NTNU
Norwegian University of
Science and Technology

The path to Autonomous Inspection using an Unmanned Aerial Vehicle

The Development of a Simulator Framework
and Navigation tools

Brage Gerdsønn Eikanger

Master of Science in Cybernetics and Robotics
Submission date: June 2017
Supervisor: Tor Arne Johansen, ITK
Co-supervisor: Kristian Klausen, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Abstract

Using unmanned aerial vehicles for autonomous inspections of industrial environments in general, and cargo holds and tanks in particular, is a goal that has become within reach the last couple of years, though a lot of development still remains. A problem when developing for inspection of cargo holds is the lack of real world testing opportunities, slowing down development. In this thesis a simulation framework has been developed to speed up the development of autonomous inspection. Furthermore, navigation tools needed for autonomous flight has been implemented and tested within the framework. The simulation framework uses the Gazebo simulator and the Robot Operating System environment to create a framework which focuses on clean interfaces, ease of use, and portability to real world usage. The navigation tools developed were a collision avoidance algorithm and a Simultaneous Localisation And Mapping (SLAM) algorithm, both relying on a 2d-scanning lidar. Being tested in the simulator, the navigation tools performed well, both in normal surroundings and that of a cargo hold, though the size and openness of the cargo hold made the performance very reliant on the range of the lidar. Through the development and testing of the navigation tools, the simulation framework itself was tested and proved to be a valuable and useful resource, providing a flexible and modular interface, which helped the ease of development and speed of testing significantly.

Sammendrag

Bruken av ubemannede droner i autonom inspeksjon av industrielle omgivelser, spesielt et skips lasterom og oljetanker, er et mål som har kommet innen rekkevidde i de siste årene, selv om en god del forskining fremdeles gjenstår. En utfordring når systemer skal utvikles for inspeksjon av lasterom og oljetanker er mangelen på testmuligheter, noe som sinker utviklingen. I denne avhandlingen har et simuleringsrammeverk blitt utviklet med det formål å øke hastigheten autonom inspeksjon utvikles. Videre er nødvendige verktøy for autonom inspeksjon blitt utviklet og testen med bruk av simuleringsrammeverket. Simuleringsrammeverket bruker Gazebo som simulator og Robot Operating System som utviklingsmiljø, og har med disse laget et rammeverk som fokuserer på ryddige grensesnitt, brukervennlighet og portabilitet til den virkelige verden. Navigeringsverktøyene som ble utviklet var en kollisjonsunngåelsemetode og en Simultanious Localisation And Mapping-algoritme, hvor begge bruker en 2d-lidarscanner. Ekperimenter på begge systemene ble gjennomført i simulatoren og begge presterte godt, både i vanlige omgivelser og i lasterom, selv om utstrekningen og åpenheten i lasterommene gjorde ytelsen veldig avhengig av rekkevidden på lidaren. Gjennom utviklingen og testingen av navigasjonsverktøyene ble simululeringsrammeverket i seg selv testet og viste seg å være en verdifull ressurs som tilbyr et fleksibelt og modulbasert grensesnitt, noe som i høy grad forenklet utviklingen av nye verktøy og økte hastigheten på testingen betraktelig.

Preface

This thesis is submitted in partial fulfilment of the requirements for the Master of Science degree at the Department of Engineering Cybernetics at the Norwegian University of Science and Technology (NTNU)

This thesis has allowed me to combine two of my biggest interest: autonomous flight and creating tools which can help others in the future, and to have had this opportunity I am grateful. I want to thank my supervisor Tor Arne Johansen and my co-supervisor Kristian Klausen for their guidance and insights regarding everything I have worked with this spring. Further I would like to thank my colleagues at Ascend NTNU for helping, they have been an invaluable resource of knowledge in anything related to drones and autonomy. My friends who, at the time of writing this preface, still comb through the ever growing text that is my thesis and questions my nonsensical sentences (of which there have been a few), I'm in your debt. Last but not least, for still being there after my hermit-like lifestyle when the deadline was looming, my friends, my colleagues, and my family deserve all the gratitude I can give them.

Brage Gerdsønn Eikanger

Trondheim, June 2017

Table of Contents

List of Tables

List of Figures

List of Algorithms

Abbreviations

API	=	Application Programming Interface
CEPA	=	Cushioned Extended Peripheral Avoidance
DEM	=	Digital Elevation Models
FFLAO	=	Flying Fast and Low Amongst Obstacles
FOV	=	Field of View
FPV	=	First Person View
GUI	=	Graphical User Interface
UAV	=	Unmanned Aerial Vehicle
HIL	=	Hardware In the loop
M8	=	Quanergy M8-1
Puck	=	Veldoyne Puck LITE
ROS	=	Robot Operating System
SDF	=	Simulation Description Format
SIL	=	Software In the loop
SLAM	=	Simultaneous Localisation And Mapping
TiM	=	SICK TiM571
UE4	=	Unreal Engine 4
URDF	=	Unified Robot Description Format
URG	=	Hokuyo URG-04LX-UG01
UST	=	Hokuyo UST-10LX
UTM	=	Hokuyo UTM-30LX-EW

Chapter 1

Introduction

The popularity of unmanned aerial vehicles (UAVs) is increasing fast, and for good reason. The range of use cases for UAVs are huge, and the potential impact they have on a multitude of workplaces and real life situations is similarly large. One of the fields where UAVs really shine is inspection of remote, inhospitable or dangerous areas. Industrial zones, offshore platforms, electrical grids, and wind mills are examples of such areas. Here, autonomy would offer cheaper, faster, and more reliable inspections with little to no risk to human life.

Cargo holds and tanks on ships are often cumbersome to inspect. Much of this is due to their sheer size of the rooms. To get close enough to inspect the walls and roof, scaffolding needs to be erected. In the case of oil and gas tankers, the tank itself needs to be aired out before personnel can enter. Autonomous inspection would drastically reduce the time needed to inspect the tanks and holds on ships, and as no scaffolding erection or usage is required, it would also reduce the risk of accidents.

To become fully autonomous, many tools have to be developed. Some of the most fundamental pieces needed for autonomous inspection are the navigation tools the UAV uses to understand its environment and to move around within it. Understanding the environment comes in many forms, with one of the most useful being to create a map, which would allow for more advanced path-planning and, of course, visualisation. When moving around in the environment the UAV needs to be able to avoid colliding with objects. To guarantee this a collision avoidance scheme is introduced. The collision avoidance scheme needs to be fast, reliable, and should not dependent on other systems to function.

Developing the tools needed for autonomous inspection is difficult without being able to properly test during development. Unfortunately, real life testing is time consuming, the hardware and test facilities are not always available, and can be, if one has a half-working collision avoidance scheme, expensive. Simulation is the answer to this problem. Simulation offers a platform to test the systems in development, and often help with debugging by visualising the problem.

Simulation does however come with its own set of challenges. First, the main point of a simulator is to replace the real world, as such, special care have to be taken to make sure the simulator does not diverge too far from the real world. If something works in the simulator, it ought to work more or less the same in the real world. Secondly, you should be able to switch between the real world and the simulator without the rest of the system noticing. Meaning there needs to be a shared interface between the hardware and actuators of real life, and their simulated counterparts. Lastly, the simulator should be easy to work with. If the simulator increases the overall work needed, the simulator might as well be removed in its entirety.

1.1 Motivation

“Using drones to visually check the condition of remote structural components has the potential to significantly reduce survey times and staging costs, while at the same time improving safety for the surveyors.” (Adams (2016) — DNV GL)

Manual inspections in an industrial environment are time consuming, monotone, and potentially dangerous. These aspects make the task perfect for automation. In particular, the process of inspecting a cargo tank often takes many days, as scaffolding needs to be erected and taken down. The downtime caused by this inspection is very costly, and reducing the time from a couple of days to a couple of hours would go miles to reduce the cost. Letting UAVs take care of inspection would also reduce the chance of any injuries to humans by reducing the number of personnel needed, and eliminating all work with the scaffolding. Until the automation is perfected the UAV could be piloted by a human pilot with many of the same benefits. In support of this, the quote above comes from an article by DNV GL, where two DNV GL surveyors piloted a quadrotor to inspect 14 tanks on a chemical tanker in two and a half days (Adams, 2016).

Whether the UAV is fully autonomous or piloted by a human, a reactive collision avoidance scheme is necessary. Neither the UAV’s sensors nor the pilot’s eyes are perfect and will sometimes steer the UAV on a path leading to collision, which is why we need a collision avoidance scheme. As the dynamics of a typical UAV are fast, the collision avoidance also needs to be fast, and can therefore not be too complex, merely reacting to the potentially dangerous situation and ensuring the UAV does not collide. To ensure fast enough reaction from the collision avoidance is placed close to the hardware ready to step in at a moments notice of a collision is imminent, overwriting the control signals from other modules.

When working on projects in a university setting time is usually of the essence. Taking the time to familiarise oneself to the task and any previously constructed software is often just as time consuming as completing the given task. Reducing this time would significantly increase the output of any person involved. Especially when working with short time spans, such as master theses.

Both for this project and for other projects, a well defined simulation framework for developing algorithms for multi rotors is very time saving and useful. A master thesis only lasts for half a year, having a clear and useful framework is important. Developing your own system each time takes a lot of time, and creates as many systems as there are people. Already having a functioning framework allows a quick setup, and leaves more time for development of new systems. It makes it easier to use and understand previously developed code, and does to some degree guarantee the compatibly of the different software. If the software works in the framework, it will work with the rest of the system.

1.2 Previous Work

There have been many attempts to develop automated inspections. The research cover a wide field; systems for UAV inspections are already in development for inspection of bridges (Mader et al., 2016; Chen et al., 2016), wind mills (Schäfer et al., 2016), culverts (Serrano, 2011), and facades (Eschmann et al., 2012). The two research projects MINOAS (2009) (Marine INspection rObotic Assistant System) and INCASS (2013) (INspection CApabilities for enhanced Ship Safety) has worked on automated inspections of cargo holds for almost a decade (Ortiz et al., 2010).They have developed several solutions to problems such as localisation (Koch et al., 2016), identification of defects (Bonnin-Pascual and Ortiz, 2016), mosaicing visual data (Garcia-Fidalgo et al., 2016). They have also developed a wall-crawler that, with magnetic wheels, drives around on the metallic walls of the cargo holds inspecting the wall as it drives by (Eich and Vögele, 2011).

The usage of Gazebo in conjunction with ROS is common in research, but the setup is seldom explained and often quite narrow in scope. That is not to say there is no work done to create a broad and functional simulator framework. Titled ”Comprehensive simulation of quadrotor UAVs using ROS and Gazebo” Meyer et al. (2012) goes through the nuts and bolts of simulating a quadrotor in a simulator, but how to develop for the quadrotor is not explained. Though through the Hector project developed by TU Dramstadt (Meyer and Kohlbrecher, 2014), several ROS packages has been developed to help development of multirotor systems using ROS and Gazebo.

Taking a look at SLAM and reactive collision avoidance where a lidar is the main sensor, we find quite a lot of work being done in the last few years. Starting with SLAM, a lot of work has gone into creating 3d maps from the sweeps of a 2d lidar. To create 3d maps from a 2d lidar the pose of the lidar needs to vary. Research has looked into both using the movement and natural tilting of a multirotor (Předota, 2016; Nex and Remondino, 2014; Roca et al., 2016; Mader et al., 2015) to vary the pose, and fixing the lidar to a movable platform (Bosse et al., 2012; Wulf and Wagner, 2003; Conrose et al., 2013) to achieve the same effect, only with greater control. Creating 3d maps using SLAM is beyond the scope of this thesis, but when choosing a SLAM method the ability to be expanded from 2d to 3d is important to consider.

Moving on to reactive collision avoidance. For multirotors Scherer et al. (2007) developed a method called FFLAO (Flying Fast and Low Amongst Obstacles) for flying fast in an unknown obstacle dense environment. Through over thousand test flights at speeds up to 10m/s, FFLAO has proved to be reliable and flexible. Much of the performance can be attributed to the lidar used, which is able to detect a 6mm wire from an impressive 58 meters. Unfortunately the lidar is not commercially available and would be too heavy for a small multi rotor. Jackson et al. (2016) developed a lidar based reactive collision avoidance algorithm called Cushioned Extended Peripheral Avoidance (CEPA). CEPA is specialised for multirotors flying in unknown environments, using a flexible weight function to decide the best velocity. CEPA and FFLAO are compared in simulations, with CEPA performing significantly better at especially obstacle dense environments. CEPA, being proposed in the summer of 2016 is still quite rough around the edges and has some fatal flaws; mainly the emergency avoidance provided, which will, in certain scenarios, set a course strait into an obstacle. Improving upon the CEPA algorithm will be one of the goals of this thesis.

1.3 Contributions

This master thesis has two main goals, to develop a useful, feature-rich, and easy to use UAV simulator framework, and, using that framework, take the first steps towards fully autonomous inspection by developing, implementing, and testing collision avoidance and localisation algorithms. For this thesis we will limit our self to using a multirotor as an UAV and a lidar as the main sensor. Through these two goals, the following contributions are made:

- **Evaluate simulators and companion software.** An evaluation of the different simulators available, their pros and cons, and their suitability for our simulation needs. Explain in further detail how the simulator of choice is constructed, its features and usages. Additionally, an evaluation and explanation of the other software packages we need in order to develop the simulator framework, will be given.
- **Develop a simulation framework for simulation of a multirotor.** The development of the simulation framework as a whole. Bridging the gap between the simulator and the different software packages, and developing tools for easy development.
- **Implement collision avoidance.** Starting the journey to completely autonomous flight, a reactive collision avoidance scheme is developed and implemented.
- **Implement SLAM.** Explain the choice of SLAM-method and incorporate an implementation of that Simultaneous Localisation And Mapping (SLAM) system based on a 2D-scanning lidar within the simulation framework.
- **Test implementations.** Design experiments to test the implementations of collision avoidance, and experiments to test the choice of SLAM method and the effect of different lidars using the selected method. All this while also testing the simulator framework.

1.4 Outline

The master thesis will be divided as follows:

- Chapter 2: Nomenclature
- Chapter 3: Simulator. A review of different simulators and an in depth look at the Gazebo simulator.
- Chapter 4: Companion Software. Choosing firmware to represent the flight controller on the multirotor; and choosing environment in which to communicate with the simulator and develop custom software.
- Chapter 5: Simulator Framework. Designing the simulation framework, and developing the tools needed for further software development.
- Chapter 6: Collision Avoidance. Development of a reactive collision avoidance algorithm which can be used during autonomous or piloted flight.
- Chapter 7: SLAM. Introduction to the SLAM problem in general and an explanation of what is needed for our mission. Explains the implementation of the lidar based SLAM we are using. .
- Chapter 8: Experiments and Results. The design and execution of several experiments to test the capabilities of the developed collision avoidance and SLAM, as well as seeing how well the simulator framework works.
- Chapter 9: Discussion. A discussion of the simulator framework with respect to usefulness, utility, and flexibility, in addition to a discussion of the results from the collision avoidance and Slam.
- Chapter 10: Conclusion and Future Work. Concludes the findings in this master thesis and suggest future work.

Chapter 2

Nomenclature

Coordinate frames

Here the different coordinate frames used in this thesis are introduced, including giving information about their usage and which other coordinate frames they are connected to. All coordinate frames are north-west-up or front-left-up, as this is what is used in Gazebo. These are not all the coordinate frames that exist when running the simulations, but these are the most important.

world	The world frame represents reality. It is how the world actually is in the simulator, and all models in the simulator is connected to the world frame if not specified otherwise. Gazebo sends out information about the pose and velocity of any model in its simulator, with the transform being from the world frame
base _frame	The base_frame is the main frame of the UAV. This frames connects the UAV to the world frame as well as the odom frame. The frame is also the base for other coordinate frames that exist on the models, like propellers
hokuyo	The hokuyo frame is the frame the lidar. All scan messages are in this frame. This frame is usually fixed and translated a bit up from the base_frame frame
odom	odom is a world fixed frame with origin in the start position of the UAV. The difference between this frame and the base_frame is based on data given by the UAV's odometry. The odom frame will drift after some time, but will never have any sudden jumps. In addition to be attached to the base_frame the odom frame is also connected to the map frame.
map	The map frame is the world as created by SLAM. If the SLAM algorithm is working well, the world and the map frames should be similar. The map frame is connected to the odom frame. Opposite to the odom frame, the map frame will not drift, but may suddenly jump when SLAM or other software corrects it.

CEPA

Here we take a look at the different parameters and symbols used in the CEPA algorithm.

v_{ref}	The velocity which an external controller wants to give to the UAV.
v_{out}	The velocity CEPA thinks is the best velocity. CEPA ties to keep v_{out} as close as possible to v_{ref} while still remaining safe.
k_1	Tuning parameter for the steering algorithm. Represents the which to keep v_{out} close to v_{ref}'s direction
k_2	Tuning parameter for the steering algorithm. Represents the which to keep v_{out} close to v_{ref}'s magnitude
k_3	Tuning parameter for the steering algorithm. Represents the which to keep a safe distance to obstacles
T	Tuning parameter for the steering algorithm. Represents the time horizon the algorithm look a-head in
K	Tuning parameter for the emergency avoidance. A low value means closer objects are weighted more.
SC_r	The safety cushion defined by r
r_{LB}	The lower proximity limit of the UAV. Any obstacles which are closer than this distance away triggers the emergency avoidance.
r_{UB}	The outer bound for the ”comfort zone” of the UAV. Any obstacles within the cushion defined by this radius is unwanted but can be tolerated.
ψ	The yaw angle of the UAV. $\psi = 0$ is not allways straight ahead, but will in the steering algorithm start at the direction being tested.
$LS(\psi)$	Laser scanner measurements. Measures the distance to the closest obstacle in the ψ direction.

Chapter 3

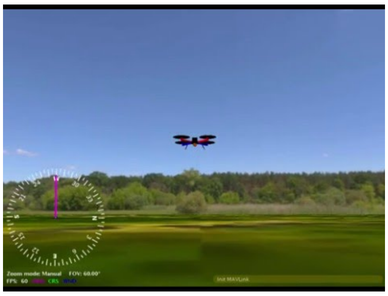
Simulator

In a simulator framework, the simulator is, as the name might suggest, quite important. What we look for in a simulator is a replacement for reality. And we want the transition between the real world and simulation to be as smooth as possible. To accomplish a smooth transition we need two things. First we need a simulator which is close enough to reality that what works in the simulator works in the real world. This does however not mean that the simulator should try to calculate everything. When flying a couple of meters above the ground, gravity is constant. When flying relatively slow using position or velocity setpoints, airflow around the multicopter does not matter. Choosing what to calculate accurately and what to simplify is an art in itself. In this thesis this first aspect will not be much of a problem as we have chosen to look at simulators which are created to simulate multicopters or objects with similar dynamics. We do, however, need to take close look at the second aspect: the interface between the software on one side, and the simulator or the world on the other side. The software we write and test in the simulator should not need to be changed when transitioning to the real world. Somewhere in the system the hardware drivers has to be launched instead of the simulated drivers, but as few files as possible should be affected by this change. To ensure this, we will take an extra look at the interface of the different simulators.

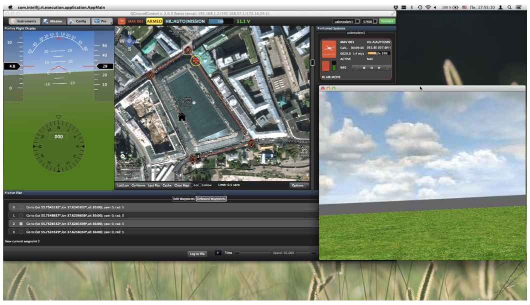
3.1 Choosing a Simulator

There is no one simulator which suits all purposes. Depending on what aspects are the most important for our simulation different simulators with different physics engines are correct. In this thesis we are looking for a simulator which can simulate a multicopter, create a world to fly in, and allow for a multitude of sensors. Physical aspect such as accurate turbulence and other fluid mechanics are not important. As we are going to simulate an inspection the possibility of a camera as a sensor is important. Similarly, as we are going to use a lidar to navigate, support for this sensor is also vital. When flying in the real world we use a flight controller to take care of the low level control of the multicopter. Most firmware on the flight controllers talk to the outside world using a protocol called MAVlink, and having the simulator be able to interface with MAVlink would drastically reduce the work needed to move the software from a simulator to the real world. In addition we are also looking at the API and support of other software in general.

3.2 JMavSim



(a) Flying a multirotor in jMavSim. Photo by PX4 Autopilot Project (2016)/CC BY 3.0



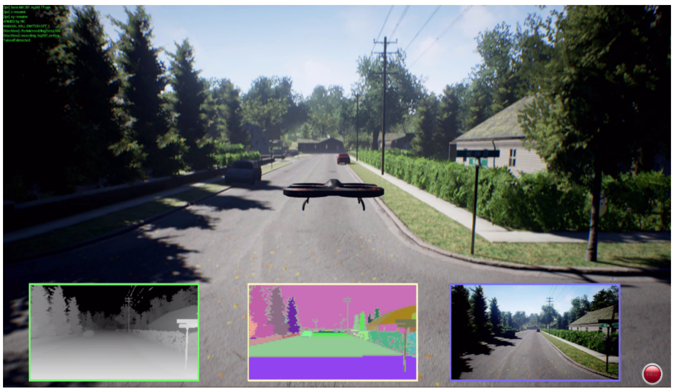
(b) Typical jMAVSim setup with QGroundControl. Photo by PX4 Autopilot (2016)/CC BY 4.0

Figure 3.1: The jMAVSim simulator

The first simulator we take a look at is jMAVSim. jMavSim is developed by PX4 (see chapter 4) and is both a simple and very lightweight multirotor simulator. The simulator has the ability to run SIL and HIL. It has a MAVlink interface and, as it is developed by PX4, is continuously integrated with the PX4 firmware. The graphical aspect of jMAVSim is created by Java3d. jMavSim can be incorporated with both ROS and flight controller firmware. Otherwise it can be communicated with using UDP.

jMAVSim is great for quickly setting up a HIL-simulation to test that the hardware and firmware is working, but for our purposes it is quite limited. With no easy way to incorporate extra sensors or obstacles into the simulation and with the graphics quite lacking, the possibility of simulating a visual inspection is not there.

