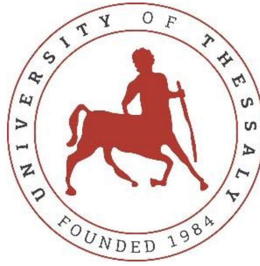


UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

IMPLEMENTATION OF AN AUTONOMOUS NAVIGATION SYSTEM FOR UNMANNED GROUND VEHICLES

Diploma Thesis

Emmanouil Maroulis

Supervisor: Athanasios Korakis

Volos 2021

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΥΛΟΠΟΙΗΣΗ ΣΥΣΤΗΜΑΤΟΣ ΑΥΤΟΝΟΜΗΣ ΠΛΟΗΓΗΣΗΣ ΓΙΑ ΜΗ ΕΠΑΝΔΡΩΜΕΝΑ ΕΠΙΓΕΙΑ ΟΧΗΜΑΤΑ

Διπλωματική Εργασία

Εμμανουήλ Μαρούλης

Επιβλέπων: Αθανάσιος Κοράκης

Βόλος 2021

Εγκρίνεται από την Επιτροπή Εξέτασης:

Επιβλέπων/πουσα **Κοράκης Αθανάσιος**

Αναπληρωτής καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Πανεπιστήμιο Θεσσαλίας

Μέλος **Αργυρίου Αντώνιος**

Αναπληρωτής καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Πανεπιστήμιο Θεσσαλίας

Μέλος **Μπαργιώτας Δημήτριος**

Αναπληρωτής καθηγητής, Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Πανεπιστήμιο Θεσσαλίας

Ημερομηνία έγκρισης: 09-07-2021

Acknowledgements

Firstly, I would like to thank my supervisor, Professor Athanasios Korakis, for his support and for giving me the chance to handle such an amazing project. In addition, I am grateful to Giannis Kazdaridis for his help and valuable advice throughout this project. Also, I would like to thank NITLab's team for their support and guidance, which helped me improve.

Finally, I would like to thank my friends and family for always supporting me.

Dedicated to my mother.

ΥΠΕΥΘΥΝΗ ΔΗΛΩΣΗ ΠΕΡΙ ΑΚΑΔΗΜΑΪΚΗΣ ΔΕΟΝΤΟΛΟΓΙΑΣ ΚΑΙ ΠΝΕΥΜΑΤΙΚΩΝ ΔΙΚΑΙΩΜΑΤΩΝ

Με πλήρη επίγνωση των συνεπειών του νόμου περί πνευματικών δικαιωμάτων, δηλώνω ρητά ότι η παρούσα διπλωματική εργασία, καθώς και τα ηλεκτρονικά αρχεία και πηγαίοι κώδικες που αναπτύχθηκαν ή τροποποιήθηκαν στα πλαίσια αυτής της εργασίας, αποτελεί αποκλειστικά προϊόν προσωπικής μου εργασίας, δεν προσβάλλει κάθε μορφής δικαιώματα διανοητικής ιδιοκτησίας, προσωπικότητας και προσωπικών δεδομένων τρίτων, δεν περιέχει έργα/εισφορές τρίτων για τα οποία απαιτείται άδεια των δημιουργών/δικαιούχων και δεν είναι προϊόν μερικής ή ολικής αντιγραφής, οι πηγές δε που χρησιμοποιήθηκαν περιορίζονται στις βιβλιογραφικές αναφορές και μόνον και πληρούν τους κανόνες της επιστημονικής παράθεσης. Τα σημεία όπου έχω χρησιμοποιήσει ιδέες, κείμενο, αρχεία ή/και πηγές άλλων συγγραφέων, αναφέρονται ευδιάκριτα στο κείμενο με την κατάλληλη παραπομπή και η σχετική αναφορά περιλαμβάνεται στο τμήμα των βιβλιογραφικών αναφορών με πλήρη περιγραφή. Αναλαμβάνω πλήρως, ατομικά και προσωπικά, όλες τις νομικές και διοικητικές συνέπειες που δύναται να προκύψουν στην περίπτωση κατά την οποία αποδειχθεί, διαχρονικά, ότι η εργασία αυτή ή τμήμα της δεν μου ανήκει διότι είναι προϊόν λογοκλοπής.

Ο/Η Δηλών/ούσα

Μαρούλης Εμμανουήλ

Ημερομηνία 09/07/2021

ΠΕΡΙΛΗΨΗ

Στόχος αυτής της εργασίας είναι η ανάπτυξη και υλοποίηση ενός συστήματος αυτόνομης πλοήγησης, το οποίο θα έχει την δυνατότητα να πλοηγείτε σε γνωστά ή άγνωστα περιβάλλοντα, στα οποία δεν υπάρχει δυνατότητα χρήσης GPS. Επίσης, μπορεί να αποφύγει στατικά αλλά και δυναμικά εμπόδια σε πραγματικό χρόνο. Χρησιμοποιούμε και δοκιμάζουμε διάφορους αισθητήρες (κάμερες, LIDAR, IMU), διαμορφώνουμε δυο διαφορετικού είδους SLAM αλγορίθμους για να πετύχουμε το επιθυμητό αποτέλεσμα. Ο πρώτος αλγόριθμος SLAM βασίζεται σε δεδομένα βάθους στον δισδιάστατο ή τρισδιάστατο χώρο, ο δεύτερος αλγόριθμος βασίζεται σε οπτικά δεδομένα, δηλαδή εικόνες από κάμερα. Το τελικό σύστημα αποτελείται από δυο διαφορετικές κάμερες και ένα αισθητήρα IMU, εκτελεί εντολές πλοήγησης με επιτυχία στο διαμορφωμένο εσωτερικό χώρο του Network Implementation Testbed Laboratory (NITLab) με ακρίβεια θέσης $\pm 4\ cm$.

ABSTRACT

The goal of this thesis is the development and implementation of an autonomous navigation system, which will be able to navigate through known or unknown GPS-less environments. Also, it will be able to avoid static and dynamic obstacles in real time. We are using and testing a set of sensors (cameras, LIDAR, IMU) and configure two different kinds of SLAM algorithms to accomplish our goal. The first SLAM algorithm uses 2D or 3D depth information, the second algorithm is based on visual data only, like camera images. Our final system consists of two different cameras and an IMU sensor, it successfully executed navigation commands in the Network Implementation Testbed Laboratory (NITLab) indoor testbed environment and achieved a position accuracy of $\pm 4\ cm$.

Table of Contents

Chapter 1 Introduction 1

1.1 Problem description 1

1.2 Purpose of this project 1

1.3 Thesis Structure 2

Chapter 2 Infrastructure and tools 2

2.1 **Software components** 3

2.1.1 ROS 3

2.1.2 Visualization 3

2.1.3 Publish/Subscribe pattern 4

2.1.4 Nodelets 4

2.1.5 Point Cloud 5

2.1.6 Sensor transforms 5

2.2 **Hardware components** 6

2.2.1 RICOH THETA V 6

2.2.2 Inertial Measurement Unit (IMU) 6

2.2.2.1 Gyroscope 7

2.2.2.2 Accelerometer 8

2.2.3 Intel RealSense D435i 8

2.2.4 Light Detection and Ranging (LiDAR) sensor 9

Chapter 3 Simultaneous Localization And Mapping (SLAM) 9

3.1 **SLAM with proximity sensor** 10

3.1.1 Sensor input 11

3.1.2 Local SLAM 12

3.1.3 Global SLAM 12

3.2 **Visual SLAM** 14

3.2.1 Feature detection 14
3.2.1.1 Features from Accelerated Segment Test (FAST) 15
3.2.1.2 Oriented Fast and Rotated BRIEF (ORB) 15
3.2.2 Algorithm walkthrough 16
Chapter 4 Autonomous Navigation 17
4.1 **move_base package** 17
4.1.1 Global, local cost-map 19
4.1.2 Path planning 20
4.1.2.1 Dijkstra's algorithm 20
4.1.3 Recovery behaviour 22
4.2 **Algorithm walkthrough** 22
4.2.1 Base controller 22
4.3 **Sensor fusion and Kalman filter** 24
4.3.1 Basic Terms 25
4.3.2 Kalman filter algorithm 26
Chapter 5 System integration 28
5.1 **LiDAR implementation** 28
5.2 **RICOH THETA implementation** 29
5.3 **Intel real-sense implementation** 33
5.4 **Multicamera system** 37
5.4.1 Kalman filter integration 40
Chapter 6 Conclusions & Discussion 41
6.1 **Summary** 41
6.2 **Discussion** 41
6.2.1 Implement parts of the system without ROS 41
6.2.2 Hardware upgrade 42
6.2.3 3D SLAM 42

6.3 **Future work** 42
6.3.1 Virtual Reality teleoperation 42
6.3.2 Object detection 43
Bibliography 43

List of figures

Figure 2.1 Robot in a map. 4
Figure 2.2 Transform tree. 5
Figure 2.3 RICOH THETA V camera. 6
Figure 2.4 Inertial Measurement Unit 7
Figure 2.5 Simple gyroscope. 7
Figure 2.6 Intel RealSense D435i camera. 8
Figure 2.7 RPLidar A1M8 device 9
Figure 3.1 Google cartographer structure. 10
Figure 3.2 The effect of voxel grid filter on 3D. 11
Figure 3.3 Global optimization results. 13
Figure 3.4 Example Lua file. 14
Figure 3.5 Multiscale pyramid. 16
Figure 3.6 Visual SLAM. 17
Figure 4.1 Global cost map configuration file. 18
Figure 4.2 move_base set of nodes. 19
Figure 4.3 Global cost map. Black cells are inflated objects, green rectangle is the robot and the red arrow represents its orientation. 20
Figure 4.4 local cost map aligned with a normal map. 20
Figure 4.5 Starting cell is “s”, adjacent cells are considered neighbors. 21

