Professional Bachelor Applied Information Science

UAVs autonomously navigating dynamic indoor environments

Vic Segers

Promoter: Tim Dupont

Bachelor paper academic year 2019-2020

Professional Bachelor Applied Information Science

UAVs autonomously navigating dynamic indoor environments

Vic Segers

Promoter: Tim Dupont

Bachelor paper academic year 2019-2020

Acknowledgements

Abstract

The center of expertise PXL Smart ICT, part of the PXL University of Applied Sciences and Arts research department, has an ongoing project for enabling IT companies to implement Unmanned Aerial Vehicle (UAV) projects via rapid robot prototyping. This internship is an integral part of that research project.

One of the goals of this internship project is updating and fine-tuning an existing Smart UAV software architecture. Another objective is researching the realm of Simultaneous Localization And Mapping (SLAM) algorithms. Robots utilize these algorithms to create a map using their sensors and at the same time locate themselves within this map. Once a map of a certain area exists, a path planning method can be executed in order to navigate between points. Previous goals and objectives will be combined into a showcase where a UAV makes use of a SLAM method to fly and navigate autonomously in a previously unknown dynamic environment.

A SLAM algorithm behaves the same way a human being would when dropped in an unknown environment. A human being opens their eyes and looks around in search of reference points in their environment. These reference points are used as landmarks for their localization. However, unlike humans who use their senses, a UAV uses sensors to get information about its surroundings and uses this to search for reference points. While flying, it can estimate its position based on the movement of these landmarks.

The internship project uses a combination of diverse technologies. Python is predominately used as the programming language. Between the different hardware components and the controlling software, ROS is being used. ROS is an open-source robotics middleware. Remote controlling the UAV is done by using MAVROS. The controlling software implemented with Python and ROS uses MAVROS to communicate via the MAVLink protocol. MAVLink is a lightweight messaging protocol for communicating with UAVs and between onboard UAV components. An autopilot receives these commands and translates them to actual actions that the UAV has to execute. All autopilots used during the internship are based on the open-source PX4 flight control software for UAVs and other unmanned vehicles. For safety and testing purposes, the entire internship project is developed in an open-source 3D robotics simulator, Gazebo. To make the system flexible, modular, and consistent a multi-container Docker environment is used.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

1D	One Dimensional
2D	Two Dimensional
2.5D	Two-and-a-half Dimensional
3D	Three Dimensional
AI	Artificial Intelligence
AR	Augmentend Reality
BAIR	Berkeley Artificial Intelligence Research
BLAM!	Berkeley Localization And Mapping
CRC	Cyclic Redundancy Check
GPS	Global Positioning System
GTSAM	Georgia Tech Smoothing And Mapping
LiDAR	Light Detection And Ranging
ICT	Information Communications Technology
IMU	Inertial Mesurement Unit
IoT	Internet of Things
MAVLink	Micro Air Vehicle Link
OpenGL	Open Graphics Library
QR code	Quick Response code
ROS	Robotic Operating System
RGB-D	Red Green Blue Depth
SLAM	Simultaneous Localization And Mapping
STX	start-of-text

RTAB-Map	Real-Time Appearance-Based Mapping
UAV	Unmanned Aerial Vehicle
VM	Virtual Machine
VR	Virtual Reality

Introduction

In an era where UAVs are becoming more advanced, cheaper, and more socially accepted, they can perform tasks no one would have thought a few years ago. What if these tasks could be executed without human interference? Currently, cars are on the verge of driving fully autonomous. Then would it not be possible for UAVs to fly autonomously aswell?

This paper will focus on the autonomous navigation of a UAV in an indoor dynamic environment. A practical application could be to view or grab packages in a warehouse, another application could be the complete 3D mapping of a building.

To promote the development of UAV applications, an architecture for their development is created. This architecture is a multi-container Docker system, with Gazebo as the simulator of UAVs and the environment, and ROS to combine all used technologies. This project, with the architecture as a backbone, has a showcase where the conducted research is demonstrated.

I Traineeship report

1 About the company

The center of expertise Smart ICT of Hogeschool PXL consists of 21 all-round employees and bundles their knowledge of ICT (software, project management, software architecture, systems) and electronics (focus on hardware and embedded software). The link with education is ensured via the new department PXL-Digital, which besides the bachelor programs applied informatics and electronics-ICT also represents graduate programs Internet of Things, Programming, and Systems & Networks, with about 1500 students in total.

Smart ICT is pursuing a double course: on the one hand, efforts are being made in many vertical domains, such as VR/AR, Internet of Things (IoT), Blockchain, and Artificial Intelligence & Robotics; on the other hand, horizontal support is offered to other centers of expertise, through the development of mobile or web-based applications. Smart ICT offers support to partners from various sectors by responding to practical questions about ICT advice for companies, organizations, and smart cities. The three areas in which Smart ICT is a priority are VR and AR, Internet of Things and Artificial Intelligence & Robotics. Finally, Smart ICT has set itself the goal of evaluating the use of new technologies and transferring these insights to specific target groups, such as the construction sector, education, the retail sector, or the healthcare sector.

I chose Smart ICT because of my interest in Artificial Intelligence & Robotics. It also allowed me to continue working on a project I initiated during my bachelor program. Smart ICT is an expertise center where research is central, which I consciously chose for transferring to a master's program.

2 Technologies

This section provides a brief description of all the technologies that are used throughout the project. It also provides an explanation of why the chosen technologies are used in this project.

