Master's thesis

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

Lasse Göncz

# Development of a New Navigation Stack on the NTNU Cyborg

Master's thesis in Kybernetikk og Robotikk
Supervisor: Sverre Hendseth
June 2020

NTNU
Norwegian University of
Science and Technology

NTNU

Norwegian University of Science and Technology

# Development of a New Navigation Stack on the NTNU Cyborg

*Master's thesis in Cybernetics and Robotics*

**Author:** Lasse Göncz

**Supervisor:** Associate Professor Sverre Hendseth

**Project manager:** PhD Candidate Martinius Knudsen

June 2020

# Task Description

The provider of the navigation system on the NTNU Cyborg has gone out of business, leaving deprecated code on the Cyborg's navigational software. Development of a new navigation stack is thus required in order to fulfill the goal of autonomous navigation. The student shall:

- Develop a new navigation stack, replacing the outdated system.
- The navigation stack should perform mapping, localization, path planning, and obstacle avoidance.
- Design the navigation stack in a way that fully replace the inputs and outputs from the old stack, thus not requiring any modification to other modules in the ROS network.
- Focus on the localization system and develop a method for optimizing its performance.

# Preface

The delivery of this paper marks the end of my five year journey as a student at NTNU. Throughout these five years, I've experienced emotions all over the spectrum, from the pure joy of solving a problem I first thought was unsolvable, to the overwhelming and stressful feeling at the reading room after several weeks of studying to four exams. At this point in time, I've not only achieved an engineering degree, but also experiences and friends I will remember for the rest of my life. For that I'm very grateful.

I would like to thank my supervisor, Sverre Hendseth, for his guidance and life stories, and my project manager, Martinius Knudsen, for letting me partake in the NTNU project and co-operating along the way. Finally, I would like to thank the Cyborg team for sharing competency and coffee cups. A copious chapter in my life ends here, time to start writing a new one...

# Abstract

As part of NTNU Cyborg's long-term research effort aimed at developing an autonomous robot interacting with a biological neural network in a closed-loop system, a prototype robot named *The Cyborg* has been under development. Major parts of the current navigation system on the Cyborg consists of legacy code, thus posing limitations on navigational functionality and possibilities for future development. The objectives of this project have been to address those issues by re-implementing the navigation stack with a more robust and flexible design, and optimize the localization system based on a quantitative study.

With the Robot Operating System (ROS) navigation stack as foundation, a network of ROS nodes have been developed to solve the navigation tasks of mapping, localization, path planning, and obstacle avoidance. Various path planning algorithms have been tested and tuned based on experimental data, and the localization system have been improved based on a quantitative study of variance convergence in the estimated pose calculated by the Adaptive Monte Carlo Localization (AMCL) algorithm. The final version of the implemented navigation stack solves all four navigation tasks with a behavior customized for social human-robot interactions. Additionally, the performance of the AMCL algorithm was improved by 55.6% with respect to variance convergence time in the calculated pose estimations. Even though some navigational functionality is lost from the old system, the new navigation stack serve as a solid foundation that allows for a great number of modifications and improvements in the future. The modular design and use of open source code makes the system more robust to bugs, isolated issues, and hardware/software changes.

# Sammendrag

Som en del av NTNU Cyborg's langsiktige mål om å utvikle en autonom robot som kan kommunisere med biologiske nevrale nettverk i en lukket sløyfe konfigurasjon har en prototypet robot med navnet *The Cyborg* vært under utvikling. Store deler av navigasjonssystemet på Cyborgen består av utdatert kildekode, noe som setter funksjonelle begrensninger for navigasjon og muligheter for videreutvikling. Målet for dette prosjektet har derfor vært å re-implementere den utdaterte navigasjonsstacken med et nytt, mer robust og fleksibelt design, samt å optimalisere lokaliseringssystemet basert på en kvantitativ studie.

For å løse de fire primære navigajsonsproblemene med kartlegging, lokalisering, banestrying og kollisjon unngåelse har et nettverk med ROS noder blitt utviklet med grunnalg i Robot Operating System (ROS) sin navigasjonsstack. Ulike banestyringsalgoritmer har blitt testet og konfigurert basert på eksperimentell data, og lokaliseringssystemet har blitt forbedret basert på en kvantitativ studie av konvergenstiden for variansen i de estimerte posisjonene kalkulert av Adaptive Monte Carlo Localization (AMCL) algoritmen. Den endelige versjonen av den re-implementerte navigasjonsstacken løser alle de fire nevnte navigasjonsproblemene med en oppførsel som tilfredsstiller menneske-robot interaksjon. I tillegg har lokaliserings algoritmen blitt forbedret med 55.6 % med hensyn til konvergenstiden for variansen i de estimerte posisjonene fra AMCL algoritmen. Til tross for at ikke alle funksjonene fra det tidligere navigasjonssystemet har blitt implementert, så fungerer det re-implementerte navigasjonssystemet som et fleksibelt grunnlag som muliggjør en mengde forbedringer og videreutvikling i fremtiden. Det modulære designet og bruken av åpen kildekode gjør systemet mer robust mot bugs, programvarefeil og utbytting av maskinvare/programvare.

# Contents

# List of Figures

# List of Tables

# Abbreviations

| | |
|---|---|
| **MEA** | Micro Electrode Array |
| **GUI** | Graphical User Interface |
| **ROS** | Robot Operating System |
| **DARPA** | Defence Advanced Research Projects Agency |
| **CV** | Computer Vision |
| **ARNL** | Advanced Robot Navigation and Localization |
| **IMU** | Internal Measurement Unit |
| **ARIA** | Advanced Robot Interface for Applications |
| **RVIZ** | ROS Visualization |
| **LIDAR** | Light Detection and Ranging |
| **DWA** | Dynamic Window Approach |
| **API** | Application Programming Interface |
| **BFS** | Breadth-first-search |
| **DFS** | Depth-first-search |
| **EKF** | Extended Kalman Filter |
| **MCL** | Monte Carlo Localization |
| **AMCL** | Adaptive Monte Carlo Localization |

| | |
|---|---|
| **KLD** | Kullback-Leibler Divergence |
| **URDF** | Unified Robot Description Format |
| **SLAM** | Simultaneous Localization And Mapping |
| **CPU** | Central Processing Unit |

# 1 Introduction

## 1.1 The NTNU Cyborg

The NTNU Cyborg project is an interdisciplinary project involving the departments of Engineering Cybernetics, Computer Science, and Neuromedicine and Movement Science at the Norwegian University of Science and Technology (NTNU). The project has served as an interdisciplinary research platform for students and employees at NTNU since its beginning in 2015 with the main goal of enabling communication between living nerve tissue and a robot - thereby creating a *Cyborg*. This is done by growing a biological neural network over *Micro-Electrode Arrays* (MEAs). During development, the biological neurons organize into networks and communicate with each other through electrical signals. The MEA captures these signals, enabling an interface between the biological neural network and the robotic system. The robot will ultimately be the mechanical platform of the Cyborg system that the biological neurons communicate through. The purpose is not for the neurons to control all functions on the robot, but rather to perform *simple* tasks. The main challenge is to realize reliable communication between biology and electronics. Should this succeed, then one can start to look at the integration of technology in people with nerve dysfunctions in order to restore lost functionality. Because of this, the research not only promotes technological advancement, but it is also of great importance to the medical sector.

Since reliable communication between the neurons and the robot is yet to be achieved, the current purpose of the Cyborg is to showcase the project, and invoke interest among students in social robots by having an autonomous robot roaming the campus hallways and interacting with people it encounters.

## 1.2 Motivation and Goal

A central part of the NTNU Cyborg project is to develop an autonomous robot able to safely roam the campus hallways. For this purpose, the Pioneer LX robot was purchased in 2015 from Adept MobileRobots to serve as the foundation for future development. Since then, it has undergone several hardware and software iterations in order to realize sufficient navigational behavior. However, the core software library *Advanced Robot Navigation and Localization* (ARNL) used for autonomous navigation has until now remained unchanged.

ARNL was delivered as an embedded software solution for the navigation system on the Pioneer LX robot, with support for, and simple integration with several software applications developed by MobileRobots. This has enabled rapid prototyping of the Cyborg in which limited time and resources have been spent on developing both high- and low-level navigation software. However, the simple integration between MobileRobots' hardware and software comes at the expense of flexibility and configurability since the system is mostly constrained to its existing features, leaving limited possibilities for developers working on the navigation system. This solution, therefore, achieves simple integration at the expense of limited possibilities for future development. Additionally, Adept MobileRobots went out of business in 2018, thus ceasing further development and support for its products.

The NTNU Cyborg project utilizes the Robot Operating System (ROS) framework [1] which is widely used for robotic development, and has a large online community. Since the project's birth in 2015, there has been made continuous progress by ROS developers worldwide to develop navigational functionality for a wide variety of robots. As a result, the ROS navigation stack [2] serve as a go-to foundation for many robot developers, since it provides a great number of useful tools and is supported for most robots. Until now, there has been no research on the Cyborg project with the goal of migrating to a new navigation stack. The goals for this project is therefore to design, implement, test, and tune a navigation system, with the ROS navigation stack as a foundation. Figure 1.1 shows a high-level overview of the software architecture on the Cyborg, and where the navigation stack will be implemented.

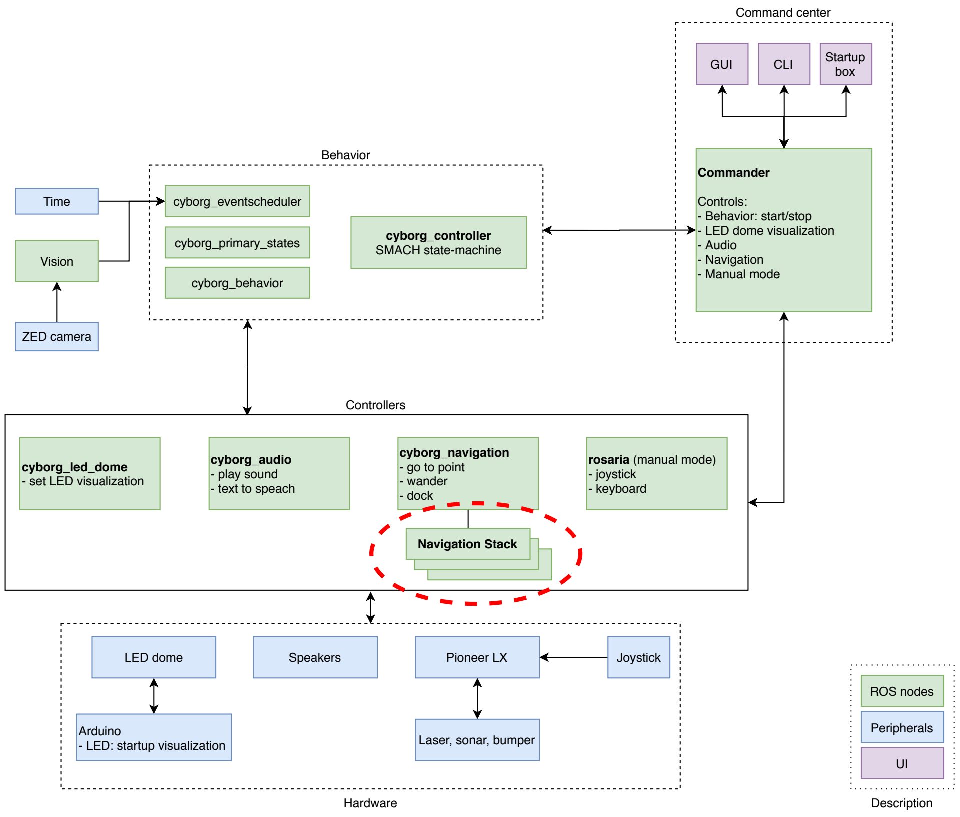

***Figure 1.1:*** *Software architecture on the Cyborg. Red circle marks where the new navigation stack will be implemented.*

## 1.3 The Issues of a Global Pandemic

Midway through the semester, a global pandemic forced a lockdown of the NTNU campus, abruptly ceasing access to the Cyborg and further testing. Even though most of the intended experiments were conducted before the lockdown, some had to be carried out in simulations and some had to be canceled. The initial plan was to configure the path planners and localization modules based on quantitative studies, however, since the lockdown happened before the scheduled experiments for the path planners, the configuration was done based on simulations and visual/empirical results. Additionally, a final test analyzing the performance of the navigation stack as a whole could not be conducted. This could have been done in a simulation, however, because of the discrepancies between the simulations and the live robot it was not prioritized.

## 1.4 Outline

This paper is divided into two parts. Part I includes the necessary background and theory for the succeeding parts of the paper. Part II documents specifications, methodology, results, and other work done by the author.

**Chapter 1** serves as introduction for the thesis. Relevant background information on the NTNU Cyborg project and project goal and motivation are presented here.

**Chapter 2** presents related work in two sections. The first section presents a literature review of relevant papers covering the implementation and design of autonomous systems, as well as and navigational concept. The second section presents the other ongoing Master's project on the NTNU Cyborg.

**Chapter 3** presents necessary background and theory. The first couple of sections aim to explain basic navigation concepts used later in the paper, followed by documentation of current software and hardware.

**Chapter 4** presents the core concepts and tools of the Robot Operating System, which is a fundamental framework on the Cyborg. This chapter aims to explain how the ROS architecture works to get a more intuitive understanding of the succeeding chapters where the different ROS concepts will be used extensively to implement the navigation stack.

**Chapter 5** aims to give a brief introduction to the field of mobile robot path planning. Sections 5.1 to 5.3 presents an overview of what the path planning problem is, followed by section 5.4 and section 5.5 explaining some theory on the A* graph search algorithm and Dynamic Window Approach (DWA).

**Chapter 6** aims to give a brief introduction to the field of mobile robot localization. Section 6.2 presents a literature review of relevant papers on the field of mobile robot localization, followed by a section looking at the differences between localization-based navigation and programmed solutions. Section 6.4 further explains the localization algorithm called Adaptive Monte Carlo Localization (AMCL) which is used on the final version of the localization system.

**Chapter 7** defines specifications and requirements for the design and implementation of the navigation stack.

**Chapter 8** presents the design and implementation of the navigation stack. In this chapter, design and implementation are interwoven, and discussion on both topics will be presented closely. Section 8.1 presents the high-level design of the navigation stack, and the succeeding section focuses closer on the specific elements presented in section 8.1.

**Chapter 9** documents the configuration of the path planners in the navigation stack. This chapter shows how the path planners and costmaps have been tuned based on visual and qualitative analysis.

**Chapter 10** presents a quantitative study of the AMCL algorithm used in the localization system. The chapter documents an experiment looking at convergence properties for variance estimated by the AMCL algorithm. The AMCL node is tuned based on the results presented here.

**Chapter 11** serves as a discussion chapter for the whole paper. Assessment of both general and specific aspects of the project will be discussed in this chapter. Some thought for future development is also included here.

**Chapter 12** gives some concluding remarks of the project.

# Part I

# Background and Theory

# 2 | Related Work

## 2.1 Literature Review

Due to the complexity of autonomous navigation systems, most of the work presenting a complete application has - in one way or another - implemented its own system architecture and navigation framework. Early papers from the eighties [3, 4] describes a complete system able to navigate outdoors in a large environment, where Carnegie Mellon University (CMU) researchers developed a system to autonomously navigate through a network of sidewalks and intersections in the CMU campus. More recently, the Defence Advanced Research Projects Agency (DARPA) grand challenge in 2005 [5], and the DARPA urban challenge in 2009 [6] boosted the development of autonomous cars, resulting in several contributions to the field and several papers from participant teams [7, 8, 9] describing the architecture and design of their navigation system. Observing these papers show that the different architectures share similar features like parallel communications, processes, tasks, etc., which could be reusable between them. This was in fact one of the major factors that motivated the creation of ROS [1]. The ROS navigation stack is the most well-known and widely spread framework to develop autonomous navigation applications. It provides a variety of useful tools, but it also has some limitations [10]: it is solely designed to work on differential drive and holonomic robots, and it assumes that the robot can be controlled by sending x, y, and theta velocities. Additionally, it requires a planar laser for mapping and localization purposes, and it performs best with robots with a circular shape. Other recent literature presents some complete applications like [11, 12], however, these papers aim to solve specific problems and are thus not designed for general purpose. Efforts in producing more general frameworks for different levels of autonomous systems can also be found in [13] where the focus was high-level project management and software development, and in [14] where the focus was low-level trajectory planning and obstacle avoidance in car-like robots. A generic framework is presented in [15] as an alternative to the ROS navigation stack, however, it focuses mainly on planning and control of wheeled robots that have various kinematic constraints instead of covering the whole navigation problem.

## 2.2 Master's Pojects on the NTNU Cyborg Spring 2020

### 2.2.1 Graphical User Interface by C. Nilsen

C. Nilsen is developing a Graphical User Interface (GUI) module for the Cyborg. The goal for this module is to support remote control and monitoring of the Cyborg. His solution focuses on a cloud-based, reactive single-page-application that is built and tested using the Vue framework. His work allows the Cyborg to be remotely controlled in real-time with a click-to-send interactive map. Additionally, the GUI supports teleoperation control with an on-screen joystick. The GUI also enables monitoring and control of states in the Cyborg's behavior system.

### 2.2.2 Computer Vision technology by O. M. Brokstad

O. M. Brokstad is developing a Computer Vision (CV) module for the Cyborg. This includes the development and configuration of vision hardware and object detection software. His project is motivated by the several advantages of improving interactions between the Cyborg and people it encounters. The aim is to implement a system capable of detecting and classifying human behaviors, allowing the Cyborg to become more socially intelligent.

### 2.2.3 Behavior system by J. Kalland

J. Kalland is researching on the use of behavior trees and how to implement them in the Cyborg's software. Her work also includes augmenting the visual and auditory functions on the robot, as well as researching the use of PAD values in the Cyborg's behavior.

# 3 | Background

## 3.1 Introduction

This chapter aims to give some brief background information on concepts within navigation theory (section 3.2), followed by documentation of the current software and hardware architecture (section 3.3 and section 3.4, respectively).

## 3.2 Navigation Theory

For any mobile robot, the ability to navigate in its environment is essential. When a mobile robot wants to move to a specific location, it must find that location and calculate a path that it can move along while simultaneously avoiding obstacles. This section will briefly present the basic concepts of a robot navigation system, which refers to the robot's ability to plan a path towards a goal location given its position relative to a reference coordinate frame. These two fundamental concepts are *localization* and *path planning*.

### 3.2.1 Localization

Robot localization is the process of determining where a robot is with respect to its environment. Localization is a fundamental concept in autonomous robots since positional awareness is essential in order to make decisions about future actions. In a typical robot localization system, a map of the environment is available, and the robot is equipped with sensors capable of sensing the environment and monitoring the robot's motion. The localization problem then becomes the task of estimating the position and orientation of the robot in the map by using these sensors. Since the sensor readings rarely exhibit exact values, the localization system needs to be able to deal with noisy data and generate not only an estimate of the robot's location but also an uncertainty measure of the location estimate.

### 3.2.2 Path Planning

Planning a path from location $x$ to a location $y$ while avoiding obstacles and reacting to environmental changes might be a trivial task for a human, but not so straightforward for an autonomous robot. A robot can use different sensors to perceive the environment and to update its environment map, both with a level of uncertainty. In order to calculate some motion actions that lead to the desired goal location, it can use a variety of different decision and planning algorithms that take into account the specific robot's kinematic and dynamic constraints.

Path planning is used in different fields where the environment is either fully known, partially known, or entirely unknown. All cases are still active fields of research, where different methods and algorithms are developed to solve a specific path planning problem. For the Cyborg, the environment will, for the most part, be partially unknown.

### 3.2.3 Odometry

Odometry is the use of data from motion sensors to estimate the change in position over time. It is used in robotics to estimate the position of the robot relative to a starting location. On a wheeled robot like the Cyborg, the odometry is often calculated from wheel encoders and/or Internal Measurement Units (IMU). Wheel encoders can measure how far the wheels have rotated, and based on the circumference of the wheels, it can compute traveled distance. However, this method is sensitive to errors since what essentially happens is the integration of velocity measurements over time to give position estimates. Therefore, precise data collection from sensors, instrument calibration, and processing is required for odometry to be used effectively.

### 3.2.4 Costmap

The pose of a robot, and its distance to obstacles, are estimated based on the odometry and readings from its sensors. Using this information, an *occupancy grid map* can be generated by a mapping algorithm to define the occupied, free, and unknown area in the environment. A costmap takes the occupancy grid map as input and calculates movable-, possible collision-, and obstacle area when taking into account the shape of the specific robot.

The costmap is generally divided into two separate costmaps: a *global costmap* used to calculate a path plan for navigating the global area of the fixed map, and a *local costmap* used for path planning and obstacle avoidance in the robots local area. Although the purpose of the global and local costmaps differ, they are represented in the same way. Both costmaps consists of cells with values ranging from $0 - 255$ that are used to identify whether the robot is movable or colliding with an obstacle. How these calculations are made depends on the developers configuration (see section 9.4 for the Cyborgs costmap configuration). Figure 3.1 show the relationship between costmap values and the corresponding distance to obstacles.

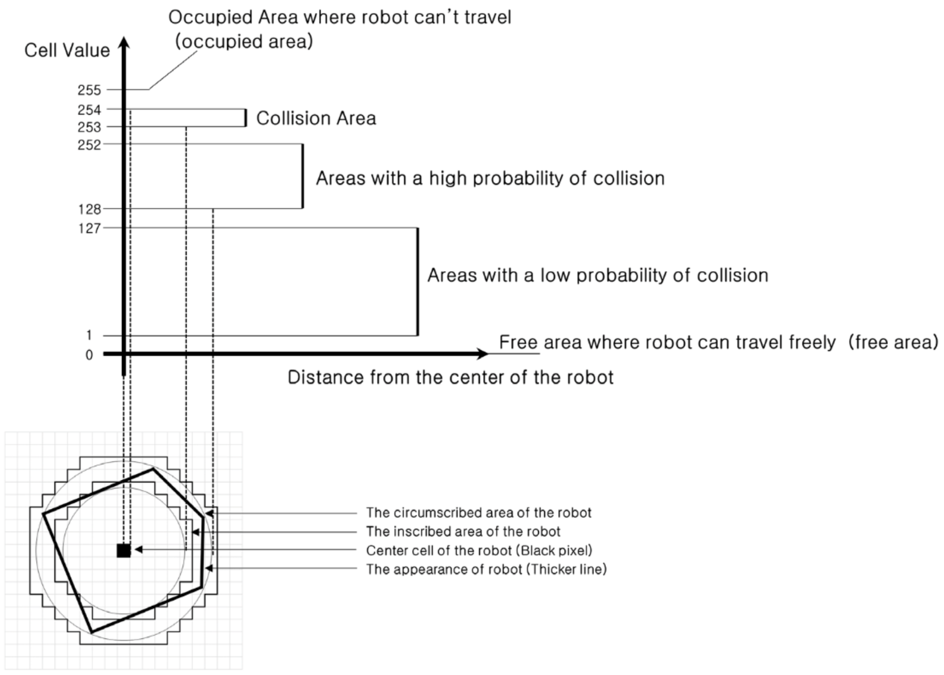
Occupied Area where robot can't travel
(occupied area)
Cell Value
255
254
253
252
128
127
1
0
Collision Area
Areas with a high probability of collision
Areas with a low probability of collision
Free area where robot can travel freely (free area)
Distance from the center of the robot
The circumscribed area of the robot
The inscribed area of the robot
Center cell of the robot (Black pixel)
The appearance of robot (Thicker line)

***Figure 3.1:*** *Relation between costmap values and distance to obstacle [16].*

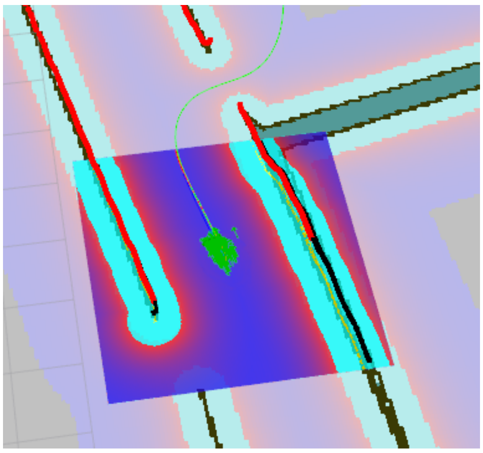

***Figure 3.2:*** *Visualization of the costmap in RVIZ.*

Visualizing the costmap can be useful to get a sense of what the navigation system can "see" and where it is desirable to move. In fig. 3.2, the costmap is visualized as a color gradient ranging from dark blue (area with a low probability of collision) to red (area with a high probability of collision) to turquoise (collision area). The square shape with strong colors about the robot represents the *local* costmap, whereas the faded colors on the rest of the map represent the *global* costmap.