3.3 Microsoft AirSim


(a) Flying a multirotor in Microsoft AirSim. The three boxes at the bottom are depth camera, depth segmentation, and First Person View (FPV)

(b) Flying in FPV in a forest

(c) Flying in FPV in a neighbourhood

Figure 3.2: Screenshots from Microsoft AirSim, running the Unreal 4 engine. Photos by Microsoft AirSim (2017)/MIT

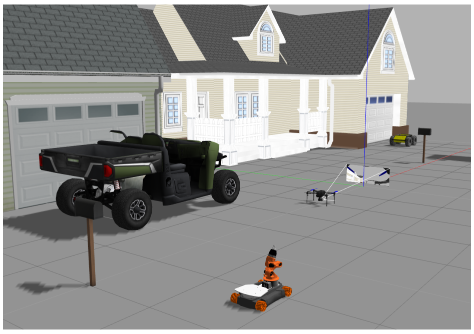
Microsoft AirSim (officially The Aerial Informatics and Robotics platform) is a simulator developed by Microsoft to help the development of machine learning. The project was revealed in February 2017, and is rapidly being developed, though still in its infancy. The main selling point of the simulator is that it uses Unreal Engine 4 to render the simulation, giving it an almost photo-realistic look, as can be seen in Figure 3.2. This photo-realism puts it ahead of the pack when it comes to visual navigation. The simulator comes with the possibility to interface it with Mavlink, making it possible to run SIL and HIL with pixhawk firmware such as PX4 and Ardupilot. APIs for both C++ and Python exist, and an interface for ROS is coming soon. While having implemented both monocular and depth cameras, other sensors, like lidars, are yet to be developed. The simulator is currently only a quadrotor simulator, but other vehicles will be added later.

Microsoft AirSim has great potential, but has not gotten far enough in development to be usable for this thesis. The lack of support for ROS and lidar is quite damning, and because the API still changes at every big update, developed software could be outdated fast. The great visuals can also be a downside, demanding powerful GPUs to run (TitanX or better). The author has no doubt the simulator will develop into a great simulator, but at the time of writing the simulator is not developed enough to warrant use.

For more information about the simulator see Shah et al. (2017b) for general information, and Shah et al. (2017a); Microsoft AirSim (2017) and Appendix B for technical information and tutorials.

3.4 Gazebo

Gazebo is a simulator devloped at the University of Southern Californa as early as 2002. It has been used together with ROS since 2009 and is part of the Virtual Robotics challenge a part of DARPA. Gazebo is currently being developed by the Open Source Robotics Foundation. Unlike the two previous simulators discussed Gazebo is not only a quadrotor simulator, but can simulate any robot, as seen in Figure 3.3a. Gazebo uses Ogre3D as a visualisation tool and allows for a choice between four different physics engines: ODE, Bullet, Simbody, and DART, giving it great flexibility. Support for Mavlink and therefore PX4 and Autopilot with SIL and HIL exist in Gazebo.


(a) Different models in a Gazebo world

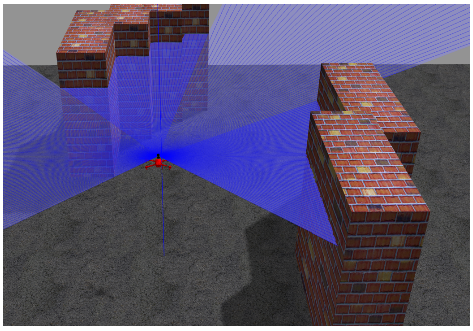
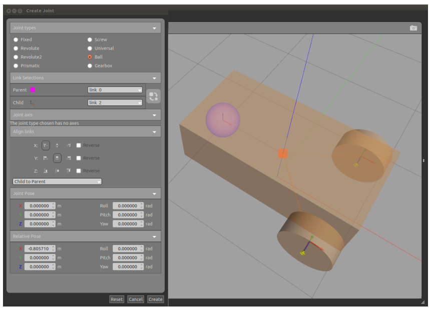

(b) Quadrotor with a lidar simulated in Gazebo. The lidar is visualised with blue rays

Figure 3.3: Examples from the Gazebo simulator

Creating worlds and models is easy in Gazebo. Both are described in a custom xml format called Simulation Description Format (SDF). To create worlds and models one can either use a GUI (see Figure 3.4a) or do it directly in a SDF file. Models can also be imported from meshes created in other programs such as Solidworks or Inkscape, though some extra information about mass and inertia needs to be added. The model editor is shown in Figure 3.4a. The library of models and worlds in Gazebo is quite large, softwares such as ROS and PX4 comes with their own Gazebo models to be used in their simulations, and the community has created even more models, combined they give Gazebo an impressive amount of models to play around with. Gazebo offers the possibility of importing a heightmaps or Digital Elevation Models (DEM). If flying outside, one can import a DEM of the area and practice flying in the same environment. A DEM of Mount Saint Helena is shown in Figure 3.4b.



Fixed, Revolute, Revolute2, Prismatic, Screw, Universal, [✓ Ball], Gearbox

Parent: link_0
Child: link_2

The joint type chosen has no axes

X: Reverse
Y: Reverse
Z: Reverse
Child to Parent

Joint Pose

X	0.000000	m	Roll	0.000000	rad
Y	0.000000	m	Pitch	0.000000	rad
Z	0.000000	m	Yaw	0.000000	rad

Relative Pose

X	-0.805710	m	Roll	0.000000	rad
Y	0.000000	m	Pitch	0.000000	rad
Z	0.000000	m	Yaw	0.000000	rad

(a) The models editor in Gazebo. Figure by Gazebo (2017)/Apache 2.0

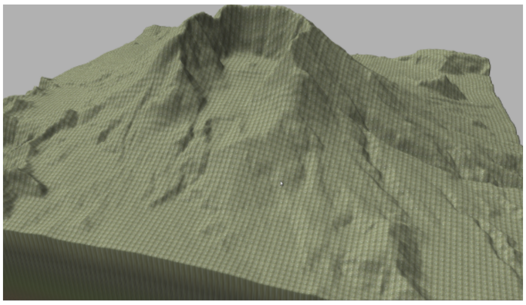

(b) Mount Helena imported into Gazebo as a DEM. Figure by Gazebo (2017)/Apache 2.0, figure has been cropped

Figure 3.4: Pictures from the world and model editor

Sensors in Gazebo are based on highly modifiable generic sensors. The sensors are written as plugins which is connected to any object in the simulation. The list of sensors supported includes: cameras, depth cameras, imus, point lidars, 2d-scanning lidars, 3d-scanning lidars, pressure sensors. Because Gazebo allows for modifying any of these sensors in a model's SDF-file, it is easy to test the performance of different variations of the same sensor in the simulator. Using a 2d laser scanner as an example, the range, frequency, resolution, and FOV can easily be varied; one can therefore easily check the performance difference between a cheap URG-04LX-UG0 and a more expensive UST-20LX, two Hokuyo lidars, to indicate if the price jump is worth it. Templates for creating custom plugins are also available.

Gazebo will be our simulator of choice. While not as visually stunning as Microsoft's AirSim, it has all the functionality we need. It offers easy-to-use interfaces towards other software typically used (ROS, and ArduPilot- or PX4-firmware), and is very flexible when it comes to models and sensors. And though the camera sensors are not as accurate as those in AirSim, they are good enough for our purposes. In the next chapter we will take a look at the other software we need to create a simulation framework, and while doing so we will also look into how they interact with Gazebo.

Chapter 4

Companion Software

To complete the simulator framework we need more than just a simulator. First we need an interface to the hardware on our multirotors. Secondly, to efficiently develop software for the simulator (and a real physical UAV) there needs to be some environment in place to create some structure and standardisation. This is especially important in an academic environment, where many people are to some extent involved. Everyone using the same environment increases the re-usability and collective understanding of the software.

4.1 Simulating a Flight controller

When flying a quadrotor a flight controller is used to handle the low level controls. The flight controllers handle hardware drivers, filtering of the IMU, and converting setpoints to actuator signals. Called firmware as it sits between the hardware and the software, it offers an interface to develop software for. The most common interface for flight controllers is called MAVlink, which was developed in 2009 by Lorentz Meier (also the creator of QGC and PX4). Both flight controllers considered here PX4 and ArduPilot uses MAVlink.

4.1.1 Firmware

PX4 was developed at the Computer Vision and Geometry Lab of ETH Zurich (Swiss Federal Institute of Technology). First both hardware and firmware under the Pixhawk project, now split into PX4 (firmware) and Pixhawk (Hardware). Ardupilot comes from the hobbyist groups at DIYdrones, and started with development of both hardware and firmware, but has switched to firmware only. Both Ardupilot and PX4 runs on the Pixhawk hardware.

ArduPilot and PX4 develops different ground control stations. Ground control stations offers vehicle setup, such as calibration, and flight planner. Ardupilot develops Mission Planner, while PX4 develops QGroundControl (QGC). Mission Planner only runs on Windows, but ArduPilot has a slightly less feature rich version called APM Planner 2.0 which runs on linux and OSX. QGC runs on all operating systems as well as tablets and mobile. All that being said, as all ground control stations supports both ArduPilot and PX4 fimrware, there is little difference.

Both are open source. If planning to change the source code, the quality of the code becomes important. ArduPilot and PX4 have a bit different structure. ArduPilot is to some extend created in Matlab and automatically ported to C++. This creates quite bloated and confusing code, but every variable and function is commented with an explanation. PX4 on the other hand is more tightly structured, but is not so rigorous with comments. As an example The Kalman Filter in PX4 has 1500 lines of code while the Kalman Filter in ArduPilot has 10'000 lines of code.

Another nitpick is that the Gaezbo-plugin for ArduPilot in is not yet part of the master branch of Gazebo, while the PX4-plugin is. Ardupilot still works well with Gazebo, but takes slightly longer time to set up.

They are very similar in functionality. Both open source and have similar communities. Which firmware comes more down to personal preference. Personally I prefer PX4, so that is what I will use for the rest of this thesis. But the framework will be created such that PX4 could be switched for ArduPilot is needed.

4.2 ROS

Robot Operating System or ROS is not as the name suggest an operating system, but rather a framework providing functions similar to an operating system. ROS was developed in 2007 by Stanford Artificial Intelligence Laboratory under the name Switchyard (Quigley et al., 2007), before being introduced under the name ROS in a paper in 2009 (Quigley et al., 2009). ROS is developed to be reusable, providing a framework for developing software with clean interfaces, creating functional packages which can be connected to any other packages. C++, Python and Lisp is officially supported by ROS, and can be used interchangeably. Full support for ROS is currently only available in Unix-based systems, where all code is tested on Ubuntu and OS X, while the community has developed support for many other distributions. ROS is available in Windows, but must be built from source, and many tools do not work.

It should be noted that ROS 2.0 is currently in development, and has a planned release date of December 13th (Arguedas, 2017). ROS 2.0 is based on the same principles as ROS, but addresses some of the weaknesses of the original ROS, such as limited support for real time programming and cross platform usage. For a complete list of the differences between ROS and ROS 2.0 see `http://design.ros2.org/articles/changes.html`.

ROS runs on everything that can run Ubuntu. For multirotors this means that companion computers typically found on-board multirotors can run ROS. Both lightweight computers such as the Odroid and the BeagleBone Black, and more powerful computers such as the IntelNuc and JetsonTX1/2, can all run ROS without any problems.

To explore the capabilities of the ROS environment we will first take a look at what the ROS architecture looks like. Note that this is not an explanation of the ROS file-system, which will only be mentioned briefly, but how the the different ROS executable run and communicate. For more information about the file system see cite. After the architecture introduction, the different tools that makes ROS a great environment are presented.

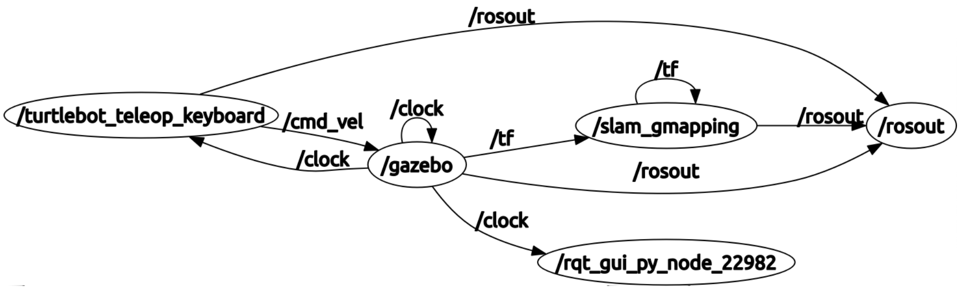


Figure 4.1: Graph created of a simple ROS program. The arrows between the five nodes indicates who communicates with who, the direction of information flow, and the names of the topics. This graph was created using `rqt_graph`.

4.2.1 ROS Architecture

ROS is developed to be modular, peer-to-peer. The environment as a whole is distributed as nodes each executing a small function of the whole program. The nodes communicate with each other either through topics or services. Topics are an a-synchronous communication method where one or more nodes broadcasts messages to a topic; other nodes can subscribe to the topic and will receive any message published to that topic. The node publishing to the topic does not know or care if anyone is listening to the topic, or if other nodes are also publishing to the topic. The subscriber polls the topic and runs a special function called a callback function each time new information is received. The callback function is used to process the incoming message. Services are similar to function calls. A node offering a service will notify the master of its service which is available for any node to see; if a node wants to use the service it contacts the master to receive the contact information of the service provider, it then contacts the service provider and call upon its service. A sample of nodes and topics can be seen in Figure 4.1.

The information is sent as predefined messages (custom messages are possible and easy to implement and use). Messages are structures containing a collection of native variables, (booleans, ints, floats, strings ect.). Messages are constructed to create a common interface for typical information one sends in a robot: geometry messages, navigation messages, or sensor messages are some examples.

Central to the ROS environment is the ROS master or ROS core. The master keeps track of all the different nodes, topics, and services in the the whole eco-space. When a node publishes to a topic, offers a service or in any way tries to obtain contact with another node, they first contact the master to receive contact information about the relevant nodes. The two nodes then create a direct connection using TCP/IP or UDP to transfer data between them. No data passes through the master. The process is visualised in Figure 4.2. The master identifies the different nodes, topics, and services by a string identifier. To make sure no nodes have the same identifier, ROS has good support for namespaces.

In addition to the different nodes, services, and topics, the ROS master also controls the parameter server, which allows shared parameters to be available to every node. More about the parameter server in Subsection 4.2.6.

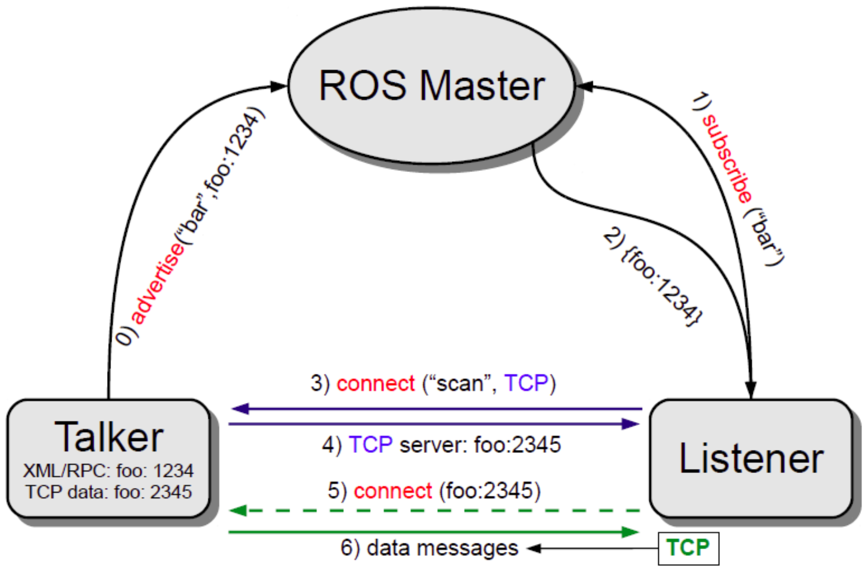


Figure 4.2: The process of connecting two nodes over a topic. The node named Talker tells the master it publishes on the topic ”bar” and can be reached at ip:port foo:1234. The node called Listener tells the master it want to subscribe to the topic ”bar”. The master sees that Talker publishes to ”bar”, and gives Listener Talker’s contact information. Listener contacts Talker and they establish a TCP connection where Talker sends information to Listener. Blasco (2012)/CC BY-NC-SA, figure has been cropped and superfluous parts has been removed

Lastly we come to bags. Bags are the recording functionality of ROS. When running a program, ROS can store the information from topics and store them in bags. Bags can contain information from one topic, all topics, or a select few. The bags can be replayed later. When the bags are played they publish the information they have recorded in the same way they recorded it and on the same topics. When conducting an experiment in real life, all the data can be recorded to be used for further development later.

The distributed architecture of ROS gives us modularity and decoupling. Due to the topics and service interface it has great re-usability, and allows us to select and reconfigure nodes as we please, only renaming the different topics to change the graph of nodes. The reuse of nodes is not limited to self-created nodes, due to the ubiquity of ROS and the well defined interfaces, ROS software from developers around the world is easily reused, no extra setup required. The distributed nature of ROS also allows for ROS programs to run on many different computers at once, all communicating together. The architecture also has a few downsides. First is real time programming; though some community created packages exist to help with real time programming, there is no official support for real time programming in ROS. Luckily ROS 2.0 will have support for real time programming. The second problem comes from the granularity of the ROS nodes. ROS nodes are created to be small and plentiful. When creating a large program the high number of nodes becomes hard keep track of (for the developer, not the ROS master). To solve this ROS offers a plethora of visualisation aids which helps keeping everything organised.

4.2.2 Communicating with ROS

Communicating within the ROS environment is all well and good, but to be truly useful ROS has to be able to communicate with the outside world: e.g. hardware, simulators, and other software.

To interface ROS with Gazebo `gazebo_ros_pkgs` is used. `gazebo_ros_pkgs` creates ROS topics for all the information going in and out of the simulator. ROS and Gazebo are so intertwined that a full installation of ROS includes Gazebo and all the tools needed to use both together.

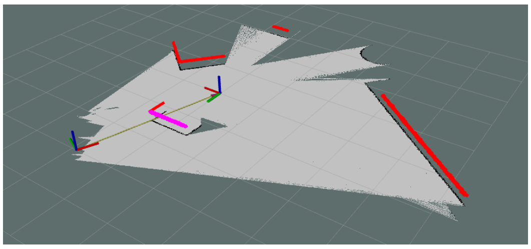
To interface ROS with MAVlink the package `MavROS` is used. `MavROS` works for both PX4 and ArduPilot. The interface consists of topics for commands going into the flight controller and estimation and status updates published by the firmware, and services for switching flight modes, and other options, in the firmware.

When interfacing with hardware, several different solutions are used. For many sensors there already exist drivers, such as the `urg-node` for Hokuyo lidars. For hardware without pre-made drivers, the package `ROSserial` can be used. `ROSserial` offers a serial interface which used to connect to different hardware. For much used hardware such as Arduino-boards there exist specialised packages which incorporates the hardware into the ROS environment, making the software running on the hardware equivalent to the software running on the connecting computer. For a full list of specialised hardware packages see RoboSavvy (2015).

`Rosbridge_suite` allows other non-ROS software to connect to ROS using a JSON API. Other software can subscribe and publish to topics as if they where part of the ROS environment. This also goes for web applications, as `Rosbridge_suite` offers a WebSocket for browsers to interact with.

Last but not least an officially supported MATLAB toolbox lets MATLAB programs be a fully Incorporated part of the ROS environment. This includes creating ROS nodes using MATLAB or Simulink, visualising data live or using bags, and using MATLAB as the ROS master. It also allows for nodes created in MATLAB to be transformed into ROS nodes written in C++. MATLAB can even run the ROS environment all on its own. As MATLAB is much used in academia, allowing all the software developed in MATLAB to be used with ROS and the Gazebo simulator is a great plus.

4.2.3 Rviz


(a) View of a SLAM algorithm creating a map. The red lines are sensor date from a lidar. The grey area is explored and is open, while the black lines are occupied areas.

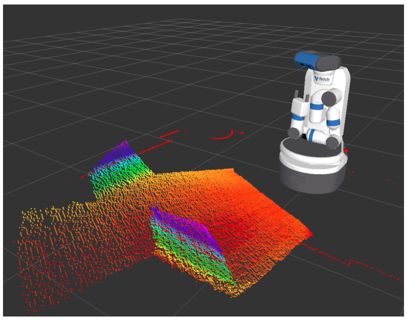

(b) Robot using a depth camera to create a point cloud. Photo by Fetch Robotics (2016) /CC BY NC ND 4.0, photo has been cropped

Figure 4.3: Two different visualisations in Rviz

Rviz is a 3D visualiser for ROS. Seeing how we already have Gazebo to show us how the world looks like, it might seem unnecessary to add an additional bit of software to visualise what we already know. What we get from Rviz however is quite invaluable: we get to see the world though the robot's eyes (or lidars). What is "obvious" to a human observer, might be unobtainable information for the a robot given its sensors and knowledge of the world. Rviz helps us help them (help us).