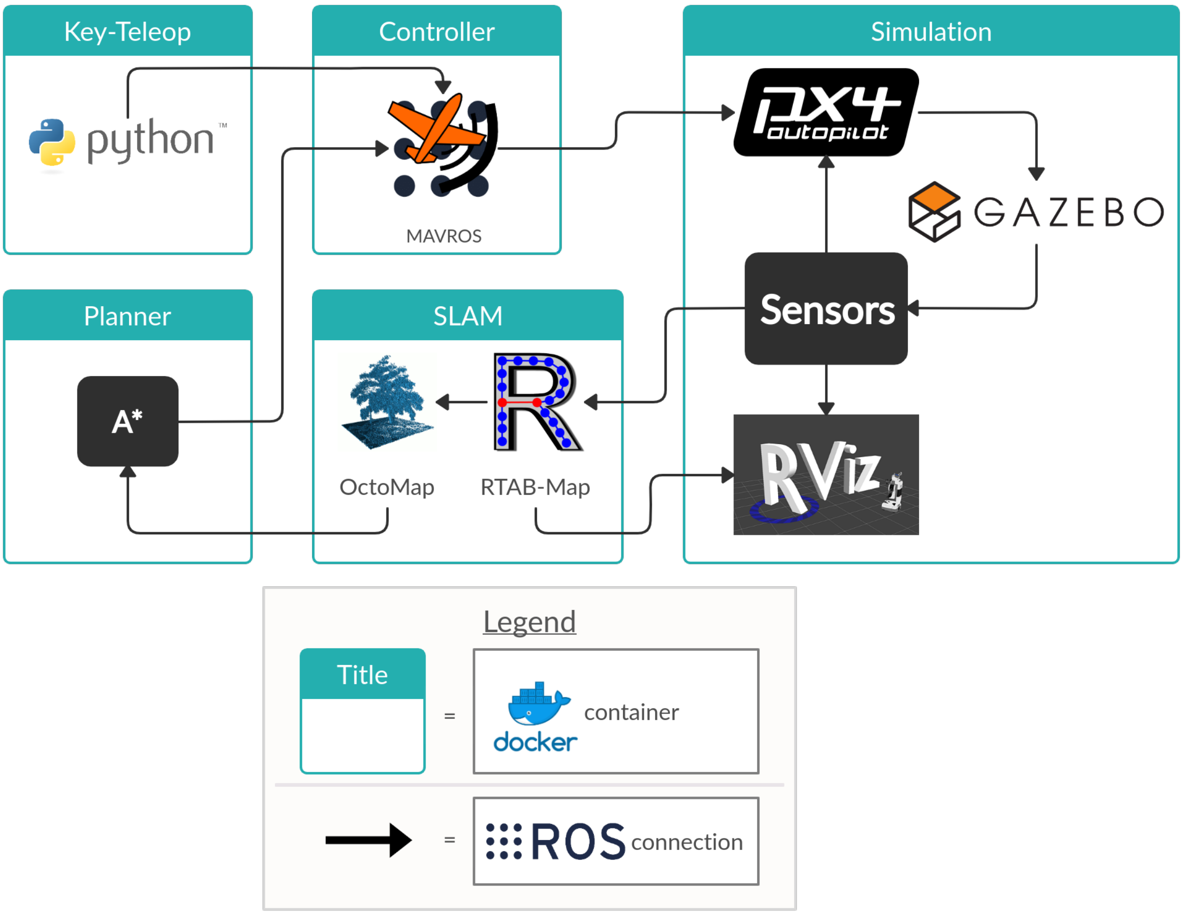

Figure 1: Scheme of the used technologies

2.1 ROS

Robotic Operating System (ROS) is an open-source, meta-operating system for robots. A meta-operating system provides services expected from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. ROS also provides tools and libraries for obtaining, building, writing, and running code across multiple computers. [1]

The primary goal of ROS is to support code reuse in robotics research and development. ROS is a distributed framework of processes - also called nodes - that enables executables to be individually designed and loosely coupled at runtime. By grouping these processes, packages are formed. These packages can be easily shared and distributed. By supporting a federated system of code repositories, ROS enables the distribution of collaboration. This design enables independent decisions about development and implementation, but can be brought together with the ROS infrastructure tools. [2]

The nodes of ROS communicate with each other by publishing messages to topics. A message is a simple data structure, consisting of typed fields. The standard primitive types (integer, floating-point, strings, etc.) are supported in these messages, as are arrays of the primitive types. ROS allows custom defined messages to be sent over its network. Other nodes can subscribe to topics and receive all messages sent to those topics. When a message is received, a callback is triggered that handles the message or acts on something. The ROS Master node provides names and registration services for the rest of the nodes in the system. This is how a single node can find another. [3]

In this project, ROS is used as a communication medium. It allows the chosen technologies to talk with another in a reliable and standardized manner. ROS can be considered as the glue that combines the different technologies and creates a whole.

2.2 Gazebo

Gazebo is an open-source 3D dynamic simulator. It can simulate populations of robots in complex environments with high accuracy and efficiency. Gazebo is similar to game engines, but with a much higher degree of fidelity. Sensors simulated in this environment use this fidelity to function almost in the same way as they would in the real world. [4] [5]

Gazebo is able to connect with ROS and be used as a replacement of the real world. The connection with ROS is realized through a set of ROS packages named *gazebo_ros_pkgs*. This set contains a package that stores all messages and service data structures needed for interacting with Gazebo. Another package provides robot-independent Gazebo plugins for sensors, motors, and dynamic components. The set also has a package that allows for interfacing Gazebo with ROS. [6] [7]

Gazebo is the standard for simulation when developing ROS projects because of its great compatibility with ROS. That is why Gazebo is chosen for this project. The main reasons for developing in a simulation are safety, consistency, economic, and testing purposes. A UAV can be a dangerous and expensive machine. If something went wrong during testing, the UAV could be damaged, or even worse a human could get hurt. That is why Gazebo is a great option for rapid robot prototyping.

