TREBALL FINAL DE GRAU

TÍTOL: Obstacle Avoidance for an Autonomous Rover

TITULACIÓ: Grau en Enginyeria d'Aeronavegació

AUTOR: Marc Chesa Roldàn

DIRECTOR: David Calero Scanlan

SUPERVISOR: Oscar Casas Piedrafita

DATA: 23 de juny del 2017

Títol: Obstacle Avoidance for an Autonomous Rover

Autor: Marc Chesa Roldàn

Director: David Calero Scanlan

Supervisor: Oscar Casas Piedrafita

Data: 23 de Juny de 2017

Resum

L'objectiu d'aquest projecte es millorar un rover autònom indoor donant-li capacitat per a evadir obstacles.

El rover utilitzat com a punt de partida, anomenat *Gauss*, es una plataforma robòtica amb quatre rodes, motors, un controlador de motor, un microcontrolador i sensors de localització i percepció. *Gauss* fa servir tot aquest hardware per proporcionar mòduls de control, localització, planificació de ruta i detecció d'obstacles en 2D. El rover resultant després de la millora, anomenat AROA (Autonomous Rover for Obstacle Avoidance), ha afegit nou hardware al sistema i ha millorat l'arquitectura software.

El nou hardware implementat al robot consisteix en una càmera RGB-D (Red-Green-Blue Depth), un microprocessador, un regulador de tensió i una bateria. La càmera RGB-D es el nou sensor de percepció, que permet detecció d'obstacles en 3D. La bateria i el regulador s'utilitzen per proveir i distribuir potencia a aquest nou sistema. Una selecció de components s'ha realitzat per tal d'escollir el nou hardware entre les diferents opcions del mercat.

L'arquitectura software ha estat renovada i ara està basada en ROS (Robot Operating System). S'han migrat els mòduls de planificació de ruta i detecció d'obstacles al nou microprocessador i dissenyat un algoritme d'evasió d'obstacles. El mòduls de localització i control s'han mantingut al microcontrolador i han estat adaptats per treballar amb ROS.

La càmera RGB-D, Kinect v2, ha estat calibrat per tal garantir una bona detecció d'obstacles. El procés consisteix en un calibratge geomètric de les dues càmeres que conté el sensor, i d'un calibratge de profunditat.

Per últim, per verificar els diferents mòduls, s'han realitzat diferents tests. S'han avaluat el mòdul de planificació de ruta, el de detecció d'obstacles i el de evasió d'obstacles.

Title: Obstacle Avoidance for an Autonomous Rover

Author: Marc Chesa Roldàn

Director: David Calero Scanlan

Supervisor: Oscar Casas Piedrafita

Date: June 23rd 2017

Overview

This project objective is to improve an indoor autonomous rover with obstacle avoidance capability.

The rover used as starting point, called *Gauss*, is a robotic platform with four wheels, motors, a motor controller, a microcontroller and localization and perception sensors. *Gauss* uses all of this hardware to provide control, localization, path planning and 2D obstacle detection modules. The resulting rover after the improvement, called AROA (Autonomous Rover for Obstacle Avoidance) has added new hardware to the system and improved the software architecture.

The new hardware implemented to the robot consists on a RGB-D (Red-Green-Blue Depth) camera, a microprocessor, a voltage regulator and a battery. The RGB-D camera is the new perception sensor that allows for 3D obstacle detection. The battery and the voltage regulator are used to provide and distribute power supply to this new system. A component selection has been made to choose the new hardware among the different options in the market.

The software architecture has been renewed to be ROS (Robot Operation System) based. The path planning, obstacle detection modules have been migrated to the new microprocessor, and an obstacle avoidance module has been designed. The localization and control modules have been maintained at the *Gauss* microcontroller and have been adapted to work with ROS.

The RGB-D camera, Kinect v2, has been calibrated to guarantee a good obstacle detection. The process consist on a geometrical calibration of the two cameras on the sensor, and a depth calibration.

To verify the different modules, different test have been performed. Path planning, obstacle detection and obstacle avoidance modules have been evaluated.

ACKNOWLEDGMENTS

I want to thank David Calero Scanlan, my tutor form the CTTC, for all his support and patience while guiding me during this project.

I also want to thank Oscar Casas Piedrafita, my project supervisor form the UPC, for offering me the opportunity to do this project.

I also want to thank all the people at the Geomatics department of the CTTC, but specially Enric Fernandez Murcia and Eduard Angelats Company for their proposals and revisions of this memory.

Finally, I would also thank my family for their moral support me during all the process, but especially I want to thank Irene Radcliffe for her support in the hardest moments.

TABLE OF CONTENTS

CHAPTER 1. INTRODUCTION 1

1.1. Project Scope 1

1.2. Document overview 1

CHAPTER 2. STATE OF THE ART 3

2.1. General view on autonomous rovers 3

2.2. Sensors for autonomous rovers 5
2.2.1. Depth perception sensors 5
2.2.2. Localization sensors 7

2.3. Obstacle avoidance techniques 9

CHAPTER 3. SYSTEM DESIGN 13

3.1. System Requeriments 13

3.2. Gauss Design 13

3.3. AROA Design 17

CHAPTER 4. HARDWARE DESIGN 21

4.1. Component selection 21
4.1.1. 3D Range finder 21
4.1.2. Microprocessor 23
4.1.3. Battery 24
4.1.4. Voltage Regulator 24

4.2. Hardware architecture 25

4.3. Power consumption and total cost 25

CHAPTER 5. SOFTWARE DESIGN 27

5.1. Software environments 27
5.1.1. Arduino 27
5.1.2. ROS 27

5.2. Software architecture 29
5.2.1. Localization 30
5.2.2. Control 31
5.2.3. Path planning 32
5.2.4. Obstacle detection 34
5.2.5. Obstacle avoidance 36

CHAPTER 6. KINECT CALIBRATION 39

6.1. Camera geometric calibration 40

6.2. Depth calibration 42

CHAPTER 7. TEST AND VALIDATION 45

7.1. Path planning validation 45
7.1.1. Path planning test 45

7.2. Obstacle detection validation 49
7.2.1. Frame rate test 49
7.2.2. Reaction time test 49
7.2.3. False data probability tests 50
7.2.4. Dynamic obstacles test 51

7.3. Obstacle avoidance validation 52
7.3.1. Turn criteria test 52
7.3.2. Obstacle avoidance test 52

CHAPTER 8. CONCLUSIONS 55

BIBLIOGRAPHY 59

APENDIX A. GAUSS HARDWARE COMPONENTS 63
Robot Platform 63
Motors.. 64
PID Motor Controller 64
Encoders 65
Gyroscope 65
IR Sensors 66
Microcontroller 66

APENDIX B. ARDUINO CODE 67

APENDIX C. ODROID CODE 81

Main 81

Path planner 91

File reader 93

Classes 94
Point2D 94
Rover command 94

LIST OF FIGURES

Figure 1. Pictures of *Roomba, Amazon robot, Mars Curiosity* and *Google self-driving* car. [4], [6], [5], [8] ... 3
Figure 2. ToF diagram [18] ... 5
Figure 3. Versions 1, 2 & 3 of the "Bug algorithm" [26]. ... 9
Figure 4. Dijkstra algorithm [27]. ... 10
Figure 5. A* algorithm [27]. ... 11
Figure 6. D* algorithm working without previous map [28]. ... 11
Figure 7. Picture of *Gauss*. ... 14
Figure 8. Schematic of the skid steer drive [30] ... 15
Figure 9. *Gauss* reference framework ... 16
Figure 10. *Gauss* system architecture. ... 16
Figure 11. Picture of AROA ... 17
Figure 12. *AROA* system architecture ... 19
Figure 13. Robot State interaction diagram ... 19
Figure 14. Schematic of a Kinect V2 [39]. ... 22
Figure 15. Picture of an Odroid XU4 [40]. ... 23
Figure 16. Sample picture of a Li-Po battery [43]. ... 24
Figure 17. Picture of HE104+DX voltage regulator [46]. ... 24
Figure 18. Diagram of the hardware architecture. ... 25
Figure 19. ROS communication [48]. ... 28
Figure 20. Software System general scheme ... 29
Figure 21. Localization module algorithm ... 30
Figure 22. Control algorithm ... 31
Figure 23. Waypoint threshold ... 32
Figure 24. Path planning module ... 33
Figure 25. Obstacle detection area ... 35
Figure 26. Obstacle detection algorithm ... 36
Figure 27. OA conceptual design ... 37
Figure 28. Obstacle avoidance algorithm ... 38
Figure 29. Radial and tangential distortion effects [52]. ... 39
Figure 30. Calibration process ... 41
Figure 31. Uncalibrated and calibrated image comparison ... 42
Figure 32. Distance difference relative to X-coordinate ... 42
Figure 33. Distance difference relative to Y-coordinate ... 43
Figure 34. Distance difference relative to XY-coordinate ... 43
Figure 35. Ideal path ... 45
Figure 36. Path Planning test layout ... 46
Figure 37. Path Planning test results ... 48
Figure 38. False positive test layout ... 50
Figure 39. False negative test set up ... 51
Figure 40. Turn criteria test set up (right turn) ... 52
Figure 41. Ideal obstacle avoidance ... 53
Figure 42. Obstacle avoidance test set up ... 53
Figure 43. Lynxmotion A4WD1 chassis and wheels mounted kit [56] ... 63
Figure 44. GHM-16 motor [57] ... 64
Figure 45. RoboClaw 2x15A motor controller [58] ... 64
Figure 46. E4P-100 encoder [59] ... 65
Figure 47. L3GD20 [60] ... 65
Figure 48. Sharp GP2D12 IR sensor [61] ... 66
Figure 49. Picture of a BotArduino [62] ... 66

CHAPTER 1. INTRODUCTION

1.1. Project Scope

The world is evolving to the automation of processes, and robotics is the spare head of this phenomenon. From the small vacuum cleaners to planet exploration vehicles in space missions, and passing through robots used in factories, agriculture, warehouses or mining, they have invaded lots of aspects of our lives; they are here to stay, and they are the future.

The geomatics department of the CTTC (Centre Tecnològic de Telecomunicacions de Catalunya) proposed me a project for my degree final thesis, that consisted on improving an autonomous rover. This rover, developed by CTTC and named *Gauss*, had a very simple 2D obstacle detection system. The objective of the project was to improve the obstacle detection and to design a new obstacle avoidance system with 3D detection capability, and to implement it on the rover.

The new system, had to include new hardware components, and a renovated software architecture supporting ROS (Robot Operating System). The resulting system has been named AROA (Autonomous Rover for Obstacle Avoidance).

The main motivation is learning more about the field of robotics performing an experimental project, and contributing to CTTC robot development.

Summarizing, the objective of this project is to improve the capability of the current autonomous robot in order to make it capable of performing 3D obstacle detection and avoidance.

1.2. Document overview

This document is organized in 8 chapters, three annexes and this introductory section, summarized as follows:

Chapter 2, *State of the Art*, presents sensors and technologies used for wheeled robots, from a hardware and a software point of view. In addition, the most used obstacle avoidance techniques are presented.

Chapter 3, *System Design*, introduces *Gauss*' hardware and software architecture. Moreover, it presents a list of the new requirements and AROA's conceptual design.

Chapter 4, *Hardware Design,* explains in detail the components selection and the hardware architecture of the system.

Chapter 5, *Software Design*, presents the renovated software architecture and the algorithms and modules of the new system.

Chapter 6, *Kinect calibration,* presents the calibration process performed with the RGB-D camera Kinect, and the obtained results.

Chapter 7, *Test and Verification*, explains the different test scenarios and results of the different robot subsystems, including the obstacle detection and avoidance capability.

Chapter 8, *Conclusions*, discusses the results of the project and proposes further improvements that can be performed in the rover.

The Annexes present the hardware used by *Gauss* and the used code.

CHAPTER 2. STATE OF THE ART

An autonomous rover is a robot capable of navigating throughout an environment without the intervention of a human operator. Robots, and especially autonomous rovers, provide efficient and practical utility for business, particularly in industrial and space applications. Never sleeping, never tired, never bored, never having lack of motivation and never being sick. That is why robots are already preferred over humans in some areas, especially on hazardous environments or situations [1].

2.1. General view on autonomous rovers

Robots can be categorised in multiple ways, but usually are classified according to type of platform and/or the application. Robots can fit in five categories from the **platform** point of view: stationary, legged, wheeled, flying or swimming robots [2]. According to this classification, an autonomous rover is a wheeled robot. Wheeled robots are defined as robotic platforms that navigate around the ground using motorized wheels to propel themselves [3]. The main advantages they offer are that are easy to implement in mechanical terms and that are very stable in most environments, compared to other mobile platforms such as legged robots. The main drawback of wheeled robots is that they are not able to perform a good navigation over some surfaces, such as rocky terrain, sharp declines, or areas with low friction.

In terms of **application**, autonomous wheeled robots are one of the most used platforms in nowadays engineering: For instance, domestic robots such as *Roomba* vacuum cleaner [4], rovers used in space applications such as the Mars Curiosity [5], robots in Amazon automated warehouses [6], multipurpose robots designed by Clearpath robotics [7] or the Google self-driving car [8] (Figure 1).

Figure 1. Pictures of *Roomba, Amazon robot, Mars Curiosity* and *Google self-driving* car. [4], [6], [5], [8]

Each platform and specific application, require from specific hardware and software systems.

As for **hardware**, robots carry different types of sensors (which will be discussed in section 2.2), actuators and integrated circuits (IC). The most used types of IC are programmable chips, and among them, the microprocessor, the microcontroller, the FPGA (Field Programmable Gate Array). A robotic platform can have one or multiple ICs of a single or different types, depending on the task that it needs to perform. For example, a robotic arm that performs an easy motion may use a single IC whereas Mars rovers, which are complex systems, may use an IC for each different task (One for navigation, one for sensors acquisition, one for the robotic arms…).

In general, microprocessors (μP) have a dedicated CPU (Central Processing Unit) where the RAM (Random Access Memory), ROM (Read-Only Memory), etc., run on different chips [9], whereas microcontrollers (μC) usually have a single embedded chip that includes and manages the CPU, RAM, ROM and I/O (Input/Output) ports [10], although there are exceptions. Also, μP clocks usually run faster than μC clocks. This is why μP are usually used for processes that require more processing capacity. An example of μC and μP are Arduino [11] and Raspberry Pi [12], respectively. FPGAs are a type of IC that has reprogrammable logic gates, whereas μP and μC cannot. FPGAs are more expensive and more power consuming than regular microcontrollers or microprocessors, but in exchange can perform dedicated tasks at faster speeds than μP and μC. The reason is that FPGAs have reprogrammable logical gates. Xilinx Spartan-6 is an example of a FPGA board [13].