Watching the map grow during a Slam algorithm, or detecting the behaviour of an collision avoidance or path-planning algorithm lets you detect everything from small odd behaviours, to the biggest flaws. What would be weird, erratic and inexplicable behaviour in Gazebo will many times be easy to explain after using Rviz.

Rviz is highly flexible and can be modified to show most message types defined in ROS. In addition you can impose your own markers into Rviz.

Rviz works as any other node in ROS, listening to selected topics and using the information. To relate the different topics to each other Rivz uses TF (see the next subsection). As long as the transformations exist and the messages are related to a frame Rviz can visualise all of them in the same window.

To describe models Rviz uses a XML-format called URDF (Unified Robot Description Format). This format is similar to the SDF format used in Gazebo but they are not interchangeable. URDF files can be used in Gazebo by using special <Gazebo> tags to give

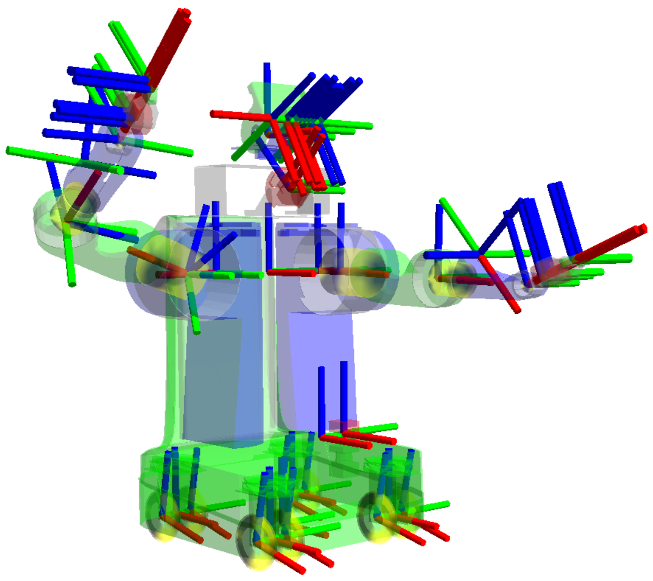


Figure 4.4: Picture of a robots with all its coordinate frames visualised. The sheer amount of frames makes it impossible to keep control manually. Photo from Saito (2015)/CC BY 3.0

Gazebo the extra information it needs. Rvis can however not use SDF-files to visualise models.

4.2.4 TF

In the same way the topics and nodes can get complex, so can keeping track of all the different coordinate frames and the transformation between them. ROS has an elegant solution to this: the TF library.

The TF library is a library dedicated to keeping track of all the coordinate frames and the transformations between them. TF can be called by any node to receive the transformation between two different frames. The frames do not even have to be explicitly given to the TF library: as long as the two frames are connected TF can calculate the transformation. In figure 4.5 we can for example ask for the transformation between the camera and the map. To use TF, simply define your new frame relative to a known frame, and you are set. When creating models for Rviz or Gazebo (using URDF, not SDF), the format automatically specifies the frames and transforms, and can be directly published to TF.

Messages sent in the ROS environment can have headers with information about which frame the message is related to. Using the TF library these messages can be transformed into any other frame. For example if you get a velocity in the world frame, TF can transform that velocity info the body frame and back. A node subscribing to a velocity topic can also check what frame the incoming velocity is represented in, and automatically transform it to the necessary frame. The incoming velocities could even have alternating coordinate framed associated with them and the subscribing node would be unaffected.

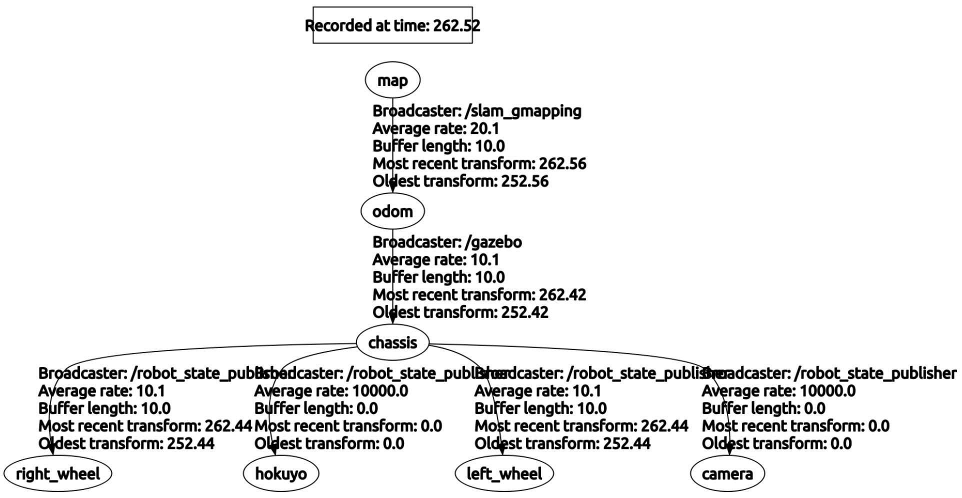


Figure 4.5: Graph of the different frames. One can see which frames are connected to each other, and which node publishes the transforms. Created using `rqt_tf_tree`

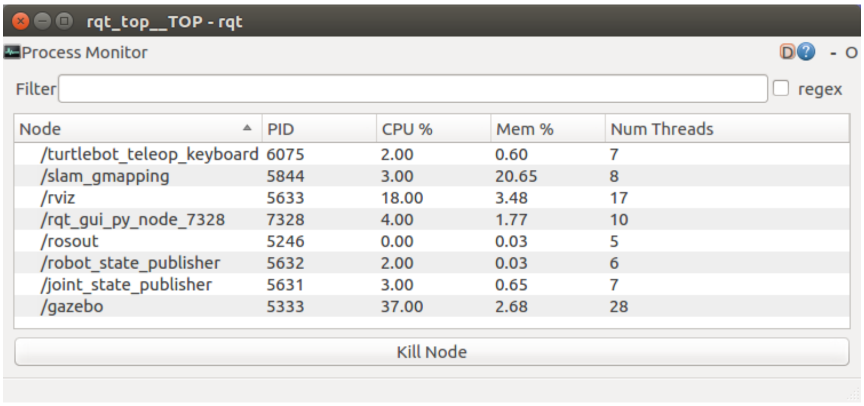
4.2.5 RQT

Rqt is a Qt-based framework for GUI development in ROS. In rqt one can either develop custom GUIs or use one of the many already created. We have already seen two rqt tools in action both Figure 4.1 and 4.5 are created using rqt. Other rqt-packages to visualise different aspects of ROS include `rqt_top` which delivers information about the ROS processes (see Figure 4.6), and `topic_monitor` which delivers information about topic, including publishers and subscribers. Rqt is not only used to inspect, it can also be used to interact with the system. `rqt_publisher`, as seen in Figure 4.7, allows us to easily publish different topics. It is also possible to reconfigure parameters live using `rqt_reconfigure` (see Figure 4.8) which we will talk more about in the next subsection.

Everything rqt does can also be done in a terminal, but the visual aspect helps keep in control even when running a complex system. For a complete list of rqt plugins see (Ze, 2016).

4.2.6 Adjusting without recompiling

When developing systems there is a lot of testing to be done. This often boils down to making a small adjustment in the code, compiling, starting up the system, notice it didn't quite work, and repeat. Compiling takes time. Starting up Gazebo takes time. Changing one parameter does not, but if we have to change the parameter in the source code, we will have to recompile for it to take effect. In ROS there are several tools developed to reduce the amount of time used to recompiling or relaunching the code.


rqt_top__TOP - rqt

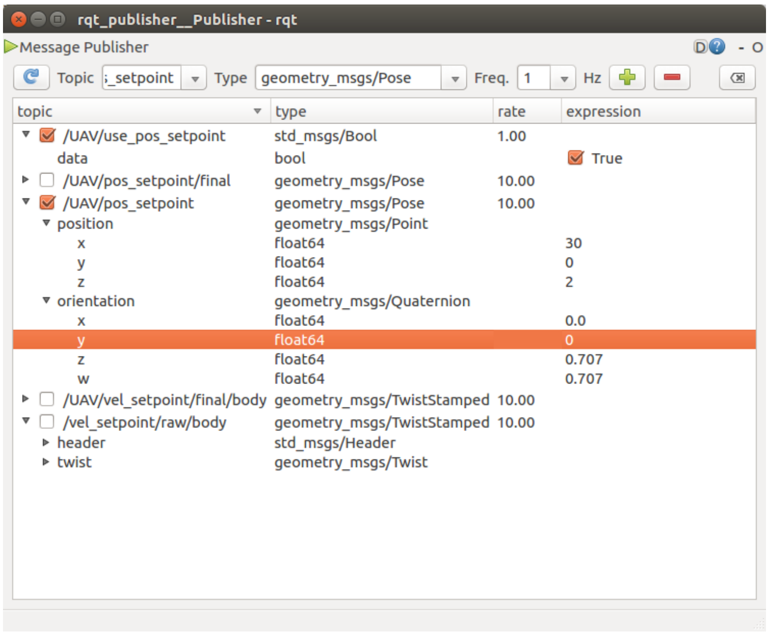
Process Monitor

Filter ☐ regex

Node	PID	CPU %	Mem %	Num Threads
/turtlebot_teleop_keyboard	6075	2.00	0.60	7
/slam_gmapping	5844	3.00	20.65	8
/rviz	5633	18.00	3.48	17
/rqt_gui_py_node_7328	7328	4.00	1.77	10
/rosout	5246	0.00	0.03	5
/robot_state_publisher	5632	2.00	0.03	6
/joint_state_publisher	5631	3.00	0.65	7
/gazebo	5333	37.00	2.68	28

Kill Node

Figure 4.6: The rqt process monitor. This GUI allows us to keep an eye on the performance of the different nodes. Especially helpful if using large simulations


rqt_publisher__Publisher - rqt

Message Publisher

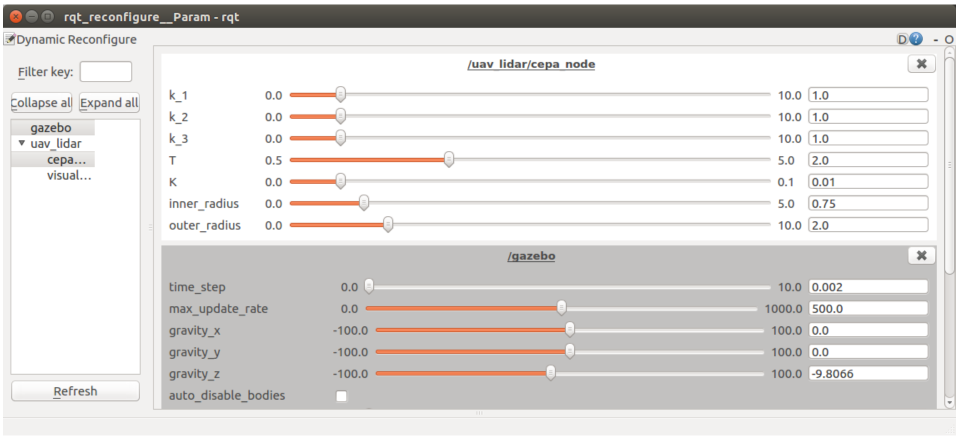
Topic ;_setpoint Type geometry_msgs/Pose Freq. 1 Hz

topic	type	rate	expression
▼ ✓ /UAV/use_pos_setpoint	std_msgs/Bool	1.00	
data	bool		✓ True
▶ ☐ /UAV/pos_setpoint/final	geometry_msgs/Pose	10.00	
▼ ✓ /UAV/pos_setpoint	geometry_msgs/Pose	10.00	
▼ position	geometry_msgs/Point		
x	float64		30
y	float64		0
z	float64		2
▼ orientation	geometry_msgs/Quaternion		
x	float64		0.0
y	float64		0
z	float64		0.707
w	float64		0.707
▶ ☐ /UAV/vel_setpoint/final/body	geometry_msgs/TwistStamped	10.00	
▼ ☐ /vel_setpoint/raw/body	geometry_msgs/TwistStamped	10.00	
▶ header	std_msgs/Header		
▶ twist	geometry_msgs/Twist		

Figure 4.7: The rqt publisher. It provides a graphical interface to publish to different topics. Publishing can easily be turned on and off.

Launch files

ROS nodes are fine grained, and a fully fledged robot may have ten or hundreds of nodes running at the same time. Running each of these nodes in a terminal window of its own would be almost impossible. In a project some nodes are typically official ROS nodes, some are made by the user and some are made by a third party. Combining all these nodes



/uav_lidar/cepa_node

k_1	0.0	10.0	1.0
k_2	0.0	10.0	1.0
k_3	0.0	10.0	1.0
T	0.5	5.0	2.0
K	0.0	0.1	0.01
inner_radius	0.0	5.0	0.75
outer_radius	0.0	10.0	2.0

/gazebo

time_step	0.0	10.0	0.002
max_update_rate	0.0	1000.0	500.0
gravity_x	-100.0	100.0	0.0
gravity_y	-100.0	100.0	0.0
gravity_z	-100.0	100.0	-9.8066
auto_disable_bodies	☐		

Figure 4.8: The rqt dynamic reconfigurer. This GUI allows us to change the values of different parameters while the system is running. In this figure the CEPA collision avoidance system, and the Gazebo simulator can be tuned live.

to match node and topic names by changing the source code in each and every node would be tiresome and clunky. To launch multiple nodes at once, and change topic names of the nodes, ROS uses launch files. Launch files are XML-files that explains to ROS which nodes to run and with what arguments they should be given. Changing the topic names in launch files can easily alter which nodes are connected. In addition to nodes, launch files can also launch other launch files. Letting us create a layered approach which gives great flexibility and re-usability of the launch files.

Parameter Server

Often there are some constants throughout the project which are shared. If defined in each source file changing the parameters takes time and it is easy to forget some of them. The parameter server in ROS allows for a centralised storage of the shared parameters. Any node that need one or more of the parameters needs only to receive them when they start up. The parameters are stored in a YAML-file, an understandable and simple text file which can easily be changed.

Dynamic Reconfigure

Even changing parameters in a lunch file or parameter file, is sometimes too time consuming. When tuning a controller or similar, changing the parameters of an algorithm while the programming is running drastically speeds up the tuning process. Dynamic reconfigure is ROS' tool for this purpose. Dynamic reconfiguration is implemented in the code the same way callback function are, being called every time a variable is changed. This means that except for the callback function the rest of the code remains unchanged. Dynamic reconfiguration can be done via the terminal, or by using `rqt_dynamic_param` which gives us a graphical interface (see Figure 4.8).

ROS Console

Sometimes when debugging, printing a message to the terminal is the best way to figure out what is going on. This does however have the unfortunate consequence of littering the code with a lot of print-statements, which regularly will be added, removed, commented and un-commented as development moves forward. To bring some stability to this ROS provides `rosconsole`. `rosconsole` has five different outputs of different levels of importance: debug, info, warning, error, and fatal. Print-statements are written with a predefined importance level. Which messages are printed is called the verbosity level and can be controlled in launch files, parameter files, terminals, or using other tools. The verbosity level can be changed for all nodes or for one singular node. When not outputting the print-statement the overhead for having the statement there is minimal. It is also possible to write conditional outputs, that only print if some statement is true, or throttled outputs, which only outputs at a fixed rate.

Being able to not only turn a node's console output on and off, but also be able to let some messages be printed, while others are ignored, is very practical. One can turn off all debugging-statements until something weird happens, and then turn them on. Not cluttering the console when not needed, and able to turn on while running, instead of shutting everything down, un-commenting the print-statement, recompiling, and launching everything again. It is recommended to use the GUI `rqt_console` to view all messages as it has great support for sorting and filtering the messages.

4.2.7 ROS file system

This section is dedicated to explain the fundamental building blocks of the ROS file system. This will not be a comprehensive explanation; the scope is to ensure the reader is familiar with and understand the meaning of terms used later in this thesis.

Packages are the most fundamental part of the ROS filesystem. Packages can contain nodes, messages, services, libraries, parameter serves among other things, as long as they are related to the function of the package. Most software written for ROS is distributed as packages. As part of each package both a manifest and a make file is contained within it. The make file contains information about what piece of the package should be compiled and distributed, and how to do so. The manifest is a file containing all the meta data about the package, such as which dependencies is has, who the maintainer is, and which licence it is distributed under. The names and directories within the packages are for the most part set, meaning locating needed files are easy. For example all launch files are in the launch directory,all headers are in the include directory, and all source code is in the src directory.

Examples of a packages is the `cepa` package, the software for which we will develop in Chapter 6. The `cepa` package contains a the implementation of an algorithm called CEPA, which is implemented as a node. It also contains a node which sends visualisation data to Rviz. In addition to this the package contains a Hokuyo lidar models which allows us to test the algorithm without running a full quadrotor each time. It contains several launch files to launch the nodes in different configurations, and a configuration file to run with dynamic parameters.

Chapter 5

Simulation Framework

In this chapter we will pieces together the systems presented in the last two chapters into a simulator framework. The goal of the framework is to offer a place where it is easy to start development, and where conducting test are quick. There will additionally be a focus on creating reusable modules. Lastly we will look at the master thesis package, which is not a part of the simulator framework, but is used to run the experiments posed in this thesis.

5.1 Overview

The most important part of any simulator framework is the simulator. For our framework we use the Gazebo simulator which is well rounded and flexible simulator. In this thesis we are going to simulate a quadrotor, and to get an as accurate representation of the quadrotor as possible, we simulate the flight controller that would run on the physical quadrotor. Here we use the PX4 flight controller and run the controller as Software-in-the-loop (SIL), simulating the physical hardware the flight controller would normally run on. PX4 uses a protocol called Mavlink to receive and send commands. In addition to the flight controller we need to have a model for the rest of the multirotor, sensors, and worlds to fly around in. These custom models and worlds are connected directly to the Gazebo simulator.

The rest of the software we run in the Robot Operating System (ROS) environment. ROS provides basic communication tools as well as some software packages with a lot of useful functionality. The most prominent of the ROS software packages we will be using are:

- MavROS. MavROS creates a bridge between PX4's MAVlink and the communication tools of ROS.
- Rviz. Rviz is a 3d visualisation tool which can visualise data from the different sensors on the quadrotor
- TF. TF is a library which stores and keeps track of all the coordinate frames and transformations in the system.

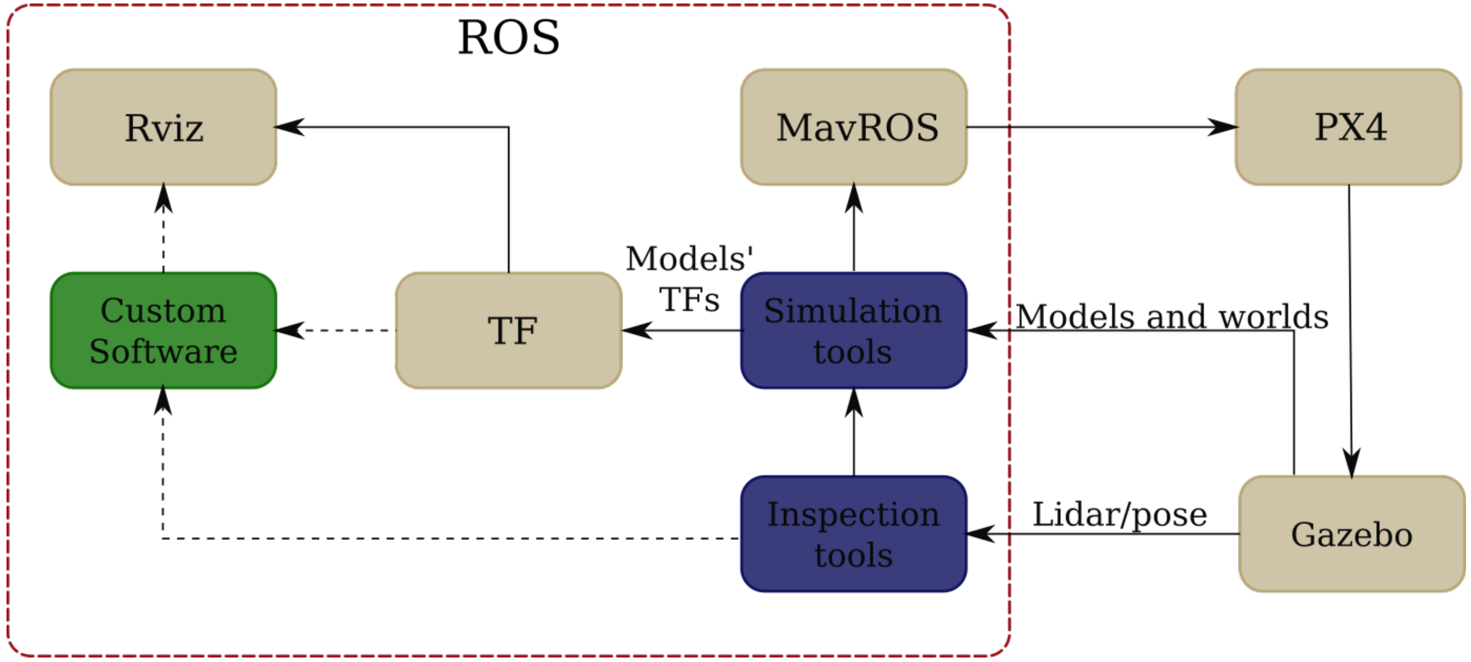


Figure 5.1: An overview of the simulation framework. The blue part are the two development tool which are developed for this framework, while the green parts are software which uses the simulation framework.