Figure 4.6 Red and pink cells represent obstacles, blue cells represent the inflation of obstacles, green cells represent the path. 21
Figure 4.7 Motor control pseudocode. 23
Figure 4.8 Motor driver. 24
Figure 4.9 Gaussian distribution $N\chi, \sigma 2$ 25
Figure 4.10 Covariance matrix. 26
Figure 4.11 Kalman filter. 27
Figure 5.1 Robot platform. 28
Figure 5.2 System with LiDAR sensor. 28
Figure 5.3 System structure using LiDAR sensor. 29
Figure 5.4 System with 360o camera. 29
Figure 5.5 Map created by openVSLAM, white cells are features, green cells represent the path followed. 30
Figure 5.6 Each rectangle represents a node in the ROS system. 31
Figure 5.7 Pseudo code of the SLAM implementation and filtering procedure. 32
Figure 5.8 Black cells represent obstacles, white cells represent free space, red arrow represents the robot's pose. 32
Figure 5.9 Rover with IntelRealSense D435i camera. 33
Figure 5.10 Pseudo code of the SLAM system, using IntelRealSense D435i camera. 34
Figure 5.11 Yellow cells are features, algorithm is mapping. 35
Figure 5.12 Camera is turned, and algorithm is lost. 35
Figure 5.13 The chair is moved out of the field of view, camera is back on the starting position and the algorithm is still lost. 35
Figure 5.14 Picking the chair and moving it. 35
Figure 5.15 Algorithm still cannot match the features. 35
Figure 5.16 Chair back on starting position, algorithm tracks the features again. 35
Figure 5.17 Different colors of the line represent different distances from the camera. 36

Figure 5.18 System structure using only IntelRealSense D435i. 36

Figure 5.19 Multicamera system 37

Figure 5.20 Position and orientation coordinates of odometry. 38

Figure 5.21 System structure using two cameras. 39

Figure 5.22 Red arrow represents the orientation and position of the robot in the map.......... 39

Figure 5.23 System structure including an Extended Kalman Filter. 40

Chapter 1 Introduction

1.1 Problem description

Nowadays, engineers are trying to automate many procedures, in order to accomplish complex tasks with little human effort and risk. One of the most challenging tasks, is autonomous navigation, which includes vehicle navigation in public streets, ground or aerial robot navigation in indoor or outdoor environments. To achieve this goal, many techniques are developed, especially in the robotics field. These techniques include algorithms and mathematic models, that process data in fast and efficient ways, hardware components that produce those data, in order to understand the surrounding environment and the robot's or vehicle's state.

Distinct sensors that produce information about the environment are designed and integrated into autonomous mobile systems, such as laser sensors and cameras. Laser sensors are very common, and can be found in almost every mobile vehicle, they provide depth information, which is important for a mobile robot. Cameras especially, are used in a lot of tasks in autonomous systems, like object detection, Simultaneous Localization And Mapping (SLAM), visualization for remote control etc. However, it is difficult to integrate them into a system and configure them properly.

1.2 Purpose of this project

The goal of this project is to create an autonomous mobile robot, which can localize itself, navigate in (un)known GPS-less environments, create a map and avoid obstacles, if any, simultaneously. All those objectives have to be accomplished in real time, otherwise the system will not correspond to real life scenarios and we will not be able to deploy it in future experiments. This kind of system has many applications in everyday life such as health, indoor object search, work, military and in mining sites. We are using cameras, laser sensors and an IMU in our system and test two SLAM algorithms in our implementation. The robot is tested

at the NITLab's indoor GPS-less testbed, in which our robot is deployed as a mobile node, so that experiments can be conducted with it.

1.3 Thesis Structure

The thesis is divided in six Chapters with the following content.

Chapter 2: illustrates the technologies and techniques, a reader should know, in order to comprehend the rest of the content. It is divided in two categories, the first is the software we used in the project and the second one is the hardware, we examine each component individually.

Chapter 3: investigates in depth the SLAM algorithms we use, more specific, the way a map is constructed and the way a robot obtains its position in it. Also, it introduces the concept of sensor fusion and a specific algorithm our system includes, Kalman filters.

Chapter 4: describes the algorithms used to achieve autonomous navigation and the prerequisites. Examines the procedure of how a robot avoids obstacles and how path planning works. In the end, an overall description of how all the algorithms cooperate to get an optimal result is given.

Chapter 5: contains all the systems implemented and tested. It describes the reasons why some specific implementations cannot work in real situations and why others can. In the end, a complex system is built, in order to tackle all the problems, other systems face.

Chapter 6: summarizes the thesis and looks ahead for the next steps of the project.

Chapter 2 Infrastructure and tools

The making of an autonomous navigation system consists of different tools. Hardware components like minicomputers, microcontrollers, sensors, batteries etc., software is included

and developed to fetch and process data from available resources (hardware). Below we take a better look at all the components our system consists of.

2.1 Software components

Most of the software developed is written in C++, also many ready to use packages are included to develop our project. The platform that assisted us with the packages needed is ROS. Below, we clarify the resources we use and why.

2.1.1 ROS

ROS[1] [1] is an opensource meta-operating system for robots and is mostly used in Unix operating systems. It provides the same utilities as a normal operating system does in a computer, some of these utilities are, hardware control, package manager for easier finding and installation and a communication system between processes. Also, it provides tools that help create and execute code, which can be distributed among different computers. This is achieved due to the architecture of the system, it uses Nodes[2] to execute code, Topics[3] and Messages[4] for the intra-Nodes communication. This operating system is used by many developers, that develop robot software. In our implementation ROS is used as the core of the system.

2.1.2 Visualization

ROS offers tools to visualize different kinds of procedures, among others we can visualize the map of an environment, the path a robot is going to follow and the position of the robot in the map. The tool we use in this project is rviz[5], because it gives us the capability to send commands to the robot and debug our system errors. An example map and a robot are shown below.

---

[1] https://www.ros.org/about-ros/

[2] http://wiki.ros.org/Nodes

[3] http://wiki.ros.org/Topics

[4] http://wiki.ros.org/Messages

[5] http://wiki.ros.org/rviz

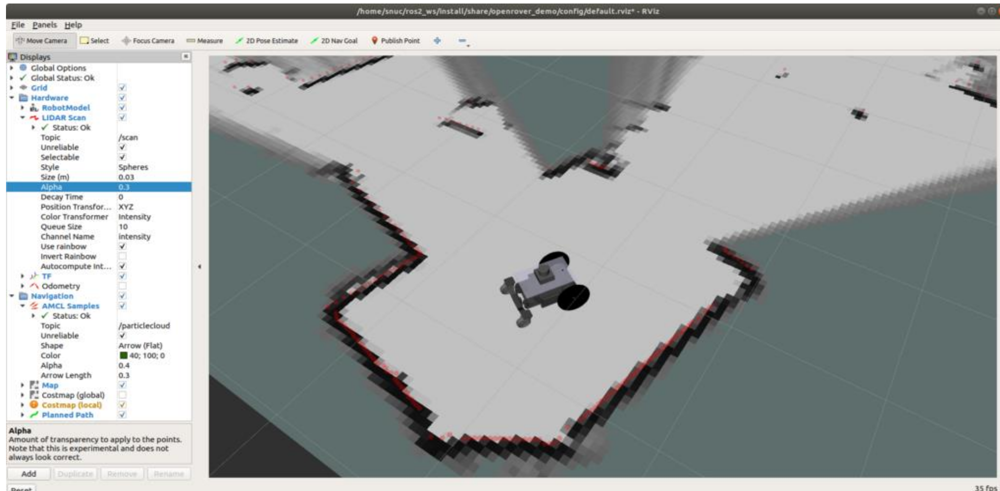


Figure 2.1 Robot in a map.

6

2.1.3 Publish/Subscribe pattern

In software architecture, publish-subscribe is a communication pattern, where senders of messages, called publishers, do not send messages to a specific receiver (subscriber), rather than classify their messages into groups, without knowledge of which subscribers, if any, there may be. In a similar way, subscribers express interest in one or more classes and receive messages that are of interest, without knowledge of which subscribers, if any, there are. ROS is based on this architecture for the Nodes communication.

2.1.4 Nodelets

On many occasions, message communication is very time consuming because there are a lot of data to transfer. Nodelets[7] are designed to provide a way to run multiple algorithms on a single machine, in a single process without incurring copy costs when passing messages intra-process. In conjunction with the publish/subscribe pattern, nodelets are used in our system to transfer images from the cameras efficiently.

[6] *https://blog.roverrobotics.com/ros-2-maps-maps-maps/*

[7] http://wiki.ros.org/nodelet

2.1.5 Point Cloud

Point cloud is a set of data points in space. The points represent a 3D shape or object. Each point has its set of X, Y, Z coordinates. In our implementation point clouds are produced from depth images and are processed accordingly to map an environment.

2.1.6 Sensor transforms

A very frequent problem when dealing with robots, with multiple joints or sensors is the wrong reference system transformation. ROS offers a package that is designed to refer data to multiple reference systems, that can vary over time, also it maintains the relationship between coordinate frames in a tree structure buffered in time. This package makes the coordinates frames available to all ROS nodes on any computer connected to the system. The visualization of these transforms has a tree structure, so we refer to it as transform tree. Below is shown a simple transform tree.

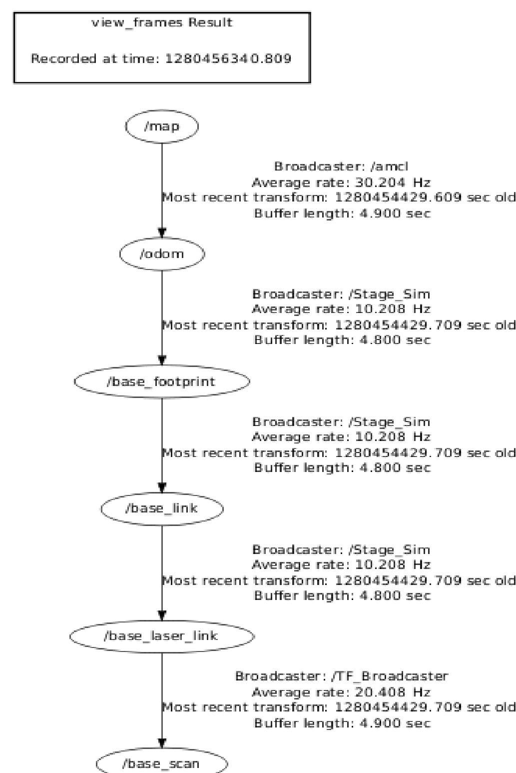


Figure 2.2 Transform tree.

2.2 Hardware components

Robots are built from many different components, that provide a robot with the ability to sense and perceive an environment. Most of the hardware we utilize, are commercial out of the shelf products. Below, we elaborate on the reasons why we chose each component.

2.2.1 RICOH THETA V

RICOH THETA V[8], is a 360o camera, with an improved video performance, transfer speed and extensibility. It produces high resolution videos while streaming live and has opensource plugins for developers to hack. It is supported by the Linux operating system, which makes it easy to connect on small computers (like Raspberry pi computers). We use it as an input of images, in order to localize our robot and navigate to unknown environments.

Figure 2.3 RICOH THETA V camera.