The **software** framework of robots is usually attached to the type of IC in use. For instance, MyRio is prepared to work with LabView or Arduino with to work with Arduino programming code and the Arduino IDE. Although it is possible to code specific ICs with other languages, it is unusual to see it. Even tough, there are some open-source software platforms in robotics, being the most representative ROS (Robot Operative System), which can be used with multiple hardware devices and robot platforms.

ROS is a flexible framework for writing robot software, as defined in their web site [14]. It compiles a collection of tools, libraries and conventions (grouped in what they call nodes) that aim to simplify the task of creating a complex and robust robot behaviour across a wide variety of robotic platforms. ROS implements many nodes that perform most of the tasks that a robot might need to be developed. For instance, there are algorithms that perform sensor data acquisition or control. Among all of them, the most interesting for robot control with obstacle avoidance is the ROS navigation stack [15]. The navigation stack is a set of ROS nodes that work together to provide velocity commands to the rover base, taking into account the position, the obstacles and plenty of other parameters critical for robot movement through an environment.

ROS has three main advantages over other robotic frameworks; community, standardization and the fact that it is open-source.

ROS **community** is becoming more popular day by day. It has a huge community that maintains and creates new libraries and tools for plenty of robot sensors and devices. Some of the nodes previously stated, are created by the community.

As stated before, ROS is **open-source**. That is the reason why it is compatible with plenty of hardware and robotic platforms. For instance, any Ubuntu or Debian Linux distribution can support ROS. ROS is also free to use for developers, meaning that all the libraries can be downloaded with no cost and are modifiable. Even tough, for commercialization it is required to use its industrial version, ROS-industrial [16], that requires to pay for its licences.

All the code and messages exchanged among ROS nodes are **standardized**. This means that an algorithm that works for a robot can be used in any other ROS-based robot with few modifications. It also allows to easily implement new hardware on a system, as long as it can provide the specified ROS messages.

2.2. Sensors for autonomous rovers

In order to be autonomous, wheeled robots need to know the environment in which they are moving and their current localization, which is the position and orientation they have respect this environment. The hardware used to obtain this data can be classified in depth perception and localization sensors.

2.2.1. Depth perception sensors

Depth perception consists on determining the distance between the robot and the surrounding objects than conform the environment. This is done by taking measurements using one or a set of sensors, and then extracting meaningful information from those measurements. There are a lot of types of depth perception sensors, and most of them imitate human or animal senses. For instance an ultrasonic sensor imitates bats echolocation [17]. Autonomous rovers usually relay on active-ranging sensors. This type of sensors provide direct measurement of distance to the surrounding objects interacting with the environment. There are different technologies used by these sensors, but the most used in robotics is the time of flight (Figure 2).

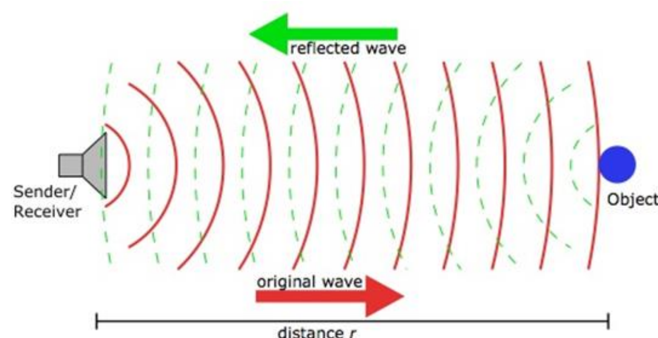

Figure 2. ToF diagram [18]

Time-of-flight (ToF) ranging makes use of the propagation speed of the sound or of an electromagnetic wave. The distance to objects (d) is computed by measuring the time (t) that the wave travels to the object based on the characteristic propagation speed of the wave (c).

$$d = \frac{t\ x\ c}{2} \tag{2.1}$$

The most common ToF sensors are infrared (IR), ultrasounds, laser rangefinders and RGB-D (Red-Green-Blue Depth) [19]. IR sensors emit pulsed light, ultrasonic sensors emit sonic waves, laser rangefinders emit a collimated light beam and RGB-D emit a pulsed light pattern (usually known as structured light).

IR sensors use infrared light, which in general makes them less susceptible to interferences than other ToF based sensors. Even tough, in areas where there is another IR source, for instance under the effect of sunlight, an IR sensor could be partially or totally blocked. In addition, they do not detect some surfaces, such as glass [20], because emitted light has not good reflection.

Ultrasonic have the advantage that sound waves are reflected by surfaces than other ToF cannot detected, like glass or a mirror. In addition, they are the cheapest sensor of this kind in the market. Its main drawback is the lack of resolution compared to other ToF sensors, and the susceptibility to interferences produced by other ultrasonic sources [19].

RGB-D are a type of depth sensing devices that work in association with a RGB camera. The sensors is able to augment the conventional image with depth information about each pixel. In this sensors, the emitter projects a known pattern onto the environment. The projected pattern is then captured by an infrared camera in the sensor and estimates the per-pixel depth [21]. This sensors use the technology combined with a RGB camera to create a coloured set of points with known 3D coordinates, called *point clouds* [22], of the objects in front of the sensors. A RGB-D sensor provides high accuracy and resolution compared to IR and ultrasonic sensors. In the other hand, are more expensive than the other sensors and their operating range is limited by the geometry of the sensor.

The fact that **laser rangefinders** emit light in a directive beam, means that it is more resistance to interferences and has a greater detection range compared to other ToF sensors [20]. This type of sensor can be attached to a rotatory mechanism that aims the light beam to different positions to obtain 2D or 3D information of the environment. When a range finder has these characteristics is called LIDAR (LIght Detection And Ranging), which is one of the most used active range finders in robotics. Laser rangefinders main drawback is that they are the most expensive ToF sensors in the market.

2.2.2. Localization sensors

Localization, or positioning, consists on determining where the robot is in the environment. In autonomous rovers, localization consist on determining a set of TPVA (Time-Position-Velocity-Attitude). TPVA describes the robot position, velocity and attitude at a particular instant of time. Localization sensors can be classified as inertial and non-inertial sensors.

Inertial sensors are instruments that measure rotation and translation forces observed by the object. These sensors are usually used to perform dead reckoning, which is a method of localization that relies on estimating the position, speed and orientation of the robot based on earlier known positions [23]. The most used inertial sensors in autonomous rovers are gyroscopes and accelerometers.

Gyroscopes are used to measure rotational forces. In autonomous rovers they are used mainly to determine the heading of the vehicle. Gyroscopes measure reactive torque that is produced due to the movement of the sensor to give absolute orientation respect their spin axis. Torque (τ) is proportional to the spinning speed (ω), the precession speed (Ω), and the wheel's inertia (I) [19].

$$\tau = \mathrm{I}\omega\Omega \tag{2.2}$$

The main drawback of the gyroscopes used for dead reckoning is that the typical error of the system is accumulative. This means that if they are used for a long time, a small constant drift will provide a big error.

An **accelerometer** is a device that measures static (gravitational) and dynamic acceleration forces [24]. Given the mass (m) of the accelerometer, the measure of the dynamic forces (F) provides the acceleration (a) of the sensor in every axis:

$$\mathrm{a} = \frac{\mathrm{F}}{\mathrm{m}} \tag{2.3}$$

With the acceleration, the speed and position are easy to obtain with integral calculation [24].

An **IMU** (Inertial Measurement Unit) is an embedded sensor that usually combines 3-axis accelerometer and 3-axis gyroscope orthogonally placed between them. In some cases, IMUs also carry magnetometers. IMUs are the most used inertial sensor in robotics because are capable of measuring translation and rotation, by integrating the different sensors previously stated.

Non inertial sensors do not measure directly the motion of the object. There are plenty of sensors and technologies that can be used for localization (cameras, barometers, Wi-Fi, UltraWide band, etc.), but the most used on autonomous

rovers are magnetometers, rotatory encoders and GNSS (Global Navigation Satellite System). In addition, RGB-D or LIDAR sensors can be used for localization using techniques such as SLAM (Simultaneous Localization And Mapping) [25].

Magnetometers determine the direction of Earth's magnetic north, and there are two types: Hall Effect and flux gate compasses. Regardless of the type of compass used, a major drawback concerning the use of the Earth's magnetic field for mobile robot applications involves disturbance of that magnetic field by other magnetic objects. This issue is greater for indoors robots, since most buildings have metallic structures that can disturb the magnetic field inside.

Rotatory encoders track the angular position of any rotatory device to generate digital information. In rovers, are used to determine odometry (change of position over time) by monitoring the number of turns made by the wheels. If an encoder is able of determining the direction of the rotation, it is called quadrature encoder. There are many types of encoder, like magnetic or mechanical, but one of the most used in robotics is the optical encoder. The resolution is measured in cycles per revolution (CPR) [19].

The **GNSS** is a beacon based localization [19]. There are at least twenty-four operational GNSS satellites at all times. Each satellite continuously transmits data that indicate its location and the current time. The GNSS satellites synchronize their transmissions so that their signals are sent at the same time. When a GNSS receiver receives the RF (Radio-Frequency) signal of a satellite, the arrival time is measured and used to determine the relative distance to this satellite, usually known as pseudorange. By combining four or more pseudoranges, the position of the receiver is determined by triangulation means. The main advantages of this system is low price of GNSS receivers and the fact that they do not have accumulative error. In the other hand, GNSS receivers cannot be used as a reliable localization sensors on robots that have to work in low coverture areas, such a dense forest or indoors.

SLAM is a technique that involves the use of perception sensors, such as RGB-D sensors, LIDARs, cameras and sometimes inertial sensors. It consists on trying to simultaneously localize the sensor with respect to its surroundings, while at the same time mapping the structure of that environment [25]. The sensors are obtaining depth measurements from the environment and comparing the obtained pattern with a map of the surroundings. The map can be introduced by the user or generated and actualized during the operation of the robot. If the detected pattern coincides with a pattern of the environment, the robot is able to get a localization. At the same time, the environment is being mapped. It is a good localization technique, but requires a post process period that is time consuming [25].

2.3. Obstacle avoidance techniques

One of the most critical issues a rover design is deciding how it reacts to the environment. If there is an obstacle in the middle of the path, the robot has to decide how to surpass it. Obstacle avoidance techniques aim to solve this problem. Since the route has to be recalculated, it is common to see path obstacle avoidance algorithms planning that include the robot path planning algorithm. There are multiple obstacle avoidance algorithms. They can be classified between map based and non-map based algorithms.

The non-map based algorithms do not relay on previous data of the surroundings, but usually are more time consuming. There are plenty of algorithms in this category, but the most representatives are the bug algorithm, the artificial potential field method, the vector field histogram and follow the gap method.

Bug algorithm is the older one and the most basic algorithm. It plans a direct path to the destination and only intervenes when an obstacle interferes the path, then it follows the edge of the obstacle and takes a decision, that depends on the bug algorithm version (there are 3 of them, Figure 3). The first version optimizes the shortest distance, circling the obstacle until it reaches the starting point, and then in computes the best “obstacle leaving point”. Once the “leaving point” is reached again, it starts moving to the new generated path. The second version generates a slope from the initial point and when an obstacle is found, it follows the edge until this slope is reached again. The third version tries to optimize the total travelled distance. When the robot is circling the obstacle, simultaneously it computes the shortest “clear of obstacle” distance to the destination point, and sets a new path to the destination. All three versions’ main advantage over other obstacle avoidance algorithms is its reliability, because they always lead to the final goal, but its main issue is that they not the best time saving option.

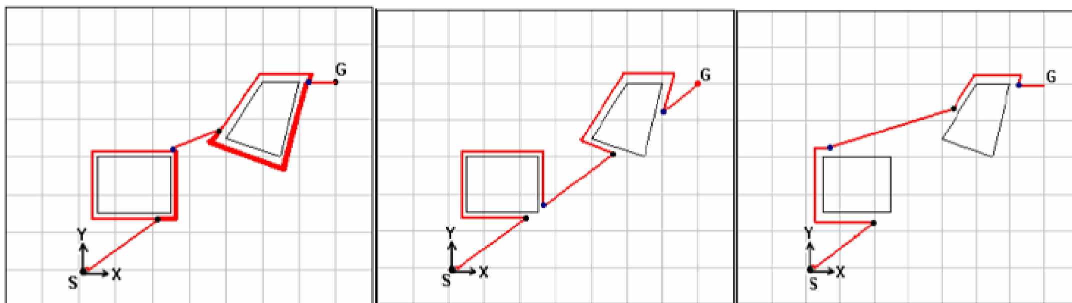

Figure 3. Versions 1, 2 & 3 of the “Bug algorithm” [26].

The **artificial potential field method** (APF) is a goal oriented algorithm, based on the “potential field” principle. The robot and the obstacles are considered positive charges and the goal a negative one, which means that the goal will “attract” the robot and obstacles will “repel” it. The magnitude of the forces depend on the distance to the elements of the environment (closer obstacles induce greater charges). This method always selects the shortest path, but has

a local minima problem, since symmetric or U-shaped obstacles can create an equipotential sink that the robot is not be able to leave.

Vector field histogram (VFH) is a method based on a generated distance histogram of the surroundings obstacles, which is used to compute the optimal route. It first makes a 2D histogram generated with the obstacles detected by the sensors on the environment, which is converted into a polar histogram that is used to decide the path with less “polar obstacle density”. The density is assigned according to the distance of the obstacles (closer obstacles will generate greater obstacle density). It is a good method to find the optimal path, but it needs lots of calculations, which makes it time consuming.

Follow the Gap Method (FGM) is a method is based on finding the gap between obstacles. First it uses the sensors to find all the gaps between obstacles on the trajectory, and computes the heading angle needed to go to the biggest gap. Like the APF and the VHF, it is a goal oriented system that also selects the shortest path with little computation, but it also has a problem with U and H shaped obstacles.

The grid map-based algorithms use a map of the environment in which the robot operates. There are multiples map based algorithms, but the most important are Dijkstra, A* and D*.

Dijkstra algorithm works by visiting grid squares in the graph starting in the robot origin. It then repeatedly examines the closest non visited grid squares. It expands from the starting point until it reaches the goal [27]. This algorithm is easy to implement and computes the shortest path from the origin to the destination, but for goals at long distances it is time consuming, since it scans the whole map, as can be seen at Figure 4.

