dphi2 = ((float)dEncoder2 * _encoder->_enc2rad);

  // For encoder index and motor position switching (Right is 1, Left
    is 2)
  _dPhiR = dphi1;
  _dPhiL = dphi2;

  _encoder1CountPrev = encoder1Count;
  _encoder2CountPrev = encoder2Count;
}

```

```
void Odometer::evaluateRobotPose(unsigned long diff_time)
{
  float dTh = _encoder->_wheelRadius/(_track) *(_dPhiR - _dPhiL);
  float dist = _encoder->_wheelRadius *(_dPhiR + _dPhiL) / 2;
  float dx = _encoder->_wheelRadius/2 * (cos(_th)*_dPhiR + cos(_th)*
    _dPhiL);
  float dy = _encoder->_wheelRadius/2 * (sin(_th)*_dPhiR + sin(_th)*
    _dPhiL);
  long dt = float(diff_time)/1000;

  _th+= dTh;
  _x+=dx;
  _y+=dy;
  _vx = dist/dt;
  _vTh = dTh/dt;
  _pathDistance = _pathDistance + sqrt(dx*dx + dy*dy);

  Serial.println("Math stuff = "+String((int32_t)_pathDistance));
}

void Odometer::publish_odom(ros::Time current_time)
{
  odom_msg.header.stamp          = current_time;
  odom_msg.header.frame_id       = odom;
  odom_msg.child_frame_id        = base_link;

  odom_msg.pose.pose.position.x  = _x;
  odom_msg.pose.pose.position.y  = _y;
  odom_msg.pose.pose.position.z  = 0.0;
  odom_msg.pose.pose.orientation = tf::createQuaternionFromYaw(_th);

  odom_msg.twist.twist.linear.x  = _vx;
  odom_msg.twist.twist.linear.y  = 0;
  odom_msg.twist.twist.angular.z = _vTh;

  odometry_pub.publish(&odom_msg);
}

void Odometer::broadcastTf(ros::Time current_time)
{
  t.header.stamp             = current_time;
  t.header.frame_id          = odom;
  t.child_frame_id           = base_link;

  t.transform.translation.x = _x;
  t.transform.translation.y = _y;
  t.transform.translation.z = 0.0;
  t.transform.rotation      = tf::createQuaternionFromYaw(-_th);

  odom_broadcaster.sendTransform(t);
}
```

Appendix D

phylax_description

D.1 description.launch

```
<?xml version="1.0"?>
<launch>
  <arg name="config" default="front_laser" />

  <param name="robot_description"
         command="$(find phylax_description)/urdf/configs/$(arg config)
                  $(find xacro)/xacro $(find phylax_description)/urdf/phylax.urdf.xacro
                  --inorder" />
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher" />
</launch>
```

D.2 phylax.urdf.xacro

```xml
<?xml version="1.0"?>
<robot name="phylax" xmlns:xacro="http://www.ros.org/wiki/xacro">

<!--Setting parameters for the robot model-->
<xacro:property name="deg_to_rad" value="0.017453"/>
<xacro:property name="PI" value="3.14"/>

<xacro:property name="chassis_len" value=".380"/>
<xacro:property name="chassis_width" value="0.165"/>
<xacro:property name="chassis_height" value="0.070"/>

<xacro:property name="wheel_radius" value="0.065"/>
<xacro:property name="wheel_width" value="0.06"/>
<xacro:property name="wheel_height" value="0.13"/>
<xacro:property name="wheel_mass" value=".480"/>

<xacro:property name="dummy_inertia" value="1e-09"/>

<xacro:macro name="cylinder_inertia" params ="m r h">
  <inertial>
   <mass value="${m}"/>
   <inertia ixx="${m*(3*r*r+h*h)/12}"  ixy="0.0" ixz="0.0"
            iyy="${m*(3*r*r+h*h)/12}"  iyz= "0.0"
            izz="${m*r*r/2}"/>
   </inertial>
</xacro:macro>

<!--Describe the relationship between the actuator and joint-->
<xacro:macro name="Transmission_block" params="joint_name">
   <transmission name="${joint_name}_trans">
    <type>transmission_interface/SimpleTransmission</type>
     <joint name="${joint_name}">
       <hardwareInterface>PositionJointInterface</hardwareInterface>
     </joint>
    <actuator name="${joint_name}_motor">
      <mechanicalReduction>1</mechanicalReduction>
     </actuator>
    </transmission>
 </xacro:macro>