### 3.2.5 Coordinate Transforms

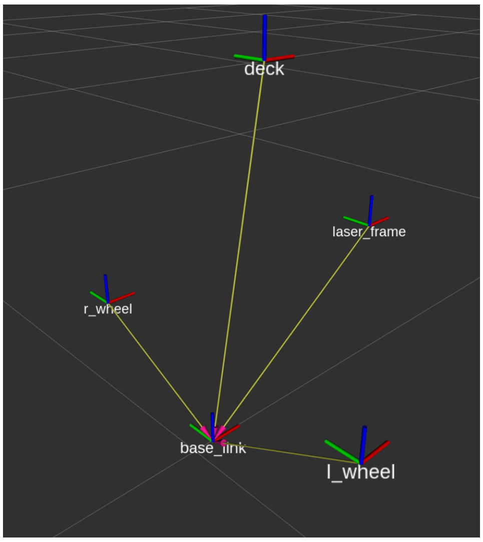

***Figure 3.3:*** *Five of the coordinate frames on the Cyborg.*

Coordinate transforms is an important concept to understand when it comes to robotics. A robotic system might contain several subsystems with certain functionality, and to predictably control these subsystems, knowledge about their respective coordinate frames in relation to each other is essential. Otherwise, the objective of controlling them would be impossible.

An example of two subsystems often found on a robot is its wheels and laser sensor. The laser senses the environment to perform navigation, and the wheels are used to move the robot around. Since the wheels and laser are not located at exactly the same spot on the robot, the navigation system needs to be provided with this information to make use of them.

The wheels and laser are obviously not located at exactly the same spot on the robot, so in order to make use of them, information about exactly *where* they are on the robot is needed. Specifically, information about where on the robot they are placed and what way they are facing, in other words, their *position* and *orientation*.

- **Position:** A vector of three numbers $(x, y, z)$ that describes how far an object has traveled along each axis with respect to some origin.

- **Orientation:** A vector of three numbers $(roll, pitch, yaw)$ that describes how far an object has rotated about each axis with respect to some origin.
- **Pose:** The position and orientation paired together is called the pose. The pose varies in six dimensions and is sometimes referred to as a 6D pose.

A common convention is to reference the *origin* of the different parts in relation to a common origin on the robot base. Such a reference frame is usually chosen to be the geometric centroid on the robot. For the Cyborg, this reference frame is called "`base_link`", and is located on ground level, in the center between the two wheels. The positive x-axis is pointing forward, the positive y-axis is pointing left, and the positive z-axis is pointing up. Figure 3.3 shows five of the coordinate frames on the Cyborg. Note that all frames are connected to `base_link` by yellow lines. These represent the *transformation* between the `base_link` coordinate frame and the other frames.

## 3.3 Software

### 3.3.1 The Navigation Module

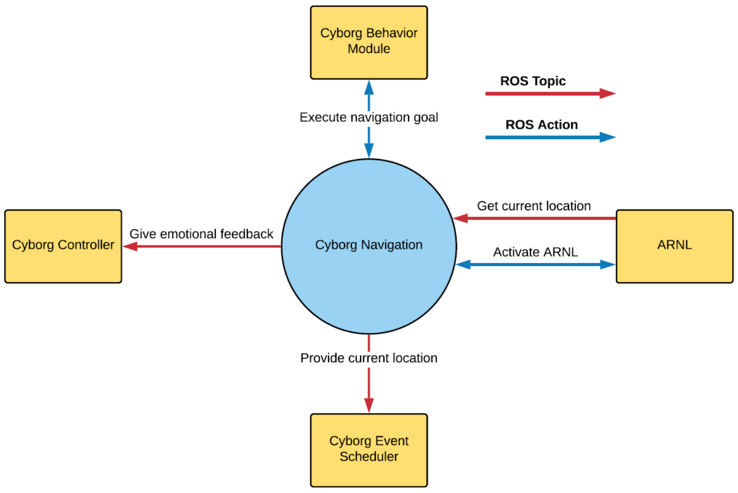

***Figure 3.4:*** *Context diagram of the navigation module prior to the work done in this project [17].*

The software architecture on the Cyborg consists of several modules responsible for high level control of auditory, visual, emotional, and navigational behavior. The most recent work on the NTNU Cyborg project implemented a *navigation module* responsible for high level navigation control of the Cyborg. It's main purpose is to interface the navigation controller (previously ARNL) with other modules in the network. Figure 3.4 shows a context diagram of the navigation module as a result of A. Babayan's work in [17]. The module interfaces the ARNL node through an *action interface* and receives current location updates on a *ROS topic*. These concepts will be explained in chapter 4. Figure 1.1 shows where the navigation module will be placed in the new system architecture.

### 3.3.2 ARIA

*Advanced Robot Interface for Applications* (ARIA) is the core development library on the robot. Written in the C++ language, ARIA is a client-side software for easy, high-performance access to and management of the Pioneer LX base, as well as the many accessory robot sensors and effectors. This makes it possible to control navigational parameters and receive sensory data from its internal sensors. Accessing ARIA can be done either through low-level commands or through its high-level action infrastructure. Supported programming languages are Python, Java, and MATLAB. ARIA automatically handles all communication with the components on the robot including (but not limited to) the laserscanner, sonar, and bumpers sensor, by sending and receiving messages with the robot's embedded firmware.

### 3.3.3 RVIZ

ROS visualization - or RViz - is a general purpose 3D visualization environment for robots, sensors, and algorithms. This tool is widely used for robots developed with ROS because it can be used for any robot and is configurable for any particular application. Data can easily be visualized in RViz by subscribing to topics that have built-in plugins for ROS visualization.

### 3.3.4 MATLAB

MATLAB is a programming platform designed specifically for engineers and scientists, and is widely used to analyze data, develop algorithms, and create models and applications, among other things. The basic data element in MATLAB is an array that does not require dimensioning. This allows for solving technical computing problems, especially those with matrix and vector formulations, in a fraction of the time it would take to write a program in a scalar non-interactive language such as C.

Several tools have been developed for MATLAB. The ROS Toolbox enables access to ROS functionality in MATLAB, such as communication with a ROS network, interactively explore robot capabilities, and visualize sensor data. During this project, MATLAB and its ROS toolbox have been used to analyze and plot data.

## 3.4 Hardware

### 3.4.1 Pioneer LX - The Cyborg Base

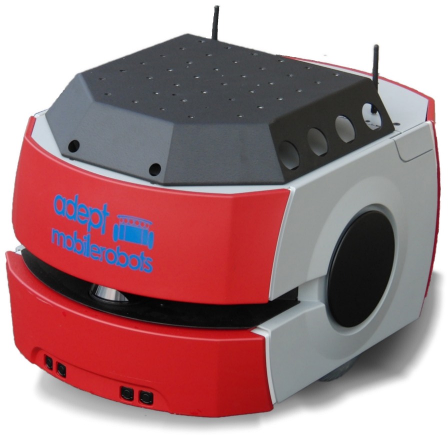

***Figure 3.5:*** *The Pioneer LX base.*

The MobileRobots Pioneer LX is an autonomous navigation robot developed by Adept MobileRobots. It is capable of carrying loads up to 60 kg and serves as the Cyborg's navigational base. It includes a robot control system and embedded computer capable of running Linux, differential drive system with encoder feedback, as well as a laser rangefinder sensor, ultrasonic sensors, and bumper sensors. The robot base is delivered with a software development kit including pre-installed software and tools for control, navigation, and communication.

The Pioneer LX has two degrees of freedom and is controlled by requesting translational and rotational velocity. The Pioneer LX controller automatically controls the drive system to achieve the requested velocities. The controller uses encoders to automatically integrate wheel odometry to maintain requested velocity, and combined with data from an internal gyroscope, computes an estimation of the robot pose [18]. The Pioneer LX features the following hardware:

- Intel D525 64-bit dual core CPU @1.8 GHz
- Intel GMA 3150 integrated graphics processing unit
- Intel 6235ANHMW wireless network adapter
- Ports for ethernet, RS-232, USB, VGA, and various other analog and digital I/O
- SICK 300 and SICK TiM 510 laser scanner
- Sonar sensors and a bumper panel
- Joystick for manual control

- A 60 Ah battery
- Chargin station

### SICK S300 Laser Scanner

Light Detection And Ranging (LiDAR) sensors, also referred to as laserscanners, is a sensor used to measure distance to objects by using a laser as its source. Laserscanners have the advantage of high speed, high performance, real time data acquisition, and is widely used in the field of robotics. They work by calculating the difference of the wavelength when the laser source is reflected by an object, and they often measure a windows between 180 to 360 degrees. Even though it is not necessary to know how exactly a LiDAR works, it is important to be aware of possible limitations and warnings: First, the strong laser beam used as light source can be damaging for the eye. Secondly, surfaces like glass and transparent plastic tend to reflect and scatter the light source in many directions, leaving inaccurate measurements. Lastly, only the horizontal plane is scanned, thus resulting in 2D data.

The on-board SICK S300 laserscanner is a precise scanning sensor that provides 500 readings in a 250-degree field of view with a maximum range of 15 meters. The laser operates in a single plane positioned about 19.1 cm above the floor.

### Sonars

The Pioneer LX contains four short-range sonar sensors for extra sensing near the floor both in the front and rear. The rear sonar is especially useful when docking since the robot then has to back up onto the docking station.

### Bumpers

A bumper panel with two pairs of sensors is mounted at the front of the base, should the navigation system fail and crash into an obstacle. It can then indicate a left, middle or right side bumper hit.

### Encoders and Gyroscope

Each wheel on the robot has an encoder that tells how far the wheels have turned, and in which direction. Each wheel also has a Hall sensor, and the core contains a gyroscopic sensor to measure rotation. These sensors are used to calculate the odometry, and they provide feedback to the robot controller as it maintains the requested velocities.

# 4 | Robot Operating System

## 4.1 Introduction

This chapter presents the necessary background theory on the *Robot Operating System* framework, which is a fundamental concept to understand for the succeeding parts of this paper. The chapter is organized as follows: Section 4.2 gives a brief explanation of what ROS is and why it is useful for robot developers. Section 4.3 presents the ROS architecture and the three main levels of concept. This section explains the concepts of nodes, messages, topics, etc., and how the processes in a ROS network are connected. section 4.4 introduces the ROS graph, which is a useful tool when working with ROS.

## 4.2 What is The Robot Operating System?

The Robot Operating System, or ROS in short, is an open-source framework widely used in the field of robotics. The purpose of ROS is to serve as a common software platform for developers who are building different kinds of robots. The platform enables people to share code with certain functionality that with minor changes can be implemented in another robotic system. A common phrase to describe the main benefit of ROS is that developers do not “reinvent the wheel”. The framework provides services like hardware abstraction, low-level device control, message-passing between processes, implementation of commonly-used functionality, and package management. It also provides libraries and tools for obtaining, building, writing, and running code across multiple computers [19].

The argument for using the ROS framework is that it provides all the parts of a robot software system that would otherwise have to be written manually. It allows the developer to focus on the parts of the system they *do* care about without spending an excessive amount of time with the parts they *don't* care about.

## 4.3 The ROS Architecture

The ROS architecture can essentially be divided into two conceptual levels. The *filesystem level* and the *computation graph level*. These will be further explained in the following sections. In section 4.3.1, some concepts will be used to explain how ROS is formed internally, i.e. the folder structure and required files it needs to work, and section 4.3.2 will present how processes and systems communicate with each other.

### 4.3.1 The Filesystem Level

ROS is often referred to as a meta-operating system since it not only offers tools and libraries, but also functions often seen in operating systems like hardware abstraction, package management, and a developer toolchain. Similar to a real operating system, ROS files are organized in a particular manner, depicted in fig. 4.1.

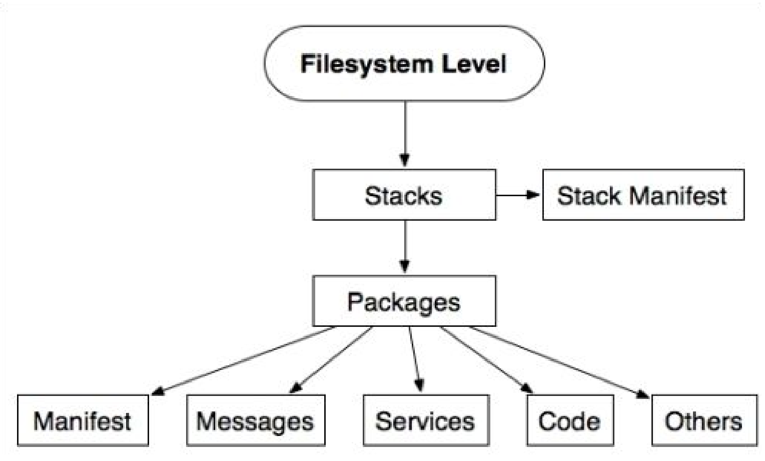

***Figure 4.1:*** *The ROS filesystem level*

#### Packages

Packages are the main unit for organizing software in ROS. They form the atomic level and has the minimum structure and content needed to create a program within ROS. A package may contain runtime processes (nodes), libraries, configuration files, and so on. The goal of a package is to provide just enough functionality such that it can easily be reused. There are several tools for managing packages, all of which are well documented on the ROS Wiki website [20]. A common convention for the directory-structure of ROS packages looks like this:

- `include/package_name`: Includes the headers of required libraries.
- `msg/`: Contains the message types.

- `src/package_name/`: Contains the source files of the programs.
- `srv/`: Contains the service types.
- `scripts/`: Contains the executable scripts.
- `CMakeLists.txt`: This is the CMake build file which is the input to the CMake build system for building packages.
- `manifest.xml`: This is the package manifest file that defines properties about the package such as version number, package name, authors, dependencies and catkin packages.

### Manifests

The manifest file (`manifest.xml`) is found in a package directory and it contains a minimal specification about the package. The main role of this file is to declare dependencies in a language-neutral and operating-system neutral way. The most used tags in the manifest file are `<depend>` that shows which packages that must be installed before installing the current package, and `<export>` which tells the system what flags should be used to compile it.

### Stacks

Packages in ROS are organized into *stacks*. While the goal of packages is to create minimal collections of code for easy re-use, the goal of a stack is to simplify the process of code sharing, thus being the primary mechanism in ROS for distributing software. Stacks collect packages that together provide some kind of functionality, e.g. a navigation stack. They need a basic structure of files and folders which can be created manually or with the command tool `roscreate-stack`.

### Stack Manifests

Similar to the manifest file of a package, the stack manifest file (`stack.xml`) provides metadata about the stack and declares dependencies on other stacks.

### Message types

ROS uses a simplified message description language for describing the data values that ROS nodes publish. With this description, ROS can generate the right source-code for these types of messages in several programming languages. There are two parts to a message file: *fields* and *constants*. Fields define the type of data to be transmitted in the message, e.g. `string` or `int32`, while the constants define the name of the fields. A table of the supported standard built-in types can be found in the ROS Wiki [20]. Listing 1 shows an example of a `.msg` file.

```
int32 id
string name
float32 vel
```

***Listing 1:*** *An example of a message file.*

### Service types

ROS uses a simplified service description language for describing ROS service types. It builds upon the message format to enable request/response communication between nodes. The service descriptions are stored in the `srv/` sub-directory of a package as `.srv` files.

## 4.3.2 The Computation Graph level

ROS creates a network where all the processes are connected. The basic concepts of a computation graph are *nodes*, *master*, *parameter server*, *messages*, *services*, *topics* and *bags*, all of which provide data to the graph in different ways.

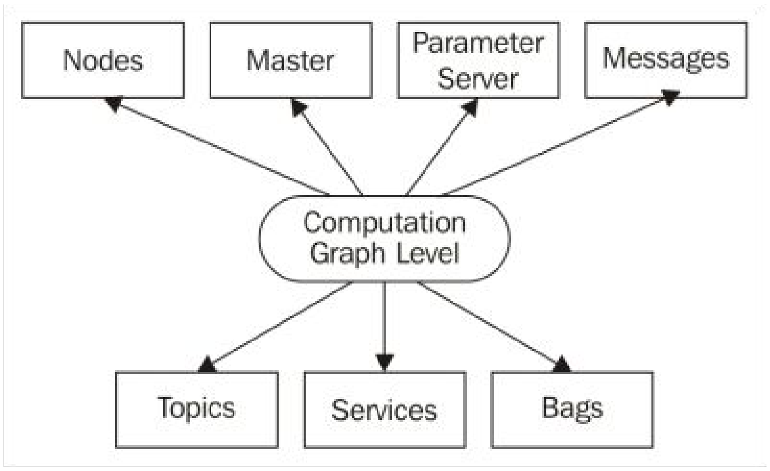

***Figure 4.2:*** *The ROS Computation graph level.*

### Nodes

A node is essentially a process that performs computation. A typical robot control system will comprise many nodes that control different functions, i.e. one node for controlling the wheel motors, one node to perform localization, one node to perform path planning, and so on. A good convention is to have many nodes that perform a specific function rather than a large node that makes everything happen in the system.

The use of nodes provides several benefits to the system: Debugging becomes easier since the node separates the code and functionalities such that crashes are isolated to individual nodes. The code complexity is also reduced compared to monolithic systems where functionally distinguishable aspects are interwoven. Another powerful feature of ROS nodes is the possibility to change parameters when starting the node, i.e. the node name, topic names, and parameter values. This is a useful way of re-configuring the node without having to recompile the code. Nodes communicate with each other using *topics*, *services* and the *parameter server*, all of which will be further explained in the next couple of sections.

### Master

The ROS Master is a name service for ROS. It keeps track of all the running nodes, topics, and services available, and enables nodes to locate one another. Once the nodes have located each other, they communicate in a peer-to-peer fashion. The ROS master makes communication between nodes simple by initializing all the messages and services without actually connecting the nodes, as illustrated in fig. 4.3.

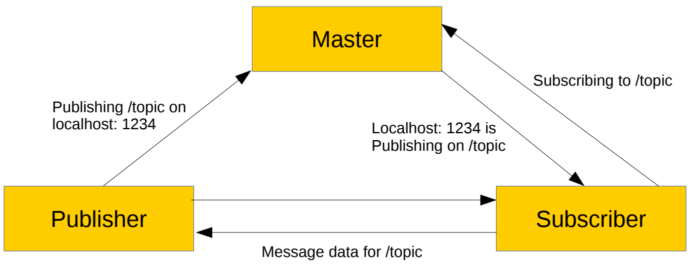
Master
Publishing /topic on localhost: 1234
Subscribing to /topic
Localhost: 1234 is Publishing on /topic
Publisher
Subscriber
Message data for /topic

***Figure 4.3:*** *Illustration of how the ROS master enables communication between a publisher and subscriber.*

### Parameter Server

The parameter server is a dictionary that nodes use to store and retrieve parameters at runtime. Since it's not designed for high-performance, it is best used for storing static data such as configuration parameters. The parameter server is meant to be globally viewable and is accessible via network application programming interfaces (APIs) such that the configuration state of the system can be monitored and modified if necessary. The provided command-line tool *rosparam* can be used to access and modify the stored parameters. Some commonly used code for the parameter server in Python are listed in listing 2.

```python
# Get parameter
value = rospy.get_param('/node_name_space/parameter_name')

# Check existence of parameter
rospy.has_param('parameter_name')

# Set parameter
rospy.set_param('parameter_name', parameter_value)
```

***Listing 2:*** *Commonly used code for the parameter server in Python.*

### Messages

Nodes communicate with one another by publishing messages to *topics*. A message is a simple data structure supporting standard primitive types like integers, floating points, Boolean, as well as arrays and customized types developed by the user. The format of a message file is simply a field and a constant on each line as shown in listing 1.

### Topics

Topics are buses used by nodes to transmit data in a publish/subscribe fashion intended for unidirectional, streaming communication. A node can send a message by *publishing* it to a given topic, and a node that wishes to receive this data can *subscribe* to the same topic, given that it has the same message type as the publisher. This way of communication decouples the publisher from the receiver, resulting in nodes not necessarily knowing whom they are communicating with. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics.

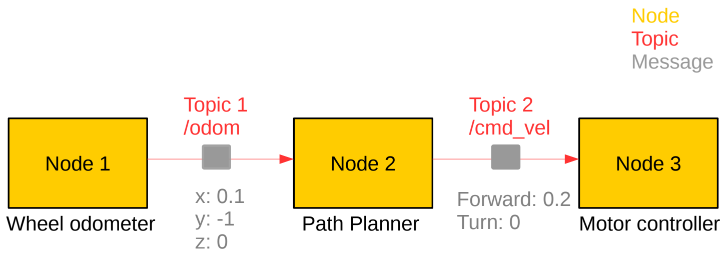

***Figure 4.4:*** *Illustration of ROS nodes, topics and messages.*

Figure 4.4 is depicting how the concept of nodes, topics, and messages work in an example with three nodes responsible for a specific navigation function. This system of nodes will together perform a simple navigation task. Declaration of simple publishers and subscribers in Python is shown in listing 3 and listing 4.

```
1 def talker():
2     pub = rospy.Publisher('chatter', String, queue_size=10)
3     rospy.init_node('talker', anonymous=True)
4     rate = rospy.Rate(10)
```

***Listing 3:*** *Declaring a publisher node in Python.*

Line 2 in listing 3 declares that the node is publishing to the `chatter` topic using the message type `String`. The `queue_size` limits the amount of queued messages if any subscriber is not receiving them fast enough. Line 3 tells rospy the name of the node, in this case, `talker`. Line 4 creates a `Rate` object, `rate`, which is a convenient way of looping at the desired rate (10 Hz in this case).

```
1 def callback(data):
2     rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)
3
4 def listener():
5     rospy.init_node('listener', anonymous=True)
6     rospy.Subscriber("chatter", String, callback)
7     rospy.spin()
```

***Listing 4:*** *Declaring a subscriber node in Python.*

The code for the subscriber is similar to that of the publisher, except a new callback-based mechanism is introduced for subscribing to messages. Line 6 in listing 4 declares that the node is subscribing to the `chatter` topic. When a new message is received, the `callback` function is invoked with the message as the first argument. Line 5 tells rospy the name of the node. The `anonymous=True` flag tells `rospy` to generate a unique name for the node such that multiple `listener` nodes can run easily. Line 7 simply keeps the node from exiting until the node has been shutdown.

### Services

In cases where it necessary to communicate with nodes and receive a reply, topics do not suffice since they work in a unidirectional fashion. This request/reply model is realized via *services*. Services are just synchronous remote procedure calls - they allow one node to call a function that executes in another node. Service calls are well suited to things that only need to be executed occasionally, and that take a bounded amount of time to complete. An example of this can be a discrete action that a robot might do, such as taking a picture with a camera or turning on a sensor.

A service is defined by a pair of messages - one for the request and one for the reply. A node can offer a service under a specific name that a client can call by sending it a request message. A client can also make a persistent connection to a service, which enables higher performance at the cost of less robustness to service provider changes [21].