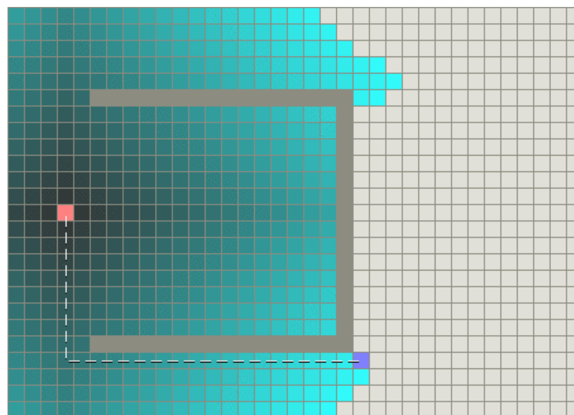

Figure 4. Dijkstra algorithm [27].

A* is an algorithm similar to Dijkstra, but with some improvements. Instead of going to the next non visited grid square, it decides the next one using the following formula:

$$f(n) = g(n) + h(n) \tag{2.4}$$

g(n) represents the cost of the path from the starting point to any vertex n, and h(n) represents the heuristic estimated cost from vertex n to the goal. A* balances the two as it moves from the starting point to the goal. Each time through the main loop, it examines the vertex n that has the lowest f(n). In Figure 5 the improvement in the number of grid squares visited respect Dijkstra is easily appreciated [27].

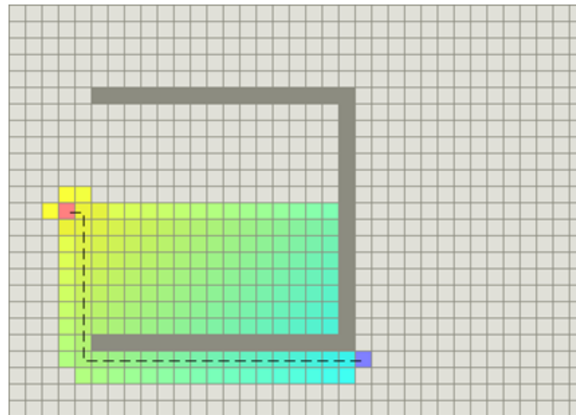

Figure 5. A* algorithm [27].

The problem with Dijkstra and A* is that they are not suitable to work with a dynamic map or without it. The **D*** algorithm (Dynamic A*) was developed to solve this issue, guarantying that the initial path planning can be repaired in real-time [28]. D* produces an initial plan based on known and assumed information with A*, and then incrementally repairs the plan as new information is discovered about the environment. If there is no map, D* assumes that there are no obstacles and plans the shortest path. As it follows the path, it scans for obstacles. The D* algorithm is used to re-plan each time a discrepancy is seen (Figure 6).

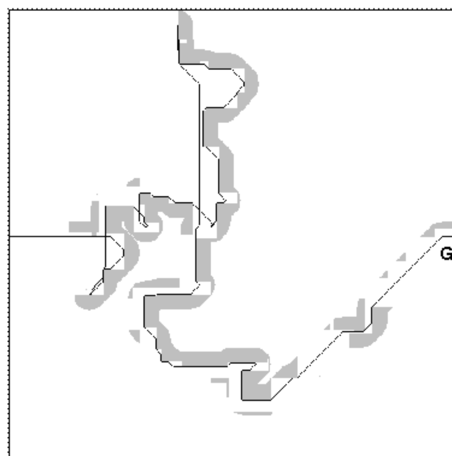

Figure 6. D* algorithm working without previous map [28].

CHAPTER 3. SYSTEM DESIGN

As stated in the introduction, the implemented robot design is built on an existing CTTC robot platform, *Gauss.* Using this robot prototype as a starting point, AROA has been designed taking into account the system requirements established by CTTC.

3.1. System Requeriments

The system requirements, established by the CTTC to evolve the *Gauss* project, are the following:

- Design and implement an improved **obstacle detection** module. *Gauss* has a simple 2D obstacle detection. It should be able to detect 3D static obstacles in the path of the robot. It should include a calibration of the obstacle detection sensors.

- Design an **obstacle avoidance** module that provides an algorithm to determine a new path to follow in the event of obstacle detection. The new system must be compatible with *Gauss,* both in software and hardware.

- The system must be **ROS** based, and compatible with Linux OS processors.

- The cost of the new hardware required by the system should not be higher than 500 €.

3.2. Gauss Design

Gauss was designed as a part of *OpenInLocation* project [29], whose mission was to accelerate the development of indoor location solutions. It features a mobile autonomous platform with the required hardware and software equipment to navigate in indoor real-life scenarios. The role of *Gauss* robot was focused on providing a robot with localization, path planning and obstacle detection capabilities.

Figure 7. Picture of *Gauss*.

The robot is a four-wheeled mobile platform composed of four motors, two encoders, a PID (Proportional-Integrative-Derivative) motor controller, a 3-axis gyroscope, four IR distance sensors and an Arduino microcontroller (
Figure **7**).

The system architecture can be defined in four big blocks or modules: localization, control, obstacle detection and path planning.

The **localization** is in charge of obtaining displacement measurements from the encoders and angular speed measurements from the gyroscope. This data is processed to obtain a 2D position and a heading angle. To obtain position, encoder counts are converted to meters. The encoder counts per meter is a parameter that can be obtained measuring the wheel perimeter and the CPR (cycles per revolution). In this particular case, the encoder counts are 27700 per meter. To compute the travelled distance in meters, the average between the two encoders is calculated:

$$\text{distance (m)} = \frac{(\text{encoder one counts} + \text{encoder two counts})}{2 \times \text{encoder counts per meter}} \qquad \textbf{(3.1)}$$

Then, X and Y coordinates that give the position are calculated by means of trigonometry, using the heading angle:

$$\text{X} = \text{distance} \times \cos(\text{heading}) \qquad \textbf{(3.2)}$$

$$\text{Y} = \text{distance} \times \sin(\text{heading}) \qquad \textbf{(3.3)}$$

To obtain the heading increments on a time instant, the measured angular speed has to be converted into angular displacement multiplying by the sampling time period. The total heading consists of the previous heading plus the angle increase of the new measure.

$$\text{heading}\ (^{\circ}) = \text{previous heading} + \text{angle speed} * \text{time} \tag{3.4}$$

The **control** is based on skid steer motion (Figure 8), which is based on differential drive. This type of motion is used by military tanks and bulldozer vehicles [30]. This means that the four robot motors work paralleled in two motor drives, left and right. The motors are controlled by the motor controller, which receives direct commands from the Arduino microcontroller. Arduino has libraries that allow an easy communication with the motor controller. *Gauss* control is based on two commands, turn and advance. At turns, the motor controller receives the required turning angle in degrees, and actuates the both motor drives in consequence. To advance, the motor controller receives the required distance, speed and acceleration for both motor drives, left and right. The speed and acceleration are fixed. After finishing any command, the robot stops automatically

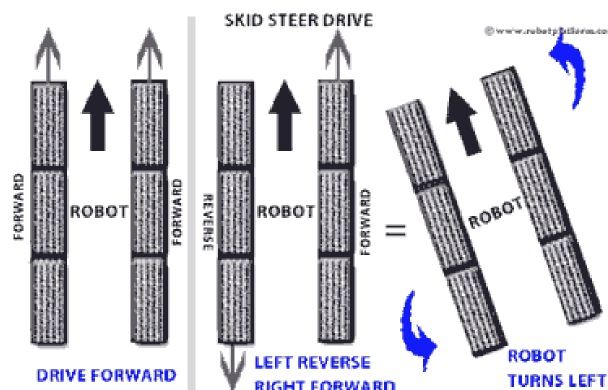

Figure 8. Schematic of the skid steer drive [30].

The **obstacle detection** module is capable of detecting obstacles in two different plains, both vertical and horizontal. To detect the obstacles, the robot is endowed with four IR sensors, two of them oriented to the front of the platform to detect obstacles in the horizontal plane and the other two oriented downward to detect in the vertical one. Horizontal obstacles are the ones found in front of the robot and vertical are situated below the robot, for instance, cliffs. Obstacle detection is enabled at all times, interrupting the motion if necessary. If an obstacle is detected, the detection module sends the proper command to stop the robot. If an obstacle has not been encountered, the robot continues moving to the destination waypoint. If an obstacle is encountered, the robot does not attempt to determine an alternative route to reach the destination. This is the reason why this robot is not considered to implement an obstacle avoidance system.

The **path planning** module computes the route to follow. The robot framework is based on local coordinates, where the robot starts at the origin of coordinates (0,0) and oriented to a 0º heading (Figure 9). It is important to state that *Gauss* does not use the same local coordinates for all of its operation; the coordinates and heading are reset to zero each time the destination is reached.

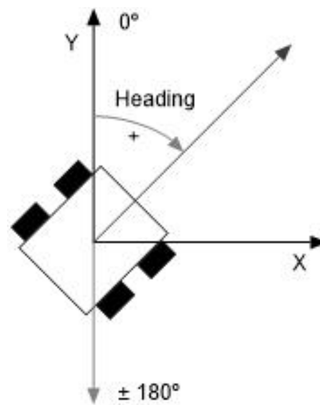

Figure 9. *Gauss* reference framework

Path planning starts by reading a set of 2D coordinates, called waypoints, containing the destinations. Then, it computes the distance and angle between its current localization and the next waypoint. Then sends this data to the motor controller, which actuates in the motors to turn first and advance later. During this process, localization provides feedback to the system. The robot will go to all these waypoints following the established order, and stop when the last waypoint is reached.

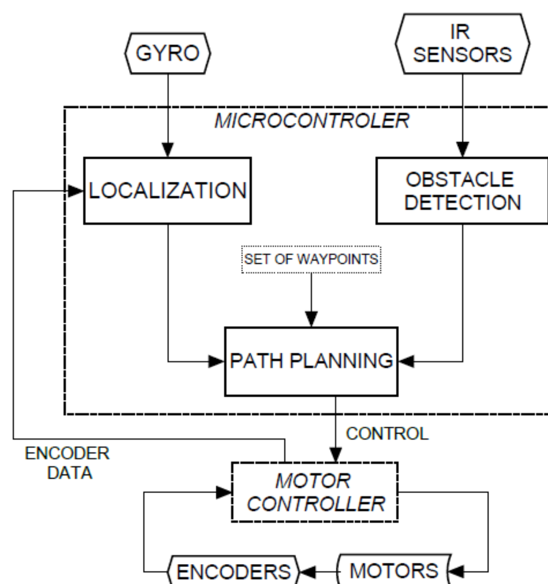

Figure 10. *Gauss* system architecture.

In order to log the positions walked through, the robot has its own serial communication protocol. All the software architecture is based on Arduino and therefore all robot functions are programmed using the Arduino IDE.

There is further information on the hardware used on *Gauss* at APENDIX A. GAUSS HARDWARE COMPONENTS.

3.3. AROA Design

AROA is an evolution of the *Gauss* rover that takes into account the targeted requirements with a renewed system architecture. It improves the previous obstacle detection and introduces new obstacle avoidance and path planning modules. The system is now able to detect 3D obstacles and give a new path solution in order to avoid them. These new modules are implemented in a new microprocessor, whose objective is to support ROS system (Arduino does not support ROS because it requires to be installed in a Linux OS). The localization and control modules are maintained on the Arduino with few modifications, although they have been adapted to work with ROS.

To fulfil the obstacle detection and avoidance module requirements, the new design includes a RGB-D sensor as a range finder. In addition, since the new microprocessor and RGB-D work at different voltages, a new voltage regulator and a battery are also included. This new hardware is provided with power supply by an independent new battery. The hardware used in *Gauss*, (Arduino, motor controller, gyroscope, encoders and motors) is maintained.

Figure 11. Picture of AROA

The new system architecture is now divided in five conceptual modules: localization, obstacle detection, obstacle avoidance, path planning and control.

The **localization** module is based on the *Gauss* system. The Arduino acquires data from the navigation sensors, encoders and gyroscope, and computes the localization of the robot. The main difference is that now the data is sent to the path planning module located in the microprocessor via serial bus.

The **control** is also based on *Gauss*, although it has some changes. The Arduino receives a command from the path-planning module via serial bus, and executes the corresponding motor controller functions. The motor uses the same function for turns, but not for advancing. To advance now only sets a fixed speed at both motor drives. To stop the robot, a new stop function is included. This function sets the speed to zero on both motor drives. As for the motion, the skid steer drive is maintained.

The **obstacle detection** has been upgraded and migrated to the microprocessor. It has two main functions: to acquire data from the RGB-D camera and to decide whether an object in path of the robot is an obstacle or not. If an obstacle is detected, it sends the position of this obstacle to the obstacle avoidance module. The design was intended to incorporate the IR, but the lack of memory after implementing ROS on the current Arduino does not allow this feature. Even tough, the system architecture is prepared to receive and process IR data.

The **obstacle avoidance** module is performed at the microprocessor. The used algorithm has been designed especially for AROA, and is introduced later (5.2.5). The general operation works as follows: the module is only enabled when an obstacle has been detected. Its main task is to compute a new waypoint that guides the robot into avoiding the obstacle. To do so, the robot performs a scan of the environment and searches obstacle free areas. When a suitable area is meet, the algorithm creates a new point on the route. The pathfinder module reads this as the new waypoint.

The **path planning** has been renewed and migrated to the microprocessor. In AROA, the framework of the robot is still based on local coordinates, but the difference is that now are maintained during all the operation. It uses the localization data sent by the localization module, and a file where all the waypoints of the route are stored. It gives a path solution according to the received inputs. The module receives the current coordinates and the new destination coordinates (which are obtained from the file) and sends a control command to the Arduino. There are four types of output commands: forward, turn right, turn left and stop.

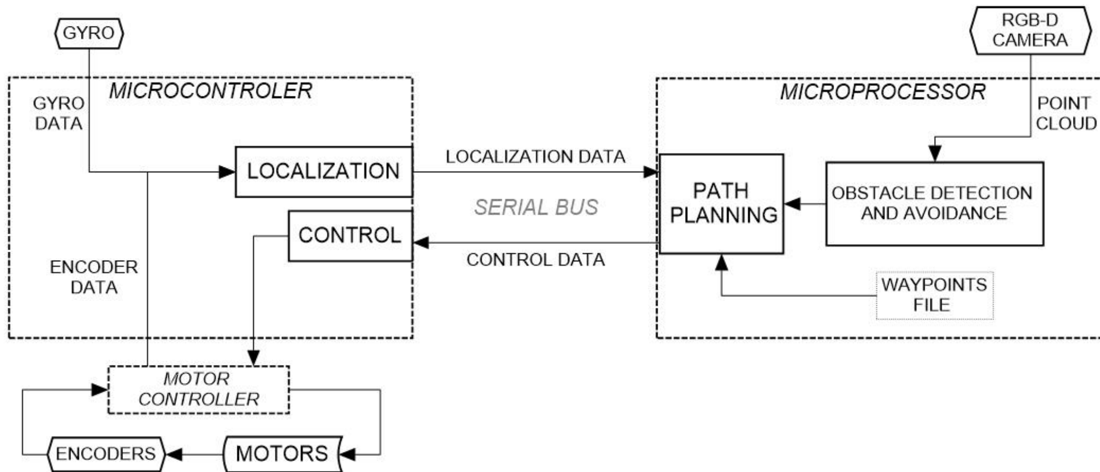

Figure 12. *AROA* system architecture

The modules work with a four-state system; STOP (1), FORWARD (2), TURN RIGHT (3) and TURN LEFT (4). The state, called Robot State (RS), is defined by path planning and is sent as a robot command to the Arduino. Arduino updates the state with the robot command, and performs specific actions depending on the state. All robot actions end with a STOP command, which is the starting point for each motion.