<!--Defining the different frames for the robot-->
  <link name="base_link">
    <visual>
      <geometry>
          <box size="0.42 0.290 0.000001"/>
      </geometry>
      <material name="saffron">
          <color rgba="0.95 0.81 0.24 1"/>
      </material>
    </visual>
    <collision>
      <geometry>
        <box size="0.42 0.290 0.000001"/>
      </geometry>
    </collision>
  </link>


  <link name="imu_link">
    <inertial>
      <mass value="0.001"/>
      <origin="0 0 -0.12" rpy="0 0 0"/>
      <inertia="ixx="${dummy_inertia}" ixy="0.0" ixz="0.0" iyy="${dummy_inertia}" iyz="0.0" izz="${dummy_inertia}"/>
    </inertial>
  </link>
  <joint name="imu_joint" type="fixed">
    <parent link="base_link"/>
    <child link="imu_link"/>
  </joint>


<xacro:macro name="wheel" params="pos side xyz xyzi cy">
  <link name="${pos}_${side}_wheel">
    <visual>
      <origin xyz="${xyz}" rpy="${-PI/2} 0 0"/>
      <geometry>
        <mesh filename="package://phylax_description/meshes/${side}_wheel.stl" scale="0.001 0.001 0.001"/>
      </geometry>
    </visual>
    <collision>
      <origin xyz="0 ${cy} 0" rpy="${PI/2} 0 0"/>
      <geometry>
        <cylinder radius="0.06" length="0.062"/>
      </geometry>
    </collision>
    <xacro:cylinder_inertia m="${wheel_mass}" r="${wheel_radius}" h="${wheel_width}"/>
  </link>

  <joint name="${pos}_${side}_wheel_joint" type="continuous">
    <parent link="base_link"/>
    <child link="${pos}_${side}_wheel"/>
    <axis xyz="0 1 0"/>
    <origin xyz="${xyzi}" rpy="0 0 0"/>
    <limit effort= "100" velocity="100"/>
    <dynamics damping="0.0" friction="0.0"/>
  </joint>

  <transmission name="${pos}_${side}_wheel_trans" type="SimpleTransmission">
    <type>transmission_interface/SimpleTransmission</type>
    <joint name="${pos}_${side}_wheel_joint">
      <hardwareInterface>hardware_interface/VelocityJointInterface</hardwareInterface>
    </joint>
    <actuator name="{pos}_${side}_wheel_motor">
      <hardwareInterface>hardware_interface/VelocityJointInterface</hardwareInterface>
      <mechanicalReduction>1</mechanicalReduction>
    </actuator>
  </transmission>
</xacro:macro>

<xacro:wheel pos="front" side="left"  xyz="-0.023 -${chassis_width
    /2-0.01} 0.068" xyzi="0.150 ${chassis_width/2} 0" cy="0.036"/>
<xacro:wheel pos="front" side="right" xyz="-0.023 -${chassis_width
    -0.02} 0.068" xyzi="0.150 -${chassis_width/2} 0" cy="-0.036"/>
<xacro:wheel pos="center" side="left" xyz="-0.023 -${chassis_width
    /2-0.01} 0.068" xyzi="0 ${chassis_width/2} 0" cy="0.036"/>
<xacro:wheel pos="center" side="right" xyz="-0.023 -${chassis_width
    -0.02} 0.068" xyzi="0 -${chassis_width/2} 0" cy="-0.036"/>
<xacro:wheel pos="rear" side="left" xyz="-0.023 -${chassis_width
    /2-0.01} 0.068" xyzi="-0.150 ${chassis_width/2} 0" cy="0.036"/>
<xacro:wheel pos="rear" side="right" xyz="-0.023 -${chassis_width
    -0.02} 0.068" xyzi="-0.150 -${chassis_width/2} 0" cy="-0.036"/>


</robot>
```

Appendix E

phylax_control

E.1 teleop.launch

```
<launch>
  <!--Set control unit-->
  <arg name="joy_dev" default="/dev/input/js0" />
  <arg name="joystick" default="true" />
  <arg name="keyboard" default="false" />