Listing 5 shows how a simple service node can be written. The node is declared in line 5 with the name `add_two_ints_server`, and the service is declared in line 6. This line declares a new service named `add_two_ints` with the `AddTwoInts` service type. All requests are passed to `handle_add_two_ints` function which returns instances of `AddTwoIntsResponse`. Additionally, just like the subscriber in the previous section, `rospy.spin()` on line 7 keeps the code from exiting until the service is shut down.

```python
1 def handle_add_two_ints(req):
2     return AddTwoIntsResponse(req.a + req.b)
3 
4 def add_two_ints_server():
5     rospy.init_node('add_two_ints_server')
6     s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
7     rospy.spin()
```

***Listing 5:*** *Service node example in Python.*

Listing 6 shows how a simple client node can be written. Line 2 is a convenient method that blocks until the service named `add_two_ints` is available. On line 4 the handle `add_two_ints` is created, which can be used just like a normal function in Python. The exception on Line 7 will run if the call fails.

```python
1 def add_two_ints_client(x, y):
2     rospy.wait_for_service('add_two_ints')
3     try:
4         add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
5         resp1 = add_two_ints(x, y)
6         return resp1.sum
7     except rospy.ServiceException, e:
8         print "Service call failed: %s"%e
```

***Listing 6:*** *Client node example in Python.*

### Actions

ROS services are useful for synchronous request/response interactions in the cases where the asynchronous ROS topics don't fit. However, services aren't always the best fit either, particularly when the request is more than a just a "set the value of $x$" instruction. An example of this the case when the robot is tasked to move to a specific goal location. In the case of services, a caller sends a request containing the goal location, then waits for an unknown amount of time to receive the response of what happened. When using services, the caller has no information about the robot's progress towards the goal, and the caller can't cancel or change the goal. ROS *actions* address these shortcomings.

ROS actions are intended at controlling time-extended, goal-oriented behaviors like in the case mentioned above. Unlike synchronous services, actions are *asynchronous*. An action uses a *goal* to initiate a task, and sends a *result* when the task is complete. It also uses *feedback* to provide updates on the progress towards the goal, and also allows for the goals to be *cancelled*. In the case of a moving robot, a goal could be a message that contains information about where the robot should move to in the world. The feedback provided could be the robot's current pose along the path, and the result could be the final pose of the robot. These three parameters are defined in an *action specification* file. The layout of this `.action` file is shown below:

```
# Define goal
goaltype goalname
---
# Define result
resulttype resultname
---
# Define feedback
feedbacktype feedbackname
```

***Listing 7:*** *Example layout of an action specification file.*

The *ActionClient* and *ActionServer* communicate via a *ROS action protocol*. The client and server then provide a simple API in order to request goals on the client side, or to execute goals on the server side, via function calls and callbacks. Figure 4.5 illustrates this concept.

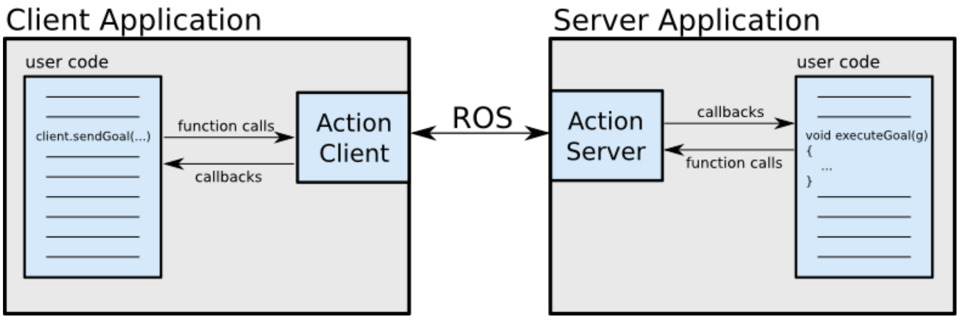

***Figure 4.5:*** *Client-Server interaction via the ROS action protocol [22].*

```
1  from example.msg import example_action, example_goal
2
3  # Create client and connect to server
4  rospy.init_node('example_client', example_action)
5  client = actionlib.SimpleActionClient('example', example_action)
6  client.wait_for_server()
7
8  # Create and send goal
9  goal = example_goal()
10 client.send_goal(goal)
11 client.wait_for_result(rospy.Duration.from_sec(5.0))
```

***Listing 8:*** *Simple ActionClient in Python.*

The code in listing 8 shows how to set up a simple ActionClient and send a goal to an ActionServer. Line 1 imports the action type and messages. Line 4 Initializes the node with the name `example_client`. Line 5 initializes the client and connects it to the action server with type `example_action`. Line 6 waits until the client is properly connected to the server before execution. Line 9 creates the goal and on line 10 the goal is sent to the server. Line 11 waits for the result for 5.0 seconds.

```
1  from example.msg import example_action, exaple_result
2
3  # Create and start server
4  ActionServer = actionlib.SimpleActionServer('server', example_action, execute,
   ↪  auto_start = False)
5  ActionServer.start()
6
7  def execute(goal):
8      # Implement functionality for the robot here
9      ActionServer.set_succeeded()
```

***Listing 9:*** *Simple ActionServer in Python.*

The code in listing 9 shows how to setup a simple ActionServer. Line 1 imports the action type and messages. Line 4 creates the ActionServer named `server` with action type `example_action`. The function `execute` runs when a goal arrives. Line 5 starts the ActionServer. Line 7 defines the function `execute` where the functionality of the action is implemented. Line 9 sets the terminal state of the ActionServer and publishes the result message to the client.

### Rosbag

A Rosbag is a file created by ROS to store message data. A variety of tools have been developed for bag-files, making it possible to store, process, analyze, and visualize the data. They are commonly used to “record” a session in ROS in order to reproduce the same exact same data transmissions when analyzing or debugging algorithms. This is done by sending the topics and messages at the same time as when they were recorded.

## 4.4 The ROS graph

A simple and intuitive way of illustrating the current state of a ROS session is with a directed graph depicting running nodes and the publisher-subscriber connections between those nodes through topics. A tool for generating such a graph is the **`rqt_graph`** [23]. The ROS graph in fig. 4.6 shows an example of how such a graph might look like. The graph in this figure is actually showing the ROS graph of the navigation stack during the development process.

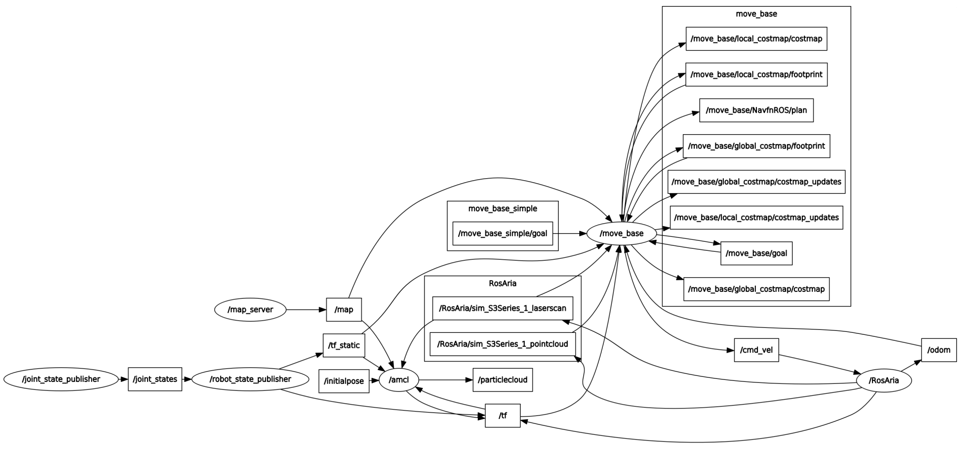
move_base
/move_base/local_costmap/costmap
/move_base/local_costmap/footprint
/move_base/NavfnROS/plan
/move_base/global_costmap/footprint
/move_base/global_costmap/costmap_updates
move_base_simple
/move_base/local_costmap/costmap_updates
/move_base_simple/goal
/move_base
/move_base/goal
RosAria
/move_base/global_costmap/costmap
/map_server
/map
/RosAria/sim_S3Series_1_laserscan
/tf_static
/RosAria/sim_S3Series_1_pointcloud
/cmd_vel
/odom
/joint_state_publisher
/joint_states
/robot_state_publisher
/initialpose
/amcl
/particlecloud
/RosAria
/tf

***Figure 4.6:*** *Example of a ROS graph. Oval shapes represent nodes, rectangles represent topics.*

# 5 Mobile Robot Path Planning

## 5.1 Introduction

Moving from one place to another is a trivial task for a human who can interpret and calculate how to move in a split second. For a robot, however, such an elementary task is a major challenge. The problem of *path planning* is a fundamental problem in the field of autonomous robotics - namely finding a path for the robot to move along while avoiding obstacles. Safe and efficient robot navigation requires strong and robust path planning algorithms since the generated path greatly affects the performance of the robot application. The principal objective of the navigation process is to minimize the traveled distance as this also influences other metrics such as energy consumption and processing time.

This chapter presents a brief overview of mobile robot path planning and provides the necessary background on this topic for the succeeding parts of the paper. Section 5.2 and section 5.3 aims to give a brief introduction to the path planning problem and the difference between local and global planning. Section 5.4 presents the A* graph search algorithm used in the navigation stack, followed by a brief explanation of the Dynamic Window Approach (DWA) in section 5.5.

## 5.2 Overview of the Path Planning Problem

Recent years have seen a revolution in robotics. A variety of robotic systems have been developed, and they have shown their effectiveness when performing different tasks in different areas such as factory robotics, airports, home environment, and so on. The robot needs to be embedded with intelligence to ensure optimal executing of the task at hand. However, implementing intelligence in robotic systems imposes a huge number of research challenges, navigation being one of the most fundamental ones. For a robot to successfully finish the navigation task, it has to know its position relative to the position of its goal. Additionally, it has to consider its immediate environment and be able to dynamically adjust its actions in order to reach its goal. In other words, to solve the navigation problem, the robot needs to know the answer to the three following questions: *Where am I? Where am I going? How do I get there?*. These questions relate to the three fundamental navigation concepts *localization*, *mapping* and *path planning*, respectively.

- **Localization:** The ability of the robot to determine its location in the environment. The location can be presented as a reference relative to a local environment (e.g. center of a hallway), topological coordinate (e.g. in room 12), or in absolute coordinates (e.g. longitude, latitude, altitude).
- **Mapping:** To identify where the robot has been moving so far, it requires a map of its environment. The map can either be placed directly into the robot's memory, or it can be gradually generated when the robot explores and senses its environment (Simultaneous Localization and Mapping).
- **Path planning:** To find a path for the robot, the goal position must be given in advance which requires an addressing scheme that it can follow. The addressing scheme indicates where the robot has to go from its starting position. A robot might for example be requested to go to a certain location on a school campus by simply giving it the location name (e.g. cafeteria). In other scenarios, addresses can be given in relative or absolute coordinates.

Path planning is the aspect of navigation that answers the question: *What is the best way to get there?* There are, however, several issues that need to be considered in the path planning problem, as shown in fig. 5.1. Most of the proposed solutions in previous research have been focusing on finding the shortest path from the start position to the goal position. Other approaches have been focusing on optimizing computational time and enhancing smooth trajectory of the robot [24]. Research has also been done on navigating autonomous robots in complex environments [25].

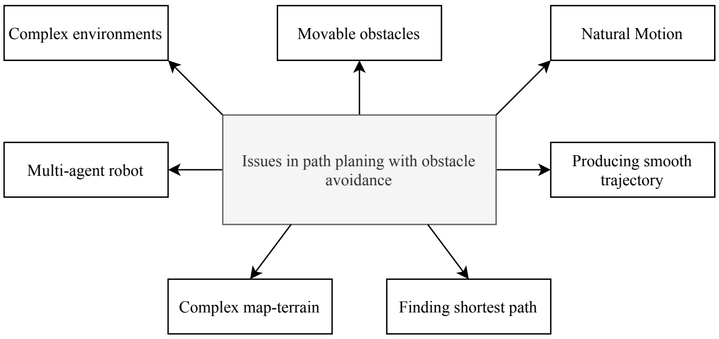

***Figure 5.1:*** *Issues related to path planning.*

Independent of the issue considered in the path planning problem, three important concerns need to be considered: safety, efficiency, and accuracy. Ideally, the robot should find its path in a short amount of time while using as little energy as possible. Besides, it should safely avoid obstacles that exist in the environment, and it should accurately follow the generated path.

## 5.3 Path Planning Categories

The main problems in mobile robot path planning can be divided into three categories relating to the knowledge the robot has about its environment, the environment nature, and the approach used to solve the problem.

**Environment nature:** Robots might need to perform path planning in both static and dynamic environments. A static environment does not change, i.e. the start and goal positions are fixed, and obstacles do not vary locations over time. A dynamic environment on the other hand, might include obstacles and goal positions that vary over time. Path planning in dynamic environments is therefore a more complex problem than in static environments due to the uncertainty of the environment. Consequently, the algorithms need the ability to adapt to any unexpected change such as a moving goal location or moving obstacles in the pre-planned path.

**Map knowledge:** Mobile robots rely on existing maps when performing path planning. They use the map as a reference to identify initial and goal location, and the link between them. The amount of knowledge about the map is an important factor when designing the path planning algorithms, in fact, path planning can be divided into two categories based on this knowledge: In the first class, the robot is provided with a map a priori. This class of path planning is known as *global* path planning. The second class assumes no a priori knowledge about the environment (i.e. no map). As a result, it has to use sensors to determine the location of obstacles, and construct an estimated map in real-time during the search process to acquire an appropriate path towards the goal while avoiding obstacles. This type of path planning is known as *local* path planning. Table 5.1 shows the differences between the two classes.

**Completeness:** The path planning algorithm can be classified as either exact or heuristic, depending on its completeness. An exact algorithm finds the optimal solution (if it exists), whereas heuristic algorithms find a “good enough” solution in a shorter amount of time.

| **Local path planning** | **Global path planning** |
|---|---|
| Reactive navigation | Deliberative navigation |
| Sensor-based | Map-based |
| Fast response | Slower response |
| Assumes incomplete workspace area | Workspace area is known |
| Generates a path and moves towards the goalwhile avoiding obstacles | Generates a feasible path before moving towards the goal |
| Done online | Done offline |

***Table 5.1:*** *Differences between local and global path planning.*

## 5.4 The A* Graph Search Algorithm

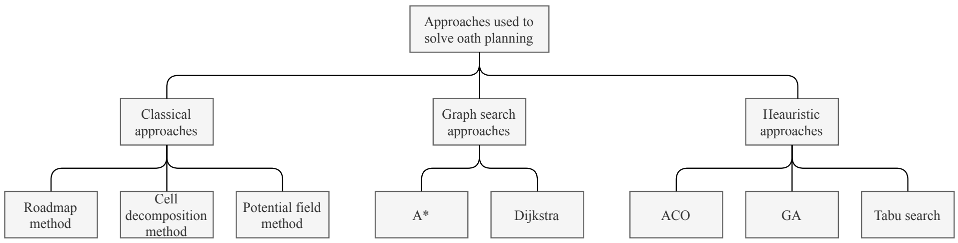

***Figure 5.2:*** *Different approaches used to solve the path planning problem.*

Since the mid 20th century when research on path planning started, there have been numerous design solutions attempting to solve the path planning problem. They can be generalized into three categories: classical approaches, heuristic approaches, and graph search approaches, as depicted in fig. 5.2. The early stages of path planning research were dominated by classical approaches such as roadmap, potential field, and cell decomposition. However, they have been shown to pose several shortcomings such as deficiencies in global optimization and robustness. The heuristic approaches aimed at solving the shortcomings of the classical approaches. There have also been developed a wide variety of graph search algorithms over the last decades that have been tested for path planning such as Dijkstra, breadth-first search (BFS), depth-first search (DFS), Bellman-Ford, A* [26], etc. This section will cover the A* (*pronounced "A star"*) algorithm which is an extension of Dijkstra's algorithm. A* is one of the most efficient algorithms for path planning, however, it can be time-consuming to reach the optimal solution depending on the number and density of the obstacles. The algorithm is presented in algorithm 1.

When finding the shortest path, A* evaluates each grid cell in the map according to an evaluation function given by:

$$f(n) = h(n) + g(n) \tag{5.1}$$

where $g(n)$ represents the accumulated cost of reaching the current cell $n$ from the start position $S$:

$$g(n) = \left\{ \begin{matrix} g(S) = 0 \\ g(parent(n)) + dist(parent(n), n) \end{matrix} \right\} \tag{5.2}$$

$h(n)$ is the estimated cost of reaching the goal position $G$ from the current cell $n$ in the least path, defined as the Euclidean distance from $n$ to $G$. This estimated cost is known as the heuristic. $f(n)$ is the estimated minimum cost of all paths to the goal cell $G$ from the start cell $S$. The tie-breaking factor *tBreak* is multiplied with $h(n)$ in order to favor a certain direction in case of ties. This ensures that the algorithm doesn't explore all equally likely paths at the same time, which in a big grid environment would be very costly. The tie-breaking coefficient is often chosen as:

$$tBreak = 1 + \frac{1}{length(Grid) + width(Grid)} \tag{5.3}$$

A* relies on two lists: an open list and a closed list. The open list contains cells that *might* fall along the best path, and should thus be checked out. The closed list on the other hand contains cells that have already been explored. Each cell is characterized by five attributes: *ID*, *parentCell*, *g_cost*, *h_cost*, and *f_cost*. When starting the search, the neighboring cells of the start position $S$ is expanded, and the cell with the lowest *f_cost* is selected from the open list, expanded, and added to the closed list. This process is repeated for each iteration. Additionally, two conditions are checked when exploring the neighbor cells of the current cell:

1. The cell is ignored if it already exists in the closed list.
2. If the cell already exists in the open list, then the *g_cost* of this path to the neighbor cell is compared with the *g_cost* of the old path to the neighbor cell. If the cost of using the current cell to get to the neighbor cell is lower, then the parent cell is changed to the current cell, and $g$, $h$, and $f$ costs of the neighbor cell are recalculated.

This whole process is repeated until the goal position is reached. By working backward from the goal cell $G$, the algorithm goes from each cell to its parent until it reaches the starting cell $S$, and the shortest path in the grid map is found.

**Algorithm 1** A_star(Grid, Start, Goal)

```
Initialization:
$ClosedSet$ = empty set;                              ▷ Set of already evaluated nodes
$OpenSet$ = Start;                                    ▷ Set of nodes to be evaluated
$came\_from$ = the empty map;                         ▷ Map of navigated nodes
$tBreak$ = 1+1/(length($Grid$)+width($Grid$));        ▷ Coefficient for breaking ties
$g\_score[Start]$ = 0;                                ▷ Cost from Start along best known path
$f\_score[Start]$ = heuristic_cost(Start, Goal);      ▷ Estimated total cost from
Start to Goal

while $OpenSet$ is not empty do
    $current$ = the node in $OpenSet$ having the lowest $f\_score$;
    if $current = Goal$ then
        return reconstruct_path|($came\_from$, $Goal$);
    end
    Remove $current$ from $OpenSet$;
    Add $current$ to $ClosedSet$;
    for each free neighbor $v$ og current do
        if $v$ in closedSet then
            continue;
        end
        $tentative\_g\_score = g\_score[current] + dist\_edge(curent, v)$;
        if $v$ not in OpenSet or $tentative\_g\_score < g\_score$ then
            $came\_from[v] = current$;
            $g\_score[v] = tentative\_g\_score$;
            $f\_score[v] = g\_score[v] + tbreak$ * heuristic_cost|($v, Goal$);
            if neighbor not in OpenSet then
                Add neighbor to $OpenSet$;
            end
        end
    end
end
return failure;
```

## 5.5 Dynamic Window Approach

*Dynamic Window Approach* (DWA) is an online collision avoidance strategy developed by S. Thrun et al. in 1997 [27]. It is a method for selecting a velocity that quickly reaches the target point by taking into account the specific dynamics and constraints on a particular mobile robot. The two main components in the DWA algorithm consists of generating a valid search space and selecting an optimal solution in the search space. The optimization goal is to select a velocity and heading that takes the robot to its goal with maximum clearance from any obstacles. Figure 5.3 illustrates how a robot's velocity search space might look like, with the translational velocity $v$ and the rotational velocity $\omega$ as axes. In the velocity search space, the robot has a maximum allowable velocity in both directions based on hardware limitations and configuration. This is called the *Dynamic Window.* In the dynamic window, the objective function $G(v, \omega)$ is used to calculate the translational and rotational velocity that maximizes the objective function by taking into account the direction, velocity, and collision of the robot.

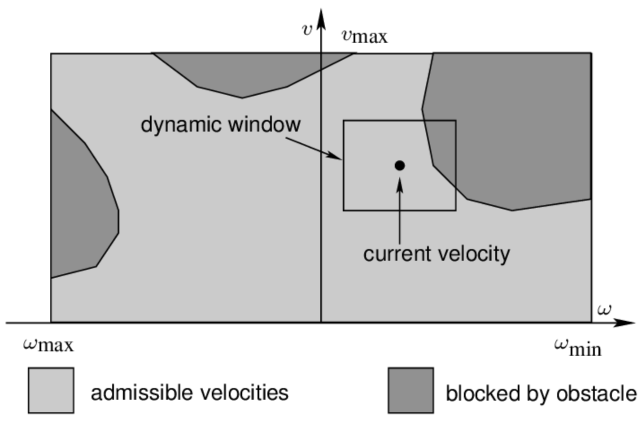

***Figure 5.3:*** *Velocity search space and the dynamic window.*

# 6 Mobile Robot Localization

## 6.1 Introduction

This chapter will introduce the concept of mobile robot localization, ultimately presenting the probabilistic localization scheme called *Adaptive Monte Carlo Localization* (AMCL), which is the system that is integrated and analyzed as part of this thesis. Mobile robot localization refers to the problem of determining the pose of a robot relative to a given map of its environment. It is sometimes called *position tracking* or *position estimation*. Mobile robot localization in particular is a subset of the general localization problem, which is the fundamental perceptual problem in robotics. This is because most robotic applications require knowledge of the location of the robot with respect to objects that are being manipulated or avoided.

The localization problem can essentially be seen as the problem of coordinate transformation. Maps are represented in a global coordinate system independent of the robot's pose. Localization is the process of establishing the relation between the map coordinate system and the robot's local coordinate system. If the robot knows this coordinate transformation it can express the location of objects of interest within its own coordinate system, which is a crucial prerequisite for robot navigation.

Unfortunately, the pose of the robot can usually *not* be sensed directly, since most robots do not have a noise-free sensor for measuring pose. Therefore, the pose has to be calculated based on data from sensors. The key difficulty then arises from the fact that the pose can not be derived from a single measurement. Instead, the robot has to integrate sensor data over time to sufficiently determine its pose. Figure 6.1 shows a general localization schematic for a mobile robot localization system.

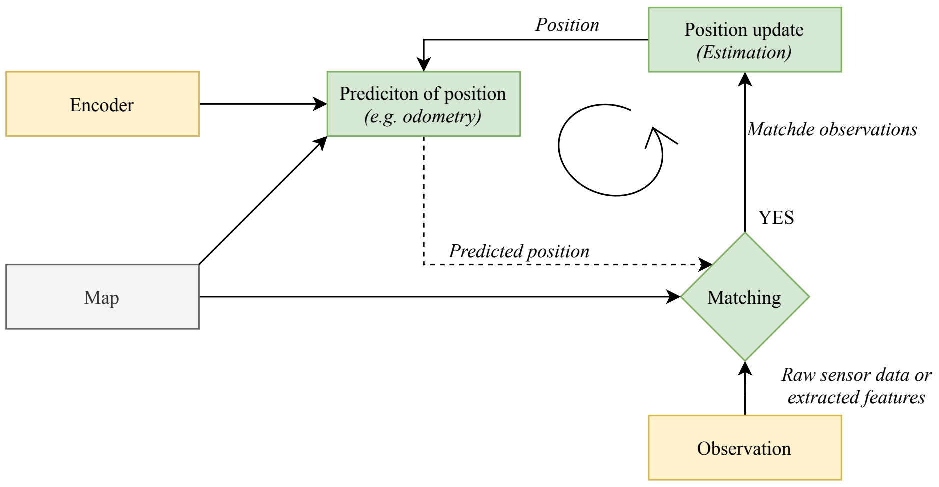

***Figure 6.1:*** *General schematic for mobile robot localization.*

## 6.2 Related Work

Mobile robot localization is a well studied field in robotics, and different approaches for solving the localization problem have been proposed in the past. Some relevant research include the use of panoramic vision [28], omnivision [29], perspective cameras [30], condensation algorithm for vision-based localization [31], Monte-Carlo localization with stereo vision [32], and localization by tracking geometric beacons [33]. Probabilistic approaches have also been successfully applied to localize mobile robots with respect to a given map. Such approaches often rely on techniques such as histogram filters [34], Extended Kalman Filters (EKF) [33], or particle filters, often referred to as Monte-Carlo localization (MCL) [35].