5.2 Development tools

Even with the powerful software of ROS, Gazebo and PX4 at our disposal, there are still functions that are needed to create the framework we want. In this section we will talk about the development tools which have been created as part of the simulation framework. The tools have been split into two packages with different responsibilities: simulation tools and inspection tools. The simulation tools creates an easier interface towards PX4 and Gazebo, while the inspection tools contain several helpful functions for developing UAV inspection software.

5.2.1 Simulation Tools

To ease our interface with the simulator the simulation tools package has been created. The goal of this package is to simplify the start-up phase and provide commonly needed tools. It is also a place to store all our custom models and plugins. A simple setup script will make sure Gazebo can use this package and all the models and plugins in it.

The tools this package provide are as follows:

MavROS Publisher

The MavROS Publisher is a node who provides an interface to the mavlink protocols. It sits between the MavROS node and the rest of the system. It takes care of publishing at a high enough frequency to satisfy the mavlink protocol, and sets the flight controller in the correct modes. the node starts up the UAV, and sends it to hover 2 meters over the ground, awaiting further orders.

This package was created to easily give both velocity and position setpoints without needing to know the intricacies of the mavlink protocol. As all information to the flight controller goes through this node it also ensures that not both velocity and position setpoints are sent to the flight controller at the same time. Whether to output velocity setpoints or position setpoints can be controlled via parameters, and altered during the simulation.

TF publisher

As SDF files can not be directly imported into the TF-library, a translator is needed to publish transforms. This node takes information from given by the simulator and publishes it to the TF-library. This includes the transformation from world to base_frame and odom to base_frame. The latter is needed to perform SLAM. See the nomenclature for more information about the frames.

Models

The package contains directories in which to store models. Gazebo does not use models outside its own libraries without being told so. A script in this package sets up all the necessary environment variables needed for Gazebo to work with these models.

5.2.2 Inspection Tools

This package has been developed to provide small helping function when developing systems for autonomous flight. These are created to allow the developer to focus on developing their system without needing to create and re-create helper functions.

The tools this package provide are as follows:

Yaw regulator

The Yaw regulator is a node which regulates the orientation of the UAV such that the front of the UAV is facing the direction of travel. The regulator turns of when hovering to not spin around due to small adjustments.

Position Controller

The position controller takes in a position setpoint and outputs a velocity setpoint. It is not path planner only aiming directly for the position setpoint. The information regrading the position of the UAV can be obtained from the odometry of the UAV, or one can cheat and take the exact position directly from the simulator.

Lidar median filter

Both an actual lidar and the simulated lidar in Gazebo has noisy measurements. This node offers a median filter to remove the noise.

Lidar padding

Most lidars does not have 360 degree FOV. For those methods that need it this node will ”pad” the measurement such that it covers 360 degrees. The value of the fake laser scans added in the padding can be adjusted.

5.3 The Master Thesis Package

Outside the simulation framework itself, but necessary to this thesis is the Master thesis package. The master thesis package is created to contain all the launch files and setup files needed to run the experiments in described in this thesis. This package could have been distributed over the other packages, but to keep the other packages from being dependent on each other, and to not clutter the packages with files unnecessary outside the master thesis, the master thesis package exist. The development tools are created to be reused. This package is created for this thesis only.

The master thesis package includes instructions on how to setup the different experiments. It contains all the necessary launch files, Rviz setup-files and world-files. The master package does not offer any new functionality, it only takes advantage of the functionality of the development tools, ROS, Gazebo, and Px4 to conduct the experiments explained in Chapter 8 on the CEPA-algorithm and the SLAM algorithm.

5.4 Example of usage

To better understand how the different nodes and packages work together, an example of the simulator framework in use is given below. Here we see how the collision avoidance algorithm communicates with the different nodes of the simulation framework

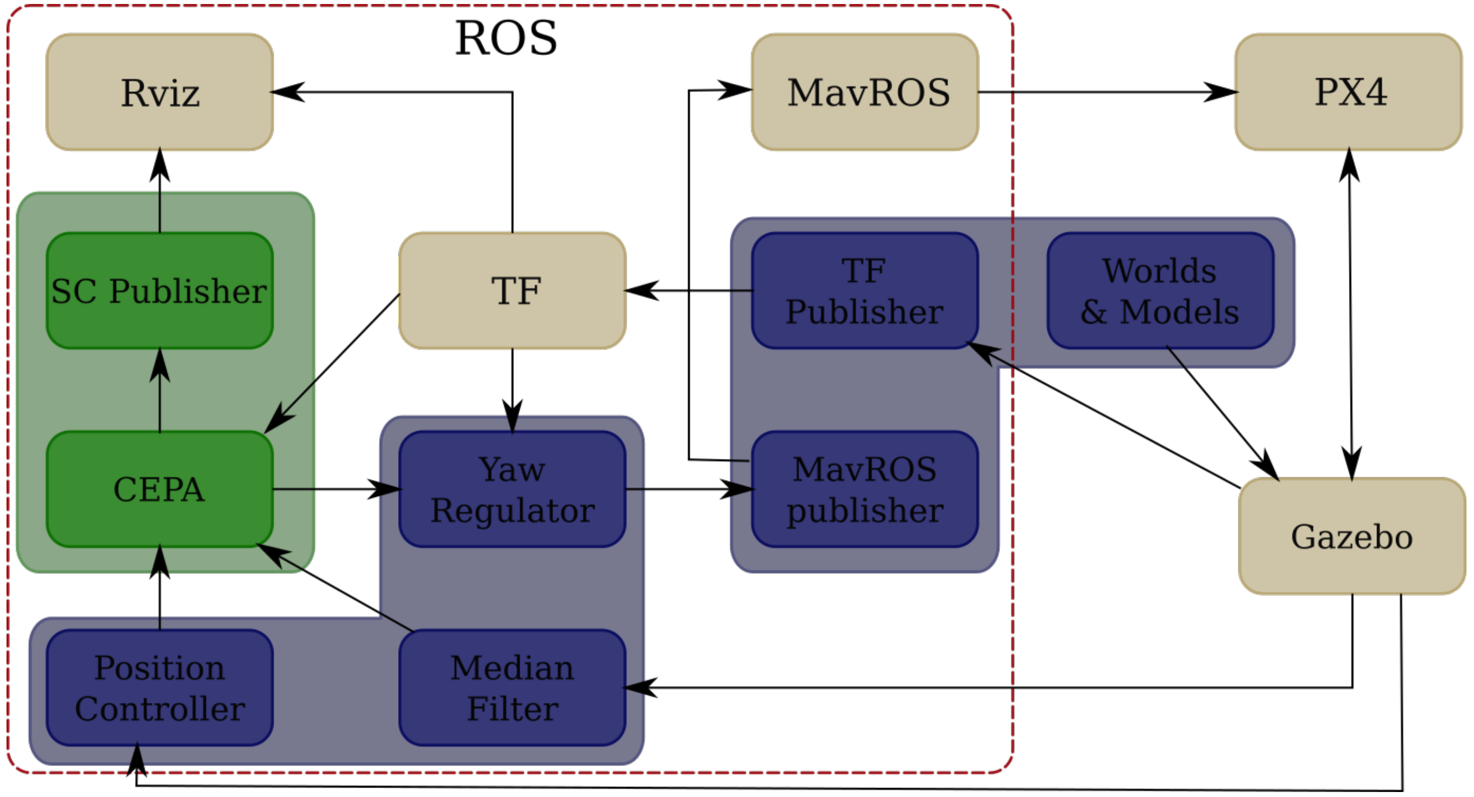


Figure 5.2: The Simulation Framework used in conjunction with CEPA (see the next chapter), the larger blobs are packages, while the smaller blobs are ROS nodes.

Chapter 6

Collision Avoidance

Avoiding collisions is a necessary part of any UAV's flight. Whether the UAVs is wholly autonomous or piloted by a human, avoiding crashing into the surroundings is vital. Global collision avoidance in the form of path planners take steps to ensure a safe flight by making sure to avoid known obstacles, but to ensure no collisions, there needs to be a system in place to take care of unknown and/or dynamic environments. For this purpose we have reactive collision avoidance, which works in the local level, and quickly reacts to any potential obstacle by changing the trajectory of the UAV. The change of trajectory might have some additional effects, like keeping the same height or speed, but mainly the focus is on avoiding a crash.

In the rest of the chapter we will introduce a reactive collision avoidance scheme proposed by Jackson et al. (2016). After the introduction, some issues the scheme has is revealed and solutions to these issues are developed. In the end the implementation of the scheme and how it fits into the simulation framework is explained.

6.1 Cushioned Extended-Periphery Avoidance: a Reactive Obstacle Avoidance Plugin

Jackson et al. (2016) presented the paper describing the Cushioned Extended-Periphery Avoidance (CEPA), at the 2016 International Conference on Unmanned Aircraft Systems. In this paper, we see the development of a reactive object avoidance system for a multirotor, based on the data gathered from a 2d scanning lidar.

CEPA consists of two parts: a steering algorithm and emergency avoidance. The steering algorithm tries to balance two factors: keeping the output velocity as close to the reference velocity as possible, and avoiding obstacles. The reference velocity (v_{ref}) is typically given by a path planner, but can come from other sources, such as a pilot as well. During normal operation, the emergency avoidance remains dormant, only waking if any obstacle comes dangerously close. The emergency avoidance ignores any reference velocity in favour of moving away form obstacles as fast as possible. For the sake of low latency, the whole algorithm is performed in the lidar's reference frame, using polar coordinates. Each lidar measurement becomes one possible direction the algorithm considers, in other words the angular resolution of the algorithm is equal to that of the lidar used.

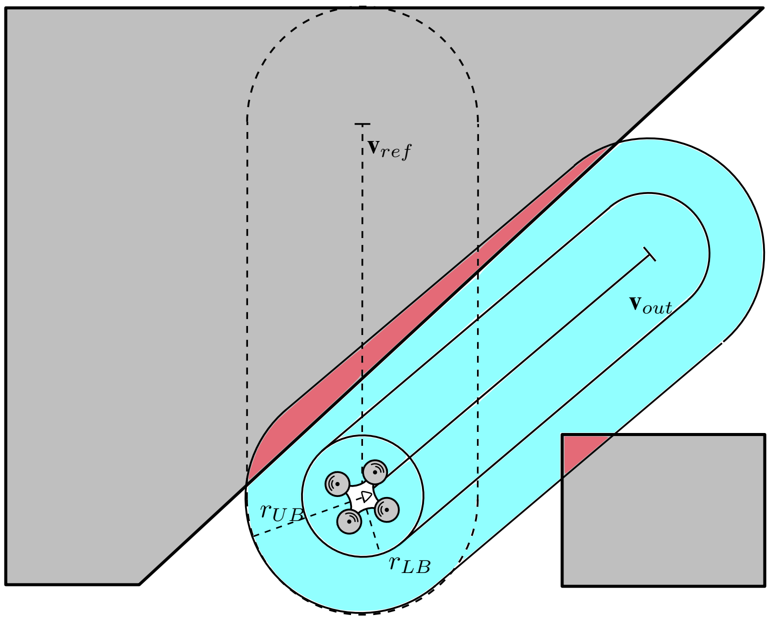


Figure 6.1: The reactive avoidance of CEPA. $\mathbf{v}_{ref}$ is the input velocity, $\mathbf{v}_{out}$ is the output velocity,r_{LB} and r_{UB} are lower and upper radius. Figure redesigned based on a figure from Jackson et al. (2016)

The main idea behind CEPA is two cushions which are projected forwards, where keeping obstacles from intruding into the cushions is the main idea. The inner cushion represents where the UAV physically will travel, while the outer cushion is a comfort zone. No direction where obstacles intrude the inner cushion is considered, while the outer cushion can be intruded, but intrusions will count against the direction when choosing the optimal direction. Iterating through the lidar measurements, each angle is measured based on a cost function where the lowest score to the highest degree complies with the factors mentioned in the last paragraph. Figure 6.1 shows the basic concept. As we can see, the reference velocity ($\mathbf{v}_{ref}$) would see the UAV drive into a wall. After running the steering algorithm $\mathbf{v}_{out}$ is chosen to be the best direction for the UAV.

The next subsections describes in detail how the steering algorithm and the emergency avoidance are constructed, but a brief pseudo code for the CEPA algorithm as a whole is given in Algorithm 1

Algorithm 1 The CEPA algorithm, overview

```
1:  function CEPA($\mathbf{v}_{ref}, laser\_scan$)
2:      if $emergency(laser\_scan)$ then
3:          $\mathbf{v}_{out} \leftarrow getEmergencyVelocity(laser\_scan)$
4:      else
5:          for all $measurements$ in $laser\_scan$ do
6:              $weight \leftarrow getWeight(measurement, laser\_scan)$
7:              if $weight < weight\_min$ then
8:                  $weight\_min \leftarrow weight$
9:                  $\mathbf{v}_{out} \leftarrow getVelocity(measurement)$
10:             end if
11:         end for
12:     end if
13:     return $\mathbf{v}_{out}$
14: end function
```

6.1.1 Steering Algorithm

The steering algorithm is the main part of CEPA. The steering algorithm takes into account both the reference velocity and keeping a safe distance to obstacles. This is accomplished by a cost function. The cost function is used give a weight to each possible direction the UAV can travel, where the lowest weight is the most optimal direction, and is passed on to the UAV. Figure 6.2 shows the physical interpretation of most of the variables used in the following description. Note: $\mathbf{v}$ denotes the velocity being sent through the cost function, while $\mathbf{v}_{out}$ is the velocity which is sent onward from the collision avoidance: the best of all tested $\mathbf{v}$s.

The cost function consists of three terms. The two first terms are based on the factor that the output velocity should be as close to the reference velocity as possible. This can be represented as the inner product between the reference velocity and the output velocity, and the magnitude of the output velocity relative to the reference velocity. With k_1 and k_2 as tuning parameters, the two first terms in the cost function are:

$$- k_1(\mathbf{v}^T\mathbf{v}_{ref}) - k_2\frac{|\mathbf{v}|}{|\mathbf{v}_{ref}|} \tag{6.1}$$

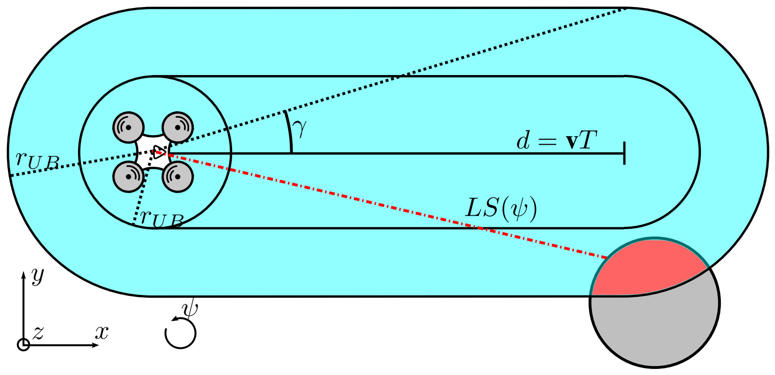


Figure 6.2: Showing the inner and outer Safety Cushions, defined by r_{LB} and r_{UB} respectively. γ is here shown for the outer cushion. $LS(\psi)$ is the distance measurement from the lidar as angle ψ.

The last term is used to keeping a distance to obstacles. By projecting two cushions (Figure 6.2) in-front of the UAV, CEPA can look ahead to see if any obstacles are in the current path or are close to the path. The edge of the cushions are defined by:

$$SC_r(\psi, \mathbf{v}) = \begin{cases} r\ csc(\psi) & \psi \in [\gamma, \frac{\pi}{2}) \\ r & \psi \in [\frac{\pi}{2}, \frac{3\pi}{2}] \\ -r\ csc(\psi) & \psi \in (\frac{3\pi}{2}, 2\pi - \gamma) \\ d\ cos(\psi) + \sqrt{r^2 - d^2 sin(\psi)} & \psi \in [2\pi - \gamma, \gamma) \end{cases} \tag{6.2}$$

Where $d = ||\mathbf{v}||T$, T being a parameter denoting the time horizon the algorithm looks ahead in. $\gamma = atan2(r, d)$, is the angle where the straight part of the cushion and the circular part meets. Creating two cushions, SC_{LB} and SC_{UB}, defined by the lower (r_{LB}) and upper (r_{UB}) radiuses respectively, allows us to calculate the cost or weight of a direction. This is written as:

$$\Omega(\mathbf{v}) = \sum_{i=0}^{N} \kappa(\psi_i) \tag{6.3}$$

where N is the number of measurements in one scan of the lidar, and

$$\kappa(\psi_i) = \begin{cases} \infty & LS(\psi_i) \in [0, SC_{LB}(\psi_i)] \\ f(SC_{UB}(\psi_i) - LS(\psi_i)) & LS(\psi_i) \in (SC_{LB}(\psi_i), SC_{UB}(\psi_i)) \\ 0 & LS(\psi_i) \in [SC_{UB}(\psi_i), \infty) \end{cases} \tag{6.4}$$

Where $f(x)$ is a positive definite function, e.g. $f(x) = x^2$. $\Omega(\mathbf{v})$ is the last term in the cost function. k_3 being a tuning parameter, the cost function looks like

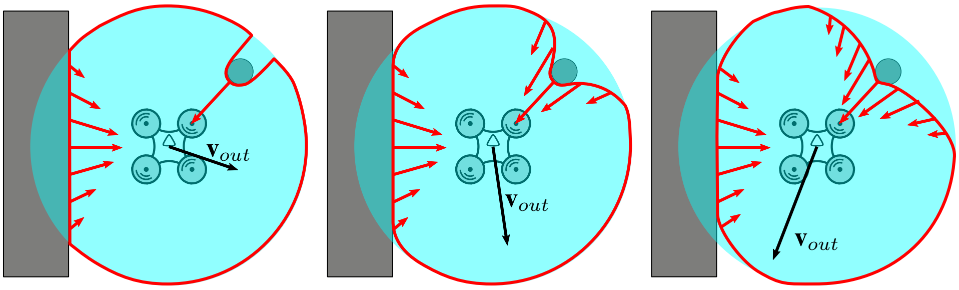
$$k_3\Omega(\mathbf{v}) - k_1(\mathbf{v}^T\mathbf{v}_{ref}) - k_2\frac{|\mathbf{v}|}{|\mathbf{v}_{ref}|} \tag{6.5}$$

Varying k_1, k_2, and k_3, the cost function can be modified to prioritise keeping the same heading, keeping the same speed, or staying clear of obstacles respectively. $\mathbf{v}_{out}$ is the $\mathbf{v}$

which minimises the cost function. Formally written as:

$$\mathbf{v}_{out} = \underset{\mathbf{v}}{\operatorname{argmin}} \left\{ k_3 \Omega(\mathbf{v}) - k_1 (\mathbf{v}^T \mathbf{v}_{ref}) - k_2 \frac{|\mathbf{v}|}{|\mathbf{v}_{ref}|} \right\} \tag{6.6}$$

6.1.2 Emergency Avoidance



(a) $K = \infty$, equal to no filtering. Emphasising size over intrusion **(b)** $K \approx 0.01$, emphasising both size and intrusion **(c)** $K \approx 0.005$, emphasising intrusion over size

Figure 6.3: The emergency avoidance with different Ks

If any obstacle intrudes on the circle defined by the inner radius (r_{LB}), the emergency avoidance kicks in. The emergency avoidance ignores the reference velocity. The direction and the magnitude of the output velocity is calculated as follows. Sum up all the intrusions by obstacles into the circle defined by the inner radius (see red arrows on Figure 6.3); the output velocity is the sum of these intrusions. To ensure smaller, but closer obstacles are not out-weighted by larger objects which are farther away, the laser scan is sent through a filter which limits the maximum difference between one laser scan measurement and the next (in space, not in time). How big the difference can be is decided by the parameter K. The difference in K can be seen in the figure 6.3. This emergency avoidance has several benefits. It is very fast, and as the magnitude of $\mathbf{v}_{out}$ scales with the level of intrusion, the reaction is typically more smooth.

6.1.3 Conclusion

CEPA is promising in a lot of aspects, the reactive avoidance especially. The cost function is highly modifiable and is easy to tune for different UAV behaviours, tuning it live or changing the parameters when changing from one state to another in some state machine could be very helpful. Calculating everything in the natural frame the sensor uses, in this case the lidar, speeds up the process a lot, it really uses its strengths well. There are still some potential improvements which we are going to discuss in the next section.

6.2 Improved CEPA

CEPA as originally proposed is quite un-optimised. While still lightweight, the algorithm consumes up to several orders of magnitude more processor time than needed, depending on the situation. More critically still, the emergency avoidance, which has the singular task of ensuring no collision will, in situations described below, potentially be the cause of collisions. In this section the problems concerning the original algorithm will be laid out, and solutions are given.