  <!--Necessary nodes are initialized and config files are uploaded-->
  <group ns="keyboard_teleop" if="$(arg keyboard)">
    <node pkg="teleop_twist_keyboard" type="teleop_twist_keyboard.py"
    name="teleop_kbd" output="screen"/>
  </group>
  <group ns="joystick_teleop" if="$(arg joystick)">
    <rosparam command="load" file="$(find phylax_control)/config/
    teleop_ps4.yaml" />
    <node pkg="joy" type="joy_node" name="joy_node" />
    <node pkg="teleop_twist_joy" type="teleop_node" name="
    teleop_twist_joy"/>
   </group>
  <node pkg="interactive_marker_twist_server" type="marker_server"
    name="twist_marker_server">
    <param name="marker_size_scale" value="2" />
  </node>
</launch>
```

E.2 control.launch

```xml
<launch>
  <!--Upload config file-->
  <rosparam command="load" file="$(find phylax_control)/config/control.yaml" />

  <!--State the needed PID controllers and initialize node-->
  <node name="controller_spawner" pkg="controller_manager" type="spawner"
        respawn="true" output="screen"
        args="joint_state_controller
              diff_drive_controller
              --shutdown-timeout 3"/>

  <!--Initialize localisation and multiplexer node-->
  <node pkg="robot_localization" type="ekf_localization_node" name="ekf_localization">
    <rosparam command="load" file="$(find phylax_control)/config/robot_localization.yaml" />
  </node>

  <node pkg="twist_mux" type="twist_mux" name="twist_mux">
    <rosparam command="load" file="$(find phylax_control)/config/twist_mux.yaml" />
    <rosparam param="locks">[]</rosparam>
    <remap from="cmd_vel_out" to="/diff_drive_controller/cmd_vel"/>
  </node>

</launch>
```

E.3 teleop_ps4.yaml

```yaml
# Teleop configuration for PS4 joystick(s)
  teleop_twist_joy:
    axis_linear: 1
    scale_linear: 1
    scale_linear_turbo: 2.0
    axis_angular: 5
    scale_angular: 1
    enable_button: 4
    enable_turbo_button: 5
  joy_node:
    deadzone: 0.1
    autorepeat_rate: 20
    dev: /dev/input/js0
```

E.4 control.yaml

```yaml
joint_state_controller:
  type: "joint_state_controller/JointStateController"
  publish_rate: 50

diff_drive_controller:
  type: "diff_drive_controller/DiffDriveController"
  publish_rate: 50

  left_wheel: ['front_left_wheel_joint','center_left_wheel_joint','
    rear_left_wheel_joint']
  right_wheel: ['front_right_wheel_joint','center_right_wheel_joint','
    rear_right_wheel_joint']

  pose_covariance_diagonal: [0.001, 0.001, 1000000.0, 1000000.0,
    1000000.0, 0.03]
  twist_covariance_diagonal: [0.001, 0.001, 0.001, 1000000.0,
    1000000.0, 0.03]
  cmd_vel_timeout: 0.25

  k_l: 0.1
  k_r: 0.1

  # Odometry fused with IMU is published by robot_localization
  enable_odom_tf: false

  # Wheel separation and radius multipliers
  wheel_separation_multiplier: 1.5 # default: 1.0
  wheel_radius_multiplier    : 1.0 # default: 1.0

  # Velocity and acceleration limits
  linear:
    x:
      has_velocity_limits    : true
      max_velocity           : 2.0   # m/s
      has_acceleration_limits: true
      max_acceleration       : 20.0   # m/s^2
  angular:
    z:
      has_velocity_limits    : true
      max_velocity           : 4.0   # rad/s
      has_acceleration_limits: true
      max_acceleration       : 25.0   # rad/s^2
```

E.5 robot_localization.yaml

```
#Configuation for robot odometry EKF
#
frequency: 50

odom0: /diff_drive_controller/odom
odom0_config: [false, false, false,
               false, false, false,
               true, true, true,
               false, false, true,
               false, false, false]
odom0_differential: false

imu0: /imu/data
imu0_config: [false, false, false,
              true, true, true,
              false, false, false,
              true, true, true,
              false, false, false]
imu0_differential: false

odom_frame: odom
base_link_frame: base_link
world_frame: odom
```

E.6 twist_mux.yaml