Regarding vehicle localization, the most commonly used sensors are cameras [36, 30, 32, 29], laser scanners [35, 37], or GPS receivers. Laser rangefinders are especially popular for robotic applications, since they provide precise data about the distance to obstacles and they require very little pre-processing of the data itself. Position errors for these lasers between 0.05 m and 0.2 m have been reported in [35] and [38] for mobile robots using SICK laser scanners with the standard Monte-Carlo localization approach.

## 6.3 Localization-based Navigation versus Programmed Solutions

Suppose a mobile robot in an indoor environment is tasked with navigating from a room **A** to a room **B**. When developing its navigation system, it is clear that the robot will need sensors and motion a control system to achieve this task. It is less evident, however, whether the robot requires a localization system or not. Localization may seem mandatory in order to successfully navigate between two rooms and accurately predict its location with respect to a map as well as detecting when it has arrived at the goal location. However, explicit localization to a given map is not the only approach for goal detection.

An alternative approach suggests that, since sensors and effectors are noisy and can be information-limited, one should avoid using a geometric map for localization. Instead, a set of behaviors can be designed that together results in the desired robot motion. This *behavior-based* approach avoids explicit reasoning about the robots location and position, and thus generally avoid explicit path planning as well. Instead, the robot can rely on procedural solutions to its navigation task. For example, the behavioralist approach to navigation between the two rooms can be to design a right-wall-following behavior and a detector for room **B** that is triggered by some condition, such as the color of the floor. An example architecture of this specific problem is shown in fig. 6.2. The key advantage of this approach is that it is easy to implement in a single environment with few goal positions. However, it suffers from several disadvantages: It does not scale to other or more complex environments, and the underlying procedures such as the right-wall-follow behavior can often not be applied in different circumstances.

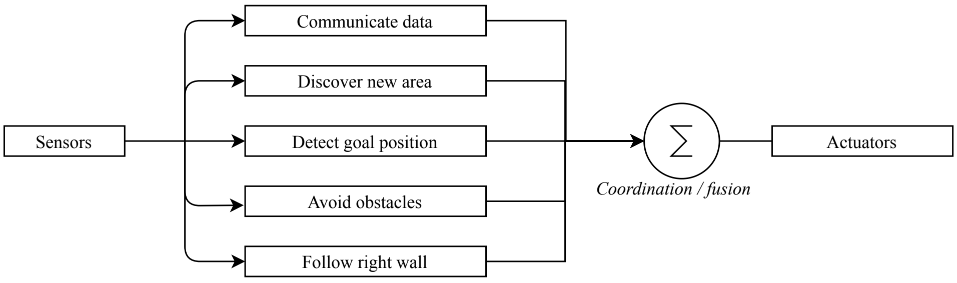

***Figure 6.2:*** *Example architecture for behavior-based navigation.*

In contrast to the behavior-based approach, the map-based approach includes both localization and path planning modules. In map-based navigation, the robot continuously tries to localize by using sensor data to update its estimated position with respect to a map of the environment. Some of the key advantages for such a map-based approach is that the robot's belief about its position is available to the human operators, and it's scalable to new environments. However, there is also a disadvantage regarding the map itself: Since the given map is "trusted" by the robot, a wrong or inaccurate map might result in undesirable behavior.

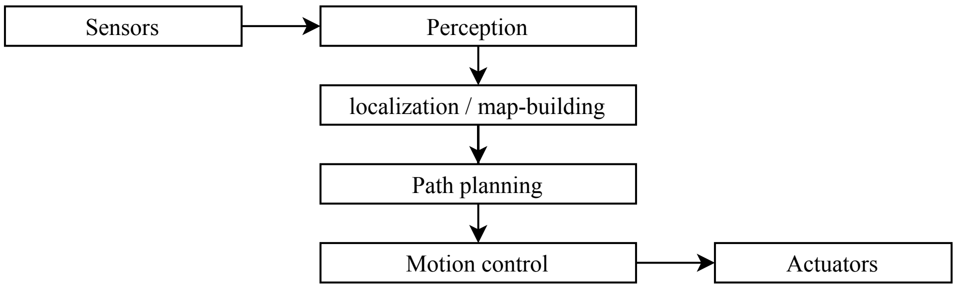

***Figure 6.3:*** *Example architecture for map-based navigation.*

## 6.4 Adaptive Monte Carlo Localization

*Adaptive Monte Carlo Localization* is a probabilistic localization system for robots moving in two dimensions, developed by Thun et al. [39]. The system implements a set of probabilistic localization algorithms in order to solve the localization problem. This section aim to describe how the algorithm works.

*Monte Carlo Localization* (MCL), also known as particle filter localization, is an algorithm commonly used for robots to localize using a particle filter. The algorithm uses a given map of the environment to estimate the position and orientation of a robot as it moves and senses its environment. A particle filter is used to represent the distribution of likely states, where each particle is representing a possible state, i.e., a hypothesis of where the robot is. The algorithm can either be initialized with an initial position estimate defined by the operator, or with a uniform random distribution of particles over the whole configuration space, meaning the robot have no information about its initial position, and assumes it is equally likely to be at any point in the environment. Whenever the robot moves, the algorithm shifts the particles to predict its new state after the movement. When the sensors on the robot senses something, the particles are resampled based on recursive Bayesian estimation, i.e., how well the predicted state correlate with the actual sensed data. Ultimately, the particles converge towards the actual position of the robot.

The state of a robot depends on the application and design. For example, the state of a two dimensional robot typically consists of a tuple $(x, y, \theta)$ for position $x, y$ and orientation $\theta$. The estimate of the robot's current state is a probability density function distributed over the state space. In the MCL algorithm, the estimate at time $t$ is represented by the set $X_t = \{x_t^1, x_t^2, \ldots, x_t^M\}$. Each particle contains an estimate of the robots state, and the regions in the state space with many particles correspond to a greater possibility that the robot will be there. Additionally, the MCL algorithm assumes the *Markov property*, that is, the current state's probability distribution only depends on the previous state and not states before that, i.e. $X_t$ only depends on $X_{t-1}$. This means that the algorithm only works if the environment is static and does not change over time.

Given a map of the environment, the goal of the algorithm is to determine the robots pose within the environment. At each time $t$, it takes as input the previous estimate $X_{t-1} = \{x^1_{t-1}, x^2_{t-1}, \dots, x^M_{t-1}\}$, an actuation command $u_t$, and sensor data $z_t$, and outputs the updated estimate $X_t$ (see algorithm 2).

**Algorithm 2** $MCL(X_{t-1}, u_t, z_t)$

```
$\bar{X}_t = X_t = \emptyset$;
for $m$ = 1 to $M$ do
    $x^{[m]}_t = motion\_update(u_t, x^{[m]}_{t-1})$;
    $w^{[m]}_t = sensor\_update(z_t, x^{[m]}_t)$;
    $\bar{X}_t = \bar{X}_t + \langle x^{[m]}_t, w^{[m]}_t \rangle$;
end
for $m$ = 1 to $M$ do
    Draw $x^{[m]}_t$ from $\bar{X}_t$ with probability $\propto w^{[m]}_t$;
    $X_t = X_t + x^{[m]}_t$;
end
return $X_t$
```

Monte Carlo localization can be improved by adaptively sampling the particles based on an error estimate using the *Kullback-Leibler Divergence* (KLD). Initially, the algorithm requires a large sample size $M$ in order to cover the entire map with a uniformly random distribution of particles. However, maintaining such a large sample size when the particles converges is computationally inefficient. AMCL (also known as KLD-sampling) is a variant of Monte Carlo Localization where the sample size, $M_x$, at each iteration is calculated such that, with probability $1 - \delta$, the error between true posterior and the sample-based approximation is less then the variable $\epsilon$. The idea in AMCL is to create a histogram overlaid on the state space. The bins are initially empty, and at each iteration, a new particle is drawn from the previous weighted particle set with a probability that is is proportional to its weight. Instead of the resampling done in MCL, the adaptive MCL algorithm draws particles from the previous weighted set and then applies sensor and motion updates before placing the particles into their bins. The algorithm tracks the number of non-empty bins, $k$, and if a particle is placed in a previously empty bin, the value of $M_x$ is recalculated. This process is repeated until the sample $M$ is the same as $M_x$.

Since AMCL removes redundant particles from the particle set by only increasing $M_x$ when a new bin has been filled, it consistently outperforms and converges faster the classic MCL.

# Part II

# Navigation Stack Development

# 7 Specifications and Requirements

## 7.1 Specifications

Specifications for the navigation stack have been selected based on the problem description on page i.

1. The navigation stack must be implemented in ROS as a stack of ROS nodes.
2. Communication with other ROS nodes must use ROS protocols.
3. The navigation stack must provide the navigation module presented in section 3.3.1 with location updates on a ROS topic with message type `geometry_msgs/PoseWithCovarianceStamped`.
4. The navigation stack must provide an implementation of an action interface that interfaces the navigation module. Specifically:
   (a) The ActionServer must take in goals containing `geometry_msgs/PoseStamped` messages.
   (b) The ActionServer must provide feedback containing the current position of the Cyborg in the environment.
   (c) The ActionServer must provide status information on the goals that are sent to navigation stack.
   (d) The action interface should enable the navigation module to cancel goals.
5. The navigation stack must provide localization functionality as a ROS node.
6. The navigation stack should utilize a map of the environment, provided by a ROS node.
7. The navigation stack must include a ROS node interfacing the peripherals on the Cyborg.
8. The navigation stack must include a navigation controller performing path planning and obstacle avoidance. Specifically:

   (a) The navigation controller must output velocity commands on a ROS topic.
   (b) The navigation controller should include recovery functionality for when the Cyborg gets stuck.
   (c) The navigation controller should utilize local and global costmaps to perform local and global path planning.

9. The navigation stack must include a ROS node interfacing the robot controller on the Pioneer LX base.

## 7.2 Requirements

1. The navigation stack must be able to control the Cyborg, which is a differential drive robot.
2. The navigation stack must be able to utilize the SICK S300 laserscanner mounted on the Cyborg.
3. The navigation stack must be designed to work on the oval shape of the Cyborg.

# 8 Design and Implementation

## 8.1 System Overview

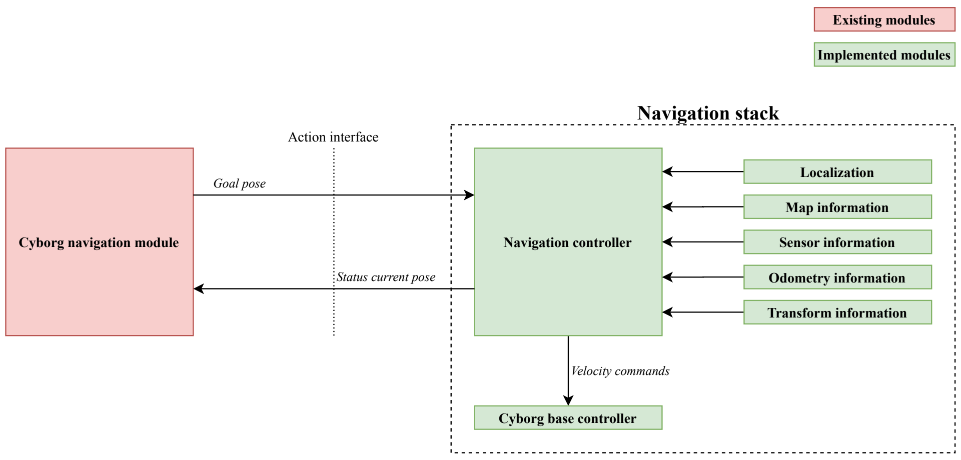

***Figure 8.1:*** *High-level overview of the navigation stack design.*

The navigation stack is designed as a set of nodes and algorithms that use sensor and odometry information, as well as transformations and goal positions in order to produce safe velocity commands to the Cyborg base controller. Figure 8.1 shows a high-level design overview of how the navigation stack is implemented in the existing Cyborg ROS network. The navigation controller module is responsible for moving the Cyborg to a desired location by linking together a global and local path planner to accomplish its global navigation task. It relies on inputs from a localization algorithm, as well map, sensor, odometry, and transform data to output velocity commands that will move the Cyborg to the desired location.

At a high level, the navigation stack works as follows:

1. A *navigation goal* is sent to the navigation stack using an action call specifying a goal pose in the map coordinate frame.
2. A path planner algorithm in the global planner calculates a shortest path from the current pose to the goal pose by using the map.
3. The path from the global planner is passed to the local planner, which tries to follow the global path when taking local changes into account. It uses information from the laserscanner to avoid obstacles, and if it gets stuck, it can ask the global planner to calculate a new path for it to follow.
4. When the Cyborg gets close to its goal pose, the action terminates, and the Cyborg has arrived.

This chapter will cover the lower-level design and implementation of the green modules in fig. 8.1, going more into detail on their functionality and inner workings. A full list of topic published by each node in the network is presented in appendix A.

## 8.2 Navigation Controller

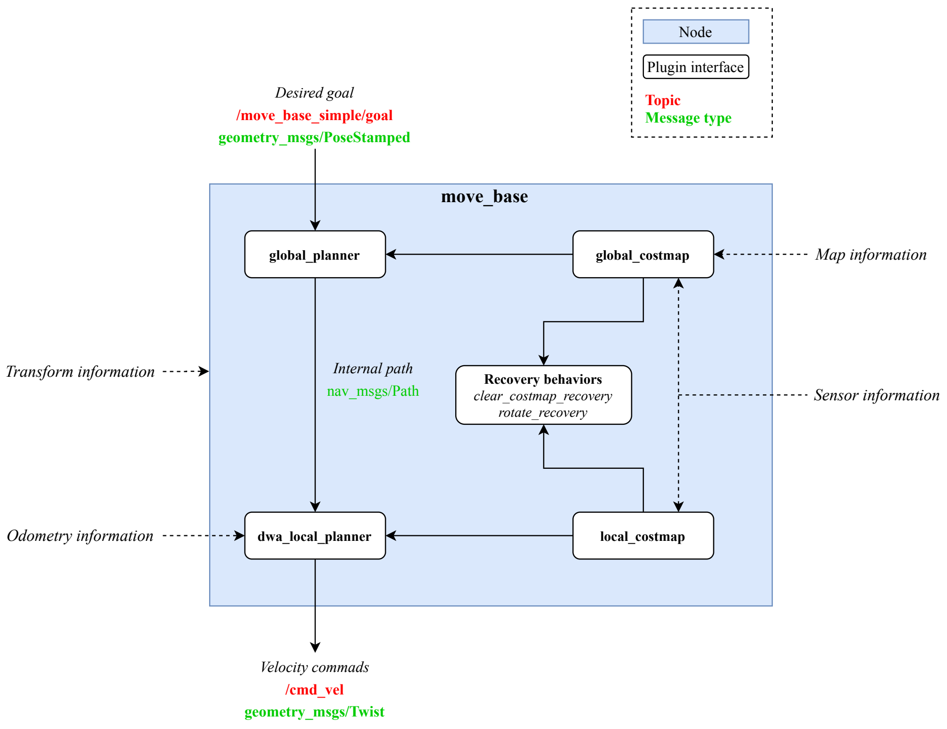

***Figure 8.2:*** *The move_base node.*

The main function of the navigation controller is to move the Cyborg from its current position to a desired goal position. The existing navigation module (not to be confused with the navigation *controller*) was designed to interface the navigation controller through an action server. The `move_base` package [40] is therefore implemented as the main controller node, as it provides an implementation of an action that, given a goal in the world, will attempt to reach it with the mobile base.

The navigation task within the `move_base` node takes place a two distinct levels. The global planner calculates a path from the Cyborg's current pose to a given goal pose, while the local planner provides movement towards a general direction while allowing for path flexibility to avoid obstacles. When the `move_base` node receives a goal pose, it computes a path to the given location and then successively produces velocity commands to the Cyborg's base controller. If at some point the Cyborg is unable to follow its plan, for example if a narrow doorway is being blocked, it will enter a recovery behavior and re-plan accordingly. If the recovery behavior also fails then the task will be aborted. Throughout the whole process, the `move_base` node provides constant feedback as to the current location of the Cyborg, as well as the status of the navigation process through the action interface. The goals can also be preempted, meaning that navigation towards some location will give up if it is given a new goal pose.

The `move_base` node also incorporates global and local costmaps that are being used with the global and local planners, respectively. The costmaps store and maintain information in the form of an occupancy grid about the obstacles in the environment, indicating where the Cyborg should navigate. The costmaps are constantly being updated based on the sensor readings to include dynamic obstacles or obstacles not pre-defined in the map.

### 8.2.1 Global Planner

The global path planner operates on the global costmap, which is initialized from the generated map of the NTNU campus. It is responsible for calculating a long-term plan that takes the Cyborg from its current position to the goal position before it starts moving. It does this by taking the Cyborg's current position, goal position, and global costmap as input, and then uses a grid-based search algorithm to compute a shortest, collision-free path.

#### Selecting a global planner

Three global planner packages were considered when deciding which method to use for creating global plans for the Cyborg. These were `carrot_planner` [41], `navfn` [42] and `global_planner` [43].

- `carrot_planner`: This planner is the simplest of the three. It checks if a given goal point is an obstacle, and if it is, it walks back along the vector between the robot and the goal until a goal point that is not in an obstacle is found. Eventually it passes the valid goal as a plan to the local planner, thus not doing any global path planning which in complicated indoor environments is not very practical.
- `navfn` and `global_planner`: `navfn` provides a fast interpolated navigation function that is used to create plans. It assumes a circular robot and operates on a costmap to find a minimum cost plan from a start point to an end point using Dijkstra's algorithm. `global_planner` is a more flexible replacement of `navfn` with additional options such as support for the A* search algorithm, quadratic approximation and grid path. figs. 8.3 to 8.6 illustrates the concept of these options.

The flexibility of `global_planner` with support for different robot shapes and search algorithms was deemed the best solution. The tuning of global planner parameters as well as other relevant navigation parameters will be discussed in chapter 9.

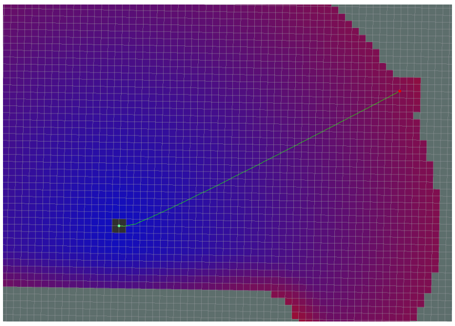

***Figure 8.3:*** *Dijkstra's path.*

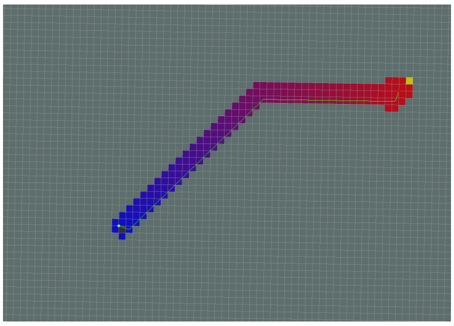

***Figure 8.4:*** *A* path.*

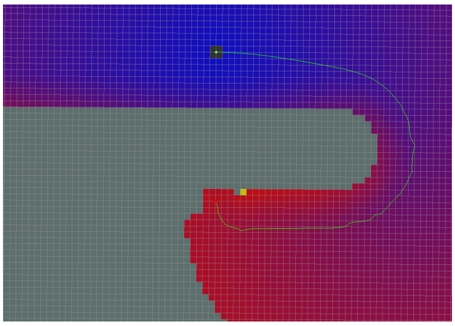

***Figure 8.5:*** *Standard behavior.*

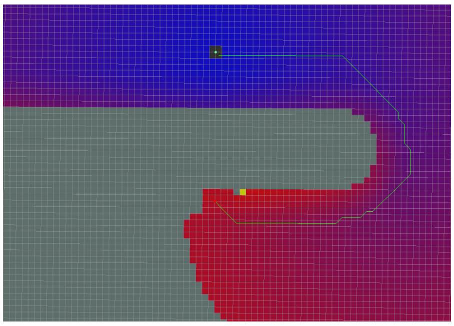

***Figure 8.6:*** *Grid path.*

### 8.2.2 Local Planner

In order to transform the global path into useful waypoints, the local planner calculates new waypoints by taking into consideration dynamic obstacles and the Cyborgs vehicle constraints. In order to calculate the local path at a specified rate, the map is reduced to the surroundings of the Cyborg which is updated as the Cyborg is moving around. It is not possible to utilize the whole map because of computational constraints and limited range of the sensors. Therefore, with the updated local map and the global waypoints, the local planner generates avoidance strategies for dynamic obstacles and attempts to match the trajectory as much as possible to the global path from the global planner. Ultimately, it outputs appropriate velocity commands to the Cyborg base controller as `geometry_msgs/Twist` messages on the `/cmd_vel` topic.

#### Selecting a local planner

Three packages was considered when deciding which method to use for generating local plans for the Cyborg. These were `dwa_local_planner` [44], `eband_local_planner` [45] and `teb_local_planner` [46].

- `dwa_local_planner`: This planner provides an implementation of the

Dynamic Window Approach (DWA) to local robot navigation. Given a global plan to follow and a local costmap, the `dwa_local_planner` computes velocity commands to the robot base.

- `eband_local_planner`: This planner computes an elastic band within the local costmap, and attempts to follow the path generated by connecting the center points of the band using various heuristics. This method is further explained in [45].
- `teb_local_planner`: Timed Elastic Band (TEB) locally optimizes the robot's trajectory with respect to trajectory execution time, separation from obstacles, and compliance with kinodynamic constraints at runtime. This method is further explained in [46].

Since the global and local path planners in the navigation stack are designed to work with costmaps, the `dwa_local_planner` was chosen. This decision is also supported by the fact that `eband_local_planner` has limited support for differential drive robots, and `teb_local_planner` requires additional data about obstacles.

### The DWA algorithm

The basic steps of the DWA algorithm is as follows:

1. Discretely sample in the Cyborg's control space $(dx, dy, d\theta)$
2. For each sampled velocity, perform forward simulation from the Cyborg's current state to predict what would happen if the sampled velocity were applied for some (short) period of time.
3. Evaluate each trajectory resulting from the forward simulation, using a metric that incorporates characteristics such as; proximity to obstacles, proximity to the goal, proximity to the global path, and speed. Discard illegal trajectories.
4. Pick the highest-scoring trajectory and send the associated velocity to the Cyborg base.
5. Rinse and repeat.

The DWA performs local path planning by using sample-based optimization. The algorithm samples a control action in the feasible velocity space, which for the Cyborg is a translational and angular velocity pair, and simulates the trajectories according to a given simulation length based on the Cyborg's motion model. An important thing to note is that the control action is kept constant along the whole prediction horizon, meaning that it cannot predict motion reversals, etc. After simulating the trajectory from each sample, the best candidate is selected based on a specific cost function and constraints, such as distance from the global path, distance from obstacle, smoothness, etc. Consequently, the DWA includes two simplifications in order to reduce the computational load while still achieving a certain amount of control performance. The cost function can be non-smooth, which makes it well-suited for grid-based evaluations such as evaluating the occupancy grid in the costmap. Furthermore, the DWA does not get stuck in local minima based on its initialization.

### 8.2.3 Local and Global Costmaps

The `costmap_2d` package [47] is used to provide an implementation of a two-dimensional costmap that takes sensor data from the Cyborg's environment, builds a 2D occupancy grid, and inflates the obstacles based on a specified inflation radius. In `costmap_2d`, the costmaps are composed of three layers: a static layer, obstacle layer, and inflation layer. The static map layer directly interprets the map provided to the navigation stack by the `map_server` node. The obstacle map layer includes 2D and 3D obstacles, and the inflation layer is where those obstacles are inflated in order to calculate the cost for each 2D costmap cell. Figure 8.7 shows how these layers are combined to generate a "master" costmap.

The navigation controller utilizes two costmaps: a global costmap and a local costmap. The global costmap is generated by inflating the obstacles on the map provided by the map server node, whereas the local costmap is generated by inflating obstacles detected by the Cyborg's sensors.