6.2.1 Steering Algorithm

The steering functions in the original CEPA has no fatal flaw but is very inefficient. Every time a new output velocity has to be calculated the algorithm goes through every possible direction. As neighbouring directions only differ by the resolution of the lidar (typical 0.25 degrees) the overlap between the cushions tested in each direction is huge. Further, the algorithm will run through all directions regardless of weather or not it is possible to get a lower result from the cost function. There is also no regards to the dynamics of the UAV: whether the UAV is speeding along at top speed or slowly hovering, makes no difference. The improvements proposed are as follows:

Testing the Optimal Direction First

When flying in areas with no obstacles, or no obstacles in the direction given by the reference velocity, the optimal $\mathbf{v}_{out}$ is $\mathbf{v}_{ref}$. All other directions will give a higher score, and thus there is no reason to test them. After checking for an emergency, but before starting to calculate the cost of all the velocities we check if there are any obstacles in the outer cushion in the direction of $\mathbf{v}_{ref}$, if not we set $\mathbf{v}_{out} = \mathbf{v}_{ref}$ and return. This is shown in algorithm 2 in lines 8-9.

Removing Unviable Directions

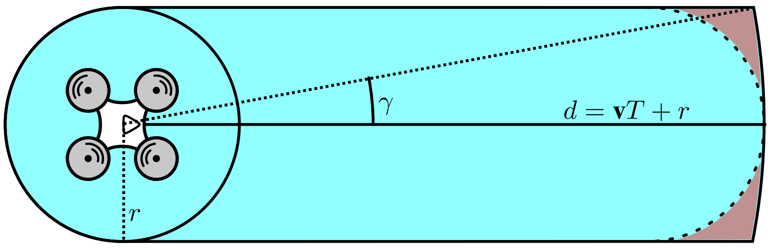


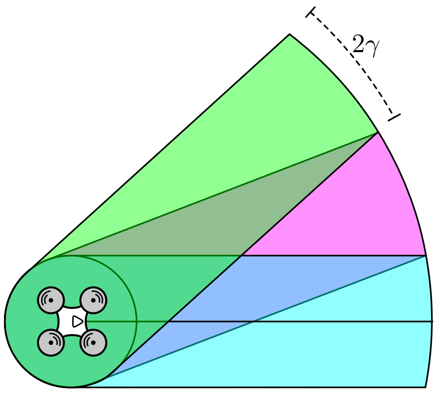
Figure 6.4: Showing CEPA Improve safety cushions. The red shaded areas are the extra area covered. The old cushion is marked in a stapled line. Though the extra area does not seem big, it allows for whole circle arcs to be checked at once.

As any intrusion into the inner cushion disqualifies the direction from being a possible choice, and by realising the intrusion will most appear in the subsequent direction due to almost full overlap, we can potentially reduce the amount of velocities we need to calculate the cost of.

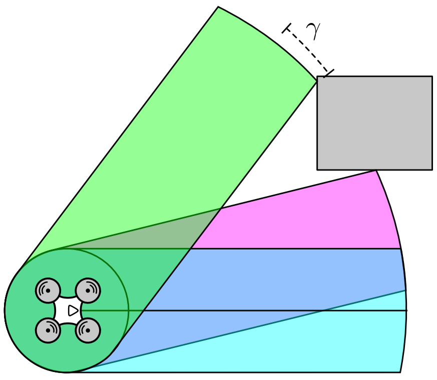
Before we continue with this thought we need to adjust the safety cushion. The original safety cushion as seen in Figure 6.2 has its ends with circles defined by the same radius as the safety cushion (SC_{LB} or SC_{UB}). On the positive side, this ensures we do not check any excess areas, but on the negative side it also ensures that we have to check every direction to guarantee we do not miss any obstacles. By extending the cushion as seen in Figure 6.4 we let the outer edge of the safety cushion follow the arc of a circle with a radius of $\mathbf{v} * T + r$, where r is r_{LB} or r_{UB}. The new safety cushion is defined by the equation:

$$SC_r(\psi, \mathbf{v}) = \begin{cases} r\ csc(\psi) & \psi \in [\gamma, \frac{\pi}{2}) \\ r & \psi \in [\frac{\pi}{2}, \frac{3\pi}{2}] \\ -r\ csc(\psi) & \psi \in (\frac{3\pi}{2}, 2\pi - \gamma) \\ d & \psi \in [2\pi - \gamma, \gamma) \end{cases} \tag{6.7}$$

Where $d = ||\mathbf{v}||T + r$ and $\gamma = asin(r, d)$. Now that we have our new safety cushions we can follow the radius of the circle defined by d and check arcs of up to 2γ degrees at once. Even more important, if an obstacle is detected we can skip past it easily, we do not have to wastefully check all the directions which collide with that obstacle. When an obstacle is detected we know that the closest $\mathbf{v}$ that avoids a collision is γ degrees from either side. Figure 6.5b visualises the process. The magenta scan detects an obstacle and measures its width. Velocities γ away from the obstacle are marked as the edges towards that obstacle, and every velocity in between is discarded as invalid. This binary nature is only valid for r_{LB}. This is represented in algorithm 2 in line 11.



(a) CEPA Improved safety cushion



(b) CEPA Improved skipping over obstacles

Figure 6.5: CEPA Improved's new safety cushiones.

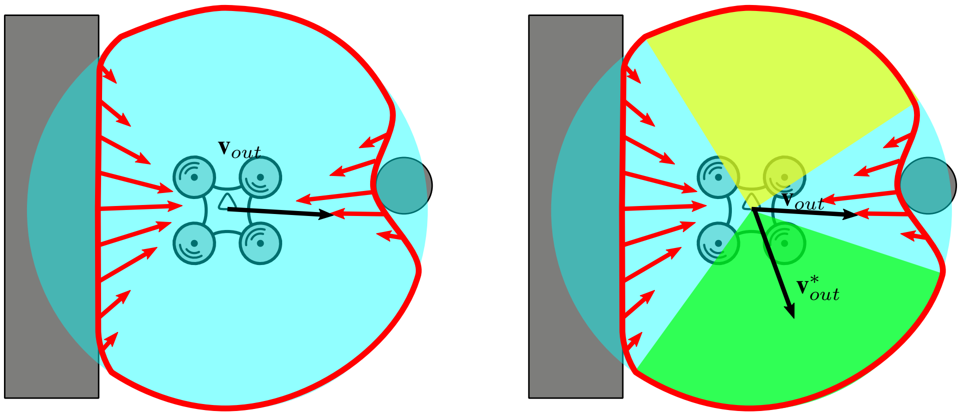
Finding the best Velocity in a Sector

Many sectors of the laser scan are monotonically increasing in cost from one side to the other. This work especially well in open sectors, where no objects are present. In these sectors there is a possibility for increased efficiency. In a monotonically increasing sector the edges are the maxima and minima of the cost function. Choosing the minima of the edges gives us the best velocity possible, making it possible to skip all the other velocities in the sector.

Adjusting tuning parameters based on speed

The cost of changing direction at high velocities is much higher tan at low velocities. The weight on keeping the same direction should therefore increase as the velocity increases. Prioritising to slow down before turning. By adjusting the weight k_1 we can accomplish this. Setting $k_1 = ||\mathbf{v}_{UAV}||k_4 + k_5$ the value of k_1 will vary with the speed of the UAV. And we can still tune k_4 and k_5.

6.2.2 Emergency Avoidance



(a) CEPA emergency avoidance failure

(b) When the suggested velocity $\mathbf{v}_{out}$ causes a collision, the failsafe kicks in and finds a clear sector the UAV can drive in. $\mathbf{v}^*_{out}$ is the middle of the clear sector closest to the original $\mathbf{v}_o ut$.

Figure 6.6: Emergency avoidance failure and solution

The emergency avoidance proposed in the original CEPA has many good qualities, but has one fatal flaw: it causes collisions. When confronted by a symmetrical environment the summation of the intrusion (the red arrows, see Figure 6.6a) will either equal zero or set the output velocity towards one of the obstacles. Obviously this is no good. As the emergency avoidance works well in most other circumstances, we will leave it as is, but add a sanity check at the end. If the proposed direction is toward an obstacle, rather than move in that direction, move instead towards the centre of the closest open section. Figure

6.6b visualises this. The emergency avoidance orders the UAV to avoid collision by flying into the cylinder to the right. There are two open sections where the UAV can fly without colliding marked in yellow and green; both sections are possible, but we would like the one with the edge closest to $\mathbf{v}_{out}$. The green section is closer to $\mathbf{v}_{out}$ than the yellow section, so the green section is selected. The new, safe, velocity ($\mathbf{v}^*_{out}$) is then the middle of the green section. The reason we choose the centre of the section, not the edge closest to the original proposed $\mathbf{v}_{out}$, is because the UAV has a physical radius and would collide with the obstacle even if the infinitesimally thin ray from the lidar did not.

In algorithm 2 this is implemented in line 4-6

6.2.3 Conclusion

By implementing the improvements listed above we get the following algorithm:

Algorithm 2 The CEPA Improved algorithm, overview

```
 1: function CEPA_IMPROVED($\mathbf{v}_{ref}, laser\_scan$)
 2:     if $emergency(laser\_scan)$ then
 3:         $\mathbf{v}_{out} \leftarrow getEmergencyVelocity(laser\_scan)$
 4:         if $isSafe(\mathbf{v}_{out}) = false$ then
 5:             $\mathbf{v}_{out} \leftarrow getClosestSafeVelocity(\mathbf{v}_{out}, laser\_scan)$
 6:         end if
 7:     else
 8:         if $cushionClear(\mathbf{v}_{ref})$ then
 9:             $\mathbf{v}_{out} \leftarrow \mathbf{v}_{ref}$
10:         else
11:             $valid\_directions \leftarrow getValidDirections(laser\_scan)$
12:             $sectors \leftarrow splitIntoSectors(valid\_directions)$
13:             for all $sectors$ do
14:                 $\mathbf{v} \leftarrow sector.best\_v$
15:                 $weight \leftarrow getWeight(\mathbf{v}, laser\_scan)$
16:                 if $weight < weight\_min$ then
17:                     $weight\_min \leftarrow weight$
18:                     $\mathbf{v}_{out} \leftarrow \mathbf{v}$
19:                 end if
20:             end for
21:         end if
22:     end if
23:     return $\mathbf{v}_{out}$
24: end function
```

Lines 4-6 ensures the emergency avoidance does not cause a collision. Lines 8-10 passes $\mathbf{v}_{ref}$ on to $\mathbf{v}_{out}$ if the cushion in the $\mathbf{v}_{ref}$ direction is clear. Line 11 does a coarse search, removing any directions where the UAV is guaranteed to collide. Line 12 divides the remaining directions into sections where finding the best **v** is relatively simple (monotonically increasing cost). Lines 13-18 calculates the best **v** from each sector, and chooses the best to be returned.

This improved version of CEPA has sacrificed some of its simplicity for efficiency. The explanations in the previous subsections hopefully justifies the changes, but to truly know the difference they make testing has to be done.

6.3 Implementation

To be able compare CEPA and Improved CEPA both algorithms were implemented, though in the following explanation only CEPA is mentioned. No pseudo code or implementation of the original CEPA is available to the authors knowledge, meaning the implementation of the original CEPA might differ in performance from the one described in Jackson et al. (2016). To get the implementation described here see(ref Appendix A).

Using ROS and C++, CEPA is implemented as a class. To incorporate the algorithm into the ROS environment we make the whole thing one node which subscribes to the laser scan and $\mathbf{v}_{ref}$ topics, and publishes $\mathbf{v}_{out}$. As CEPA has quite a lot of parameters which need tuning, the algorithm was outfitted with the dynamically configurable parameters, meaning the algorithm could be tuned while flying. The algorithm works in the body frame of the UAV. To ensure all incoming reference velocities are transformed into the body frame, the TF-library is used. Checking the frame of the reference velocity, the implementation transforming reference velocity to the body frame if needed. If no frame is given in the reference velocity, the UAV's body frame is assumed.

There are some differences between the pseudo code given above and the implementation. The information flow is the same, but as CEPA is implemented as a class, such data as the lidar scan is stored in a member variable, and does not need to be given to the different functions.

6.4 Testing

A detailed explanation of all experiments conducted is given in chapter 8. This section exist only to list the relevant experiments, and give an overview over the different experiments.

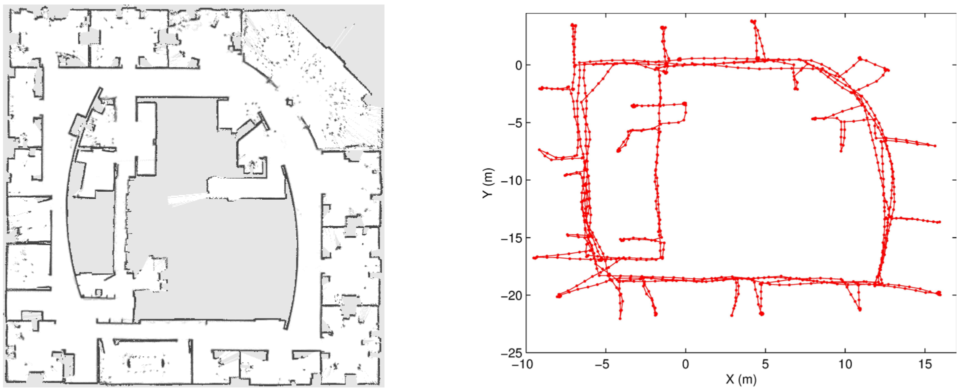
There are two aspects we are going to test in the experiments. First is the improved performance in the steering algorithm of CEPA. Secondly is the improved emergency avoidance, where we will check it can avoid collisions where the original emergency avoidance would cause them.

Chapter 7

SLAM

In this chapter we will look at Simultaneous Localisation And Mapping or SLAM. We will, in this chapter, look at SLAM from a practical standpoint rather than presenting everything in a purely theoretical fashion. First we will present the different challenges our use case has, and find the version of SLAM best suited to face them. After the best theoretical solutions is selected we look at how to implement that version of SLAM in our simulator framework.

7.1 Quick Introduction to SLAM



(a) The Intel building made by gmapping, a particle filter graph-based SLAM. Figure by Stachniss (2014)/CC BY 3.0

(b) The Intel building mapped by linear SLAM, a feature based SLAM. Figure by Zhao et al. (2013)/CC BY-NC-SA 3.0

Figure 7.1: Two different maps created by two different SLAM methods. Both based on the same dataset.

In this section a short introduction to SLAM will be presented. The introduction is not comprehensive and will not go into the mathematical notation. It is a introduction of the principals of SLAM, and veterans of SLAM can safely skip to the next section, where we use the information gained in this section is used to choose the version of SLAM we are going to use.

SLAM is a tool used for both mapping an unknown environment, and localising yourself in it, which in a bit of an oxymoron, as we need a map to localise ourselves, and we need to know our position to create a map. SLAM methods tries to find the best possible map it can with a probability distribution which describes how certain it is of its correctness. SLAM is more a tag given to software trying to solve the SLAM problem than one given method, as there are many different theories and implementations all trying to solve the SLAM problem. To roughly split the methods in two we haves: feature based and graph based SLAM.

Feature based SLAM finds different features in the environment and keeps track of information about the feature's localisation with regards to itself and the other features. Features can be corners or visual patterns for SLAM using cameras while it can be structural protrusions or intrusion for lidar based SLAM. For both lidar and camera based SLAM the features should the unique and recognisable. A graph based map created of the Intel building (a common dataset to test SLAM-algorithms) can be seen in Figure 7.1b.

Graph based SLAM is another method, where we create an occupancy map instead of a map of features. The occupancy map is made up of a grid where each square represent a physical localisation. Each square is binary: either occupied by an object or empty. The graph based SLAM gives an easier to understand map for humans, but comes at the cost of taking up more space. As opposed to feature based SLAM, where information about the features' localisation to each other is stored, graph based SLAM's occupancy grid operates under the assumption that no squares on the grid are related to each other in any way. This makes computation much easier, but also means the method does not care if the map created is physically feasible, or to some extent makes sense compared to previous measurements. A map of the Intel building created by graph based SLAM can be seen in Figure 7.1a.

Another way of splitting SLAM methods in half is between Kalman filter based SLAM and particle filter SLAM. Kalman filter SLAM, or EKF (Extended Kalman Filter) SLAM, is based around representing the position and the map as a normal distribution. This has several benefits, mainly computational, as a normal distribution can be represented by two parameters (mean and variance) no matter how the distribution looks, and creates a linear representation. EKF SLAM are typically feature based SLAM. Some of the issues concerning EKS SLAM comes from its representation of the probability as a normal distribution. If the actual probability distribution strays to far away from a normal distribution, or worse: is multi-modal, EKF SLAM still has to represent the probability as a normal distribution and can therefore stray far away from the actual map, or, in the case of a multi modal distribution, chose the wrong mode, and forever after be on the wrong track. Particle filters solves these problems at the cost of being more heavy on processing and memory. In the particle filter the probability distribution is made up of many particles distributed proportionally around the probability distribution. Because of this particle filter SLAM works on any probability distribution, but need to store information about each particle, as opposed to just the mean and the variance. To accurately represent a distribution, the SLAM method need enough particles, and how to reduce the number of particles while still retaining a good representation is a much researched problem.

A special version of the particle filter is called Rao-Blackwell particle filters. Here each particle retains a complete map. This is even heavier on memory (and the problem gets worse the bigger the map is), but allows for much fewer particles. The maps of the different particles are at each iteration checked up against the likelihood of getting the sensor information just received given that the map is correct. If some particle have score much lower than the rest these particles are removed and replaced by clones of the best particles. This will in the end lead to the best map, but care must be taken not to remove particles to often less one is quickly to end up with one guess and we are back to the problems of EKF SLAM. This problem is called particle starvation.

The SLAM algorithms are implemented using different sensors. The two most common being cameras and lidars. The lidars automatically finds volumetric information and is well suited for graph based SLAM, though it can be used for feature based SLAM, but the features needs to be structural. SLAM using cameras is widely used and is very flexible, being used both in graph and feature based SLAM. Camera based SLAM does however demand high processing power to analyse the pictures at a high enough speed. The benefit of camera based SLAM is however that cameras are much cheaper and lighter than lidars, which both are welcome features, and a frame from a camera typically contains much more information than a lidar scan, it is just much harder to get the information out of the sensor data.

7.2 Our Choice of SLAM Method

For this choice we will mainly focus the mission of inspecting a ship's cargo holds and gas tanks. These rooms are huge, often spanning tens of meters in each direction. The walls are not entirely featureless, they often have some repetitive pattern. Most of the time however the walls vary in the horizontal direction while being identical in the vertical. The walls are metallic, but are coated with protective layers which vary the reflectivness and sheen they have.

By the scope of this thesis, a lidar is used as the main sensor. The lidar will give us important structural information about the rooms we are in which can later be used to create paths for the UAV to follow when inspecting.

Given the repetitive nature of the walls the choice of feature based SLAM would be a poor one. There are features in the rooms, but many will be indistinguishable from each other. Using graph based SLAM would also allow us to create a physical model of the room where we later can project photos to using mosaicing techniques (see Garcia-Fidalgo et al. (2016)). As we do not from the start know where the UAV will start, and due to the repetitive nature of the walls, there is a high risk to "lock on" to the wrong protrusion or column. The SLAM algorithm should therefore carry with it information about multiple possibilities to increase the chances of correct localisation. To accomplish this we use a SLAM method based on a particle filter. As the rooms (though big compared to other rooms) are quite small in area we can afford to have every particle in the filter associated with a map as in a Rao-Blackwell particle filter.

7.3 Implementation

A website called Openslam (`openslam.org`) is dedicated to sharing SLAM implementations and is where we look to see if anyone has already created an implementation of the version of SLAM we are going to use. A library called `gmapping` has implemented the version of SLAM we want to use, and we will use this library. `gmapping` is developed by Giorgio Grisetti, Cyrill Stachniss and Wolfram Burgard at the University of Bonn. This implementation takes in the data from a lidar scanner; it also needs the transformations from the odometry frame to the base_frame, and from the lidar (Hokuyo) frame to the base_frame. It output the transformation between the map frame and the odometry frame. For more information about the frames see the nomenclature (Chapter 2).

7.4 Testing

The SLAM algorithm will be tested in the simulator. There is however not a agreed-upon method of measuring the quality of SLAM methods (Kummerle et al., 2009). This is mostly due the lack of the ground truth, the absolute knowledge of where the UAV was, and where the obstacles is not available. In this thesis we are going to rely on visual inspection of the maps produced by the SLAM algorithm with a variety of lidars.

We will test the SLAM implementation in both a normal environment and one similar to that of a cargo hold. We will also test with different lidars. A full list of the experiments as well as a more in-depth description is given in Chapter 8.

Chapter 8

Experiments and Results

In this chapter experiments for the navigation tools developed in the previous two chapters are proposed and the results are presented. Each experiment will have an explanation of its purpose and its method, as well a results presented in the most suitable fashion. The discussion of the results will happen in the next chapter. In addition to experiments on the navigation tools, some measurements regarding the performance of the simulator are recorded. These measurements are taken from the other experiments, and the method describing the experiment is therefore omitted, replaced with a note on how the measurements were recorded.

Information about how to run or replicate these experiments yourself is given in Appendix A.

8.1 CEPA

There are mainly two elements we want to test with CEPA. First is the improvements made in run time in the overall algorithm. Here we will run both the original CEPA and the improved CEPA in the same environments with the same parameters and inputs to see the difference. Secondly is to see the improvements made in the emergency avoidance. Here we will test normal scenarios and scenarios where the original emergency avoidance will struggle.

Experiment 1: CEPA Runtime

In this section we will measure the runtime of the CEPA algorithm. The original CEPA is already fast enough to complete within the time allotted by every lidar scan, which run at 40Hz (giving the algorithm 25ms to complete). There are usually other, more time intensive algorithms running on-board a UAV, and any time reduced will give those algorithms more wiggle room.