```yaml
topics:
- name    : keyboard
  topic   : keyboard_teleop/cmd_vel
  timeout : 0.5
  priority: 9
- name    : joystick_teleop
  topic   : joystick_teleop/cmd_vel
  timeout : 0.5
  priority: 10
- name    : interactive_marker
  topic   : twist_marker_server/cmd_vel
  timeout : 0.5
  priority: 8
- name    : autonav
  topic   : cmd_vel
  timeout : 0.5
  priority: 1
locks:
- name    : e_stop
  topic   : e_stop
  timeout : 0.0
  priority: 255
```

Appendix F

phylax_base

F.1 phylax.launch

```xml
<launch>

  <!--Upload robot model-->
  <include file="$(find phylax_description)/launch/description.launch" />

  <!--Initialize node-->
  <node pkg="phylax_base" type="phylax_node" name="phylax_node" output="screen">
    <remap from="/phylax_node/feedback" to="feedback" />
</node>


  <node pkg="rosserial_server" type="socket_node" name="rosserial_server" />
  <node pkg="rosserial_python" type="message_info_service.py" name="rosserial_message_info" />
  <node pkg="pololu_driver" type"=pololu_driver_node" name="pololu_driver_node" />


  <!--Launch-->
  <include file="$(find phylax_control)/launch/control.launch" />
  <include file="$(find velodyne_pointcloud/launch/VLP_16points.launch"/>
  <include file="$(find phylax_description/launch/point2laser.launch"/>

</launch>
```

F.2 phylax_simple.launch

```xml
<launch>

  <!--Launch-->
  <include file="$(find phylax_description)/launch/description.launch">
  <include file="$(find velodyne_pointcloud/launch/VLP_16points.launch"/>
  <include file="$(find phylax_description/launch/point2laser.launch"/>

  <!--Initalize nodes-->
  <node pkg="phylax_base" type="simple_joy_node" name="simple_joy_node" output="screen">
    <remap from="/cmd_drive" to="/phylax_node/cmd_drive" />
  </node>

  <node pkg="rosserial_server" type="socket_node" name="rosserial_server" />
  <node pkg="rosserial_python" type="message_info_service.py" name="rosserial_message_info" />
  <node pkg="pololu_driver" type="pololu_driver_node" name="pololu_driver_node" />

</launch>
```

F.3 point2laser.launch

```xml
<launch>

<!--Static transformation from veldoyne frame to base_link-->
<node pkg="tf" type="static_transform_publisher" name="
  velodyne_to_base_link" args="0 0 -0.2 0 0 0 base_link velodyne 100
  " />

<!--Initialize node and set parameters-->
<node pkg="pointcloud_to_laserscan" type="
  pointcloud_to_laserscan_node" name="pointcloud_to_laserscan">

  <remap from="cloud_in" to="/velodyne_points"/>
  <remap from="scan" to="/front/scan"/>
  <rosparam>
      transform_tolerance: 0.01
      min_height: 0.25
      max_height: 0.75

      angle_min: -3.1415
      angle_max: 3.1415
      angle_increment: 0.01
      scan_time: 0.1
      range_min: 0.9
      range_max: 130
      use_inf: true
      concurrency_level: 0
  </rosparam>

</node>
</launch>
```

F.4 phylax_base.cpp

```cpp
#include <boost/thread.hpp> //C++ library for managing threads
#include <controller_manager/controller_manager.h> // ROS library for
    handling controllers
#include "phylax_base/phylax_hardware.h"

typedef std::chrono::system_clock time_source;

// Spawns a thread that reads the value from the robot joints position
void controlLoopThread(phylax_base::PhylaxHardware *pb,
    controller_manager::ControllerManager* cm, ros::Rate rate)
{
  time_source::time_point last_time = time_source::now();

  while (1)
  {
    std::chrono::system_clock::time_point current_time = time_source::
    now();
    std::chrono::duration<double> elapsed_time = current_time -
    last_time;
    ros::Duration elapsed(elapsed_time.count());
    last_time = current_time;

    pb->read();
    cm->update(ros::Time::now(), elapsed);
    pb->write();
    rate.sleep();
  }
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "phylax_node");
  ros::NodeHandle controller_nh("");