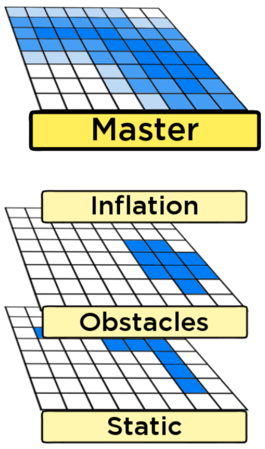

***Figure 8.7:*** *Costmap layers [48].*

### 8.2.4 Recovery Behaviors

The `move_base` node has been designed to enter a recovery behavior state if the Cyborg gets stuck and fails to find a valid plan. This might for example happen if a narrow doorway is being blocked, or if the Cyborg is surrounded by people. In this case the Cyborg will take the following actions to attempt to clear out space; First, obstacles outside of a defined region will be cleared from the Cyborg's map. After that it will perform an in-place rotation to clear out space. In the case where this also fails, the Cyborg will more aggressively clear its map by removing all obstacles outside the region of which it can rotate in-place, followed by another in-place rotation. Should all this fail, the Cyborg will consider its goal infeasible and the goal will be aborted. A flowchart illustrating the Cyborg's recovery behavior is shown in Figure 8.8.

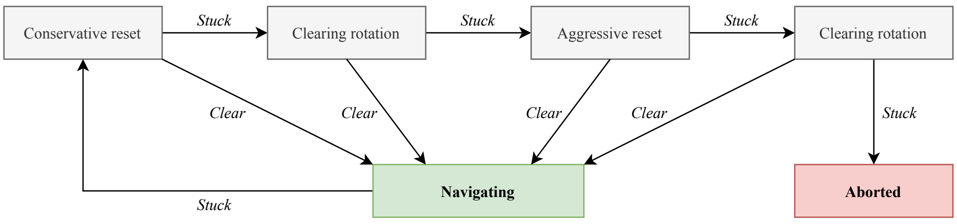

***Figure 8.8:*** *Recovery behavior state.*

## 8.3 Transform Information

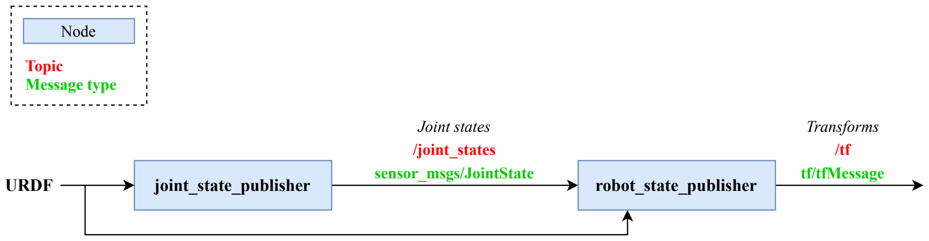

***Figure 8.9:*** *Diagram illustrating the flow from URDF to transform frames.*

The robotic system on the Cyborg has seven 3D coordinate frames that change over time. The `tf` package [49] is used to keep track of these frames as they transform over time. `tf` maintains the relationship between coordinate frames in a tree structure buffered in time and enables the transformation of points and vectors between any two coordinate frames at any desired point in time. This approach builds on the ROS philosophy of taking a distributed approach, using ROS topics to share transform data. Any node can publish the current information for some transform(s), and any node can subscribe to transform data. Data from all nodes together complete the picture of the robot. `tf` keeps track of all transformations by sending messages that contain a list of transforms specifying the names of the frames involved (parent and child), their relative pose, and the exact time that their transforms were measured.

In order to generate the internal transforms on the Cyborg, that is, the transforms between `base_link` and `l_wheel`, `r_wheel`, the two packages `joint_state_publisher` [50] and `robot_state_publisher` [51] are used in conjunction with a model description of the Cyborg in the *Unified Robot*

*Description Format* (URDF), which is an `.xml` file that represents the model of the Cyborg. The `joint_state_publisher` node reads the URDF and publishes the corresponding joint state values on the `/joint_states` topic as a `sensor_msgs/JointState` message. This message contains the state of each joint, i.e. the position and velocity of the joint, and the effort that is applied in the joint. The `robot_state_publisher` uses this data to calculate the forward kinematics of the Cyborg and publishes the resulting transform tree via `tf`. This process is illustrated in Figure 8.9. Transformations between `map` → `odom` and `odom` → `base_link` are provided by the `amcl` and `RosAria` nodes respectively. All nodes are configured to check for changes to the transform tree at a rate of 10 Hz. The resulting transform tree is depicted in Figure 8.10.

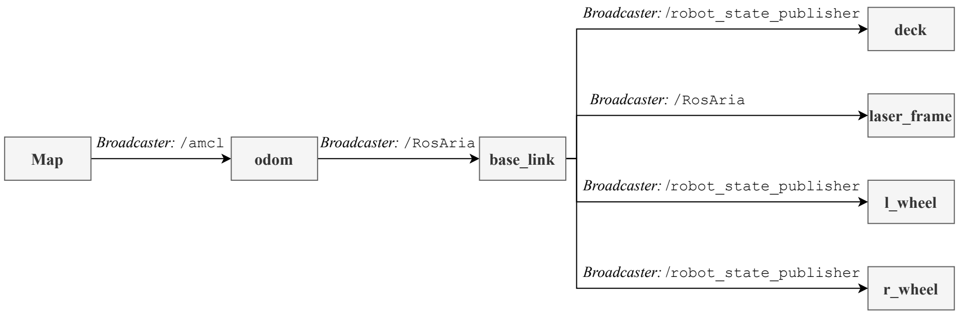
Map
Broadcaster: /amcl
odom
Broadcaster: /RosAria
base_link
Broadcaster: /robot_state_publisher
deck
Broadcaster: /RosAria
laser_frame
Broadcaster: /robot_state_publisher
l_wheel
Broadcaster: /robot_state_publisher
r_wheel

***Figure 8.10:*** *Transform tree of the Cyborg.*

## 8.4 Map Information

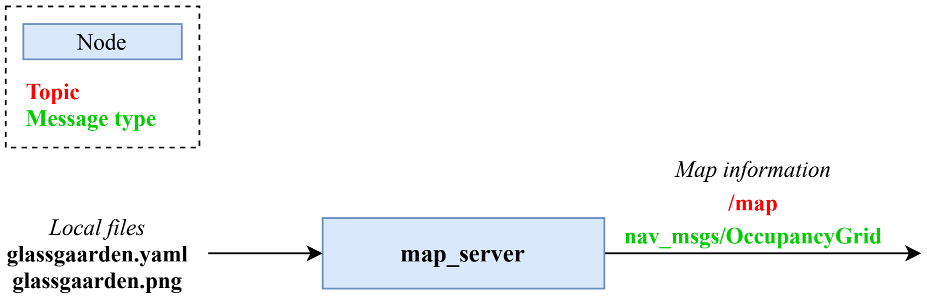

***Figure 8.11:*** *Map server*

The navigation stack does not necessarily need a-priori map information in order to function. If initialized without a static map it will only "see" obstacles that are detected by its sensors, and thus only be able to avoid those. For the unknown areas, it will generate an optimistic global path that might (most likely) hit unseen obstacles. Even though it is able to re-plan a path around these unseen obstacles when they are detected, a better solution is to initialize the navigation stack with a pre-generated static map. This is handled by the `map_server` package [52] which takes a generated map as input and serves the map to the navigation controller node on the `/map` topic. The generated map is stored as a pair of files: a `.yaml` file that describes the map meta-data and points to the image file, and a `.pgm` image file that encodes the occupancy data. The image does this by representing the state of each cell as either free (white pixels), occupied (black pixels), or unknown (grey pixels). Figure 8.12 shows the partially generated map of Glassgården that is being used by the navigation stack.

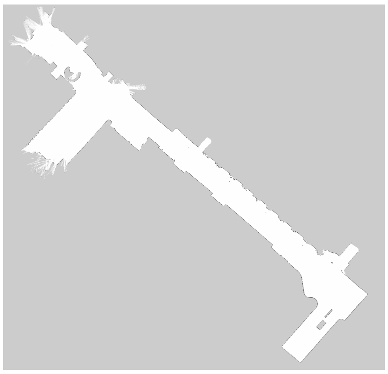

***Figure 8.12:*** *Static map of Glassgården used by the* `map_server`.

## 8.5 Sensor and Odometry Information

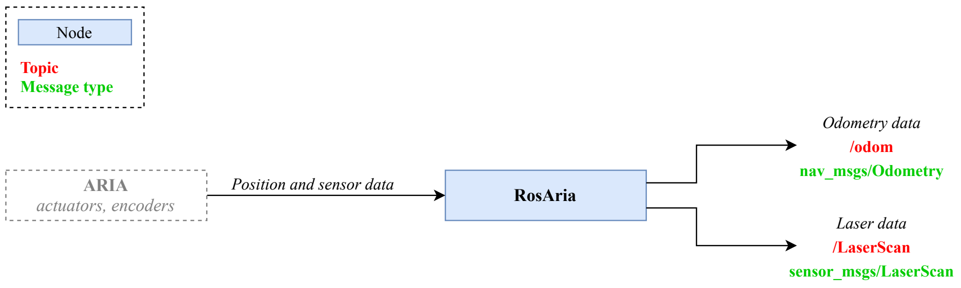

***Figure 8.13:*** *Sensor and odometry information*

Odometry data of the Cyborg provides information about its current position with respect to the starting position. This information is essential for the local planner in the navigation controller to perform path planning. The main odometry sources on the Cyborg are the wheel encoders and Internal Measurement Units (IMU). Furthermore, the navigation controller is dependent on sensor data in order to update its costmaps and perform obstacle avoidance. The main sensor source on the Cyborg is the SICK S300 laserscanner.

The ARIA library provides both odometry and sensor data from the Pioneer LX base, which is interfaced with ROS through the `RosAria` package [53]. `RosAria` publishes odometry information on the `/odom` topic as `nav_msgs/Odometry` messages containing estimates of the position and velocity of the Cyborg. Additionally, laserscan data is published on the `/LaserScan` topic as `sensor_msgs/LaserScan` messages.

## 8.6 Localization

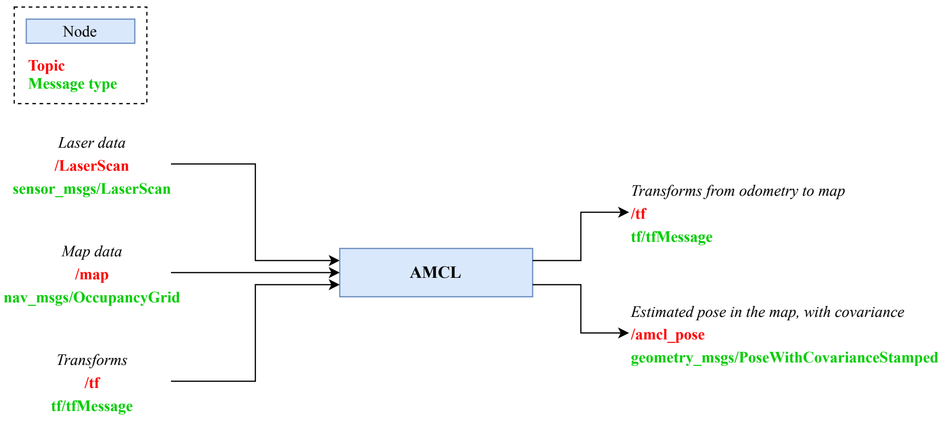

***Figure 8.14:*** *Localization, transforms from odometry to map.*

The Adaptive Monte Carlo Localization algorithm is used to perform localization on the Cyborg. The AMCL algorithm is implemented in the `amcl` package [54], which consists of a node that reads laserscan data, map data, and transform data in order to output an estimated pose in the map on the `/amcl_pose` topic as `geometry_msgs/PoseWithCovarianceStamped` messages. Additionally, the `amcl` node publishes transforms between the odometry and map frame to the transformation tree.

The `amcl` node also requires an initial pose estimation. This value can be set by configuring the node to initialize with a specific initial pose, or by publishing a `geometry_msgs/PoseWithCovarianceStamped` message to the `/initialpose` topic. Furthermore, the `global_localization` service allows for the algorithm to initialize with a randomly distributed initial pose estimation over the whole map.

## 8.7 Base Controller

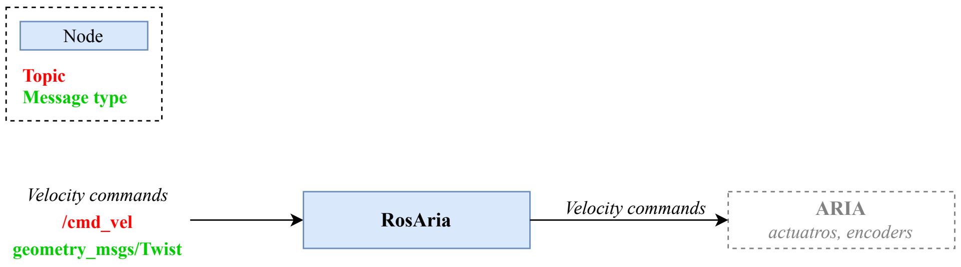

***Figure 8.15:*** *The base controller*

The local planner in the navigation controller outputs velocity commands as `geometry_msgs/Twist` messages which consist of velocities in free space broken into its linear and angular parts. As mentioned in section 8.5, the `RosAria` node interfaces ARIA with ROS and is thus responsible for relaying the velocity commands to ARIA, which in turn sends the commands to the Cyborg's embedded motion controller to perform motor control.

## 8.8 Launching the Navigation Stack

All ROS files on the Cyborg is located in the catkin workspace source directory. All ROS files for the navigation stack is organized as presented on the following page. The two ROS launch files, `cyborg_config.launch` and `move_base.launch`, located under `cyborg_2dnav` are responsible for launching all the necessary nodes in the navigation stack in a simple fashion. The contents of these files are shown in appendix C.1 and appendix C.2. Launching the navigation stack is done by typing "`roslaunch cyborg_config.launch`" and "`roslaunch move_base.launch`" in two separate terminals. The two terminal windows will then print information on robot specific elements and transform data in one terminal, and general navigation feedback in the other terminal.

```
/catkin_ws/src/navigation
│
├─ amcl
│
├─ base_local_planner
│
├─ clear_costmap_recovery
│
├─ costmap_2d
│
├─ cyborg_2dnav
│   │
│   ├─ base_local_planner_params.yaml
│   │
│   ├─ costmap_common_params.yaml
│   │
│   ├─ global_costmap_params.yaml
│   │
│   ├─ local_costmap_params.yaml
│   │
│   ├─ cyborg_config.launch
│   │
│   └─ move_base.launch
│
├─ dwa_local_planner
│
├─ global_planner
│
├─ map_server
│
├─ move_base
│
├─ nav_core
│
├─ navfn
│
├─ navigation
│
├─ robot_pose_ekf
│
├─ rotate_recovery
│
├─ joint_state_publisher
│
├─ robot_state_publisher
│
├─ rosaria
│
└─ rviz
```

## 8.9 Conclusion

The navigation stack has been implemented as a set of ROS nodes that together perform the navigation tasks of mapping, localization, path planning, and obstacle avoidance. Figure 8.16 shows the resulting design of the navigation stack when putting all the aforementioned parts together. The topics on which the nodes publish to are depicted in red. The presented design replaces the previous ARNL based system in a “plug-and-play" fashion in which the inputs and outputs of the navigation stack remain the same. This way, other modules in the Cyborg’s ROS network is not affected and does not have to be modified.

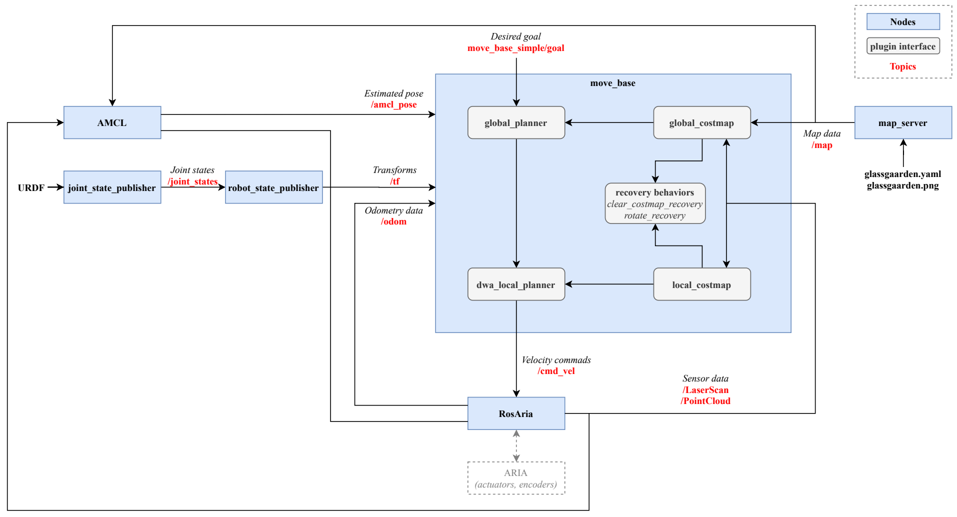

***Figure 8.16:*** *Resulting design architecture of the implemented navigation stack.*

# 9 Configuration of Path Planners

## 9.1 Introduction

The job of the navigation system is to calculate a safe path for the Cyborg to execute, by processing data from sensors, odometry and the environment map. However, achieving desired navigational behaviour and maximizing its performance requires some fine tuning of parameters. Section 9.2 and section 9.3 presents the process of tuning the local and global path planners, respectively, and in section 9.4 their corresponding costmaps are configured.

## 9.2 Local Planner

### 9.2.1 Robot Configuration Parameters

| Parameter | Value | Description |
|---|---|---|
| `acc_lim_x` $[m/s^2]$ | `1.0` | Translational acceleration limit |
| `acc_lim_theta` $[rad/s^2]$ | `0.36` | Rotational acceleration limit |
| `max_vel_x` $[m/s]$ | `1.7` | Maximum translational velocity |
| `min_vel_x` $[m/s]$ | `-0.1` | Minimum translational velocity |
| `max_vel_theta` $[rad/s]$ | `0.17` | Maximum rotational velocity |
| `min_vel_theta` $[rad/s]$ | `-0.17` | Minimum rotational velocity |

***Table 9.1:*** *Robot configuration parameters.*

#### Velocity and Acceleration

The dynamics of the Cyborg, e.g. velocity and acceleration, is essential for the local planner which takes odometry data as input and outputs velocity commands that control the Cyborg's motion. Setting maximum and minimum velocity and acceleration correctly is therefore important for the local planner to behave optimally.

**Obtaining maximum velocity:** According to the documentation of the Pioneer LX [18], the maximum translational velocity is 1.8 $m/s$, and its maximum angular velocity is 0.18 $rad/s$. These values were verified by controlling the base manually with a joystick while subscribing to the odometry topic, which relays linear and angular velocities. Running the Cyborg forward until it reached constant speed revealed that the maximum translational velocity actually was 1.7 $m/s$, slightly less then what was specified in the documentation. This difference is likely due to wear of the Cyborg's motors after several years of testing, as well as additional weight on top of the base. Manually rotating the Cyborg in place until reaching constant angular velocity revealed an actual maximum angular velocity of 0.17 $rad/s$, again slightly less than the documented value.

**Obtaining maximum acceleration:** According to the documentation of the Pioneer LX base the maximum translational acceleration is 1.0 $m/s^2$ and the maximum rotational acceleration is 0.36 $rad/s^2$. Verification of these values was done by echoing odometry data which includes timestamps on each reading. The acceleration was calculated by logging the time it took for the Cyborg to reach constant maximum velocity ($t_i$) while reading position and velocity information from the odometry data. By denoting $t_t$ and $t_r$ as the time used to reach maximum translational and angular velocity respectively, the maximum accelerations can be calculated by the following equations:

$$a_{t,max} = max\frac{dv}{dt} \approx \frac{v_{max}}{t_t} \tag{9.1}$$

$$a_{r,max} = max\frac{d\omega}{dt} \approx \frac{\omega_{max}}{t_r} \tag{9.2}$$

**Minimum values:** The minimum translational velocity allowed for the Cyborg was set to $-0.1$ $m/s$ in order to enable it to back off when it needs to unstuck itself. The minimum rotational velocity was also set to a negative value of -0.17 $rad/s$ to allow for rotations in both directions.

### 9.2.2 Forward Simulation

| Parameter | Value |
|---|---|
| `sim_time` | `1.5` |
| `vx_samples` | `10` |
| `vth_samples` | `20` |
| `sim_granularity` | `0.04` |
| `controller_frequency` | `20` |

***Table 9.2:*** *Forward simulation parameters.*

The second step of the DWA algorithm (section 8.2.2) performs forward simulation. Here, the local planner takes as input velocity samples and examines their respective circular trajectories. Each sample is simulated as if it was applied to the robot base for a given time interval. This time interval is controlled by the `sim_time` parameter. Longer simulation time makes the local planner produce longer paths which are often desirable, however, a longer simulation time also requires more computational power. Setting `sim_time` to a very low value ($\leq$ 1.0) resulted in poor performance, especially when navigating through narrow doorways. This is because of insufficient time to calculate an optimal path that actually goes through the doorway. On the other hand, since the DWA algorithm produces trajectories as simple arcs, setting `sim_time` to a very high value ($\geq$ 5.0) resulted in long curves that are not very flexible. Additionally, it was found that `sim_time` values above 2.0 caused unwanted stuttering motion as the controller was computationally saturated. A forward simulation time of 1.5 seconds was found to work the best. Figures 9.1 to 9.2 illustrates the effect of `sim_time` on the local plan where the yellow line represents the the local path.

The two parameters `vx_samples` and `vth_samples` defines the number of samples to use when exploring the x and theta velocity space. By similar reasoning as with the forward simulation time, high values (more samples) often achieve better performance. However, these values also affect the computational load. Setting the number of samples in translational directions to 10 was found to work well. Experimentation also showed that prioritizing the theta velocity space by setting `vth_samples` higher then the translational samples resulted in better performance. This is likely due to rotation being a more complicated condition than moving straight ahead. Consequently, `vth_samples = 20` was found to work well.

The `sim_granularity` parameters define the step size in meters to take between points on a given trajectory. A lower value means that more points on the trajectory will be examined, thus affecting computational load. Again, a trade-off between performance and computational load was made, and a `sim_granularity` value of 0.04 was found to result in sufficient performance.

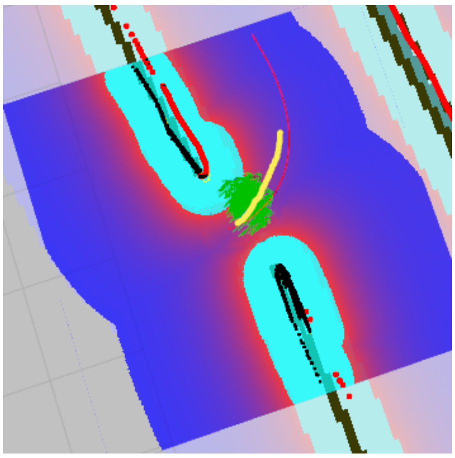

***Figure 9.1:*** `sim_time = 1.5`

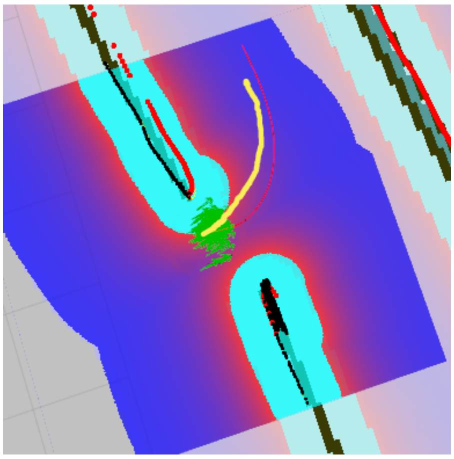

***Figure 9.2:*** `sim_time = 4.0`

### 9.2.3 Trajectory Scoring

| **Parameter** | **Value** |
|---|---|
| `path_distance_bias` | 32.0 |
| `goal_distance_bias` | 20.0 |
| `occdist_scale` | 0.02 |

***Table 9.3:*** *Trajectory scoring parameters.*

The DWA algorithm maximizes an objective function to calculate optimal velocity. The cost of this objective function is calculated as follows:

$$C = \texttt{path\_distance\_bias} \cdot D_1 + \texttt{goal\_distance\_bias} \cdot D_2 + \texttt{occdist\_scale} \cdot C_o \tag{9.3}$$

where

$$\begin{aligned} C &= \text{Cost of the DWA local planner objecvtive function} \\ D_1 &= \text{Distance [m] to path from the endpoint of the trajectory} \\ D_2 &= \text{Distance [m] to local goal from the endpoint of the trajectory} \\ C_o &= \text{Maximum obstacle cost along the trajectory in obstacle cost (0-254)} \end{aligned}$$

In eq. (9.3), `path_distance_bias` is the weight of how much the local planner should try to stay close to the global path. Experimentation showed that a high value made the local planner prefer trajectories on the global path, but as a result, the Cyborg had trouble adjusting to dynamic changes in its local environment. On the other hand, a low value resulted in too much deviation from the global path. `path_distance_bias = 32.0` was found to result in sufficient behavior.