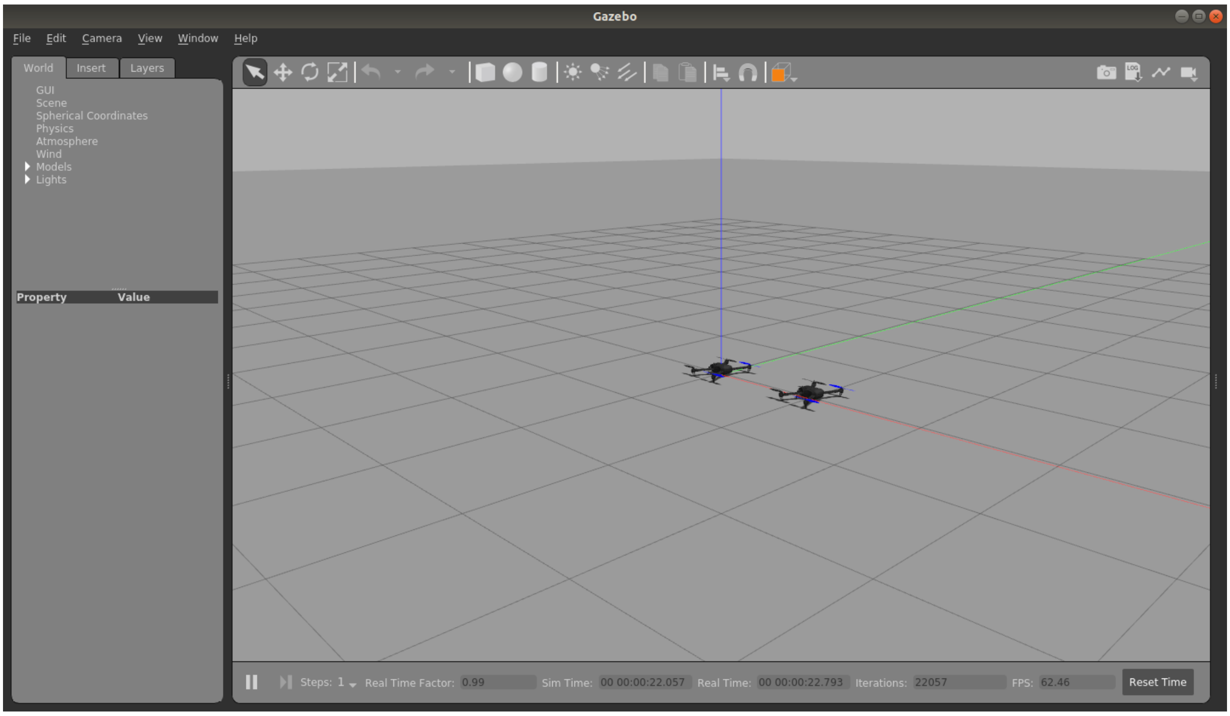

Figure 2: Gazebo simulation

2.3 MAVROS

MAVROS is an open-source translation layer between ROS and MAVLink. MAVLink is a lightweight messaging protocol for communicating with unmanned vehicles. The key features of MAVLink are its efficiency, reliability, support of many programming languages, capability up to 255 concurrent systems on the network, and its ability to enable offboard and onboard communications. MAVLink 1 has just eight bytes overhead per packet and MAVLink 2 has fourteen to increase its security. Therefore this protocol is very well suited for applications with very limited communication bandwidth. [8]

In this project, MAVROS is used for the communication between the written code and the UAV. MAVROS allows for coding on a higher abstraction, with the use of its library.

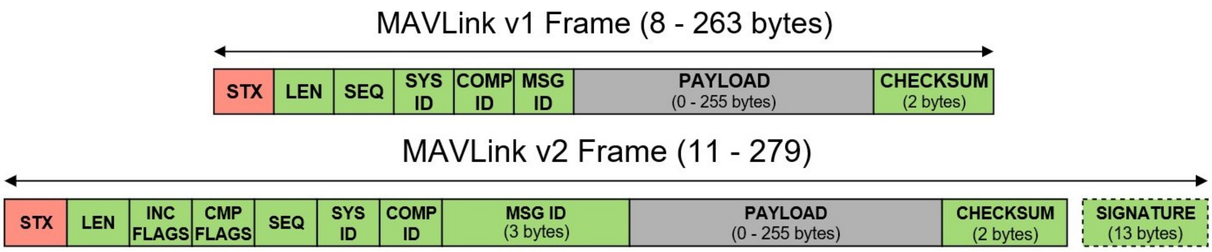

Figure 3: MAVLink packet

Name	Explanation
STX	Protocol-specific start-of-text (STX) marker used to indicate the beginning of a new packet.
LEN	Indicates length of the following payload section.
INC FLAGS	Flags that must be understood for MAVLink compatibility.
CMP FLAGS	Flags that can be ignored if not understood.
SEQ	Used to detect packet loss. Increments value for each message sent.
SYS ID	ID of system (vehicle) sending the message. Used to differentiate systems on network.
COMP ID	ID of component sending the message. Used to differentiate components in a system (e.g. autopilot and a camera).
MSG ID	ID of message type in payload. Used to decode data back into a message object.
PAYLOAD	Message data. Content depends on message type (i.e. MSG ID)
CHECKSUM	X.25 CRC for the message (excluding STX byte).
SIGNATURE	(Optional) Signature to ensure the link is tamper-proof.