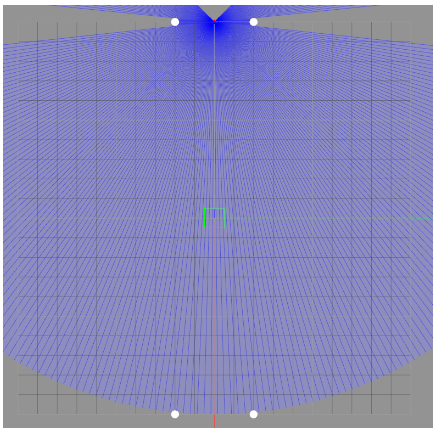
Method

We fly the UAV in different environments. A figure of the environment with the goal position superimposed is given for each environment. The position controller will try to steer the UAV directly towards the goal position. The computation times are averaged over the flight.

Results

Computation Time [ms]	Scenario 1	Scenario 2	Scenario 3	Scenario 4
CEPA	14.3	16.9	17.0	16.9
Improved CEPA	~ 0	3.2	5.1	1.1

Table 8.1: Time it takes the two CEPA versions to complete its calculations.


(a) Scenario 1: a completely empty plane

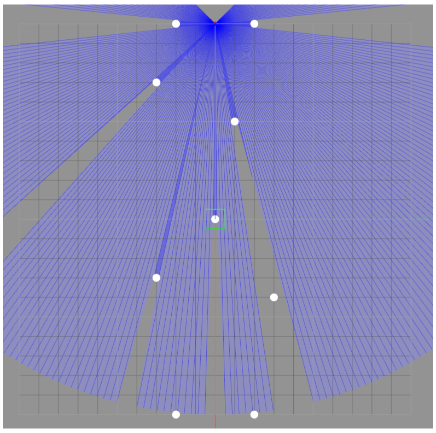
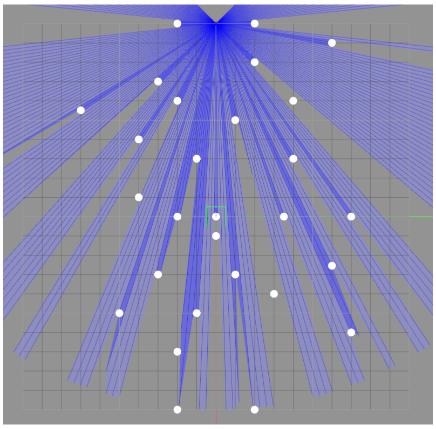

(b) Scenario 2: some columns are in the way.

Figure 8.1: Two different scenarios: a completely empty field and a field with obstacles sparsely spread around. Two start and goal posts are placed at the top and bottom.


(a) Scenario 3: many columns. Akin to a forest

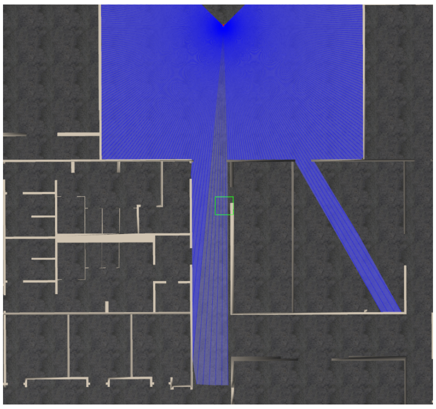

(b) Scenario 4: A hallway

Figure 8.2: Two different scenarios: a forest like environment and a indoor area with a hallway.

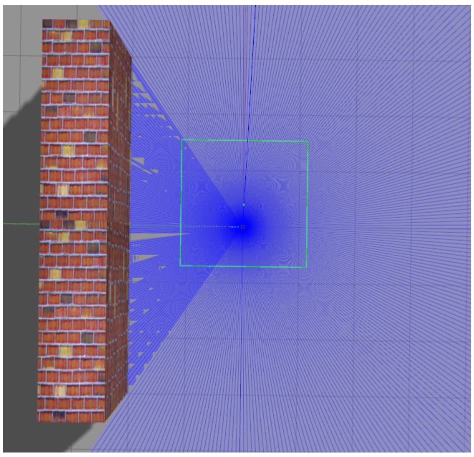
Experiment 2: Emergency Avoidance

When testing the improved emergency avoidance against the original emergency avoidance there is really only one thing to test: will the improved emergency avoidance avoid collisions in scenarios where the original emergency avoidance would cause them? First we will test both versions in normal scenarios before moving on to scenarios where the original emergency avoidance will struggle. The results for these experiments will be given by side-by-side comparison of the two versions. We will additionally check different K values.

Method

The UAV was placed hovering at some meters height, then, through the magic of simulation, time was stopped, the obstacles were put in place, and time was again unfrozen. The results show the velocities the algorithms proposed.

Results


(a) View of the Gazebo test setup

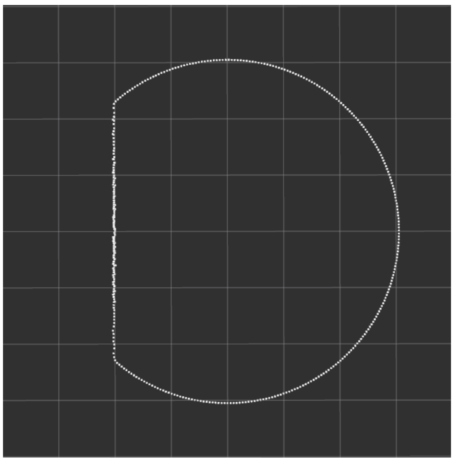
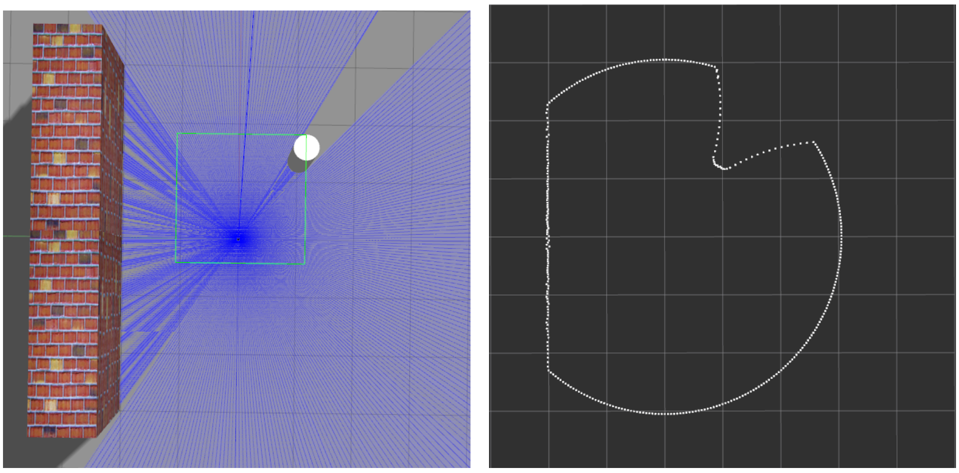

(b) The same setup as seen i Rviz. The white circle is bounded by the radius r_{LB}.

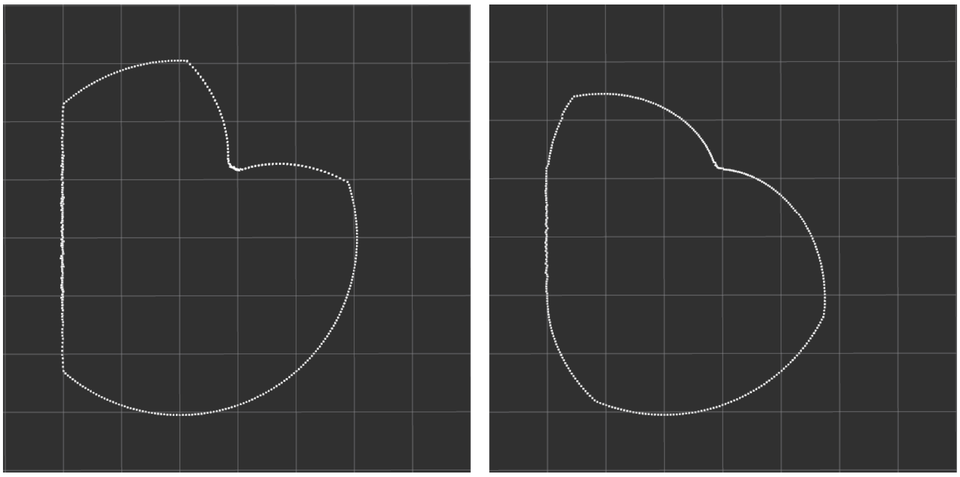
Figure 8.3: Here we can see the emergency avoidance setup, with the Gazebo world to the left and the visualisation to the right.



(a) Simple setup, similar to that shown in the explanation of CEPA

(b) Here the $K = 0.1$

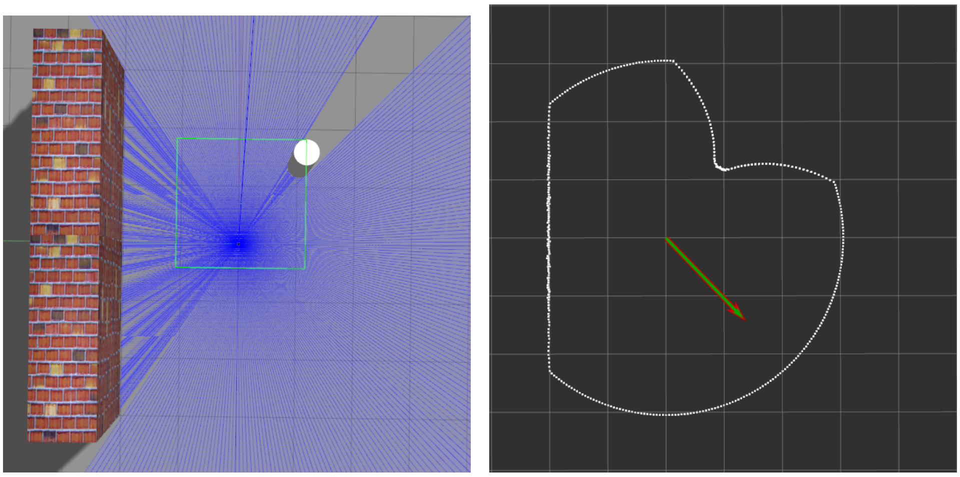
Figure 8.4: To check that our algorithm works we test the emergency avoidance with different Ks



(a) Here the $K = 0.04$

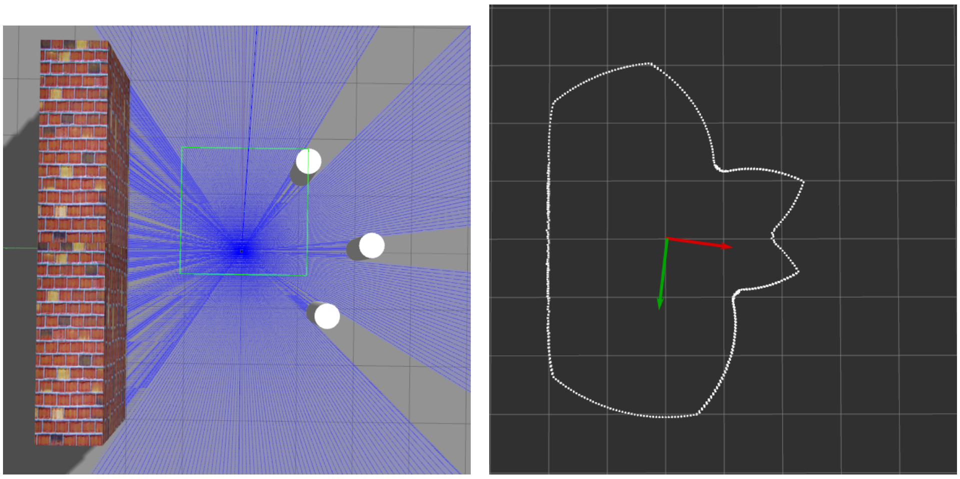
(b) Here the $K = 0.02$

Figure 8.5: Here we see the same set up but with lower Ks. With low enough Ks the figure starts to loose its shape.



(a) Wall on the left side and one column on the right. **(b)** The red arrow is the original, and the green is the improved.

Figure 8.6: Here we see the difference between the original and the improved emergency avoidance. As the original does not cause a collision in this scenario the arrows are identical.



(a) Wall on the left side and three columns on the right. Note that there is still room for the UAV to safely leave both at the top and the bottom. **(b)** The red arrow is the original, and the green is the improved.

Figure 8.7: Here we see the difference between the original and the improved emergency avoidance. The original emergency avoidance would have caused the UAV to crash into the columns, but the improved emergency avoidance finds a safe escape.

8.2 SLAM

In this section we will take a look at the SLAM algorithm provided by `gmapping`. We will be testing the algorithm in different environment, and with different lidars. If no specified elsewhere the the lidar in question is a Hokuyo UST. For the full list of lidars and their properties see Appendix D. A quick summary of the lidars is given below. The results are given as the maps produced by the different lidar in the two different scenarios. Unless some parts of the results needs special attention, all discussion of the results is given in the next chapter.

	URG-04LX-UG01	UST-10LX	UTM-30LX-EW	Puck LITE	TiM-751	M8-1
Shorthand	URG	UST	UTM	Puck	TiM	M8
FOV [deg]	240	270	270	360	270	360
Range [m]	5,6	10	30	100	25	150
Angular resolution [deg]	0,35	0.25	0.25	0.2	0.33	0.2
Frequency [Hz]	10	40	40	20	15	30

Table 8.2: Table giving some of the key characteristics of the different lidars we are going to using the experiments. For a full overview see Appendix D.

Experiment 3: SLAM in a normal environment

First it is nice to test the SLAM algorithm in a relatively normal environment to gauge the performance there, so that we have something to compare against when we repeat the experiment in a more specialised environment.

Method

We place the UAV in a clustered environment. There are some walls, some obstacles, all scattered around in no particular fashion. The maps are the result of flying around for 4 minutes.

Results

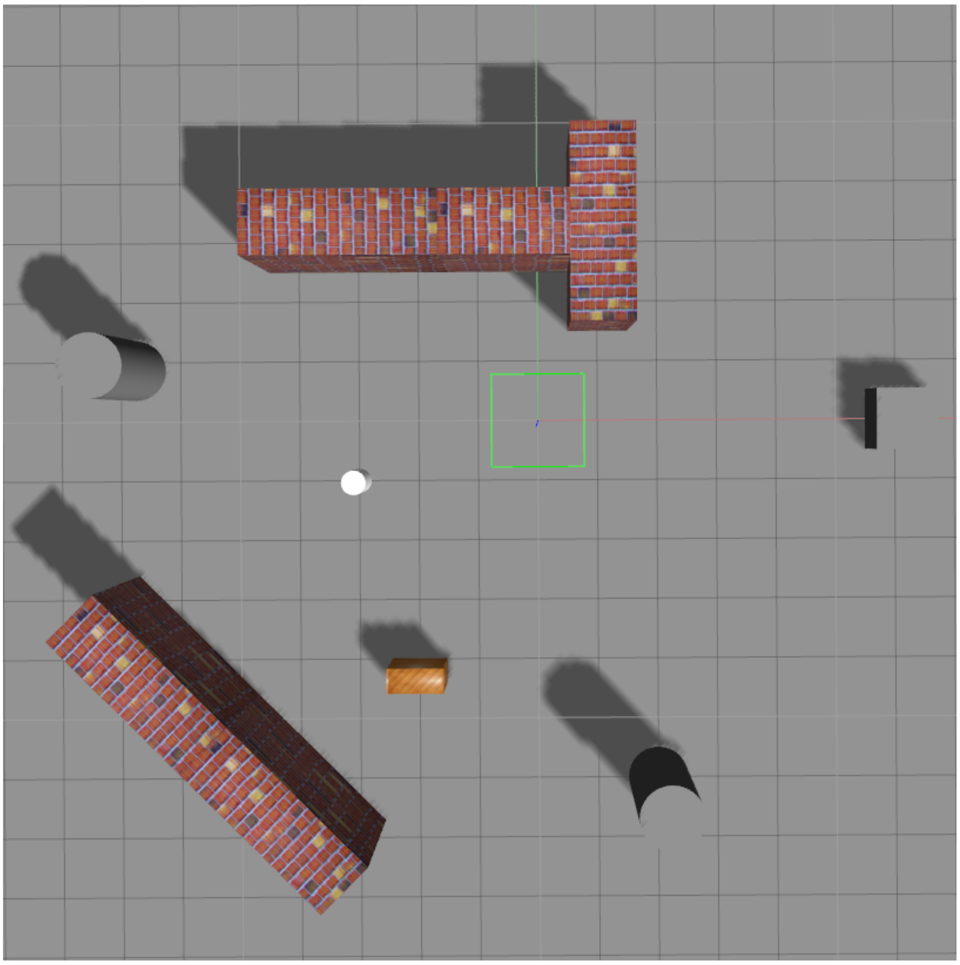
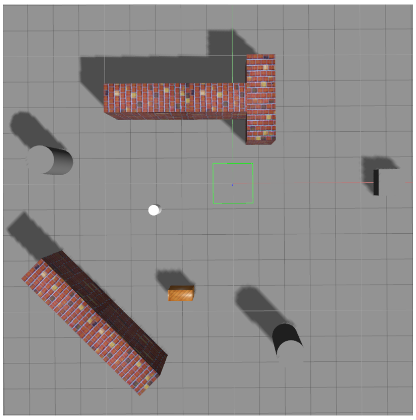


Figure 8.8: Environment created to test SLAM. Note the brown ”box” is a bookshelf and shows up on the maps as more of a [-shape than a box


(a) The world as seen from above

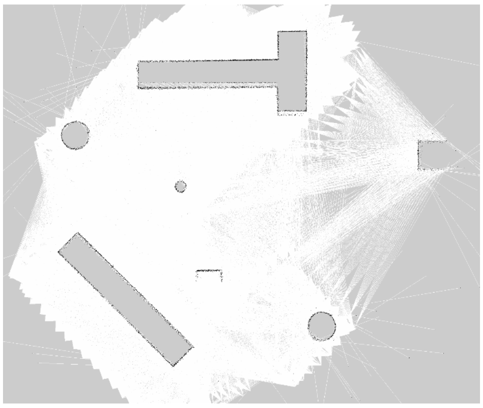
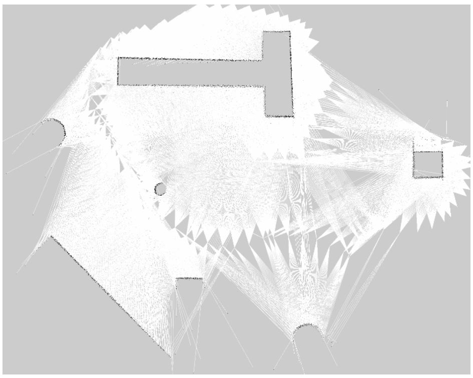

(b) The map created using SLAM with the hokuyo lidar

Figure 8.9: The simulation world and the map created by SLAM side by side. Lidar: Hokuyo UST-10LX


(a) The map created using SLAM with Hokuyo URG-04LX-UG01

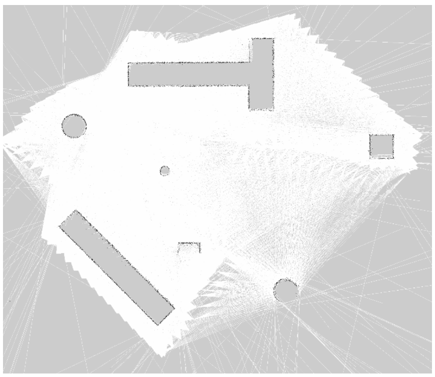
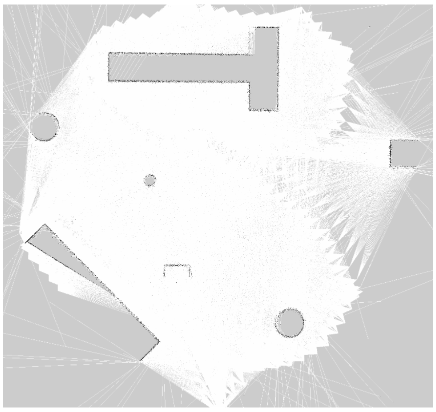

(b) The map created using SLAM with Hokuyo UTM-30LX-EW

Figure 8.10: The maps created by URG-04LX-UG01 and UTM-30LX-EW side by side.


(a) Map being created using the SICK lidar. Note the wall on the bottom left, where the odometry has drifted.