`goal_distance_bias` is the weight of how much the local planner should try to reach its local goal. Experiments showed that increasing this parameter made the local planner less attached to the global path, similar to the effect of decreasing `path_distance_bias`. This is because the local planner prioritizes its local goal more than the global goal. `goal_distance_bias = 20.0` resulted in a satisfactory behavior in which the local planner is moderately attached to the global path, with some leeway to allow for adaptivity in its dynamic local environment.

`occdist_scale` is the weight of how much the local planner should try to avoid obstacles. A high value for this parameter makes the local planner more prone to avoid obstacles. However, experiments showed that that too high values resulted in indecisive behavior in which the local planner was unable to generate paths that adhere to the obstacle avoidance weight. `occdist_scale = 0.02` resulted in a satisfactory behavior in which the local planner was able to navigate crowded and/or narrow environments while prioritizing obstacle avoidance to a moderate degree.

## 9.3 Global Planner

| **Parameter** | **Value** |
|---|---|
| `allow_unknown` | `False` |
| `default_tolerance` | `0.2` |
| `use_dijkstra` | `False` |
| `use_quadratic` | `True` |
| `use_grid_path` | `False` |
| `lethal_cost` | `253` |
| `neutral_cost` | `66` |
| `cost_factor` | `0.55` |

***Table 9.4:*** *Global planner parameters.*

The most important parameters for the global planner and their respective values are shown in table 9.4.

With `allow_unknown` set to false, the global planner is not allowed to generate paths that traverse unknown space in the map. This makes the Cyborg only navigate the mapped area.

The `default_tolerance` parameter specifies the tolerance on the goal point of the global planner, that is, the planner will attempt to create a plan that is as close to the specified goal as possible but no further than `default_tolerance` (in meters) away. To allow for some flexibility on the goal position, this value was set to 0.2 meters.

`use_dijkstra` is set to false in order to use the better performing A* algorithm instead of Dijkstra. With `use_quadratic = true`, a quadratic approximation of the potential is used instead of a simpler calculation. Additionally, `use_grid_path` is set to false to allow for paths that does not follow the grid boundaries. Instead, a gradient descent method is used.

In table 9.4, the last three parameters `lethal_cost`, `neutral_cost`, and `cost_factor` determine the actual quality of the calculated global path. From the source code of the global planner [55], it can be seen that the cost values for the global planner is calculated from the following equation:

$$C = \texttt{neutral\_cost} + \texttt{cost\_factor} \cdot C_m \tag{9.4}$$

where

$$\begin{aligned} C &= \text{Cost values calculated by the global planner} \\ C_m &= \text{Incoming costmap cost values (0 - 252)} \end{aligned}$$

With `neutral_cost = 50`, and with incoming costmap values that range from 0 to 252, `cost_factor` need to be approximately 0.8 to ensure that the input values are spread evenly over the output range of 50 to 253. Setting the `cost_factor` too high will result in cost values that plateau around obstacles. In this case the planner may for example treat the whole width of a narrow hallway as equally undesirable and thus not calculate paths down the center. Extreme `neutral_cost` values have the same effect. Figures 9.3 to 9.8 show the effect of `nautral_cost` and `cost_factor` on global path planning where the green line is the generated global path. The chosen values of `neutral_cost = 66` and `cost_factor = 0.55` was found to yield the best results. `lethal_cost` was set to a relatively high value of 253, since lower values that were tested failed to produce any path, even when a feasible path was obvious.

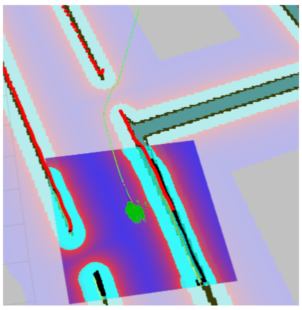

***Figure 9.3:***
`cost_factor = 0.01`

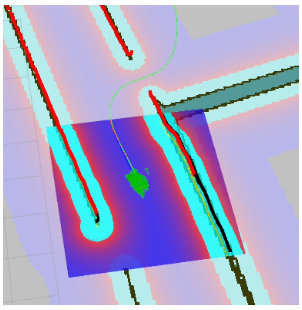

***Figure 9.4:***
`cost_factor = 0.55`

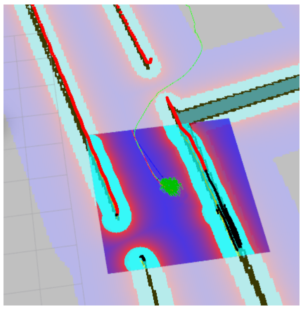

***Figure 9.5:***
`cost_factor = 3.35`

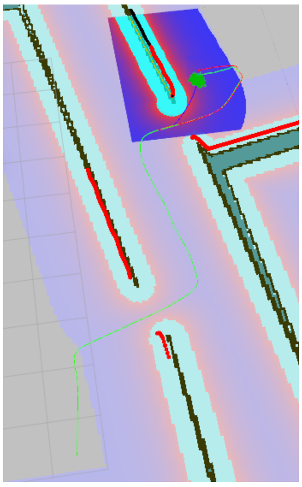

***Figure 9.6:***
`neutral_cost = 1`

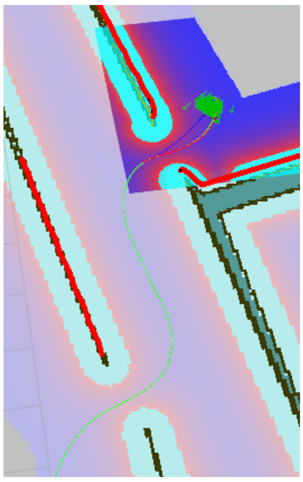

***Figure 9.7:***
`neutral_cost = 66`

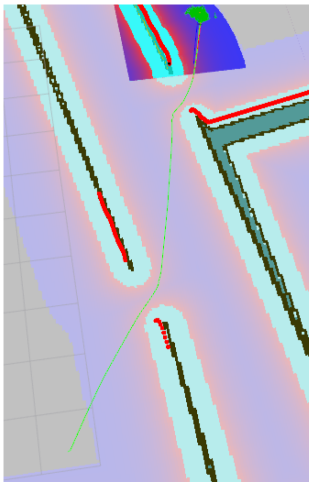

***Figure 9.8:***
`neutral_cost = 233`

## 9.4 Costmap Parameters

The performance of the path planners is highly affected by their costmaps. Correct configuration of costmap parameters is therefore essential to achieve optimal navigation. In ROS, costmaps are composed of three layers: A static map layer, obstacle map layer, and inflation map layer. The static map layer directly interprets the map provided to the navigation controller by the `map_server` node. The obstacle map layer includes 2D and 3D obstacles, and the inflation layer is where those obstacles are inflated in order to calculate the cost for each 2D costmap cell.

The navigation controller utilizes two costmaps; a global costmap and a local costmap. The global costmap is generated by inflating the obstacles on the map provided by the `map_server` node, whereas the local costmap is generated by inflating obstacles detected by the Cyborg's sensors. This section will cover the most important parameters regarding these costmaps.

### 9.4.1 Footprint

The Cyborg's footprint represents the contour of the Pioneer LX base. It is represented by a two dimensional array on the form:

$$[[x_0, y_0], [x_1, y_1], \cdots, [x_n, y_n]] \tag{9.5}$$

The array contains two-dimensional points in relation to the centroid of the Cyborg that together define its contour. Adding more points will result in a "smoother" contour, however, an approximation of the Cyborg's contour as shown in eq. (9.6) was deemed sufficient.

$$[[0.348, 0.348], [-0.696, 0.696], [-0.5, -0.5], [0.5, -0.5]] \tag{9.6}$$

The footprint is used to compute the radius of the inscribed and circumscribed circle, which are used to inflate obstacles in a way that fits the shape of the Cyborg. For safety reasons, the contour is defined to be slightly larger than the actual contour of the Cyborg.

### 9.4.2 Inflation

The inflation layer consists of cells with values ranging from 0 to 255 that determine the cost of the particular cell. Each cell is categorized as either free of obstacles, occupied, or unknown. The two parameters that define the inflation properties are `inflation_radius` and `cost_scaling_factor`. `inflation_radius` determine the radius in meters to which the map inflates obstacle cost values, that is, how far away the zero cost point is from the obstacle. `cost_scaling_factor` is a scaling factor applied to cost values during inflation that is inversely proportional to the cost of a cell. This means that higher values will make the decay curve steeper.

| **Parameter** | **Value** |
|---|---|
| `inflation_radius` | 1.75 |
| `cost_scaling_factor` | 2.58 |

***Table 9.5:*** *Inflation parameters.*

Experimenting with the inflation parameters revealed that a gentle inflation curve, as opposed to a steep curve, worked best in most situations. A gentle inflation curve causes the path planners to generate paths that are in the middle between obstacles rather than close. Having a steeper inflation curve would result in shorter paths and more effective navigation with respect to traveled distance, however, this is not an important priority for the Cyborg. When the Cyborg navigates in the middle between obstacles, it has more options to re-plan or unstuck itself if needed, and it is less likely to collide with obstacles if the localization precision is low.

sl

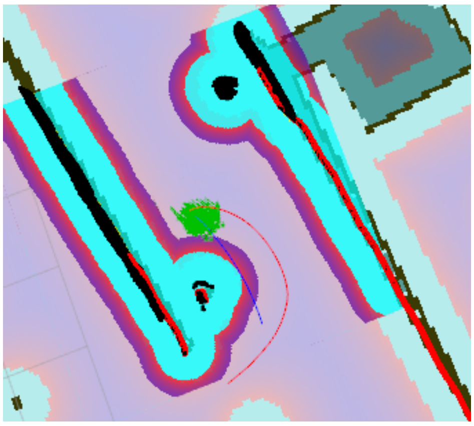

***Figure 9.9:*** *Steep inflation curve.*
`inflation_radius = 0.55`
`cost_scaling_factor = 5.0`

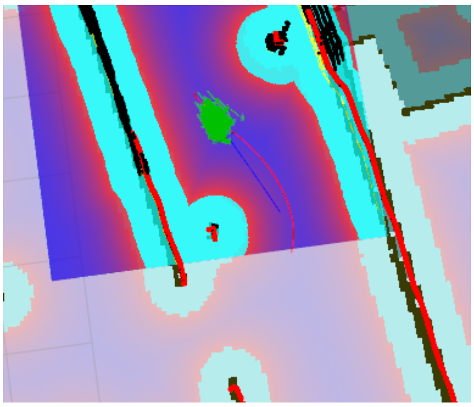

***Figure 9.10:*** *Gentle inflation curve.*
`inflation_radius = 1.75`
`cost_scaling_factor = 2.58`

# 10 Quantitative Study of the AMCL Algorithm

## 10.1 Introduction

The Adaptive Monte Carlo Localization algorithm performs localization for the Cyborg. In order to analyze and tune parameters in the localization algorithm, an experiment measuring variance in the algorithm's pose estimates was conducted. As described in section 6.4, the Monte Carlo Localization algorithm maintains two probabilistic models, a *measurement model* and a *motion model*. On the Cyborg, the measurement model corresponds to the model of its SICK S300 laserscanner, whereas the motion model corresponds to the model of its odometry. In the following sections, parameters related to the overall filter model, odometry model, and laser model will be tuned based on the experiment results.

## 10.2 The Pose Covariance Matrix

The `amcl` node outputs the Cyborg's estimated pose in the map as a `geometry_msgs/PoseWithCovarianceStamped` message which includes the variance of the $x$, $y$, and $\theta$ components. This *Covariance Matrix*, $\mathbf{M_{cov}}$, is a 6-by-6 matrix where the principle diagonal is the variance $\sigma_i^2$, with $i = \{x, y, \theta\}$. Equation (10.1) shows a simplified version of this matrix where non-interest variables are surpressed.

$$\mathbf{M_{cov}} = \begin{bmatrix} \sigma_x^2 & cov_{xy} & \dots & 0 \\ cov_{xy} & \sigma_y^2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma_\theta^2 \end{bmatrix} \tag{10.1}$$

Only five of the 36 values in the covariance matrix differ from zero. The $cov_{xy}$ values tell how much one variable influences the other in terms of direction (not intensity). because of this, we only care about the three variances on the principal diagonal in the following experiment. These values are used as metrics to measure the performance of the AMCL algorithm since they indicate how well the algorithm estimates the Cyborg's pose. The lower the variance, the more reliable the estimates are.

## 10.3 Experiment Setup

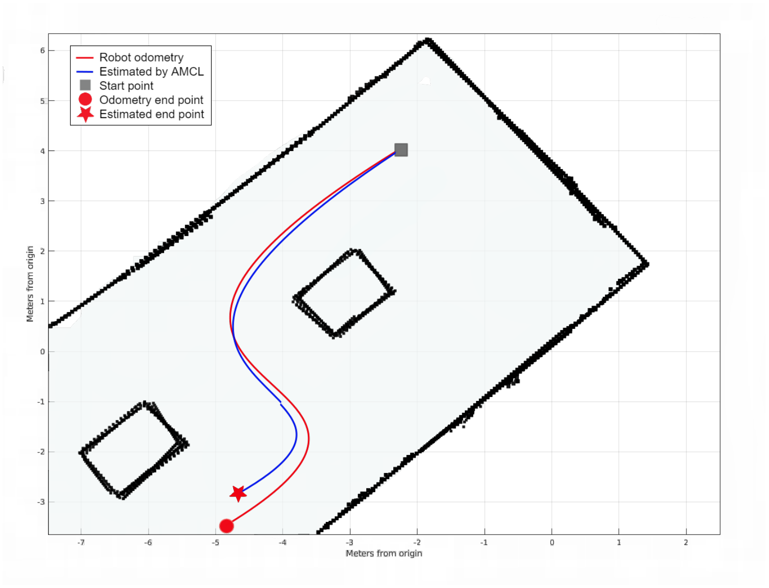

***Figure 10.1:*** *Testing environment with estimated pose from the* `amcl` *node (blue) and the Cyborg's odometry (red).*

To quantitatively and qualitatively analyze the change in variance when adjusting the `amcl` parameters, an experiment in the Cyborg's operating environment was conducted. Figure 10.1 shows the entrance area in Glassgården, used as test environment. To minimize the effect of other nodes on the `amcl` node, specifically changes in the generated path, an identical open-loop trajectory was manually published in all experiments. All experiments were initialized with the same initial pose, however, a small variance in the actual initial pose of the Cyborg is to be expected, since it was manually placed on the starting position. Several trials was conducted for each configuration, and the $x$, $y$, and $\theta$ variances was measured, combined, and averaged in order to compare their mean values from different configurations. Figure 10.1 show the Cyborg's paths over the experiment map, one estimated by the AMCL node and one measured by the Cyborg's odometry.

At each parameter change, a bagfile from all topics were recorded for posterior analysis.

The five variables found to impact the localization performance the most was the minimum and maximum number of particles used in the particle filter (`min_particles` and `max_particles`), the maximum error between the true distribution and the estimated distribution (`kld_err`), and minimum translational and angular movement required before performing a filter update (`update_min_d` and `update_min_a`). The following section presents the results of different configurations of these parameters.

## 10.4 Overall filter model

| **Parameter** | **Value** |
|---|---|
| `min_particles` | `100` |
| `max_particles` | `500` |
| `update_min_d` | `0.1` |
| `update_min_a` | `0.1` |
| `resample_interval` | `1` |
| `transform_tolerance` | `0.3` |

***Table 10.1:*** *Final filter model parameters*

### 10.4.1 Minimum and maximum number of particles

As mentioned in section 6.4, the number of particles in the filter is adaptive. The parameters `min_particles` and `max_particles` define the allowed interval that the adaptive algorithm must stay within. In this experiment, four different intervals were tested, and the results are presented in fig. 10.2. An immediate observation is that the configuration with `min_particles = 100`, `max_particles = 500` resulted in the fastest convergence at approximately 15 seconds. While in theory, one might expect that the more particles and a wider interval would yield a better result, however, this result proves otherwise. This can be partly explained by two reasons: 1) A high value of `max_particles` might not impact the algorithm too much since it is adaptive and might not use the maximum number of particles in all pose estimates. 2) Having a wide interval with more particles requires more computational power, which was repeatedly found to be a bottleneck factor on the Cyborg. The longer time the algorithm needs to calculate the estimates, the longer time it takes for it to converge.

A more complicated environment might require the use of more particles, however, in the Cyborg's operating environment, a configuration of `min_particles = 100`, `max_particles = 500` resulted in best performance.

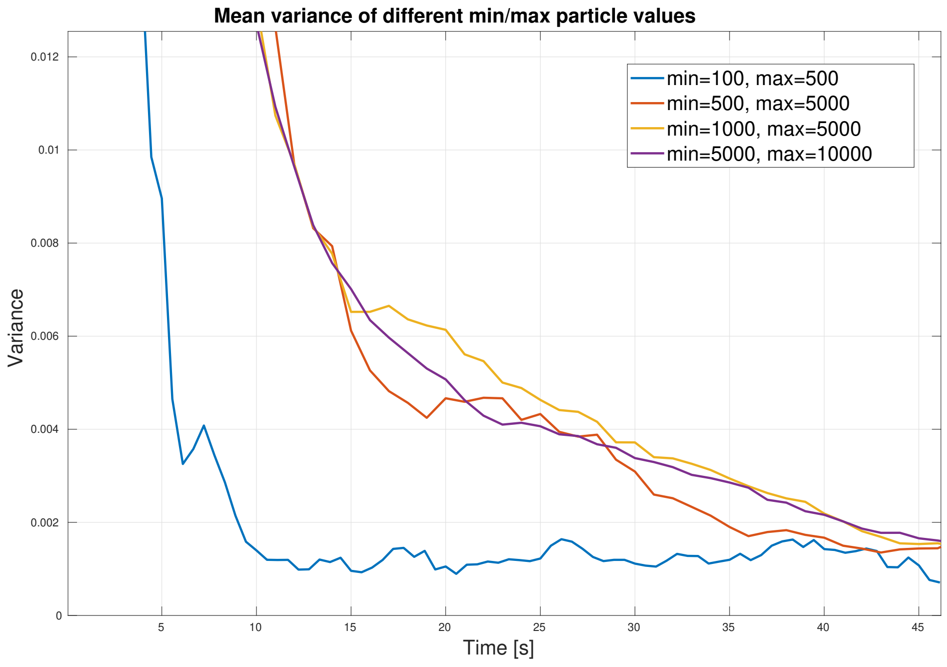

***Figure 10.2:*** *Variance convergence for different intervals of particles in the particle filter.*

### 10.4.2 Minimum translational and rotational movement

The AMCL algorithm uses odometry information to resample and update the particle filter. The two parameters `update_min_d` and `update_min_a` defines the translational and rotational movement required before performing a filter update. In theory, a higher update frequency (less movement required before performing an update) would lead to faster convergence. However, the results in fig. 10.3 show that the fastest decrease in variance occur with `update_min_a = 0.1` and `update_min_d = 0.1`. The experiments showed that more frequent updates than this was too computationally expensive for the Cyborg's computer, thus resulting in a slower convergence with `update_min_a = 0.05` and `update_min_d = 0.05`. The purple graph in fig. 10.3 shows how the algorithm fails to converge to a reasonable value when the update values are set too high. With these values, the Cyborg was unable to localize properly and the test had to be aborted after 17 seconds in order to avoid a collision.

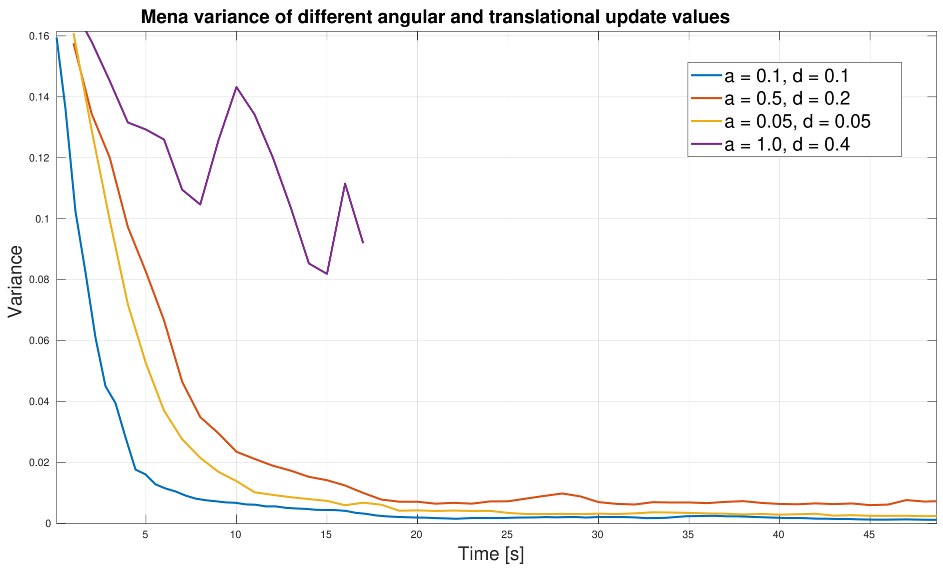

***Figure 10.3:*** *Variance convergence for different translational and angular update values.*

### 10.4.3 Resample interval and transform tolerance

The parameter `resample_interval` defines the number of filter updates required before resampling. This value is set to 1 to optimize performance, thus resampling at every filter update. `transform_tolerance` defines the time with which to post-date the localization transform, making them valid slightly longer into the future. The value of this parameter was adjusted such that it is just high enough to cover the lag in the system. With too low tolerance, the frames might never be valid when considering the lag in the system. On the other hand, too high tolerance would result in less accurate localization. For the Cyborg, a transform tolerance of 0.3 seconds was found to be a good trade-off.

## 10.5 Odometry model

| **Parameter** | **Value** |
|---|---|
| `kld_err` | `0.05` |
| `kld_z` | `0.90` |
| `odom_alpha1` | `0.2` |
| `odom_alpha2` | `0.2` |
| `odom_alpha3` | `0.8` |
| `odom_alpha4` | `0.2` |

***Table 10.2:*** *Final odometry model parameters*

### 10.5.1 Kullback-Leiber Distance Error

The `kld_err` (Kullback-Leiber distance error) parameter defines the threshold error between the true and estimated distribution. It adapts the number of samples needed for the error to respect the threshold. Figure 10.4 shows the convergence properties with four different values. The result show that the lowest threshold (`kld_err = 0.01`) yields the fastest convergence at approximately 30 second. However, all configurations converge to roughly the same variance after 40 seconds, and the difference is not as substantial as with with the particle interval in the previous section.

During experimentation, the Cyborg displayed a slight stuttering behavior with `kld_err = 0.01`. Therefore, a configuration with `kld_err = 0.05` was chosen instead, since their convergence properties are relatively similar.