Table 1: MAVLink packet explanation

2.4 PX4 Autopilot

The PX4 Autopilot is an open-source autopilot system designed for low-cost UAVs. The autopilot presents the current de-facto standard in the UAV industry and is the leading research platform for UAVs. Furthermore, it also has some successful applications for underwater vehicles and boats. The PX4 Autopilot provides guidance, navigation, control algorithms, and estimators for attitude and position. Thanks to its more flexible hardware and software, modifications are allowed to satisfy special requirements. [9] [10]

Because the software is supported in multiple simulation choices, such as Gazebo. The PX4 Autopilot uses MAVLink as a communication tool. Therefore it can be controlled by ROS. The build-in mode *OFFBOARD* provides full control of the vehicle. [11]

The PX4 software also provides a model of a 3DR Iris Quadrotor that is compatible with Gazebo. The Iris is the default fixed-wing UAV of PX4. The type of UAV is not important when simulating, therefore the default is kept.

2.5 Python

Python is a high-level general-purpose programming language, created by Guido van Rossum. It is developed as an interpreted language, the code is automatically compiled to byte code and executed. Python's philosophy emphasizes code readability achieved by its use of significant whitespace. Mostly Python is not used for its speed or performance because several studies have shown that it is slower than widely-used programming languages, such as Java and C++. However, Python has the option to be extended in C and C++ to speed it up and even be used for compute-intensive tasks. Its strong structuring constructs and its consistent use of objects, enables programmers to write clear and logical applications. [12]

ROS is mainly developed using two languages, C++ and Python. For this project Python was the obvious choice because of its simplicity and readability. The used packages are written in C++ for performance.

2.6 Docker

Docker is an open-source tool designed to simplify the creation, deployment, and running process of applications through the usage of containers. Containers allow the packaging of an application with all of the parts needed, such as libraries and other dependencies. The application is able to run on any Linux machine regardless of any customized settings of that machine, due to the isolation of these containers. [13]

The comparison with a Virtual Machine (VM) is often made. But unlike a VM, Docker does not need virtualization of a whole operating system. The applications run by Docker use the same Linux kernel as the host system. Therefore, compared to a VM, it has a significant reduction in size and a boost in performance. [13]

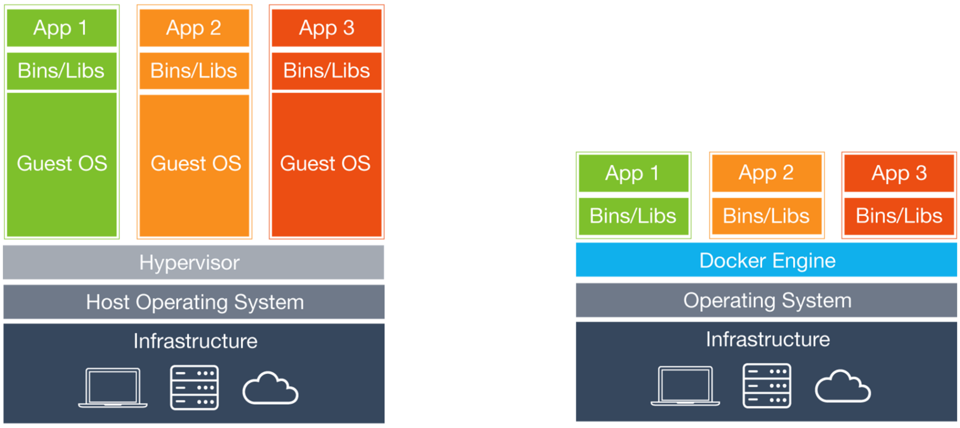

Figure 4: Virtual Machine versus Docker container

Running Docker containers requires images. An image is a read-only template with commands for creating a container. These images can be obtained by two methods: building the image or pulling it from a registry. Pulling an image from a registry is the equivalent of downloading from the internet, these images are premade. Building an image requires a Dockerfile, in this file a set of Docker commands are stated that define the image. A container is a runnable instance of an image. This container can be started, stopped, moved, or deleted. After modifying the container, it can be saved back to an image for later use. The image can then be pushed back to the registry. [14]

This project mainly uses Docker for an isolation option lighter than a VM, its consistency, and scalability. However, the increased freedom is also a bonus. Knowing there is always a stable backup no matter which software or packages are experimented with.

2.7 RViz

RViz is a three-dimensional visualization tool for ROS applications. It can provide a view of the robot model and the captured sensor information. RViz can display data from sensors like cameras and lasers in a 2D or 3D environment. In order to get the data sent through ROS, it has to be launched as a node so it can listen to all topics needed. [15]

This project uses RViz to visually test the written and implemented code. It allows playback of previously saved missions and visualizing all captured points throughout that mission. RViz can function as a replacement of Gazebo's visuals when Gazebo is run headless for increased performance.

3 Implementation

This section describes why these technologies are implemented in this project and how they interact with each other. It also explains the reasoning behind the two large components that this project contains. Those are the architecture and the showcase. The architecture is a clean environment made for developing UAV applications. The showcase in the implementation of the research topic with the architecture as a base.

3.1 Overview

Figure 5 below visualizes how all the elements of the project are connected. All elements are built and run in Docker containers. This allows for the same result on every computer that runs these isolated containers. Providing all big components a separate container, allows for a more robust system. This simulates a real-life scenario, by not having every object in the same container. In the real world, the onboard controller of the UAV is not in direct contact with for example a workstation nearby. This multi-container environment communicates through a Docker network.