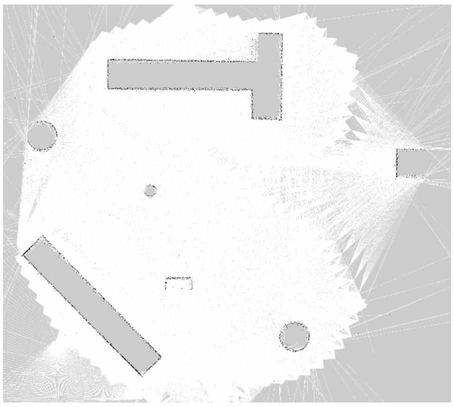
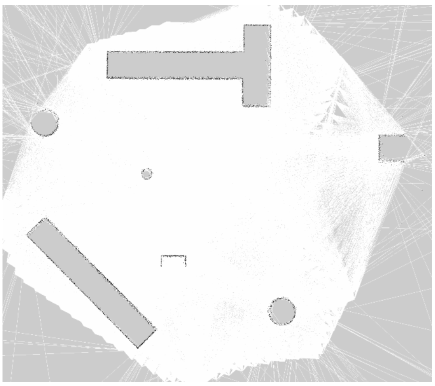

(b) Map fully created. Note how the wall down to the left has been corrected due to loop-closure

Figure 8.11: Figure showing how fast odometry can drift. even after some seconds around a corner. Lidar: SICK TiM571


(a) The map created using SLAM with the Velodyne lidar

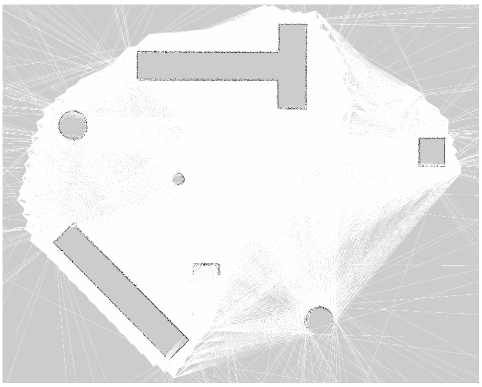

(b) The map created using SLAM with the Quanergy lidar

Figure 8.12: The maps created by the Velodyne Puck LITE and Quanergy M8-1 side by side.

Experiment 4: SLAM in a cargo hold like environment

The end goal is to use the UAV to inspect cargo hold, and oil and gas tanks, it is therefore important to test the SLAM algorithm in the UAV's intended environment. This will be a challenge due to the repetitive nature of the walls and the size of the area.

Method

We place the UAV in an environment which emulates a cargo hold. The walls have repetitive patterns, and he are is large. The results are after flying a single round-trip along the walls of the room.

Results

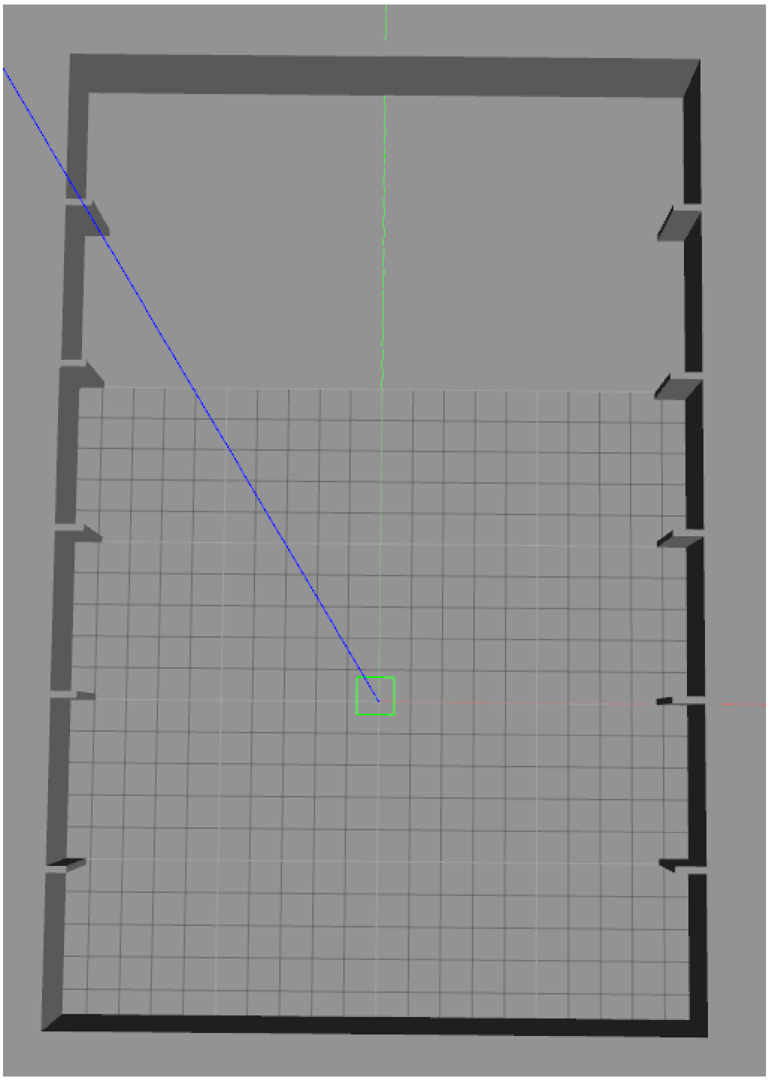
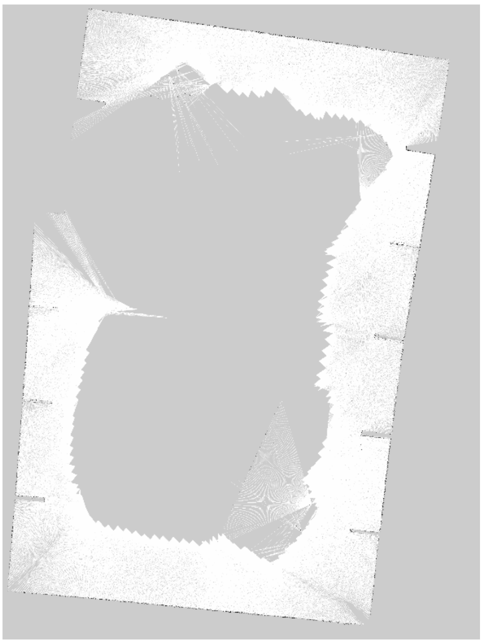


Figure 8.13: Cargo hold environment. The dimensions are 20x30 meters. The ”beams” on the sides are 1 meter wide


(a) Map of cargo hold being created. Note the misalignment and bent walls.

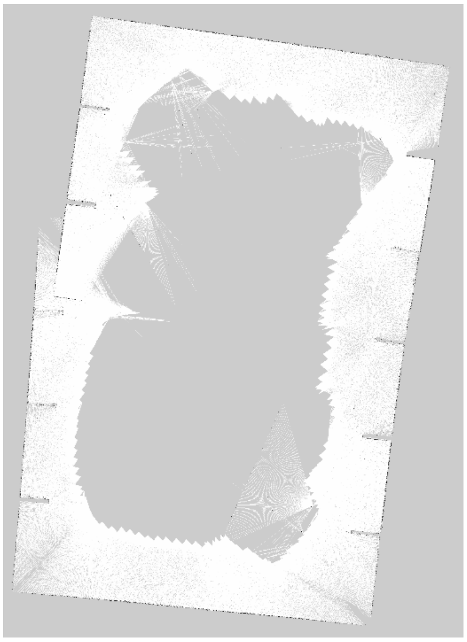
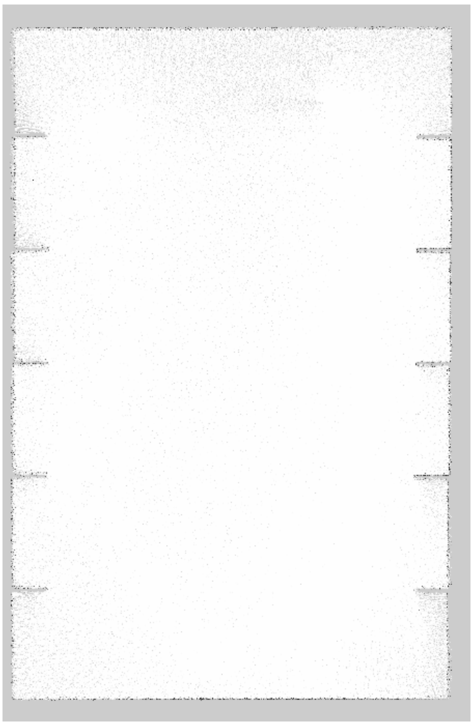

(b) Map of cargo hold after one complete lap. Again note the misalignemnt of the walls.

Figure 8.14: Map of the cargo hold. This misalignment was to great for the loop closure to fix. Lidar: Hokuyo URG-04LX-UG01

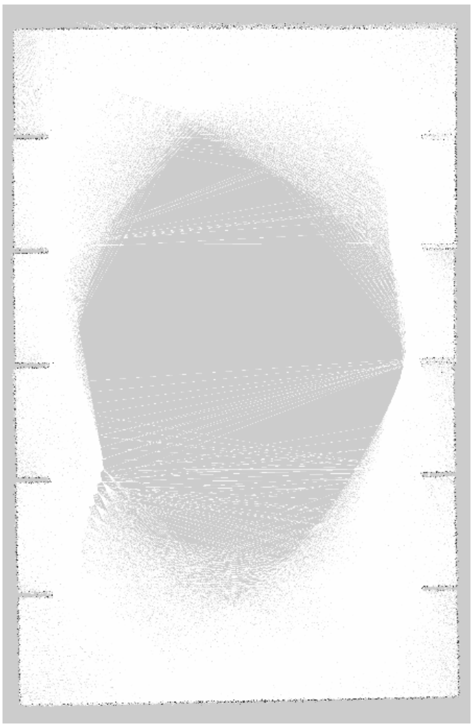
(a) Map of cargo hold being created. Note the misalignment on the left wall and the slightly bent walls.

(b) Map of cargo hold after one complete lap. The misalignment from before is fixed by loop closure.

Figure 8.15: Map of the cargo hold. The misalignment was fixed by the loop closure. Lidar: Hokuyo UST-10LX

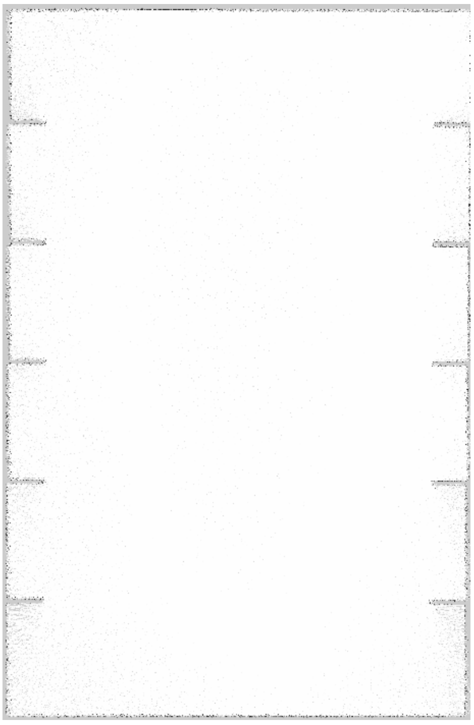


(a) Map of cargo hold after one complete lap by the UTM lidar



(b) Map of cargo hold after one complete lap by the TiM lidar. Note the slight bend in the side walls.

Figure 8.16: The maps created by UTM-30LX-EW and TiM571 side by side. The whiteness of UTM's map is due to its range which reach across the room. TiM managed to see the beams on the other side, while the walls where to far away, creating the few cross room lines one can see on the map.


(a) Map of cargo hold after one complete lap by the Puck lidar.

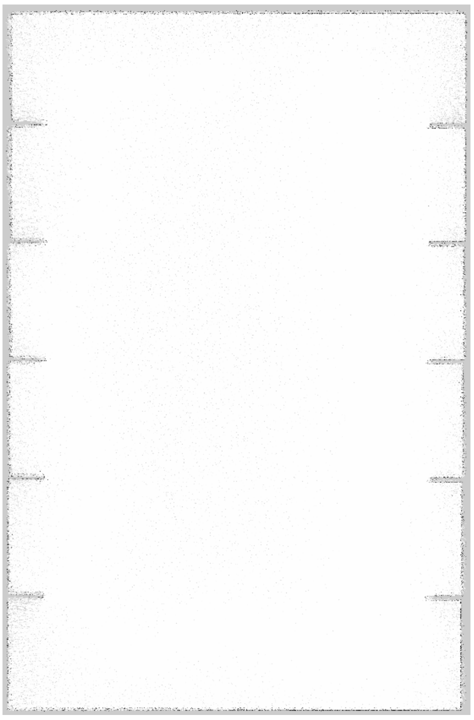

(b) Map of cargo hold after one complete lap by the M8 lidar.

Figure 8.17: The maps created by Velodyne Puck LITE and Quarnegy M8-1 side by side. The maps are virtually indistinguishable.

8.3 Simulator performance

In this part of the chapter we will take a look at some information from the simulations themselves. This is done to to some extent quantify the effect of the simulator framework. These numbers are collected from the other experiments as well as from development. The results presented in this section should be taken with a grain of salt, and are only there to indiacte trends.

Measurements 1: Compilation Time

Compilation time is the time it takes from the make command is given until all files are ready to go. Three different times are measured. Full compilation when one file is altered, full compilation when many files are altered, compilation of one ROS package.

Results

	Full, one file	Full, many files	single package
Time [s]	~ 5	~ 15	scales with files

Table 8.3: A table showing the time it takes to compile

Measurements 2: Launch Time

Launch time is the time from the launch file is launched until all nodes are good to go. The non-visual nodes, like the CEPA node or a lidar filter launches so much faster than Gazebo, PX4 and Rviz that they do not impact the results, and are therefore omitted. Shut down time is also included as Gazebo and PX4 use much more time to close properly than to launch properly.

Results

	Gazebo	Gazebo with PX4	Gazebo with Rviz
Startup [s]	5	~ 7	~ 10
Shutdown [s]	10	~ 15	~ 10

Table 8.4: A table showing the time it takes to launch the collection of nodes

Chapter 9

Discussion

This discussion will be split into two part. First the results from the experiments will be discussed. This includes the performance of CEPA, SLAM, and the simulator as a whole. Secondly the ease of using the simulation framework will be discussed. Here we will judge the framework based on its simpleness, flexibility, and features.

9.1 Experiments

9.1.1 CEPA

The emergency avoidance did not crash when confronted with symmetrical environments. The runtime for the CEPA algorithm was significantly lower. Flying in open terrain gave an increase of xxx. In more clustered environments, the increase was not that big, but still there. The adjustable speed weight made sure the UAV had time to break before reaching an obstacle. Keeping the lidar levelled did help a bit/not noticeable SLAM

Steering Algorithm

The improvements in the steering algorithm were quite clear. Looking at table 8.1 we see a huge reduction in computation time across the board!

Looking closer we can assume that the ”close to zero” times produced by the improved algorithm in Scenario 1 and 2 were due to testing v_{ref} first, and passing it on if it was clear. This reduces the (with a Hokuyo lidar, testing a each measurement) 1440 tests to 1!

When the environment gets more cluttered, the time it takes for the improved algorithm to find the best velocity increases as expected, but is still many times better than the original algorithm. Splitting the circle into different sectors where only one velocity from each sector needs to be tested is assumed to be the main reason for this time save.

The computation times for the original algorithm is quite stable as one can expect as it tests every angle no matter what. The slight time saved in Scenario 1 explained by the fact that the algorithm does not need to adjust the magnitudes of the velocities to avoid obstacles. The original is still fast enough for a lidar at 40Hz, but the increased performance gained from updating to the improved algorithm means there is no reason to still use the original.

Emergency Avoidance

Looking at the results from Experiment 2 and Figure 8.7, we can see how the improved emergency avoidance avoided collisions where the old emergency avoidance would have caused them.

There is still development to be done in the emergency avoidance. Primarily concerning the dynamics of the UAV. As the emergency avoidance does not to-date take into account the velocity of the UAV when deciding on a direction to avoid emergency. Not incorporating the velocity could output in-feasible velocity commands, where the UAV simply has to much inertia to be able to complete the command in time.

Summary

The improved CEPA algorithm is much, much faster than its predecessor. The time gained by the improvements can come in handy when incorperating other more time consuming algorithms, which always can use a bit more CPU time.

And of course, the algorithm will no longer cause collisions, which is quite a big plus given that it is a collision avoidance algorithm.

9.1.2 SLAM

The SLAM algorithm performed very well in general. Every map created in both environments where recognisable when compared to a picture of the world. The experiments also showed what aspects one should take a closer look at before testing in the real world.

Choice of SLAM algorithm

By using `gmapping` we used a particle filter based graphSLAM. This choice seems to be good. Looking at Experiment 3 and 4 generally we find the results to be well crafted maps, quite accurately representing the environment. The algorithm was not without it flaws however, looking at Figure 8.11 we see that the algorithm quickly drifts if only has one reference point. In this case the backside of the wall is separates the UAV from the rest of the map and it continues to drift. The loop closure in `gmapping` is however good enough to correct this when the UAV reappears on the other side of the wall. We can see this again in Figure 8.15, where the loop closure corrects the map. But looking at Figure 8.14, we see that the loop closure mechanism is not all-powerful as the wall has drifted so much it is no longer recognised as the same wall.

Often we have some idea of how the area looks before we are going to fly in it. If so, we wan give the SLAM-algorithm a starting map which is will continue to build upon. This could solve some of our drifting problems. In addition methods from computer vision could be used to recognise patterns in the maps which could be used to fill in blank areas or predict unexplored areas. For example, in our cargo hold there are a lot of straight walls.

The SLAM algorithm has no way of recognising this pattern, or using this for its benefits, but pattern recognition software would have no problem finding the lines that creates the wall; the line found this way could again be place into the map.

Choice of lidar

In both Experiment 3 and 4 we can see the difference different lidars can have. Choosing the correct lidar for a set mission can be hard. In this section we are going to look at the results form Experiments 3 and 4 to see how the different lidars affect the result of the SLAM-algorithm.

The frequency of lidar measurements does not seem to affect the end result to any significant degree. This can simply be explained by the fact the the SLAM algorithm takes far longer to complete at each iteration than the scan time of any of the lidars. This can most easily be seen in Figure 8.10a, where every hourglass/bow-tie shaped along the middle/right part of the figure is an iteration of the SLAM algorithm. Frequency might help in collision avoidance or other application, but does not seem to factor into the quality of maps produced.

The range of the lidars on the other hand has a huge impact. This is most clearly seen in Experiment 4. In Experiment 3 all lidars do a pretty good job. The URG-04LX-UG01 has a slightly less complete map only due to the fact that the experiment was run on a time limit and due to its shorter range (5m) the lidar had to spend more time mapping the same objects compared to the other lidars. But, as mentioned, we can see the difference range makes most clearly in Experiment 4. First we can see in Figures 8.16a, 8.17a, and 8.17b, that all the three lidars with range greater than the length and width of the cargo hold created maps pretty much equal to both each other and the actual world. Reducing the range a bit and we get more interesting results. Looking at the map created by the TiM lidar (Figure 8.16b), which has a range of 15m, we can see that even though the map is good, the wall on the long-side bulges inwards a bit. As the range of the lidar is more than half of the width of the cargo hold however, this can be corrected by flying in the middle of the room if the resulting map after a round-trip is to inaccurate. Reducing the range even further we find that (in Figures 8.14 and 8.15) for both the URG-04LX-UG01 and the UST-10LX the maps drift quite a bit, not creating the rectangular shape we would like to get. For the URG the drift is so bad that the loop-closure in the SLAM algorithm could not correct it. Correction for the UST was possible, but if the room was bigger there is no reason to believe that correction would not be impossible for the UST as well, or the other lidars for that matter.

Summary

The `gmapping` worked quite well and was really easy to setup. Although the algorithm sometimes drifted when there was little information to be gained from the lidar, the loop-closure corrected for the drift most of the time. As the problem disappeared when using lidars with greater range, the easy way to solve this seams to use a lidar with great enough range. The price of the lidars does increase with the range, and so does the weight. From the cheapest to the most expensive lidar tested here, the URG-04LX-UG01 and the Puck LITE respectively the price ranges from $1’100 to $8’000. And even if price did not matter, weight most definitely does. Increasing the weight of the payload sends cascades though the UAV hardware, the increased weight needs bigger motors, which in-turn demands bigger batteries, and so on. It it therefore important to choose the best lidar for the mission even if it is not strictly speaking the best lidar. A well rounded candidate seems to be the UTM-30LX-EW; with a range of 30m, weight of 210g, and price about $5'300, UTM has great specs for its cost and weight, and is much used in the field (Mader et al., 2015; Chen et al., 2016; Schäfer et al., 2016; Serrano, 2011).

9.2 Simulation Framework

9.2.1 Structure and Understandability

Receiving a zip-file full of code from another person is always exciting and a little scary. Sometimes everything make sense and every piece of code is where one would expect it to be, other time one is not so lucky. When developing software for ROS the file hierarchy is set. When receiving a ROS package the names of the folders and their placement is the same as any other ROS package. Though this is in part in place to help the compiler, the hierarchy helps humans too. The time it takes to get an overview of the software is drastically reduced. Each package also includes an XML-file with a description of the package, its dependencies, and other meta data. As all packages developed for the framework follows the ROS structure, their structure is easy to follow.

Some complications occur when using Gazebo with Rviz as the SDF format of Gazebo is not directly compatible with Rviz or TF. This can be solved by using URDF, ROS's model description format, or by developing custom nodes that parses the SDF-files into TF data. In the simulator tool package a node for publishing the frames and transforms was developed. The node did not publish all the frames existing in the model, but included all the necessary frames to run collision avoidance and SLAM. ROS 2.0 will address this problem to ensure SDF can be imported directly into ROS. One can simply use URDF for development as URDF can be used in Gazebo without any problems, though having a consistent file format is preferred and most Gazebo models are developed as SDF.

9.2.2 Re-usability and Portability

We will now look further into how software created in and for this framework can be reused. This includes both using the software again as-is and reconfiguring it.

Software written for ROS is naturally modular, and quite fine grained. Each package is often self sustained, only relying on standard libraries. Using the XML-file in the package, all information about needed packages is easily obtained, and the needed packages can be downloaded. This is also valid for custom packages, but as ROS typically releases a new version each year, packages will need some maintenance, and custom packages are often lacking in that respect. It is therefore recommended to have as little cross dependencies between custom packages as possible.