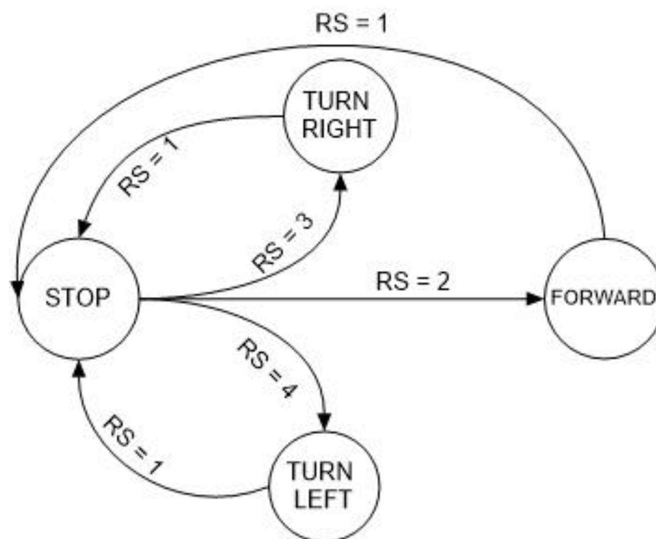

Figure 13. Robot State interaction diagram

Although the Arduino IDE is still used to program the Arduino microcontroller, the microprocessor software design relies on ROS. ROS nodes can be programmed in C++ and python, but in the AROA design all ROS code has been programmed in C++.

CHAPTER 4. HARDWARE DESIGN

4.1. Component selection

There are four new hardware components in AROA: a microprocessor board, a RGB-D camera, a voltage regulator and a new battery.

4.1.1. 3D Range finder

Before deciding to use an RGB-D camera, some other range finder sensors were considered. The selection took into account five different types of sensors: IR, LRF, ultrasonic, RGB-D and LIDAR.

As stated in section 3.3, the robot should be able to detect 3D obstacles. This fact made the IR, 2D laser rangefinders and ultrasonic sensors non-suitable as the main sensor. The reason is that they can only detect obstacles on a quasi-linear beam (1D) or 2D plain.

LIDAR and RGB-D sensors were the only options that provided a point cloud, allowing for 3D obstacle detection. LIDARs are more expensive than RGB-D sensors, with prices starting at 400 € [31], against the 100 € that a typical RGB-D costs [32]. Although LIDARs are much more precise than RGB-D sensors, the high resolution did not compensate the extra price given the fact that this is a low cost robot. In addition, RGB-D sensors provide image information besides depth, which can be useful in future for functionalities.

For the selection of an RGB-D camera, an extensive research has been made looking for the best RGD-B sensor in terms of performance and price, but also looking for a device that it can be easily integrated in ROS.

Among the multiple options in the market, only four have active support on the ROS community: *Asus Xtion Pro, Intel® RealSense™ R200, Kinect V1* and *Kinect V2* [33]. The sensors should have a good range and field of view, in order to obtain the maximum information possible from the environment. The frame rate and latency of the depth processing need to be taken into account, since they represent how fast data is acquired, processed and transmitted. Finally, the price and the type of depth measurement technology are elements to be considered. In Table 4.1 a comparative of the four RGB-D sensors is presented.

Table 4.1 RGB-D sensors comparative [34], [35]

	RealSense R200	Kinect V1	Kinect V2	ASUS Xtion
Max. Depth distance	3.5 m	4 m	4.5 m	3.5 m
Min. Depth distance	0.6 m	0.5 m	0.5 m	0.8 m
Latency	-	60 ms	20 ms - 60 ms	90 ms
Field of view (Horizontal/Vertical)	59º/ 46º	57º/43º	70º/60º	57,5º/43,5º
Frame Rate	60 fps	30 fps	30 fps	30 fps
Depth measurement technology	Computer Stereo Vision	Structured light	Structured light	Structured light
Price	151€ [36]	50€ [37]	90€ [32]	Out of Stock

The selected RGB-D sensor is Kinect 2 because is the sensor with better price-quality relation. It is the sensor with the highest operational range and field of view, and also has the smallest latency time. The frame rate is good enough for this application, even though it is worse than the offered by R200. In addition, Kinect based sensors are compatible with ROS.

Kinect 2 has two cameras with a resolution of 512x424 pixels (infrared) and 1920x1080 pixels (RGB). The maximum power consumption of the device is 32 W (12 V / 2.67 A), although the typical consumption is 15 W. The main drawback of Kinect 2 is that it requires an adapter cable that transmits the data to the computer and provides power supply to the sensor [38]. The connector of this cable is not standard, and the only way to obtain it is by purchasing the official adapter, which increases the final price in 50 €.

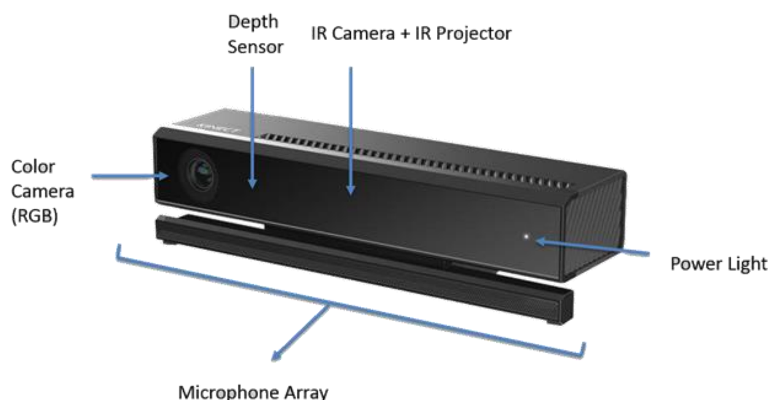

Figure 14. Schematic of a Kinect V2 [39].

4.1.2. Microprocessor

The main microprocessor candidates for the AROA system were the Raspberry Pi 3 and the Odroid XU4. The reason is that both devices are boards owned by CTTC and compatible with Linux OS.

Odroid XU4 mounts a Samsung Exynos5422 octa-core CPU, formed by two quad-cores, an ARM Cortex-A15 (2.1 GHz) and an ARM Cortex A-7 (1.5 GHz) [40]. Raspberry Pi 3 uses a single quad-core processor of 1.2 GHz. The more cores and processing speed provides a higher speed on data acquisition and processing from the Kinect. Odroid also has a 2 GB RAM memory in front of the 1 GB of the Raspberry. Both devices have a GPU (Graphics Processing Unit) that shares the RAM memory with the CPU. Raspberry has a Broadcom Videocore IV and Odroid a Mali-T628 MP6.

Although the components do not require to be purchased, the price has been taken into account in the selection process, since it influences on the final AROA system cost. Raspberry price is 31 € [41] and Odroid 53 € [42].

In addition, Odroid microprocessor has 2 USB 3.0 ports, whereas Raspberry only incorporates 2.0 USB ports. Due to the fact that Kinect V2 requires a 3.0 USB port to work properly, the selected microprocessor is Odroid XU4.

The lack of popularity of Odroid, in front of other devices such as Raspberry, is one of its main drawbacks. There is less support and troubleshooting on the internet because there are less people developing with it. The other main drawback of Odroid is the power consumption. Odroid uses 20 W at maximum workload capacity, and about 10 W at normal usage. Even tough, consumption is not a critical element in the AROA system, since it can carry high capacity batteries that could increase the robot’s autonomy.

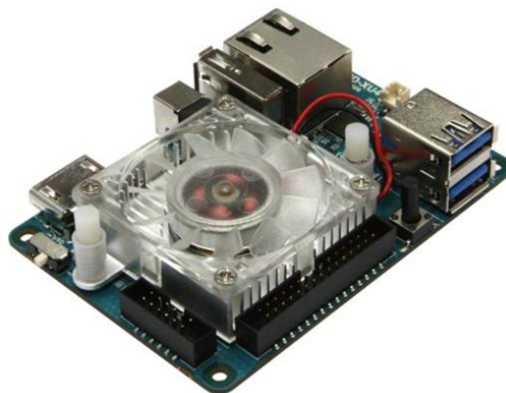

Figure 15. Picture of an Odroid XU4 [40].

4.1.3. Battery

The new system consumption is higher than in the previous Gauss design, which requires another battery. This battery only provides power to the new hardware (*Kinect* and Odroid). The maximum power consumption of the system is 50 W (30 W from Kinect and 20 W from Odroid).

The selected battery is a VoltOn Li-Po (Lithium-Polymer) that provides 4550 mAh, 14.8 V and a maximum peak of 35 C. This provides 67.3 W per hour, which is enough to power the whole module for, theoretically, at least two hours. The battery cost is 30 €.

Figure 16. Sample picture of a Li-Po battery [43].

4.1.4. Voltage Regulator

The new integrated hardware works at different voltages levels, and a voltage regulator is needed to provide the required amount of power to each device.

The selected device is a Diamond Systems HE104+DX linear voltage regulator. The HE104+DX is a high efficiency (95 %), high performance DC-to-DC converter that supplies +3.3 V (30 W), +5 V (45 W), +12 V (36 W) & -12 V (6 W) outputs [44].

Kinect works at 12 V and Odroid at 5 V making this device perfect for this purpose. The price of this component is 250 € [45]. Even tough, this device is property of CTTC and only for temporal use; at the future a lower cost voltage regulator will be used.

Figure 17. Picture of HE104+DX voltage regulator [46].

4.2. Hardware architecture

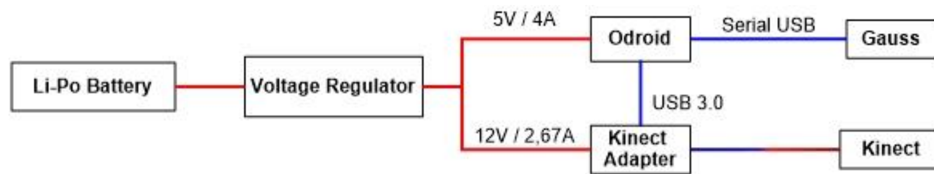

Figure 18. Diagram of the hardware architecture.

The system hardware architecture is represented in Figure 18. The red lines represent power connections and the blue lines communication buses. As it can be seen, the system power relies on the Li-Po battery. The battery is directly connected to the voltage regulator, which distributes the voltage to the Kinect adapter and the Odroid.

The Kinect adapter is connected to Odroid with a USB 3.0. The Kinect adapter is connected to Kinect in the other end with the non-standard cable. This cable transmits the acquired data to the Kinect adapter and power to the sensor.

Odroid is also connected to the Arduino, and thereby to the rest of the *Gauss* system, via serial USB to receive localization data and transmit control commands.

4.3. Power consumption and total cost

The system maximum power consumption is 50 W, as stated before, but the theoretical consumption differs from the real. Table 4.2 Power consumption and autonomyshows the power consumption based on empirical test, and the theoretical autonomy based on this consumptions.

Table 4.2 Power consumption and autonomy

System state	Power consumption	Autonomy (theoretical)
Idle	13.79 W	4h 50’
Fully operative	30.34 W	2h 12’

The total cost of the new design components is 473€, although the cost spent to develop the prototype is only 140€ because the Odroid, battery and voltage regulator where devices previously owned by CTTC (Table 4.3).

Table 4.3 List of component prices

Component	Price
Kinect 2	90 €
Kinect adapter	50 €
Odroid	31 €
Voltage regulator	250 €
Battery	30 €
TOTAL	**473 €**

CHAPTER 5. SOFTWARE DESIGN

5.1. Software environments

AROA uses two different IC boards, and each one has been coded in different languages and frameworks, Arduino and ROS.

5.1.1. Arduino

Arduino is an open-source electronics platform based on easy-to-use hardware and software [47]. It uses the Arduino programming language, which is a simplified version of C/C++, and the Arduino IDE as programming environment. Arduino contains libraries to communicate with the motor controller, to acquire data from the IMU and the IRs and to generate and receive ROS based messages.

5.1.2. ROS

ROS has been introduced in the state of the art (2.1), but this section focuses on explaining how it works in order to understand the code and algorithms designed. Here are a brief explanation of ROS most important concepts and functionalities [48].

- Packages: They are the main unit to organize software in ROS. A package can contain ROS processes (called nodes), libraries or datasets, among others.

- Message (msg) types: A message defines a data structure comprising typed fields, supporting standard primitive messages (integer, floats, bools…) among more specific messages, such as localization or IR Data messages. The messages are standard for all the ROS community, which means that all the message fields have to have a specific type of data.

- Service (srv) types: Services are based on the publish/subcribe model, and are conformed of a pair of message structures, one is used to request and the second is the reply. To use a service, there has to be a node requesting it and waiting for the response, whereas a message is published without need of being requested. The current AROA design, does not use services.

- Nodes: Nodes are processes that perform computation of a single task. A robot control system usually comprises many nodes working in parallel and interacting each other. For example, one node might be in charge of

the infrared sensors and another of the control of the motors. ROS nodes are written using a ROS client library, generally roscpp or rospy. Nodes communicate with each other by passing messages or requesting services.

- Topics: It is the name given to a specific content of a message. They are published by nodes and a node that wants to receive this topic has to subscribe to it. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics.

- Master: The ROS master is a node that provides name registration and lookup to the rest elements on the environment. In other words, it works like a DNS server, all ROS nodes have to communicate each other asking their registration information to the master first.

About its operation, the ROS master acts as a nameservice. This means that the master node contains all the registration information of all the published topics. The different nodes of the robot communicate with the master to register and receive information about other registered nodes. The master informs to these nodes when the registration table information changes, which allows nodes to dynamically create connections as new nodes are executed. After the first connection with the master has been done, nodes communicate directly among them. Nodes that subscribe to a topic, request a connection from nodes that publish that topic, and stablish communication. Once the communication is stablished, the publisher node sends messages to the subscriber node. The messages are processed in a subscriber's callback function.