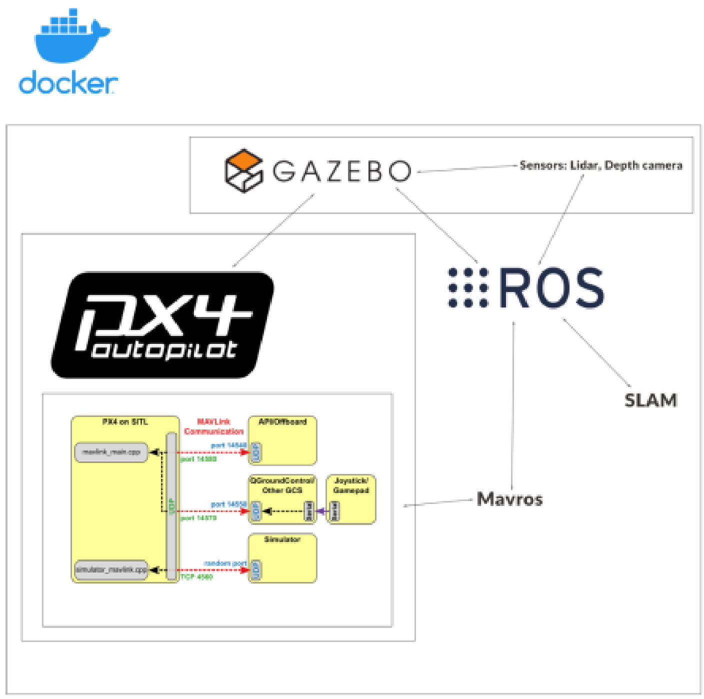

Figure 5: This is a temporary figure

3.2 Architecture

The architecture consists of three containers being a controller, keyboard teleoperation, and a simulator. The simulator runs a Gazebo world that supports multiple 3DR Iris Quadrotor UAVs with each a PX4 Autopilot. The controller sends MAVROS commands to the UAVs for arming and changing their mode to *OFFBOARD*. The *OFFBOARD* mode provides full control of the UAVs through Python code. In the keyboard teleoperation container, pressed keys are parsed to the controller where they are translated to commands and send to the UAVs.

The containers in the architecture are run from different images. The simulation image has an OpenGL base image or a specific NVIDIA base image if the host computer has NVIDIA installed. Adding to this base image, the simulation has ROS Melodic, MAVROS, Gazebo 9, the PX4 Firmware and their dependencies installed. There are two other images used in this project, a ROS base image and a MAVROS image. The keyboard teleoperation container uses the ROS base image with only ROS Melodic installed. The controller needs MAVROS to send commands to the UAVs and therefore uses the MAVROS image that extends from the ROS base image with the installation of MAVROS.

The goal of the architecture is to enable developers to develop UAV applications without having to create a workspace or installing software manually. The repository contains scripts to automate all commands to build, run and stop the containers necessary for the development. The architecture supports UAV and multi-UAV applications.

3.3 Showcase

The showcase uses the architecture as a base and adds features referencing the research topic. The simulation image of the architecture is used as a base for the one in the showcase. It adds support for a VLP-16 Velodyne LiDAR, allows OctoMap in RViz, and provides a QR code detector. A new image is added for a container than runs a SLAM algorithm. Multiple versions of this image are made for different algorithms, such as *hdl_graph_slam* and RTAB-Map. These algorithms are explained in the chapter of the research topic.

The showcase is used to conduct research, visualize this research, and show the capabilities of the architecture. The main addition to the architecture is the implementation of the SLAM algorithms and their dependencies.

II Research topic

The research topic of this paper contains, the autonomous navigation of a UAV in a dynamic indoor environment. Since this topic encloses multiple aspects, it has been subdivided into multiple sub-topics. They each answer a part of the research topic and are combined in the conclusion.

1 Dynamic environment

To understand what a dynamic environment is, its definition and properties need to be defined. This project focuses on an indoor environment, therefore specific indoor properties have to be accounted for.

1.1 Definition

A dynamic environment is an unpredictable environment, the opposite of a static environment. It contains elements that are can change over time. The environment has to be dealt with, with minimal presumptions as if it is completely unknown. This ensures a more general solution that works in most indoor environments.

1.2 Properties

The properties of the environment must be specific to one that is indoor and dynamic. Generally, an indoor environment is smaller than an outdoor environment. A dynamic environment is much more difficult to observe than a static environment because of its ability to change over time.

1.2.1 Indoor

The main property of an indoor environment is that it is enclosed with walls, a ceiling, and a floor. The most important difference with an outdoor environment is that there is no option to use a GPS. The GPS signals may be obstructed or even blocked by the walls and ceiling. Moreover, the accuracy of the signals is not precise enough to function in such a small environment. In an indoor environment, it is not possible to fly higher than the ceiling, in contrast to an outdoor environment where there is no practical height limit.

1.2.2 Dynamic

To make an environment dynamic, it must have the ability to change over time. The change involves numerous things such as moving objects or persons, light intensity, and many other elements that can be detected by the sensors.

2 Orientation

Orientation in an environment can be obtained by first observing the environment, then mapping the observations, and finally locate the current location of the UAV in this map.

2.1 Observing

The first action for orientation is observing the environment. This can be realized with sensors on the UAV that receive data related to its environment. There must also be accounted for dynamic properties.

2.1.1 Sensors

A sensor is a piece of hardware mounted on the UAV that gathers data about the environment or itself is some way. There is are numerous sensors that can be attached to a UAV. For example a camera, a LiDAR, a gyroscope, an accelerometer, and many more. The used sensors are picked by conducting research, using common sense, and by the requirements of an implemented SLAM algorithm.

The most commonly used sensors for SLAM algorithms are optical sensors. Optical sensors may be 1D, 2D, or 3D LiDARs, 2D or 3D sonar sensors, or a normal or a depth camera. All sensors have their pros and cons, for example consistency, price, or weight.