2.2.2 Inertial Measurement Unit (IMU)

A combination of sensors is important to measure and estimate a robot's position in a specified environment, to fetch this kind of information we use an IMU sensor. An IMU is an electronic

[8] https://theta360.com/en/about/theta/v.html

device that measures and reports a body's gravitational force, angular rate[9] and acceleration. A fusion of this information provides the orientation of the sensor.

To reduce the system's complexity, the final system uses the Intel RealSense D435i built in IMU. Below we elaborate on each sensor a simple IMU consists of.

Figure 2.4 Inertial Measurement Unit

2.2.2.1 Gyroscope

A gyroscope is a rotation sensor capable of measuring orientation and angular velocity based on the principles of angular momentum[10]. The device consists of a spinning wheel or disc in which the axis of rotation (spin axis) is free, so the wheel or disc can measure its orientation around every axis. When the sensor is rotated the spin axis is unaffected from the rate of change of angular displacement.

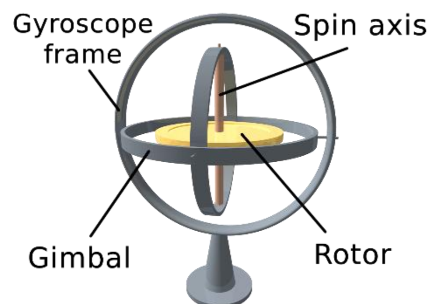


Figure 2.5 Simple gyroscope.

[9] https://en.wikipedia.org/wiki/Angular_velocity

[10] https://en.wikipedia.org/wiki/Angular_momentum#Conservation_of_angular_momentum

2.2.2.2 Accelerometer

An accelerometer is a device that measures proper acceleration in a single or more dimensions (dimensions might differ depending on the device). Proper acceleration is the rate of change of velocity of a body in own frame. For example, an accelerometer at rest on the surface of the will measure an acceleration due to Earth's gravity. To the contrary, an accelerometer in free fall (falling towards the center of Earth at rate of about $9.81\frac{m}{s^2}$) will measure zero acceleration. We use this information to measure our robot's acceleration on the floor/road.

2.2.3 Intel RealSense D435i

D435i[11] is a stereo camera, which generates RGB and depth images (RGB-D), it also has a built in IMU device. We chose this device because, there is a lot of available documentation and a Software Development Kit (SDK) provided by Intel, which helped us develop the necessary functionalities. Images from this camera are used to test a Visual SLAM algorithm (read below) and detect obstacles.

Figure 2.6 Intel RealSense D435i camera.

[11] https://www.intelrealsense.com/depth-camera-d435i/

2.2.4 Light Detection and Ranging (LiDAR) sensor

LiDAR sensor provides accurate information about the distance between the sensor and an object in a specified range in the 1D, 2D or 3D space (depends on the sensor's capabilities). In one implementation a 360o 2D LiDAR (RP LiDAR A1-M8[12]) sensor is used, to scan the environment in a certain radius.

Figure 2.7 RPLidar A1M8 device.

Chapter 3 Simultaneous Localization And Mapping (SLAM)

In this chapter we are going to take a deeper look, on a very common problem of autonomous mobile systems, localization in static and dynamic environments and mapping those environments. At first, we define those procedures.

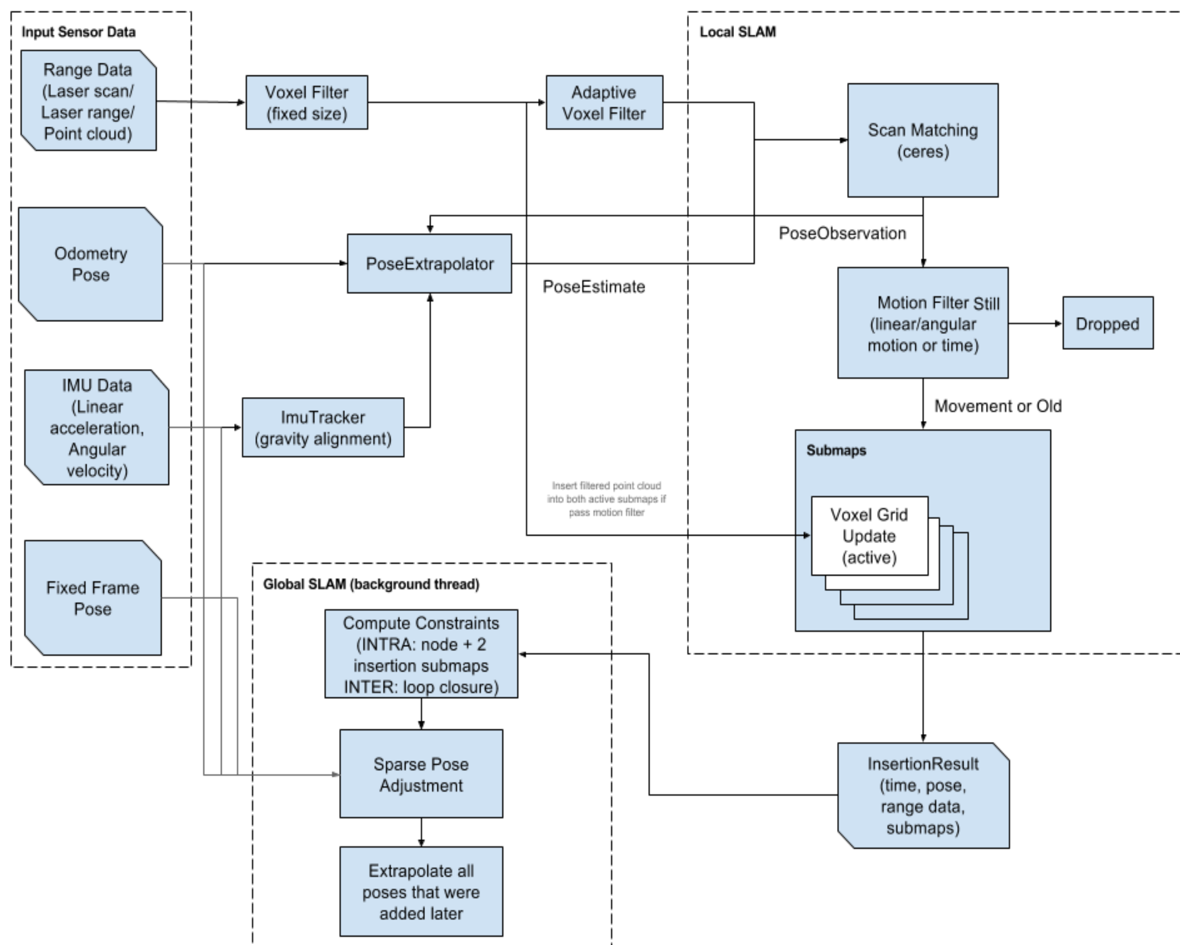
Robot localization is the process of determining where a mobile robot is located with respect to its environment. Localization is a fundamental competency required by an autonomous robot, as the knowledge of the robot's own location is a prerequisite to making decisions about future actions.

SLAM is a computational problem of constructing or updating a map of an unknown environment, while simultaneously keeping track of the robot's location within it. There are many algorithms to approach an optimal solution. Each solution is designed around the sensors that are used to perceive the environment. In our system we use different sensors to find an optimal solution.

[12] *https://www.slamtec.com/en/Lidar/A1*

3.1 SLAM with proximity sensor

One of the most common ways to apply a SLAM algorithm, is with a proximity sensor, especially 360o laser sensors, mostly because they can acquire accurate depth information of an environment in different angles. A common SLAM algorithm used is google cartographer [2], because it provides real time map and robot position data. Below we can see the algorithm's structure.



[13]

Figure 3.1 Google cartographer structure.

[13] https://github.com/cartographer-project/cartographer/blob/master/docs/source/high_level_system_overview.png

Below we explain in depth every component of the algorithm, in order to understand how it works overall.

3.1.1 Sensor input

Proximity sensors, such as lidars, provide depth information in different angles. However, some measurements are not accurate, due to dust or a robot part partially covering the sensor, also, some of the furthest measurements can be a result of reflection or sensor noise, hence they are considered as noise for SLAM. Thus, cartographer applies a bandpass filter and keeps values between a certain range.

The resulting range data are often denser in some angles than others, this happens because objects closer to the sensor provide more points, on the contrary far objects are less often hit and provide less points. The computational weight of points handling needs to be reduced, a solution is to subsample point clouds. A voxel filter is used to down sample raw points into cubes of a certain size and only keeps the centroid of each cube.

Below we can see an example of this filtering procedure.

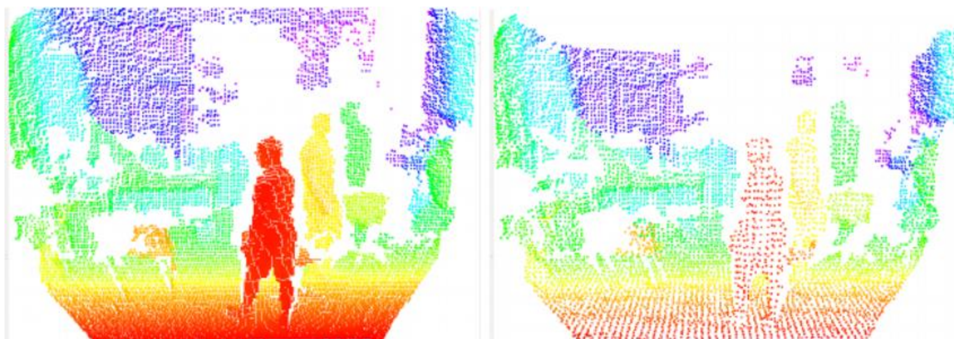


Figure 3.2 The effect of voxel grid filter on 3D.

[14]

Next, an adaptive voxel filter is applied to the data. This filter tries to find out the optimal voxel size to achieve a specific number of points.

[14] *https://www.researchgate.net/figure/The-effect-of-the-voxel-grid-filter-on-Kinect-3D-data_fig3_286624443*

The use of an IMU sensor is optional in 2D SLAM, because the cartographer can handle range data in real time. Although, it provides an accurate direction of gravity and a noisy estimation of the robot’s rotation, which is useful for SLAM overall. In 3D SLAM, IMU usage is mandatory, because it is used to make an initial guess for the orientation of the proximity sensor scans, reducing the computational weight of scan matching. Odometry sensors help the pose extrapolator to make accurate predictions (read below).

3.1.2 Local SLAM

When a scan is produced and filtered, it is handled by the local SLAM algorithm. Local SLAM inserts a new scan into its current submap construction by scan matching using an initial guess from the pose extrapolator.