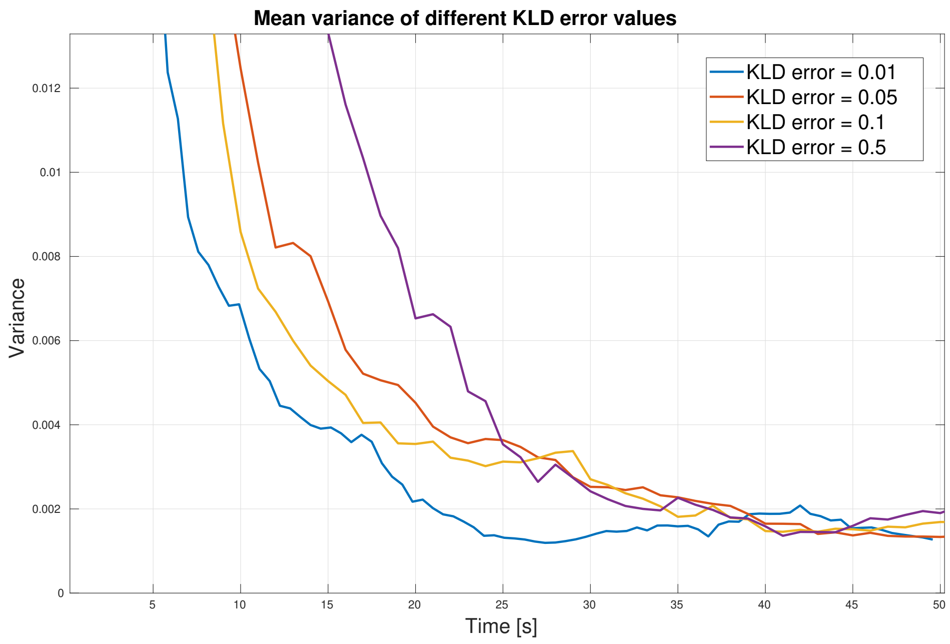

***Figure 10.4:*** *Variance convergence for different values of* `kld_err`*.*

### 10.5.2 Low vs. high noise in odometry model

Three different configurations for the odometry model parameters `kld_z` and `odom_alpha1` to `odom_alpha4` was tested. The configurations suggests an odometry model with low noise, high noise, and default noise, respectively. Table 10.3 shows the specific parameter values for each model.

| Parameter | Default | Noisy | Low-noise |
|---|---|---|---|
| `kld_z` | `0.90` | `0.5` | `0.99` |
| `odom_alpha1` | `0.2` | `0.4` | `0.05` |
| `odom_alpha2` | `0.2` | `0.4` | `0.05` |
| `odom_alpha3` | `0.2` | `0.4` | `0.05` |
| `odom_alpha4` | `0.2` | `0.4` | `0.05` |

***Table 10.3:*** *Parameter values for odometry model with default, high, and low measurement noise.*

The parameter `kld_z` is an upper standard normal quantile for $(1-p)$, where $p$ is the probability that the error on the estimated distribution will be less then `kld_err` (analogous to the variable $\epsilon$ in section 6.4). Low values thus correspond to a noisy odometry model and vice versa. The `odom_alpha1` to `odom_alpha4` parameters specifies the expected noise in odometry's rotation/translation estimates from the rotation/translational components, respectively. High values thus suggest a noisy model and vice versa.

Figure 10.5 presents the variance convergence properties when testing the three noise models on the Cyborg. $\mu(\sigma_i)$, with $i = \{x, y, \theta\}$ is the mean variance between the x, y and $\theta$ components. The result suggests moderate noise in the Cyborg's odometry, since the default parameters yields the best performance. The low noise model performs the worst with variance converging to about the double that of the default and noisy model. However, the noise in odometry's translational estimate from the translational component (`odom_alpha3`) was found to notably higher then the rest, and was thus adjusted to it's final value of 0.8.

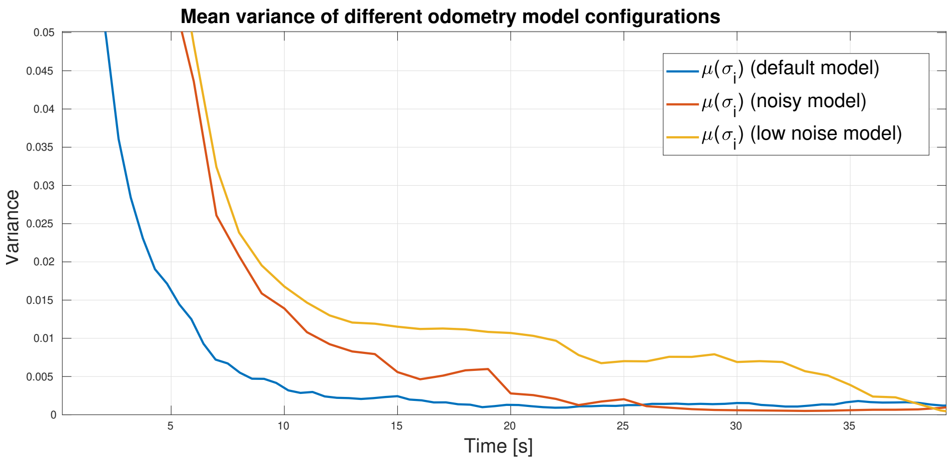

***Figure 10.5:*** *Variance convergence for different noise levels in the odometry model configuration.*

## 10.6 Laser model

| Parameter | Value |
|---|---|
| laser_max_beams | 60 |
| laser_z_hit | 0.5 |
| laser_z_rand | 0.5 |
| laser_sigma_hit | 0.2 |
| laser_likelihood_max_dist | 2.0 |

***Table 10.4:*** *Final laser model parameters.*

The most significant parameters relating the model of the SICK S300 laserscanner is presented in table 10.4. `laser_max_beams` defines how many evenly-spaced beams to use in each scan when updating the filter. More beams used generally lead to faster convergence and more accurate localization, however, this parameter greatly impacts computational load. `laser_max_beams = 60` was found to be a good trade-off between performance and computational efficiency. For the remaining four parameters `laser_z_hit`, `laser_z_rand`, `laser_sigma_hit`, and `laser_likelihood_max_dist`, three configurations suggesting a noisy laser model, low-noise model, and default model was tested.

| Parameter | Default | Noisy | Low-noise |
|---|---|---|---|
| laser_z_hit | 0.5 | 0.9 | 0.3 |
| laser_z_rand | 0.5 | 0.7 | 0.3 |
| laser_sigma_hit | 0.2 | 0.4 | 0.1 |
| laser_likelihood_max_dist | 2.0 | 4.0 | 1.0 |

***Table 10.5:*** *Parameter values for laser model with default, high, and low measurement noise.*

The `laser_z_hit` parameter is a weight factor relating to the case when a laser beam hits an obstacle. `laser_z_rand` relates to the uniform distribution which is used to model the situation in which there might exist some unexplained measurements. In the Cyborg's environment, we can expect some unmodelled obstacles (obstacles not in the map, like people), increasing `laser_z_rand` will thus model the Cyborg's environment more accurately. However, the absence of unmodelled obstacles during the experiment might lead to a misleading result in fig. 10.6, where the default configuration resulted in the best performance. Since extensive testing in a crowded environment was not conducted, `laser_z_rand` is kept to its best performing value, however, future developers should consider increasing it when tuning the system to a crowded environment.

`laser_sigma_hit` refers to the standard deviation for the Gaussian model used in the `z_hit` part of the model, and `laser_likelihood_max_dist` is the maximum distance to do obstacle inflation on the map. Both values are increased in order to incorporate higher measurement noise.

The experimentation result for the three noise models is presented fig. 10.6. The default configuration yields the best convergence properties, with convergence after approximately 15 seconds. The low-noise model performs the worst, whereas the high-noise model performs very similarly to the default model. Although the default values are chosen, future developers should highly consider the high-noise model in order to incorporate “noise” from dynamic obstacles.

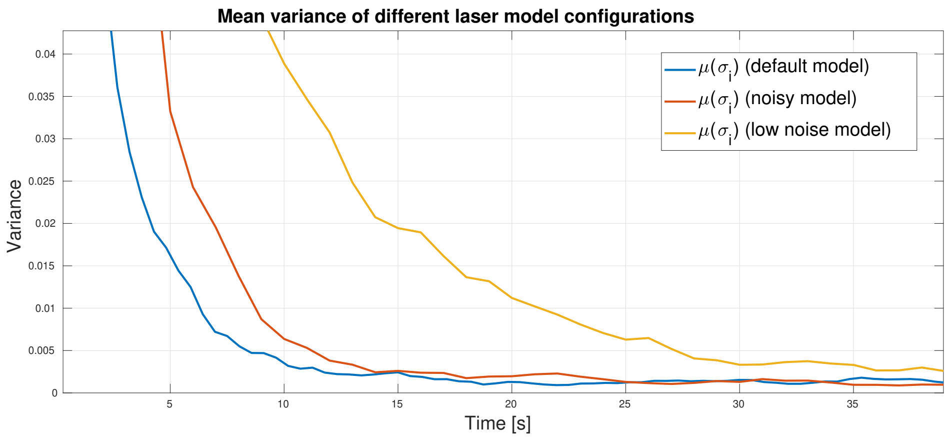

***Figure 10.6:*** *Variance convergence for different noise levels in the laser model configuration.*

### 10.6.1 LaserScan header

Readings from the Cyborg’s laserscanner are published to the `/scan` topic with type `sensor_msgs/LaserScan`, which contains a header with parameters that are dependent on the specific laserscanner used. These parameters are:

- `angle_min`: start angle of the scan [rad]
- `angle_max`: end angle of the scan [rad]
- `angle_increment`: start angle of the scan [rad]
- `time_increment`: time between measurements [s]

- `scan_time`: time between scans in seconds [s]
- `range_min`: minimum range [m]
- `range_max`: maximum range [m]

During experimentation, it was observed that incorrect values caused the laser readings to not coincide with the map of which it operates (see figs. 10.7 to 10.8, the red dots are readings received from the laserscanner). The correct values was found from the SICK S300 documentation [18] and was set to the values in table 10.6.

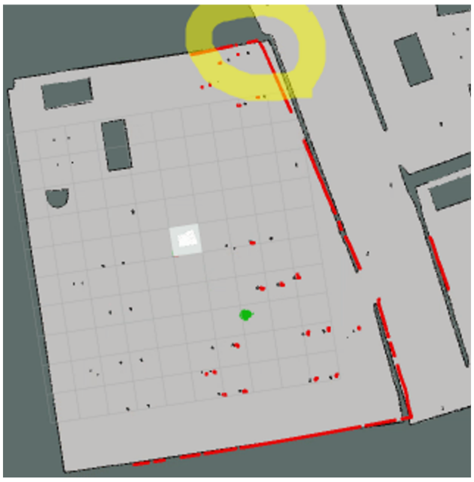

***Figure 10.7:*** *Incorrect values in LaserScan message header.*

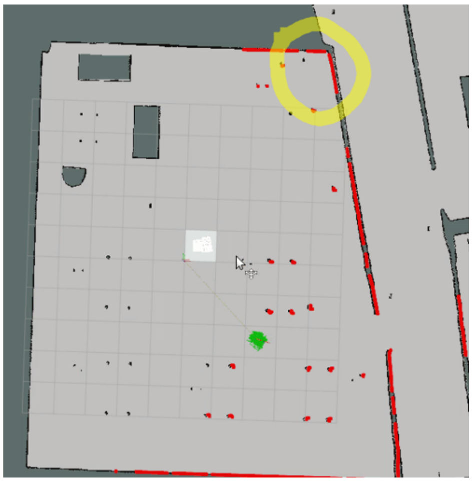

***Figure 10.8:*** *Correct values in LaserScan message header.*

| **Parameter** | **Value** |
|---|---|
| `angle_min` [rad] | `-2.3562` |
| `angle_max` [rad] | `2.3562` |
| `angle_increment` [rad] | `0.0087` |
| `time_increment` [s] | `0.00009` |
| `scan_time` [s] | `0.013` |
| `range_min` [m] | `0.0` |
| `range_max` [m] | `20.0` |

***Table 10.6:*** *Header parameters in LaserScan message.*

## 10.7 Conclusion

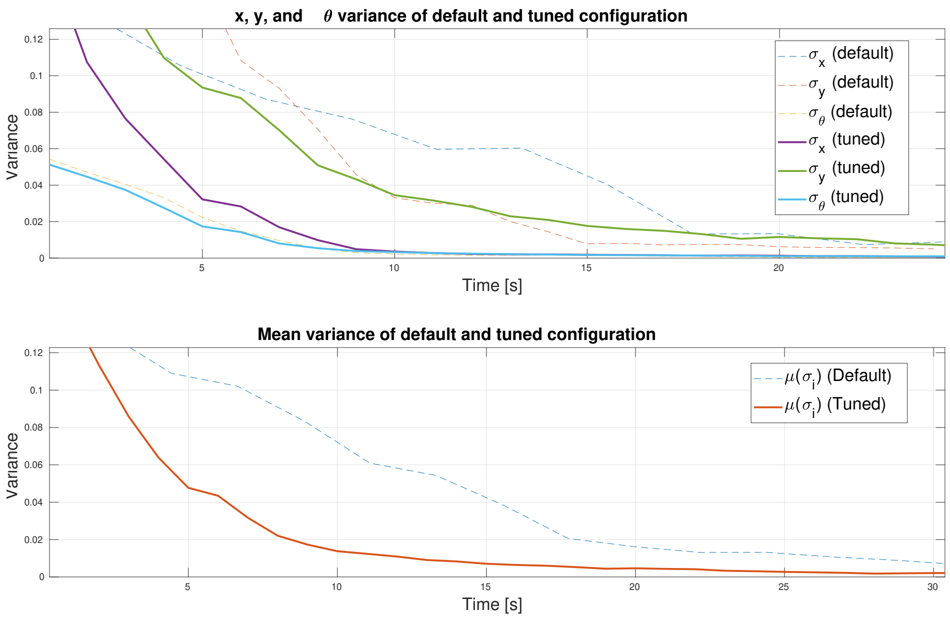

***Figure 10.9:*** *Overall difference between default and tuned AMCL parameters.*

This section examined the distinct influence of each tested parameter on the AMCL algorithm. The Cyborg ran the same paths in order to enable a comparison between parameter changes, and the result of localization was evaluated by analyzing the covariance matrix provided by the `amcl` node.

Figure 10.9 shows the difference in variance convergence between default AMCL parameters and the tuned configuration as a result of the preceding experiment. The top figure shows the specific variances of the x, y, and $\theta$ components between the default (dotted) and tuned (solid) configurations, while the bottom figure shows the mean of the three components.

The result in fig. 10.9 shows that the tuned configuration achieves considerably better convergence properties, with a mean variance of 0.02 after $\approx$ 8 seconds, compared to the same variance after $\approx$ 18 seconds for the default configuration (approximately 55.6 % decrease). From the top figure, it is clear that the biggest difference comes from the x component. A reason for this could be the specific layout/design of the testing environment, where the x component of the localization estimate was impacted the most. For example, if the Cyborg was to localize when moving through a very long and evenly spaced corridor in the horizontal x-direction, it would likely have a greater variance in the x-component, since the laser will struggle to sense unique features in the environment in the x-direction (since the laser has limited range it would not be able to detect anything straight forward in the long corridor).

All the final parameter values for the filter, odometry, and laser model are presented in table 10.7 on the following page.

| **Parameter** | **Value** |
|---|---|
| | *Filter model* |
| `min_particles` | 100 |
| `max_particles` | 500 |
| `update_min_d` | 0.1 |
| `update_min_a` | 0.1 |
| `resample_interval` | 1 |
| `transform_tolerance` | 0.3 |
| | *Odometry model* |
| `kld_err` | 0.05 |
| `kld_z` | 0.90 |
| `odom_alpha1` | 0.2 |
| `odom_alpha2` | 0.2 |
| `odom_alpha3` | 0.8 |
| `odom_alpha4` | 0.2 |
| | *Laser model* |
| `laser_max_beams` | 60 |
| `laser_z_hit` | 0.5 |
| `laser_z_rand` | 0.5 |
| `laser_sigma_hit` | 0.2 |
| `laser_likelihood_max_dist` | 2.0 |

***Table 10.7:*** *Tuned AMCL parameters.*

# 11 | Discussion

## 11.1 Introduction

The overall navigation system, as it stands at the end of this project, is a well-functioning foundation for further development. All planned modules presented in the specifications have been met. However, some modules are only capable of demonstrating basic functionality, leaving room for further improvement in terms of robustness, tuning, functionality, and ease of use. This chapter will discuss some of the work and implementations presented in chapters 8 to 10. First, an overall assessment of general topics in the project will be presented, followed by a discussion of navigation results and the quality of the AMCL study presented in chapter 10.

## 11.2 Overall Assessment

### 11.2.1 ROS as Development Framework

Choosing ROS as a development framework has likely been a contributing factor to achieving a functional solution. Despite ROS' steep initial learning curve and novel structure, it quickly proved to be a flexible and rich tool. Experienced ROS developers will likely be able to rapidly implement and test concepts regarding navigation and robotics. The worldwide community of developers using ROS as a primary framework to develop mobile robots has resulted in a rich set of tools and functionality packages that anyone can implement and further develop. This is even encouraged by many package creators. Furthermore, the node structure in ROS is a convenient way structuring the system into modular, self-contained, and reusable modules. This is what enables ROS developers to share modules, and will hopefully benefit future development on the NTNU Cyborg as well.

### 11.2.2 Limitations in Computational Power

The embedded computer on the Pioneer LX running the Intel D525 central processing unit (CPU) proved to be a bottleneck factor when configuring modules in the navigation stack. This was especially noticeable when tuning the local path planner (section 9.2), specifically the three parameters `sim_time`, `vx_samples`, and `vth_samples`, which have a great impact on obstacle avoidance and navigational behavior in general. The quality of the generated local paths when tuning these parameters to favor performance was consistently better than with a configuration favoring computation time. However, `vx_samples` is essentially limited to values below 10, as higher values cause the path planning algorithm to exceed the controller frequency, resulting in stuttering behavior in which the Cyborg repeatedly stops and waits for the planner to calculate a path, moves for a bit, and then stops again. Even though this issue was most prevalent in the local path planner, other modules such as costmap, global planner, and localization were also affected.

In order to optimize the navigation system with respect to performance and computation time, future projects should experiment both with *soft* and *hard* real-time requirements. This entails experimenting with different frequency combinations in the navigation and base controller, as well as modifying the time sequence of events. For soft real-time requirements, the time sequence of events is more important than the actual time to execute the actions. Hard real-time requirements on the other hand require the execution times to be fast enough for the time constraints to be absolutely met.

## 11.3 Navigation

Integrating the foundations of the ROS navigation stack was in itself a fairly straight forward procedure. However, customizing and tuning it to work optimally with the Cyborg was a more complicated and time-consuming process. During development, the navigation stack was tested both on a simulated version of the Cyborg and the live robot. It quickly became apparent that the navigational behavior exhibited in simulations did not tightly coincide with the real robot. This made it difficult to rapidly test and prototype navigational functionality since tests relying on simulations would be of less significance. Instead, the tests often had to be conducted on the live robot in limited public areas.

### 11.3.1 Tuning

There exist no official tuning strategies for the navigation stack in ROS besides a basic guide [56] consisting of several “change and check” procedures that are mostly based on experience and educational guesswork. This strategy is both time-consuming and most likely will not lead to an optimal solution. Because of this, a more quantitative approach focusing specifically on localization was conducted, hopefully contributing towards an optimal tuned configuration in the future.

The more qualitative experiments conducted when tuning path planners and costmaps actually prove to work pretty well. The RViz software was very helpful for visualizing how the Cyborg “sees” its environment as costmaps, and how they affected the path planners. This visually based tuning approach enabled rapid testing and resulted in satisfactory behavior from the path planners. It is, however, recommended that future projects conduct a more thorough experiment that optimizes performance based on metrics such as distance to obstacles, accuracy at target, time to reach the goal, path length, lateral and tangential stress, and so on.

### 11.3.2 Performance

Live testing showed reliable and promising results, although not entirely without some quirks. The Cyborg sometimes exhibited inconsistent navigational behavior when entering a narrow hallway or a door. This is likely due to the local costmap being generated slightly different each time the Cyborg enters the particular area, which in turn affects how the path plans are calculated. This was especially a problem with a low resolution on the local costmap. Additionally, there is no memory on the Cyborg, i.e. it does not remember how it entered a room through a doorway, so it needs to calculate both the costmap and the path each time it tries to enter the same area.

The localization performance was observed to exhibit inconsistent behavior in some areas of Glassgården where the distance to the nearest obstacle is at its highest. Particularly, the entrance area by the cafeteria was found to be a reoccurring source for this issue. Since the SICK S300 laser scanner has a max measuring distance of 15 meters, it will in some circumstances struggle to accurately sense its environment, which in turn leads to bad localization in which variance in the pose estimates increases. This was, however, not a significant issue, since the localization algorithm only showed minor spikes in variance, and quickly converged to a low value once distinct features were detected again.

The most prevalent side effect of the limitations in computational power was poor obstacle avoidance. Initial testing of the navigation system resulted in several (safe) collisions with dynamic obstacles such as test subjects walking in front of the Cyborg. The primary source for this issue was found to be too long calculation times for the local planner. Either the Cyborg was moving too fast on a collision path to be able to calculate a new, collision-free path, or the algorithm simply took too long. This forced limitations on path planner and localization configurations in which computer efficiency had to be prioritized over performance. This was, perhaps, most prevalent when tuning minimum and maximum number of particles in the AMCL particle filter, where a computationally *heavy* configuration significantly reduced performance. Figure 11.1 shows how the computationally *friendly* configuration of `min = 100, max = 500` converges $\approx$ 35 seconds faster than the rest.

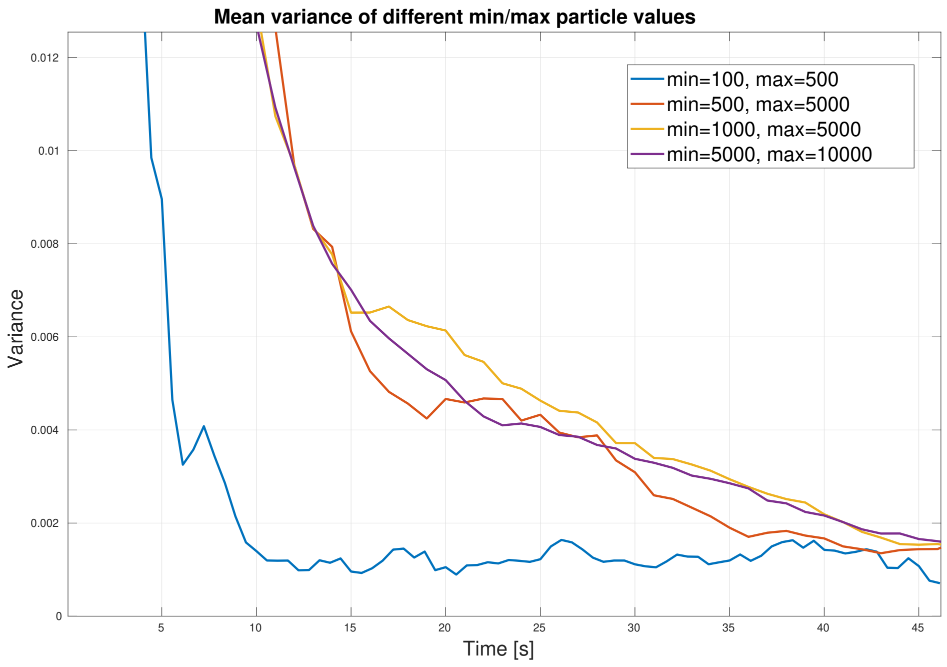

***Figure 11.1:*** *Difference in settling time between computationally friendly/heavy configurations.*

### 11.3.3 Social Navigation

A primary goal for the NTNU Cyborg project is to have a social robot roaming the campus hallways. Even though the current state of the navigation system is able to take the Cyborg from point A to point B in an efficient and collision-free fashion, one might argue that the navigational behavior is lacking social intelligence.

As an example, consider a scenario in which the Cyborg is navigating a hallway with a person as depicted in fig. 11.2. As the Cyborg and person approach each other, it is unclear what the Cyborg should do to efficiently navigate past the person. For two people passing each other, this is a trivial task in which they have a shared body of implicit knowledge about social situations, and they share several social cues in order to manage the interaction. These things typically lacks, however, in a human-robot interaction, and the person might feel anxious and threatened as a result. In fact, a study by Mutlu and Forlizzi [57] looking at autonomous delivery robots in a hospital environemnt found that patients felt “disrespected” by the behavior of the robots because of its navigational behavior.