2.1.2 Dynamic properties

2.2 Mapping and localization

In this project, mapping and localization are closely linked together. In order to create a map, the current location of the UAV in its area has to be known to be used as a reference point. However, to get the current location of the UAV in its area, the map of its area has to be known. Therefore, it can be described as a "chicken-or-egg" problem.

This issue can be overcome through the usage of a pre-existing map of an environment, for example, GPS data. However, there is no data for an indoor environment and if there was, it would not be precise enough or updated in real-time. Therefore, the introduction of a SLAM algorithm which is capable of mapping and localizing simultaneously by using feature extraction.

A SLAM system consists of four parts: sensor data, a front-end, a back-end, and the SLAM estimate. The sensor data is all the input data a SLAM system receives. In the front-end a feature extraction process is executed. These features are tracked through a stream of video in the back-end. The back-end also handles long-term associations to reduce drift and triggers loop-closures. As a result, the SLAM systems outputs all tracked features with their locations and relations, as well as the position of the sensors within the world.

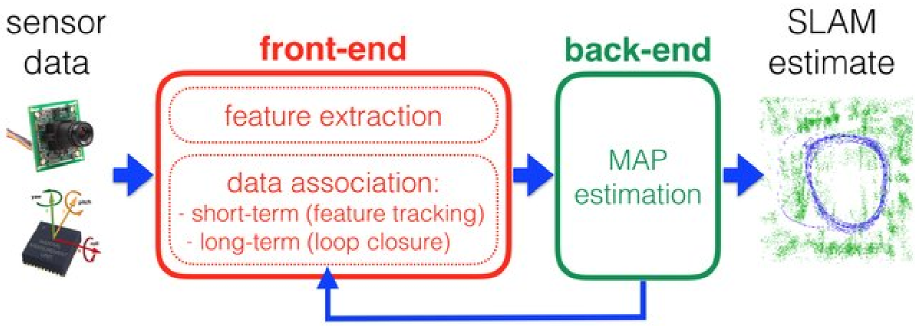

Figure 6: SLAM system

As earlier mentioned, a SLAM algorithm uses feature extraction as a way to track the movement of the UAV in space. The easiest way to explain how feature extraction works is its implementation on a camera. On every frame, features are detected. The way these features are detected differ from algorithm to algorithm. Every feature detected, needs a unique description based on its properties.

The tracking of the features throughout the frames is handled in the back-end. The descriptions of these features of different frames are compared. The motion of the UAV is based on the change of the position of these features relative to the previous frame. To keep account of moving objects and outliers, the majority of features must move in the same direction.

Not only the tracking of features is considered when estimating the motion of the UAV, but different sensor data influence the estimation. Most notable, the data from an accelerometer and a gyroscope. When the algorithm detects a frame it already has visited, it recalculates all previous locations to match this new finding. This is called a loop closure. With loop closures, a SLAM algorithm updates previous values to keep the map as accurate and truthful as possible.

2.3 SLAM algorithms

The decision of what SLAM algorithms are eligible for use, depends on a couple of requirements stated by the project. The algorithm must output a 3D dense point cloud with pose estimation of the UAV. This pose estimation is used for the path planning later on. With these constraints in mind, a few algorithms were researched.

2.3.1 ORB-SLAM2

ORB-SLAM2 is a real-time SLAM algorithm suitable for monocular, stereo and RGB-D cameras. It can handle loop closures and has relocalization capabilities. ORB-SLAM2 was developed in 2017 by Raúl Mur-Artal, Juan D. Tardós, J. M. M. Montiel, and Dorian Gálvez-López. The algorithm returns a sparse 3D reconstruction with true scale. [16]

The algorithm has an option to compiled with ROS. Therefore it seemed a reasonable option for this project. After the implementation and some experimenting with ORB-SLAM2, it became clear that the density of the point cloud was to sparse for this project.

2.3.2 Cartographer

Cartographer is a real-time SLAM system that support 2D and 3D across multiple platforms and sensor configurations. Aswell as a build-in ROS implementation. The system is developed and maintained by Google. [17]

The reason for researching this algorithm is that it is well maintained and often used in a lot of projects. The conclusion that Cartographer is not a real 3D but more a 2.5D algorithm came during the research. 2.5D is a pseudo-3D perspective. The algorithm can combine multiple 2D scans and combine them, but that is not truly 3D. [18]

2.3.3 BLAM!

BLAM! is an open-source LiDAR based real-time 3D SLAM software package. Developed in 2016 by Erik Nelson from the Berkeley Artificial Intelligence Research (BAIR) laboratory. The software is build in a ROS environment, containing two ROS workspaces. [19]

Because of its outdated dependencies and the difficulty of installation, there was no implementation possible of BLAM! in this project. BLAM! has ROS, GTSAM, and Boost as a dependency. The algorithm looked promising, but it does not support a camera as a sensor for visual odometry.

2.3.4 hdl_graph_slam

The build in a ROS package hdl_graph_slam is an open-source 3D LiDAR based SLAM. The package has been tested in indoor and outdoor environments with a Velodyne HDL-32E, a Velodyne VLP-16, and a RobotSense RS-LiDAR-16. The package is developed in 2019 by Kenji Koide, Jun Miura, and Emanuele Menegatti. It supports multiple constraits that can be individually be enabled or disabled, being odometry, loop closure, GPS, IMU acceleration, IMU orientation, and floor plane detection. [20]

This package was the first fully implemented SLAM algorithm of this project. Not being able to deal with loop closes was its mayor flaw for this project. It worked perfectly when not disturbed, therefore it was not reliable.