The scan matching strategy used in this project:

- The scan matcher node takes an initial guess as prior and finds the best spot where the scan match fits the submap. This is done by interpolating the submap and sub-pixel aligning the scan.

Pose extrapolator uses sensor data other than that of range finders to predict where the next scan should be inserted into the submap.

A submap is considered complete when the local SLAM has received a certain amount of range data. Local SLAM drifts over time, to fix this global SLAM is used. Submaps need to be small enough so their drift is below the resolution, but they must be large enough for the loop closure detection to work properly. Loop closure detection is the process of detecting whether an agent has returned to a previously visited location.

3.1.3 Global SLAM

While local SLAM produces submaps, a global optimization task is executed simultaneously. This procedure rearranges submaps in a way that the result is a coherent global map. In its core global SLAM is a pose graph optimization algorithm [3] which works by building constraints between nodes and submaps and then optimizing the resulting constraints graph.

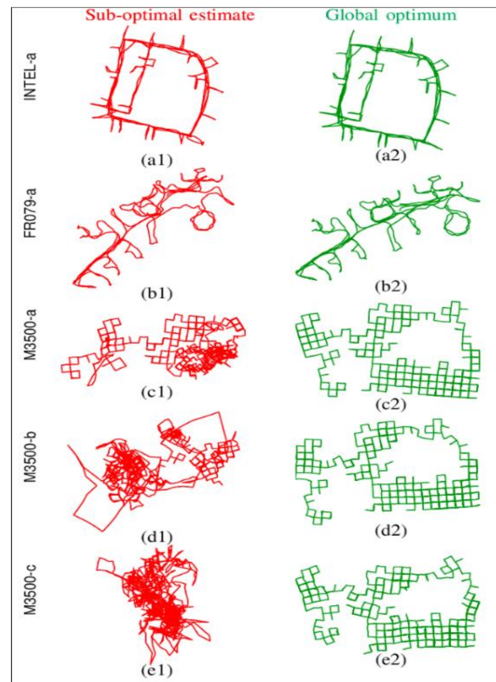


Figure 3.3 Global optimization results.

15

Google cartographer parameters are configured through Lua[16] files, below is shown an example configuration for 2D SLAM with one proximity sensor.

[15] *https://www.semanticscholar.org/paper/Planar-Pose-Graph-Optimization%3A-Duality%2C-Optimal-Carlone-Calafiore/c110a7544c4d48caccc6ec5f0c21c89523e45260/figure/0*

[16] https://en.wikipedia.org/wiki/Lua_(programming_language)

```lua
include "map_builder.lua"
include "trajectory_builder.lua"

options = {

    map_builder = MAP_BUILDER,
    trajectory_builder = TRAJECTORY_BUILDER,
    map_frame = "map",
    tracking_frame = "base_link",
    published_frame = "base_link",
    odom_frame = "odom",
    provide_odom_frame = true,
    use_odometry = false,
    use_nav_sat = false,
    use_landmarks = false,
    publish_frame_projected_to_2d = false,
    num_laser_scans = 1,
    num_multi_echo_laser_scans = 0,
    num_subdivisions_per_laser_scan = 1,
    rangefinder_sampling_ratio = 1,
    odometry_sampling_ratio = 1,
    fixed_frame_pose_sampling_ratio = 1,
    imu_sampling_ratio = 1,
    landmarks_sampling_ratio = 1,
    num_point_clouds = 0,
    lookup_transform_timeout_sec = 0.2,
    submap_publish_period_sec = 0.3,
    pose_publish_period_sec = 5e-3,
    trajectory_publish_period_sec = 30e-3,
}

MAP_BUILDER.use_trajectory_builder_2d = true

TRAJECTORY_BUILDER_2D.use_imu_data = false

TRAJECTORY_BUILDER_2D.use_online_correlative_scan_matching = true

POSE_GRAPH.optimization_problem.huber_scale = 1e2

return options
```

Figure 3.4 Example Lua file.

3.2 Visual SLAM

Visual SLAM algorithm is a solution of the SLAM problem, using camera images or other visual data only. Visual SLAM can be used as a fundamental technology for various types of applications and has been discussed in the field of computer vision, augmented reality and robotics. One implementation of our system uses openVSLAM [4], a visual SLAM algorithm in conjunction with RICOH THETA V camera, in order to achieve autonomous navigation in an unknown environment.

3.2.1 Feature detection

Most Visual-SLAM algorithms localize the system by comparing camera frames and matching key points or features. OpenVSLAM depends on Oriented FAST and Rotated BRIEF (ORB) local feature detector.

3.2.1.1 Features from Accelerated Segment Test (FAST)

Features from Accelerated Segment Test (FAST) is a corner detection method which is used by ORB algorithm. Given a pixel p in an array, FAST compares the brightness of p to a specific number of surrounding pixels that are in a small circle around p. Pixels in the circle are grouped into three classes (lighter, darker or similar to p). If more than a certain number of pixels are darker or brighter than p, it is selected as a key point. Thus, key points produced by FAST help determine edges in an image.

3.2.1.2 Oriented Fast and Rotated BRIEF (ORB)

Features produced by FAST do not have orientation and multiscale features. ORB algorithm utilizes a multiscale image pyramid. Multiscale image pyramid is a representation of a single image at different resolutions. Each level of the pyramid consists of a down-sampled version of the upper-level image, starting from the original image. FAST algorithm is executed on this pyramid to detect key points at each level, essentially locating key points at a different scale. Thus, ORB is partial scale invariant.

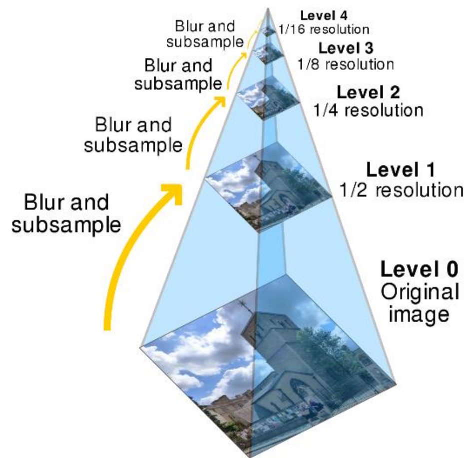


Figure 3.5 Multiscale pyramid.

[17]

The next step is to assign orientation to each key point depending on how the levels of intensity vary around each key point. Intensities are detected by using intensity centroid[18]. The intensity centroid assumes that a corner's intensity is offset from its centre and this vector may be used to impute orientation. Finally Binary Robust Independent Elementary Features [5] (BRIEF) algorithm is used as a feature point descriptor. Next, we discuss how openVSLAM algorithm works, using the feature detection method mentioned above.

3.2.2 Algorithm walkthrough

The algorithm extracts a set of points (features) through successive camera frames and triangulate their 3D position, while simultaneously using this information to estimate camera pose. A map is created and extended using the extracted features and a local bundle [19] adjustment procedure is performed. At the end, global optimization is performed, which includes loop closure detection, global bundle adjustment and pose-graph optimization.

[17] *https://en.wikipedia.org/wiki/Pyramid_(image_processing)*

[18] *https://en.wikipedia.org/wiki/Image_moment*

[19] *https://en.wikipedia.org/wiki/Bundle_adjustment*

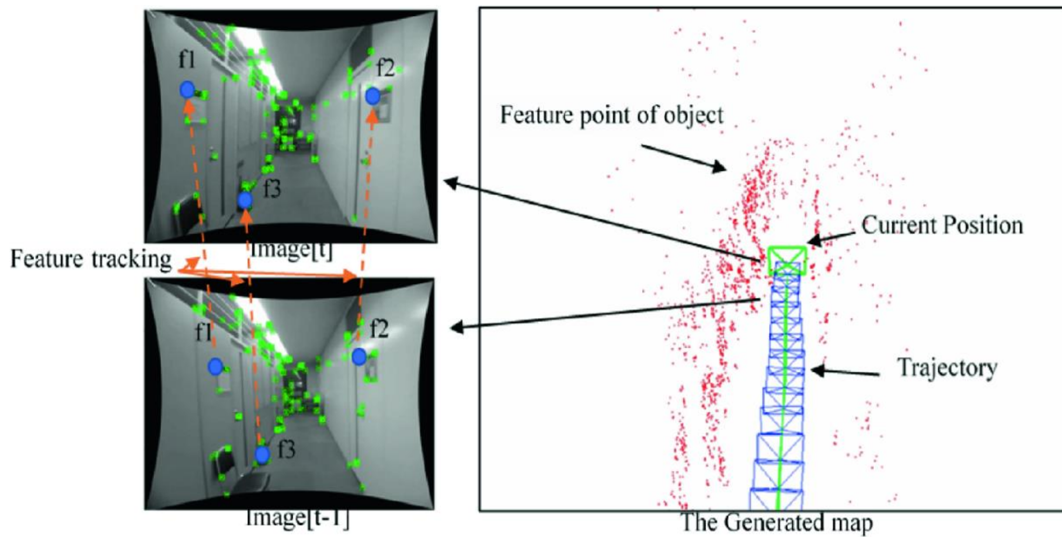


Figure 3.6 Visual SLAM.

[20]

Chapter 4 Autonomous Navigation

Autonomous Navigation can be accomplished in known or unknown environments when a mobile robot is able to localize itself and map the environment (in unknown environments) as we explained above. At this point we must have obtained a map and the robot's position in it. There are different kinds of algorithms for autonomous navigation, ROS provides a software navigation stack[21], a group of packages that includes different algorithms to handle navigation problems. We use and configure them to suit our system.

4.1 move_base package

The package we use is called move_base[22], it offers among others, algorithms for path planning (goal position is defined by the user) and obstacle avoidance in a map. Every algorithm that is

---

[20] *https://link.springer.com/chapter/10.1007/978-981-15-3651-9_5*

[21] http://wiki.ros.org/navigation

[22] http://wiki.ros.org/move_base

used is configured via yaml[23] files. Below is shown a yaml example file for a cost map algorithm we use (read below).

```yaml
global_costmap:
  update_frequency: 2.5
  publish_frequency: 2.5
  transform_tolerance: 0.5
  width: 15
  height: 15
  origin_x: -7.5
  origin_y: -7.5
  static_map: false
  rolling_window: true
  inflation_radius: 2.5
  resolution: 0.1
  global_frame: /map
  robot_base_frame: base_link
```

Figure 4.1 Global cost map configuration file.

The navigation algorithm structure is shown below.