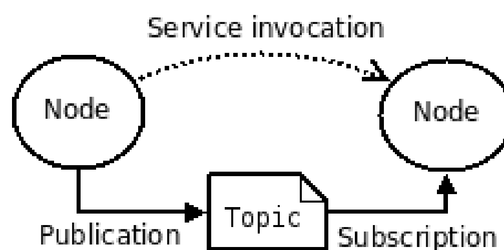

Figure 19. ROS communication [48].

As mentioned before, the software design is divided in two different hardware boards, Arduino and Odroid. The software design is also divided in five different modules or algorithms: localization, path planning, obstacle detection, obstacle avoidance and control.

5.2. Software architecture

The localization and control modules run on the Arduino board and are coded with the Arduino language. The path planning, obstacle detection and obstacle avoidance modules run in the Odroid board and are coded in C++ programming language using ROS libraries.

As for ROS, the system works with three nodes plus the master:

1. *robot_navigation_and_obstacle_avoidance* node: Formed by all the modules running on Odroid: path planning, obstacle detection and obstacle avoidance. It publishes a command and a parameter message, and subscribes to Pose2D and Point_Cloud.

2. *kinect2_bridge* node: Part of the package *iai_kinect2*. It is used to acquire data from Kinect and for calibrating the sensor. It publishes processed data to the obstacle detection module, and thereby, to the *robot_navigationa_and_obstacle_avoidance* node. It publishes Point_Cloud, although it publishes more messages that are not in use, such as the captured images or the cameras information.

3. *rosserial_arduino* node: This module permits the communication between an Arduino script and the master node. Includes the localization and control modules. It published Pose2D message, and subscribes to a command and a parameter.

Figure 20 represents the communication among ROS nodes, showing the messages that are being exchanged and how the modules interact among them.

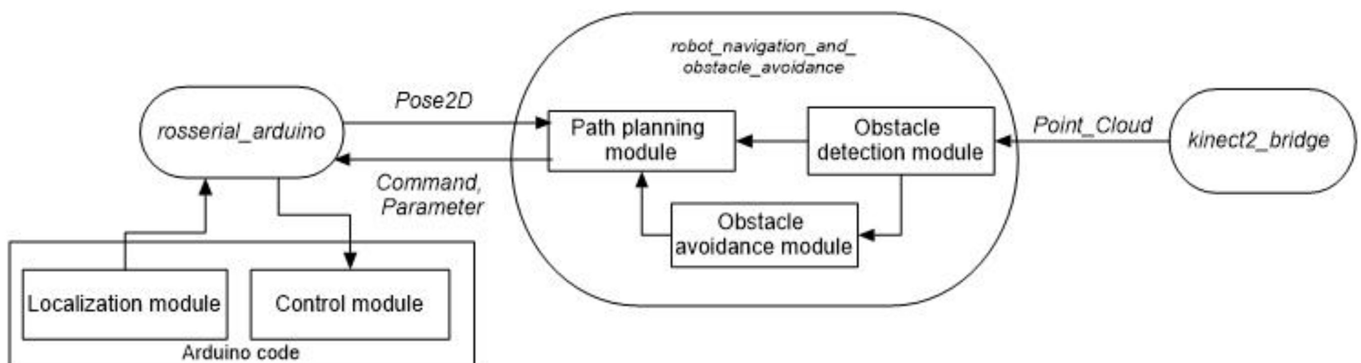

Figure 20. Software System general scheme

5.2.1. Localization

The localization module has three functions: acquires data from the sensors, processes the data measures to convert it into odometry and sends the message to the path-planning module. Depending on the robot current state, it has a different behaviour:

- If MOVE is enabled, the encoder data is acquired, but gyroscope data is not.
- When TURN LEFT or TURN RIGHT is enabled, the gyroscope calibration parameters are recomputed once, right before the turn starts. After that, gyroscope data is acquired and modified by the current calibration parameters. The calibration consists on computing a static bias to correct the error, and on a simple Kalman filter that was developed by *Gauss*' team of engineers. In the TURN state, the encoder data is not acquired.
- When STOP is enabled, no data is acquired from any of the sensors. The encoders' absolute counters are restarted to zero.

Sensor measurements are only performed at specific states. The reason is that it reduces the workload of Arduino, improving the loop speed and efficiency. Also, after performing certain actions, the sensors are restarted, reducing the cumulative error.

The processed position and heading are packaged in the form of a ROS geometry message of the type Pose2D. The message contains a 2D position in X and Y coordinates (in meters) and a heading orientation (in radians). Arduino sends this data message via serial port to the *rosserial_arduino* node, which publishes the message.

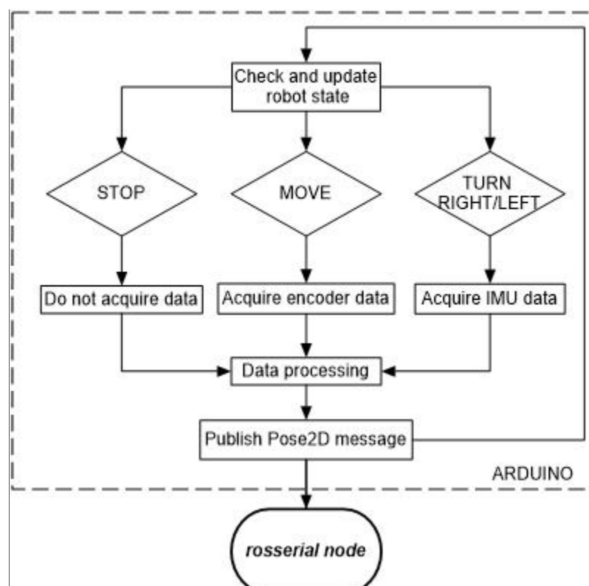

Figure 21. Localization module algorithm

5.2.2. Control

The control is performed at the Arduino. It receives two ROS standard messages from the *robot_navigation_and_obstacle_avoidance* node; a rover command (Command message) and the required heading (Parameter message). Depending on the received command, Arduino updates the robot state and sends the proper velocity command to the motor controller.

It uses two functions to interact with the motor; speedM1 for right side motor drive and speedM2 for left side motor drive. This functions require a speed in encoder counts, which is fixed to 2770 counts per second (approximately 0.1 m/s). Depending on the robot state, the value of the speed in both functions is changed.

- If the robot state is MOVE, the Arduino sets the same speed on both motor drives.
- If the robot state is TURN LEFT or TURN RIGHT, the Arduino executes a loop to turn the robot. First it computes the turning angle, which is the desired heading minus the current heading. After that, motor controller function sets a positive and a negative speed in each function; which one depends on the turning direction. During the loop, robot is reading from the gyroscope. The robot keeps turning until it meets the relative turning angle. Then it stops the vehicle and change the state to STOP.
- If the robot state is STOP, the Arduino sets velocity to zero on both motor drives.

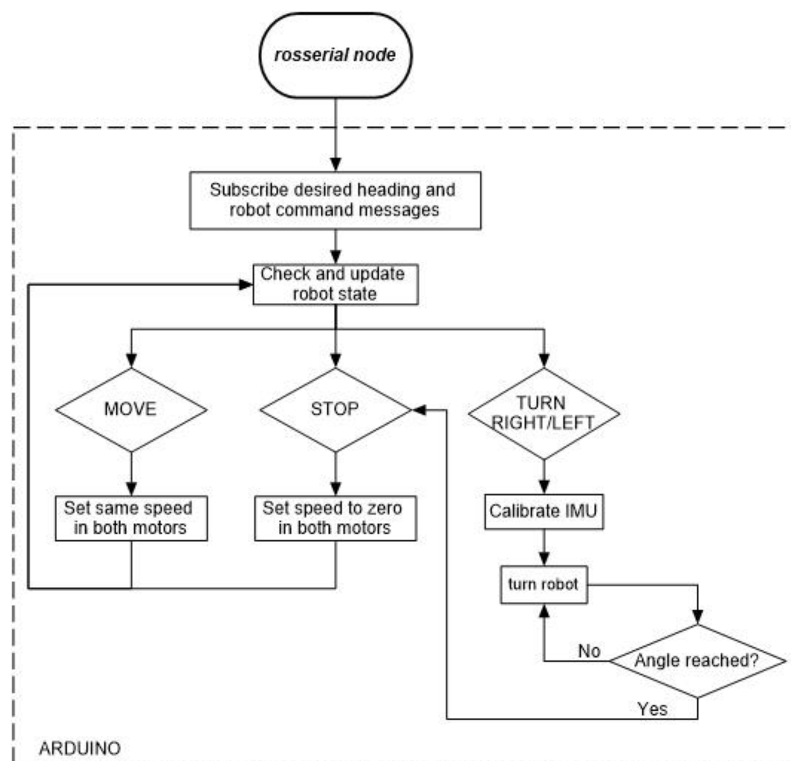

Figure 22. Control algorithm

5.2.3. Path planning

Path planning module decides the route that the robot has to follow and determines the current state of the robot. To do so, at each iteration of the code it establishes a robot command, which corresponds with the robot state.

The algorithm starts reading a waypoints file, and copy it into a vector of waypoints. The first and last position of the vector are the last waypoint and first waypoint of the route respectively.

The algorithm reads the last position of the vector, which is always the next waypoint to reach. This vector can be modified by the obstacle avoidance algorithm to re-plan the route. With the localization data received from the localization module, the path planning computes the distance and angle between the current position and the destination waypoint.

Theoretically, if the current waypoint and the destination waypoint coincide, the destination has been reached. The reality is that the localization has some error, a waypoint threshold is stablished at the destination waypoint. This threshold is a circle centred in the waypoint with a radius of 10 cm (Figure 23). When the robot localization reports that the robot is inside this area, is when the destination is considered to be reached.

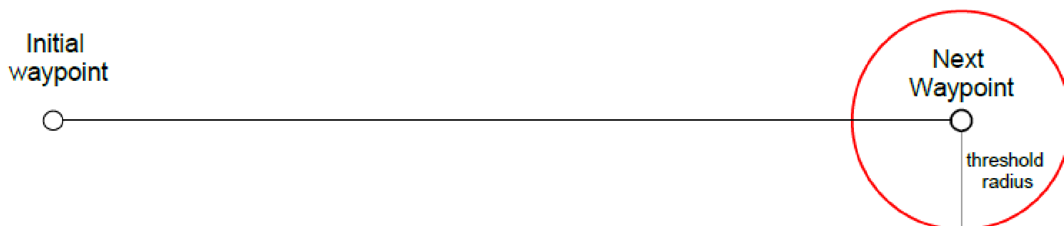

Figure 23. Waypoint threshold

In the case that the robot is inside the waypoint threshold, the robot stops and the waypoint is deleted from the vector. The next waypoint of the route is now situated in the last position of the vector. In the next iteration of the code loop, the algorithm treats this waypoint as it is the new coordinate destination.

On the contrary, if the destination has not been reached, the computed angle is used to determine the turning angle and the robot command. Since the heading can have some error, a $\pm 2^{o}$ threshold has been stablished (from now on, heading threshold). In addition, the algorithm determines whether or not the detection of obstacles is enabled:

- If the angle error is between 2° and -2°, the robot command is FORWARD. The obstacle detection is enabled.

- If the angle error is between 2º and 180º, the robot command is TURN RIGHT. The obstacle detection is disabled.
- If the angle error is between -2º and -180º, the robot command is TURN LEFT. The obstacle detection is disabled.

During turns, obstacle detection is not necessary since the robot is rotating without displacement.

This algorithm keeps iterating for each localization message received, until the waypoints vector is empty. In this case, the final destination has been reached, and the algorithm finishes.

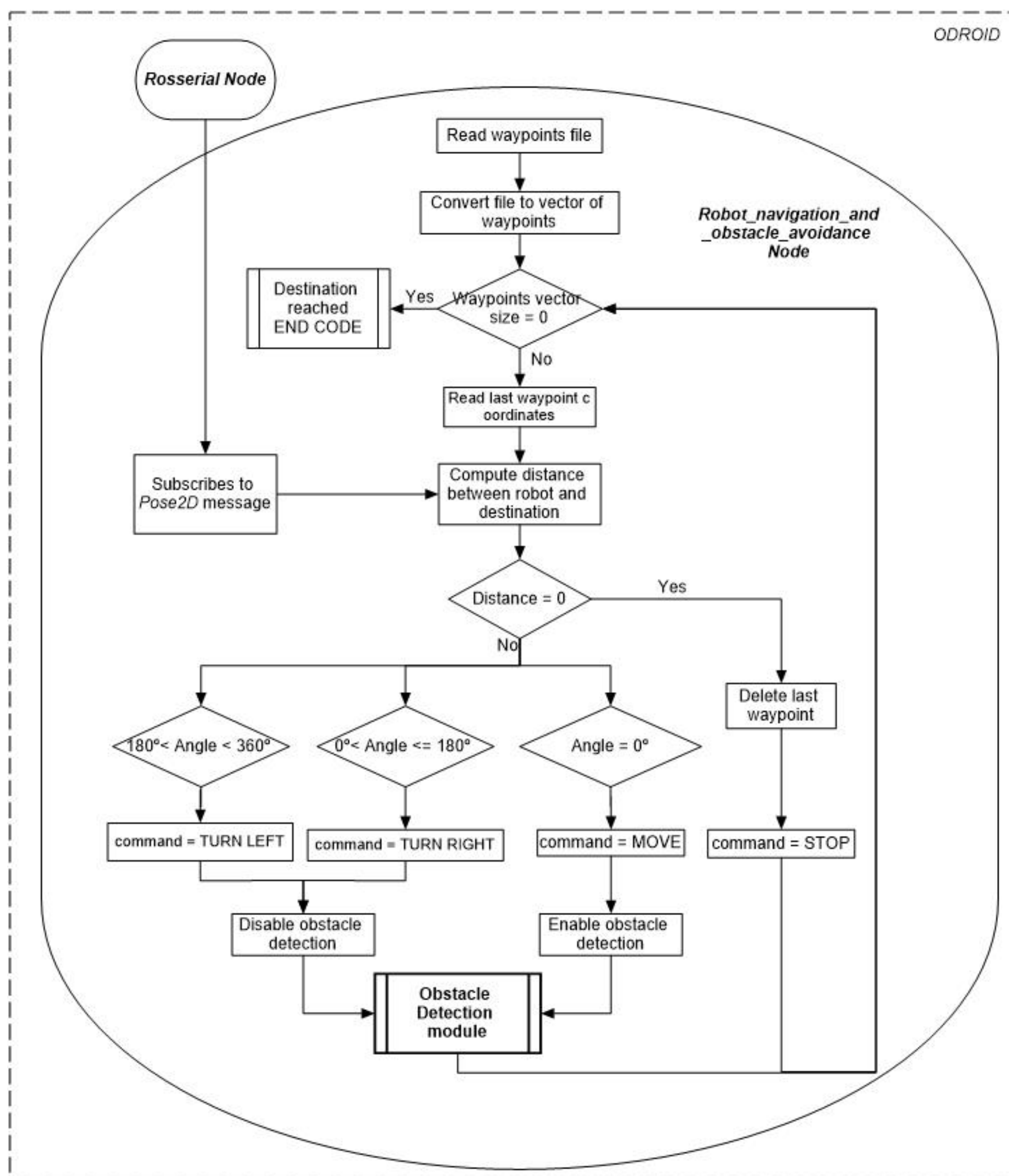

Figure 24. Path planning module

5.2.4. Obstacle detection

The obstacle detection module performs four tasks: it acquires data from the Kinect sensor, processes and selects the data of interest, determines if there is an obstacle in the path and enables/disables the obstacle avoidance algorithm.