The development tools packages followed this principle. Each package is independent of each other, with the exception of the `master_thesis` package, which uses all the custom packages we have developed. As discussed in chapter chapter 5 this is done to increase the usability of the developed packages. The development tool packages does not get cluttered with master related (though they are strictly speaking a part of this master) files which does not have any use outside this master. The launch files for the different experiments conducted are all in the master packages, the other packages blissfully unaware what nodes they communicate with or what packages the nodes comes from. This separation increases the usability of the development tools, and makes it easier to port into another project.

The development tools are not only portable themselves, but also increases the portability of other software written. When developing a lot of code in the same package, it is very easy to fall into bad habits of letting multiple nodes depend on each other and using custom messages, simply because it makes development easier there and then. The end result is however a messy structure which is hard to understand and even harder to develop further. By having an external interface, here the development tools, it forces the development to follow the standard set by ROS. Following these standards makes the software compatible with any other ROS software and therefore increases the portability.

We want the models, worlds, and other software we develop for Gazebo to be easily portable and shareable. In the following text only models will be mentioned, but the problems and solutions are equivalent for worlds, plugins and other Gazebo software. The official Gazebo tutorials for custom models, only covers how to place models into a specified directory deep within the Gazebo hierarchy. The simulator tools package creates an environment outside the native Gazebo environment where models can be placed. This has allowed all the custom models needed for the simulation to be stored in that package, which is easily obtainable and usable.

9.2.3 Developing and Experimenting

In this subsection we will take a look at the change in time distribution when developing and testing software within the simulator framework. One of the main goals of the simulator framework is to reduce the time spent waiting, or developing software needed to test your project. Several tools to reduce wasted time is already incorporated in ROS, and in creating the simulator framework others where developed.

The tools ROS is equipped with to reduce time sinks are described in detail in section subsection 4.2.6. From Measurements 1, we can see that the compiling takes 5-15s on average. But given the times we need to adjust variables, the times add up. By using parameters in launch files and from the parameter server this time is reduced to zero, assuming changing a variable in the source file takes as long as changing the variable in a launch file. Seeing how often small adjustments needs to made when experimenting this has the potential to save a lot of time.

When conducting experiments on CEPA there were a lot of parameters to be tuned. Even changing the parameters in launch files or in the parameter server would take some time, due to the fact that we would need to relaunch the simulator and all associated nodes. From Measurements 2 we can see that an the average time to shut down and relaunch Gazebo when testing with CEPA (with PX4) was about 20 seconds. Developing CEPA with dynamic reconfigurable parameters made sure tuning of the algorithm could happen in the same session of running Gazebo. Testing and retesting with new parameters as you went along. 20 seconds might not seem like much, but when fine-tuning a regulator the amount of small adjustments that need to be done are staggering, and the wasted time quickly surpasses the time it takes to implement dynamically reconfigurable parameters in the node.

9.2.4 Summary

The simulation framework have showed its strength during the development of the navigation tools. Having an already strong base in the environment of ROS and the Gazebo simulator, the development tools really helped with the development of the navigation tools. The framework allowed the author to be fully focused on the development of the navigation tools, rather than extra support functions and structures.

There are still many tools that can be developed to further increase the ease and speed of development. Some generic tools like a universal SDF to TF parser. Some tools more specialised for inspection, like an on/off button to have a gimballed lidar. As the simulator framework continues to be developed more tools will be added to help increase the effectiveness and usefulness of the framework.

When executing experiments the framework helped quickly setup the simulation world needed and adjust the settings for the experiment in question. The simulation tools developed to publish to TF were a great help and really eased the testing of SLAM especially. The built in tools from ROS also helped speed up the conduction of experiments. Especially launch files for quickly switching between simulation setups and dynamic reconfigure while running one experiment.

Chapter 10

Conclusion and Future Work

Based on the ease of conducting the different experiments in Chapter 8, we can conclude that the simulation framework significantly helps when executing many experiments with small differences. The launch files made sure each experiment could be loaded without changing any source code, and the dynamic reconfigurable variables made tuning the different parameters fast and easy within each experiment. Visualising the sensor data in Rviz, made debugging weird behaviour easier, and gave a great insight into the world as seen from the UAV. Especially after the TF-publisher node was developed, one could for example easily see how accurate the map created by SLAM was.

Since all software developed is encapsulated in executable nodes communicating over topics, everything is compatible with any other nodes developed by other people, and will be compatible with nodes created by others in the future. This portability should not be overlooked. In addition, if a pure C++ implementation is needed, the only changes needed is to change the ROS messages with structs containing the same member variables.

Gazebo has proven to be a very fitting simulator for our purposes having a simple interface with the flight controller firmware and ROS. When tweaked to use custom made models and world, any world or experiment designed in Gazebo became very portable; making sure others can conduct the same experiments or test their own implementations in the same world. Resulting from the standardised interfaces between the different software packages, mainly ROS, Gazebo, and PX4, switching firmware from PX4 to ArduPilot or simulator from Gazebo to AirSim can be done with relative ease. Nodes interfacing the firmware or simulator might need to switch topic names or names of services, but all other nodes will remain unchanged.

After the improvements suggested in Chapter 6 was implemented, the performance of CEPA increased sharply, as shown in the experiments in Chapter 8. Critically, removing the flaw from the emergency avoidance algorithm to ensure it did not cause collisions, and starting the check for the optimal velocity at the most likely angle, made CEPA a much more robust and efficient collision avoidance algorithm. There are still some rough edges that need to be smoothed out and development to expand the scope of the algorithm should continue, but the results from the experiments are very promising.

The SLAM implementation worked well during the experiments, but showed its limits when the surroundings where symmetrical over a long period of time, and the UAV could only see one wall. Using a lidar with larger range did solve the problems with symmetric rooms, but at the cost of extra weight (and increased cost) on the UAV which reduces the flight time which in turn reduces the work the UAV is able to do during one run.

10.1 Future Work

Working towards autonomous inspection there are a lot paths to take. One of the major steps that should be taken is to test the implementations in the real world. This would not only make sure the implementations of collision avoidance and SLAM works outside a simulated environment, but would also validate the simulation. One should also keep an eye out for AirSim as it is a very promising simulator, and if considering visual SLAM, or control based on cameras, it is the obvious choice.

Regarding autonomous inspection there are several possibilities. To move away from human assisted inspection and toward fully autonomous inspection a trajectory generation and path planning is needed. In conjunction with SLAM it can be used to create a complete map of the surrounding by letting the SLAM take an active role in where to explore next, such processes are described in Thrun et al. (2005). This will work well with mosaicing techniques such as those explained in Garcia-Fidalgo et al. (2016), not only creating a complete 3d map of the environment, but also superimposing on that map visual information gathered by cameras.

For CEPA there are some improvements that could be implemented to ensure an even more robust collision avoidance and increase the usability of the algorithm. One such thing would be to adjust further for vehicle dynamics, taking into account the tilt of the multirotor when it accelerate, or adjusting the weighting parameters based on the inertia of the multirotor. Using a 3d lidar such as the Velodyne Puck LITE (see Appendix D), the algorithm could also be expanded to 3d. Expanding is to pseudo-3d could also be done through incorporating other sensors such as sonars or radars, these would give a rough idea about the areas above or below the UAV, meaning the UAV could confidently move vertically and then receive the more detailed information about the surroundings from the lidar.

Bibliography

Adams, S. D., 2016. Posidonia 2016: Dnv gl performs first drone production survey.
URL `https://www.dnvgl.com/news/posidonia-2016-dnv-gl-performs-first-drone-production-survey-66660`

Arguedas, M., March 2017. Ros 2 roadmap.
URL `https://github.com/ros2/ros2/wiki/Roadmap`

Blasco, P. I., 2012. Ros distributed architecture. Retrieved from: `https://www.slideshare.net/pibgeus/21-distributed-architecture-deploymentinstrospection`.

Bonnin-Pascual, F., Ortiz, A., 2016. A generic framework for defect detection on vessel structures based on image saliency. 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA), 1–4.
URL `http://ieeexplore.ieee.org/document/7733668/`

Bosse, M., Zlot, R., Flick, P., 2012. Zebedee : Design of a Spring-Mounted 3-D Range Sensor with Application to Mobile Mapping. IEEE Transactions on Robotics 28 (5), 1104–1119.
URL `http://ieeexplore.ieee.org`

Chen, J., Wu, J., Chen, G., Dong, W., Sheng, X., 2016. Design and Development of a Multi-rotor Unmanned Aerial Vehicle System for Bridge Inspection. ICIRA 1, 498–510.

Conrose, C., Watson, A., Ulrich, J., 2013. Pidar: 3D Laser Range Finder. Tech. rep., University of Central Florida, Florida.

Eich, M., Vögele, T., 2011. Design and control of a lightweight magnetic climbing robot for vessel inspection. 2011 19th Mediterranean Conference on Control and Automation, MED 2011, 1200–1205.

Eschmann, C., Kuo, C.-M., Boller, C., 2012. Unmanned Aircraft Systems for Remote Building Inspection and Monitoring. Proceedings of the 6th European Workshop on Structural Health Monitoring, July 3-6, 2012, Dresden, Germany 2, 1–8.

Fetch Robotics, 2016. Tutorial: Gazebo simulation. Retrieved from: `http://docs.fetchrobotics.com/gazebo.html#visualizing-with-rviz`.

Garcia-Fidalgo, E., Ortiz, A., Bonnin-Pascual, F., Company, J. P., 2016. Fast image mosaicing using incremental bags of binary words. Proceedings - IEEE International Conference on Robotics and Automation 2016-June, 1174–1180.

Gazebo, August 2017. Digital tutorials. Retrieved from: `http://gazebosim.org/tutorials`.

INCASS, 2013. Description of work. FP7-TRANSPORT-2013-MOVE-1; SST.2013.4-2. Inspection capabilities for enhanced ship safety (GA 605200).
URL `http://www.incass.eu/`

Jackson, J., Wheeler, D., Mclain, T., 2016. Cushioned Extended-Periphery Avoidance : a Reactive Obstacle Avoidance Plugin. 2016 International Conference on Unmanned Aircraft Systems (ICUAS) June 7-10, 2016. Arlington, VA USA.

Koch, T., Natarajan, S., Bernhard, F., Ortiz, A., Bonnin-Pascual, F., Garcia-Fidalgo, E., Company, J. P., 2016. Advances in Automated Ship Structure Inspection. International Conference on Computer and IT Applications in the Maritime Industries (COMPIT), 84–98.

Kummerle, R., Steder, B., Dornhege, C., Ruhnke, M., Grisetti, G., Stachniss, C., Kleiner, A., 2009. On Measuring the Accuracy of SLAM Algorithms.
URL `http://www2.informatik.uni-freiburg.de/~stachnis/pdf/kuemmerle09auro.pdf`

Mader, D., Blaskow, R., Westfeld, P., Maas, H. G., 2015. UAV-Based acquisition of 3D point cloud - A comparison of a low-cost laser scanner and SFM-tools. International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences - ISPRS Archives 40 (3W3), 335–341.

Mader, D., Blaskow, R., Westfeld, P., Weller, C., 2016. POTENTIAL OF UAV-BASED LASER SCANNER AND MULTISPECTRAL CAMERA DATA IN BUILDING INSPECTION. The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences XLI (July), 12–19.

Meyer, J., Kohlbrecher, S., 2014. The hector quadrotor.
URL `http://wiki.ros.org/hector_quadrotor`

Meyer, J., Sendobry, A., Kohlbrecher, S., Klingauf, U., Von Stryk, O., 2012. Comprehensive simulation of quadrotor UAVs using ROS and Gazebo. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Vol. 7628 LNAI. pp. 400–411.

Microsoft AirSim, 2017. Airsim: docs/images. Retrieved from: `https://github.com/Microsoft/AirSim/tree/master/docs/images`.

MINOAS, 2009. Description of work. FP7-SST-RTD-1; SST.2008.5.2.1. Marine INspection rObotic Assistant System (GA 233715).
URL `http://www.minoasproject.eu/`

Nex, F., Remondino, F., 2014. UAV for 3D mapping applications: A review. Applied Geomatics 6 (1), 1–15.

Ortiz, A., Bonnin, F., Gibbins, A., Apostolopoulou, P., Eich, M., Spadoni, F., Caccia, M., Drikos, L., 2010. First steps towards a roboticized visual inspection system for vessels. Proceedings of the 15th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2010.

PX4 Autopilot, 2016. jmavsim. Retrieved from: `https://pixhawk.org/dev/hil/jmavsim`.

PX4 Autopilot Project, April 2016. jmavsim simulationkit simulator running px4 [video file]. Retrieved from: `https://youtu.be/VigPu16NaAk`.

Předota, J., 2016. LiDAR based obstacle detection and collision avoidance in an outdoor environment. Bachelor, Czech Technical University in Prague.

Quigley, M., Berger, E., Ng, A. Y., 2007. Stair: Hardware and software architecture. AAAI 2007, Robotics Workshop, 31–37.
URL `http://www.aaai.org/Papers/Workshops/2007/WS-07-15/WS07-15-008.pdf`

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., Mg, A., 2009. ROS: an open-source Robot Operating System. Icra 3 (Figure 1), 5.
URL `http://pub1.willowgarage.com/{~}konolige/cs225B/docs/quigley-icra2009-ros.pdf`

RoboSavvy, November 2015. Rosserial.
URL `http://wiki.ros.org/rosserial`

Roca, D., Martínez-Sánchez, J., Lagüela, S., Arias, P., 2016. Novel Aerial 3D Mapping System Based on UAV Platforms and 2D Laser Scanners. Journal of Sensors 2016.

Saito, I., July 2015. tf.
URL `http://wiki.ros.org/tf`

Schäfer, B. E., Picchi, D., Engelhardt, T., Abel, D., 2016. Multicopter Unmanned Aerial Vehicle for Automated Inspection of Wind Turbines. In: 24th Mediterranean Conference on Control and Automation (MED) June 21-24, 2016, Athens, Greece. pp. 244–249.

Scherer, S., Singh, S., Chamberlain, L., Saripalli, S., 2007. Flying fast and low among obstacles. Proceedings - IEEE International Conference on Robotics and Automation, 2023–2029.

Serrano, N. E., 2011. Autonomous Quadrotor Unmanned Aerial Vehicle for Culvert Inspection. Master, Massachusetts Institute of Technology.

Shah, S., Dey, D., Lovett, C., Kapoor, A., 2017a. Aerial Informatics and Robotics platform. Tech. Rep. MSR-TR-2017-9, Microsoft Research.

Shah, S., Dey, D., Lovett, C., Kapoor, A., 2017b. Airsim: High-fidelity visual and physical simulation for autonomous vehicles.

Stachniss, C., 2014. Pre-2014 robotics 2d-laser datasets. Retrieved from: `http://www.ipb.uni-bonn.de/datasets/`.

Thrun, S., Burgard, W., Fox, D., 2005. Probabilistic Robotics (Intelligent Robotics and Autonomous Agents). The MIT Press.

Wulf, O., Wagner, B., 2003. Fast 3D scanning methods for laser measurement systems. Proceedings of the International Conference on Control Systems and Computer Science, 312–317.

Ze, A., November 2016. Rqt plugins.
URL `http://wiki.ros.org/rqt/Plugins`

Zhao, L., Huang, S., Dissanayake, G., 2013. Linear slam. Retrieved from: `http://openslam.org/linearslam.html`.

Appendix A

System setup

This appendix goes through where the information needed to set up the framework as described in this thesis on your own computer exist. As well as how to recreate the experiments.

The software and experiments in this thesis was all conducted on a Asus laptop running Ubuntu16.04. The laptop had a i5-4200H, 2.8GHz CPU, 8GB RAM, and a Geforce GTX850M graphics card.

Starting with a machine running Ubuntu 16.04, install ROS following the instructions found on

`http://wiki.ros.org/ROS/Installation`

Choosing a full install will also install Gazebo. Then install PX4 from

`https://dev.px4.io/en/setup/dev_env.html`

The source files for the packages used in this thesis can be found on Github at

`https://github.com/volesi`

This is where the software packages are. As they will continue to be developed we will refer the reader to the github page to find the most updated software and instructions. The master thesis package will contain the information on how to set up the experiments as described in this thesis. The other packages will contain information on how to use them, but are developed to be stand alone tools, and will as such not contain any documentation directly connecting them to this thesis.

Appendix B

Tutorials and Guides

There are many guides and tutorials scattered around the internet. Some good, some not so much. This appendix contains some of the authors favourite guides and tutorials.

Starting with the different simulators.

jMavSim

To find information about jMavSim there are two different sites which offers general information:
`https://dev.px4.io/en/simulation/sitl.html`
and
`https://pixhawk.org/dev/hil/jmavsim`.
A tutorial to set up jMavSim on Ubuntu can be found at:
`https://uav-lab.org/2016/08/31/px4-research-log-8-hardware-in-the-loop-hitl-simulation-using-jmavsim-on-ubuntu-14-04/`

Microsoft AirSim

Microsoft AirSim is still to new to have any good tutorials, and as the API changes rapidly any tutorial might quickly be outdates. To get the newest information about the development of AirSim and how to use it see their official Github page:
`https://github.com/Microsoft/AirSim`

ROS and Gazebo

As ROS and Gazebo is used much together we will combine their sections. First, for general informaiton and guides the Gazebo and ROS websites does a good job of documenting their software. This can be found at
`http://wiki.ros.org/ROS/Tutorials`
and

`http://gazebosim.org/tutorials`.
For anyone looking to understand the basic principles of ROS a book named Gentle introduction to ROS by Jason M. O'Kane can be found free on the authors page:
`https://cse.sc.edu/~jokane/agitr/`.
The book offers all the information one need to understand ROS basics in a easy to understand fashion with lots of examples.

For a project based guide to simulation using ROS and Gazebo, including how to create models, incorporate sensors, and use the TF library, the website
`http://moorerobots.com/blog`
has some great videos.

If a more comprehensive understanding is needed the webpage
`http://www.theconstructsim.com/`
has some of the best guides, though they are locked behind a steep subscription fee of $10/week! They do however make really good youtube videos where they go through their guides and most information can be retrived for free from there.

The Erle Robotics has some great introduction videos to ROS, tough the quickly adjust the scope of their videos to always use their custom hardware. They are however the first place the, to authors knowledge, that offers guides to ROS2.0
`http://docs.erlerobotics.com/robot_operating_system`

PX4 and Ardupilot

For more information about the firmwares PX4 and Ardupilot check out their devguides at their respective webpages:
`https://dev.px4.io/en/`
and
`http://ardupilot.org/dev/`
These also include how to set up the firmware with Gazebo and ROS, and run the firmware as SIL.

SLAM

There are as mentioned many many versions of SLAM, and new papers with better solutions are steadily coming out. The guides given here are by no means cutting edge anymore (though they would have been only a couple of years ago, development happens fast), but give really good introduction to the SLAM problem and the different methods for solving the problems. First we have a book which is a comprehensive guide of all aspects of SLAM called Probabilistic Robotics Thrun et al. (2005). For those waning to learn as a lecture Cyrill Stachniss has two great courses on SLAM and photogrammetry on his Youtube page:
`https://www.youtube.com/channel/UCi1TC2fLRvgBQNe-T4dp8Eg/featured`
Lastly, if one want a close look at graph based SLAM a tutorial was written by the man behind Fast SLAM (the most used SLAM method at time of writing, and a graph based SLAM version) Giorgio Grisetti. The tutorial can be found on
`http://ieeexplore.ieee.org/document/5681215/`

Appendix C

Permissions and Licenses

List of all the licences pictures included in this thesis are shared under. Which licence is applied to which picture is noted in the caption of each picture. If no note, the picture was made by the author.

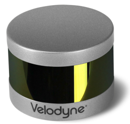
CC BY 3.0	`https://creativecommons.org/licenses/by/3.0/`
CC BY 4.0	`https://creativecommons.org/licenses/by/4.0/`
CC BY-NC-ND 4.0	`https://creativecommons.org/licenses/by-nc-nd/4.0/`
CC BY-NC-SA 4.0	`https://creativecommons.org/licenses/by-nc-sa/4.0/`
Apache 2.0	`http://www.apache.org/licenses/LICENSE-2.0.html`
IEEE	`http://www.ieee.org/publications_standards/publications/rights/permissions_faq.pdf`
MIT	`https://opensource.org/licenses/MIT`

Appendix D

Lidar overview

Table of lidar specification. Copied over from my project report.

Table D.1: Comparison of common lidars. The data is only for reference. See the lidars' data-sheet before use.

						
Name	URG-04LX-UG01	UST-10LX	UTM-30LX-EW	Puck LITE	TiM571	M8-1
Producer	Hokuyo	Hokuyo	Hokuyo	Velodyne	SICK	Quanergy
Environment	indoor	indoor	outdoor	outdoor	outdoor	outdoor
FOV [deg]	240	270	270	360hor 30ver	270	360hor 20ver
Scanning range [m]	0.02-5.6	0.06-10	0.1-30	100	0.05-25	150
Accuracy	±3cm	±4cm	±3cm	±3cm	±6cm	$<\pm5$cm
Scanning Frequency [hz]	10	40	40	5-20	15	5-30
Angular resolution [deg]	0.352	0.25	0.25	0.1-0.4 hor 2.0 ver	0.33	0.03-0.2
Power consumption	2.5W	3.6W	8W	8W	4W	15W
Weight	160g	130g	210g	590g	250g	800g
Multi echo	no	no	3	2: strongest or Last	N/A	3