[23] https://en.wikipedia.org/wiki/YAML

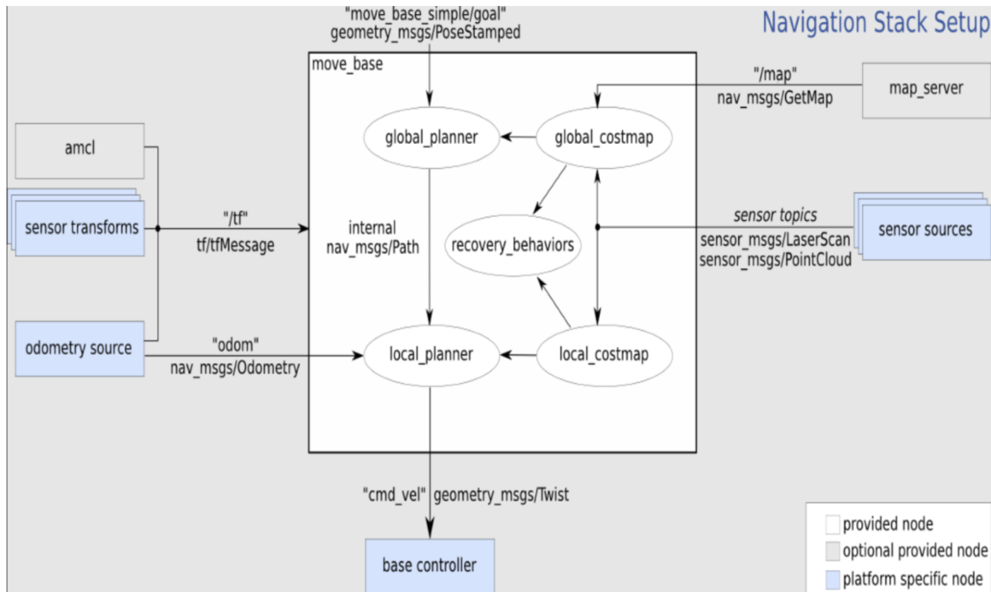


Figure 4.2 move_base set of nodes.

4.1.1 Global, local cost-map

Cost-maps are essentially grid maps, where each cell has a certain value (cost). Larger cost values indicate a greater possibility that an obstacle is in a certain position. Similarly, smaller cost values suggest less or zero possibility of an obstacle existence in a certain position. In our system's cost-map there is an inflation layer[24]. Inflation is the process of propagating cost values out from occupied cells that decrease with distance. This layer tries to help the robot keep a certain (adjustable by the user) distance from obstacles. Below an example of a local and a global cost map is shown.

[24] http://wiki.ros.org/costmap_2d/hydro/inflation

Figure 4.3 Global cost map. Black cells are inflated objects, green rectangle is the robot and the red arrow represents its orientation.

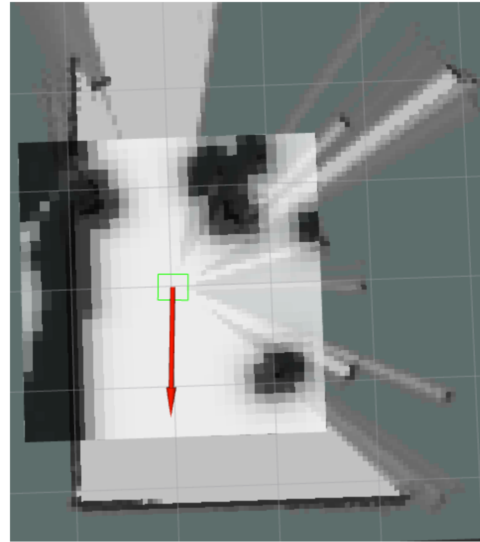


Figure 4.4 local cost map aligned with a normal map.

4.1.2 Path planning

Path planning algorithms, calculate a valid sequence of positions, that a robot must follow, in order to reach a (user defined) destination. We use Dijkstra's[25] algorithm in our implementation.

4.1.2.1 Dijkstra's algorithm

Dijkstra's algorithm is used to find the shortest path between nodes[26] in a graph, which in our case represents the map of our environment. This algorithm finds the shortest path between a source node and every other node in the graph, meaning that the path to the goal node is an optimal one. Although the map of our environment is a 2D grid, we can treat it as a graph and each grid square is connected to up to 8 adjacent grid cells.

---

[25] *https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm*

[26] *https://en.wikipedia.org/wiki/Vertex_(graph_theory)*

1	2	3
8	s	4
7	6	5

Figure 4.5 Starting cell is "s", adjacent cells are considered neighbors.

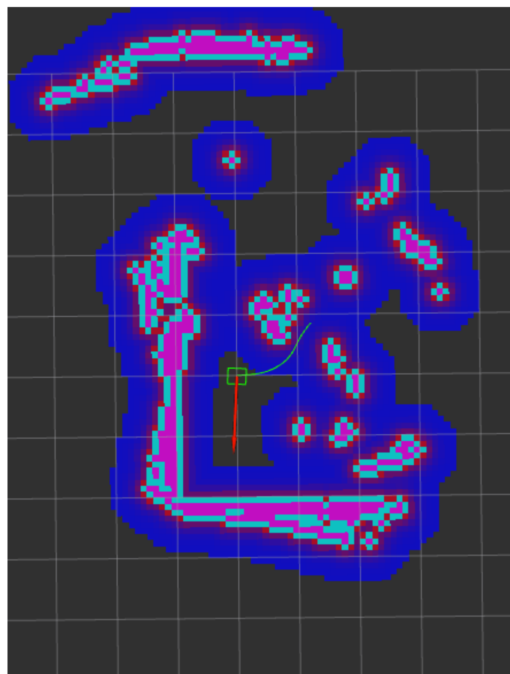


Figure 4.6 Red and pink cells represent obstacles, blue cells represent the inflation of obstacles, green cells represent the path.

In Figure 4.6 a visualization of the robot in a 2D cost map is shown, the red arrow represents the pose of our robot, and the green rectangle represents its size. We set a navigation command through rviz for the robot to execute, using Dijkstra's algorithm the path generated (green line) is an optimal one, considering all the information a cost map provides.

4.1.3 Recovery behaviour

State or behaviour recovery algorithm is a procedure a robot follows when it is not able to follow a user's command. For example, reach a certain goal position when it is surrounded by obstacles, or the goal position is inside an obstacle. In this project, we will not handle these situations.

4.2 Algorithm walkthrough

Move base uses two cost maps and two path planning algorithms, one global and one local of every kind (see Figure 4.2). The global cost map in conjunction with the global planner, optimize the complete path from a starting point until the goal point. Local cost map and local planner, optimize the path near the robot. The combination of the global and local cost maps and planners, find an optimal path, for a given goal in a map. The algorithm produces velocity messages, that are handled by the base controller.

4.2.1 Base controller

The base controller in our system is essentially a node which receives velocity messages, that include a linear speed in x-axis and an angular speed in z-axis. This node calculates speed for the wheels and sends a pulse-width modulation[27] (PWM) signal to a motor driver, which also includes GPIO pins[28] to control direction for each wheel. This motor driver decodes the signal and delivers certain power to the motors, so they can rotate at the desired speed.

Our robot is a differential wheeled robot, which means that it can change its direction by varying the relative rate of rotation of its wheels and hence does not require a steering motion. Below we can see a pseudocode of our implementation.

---

[27] *https://en.wikipedia.org/wiki/Pulse-width_modulation*

[28] *https://en.wikipedia.org/wiki/General-purpose_input/output*

```
//define which PINS are going to be used
#define GPIO_PINS ...
#define PWM_PINS ...

//define the distance between the wheels of the robot, for example width = 0.250
#define ROBOT_BASE_WIDTH 0.250

function init()
    initialize(GPIO_PINS)
    initialize(PWM_PINS)

end function


function command_velocity_callback(velocity_msg)

    //Our robot is differential drive

    //calculate speed for right and left wheels
    right_wheels_speed = velocity_msg.linear.x + (velocity_msg.angular.z * ROBOT_BASE_WIDTH / 2)
    left_wheels_speed = velocity_msg.linear.x - (velocity_msg.angular.z * ROBOT_BASE_WIDTH / 2)

    //normalize speed in range 0-100 (%)
    normalize(right_wheels_speed)
    normalize(left_wheels_speed)

    //calculate direction for each side of wheels
    left_wheels_direction = find_direction(left_wheels_speed)
    right_wheels_direction = find_direction(right_wheels_speed)

    //set the directions
    write_left_wheels_GPIO_PINS(GPIO_PINS, left_wheels_direction)
    write_right_wheels_GPIO_PINS(GPIO_PINS,right_wheels_direction)

    //set the right amount of speed through Pulse-width modulation (PWM) signal
    write_left_wheels_PWM_PINS(left_wheels_speed)
    write_right_wheels_PWM_PINS(right_wheels_speed)

end function
```

Figure 4.7 Motor control pseudocode.

The motor driver used is shown below.

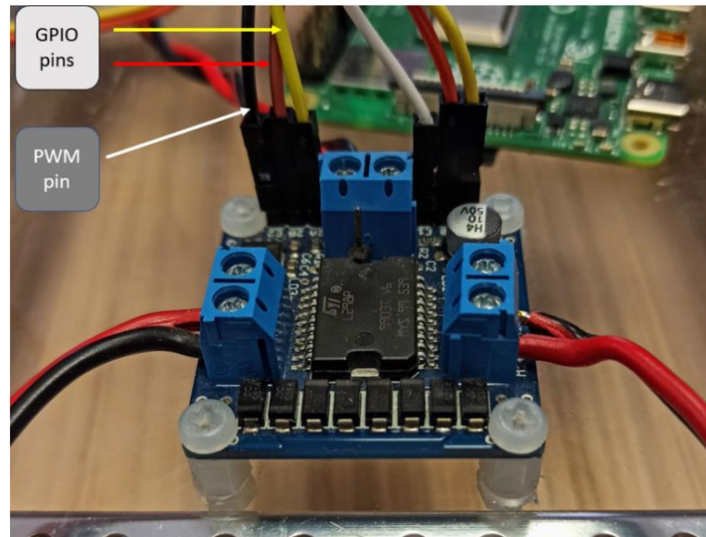


Figure 4.8 Motor driver.

4.3 Sensor fusion and Kalman filter

Nowadays, many autonomous mobile robots use more than one sensor, in order to navigate and do tasks independently. The acquired sensor data can be used together to gather more information about our system and the surroundings, thus a procedure called sensor fusion is introduced to solve this sensor combination problem.