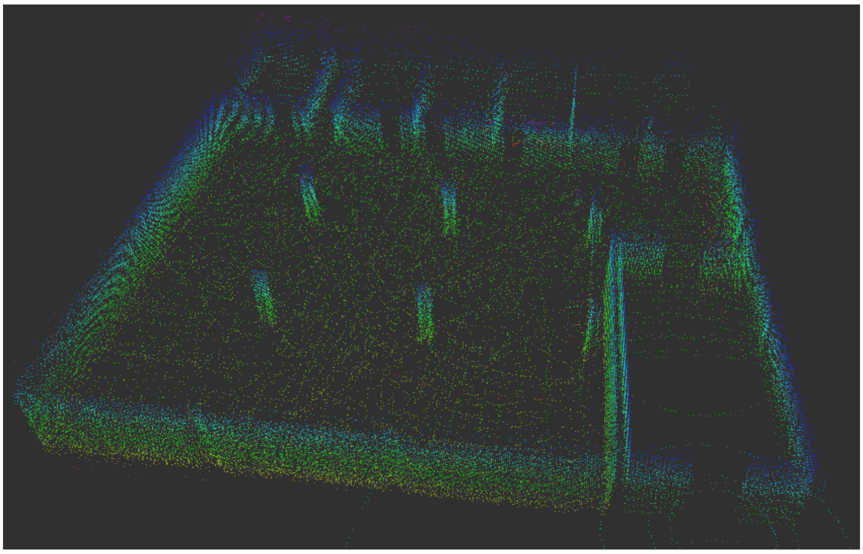

Figure 7: Implementation of hdl_graph_slam

2.3.5 RTAB-Map

RTAB-Map is an open-source RGB-D, stereo camera and, LiDAR graph-based SLAM system. The system is based on an incremental appearence-based loop closure detector. The detector uses a bag-of-words approach to determine how likely a new image comes from a previous or a new location. RTAB-Map is a standalone application available for Ubuntu, Mac OS X, Windows, and a Raspberry Pi. But, also as a ROS package and on the Google Play Store. It is a heavily maintained piece of software. [21]

The ROS package of RTAB-Map has a lot of parameters to be adjusted to a specific need. The build-in support with OctoMap allows it to be used for path planning. RTAB-Map meets all the requirements needed for this project.

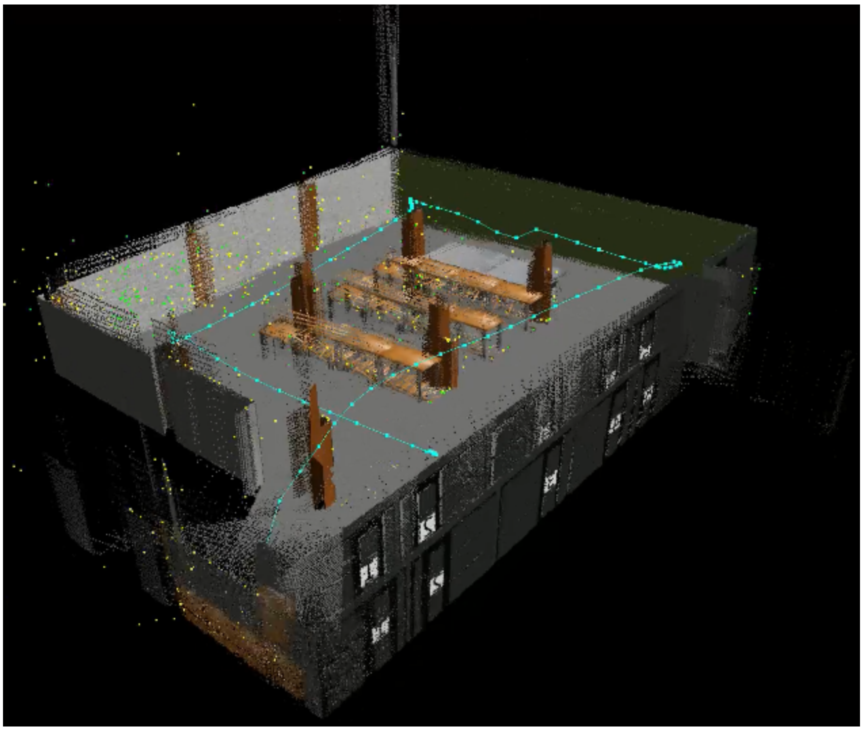

Figure 8: Implementation of RTAB-Map

2.3.6 Comparison SLAM algorithms

Algorithm	Sensors	Point cloud	Dimentional
ORB-SLAM2	monocular, stereo and RGB-D camera	sparse	3D
Cartographer	0	0	2D & 3D
BLAM!	LiDAR	0	3D
hdl_graph_slam	LiDAR	dense	3D
RTAB-Map	RGB-D, stereo and LiDAR	dense	3D

Table 2: Comparison SLAM algorithms

3 Path planning

Path planning is the task of finding a continuous path from the start to the goal. The path planning algorithm must have a definition of an executable path and a goal. The start is the current position of a UAV. The algorithm uses 3D occupancy grid where the occupied and the free space are indicated. [22]

3.1 Executable path

An executable path is defined as a path clear of obstacles and spacious enough for a UAV to fly comfortably. The algorithm uses the 3D occupancy grid and returns the shortest path in the space that is not occupied. The shortest path is checked if the size of the UAV with added margin fits in every position of the path, or else the path is modified until all requirements meet.

3.2 Goal

The goal is defined by a QR code placed in the world. One UAV explores the indoor environment searching for the QR code. Once found, another UAV plans a path with the QR code as its goal.