First, the algorithm checks if the obstacle detection mode is enabled. If not, the robot is turning and obstacle detection is not required.

Data acquisition acquires data from Kinect via *Kinect2_bridge* node. This node provides multiple ROS messages, but the only one acquired by the obstacle detection module is the point cloud message. This message contains the position of the obstacles ahead in a 3D local coordinates, with centimetric resolution. The origin of coordinates and orientation of the acquired data is the Kinect coordinates and heading, and X, Y and Z coordinates are width, height and depth respectively.

Data processing consists of three different processes: it eliminates wrong data, defines the detection envelope and creates a 2D obstacle map vector from the 3D point cloud. Wrong data are false readings from Kinect, for instance, points detected beyond Kinect detection range.

Kinect detection range goes from 0.5 m to 4.5 m. Below and beyond that range, the robot dos not provide reliable data. The obstacle detection area is the section in front of the robot in which an object becomes an obstacle (Figure 25. Obstacle detection area). Everything outside this area is not considered a menace to the motion of the robot. To define the detection envelope, several considerations have to be taken into account:

- The upper limit is defined by the robot height, plus a security margin of 5 cm. Objects that can be traversed underneath, are not considered obstacles. The upper limit is 20 cm height from the centre of Kinect.

- The lower limit is the distance to the floor. For Kinect, the ground is like an obstacle. The lower limit is -10 cm height from the centre of Kinect.

- The laterals limits are defined by the robot width, plus a security margin of 5 cm per side. Objects that are not in the path of the robot, do not represent an obstacle. The total width of the envelope is 30 cm width.

- The maximum detection distance is the distance from the robot to the next waypoint.

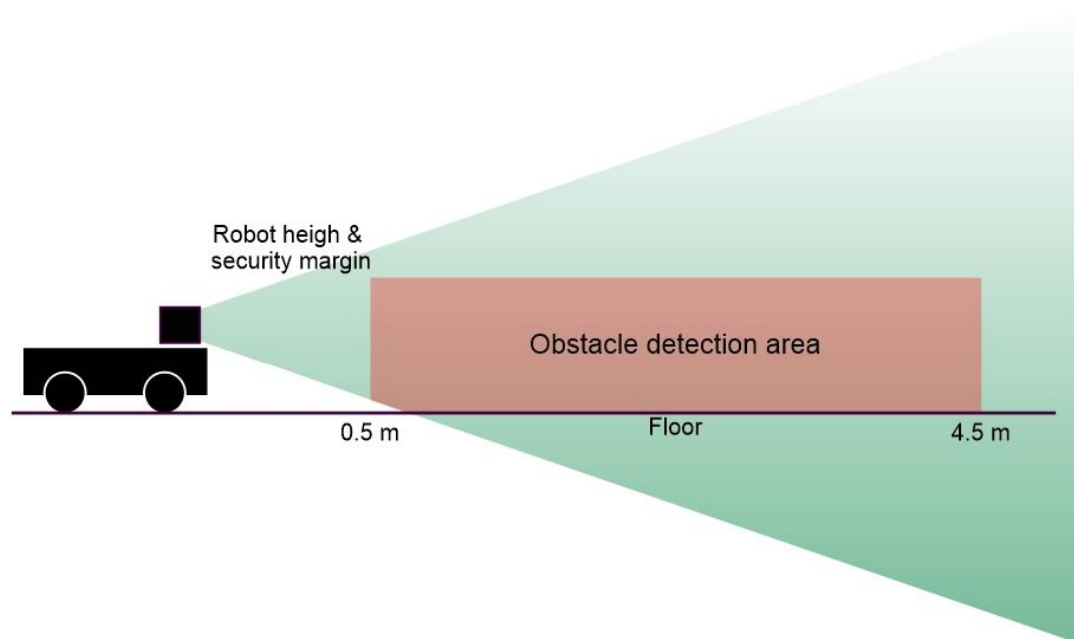

Figure 25. Obstacle detection area

The resulting 3D point cloud of the envelope, is converted into a vector that represents a 2D matrix. The X and Y coordinates of the point cloud correspond to the rows and columns of this matrix, and each value of the matrix correspond to the smallest depth value detected. The last part of the data processing is to create a new 2D vector, obstacles vector, from the 3D obstacles detected. The obstacles vector maximum length is the number of columns of the matrix. Each position of the vector is filled with the smallest depth value of each column. With this procedure, the most hazardous point of each width pixel in front of the robot is used to create the 2D vector of obstacles. The spatial resolution of each pixel decreases with more distance. The worst spatial resolution is 4 mm at 4.5 meters [49].

After the obstacle detection data processing, is the algorithm determines if an obstacle has been detected. If there is any obstacle inside the envelope, the robot stops immediately. In order to determine if it is a dynamic obstacle, it waits for the next point cloud data callback. If this callback contains no obstacles, a dynamic obstacle crossed in front of the robot, and it resumes its previous operation. If the obstacle is detected again, it is considered as a static obstacle. If a static obstacle is detected, the robot enables the obstacle avoidance algorithm.

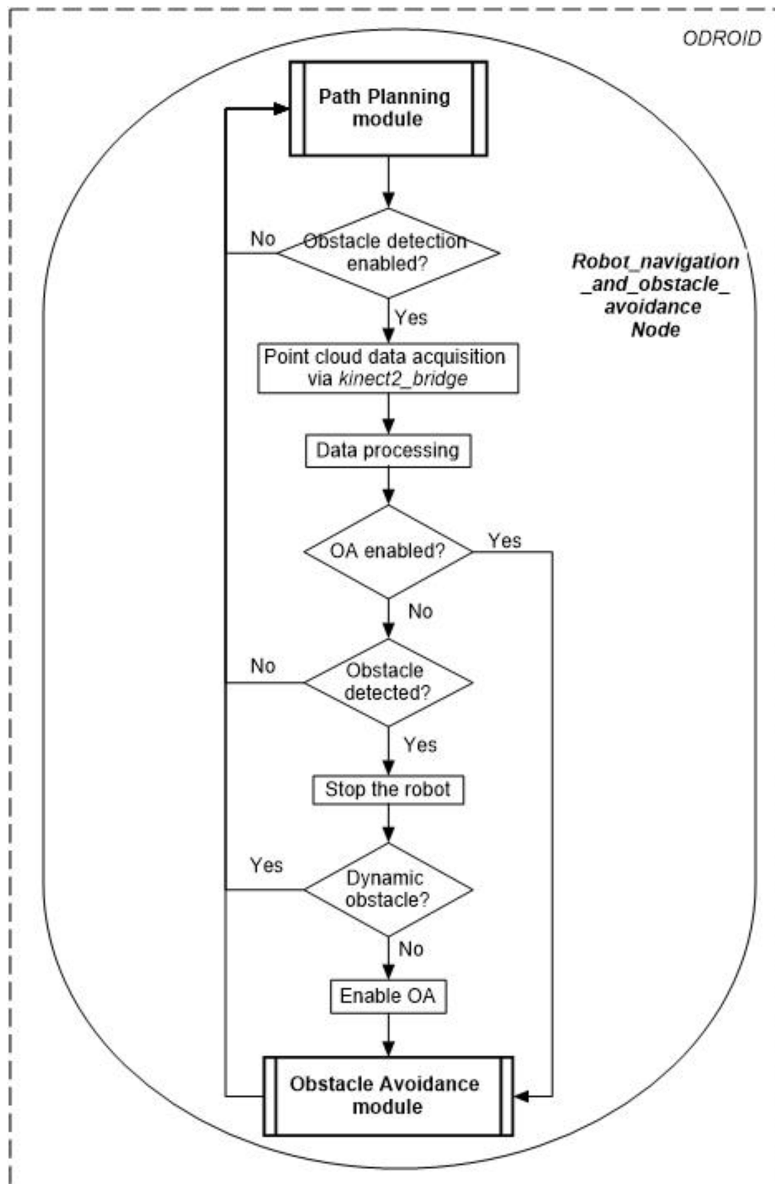

Figure 26. Obstacle detection module

5.2.5. Obstacle avoidance

The obstacle avoidance (OA) algorithm has been specifically designed for AROA, inspired by some of the state of the art obstacle avoidance techniques (2.3). The objective is to surround the obstacle, like in bug algorithm, but instead of following the border of the object the algorithm tries to find where the obstacle ends, similarly as in the "follow the gap" philosophy, to set a new waypoint beside it. To do so, the robot turns until the obstacle is no longer detected. Figure 27 explains graphically the AROA obstacle avoidance algorithm.

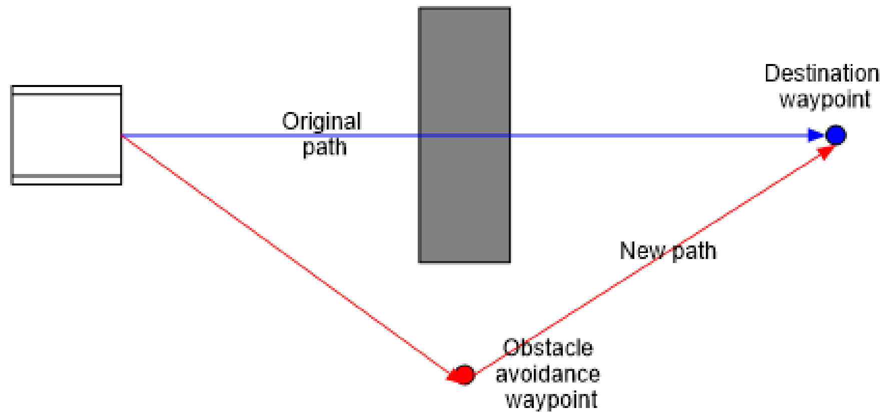

Figure 27. OA conceptual design

The algorithm only starts if the obstacle avoidance is enabled. It has two main objectives: determine the turning criteria and setting a new waypoint into the waypoint vector in order to modify the route.

The designed **turn criteria** is used to determine the direction of the turn to avoid the current obstacle. The 2D obstacle vector obtained by the obstacle detection module is used to stablish the turn criteria. Each point of the vector is treated by the algorithm as a single obstacle. Obstacles situated beyond the Kinect central point at the X axis (15 cm) are considered obstacles at left and obstacles above this points are considered to be at right.

It has been designed and implemented an algorithm to determine the turn criteria, depending on the distance to the closest obstacle and depending on the total number of obstacles detected. If there are obstacles below the 0.75 m depth threshold, they are considered to be near to the robot, so the turning criteria is determined by the position of the closest obstacle. If it is detected on the right, the turning criteria is left and vice versa. If there are obstacles further than the 0.75 m depth threshold, it is considered to be far and the turning criteria is determined by comparing the weight of the obstacles and the number of obstacles at right and left side. The weight of a set of obstacles depends on the quantity of obstacles detected at that side of the robot, and of the distance at which they are. Right and left weight are computed with the following formula:

$$Obstacle\ weight = 1 + \sum_{1}^{n} \frac{1}{obstacle\ distance_n} \qquad \textbf{(5.6)}$$

In the formula, n is the total number of obstacles in one side. More obstacles will sum up, but the distance at which those obstacles are, determines if they add more or less to the total value. Further values add less than closer ones. If right obstacle weight is greater than left one, the robot turn criteria is left and vice versa.

After determining the turn criteria, the algorithm changes the robot state to match the turning criteria, and executes a 10º turn in that direction. After the turn, the robot stops and enables the obstacle detection algorithm. If an obstacle is detected, the robot repeats the operation maintaining the turning criteria. If there are not obstacles detected, a new position coordinate is set to the waypoints vector.

The new waypoint is set at a distance equal to two times the distance to the nearest obstacle, without changing the heading. It is multiplied by two to ensure that the new waypoint is beside or beyond the detected obstacle. This new waypoint is set into the last position of the waypoint vector, in order to become the next waypoint of the route. After setting the waypoint, the obstacle avoidance is disabled.