In the current navigation algorithm, the Cyborg is programmed to take a most efficient path, often leading to a path that drives down the center of the hallway until a collision with the person is imminent. The Cyborg treats every person as it would with any other obstacle, not taking into account the fact the person is a moving, decision-making entity that will react the the movements of the Cyborg. From the perspective of the person, they will have no way to predict which side the hallway the Cyborg will pass, resulting in high uncertainty and inefficient task behavior.

Some ways to address these problems could be to incorporate a visual or auditory signal from the Cyborg, indicating that it is aware of the humans presence. This could for example be realized by incorporating computer vision and object detection systems. Another solution could be to modify the Cyborg's costmaps to reflect the social behavior wanted in the planned paths. This solution, however, requires precise and careful tuning, since imposing too hard constraints on the costmap space might lead to infeasible paths past a person in a narrow hallway. Using computer vision technology to detect which side of the hallway the person is closest to, and linearly decrease the cost in the opposite direction could, on the other hand, could be a feasible solution.

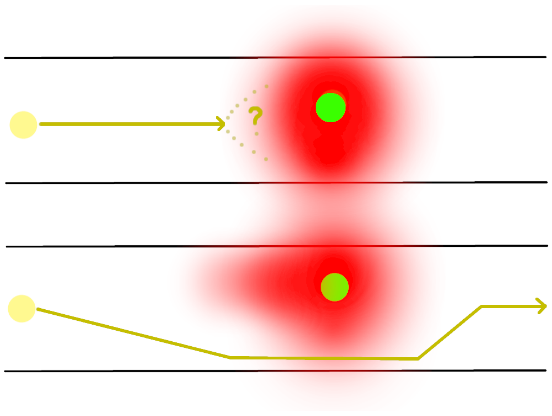

***Figure 11.2:*** *Illustration comparing current costmap configuration (top) and potential modification (bottom) enabling "social navigation". Yellow circle indicates the Cyborg, green circle indicates a person, intensity of the red color correspond to the intensity of the cost in the costmap.*

## 11.4 Quality of AMCL study

The results of the AMCL study presented in section 10.7 shows a significant improvement the AMCL algorithm in terms of variance in the estimated pose. The improvements was also visible when observing and comparing the Cyborg's behavior both with default and tuned configuration. After tuning the algorithm and achieving faster variance convergence, the Cyborg exhibited more "confident" behavior in which it seemed less hesitant and stuttery when pursuing a goal. With default configuration, the Cyborg often rotated back and forth several times when trying to pursue its local path, since the the uncertainty (high variance) in the pose estimates caused the local planner to generate inconsistent paths.

The result is clearly positive with respect to the metric studied, however, a more thorough analysis comparing different algorithms, runtime, execution time, etc., would be beneficial in terms of validity of the study. Instead of looking at the variance, the algorithm could be evaluated by manually measuring the exact pose of the Cyborg and compare it with the estimated from the AMCL algorithm. Additionally, potential cause-effect relationships between parameter changes was not researched, and the experiment was only conducted the one specific environment presented in fig. 10.1. The testing environment did not include dynamic obstacles, and is only representative for a portion of the Cyborg's operating environemnt. Empirical evidence suggest that the localization performance does not vary between different locations on the Campus, however, the absence of moving obstacles is a crucial difference between the experiment setup and the Cyborg's live environment.

Several trial runs were conducted for each parameter change, however, the number of runs was not kept consistent for all parameter changes. A minimum of 5 runs were conducted for each change, but some trials had to be run more then that. In the cases where the five first runs produced almost the same result, no more runs was carried out. However, in the cases where the results did differ, a couple of extra runs was conducted to achieve the most accurate representation of the data. This introduces a bias to the experiment which could affect the results. Ideally, more runs should be conducted for each trial, and no human bias should be introduced.

## 11.5 Proposed Future work

- **Social Navigation:** Implement social navigational behavior. This entails working with computer vision in conjunction with path planners and costmaps to develop a "socially inteligent" behavior.
- **Docking:** Docking functionality has to be implemented in order to autonomously operate on the NTNU campus. The previous ARNL system had embedded functionality for docking, however, the new navigation stack doesn't. This entails finding out how to represent the docking station in the

map, how to move to it from any point in the environment, high-level control for when the Cyborg should dock (the `RosAria` node provide battery status on the `battery_state_of_charge` topic), and low level control dealing with the docking task itself. The open source code for ARNL [58] can be useful, since it is a wrapper and uses ARIA classes (`ArDocking.h`).

- **Quantitative study of path planners:** Several algorithms exist for mobile robot path planners. The current grid based A* algorithm should be tested against other grid-based and heuristic approaches. An interesting heuristic approach are the use of neural networks, or hybrid versions of a grid-based algorithm and neural network [59].
- **More robust recovery behavior:** Augment the current recovery behavior functionality to achieve more robustness. A possible option is to develop new recovery behaviors and use SMACH to continuously run through different recovery behaviors. Possible recovery behaviors could be to back of to a previously visited point, setting a temporary goal very close to the Cyborg. More robust recovery will increase the Cyborg's durability and decreasing the need for human intervention, resulting in a higher degree of autonomy.
- **Compare the current navigation stack with previous ARNL stack:** A comparison of navigation performance between the new navigation stack and the old ARNL system was not conducted, since the primary goal was to *replace*, and not necessarily *improve* the navigation stack. However, I recommend that future projects conduct such an analysis to identify potential weak points in the system, and to further improve it.

# 12 Conclusion

This research aimed to implement a new navigation system on the NTNU Cyborg and optimize the localization performance. Based on the ROS navigation stack, the navigation system have been implemented as a set of ROS nodes that together perform the navigation tasks of mapping, localization, path planning, and obstacle avoidance. The presented result include design and implementation decisions, analysis and configuration of local and global path planners. Additionally, based on a quantitative study of variance convergence in the estimated pose calculated by the Adaptive Monte Carlo Localization algorithm, the localization system was improved, reducing the variance convergence time from 18 to 8 seconds compared to the default configuration.

The modular design of the implemented navigation stack fully replaces the inputs and outputs of the old ARNL based system. As a result, other modules in the Cyborg's ROS network are not directly affected and does not have to be modified to fully function. However, the new navigating stack does not fully replace functionalities provided by the old ARNL system such as docking, wandering behavior, and jog position mode.

The localization performance was improved by conducting a quantitative analysis of the Adaptive Monte Carlo Localization algorithm. By measuring variances in the estimated pose calculated by the AMCL node and tuning relevant parameters accordingly, convergence time was reduced from 18 seconds to 8 seconds, a reduction of approximately 55.6%.

Even though some navigational functionality is lost from the old system, the current navigation stack serve as a solid foundation that allows for a great number of modifications and improvements in the future. The modular design and use of open source code makes the system more robust to bugs, isolated issues, and hardware/software changes. Additionally, the ROS community can now be utilized fully, as modules no longer are limited to legacy code. The longevity of the system has been prolonged, and it is now up to creative minds in the future to develop the next generation of the navigation system.

# References

[1] Morgan Quigley et al. “ROS: an open-source Robot Operating System”. In: vol. 3. Jan. 2009.

[2] *navigation*. URL: http://wiki.ros.org/navigation. (accessed: 06.05.2020).

[3] Y. Goto et al. “CMU Sidewalk Navigation System: A Blackboard-Based Outdoor Navigation System Using Sensor Fusion with Colored-Range Images.” In: (Jan. 1986), pp. 105–113.

[4] Y. Goto and A. Stentz. “The CMU system for mobile robot navigation”. In: 4 (1987), pp. 99–105.

[5] Martin Buehler, Karl Iagnemma, and Sanjiv Singh. *The 2005 DARPA Grand Challenge: The Great Robot Race*. 1st. Springer Publishing Company, Incorporated, 2007. ISBN: 3540734287.

[6] M. Buehler, K. Iagnemma, and S. Singh. *The DARPA Urban Challenge: Autonomous Vehicles in City Traffic*. Springer Tracts in Advanced Robotics. Springer Berlin Heidelberg, 2009. ISBN: 9783642039911. URL: https://books.google.no/books?id=ixtrCQAAQBAJ.

[7] Sebastian Thrun et al. “Stanley: The robot that won the DARPA Grand Challenge.” In: *J. Field Robotics* 23 (Jan. 2006), pp. 661–692.

[8] Michael Montemerlo et al. “Junior: The Stanford Entry in the Urban Challenge”. In: *Journal of Field Robotics* 25 (Sept. 2008), pp. 569 –597. DOI: 10.1002/rob.20258.

[9] Chris Urmson et al. “Autonomous Driving in Urban Environments: Boss and the Urban Challenge”. In: *Journal of Field Robotics* 25 (Jan. 2008), pp. 425–466.

[10] João Fabro et al. “ROS Navigation: Concepts and Tutorial”. In: vol. 625. Feb. 2016, pp. 121–160. DOI: 10.1007/978-3-319-26054-9_6.

[11] S. Brahimi et al. “Car-like mobile robot navigation in unknown urban areas”. In: *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*. 2016, pp. 1727–1732.

[12] Gonzalo Ferrer et al. "Robot social-aware navigation framework to accompany people walking side-by-side". In: *Autonomous Robots* 41 (2017), pp. 775–793.

[13] Rick Dove, Bill Schindel, and Chris Scrapper. "Agile Systems Engineering Process Features Collective Culture, Consciousness, and Conscience at SSC Pacific Unmanned Systems Group". In: *INCOSE International Symposium* 26 (July 2016), pp. 982–1001. DOI: 10.1002/j.2334-5837.2016.00206.x.

[14] Xiaohui Li et al. "Development of a new integrated local trajectory planning and tracking control framework for autonomous ground vehicles". In: *Mechanical Systems and Signal Processing* 87 (Nov. 2015). DOI: 10.1016/j.ymssp.2015.10.021.

[15] Goran Huskić, Sebastian Buck, and Andreas Zell. "GeRoNa: Generic Robot Navigation: A Modular Framework for Robot Navigation and Control". In: *Journal of Intelligent  Robotic Systems* (Oct. 2018). DOI: 10.1007/s10846-018-0951-0.

[16] *Inflation Costmap Plugin*. URL: http://wiki.ros.org/costmap_2d/hydro/inflation. (accessed: 27.04.2020).

[17] A. Babayan. *The Cyborg v3.0*. 2019.

[18] Adept Technology. *pioneer lx User's Guide*. 2013.

[19] *ROS Introduction*. URL: http://wiki.ros.org/ROS/Introduction. (accessed: 18.03.2020).

[20] *ROS Documentation*. URL: http://wiki.ros.org/. (accessed: 18.03.2020).

[21] *Services*. URL: http://wiki.ros.org/Services. (accessed: 14.04.2020).

[22] *actionlib*. URL: http://wiki.ros.org/actionlib. (accessed: 27.05.2020).

[23] *rqt*. URL: http://wiki.ros.org/rqt. (accessed: 04.05.2020).

[24] Thaker Nayl, Dr.Mohammed Mohammed, and Saif Muhamed. "Obstacles Avoidance for an Articulated Robot Using Modified Smooth Path Planning". In: (Sept. 2017), pp. 185–189.

[25] Ronald Uriol Cabrera and Antonio Moran. "Mobile Robot Path Planning in Complex Environments Using Ant Colony Optimization Algorithm". In: (Apr. 2017).

[26] P. E. Hart, N. J. Nilsson, and B. Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107.

[27] D. Fox, W. Burgard, and S. Thrun. "The dynamic window approach to collision avoidance". In: *IEEE Robotics Automation Magazine* 4.1 (1997), pp. 23–33.

[28] H. Andreasson, A. Treptow, and T. Duckett. "Localization for Mobile Robots using Panoramic Vision, Local Features and Particle Filter". In: (2005), pp. 3348–3353.

[29] Horst-Michael Gross et al. “Omnivision-based Probabilistic Self-localization for a Mobile Shopping Assistant Continued”. In: 2 (Nov. 2003), 1505 –1511 vol.2.

[30] Maren Bennewitz et al. “Metric Localization with Scale-Invariant Visual Features Using a Single Perspective Camera”. In: 22 (Jan. 2006), pp. 195–209.

[31] Frank Dellaert et al. “Using the CONDENSATION algorithm for robust, vision-based mobile robot localization”. In: *Proceedings / CVPR, IEEE Computer Society Conference on Computer Vision and Pattern Recognition. IEEE Computer Society Conference on Computer Vision and Pattern Recognition* 2 (Jan. 1999), pp. 2588–.

[32] Pantelis Elinas and J.J. Little. “sMCL: Monte-Carlo Localization for Mobile Robots with Stereo Vision.” In: *Proceedings of Robotics: Science and Systems* (June 2005), pp. 373–380.

[33] J. J. Leonard and H. F. Durrant-Whyte. “Mobile robot localization by tracking geometric beacons”. In: *IEEE Transactions on Robotics and Automation* 7.3 (1991), pp. 376–382.

[34] Dieter Fox and Wolfram Burgard. “Markov Localization for Mobile Robots in Dynamic Environments”. In: *J. Artif. Intell. Res.* 11 (Dec. 1999).

[35] F. Dellaert et al. “Monte Carlo localization for mobile robots”. In: 2 (1999), 1322–1328 vol.2.

[36] Henrik Andreasson, A. Treptow, and Tom Duckett. “Localization for Mobile Robots using Panoramic Vision, Local Features and Particle Filter”. In: (May 2005), pp. 3348 –3353.

[37] A. Kitanov E. Ivanjko and I. Petrovic. *Mobile Robot Localization and Map Building*. 2010. Chap. Model based Kalman Filter Mobile robot self-localization, pp. 59–89.

[38] Sebastian Thrun et al. “Robust Monte Carlo Localization for Mobile Robots”. In: *Artificial Intelligence* 128 (May 2001), pp. 99–141.

[39] burgard W. Thrun S. and Fox D. *probabalistic$_r$obotics*. 2000.

[40] *move_ base*. URL: http://wiki.ros.org/move_base. (accessed: 08.04.2020).

[41] *carrot_ planner*. URL: http://wiki.ros.org/carrot_planner. (accessed: 19.02.2020).

[42] *navfn*. URL: http://wiki.ros.org/navfn. (accessed: 19.02.2020).

[43] *global$_p$lanner*. URL: http://wiki.ros.org/global_planner. (accessed: 19.02.2020).

[44] *dwa_ local_ planner*. URL: http://wiki.ros.org/dwa_local_planner. (accessed: 19.02.2020).

[45] *eband_ local_ planner*. URL: http://wiki.ros.org/eband_local_planner. (accessed: 19.02.2020).

[46] *teb_ local_ planner*. URL: http://wiki.ros.org/teb_local_planner. (accessed: 19.02.2020).

[47] *costmap_2d*. URL: http://wiki.ros.org/costmap_2d. (accessed: 13.05.2020).

[48] Dave Hershberger David V. Lu and William D. Smart. "Layered Costmaps for Context-Sensitive Navigation". In: (2016).

[49] *tf*. URL: http://wiki.ros.org/tf. (accessed: 12.01.2020).

[50] *joint_state_publisher*. URL: http://wiki.ros.org/joint_state_publisher. (accessed: 12.02.2020).

[51] *robot_state_publisher*. URL: http://wiki.ros.org/robot_state_publisher. (accessed: 12.03.2020).

[52] $map_server$. URL: http://wiki.ros.org/map_server. (accessed: 19.05.2020).

[53] *ROSARIA*. URL: http://wiki.ros.org/ROSARIA. (accessed: 24.05.2020).

[54] *amcl*. URL: http://wiki.ros.org/amcl. (accessed: 14.04.2020).

[55] *navfn.h*. URL: https://github.com/ros-planning/navigation/blob/indigo-devel/navfn/include/navfn/navfn.h. (accessed: 24.04.2020).

[56] *Basic Navigation Tuning Guide*. URL: http://wiki.ros.org/navigation/Tutorials/Navigation\%20Tuning\%20Guide. (accessed: 21.05.2020).

[57] Bilge Mutlu and Jodi Forlizzi. "Robots in organizations: The role of workflow, social, and environmental factors in human-robot interaction". In: *HRI 2008 - Proceedings of the 3rd ACM/IEEE International Conference on Human-Robot Interaction: Living with Robots* (Jan. 2008), pp. 287–294. DOI: 10.1145/1349822.1349860.

[58] $rosarnl_node.cpp$. URL: https://github.com/MobileRobots/ros-arnl/blob/master/rosarnl_node.cpp. (accessed: 23.05.2020).

[59] Anis Koubaa et al. *Robot Path Planning and Cooperation*. Jan. 2018. ISBN: 978-3-319-77040-6. DOI: 10.1007/978-3-319-77042-0.

# Appendix

# A | List of Topics

## A.1 RosAria

| **Topic name** |
|---|
| /RosAria/S3Series_1_laserscan |
| /RosAria/S3Series_1_pointcloud |
| /RosAria/battery_recharge_state |
| /RosAria/battery_state_of_charge |
| /RosAria/battery_voltage |
| /RosAria/bumper_state |
| /RosAria/motors_state |
| /RosAria/parameter_descriptions |
| /RosAria/parameter/updates |
| /RosAria/odom |
| /RosAria/sonar |
| /RosAria/sonar_pointcloud22 |

***Table A.1:*** *Topics published by the* `RosAria` *node.*

## A.2 move_base

| Topic name |
|---|
| /move_base/NavfnROS/plan |
| /move_base/TrajectoryPlannerROS/cost_cloud |
| /move_base/TrajectoryPlannerROS/cost_cloud |
| /move_base/TrajectoryPlannerROS/global_plan |
| /move_base/TrajectoryPlannerROS/local_plan |
| /move_base/TrajectoryPlannerROS/parameter_descriptions |
| /move_base/TrajectoryPlannerROS/parameter_updates |
| /move_base/cancel |
| /move_base/cmd_vel |
| /move_base/current_goal |
| /move_base/feedback |
| /move_base/global_costmap/costmap |
| /move_base/global_costmap/costmap_updates |
| /move_base/global_costmap/footprint |
| /move_base/global_costmap/inflation_layer/parameter_descriptions |
| /move_base/global_costmap/inflation_layer/parameter_updates |
| /move_base/global_costmap/obstacle_layer/parameter_descriptions |
| /move_base/global_costmap/obstacle_layer/parameter_updates |
| /move_base/global_costmap/parameter_descriptions |

| |
|---|
| `/move_base/global_costmap/parameter_updates` |
| `/move_base/global_costmap/static_layer/parameter_descriptions` |
| `/move_base/global_costmap/static_layer/parameter_updates` |
| `/move_base/goal` |
| `/move_base/local_costmap/costmap` |
| `/move_base/local_costmap/costmap_updates` |
| `/move_base/local_costmap/footprint` |
| `/move_base/local_costmap/inflation_layer/parameter_descriptions` |
| `/move_base/local_costmap/inflation_layer/parameter_updates` |
| `/move_base/local_costmap/obstacle_later/parameter_descriptions` |
| `/move_base/local_costmap/obstacle_layer/parameter_updates` |
| `/move_base/local_costmap/parameter_descriptions` |
| `/move_base/local_costmap/parameter_updates` |
| `/move_base/parameter_descriptions` |
| `/move_base/parameter_updates` |
| `/move_base/result` |
| `/move_base/status` |
| `/move_base_simple/goal` |

***Table A.2:*** *Topics published by the* `move_base` *node.*

## A.3 AMCL

| **Topic name** |
|---|
| `/amcl/parameter_descriptions` |
| `/amcl/paramete_updates` |
| `/amcl_pose` |
| `/amcl/particlecloud` |
| `/tf` |
| `/initialpose` |

***Table A.3:*** *Topics published by the* `amcl` *node.*

## A.4 joint_state_publisher

| **Topic name** |
|---|
| `/joint_states` |

***Table A.4:*** *Topics published by the* `joint_state_publisher` *node.*

## A.5 robot_state_publisher

| **Topic name** |
|---|
| `/tf` |

***Table A.5:*** *Topics published by the* `robot_state_publisher` *node.*

## A.6 map_server

| Topic name |
|---|
| /map |
| /map_metadata |

***Table A.6:*** *Topics published by the* `map_server` *node.*

## A.7 cyborg_navigation

| Topic name |
|---|
| /cyborg_navigation/current_location |
| /cyborg_navigation/navigation/cancel |
| /cyborg_navigation/navigation/feedback |
| /cyborg_navigation/navigation/goal |
| /cyborg_navigation/navigation/result |
| /cyborg_navigation/navigation/status |

***Table A.7:*** *Topics published by the* `cyborg_navigation` *node.*

## A.8 Other

| Topic name |
|---|
| /initialpose |

***Table A.8:*** *Topics either published manually or through RVIZ.*

# B | RQT Graph

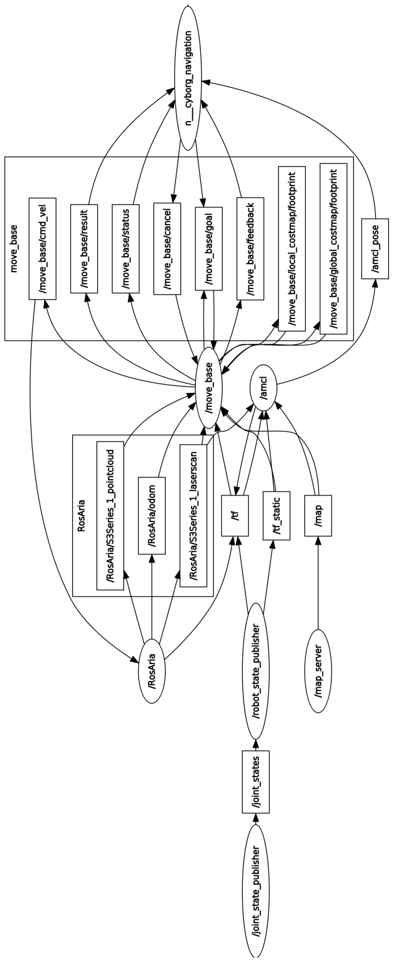

***Figure B.1:*** *RQT graph of the Navigation Stack*

# C Launch Files

## C.1 Cyborg Configuration

```
<launch>
!-- Run ROSARIA, Sensors and odometry -->
node pkg="rosaria" type="RosAria" name="RosAria" output="screen">
param name="publish_aria_lasers" value="true"/>
remap from="RosAria/pose" to="RosAria/odom" />
remap from="RosAria/cmd_vel" to="RosAria/cmd_vel" />
param name="port" value="/dev/ttyUSB0" type="string"/>
/node>

!-- Set up transform configuration -->
param name="robot_description" textfile="$(find
↪  robot_state_publisher)/amr-ros-config/description/urdf/pioneer-lx.urdf"/>
node name="joint_state_publisher" pkg="joint_state_publisher"
↪  type="joint_state_publisher" />
node name="robot_state_publisher" pkg="robot_state_publisher"
↪  type="state_publisher" />

</launch>
```

***Listing 10:*** *Cyborg configuration launch file.*

## C.2 Navigation Controller

```
launch>
master auto="start"/>
!-- Run map server -->
arg name="map_file" default="$(find map_server)/maps/glass_gang.yaml"/>
node name="map_server" pkg="map_server" type="map_server" args="$(arg map_file)"/>

!--- Run AMCL -->
include file="$(find amcl)/examples/amcl_diff.launch" />

node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen">
remap from="odom" to="RosAria/odom" />
remap from="cmd_vel" to="move_base/cmd_vel"/>

rosparam file="$(find cyborg_2dnav)/costmap_common_params.yaml" command="load" ns="global_costmap" />
rosparam file="$(find cyborg_2dnav)/costmap_common_params.yaml" command="load" ns="local_costmap" />
rosparam file="$(find cyborg_2dnav)/local_costmap_params.yaml" command="load" />
rosparam file="$(find cyborg_2dnav)/global_costmap_params.yaml" command="load" />
rosparam file="$(find cyborg_2dnav)/base_local_planner_params.yaml" command="load" />
/node>-

node name="rviz" pkg="rviz" type="rviz" args="-d $(find robot_state_publisher)amr-ros-config/description/urdf/cyborg.rviz"/>
/launch>
```

***Listing 11:*** *Navigation controller launch file*

Lasse Göncz

Development of a New Navigation Stack on the NTNU Cyborg

NTNU
Norwegian University of
Science and Technology