  ROS_INFO("Phylax is active!");

  phylax_base::PhylaxHardware phylax;
  controller_manager::ControllerManager cm(&phylax, controller_nh);

  boost::thread(boost::bind(controlLoopThread, &phylax, &cm, ros::Rate
    (50)));

  ros::spin();

  ROS_INFO("Phylax is going for a nap!");

  return 0;
}
```

F.5 phylax_hardware.cpp

```cpp
#include <boost/assign.hpp>
#include "phylax_base/phylax_hardware.h"

namespace phylax_base
{

// Defining joints were states can be read from
PhylaxHardware::PhylaxHardware():nh_("~"){
  ros::V_string joint_names = {"front_left_wheel_joint", "front_right_wheel_joint", "center_left_wheel_joint", "center_right_wheel_joint", "rear_left_wheel_joint", "rear_right_wheel_joint"};

  for (unsigned int i = 0; i < joint_names.size(); i++)
  {
    hardware_interface::JointStateHandle joint_state_handle(joint_names[i],
        &joints_[i].position, &joints_[i].velocity, &joints_[i].effort);
    joint_state_interface_.registerHandle(joint_state_handle);

    hardware_interface::JointHandle joint_handle(
        joint_state_handle, &joints_[i].velocity_command);
    velocity_joint_interface_.registerHandle(joint_handle);
  }

  registerInterface(&joint_state_interface_);
  registerInterface(&velocity_joint_interface_);

  feedback_sub_ = nh_.subscribe("feedback", 1, &PhylaxHardware::feedbackCallback, this);
  cmd_pub_.init(nh_, "cmd_drive", 1);

}

void PhylaxHardware::feedbackCallback(const phylax_msgs::Feedback::ConstPtr& msg)
{
  boost::mutex::scoped_lock lock(feedback_mutex_);
  feedback_msg_ = msg;
}

// Read method
void PhylaxHardware::read()
{
  boost::mutex::scoped_lock feedback_lock(feedback_mutex_, boost::try_to_lock);
  if (feedback_msg_ && feedback_lock)
  {
    for (int i = 0; i < 6; i++)
    {
      joints_[i].position = feedback_msg_->drivers[i % 2].measured_travel;
      joints_[i].velocity = feedback_msg_->drivers[i % 2].measured_velocity;
    }
  }
}

// Write method
void PhylaxHardware::write() {
  if (cmd_pub_.trylock())
  {
    cmd_pub_.msg_.mode = phylax_msgs::Drive::MODE_VELOCITY;
    cmd_pub_.msg_.drivers[phylax_msgs::Drive::LEFT] = joints_[0].
    velocity_command;
    cmd_pub_.msg_.drivers[phylax_msgs::Drive::RIGHT] = joints_[1].
    velocity_command;
    cmd_pub_.unlockAndPublish();
  }
}

}
```

F.6 simple_joy_node.cpp

```cpp
#include "ros/ros.h"
#include "phylax_msgs/Drive.h"
#include "sensor_msgs/Joy.h"

#include "boost/algorithm/clamp.hpp"

class SimpleJoy
{
public:
  explicit SimpleJoy(ros::NodeHandle* nh);

private:
  void joyCallback(const sensor_msgs::Joy::ConstPtr& joy);

  ros::NodeHandle* nh_;
  ros::Subscriber joy_sub_;
  ros::Publisher drive_pub_;

  int deadman_button_;
  int axis_linear_;
  int axis_angular_;
  float scale_linear_;
  float scale_angular_;

  bool sent_deadman_msg_;
};

SimpleJoy::SimpleJoy(ros::NodeHandle* nh) : nh_(nh)
{
  joy_sub_ = nh_->subscribe<sensor_msgs::Joy>("/joystick_teleop/joy",
    1, &SimpleJoy::joyCallback, this);
  drive_pub_ = nh_->advertise<phylax_msgs::Drive>("cmd_drive", 1, true
    );

  ros::param::param("~deadman_button", deadman_button_, 4);
  ros::param::param("~axis_linear", axis_linear_, 1);
  ros::param::param("~axis_angular", axis_angular_, 5);
  ros::param::param("~scale_linear", scale_linear_, 1.0f);
  ros::param::param("~scale_angular", scale_angular_, 1.0f);

  sent_deadman_msg_ = false;
}

void SimpleJoy::joyCallback(const sensor_msgs::Joy::ConstPtr& joy_msg)
{
  phylax_msgs::Drive drive_msg;