Sensor fusion is a procedure, where two or more data sources are combined in a way that generates a more consistent, accurate and dependable understanding of the system, than it would be from a single data source. Kalman filters are utilized in our system to fuse different kinds of information. Kalman filter is an estimator for what is called the linear-quadratic problem, which is the problem of estimating the instantaneous "state" by using measurements linearly related to the state but corrupted by white noise. The resulting estimator is statistically optimal with respect to any quadratic function of estimation error. Kalman filter has numerous applications and specifically extended Kalman filters. Extended Kalman filter handles nonlinearity by linearizing the system at the point of current estimate and then the linear Kalman filter is used to filter this linearized system. An application of a Kalman filter is the fusion of different sensors of a robot, in order to estimate its position and predict the next position. In our system implementation, Kalman filter is used to combine odometry and IMU data [6].

4.3.1 Basic Terms

A robot's position uncertainty can be expressed with a Gaussian distribution $N(x, \sigma^2)$. For example, we believe the robot's x-axis position is $x = 10m$ and the variance is $1\ m^2$ or $N(10,1)$.

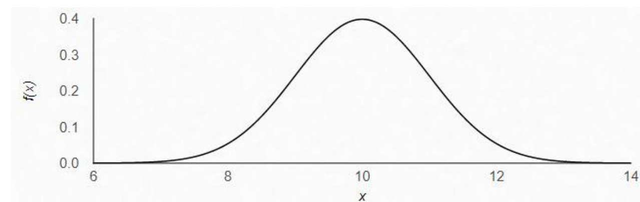


Figure 4.9 Gaussian distribution $N(\chi, \sigma^2)$.

The plot shows a probability density function which indicates every probability of the robot's position in a range of values between $x = 9m\ to\ x = 11m$.

In multivariate Kalman filters it is essential to consider how the multiple variables are correlated and how much is their covariance. Covariance describes how two variables change in relation to each other. A covariance of 0 indicates no correlation between the variances of the variables.

A covariance matrix between two variables x, y looks like:

$$
\begin{array}{cc} & \begin{array}{cc} x & y \end{array} \\ \begin{array}{c} x \\ y \end{array} & \begin{bmatrix} var(x) & cov(x,y) \\ cov(x,y) & var(y) \end{bmatrix} \end{array}
$$

Figure 4.10 Covariance matrix.

The diagonal contains the variance of each variable, and the off-diagonal elements represent the covariance between variables.

4.3.2 Kalman filter algorithm

The Kalman filter can be broken down to three stages [7]:

<u>Initialization</u>

1. Initialize the state of the filter.
2. Initialize our belief in the state.

<u>Predict</u>

1. Use process model to predict state at the time step.
2. Adjust belief to account for the uncertainty in prediction.

<u>Update</u>

1. Get a measurement and associated belief about its accuracy.
2. Compute residual between estimated state and measurement.
3. Compute scaling factor (Kalman gain) on whether the measurement or prediction is more accurate.
4. Set state between the prediction and measurement based on scaling factor.
5. Update belief in the state based on how certain we are in the measurement.

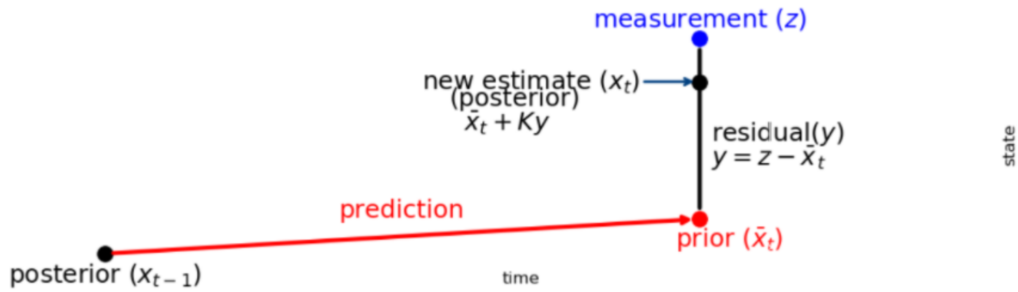


Figure 4.11 Kalman filter.

Below we examine the multivariate Kalman filter equations.

Prediction

1. $\bar{x} = Fx + Bu$
2. $\bar{P} = FPF^T + Q$

Update

1. $y = z - H\bar{x}$
2. $K = PH^T(HPH^T + R)^{-1}$
3. $x = \bar{x} + Ky$
4. $P = (I - KH)P$

The variables are defined below:

- x, P are the state mean and covariance
- F, Q are the state transition function and process covariance
- B, u are the control matrix and control input
- H is the measurement function
- z, R are the measurement mean and noise covariance
- y, K are the residual and the Kalman gain

Chapter 5 System integration

We are using a custom rover platform. It uses 4 independent dc motors and a custom battery pack for the motors. The main computer is raspberry pi 4B, which also has its own battery pack.

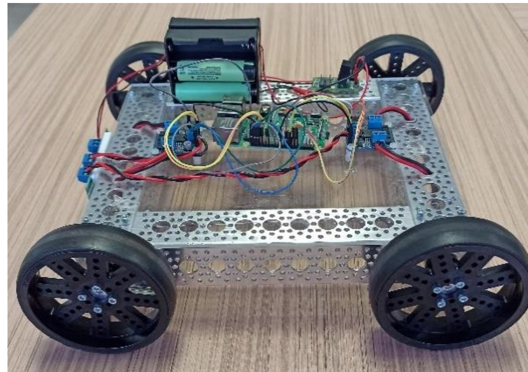


Figure 5.1 Robot platform.

5.1 LiDAR implementation

In this implementation the only sensor used is RP-lidar A1M8. The cartographer is used to create a map of the environment and localize the robot in it and the move base package is used to navigate.

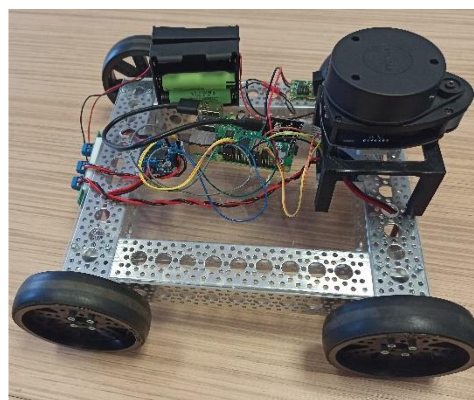


Figure 5.2 System with LiDAR sensor.

Through rviz we manage to visualize and send navigation commands to the robot from a computer station. The system successfully navigated in the testing environment and avoided obstacles. The accuracy achieved is ±5 cm in an unknown environment.

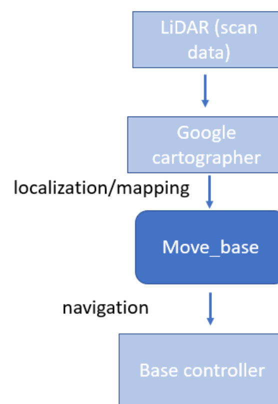
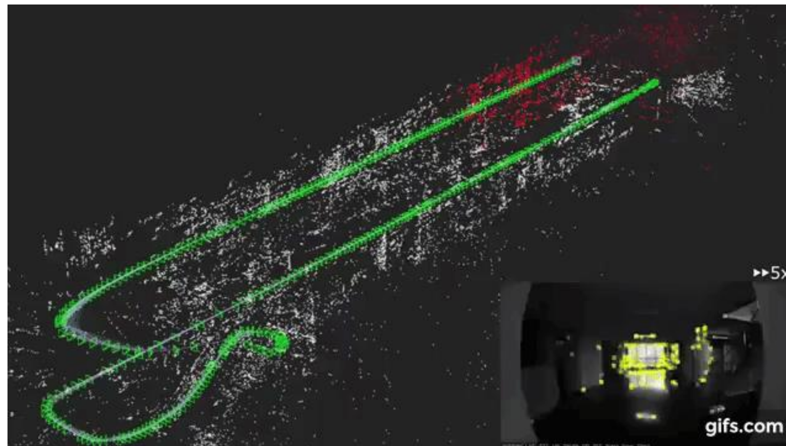


Figure 5.3 System structure using LiDAR sensor.

5.2 RICOH THETA implementation

Figure 5.4 System with 360o camera.

In this implementation, our goal is to test the accuracy of a visual SLAM algorithm. RICOH theta V is used as the only sensor for our system and OpenVSLAM is configured to work under ROS framework. Move base package is used for the navigation part along with octomap_ros[29]. At this point the robot can navigate to an environment without obstacles. Below is shown an example of a map created by openvSLAM and the robot's path in it.



[30]

Figure 5.5 Map created by openVSLAM, white cells are features, green cells represent the path followed.

OpenVSLAM produces point-clouds and octomap_ros package transforms them into a map format. We filtered the feature points the algorithm produced by keeping only those in a specified height range. This filtering procedure is important, because we want to discard features that occur from the floor (not actual obstacles) and features that occur above the robot's height (there is no chance of collision if an actual object exists above the robot's height). Below we can see the system's structure.

[29] http://wiki.ros.org/octomap

[30] *https://camo.githubusercontent.com/c17cd8535ea4ee954ffaebf923c153cfcebedfae99a5edad81fb768df8d13226/68747470733a2f2f6a2e676966732e636f6d2f38316d31514c2e676966*

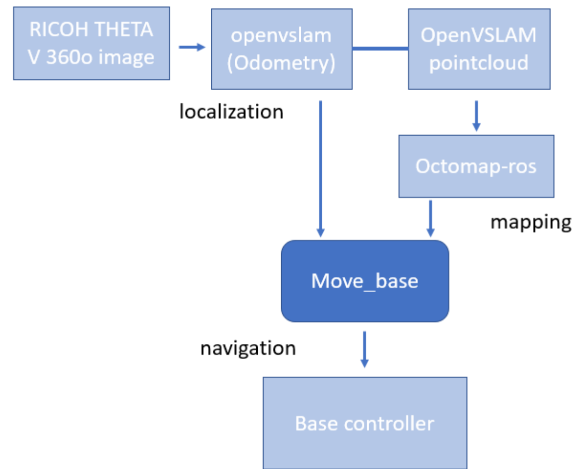


Figure 5.6 Each rectangle represents a node in the ROS system.