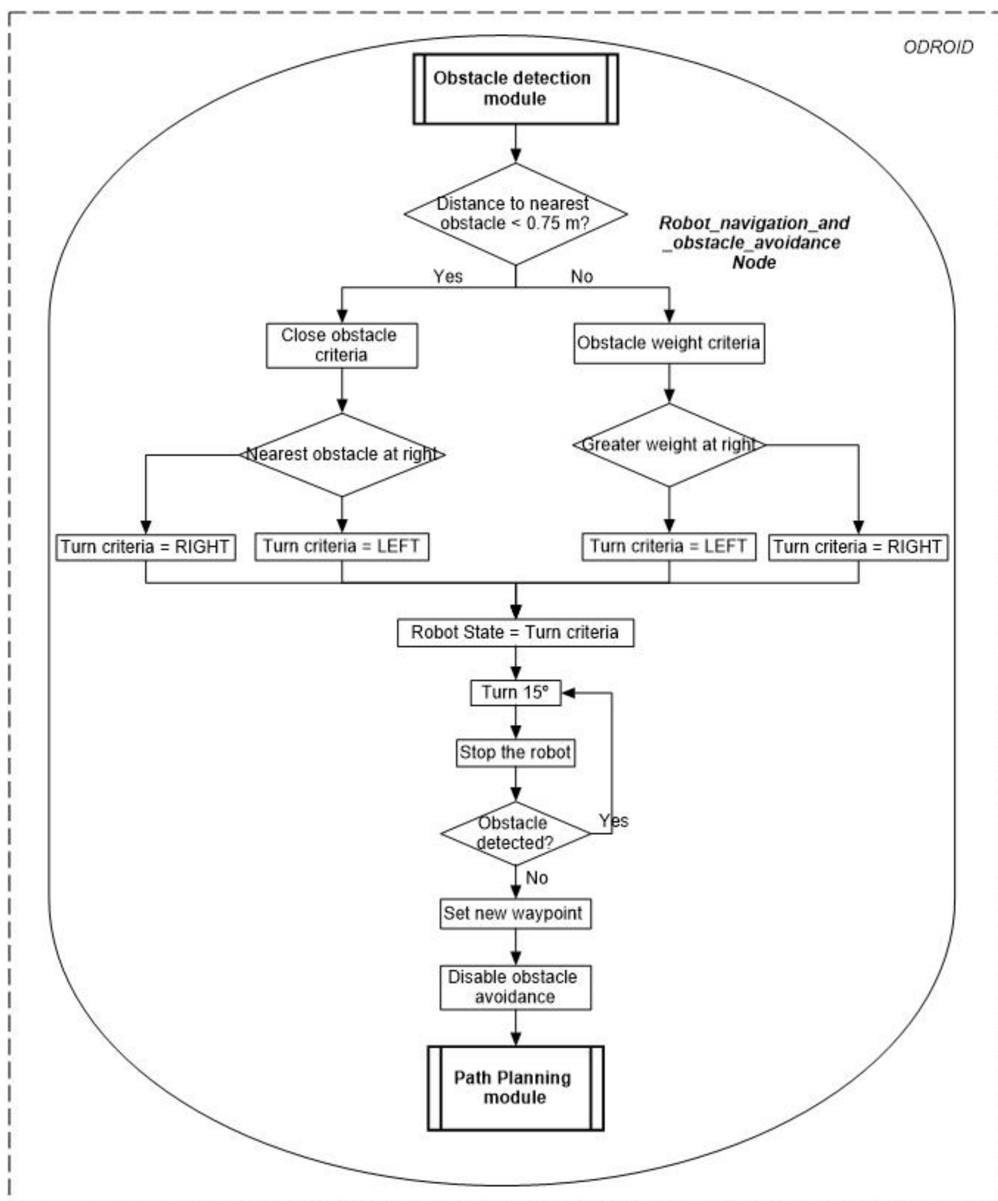

Figure 28. Obstacle avoidance module

CHAPTER 6. KINECT CALIBRATION

Kinect 2 is a sensors that uses ToF technology. It uses the IR emitter to project pulsed light beams, and an IR receiver that creates a depth image (this image is later used to create the point cloud). The problem with ToF light based sensors is that they are susceptible to light interferences, a type of random error. Kinect designers have thought on this issue and the device can use CSV (computer stereo vision), also known as stereo vision, to correct the depth error [50].

Stereo vision is a technique that takes advantage of the two cameras on Kinect (RGB and IR) to triangulate the depth to the pixels of image. In Kinect, Stereo vision depths are used as reference to characterise the ToF depth error at different distances and points of the image. Even tough, assuming that there are not depth error on stereo vision calibration is wrong. All cameras have a systematic error associated due to the camera geometry. Cameras construction and usage produce a misalignment between the camera objective and the sensors matrix causing image radial and tangential distortion (Figure 29). Radial distortion occurs because the image magnification decrease with distance from the optical axis. Tangential distortion occurs because camera lenses are not perfectly parallel to the imaging plane [51]. These errors can also be corrected with a proper calibration.

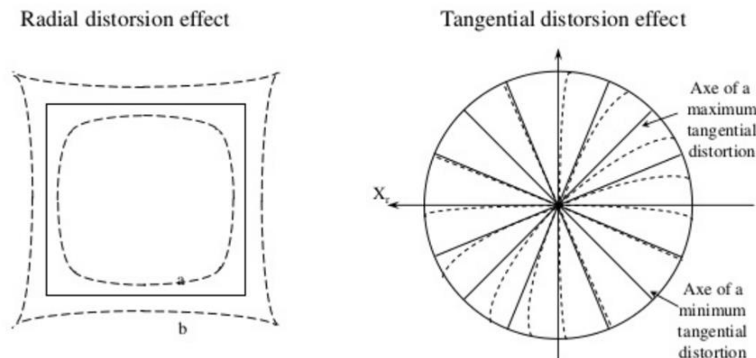

Figure 29. Radial and tangential distortion effects [52].

The objective of this chapter is to show the cameras geometric calibration and the results of the consequent depth calibration.

6.1. Camera geometric calibration

Camera calibration is the process of determining the geometric property of a camera such as its intrinsic and extrinsic parameters [52].

The intrinsic parameters of a camera define the internal geometry and camera optics, such as the focal length and the optical centre. They determine how the camera projects 3D points of the real world into a 2D frame.

The extrinsic parameters, in the case of *Kinect*, define the relation between the two cameras. They are related by a translation vector and a rotation matrix, being translation the distance between cameras and rotation the relative angle among cameras.

To perform the calibration of Kinect, a ROS package dependant of *iai_kinect2* has been used, *kinect2_calibration.* This package uses the OpenCV geometric calibration procedure adapted for Kinect, which is an open-source BSD-licenced library for computer vision algorithms [53].

This calibration is based on the Conrady-Brown model, which estimates the real position of a distorted pixel. The Conrady-Brown formulas for the radial factor are the following ones:

$$X_{corrected} = x(1 + k_1r^2 + k_2r^4 + k_3r^6) \quad \textbf{(6.1)}$$

$$Y_{corrected} = y(1 + k_1r^2 + k_2r^4 + k_3r^6) \quad \textbf{(6.2)}$$

And the Conrady-Brown formulas for the tangential distortion are the next ones:

$$X_{corrected} = x + [2p_1xy + p_2(r^2 + 2x^2)] \quad \textbf{(6.3)}$$

$$Y_{corrected} = y + [p_1(r^2 + 2y^2) + 2p_2xy] \quad \textbf{(6.4)}$$

Where x and y are the coordinates of the pixel and $x_{corrected}$ and $y_{corrected}$ are the estimated coordinates after correction. The r is the radius in pixels of the image at which the specific coefficient applies. The k and p parameters are the radial and tangential distortion coefficients respectively.

As can be seen, finding the k and p parameters is the basis of the calibration, which is done through basic geometrical equations. These equations are found by comparing pictures of known pattern, such a chessboard, at a specific distance; since the pattern is known, the difference between the expected and the real result, gives a correction equation.

The calibration consists on taking different pictures of this pattern; each found pattern results in a new equation, and more equations mean that the distortion coefficients reflect better the reality.

The complete calibration of Kinect requires to calibrate the intrinsic of each camera (IR and RGB) and the extrinsic of both, making a total of three calibrations. In all cases a chessboard pattern of 5x7x0.03 is being used, with a distance between squares of 3 cm.

The procedure to calibrate consists on placing two tripods, one holding the Kinect and the second holding the chessboard pattern, and locating them one in front of each other as can be seen in Figure 30.

Figure 30. Calibration process

The objective of each calibration is to take pictures of the chessboard at two different distances, at different positions and orientations and with the pattern vertical and horizontal, to ensure that both radial and tangential parameters are taken into account. It is recommended to take at least 100 pictures for a good calibration; in the case of this particular calibration around 120 pictures have been taken at two different distances.

After taking the pictures, it is necessary to run the calibration program in order to generate a calibration file containing the distortion coefficients matrix. With the parameters of the calibration file, *kinect2_bridge* will correct the error automatically in any picture taken by Kinect.

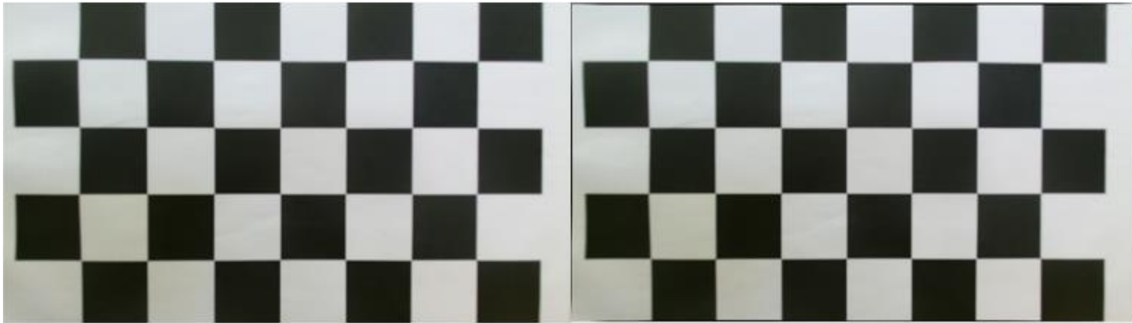

Figure 31. Uncalibrated and calibrated image comparison

In Figure 31. Uncalibrated and calibrated image comparison, the correction can be noticed on the edges of the picture, where the distortion is greater.

6.2. Depth calibration

As stated before, Kinect uses stereo vision to correct the depth error on ToF measures. The calibration consists on estimating the bias of the ToF measures using the stereo vision measured distance as a reference. This bias is later used to correct all the measures taken with ToF. The following section presents the depth error corrected by the calibration.

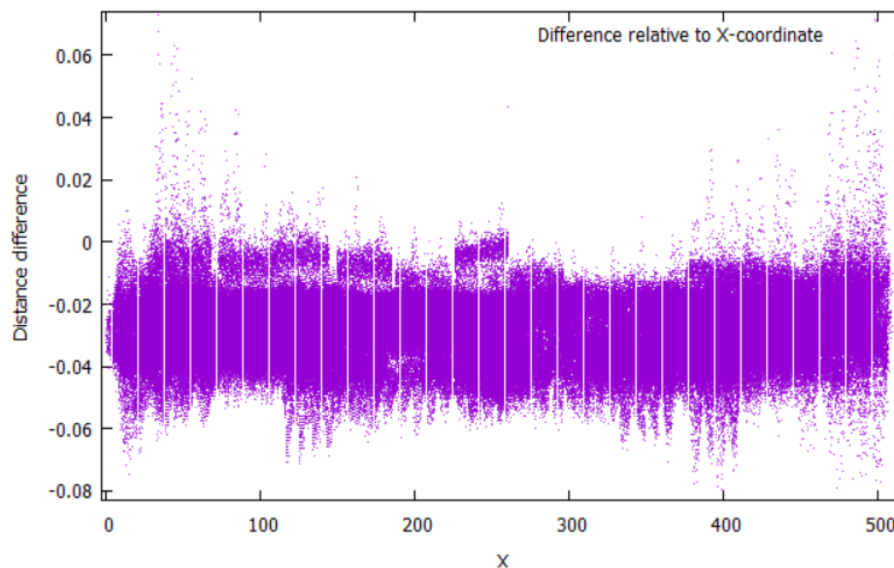

Figure 32. Distance difference relative to X-coordinate

Figure 32 presents the measured depth error in the X axis. As it can be seen, the depth data has a bias of -0.024 m approximately.

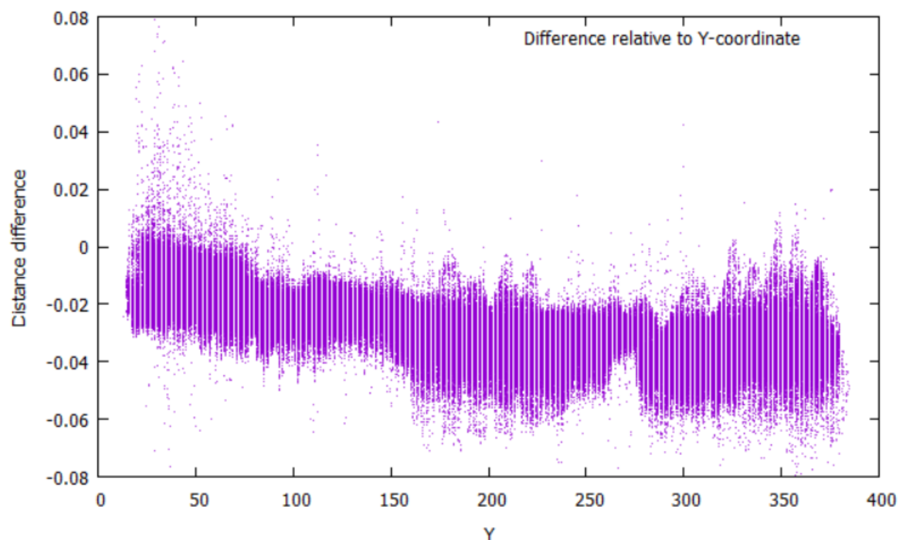

Figure 33. Distance difference relative to Y-coordinate

Figure 33 shows the measured depth error on the Y axis. As it can be seen, apart from the bias it has been detected a small drift. The bias for this axis is 0.03 m. This drift is not contemplated by the *Kinect2_bridge* calibration, and it is not corrected.

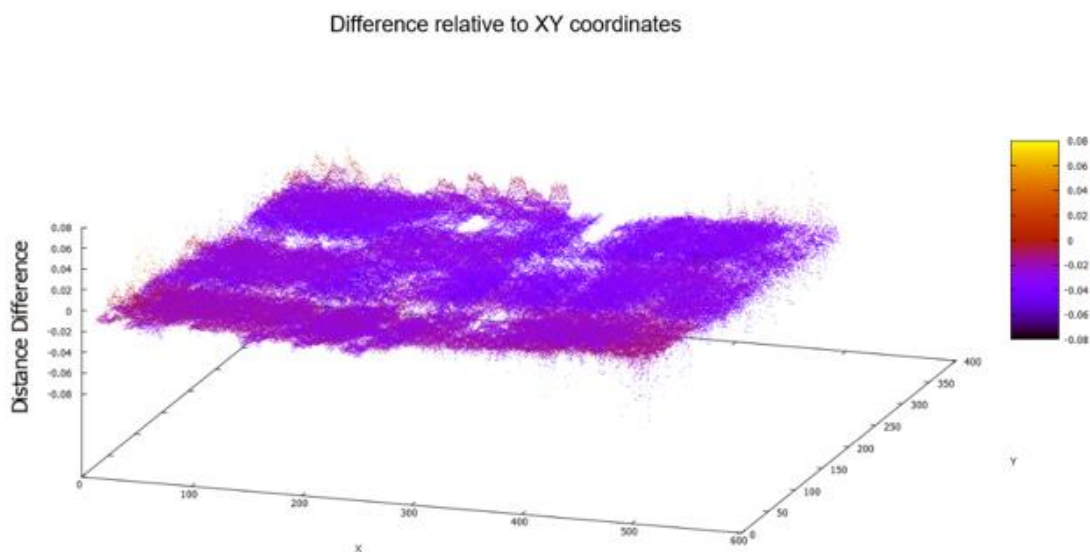

Figure 34. Distance difference relative to XY-coordinate

In Figure 34 it is represented the error of a whole 2D matrix frame. A constant bias should provide a frame of the same colour (which represents error) at all points. The fact that Y points near to 0 have a different colour than values at 400, reaffirms the fact that there is a small drift on Kinect.