  // When deadman button is pressed, set the message for motor
    controllers
  if (joy_msg->buttons[deadman_button_])
  {
    drive_msg.mode = phylax_msgs::Drive::MODE_PWM;
    float linear = joy_msg->axes[axis_linear_] * scale_linear_;
    float angular = joy_msg->axes[axis_angular_] * scale_angular_;
    drive_msg.drivers[phylax_msgs::Drive::LEFT] = -boost::algorithm::
    clamp(linear - angular, -1.0, 1.0);
    drive_msg.drivers[phylax_msgs::Drive::RIGHT] = boost::algorithm::
    clamp(linear + angular, -1.0, 1.0);
    drive_pub_.publish(drive_msg);
    sent_deadman_msg_ = false;
  }
  else
  {
    // When deadman button is released, immediately send a single no-
    motion command
    // in order to stop the robot.
    if (!sent_deadman_msg_)
    {
      drive_msg.mode = phylax_msgs::Drive::MODE_NONE;
      drive_pub_.publish(drive_msg);
      sent_deadman_msg_ = true;
    }
  }
}


int main(int argc, char *argv[])
{
  ros::init(argc, argv, "phylax_teleop_joy_pwm");

  ros::NodeHandle nh;
  SimpleJoy simple_joy(&nh);

  ros::spin();
}
```

Appendix G

phylax_navigation

G.1 gmapping.launch

```
<launch>

  <arg name="scan_topic" default="front/scan" />

  <node pkg="gmapping" type="slam_gmapping" name="slam_gmapping"
    output="screen">

    <param name="odom_frame" value="odom"/>
    <param name="base_frame" value="base_link"/>
    <param name="map_frame" value="map"/>

    <!-- Process 1 out of every this many scans (set it to a higher
    number to skip more scans)  -->
    <param name="throttle_scans" value="1"/>

    <param name="map_update_interval" value="0.5"/> <!-- default: 5.0
    -->

    <!-- The maximum usable range of the laser. A beam is cropped to
    this value.  -->
    <param name="maxUrange" value="5.0"/>

    <!-- The maximum range of the sensor. If regions with no obstacles
     within the range of the sensor should appear as free space in the
     map, set maxUrange < maximum range of the real sensor <= maxRange
     -->
    <param name="maxRange" value="10.0"/>

    <param name="sigma" value="0.05"/>
    <param name="kernelSize" value="1"/>
    <param name="lstep" value="0.05"/>
    <param name="astep" value="0.05"/>
    <param name="iterations" value="5"/>
    <param name="lsigma" value="0.0075"/>
    <param name="ogain" value="3.0"/>
```

```xml
    <param name="minimumScore" value="0.0"/>
    <!-- Number of beams to skip in each scan. -->
    <param name="lskip" value="0"/>

    <param name="srr" value="0.01"/>
    <param name="srt" value="0.02"/>
    <param name="str" value="0.01"/>
    <param name="stt" value="0.02"/>

    <!-- Process a scan each time the robot translates this far  -->
    <param name="linearUpdate" value="0.1"/>

    <!-- Process a scan each time the robot rotates this far  -->
    <param name="angularUpdate" value="0.05"/>

    <param name="temporalUpdate" value="-1.0"/>
    <param name="resampleThreshold" value="0.1"/>

    <!-- Number of particles in the filter. default 30        -->
    <param name="particles" value="30"/>

<!-- Initial map size  -->
    <param name="xmin" value="-10.0"/>
    <param name="ymin" value="-10.0"/>
    <param name="xmax" value="10.0"/>
    <param name="ymax" value="10.0"/>

    <!-- Processing parameters (resolution of the map)  -->
    <param name="delta" value="0.045"/>

    <param name="llsamplerange" value="0.01"/>
    <param name="llsamplestep" value="0.01"/>
    <param name="lasamplerange" value="0.005"/>
    <param name="lasamplestep" value="0.005"/>

    <remap from="scan" to="$(arg scan_topic)"/>
  </node>
</launch>
```

Henrik B. Norås Bergel

SLAM Applied to a UGV: design and realization

NTNU
Norwegian University of
Science and Technology

KONGSBERG