Below is shown a pseudocode of our SLAM subsystem when using RICOH THETA V camera.

```
function equirectangular_camera(...)

    system SLAM
    SLAM.initialize()

    //fetch live streaming from camera
    video = VideoCapture()


    ROS.Publisher pose_publisher.initialize()
    ROS.Publisher point_cloud_publisher.initialize()

    // run SLAM algorithm in another thread
    thread()

        while(is_not_end)

            video.read(frame);
            camera_pose = SLAM.feed_frame(frame)

            point_cloud = SLAM.get_landmarks()

            loop counter from 0 to point_cloud.size

                // obstacle above robot
                if point_cloud[counter].height > robot.height
                    delete point_cloud[counter]

                //obstacle below robot's half wheel radius, robot can override it
                else if point_cloud[counter].height < robot_wheel.radius/2
                    delete point_cloud[counter]

            end loop
        end while

            pose_publisher(camera_pose)
            point_cloud_publisher(point_cloud)

    end thread

    // user can choose whether to visualize cameras live streaming or not
    if(visualize== True)
        viewer.run()

    wait(thread)
    SLAM.shutdown()
end function
```

Figure 5.7 Pseudo code of the SLAM implementation and filtering procedure.

Unfortunately, it is not possible to create an accurate map of the environment with this data, because the depth information of the camera is inaccurate. This leads to the map shown below.

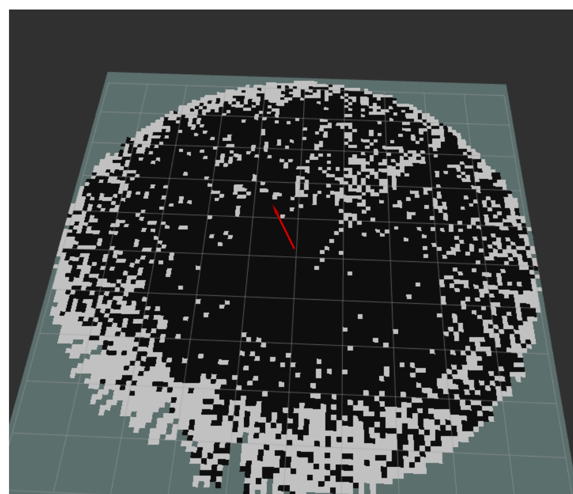


Figure 5.8 Black cells represent obstacles, white cells represent free space, red arrow represents the robot's pose.

5.3 Intel real-sense implementation

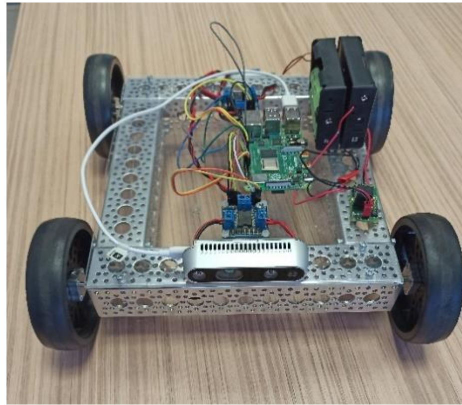


Figure 5.9 Rover with IntelRealSense D435i camera.

This time a different type of camera is tested, IntelRealSense D435i. At first, RGB and depth images are extracted from the camera and passed to the openVSLAM algorithm, which is configured to handle RGB-D images. IntelRealSense software development kit provides tools to access and handle images of the camera, so instead of using ROS functionalities to fetch and process images, we wrote code and utilized the development kit, to process images faster and more efficiently. Below we can see a pseudocode of the SLAM algorithm implementation with IntelRealSense.

```
function RGB-D_camera(....)

    // create a communication bus with IntelRealSense
    rs2::pipeline pipe
    pipe.start()

    system SLAM
    SLAM.initialize()

    ROS.Publisher pose_publisher.initialize()
    ROS.Publisher point_cloud_publisher.initialize()

    // run SLAM algorithm in another thread
    thread()

        while(is_not_end)

            // wait for next set of frames from camera
            data = pipe.wait_for_frames()
            // align depth image to RGB image
            aligned_images= align(data)

            camera_pose = SLAM.feed_frame(aligned_images.rgb_image,aligned_images.depth_image)

            point_cloud = SLAM.get_landmarks()

            loop counter from 0 to point_cloud.size

                // obstacle above robot
                if point_cloud[counter].height > robot.height
                    delete point_cloud[counter]

                //obstacle below robot's half wheel radius, robot can override it
                else if point_cloud[counter].height < robot_wheel.radius/2
                    delete point_cloud[counter]

            end loop
        end while

            pose_publisher(camera_pose)
            point_cloud_publisher(point_cloud)

    end thread

    // user can choose whether to visualize cameras live streaming or not
    if(visualize== True)
        viewer.run()

    wait(thread)
    SLAM.shutdown()
end function
```

Figure 5.10 Pseudo code of the SLAM system, using IntelRealSense D435i camera.

Again, the data (features) were filtered, and we kept those in a specified range of height.

Although, the system can localize itself in slow change of movements and create a map, it is very susceptible to quick turns and accelerations, thus the system loses track of the robot' position and the robot must return to the previous position, or the algorithm needs to be reset. This problem is caused by the narrow field of view of the camera [8] and the possible dynamic change of a featureless environment.

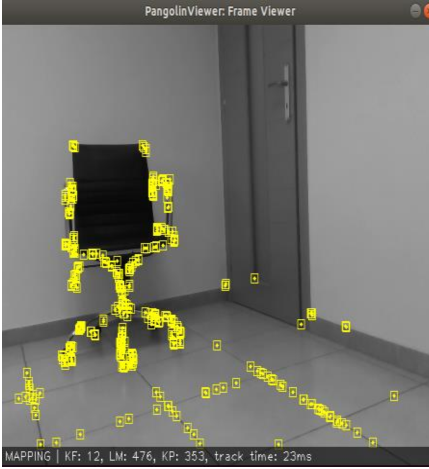


Figure 5.11 Yellow cells are features, algorithm is mapping.

Figure 5.12 Camera is turned, and algorithm is lost.

Figure 5.13 The chair is moved out of the field of view, camera is back on the starting position and the algorithm is still lost.

Figure 5.14 Picking the chair and moving it.

Figure 5.15 Algorithm still cannot match the features.

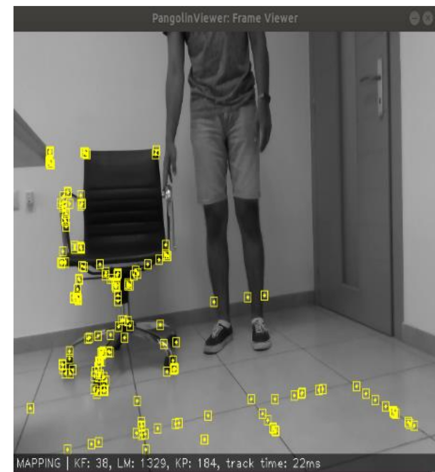


Figure 5.16 Chair back on starting position, algorithm tracks the features again.

A second experiment was conducted, with google cartographer package. Depth images are extracted from the camera and then processed by depthimage_to_laserscan [31] package. It essentially receives 3D depth images and extracts 2D distance information about objects, that are in the camera's field of view.

[31] http://wiki.ros.org/depthimage_to_laserscan

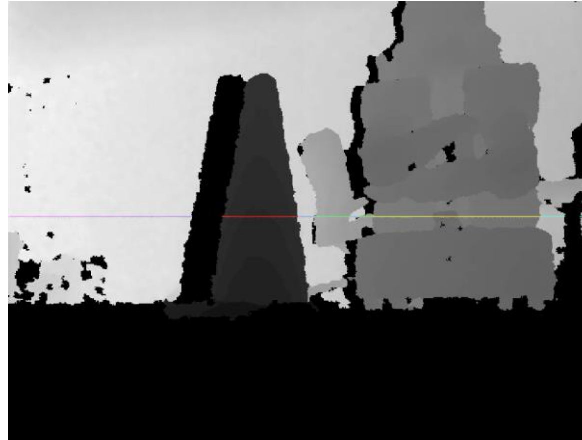


Figure 5.17 Different colors of the line represent different distances from the camera.

The 2D information needed is now extracted and passed into the cartographer along with the IMU data. The algorithm cannot perform and loses track of the robot. Submaps cannot be created correctly, because the IMU information alone cannot produce an accurate guess of the robot's position over time and the global SLAM fails to create a coherent map.

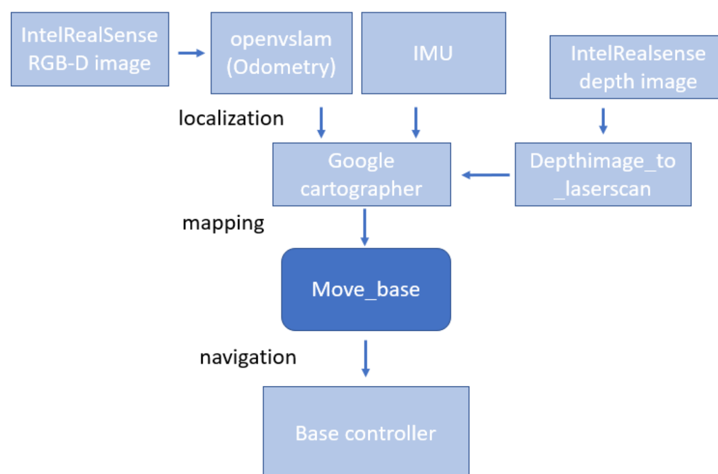


Figure 5.18 System structure using only IntelRealSense D435i.

5.4 Multicamera system

The last system implemented, consists of two cameras, Intel RealSense D435i and RICOH THETA V.

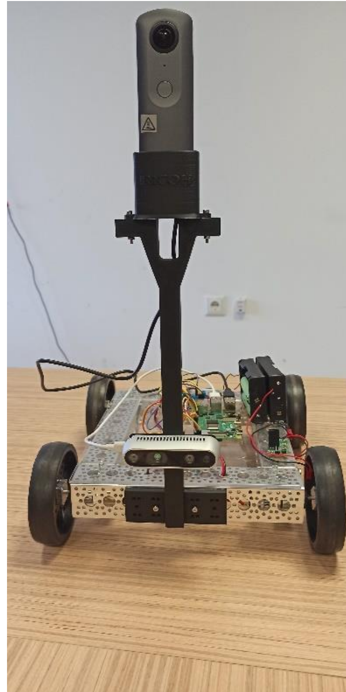


Figure 5.19 Multicamera system

Both cameras are strategically mounted on the robot, in order to get the most out of the available visual information. RICOH THETA V along with openVSLAM algorithm, are used to localize the robot and provide odometry only, we do not save a map or search for loop closure in it. Below is shown an odometry information message coming from openVSLAM.

```
header:
  seq: 1378
  stamp:
    secs: 1621592782
    nsecs: 442026911
  frame_id: "odom"
pose:
  position:
    x: -0.02173168484
    y: 0.0142315850346
    z: 0.000243609486757
  orientation:
    x: -0.00136110990298
    y: -0.00206709021039
    z: 0.162004940183
    w: 0.986786843181
---
```


Figure 5.20 Position and orientation coordinates of odometry.