The QR code is generated with a repository that outputs a model for Gazebo with an image as input. The UAV can detect this code with the use of a ROS package. Once a QR code is detected, a message is published with its location for path planning. [23] [24]

3.3 Search algorithms

3.3.1 Dijkstra

3.3.2 Theta*

3.3.3 A*

3.3.4 Comparison search algorithms

4 Path execution

Once a valid path is obtained through path planning, it has to be executed by a UAV. The path is an array of coordinates. Once the UAV has reached a coordinate, it receives the next coordinate in the array until the goal is reached. The UAV reaches a coordinate when its current position is within a certain distance of the coordinate. This value is calculated with the euclidean distance.

5 Difficulties

There is currently no ROS algorithm for exploration in a 3D environment. The first UAV should utilize such an algorithm. Because of the time constraint of the project, the first UAV executes a predetermined path that leads to the QR code which acts as the goal for the following UAV.

Conclusion

Bibliographical references

[1] *"ROS/Introduction", ROS wiki.* [Online]. Available: `http://wiki.ros.org/ROS/Introduction` (visited on Apr. 23, 2020).

[2] W. Newman, *"A Systematic Approach to Learning Robot Programming with ROS"*. CRC Press, 2017.

[3] *"Messages", ROS wiki.* [Online]. Available: `http://wiki.ros.org/Messages` (visited on Apr. 23, 2020).

[4] *"Beginner: Overview", Gazebo.* [Online]. Available: `http://www.gazebosim.org/tutorials?cat=guided_b&tut=guided_b1` (visited on Apr. 23, 2020).

[5] E. Ackerman, *"Latest version of gazebo simulator makes it easier than ever to not build a robot", IEEE Spectrum.* [Online]. Available: `https://spectrum.ieee.org/automaton/robotics/robotics-software/latest-version-of-gazebo-simulator` (visited on Apr. 23, 2020).

[6] *"ROS overview", Gazebo.* [Online]. Available: `http://gazebosim.org/tutorials?tut=ros_overview` (visited on Apr. 23, 2020).

[7] *"ROS control", Gazebo.* [Online]. Available: `http://gazebosim.org/tutorials?tut=ros_control` (visited on Apr. 23, 2020).

[8] *"MAVLink Developer Guide", MAVLink.* [Online]. Available: `https://mavlink.io/en/` (visited on Apr. 24, 2020).

[9] *"Open Source for Drones", PX4 Autopilot.* [Online]. Available: `https://px4.io/` (visited on Apr. 23, 2020).

[10] *"Projects", Dronecode.* [Online]. Available: `https://www.dronecode.org/projects/` (visited on Apr. 23, 2020).

[11] N. Mimmo and F. Mahlknecht, “Implementation of an autonomous navigation algorithm with collision avoidance for an unmanned aerial vehicle”,

[12] D. Kuhlman, *"A python book: Beginning python, advanced python, and python exercises"*. Dave Kuhlman Lutz, 2009.

[13] *"What is Docker?", Opensource.com.* [Online]. Available: `https://opensource.com/resources/what-docker` (visited on Apr. 25, 2020).

[14] *"Docker overview", Docker.* [Online]. Available: `https://docs.docker.com/get-started/overview/` (visited on Apr. 26, 2020).

[15] *"AWS RoboMaker: Developer Guide", AWS*. [Online]. Available: https://docs.aws.amazon.com/robomaker/latest/dg/aws-robomaker-dg.pdf (visited on Apr. 26, 2020).

[16] R. Mur-Artal and J. D. Tardós, "ORB-SLAM2: An open-source SLAM system for monocular, stereo and RGB-D cameras", *IEEE Transactions on Robotics*, vol. 33, no. 5, pp. 1255–1262, 2017. DOI: 10.1109/TRO.2017.2705103.

[17] *"Cartographer ROS Integration", Google Cartographer*. [Online]. Available: https://google-cartographer-ros.readthedocs.io/en/latest/ (visited on Apr. 27, 2020).

[18] J. Liang, J. Gong, J. Liu, Y. Zou, J. Zhang, J. Sun, and S. Chen, "Generating orthorectified multi-perspective 2.5 d maps to facilitate web gis-based visualization and exploitation of massive 3d city models", *ISPRS International Journal of Geo-Information*, vol. 5, no. 11, p. 212, 2016.

[19] E. Nelson, *"BLAM!", GitHub*. [Online]. Available: https://github.com/erik-nelson/blam (visited on Apr. 27, 2020).

[20] K. Koide, J. Miura, and E. Menegatti, "A portable three-dimensional lidar-based system for long-term and wide-area people behavior measurement", *International Journal of Advanced Robotic Systems*, vol. 16, no. 2, p. 1 729 881 419 841 532, 2019.

[21] *"Introlab RTAB-Map", RTAB-Map*. [Online]. Available: http://introlab.github.io/rtabmap/ (visited on Apr. 27, 2020).

[22] G. Klančar, A. Zdešar, S. Blažič, and I. Škrjanc, "Chapter 4 - path planning", in *Wheeled Mobile Robotics*, G. Klančar, A. Zdešar, S. Blažič, and I. Škrjanc, Eds., Butterworth-Heinemann, 2017, pp. 161–206, ISBN: 978-0-12-804204-5. DOI: https://doi.org/10.1016/B978-0-12-804204-5.00004-4. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780128042045000044.

[23] M. Arguedas, *"AR tags models for Gazebo", GitHub*. [Online]. Available: https://github.com/mikaelarguedas/gazebo_models (visited on Apr. 27, 2020).

[24] M. Drwiega, *"QR detector", GitHub*. [Online]. Available: https://github.com/mdrwiega/qr_detector (visited on Apr. 27, 2020).