Intel RealSense depth images are extracted and processed again by depthimage_to_laserscan package, in the end we keep 2D depth information about the environment and the built-in IMU's data. At this point, we have localized the robot and have depth information that can be used to construct a map. Google cartographer algorithm receives the above data and creates a map. The next step is to navigate and explore the area, move_base package is utilized, it receives map messages from cartographer and the robot's position in it.

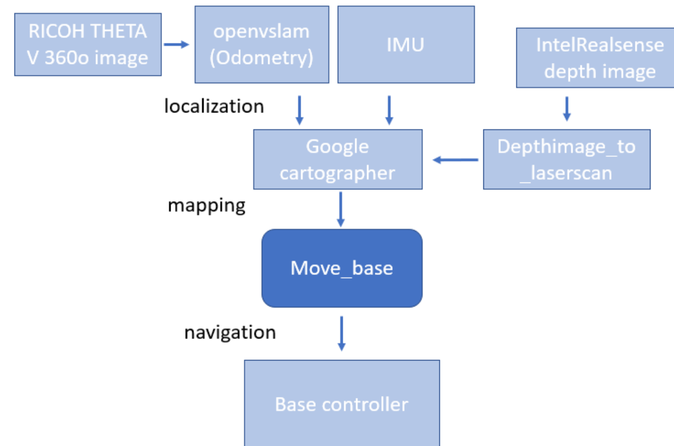


Figure 5.21 System structure using two cameras.

Our system can navigate in unknown environments and perform object avoidance at this point. The position accuracy achieved is $\pm 4\ cm$ and orientation accuracy $\pm 15\ degrees$.

Below we can see a map, that was produced during experiments.

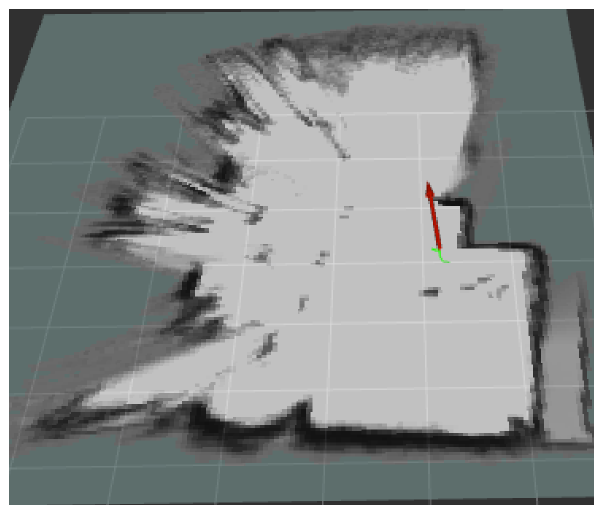


Figure 5.22 Red arrow represents the orientation and position of the robot in the map.

The outliers far away are not constructed very well, because the camera's range is constrained to 5 meters and the objects in those ranges are partially or not often hit by the camera. Finally, we can fetch the live streaming of the camera, while navigating and have a first-person view (FPV).

5.4.1 Kalman filter integration

Although the accuracy our system achieved is very good, we tried to increase it a bit more. All components are the same in this implementation, but we used one more algorithm in order to fuse odometry and IMU information. As we mentioned in a previous chapter (see Chapter 3, paragraph 3.3) an extended Kalman Filter is used to reduce drift from our sensors and increase odometry accuracy. The filter is modified to fuse information in the 2D space, odometry measurements are considered relative to the first measurement which is the starting position ($x = 0$, $y = 0$, $z = 0$, $roll = 0$, $pitch = 0$, $yaw = 0$, although z-axis and $roll$, $pitch$ measurements are not used) and IMU measurements are relative to the previous measurements, to clarify an IMU measurement at time t is considered relative to the measurement at time $t - 1$. This setting is useful because we could not set the variances of the input sources precisely, which means that the measurements may get out of sync with one another and cause oscillations in the filter. Below is shown our system's structure.

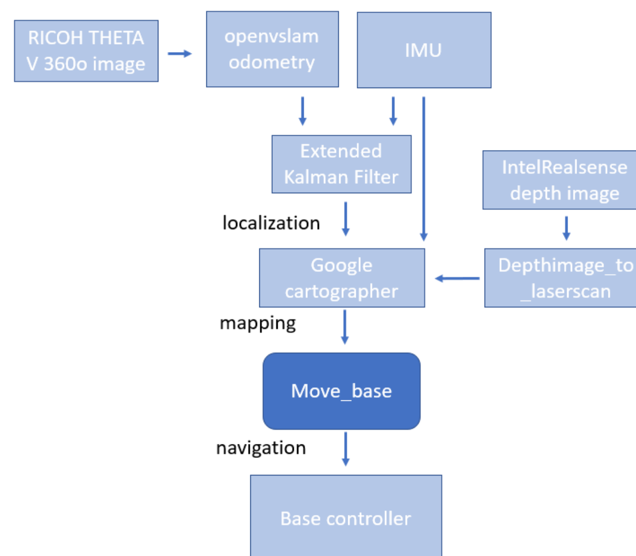


Figure 5.23 System structure including an Extended Kalman Filter.

Experiments with this structure produced better results, the position accuracy achieved is $\pm 4\ cm$ and orientation achieved is $\pm 10\ degrees$. Position accuracy is similar to the previous

experiment, although the tools used to calculate position drift cannot provide accurate information in a millimetre scale. Orientation of the robot has a very noticeable improvement, which is expected because IMU orientation information can help smooth the errors of odometry orientation.

Chapter 6 Conclusions & Discussion

6.1 Summary

Integration of many cameras in mobile robots is very common and challenging, our system can act and navigate autonomously with the information provided by two different cameras. This is achieved with the architecture introduced in the previous chapters. Based on the experiments conducted, the system can act and make decisions in a relative short amount of time. This was one of the main goals, because the system was initially designed to handle real time situations. Furthermore, the use of a 360o cameras, is what makes the system special, because there are many applications that can be developed around them.

6.2 Discussion

The introduced system can be optimized in many ways. As a graduate project thesis, a big part of the system is implemented, as we described in the previous chapters. However, there are different aspects of the system that can be optimized or even change. Below we present some of those open issues.

6.2.1 Implement parts of the system without ROS

ROS uses data streams through TCP sockets along with the publish/subscribe pattern for the inter-Node communication, which is useful in many predicaments like communication between distributed computers. When it comes to fetching data from sensors or algorithms, that do not need to be shared with other computers, the communication architecture mentioned above is redundant and sometimes it is a computational burden to apply in a single computer. Thus, raw code can be written to fetch and transport data intra-process or inter-Node, similar example are nodelets.

6.2.2 Hardware upgrade

Most of our experiments were conducted with a single main control unit, raspberry pi 4B (rpi4B). The system worked fine until the 360o camera was connected and the Visual SLAM algorithm was executed, then we realised a more powerful computer was needed, the multicamera system was tested on a more powerful computer with Intel Core i5[32] processor. This problem can be addressed by using mesh computing [], rather than having a single control unit.

6.2.3 3D SLAM

Our system was configured to navigate and create a map of a 2D environment, even though we obtained 3D images from IntelRealsense D435i camera and the google cartographer can be configured to handle 3D data, our computers could not process all data in real time, due to the problem mentioned in the previous paragraph. To clarify, the computers we had available during the development and testing of the system, could not handle the computational weight of 3D data processing, so we were restricted to the 2D implementation.

6.3 Future work

6.3.1 Virtual Reality teleoperation

The next step is to combine teleoperation features on the existing platform. We have already succeeded in navigating a rover through a Virtual Reality (VR) environment by using VR controllers and produce a 360o field of view live streaming for the user (RICOH THETA V camera). Now we want the user to experience autonomous navigation in a first-person view and be able to send goal positions in a map or even intervene in the navigation procedure while the robot is acting on its own.

[32] https://www.intel.com/content/www/us/en/products/details/processors/core/i5.html

6.3.2 Object detection

Systems with multiple cameras produce images from different angles in order to accomplish difficult tasks. One future goal for our system is to provide an object detection feature using one of the deployed cameras while performing autonomous navigation, in order to produce better information about the surrounding environment and be able to handle tougher missions.

Bibliography

[1] Quigley, Morgan & Conley, Ken & Gerkey, Brian & Faust, Josh & Foote, Tully & Leibs, Jeremy & Wheeler, Rob & Ng, Andrew. (2009). ROS: an open-source Robot Operating System. ICRA Workshop on Open Source Software. 3.

[2] W. Hess, D. Kohler, H. Rapp and D. Andor, "Real-time loop closure in 2D LIDAR SLAM," *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 2016, pp. 1271-1278, doi: 10.1109/ICRA.2016.7487258.

[3] Carlone, L., Aragues, R., Castellanos, J. A., & Bona, B. (2014). A fast and accurate approximation for planar pose graph optimization. *The International Journal of Robotics Research*, *33*(7), 965–987. https://doi.org/10.1177/0278364914523689

[4] Shinya Sumikura, Mikiya Shibuya, and Ken Sakurada. (2019). OpenVSLAM: A Versatile Visual SLAM Framework. In Proceedings of the 27th ACM International Conference on Multimedia (MM '19). Association for Computing Machinery, New York, NY, USA, 2292–2295. https://doi.org/10.1145/3343031.3350539

[5] Calonder M., Lepetit V., Strecha C., Fua P. (2010) BRIEF: Binary Robust Independent Elementary Features. In: Daniilidis K., Maragos P., Paragios N. (eds) Computer Vision – ECCV 2010. ECCV 2010. Lecture Notes in Computer Science, vol 6314. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-15561-1_56

[6] Moore T., Stouch D. (2016) A Generalized Extended Kalman Filter Implementation for the Robot Operating System. In: Menegatti E., Michael N., Berns K., Yamaguchi H. (eds) Intelligent Autonomous Systems 13. Advances in Intelligent Systems and Computing, vol 302. Springer, Cham. https://doi.org/10.1007/978-3-319-08338-4_25

[7] Labbe, Roger. Kalman and Bayesian Filters in Python. 2014 : GitHub.

[8] K. Chappellet, G. Caron, F. Kanehiro, K. Sakurada and A. Kheddar, "Benchmarking Cameras for Open VSLAM Indoors," 2020 25th International Conference on Pattern Recognition (ICPR), 2021, pp. 4857-4864, doi: 10.1109/ICPR48806.2021.9413278.