CHAPTER 7. TEST AND VALIDATION

To validate the AROA system and the obstacle avoidance algorithm, a series of tests have been designed. The aim of this tests is to determine if the requirements are satisfied. The time, the travelled distance and the accuracy of the robot are the key elements to determine the efficiency of the path planning algorithm. In addition, the behaviour against obstacles is decisive to determine the efficiency of the obstacle detection and avoidance algorithm.

The tests have been designed to evaluate the different system parts, and are classified into three validation groups: path planning, obstacle detection and obstacle avoidance.

7.1. Path planning validation

This section evaluates the efficiency of the path planning module. One test has been designed to evaluate all the aspects of the algorithm.

7.1.1. Path planning test

Evaluating the efficiency of path planner requires to evaluate if the robot is able to move through a given set of waypoints. This test is designed to evaluate the efficiency of the path planner without the presence of obstacles. Path planning algorithm receives data from the localization module, so the efficacy of their response is also being evaluated.

The path planning test consists on a U-shaped circuit. The robot has to perform this route and arrive to the final destination, marked in red in the figure. The path consists of three waypoints, and two turns of 90º (Figure 35 and Figure 36). The test was planned to be repeated five times, but it has been repeated eight times.

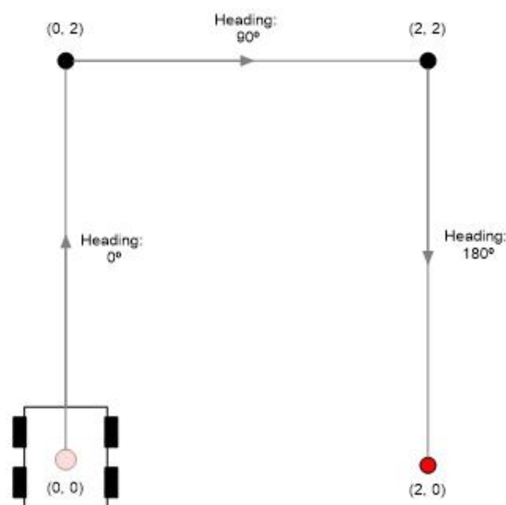

Figure 35. Ideal path

Figure 36. Path Planning test layout

Of the eight tests, the robot arrived to the final destination five times. In the other four tests, the robot did not arrive because the localization was not immediately updated after the robot turned for the first time. This makes impossible for the path planning to guide the robot to the final destination because according to the localization, it has no moved. The presented results are those obtained by the five tests that arrived to the final destination.

Tables 7.1 and 7.2 present the localization of the robot at the last waypoint. At this tables, the estimated position and heading are those detected by localization sensors. The real position is the actual position of the robot, measured externally with a tape measure.

At the tables the problems with the localization are noticeable. The difference between the sensors measurements and the real position, which is the localization error, is generally high. For instance, the mean of the average of the destination absolute error is 0.24 m for the X axis. This is a big error because, albeit the path planning module is working properly, the robot is not really inside the waypoint threshold. It can be easily seen in Figure 37.

From the point of view of the path planning module, the result of this five tests are satisfactory, since it has guided the robot according to the localization estimated by the sensors. Even tough, test number five has been analysed in detail.

Table 7.1 Final coordinates

	Final coordinates (m)			
	X		Y	
	Real	Estimated	Real	Estimated
Test nº1	1.26	1.02	0.02	0.03
Test nº2	1.26	1.02	0.02	0.03
Test nº3	1.24	1.18	0.18	0.09
Test nº4	1.26	1.02	0.02	0.03
Test nº5	1.18	1.01	0.02	0.02
Average	1.24	1.02	0.05	0.04

Table 7.2 Path Planning heading test results

	Final heading (º)	
	Real	Estimated
Test nº1	7.9	5
Test nº2	6.9	5
Test nº3	3	1.1
Test nº4	7.1	5
Test nº5	5.5	3.7
Average	6.1	4

The results presented in table 7.3 show the localization of the rover and the command emitted by the path planner. Figure 37 show the route that the robot has performed according to the localization and to real measurements.

As can be noticed, each time the rover is inside the waypoint threshold of the destination waypoints (0.1 m) it stops. For instance, when the destination was (0, 2), the robot stopped at (0, 1.93). It also can be noticed that after each STOP command, a new waypoint is introduced. Is also noticeable how after each STOP, the robot computes the turning angle and emits a TURN (RIGHT, for this case) command.

The heading threshold (±2º), is also meet in all the examples. For instance, at the first turn the planned angle was 86º, and the robot has stopped at 85º. In this particular test, the robot has had made a corrective turn at the last segment of the route. It went from angle 178º to -178º, a 4º turn to the right. At this coordinates the path planning has recalculated the route because the heading was beyond the heading threshold.

In conclusion, the path planning module is working as it should, but better localization data would improve the robot general operation.

Table 7.3 Path planning results

Next waypoint (m)	Localization (m and º)	Rover Command
(0, 2)	(0, 1.93, 0º)	FORWARD
(0, 2)	(0, 1.93, 0º)	STOP
(1, 2)	(0, 1.93, 0º)	TURN RIGHT (86º)
(1, 2)	(0, 1,93, 85º)	STOP & FORWARD
(1, 2)	(0.99, 2.02, 85º)	STOP
(1, 0)	(0.99, 2.02, 85º)	TURN RIGHT (179º)
(1, 0)	(0.99, 2.02, 178º)	STOP & FORWARD
(1, 0)	(1.03, 1.02, 178º)	STOP
(1, 0)	(1.03, 1.02, 178º)	TURN RIGHT (-178º)
(1, 0)	(1.03, 1.02, -178º)	STOP & FORWARD
(1, 0)	(1.01, 0.02, -178º)	STOP

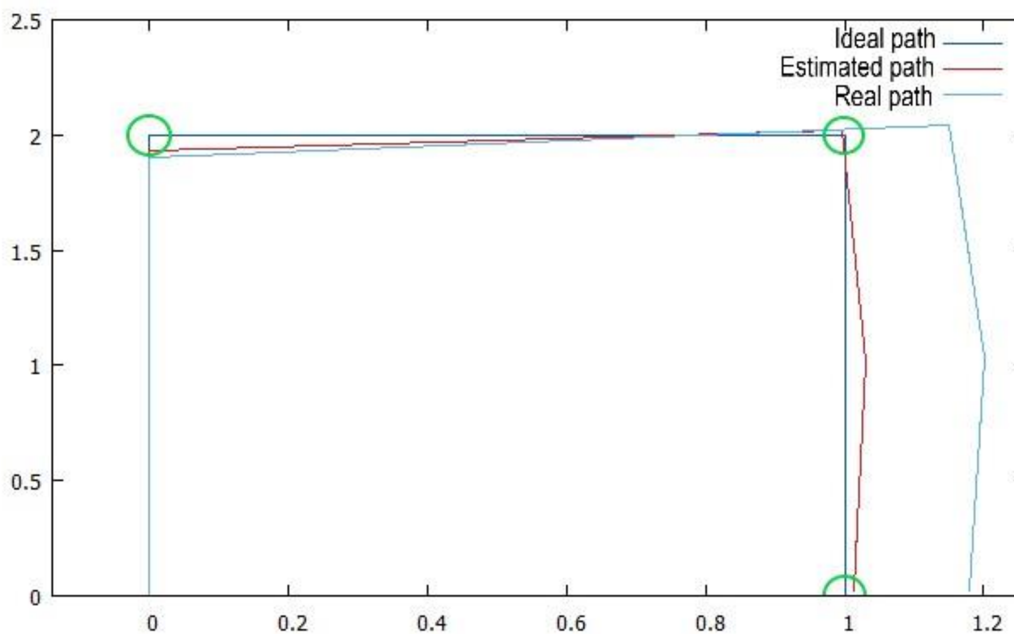

Figure 37. Path Planning test results

7.2. Obstacle detection validation

This section aims to determine the efficiency of the obstacle avoidance system. To do so, the acquisition and processing of the point clouds is evaluated. Four different tests have been designed: frame rate, reaction time, false data probability and dynamic obstacle.

7.2.1. Frame rate test

Kinect detects, processes and publishes a point cloud at a certain frame rate. This test aims to determine the frames per second (FPS) provided by Kinect.

For the test, 500 point clouds have been taken and the elapsed time has been measured. The total number of point clouds divided by the elapsed time provides FPS. The test has been performed three times.

$$FPS = \frac{nº\ of\ point\ clouds}{Total\ time} \quad \textbf{(7.1)}$$

Table 7.4 shows that the FPS average value is 0.5, which represents a point cloud every two seconds. This value is a low value for an obstacle detection application.

Table 7.4 Frame rate test

	Total acquisition time (s)	FPS
Test nº1	1013	0.49
Test nº2	1120	0.5
Test nº3	1120	0.5

7.2.2. Reaction time test

The reaction time is the elapsed time that starts when an obstacle appears in the path of the obstacle is detected and ends when the robot stops.

This test aims to stablish the reaction time of the robot. Kinect has been laid out in a static test bench, but with the wheels in motion. For each test and an obstacle has been placed in front of the robot. The timer is started at this instant. The robot wheels eventually stop, and this is the moment when the timer is stopped. This time is the reaction elapsed time. The test has been performed 15 times.

The results of the test prove a mean reaction time of 3.43 seconds. At the speed of the robot, 0.1 m/s, it means that before an obstacle is detected the robot has advanced 0.35 m. Because the near detection range of Kinect is 0.5, it can be assured that the robot stops before colliding with the obstacles at all times.

Table 7.5 Reaction time test results

Test nº	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Elapsed time (s)	1.5	2.9	4	3.5	4.5	2.5	3.6	3.1	4.3	3.4	3.8	4.1	2.7	4.1	3.7

7.2.3. False data probability tests

Sometimes, a detection iteration from Kinect emits a false positive. Also, there is the possibility that due to light interferences a false negative is emitted. This tests aim to determine the false positive and false negative probability of Kinect. A false positive is considered to be a point cloud reporting a detected obstacle when there was no obstacle. A false negative is considered to be a point cloud not reporting a detected obstacle that is in front of the robot.

The test purposes is to evaluate the number of false positives and negatives detected in 500 point clouds. For the test, Kinect has been laid out in a static test bench with two different scenarios. With no obstacles ahead, first scenario, false positives are measured (Figure 38). For an obstacle within the detection range, second scenario, false negatives are measured (Figure 39). Each test scenario has been performed three different times for each layout

Figure 38. False positive test layout

As can be seen from the Table 7.6, the mean false positive error is 1.2%, which is small. Even tough, it has been considered in the design, and whenever the

robot detects a false positive, it considers it a dynamic obstacle. This way, the obstacle avoidance algorithm is not triggered.

Table 7.6 False positive error test results

	Nº of detected false positives	Error produced by false positives
Test nº1	6	1.2 %
Test nº2	4	0.8 %
Test nº3	8	1.8 %

Figure 39. False negative test set up

The results of the test showed that the obstacle was detected by all the point clouds, proving the false negative error is 0%.

7.2.4. Dynamic obstacles test

Dynamic obstacles are those that are not part of the environment, but a mobile object. This test consists on a person crossing in front of the robot while it is in motion, to evaluate the response. The purpose is to evaluate if the robot continues the route after encountering a dynamic obstacle without enabling obstacle avoidance. The test has been performed ten different times.

The results of the test showed that the obstacle was successfully detected and that the robot stopped and resumed its motion after a few a seconds a 100% of the times. This confirms that the obstacle detection algorithm is robust against dynamic obstacles.

7.3. Obstacle avoidance validation

The objective of this tests is to validate the system reaction against obstacles. The turn criteria and the general operation are being evaluated.

7.3.1. Turn criteria test

The turn criteria, explained at 5.2.5, is the parameter that determines the direction at which the robot turns to avoid the current obstacle.

To evaluate the turn criteria, a box has been situated at the coordinates (0, 0.9) (left) for five tests, and at (0, 1.1) (right) for five more (Figure 40). The objective is to evaualte if the robot turns to the correct direction, which is the oposite to the current obstacle position.

Figure 40. Turn criteria test set up (right turn)

In the five tests in which the obstacle was at left, the turn criteria was considered correct 100% of the times. The same happened for the five tests where the obstacle was located at right. Thereby, the turn criteria is guiding the robot correctly.

7.3.2. Obstacle avoidance test

This test aims to test if the robot is capable of fixing a new waypoint to avoid the obstacle, and reach the final destination.

At this test the robot has to arrive at a waypoint situated at two meters in straight line from the origin of coordinates, the waypoint (0, 2). To do so, it has to calculate a new waypoint to avoid the obstacle. A box (0.3 m x 0.2 m) has been situated at the coordinates (0, 1) as obstacle. The turning criteria has been fixed to turn right.

It is important that all the new waypoints fixed by the obstacle avoidance module are situated beyond 0.35 m at X axis. This coordinate represent the box width

plus the robots width (with a security margin). Any waypoint below this value, will cause a collision of the robot. The test was planned to be repeated five times, but it has been repeated ten times.

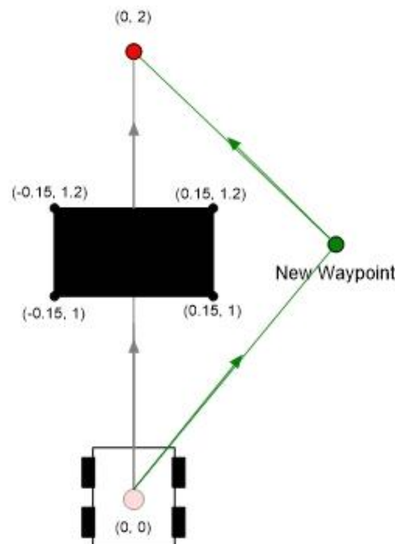

Figure 41. Ideal obstacle avoidance

Figure 42. Obstacle avoidance test set up

Of the ten tests, the robot arrived to the final destination five times. The problem with the localization stated in the Path planning test (7.1.1) has affected the other five tests, where the robot did not arrive to the final destination. The presented results are those obtained by the five tests that arrived to the final destination.

Table 7.6 shows the results of the test. The localization is the point at which the robot has stopped because the obstacle has been detected. The new waypoint is fixed by the algorithm. As can be seen from the results all the new waypoints

are beyond the 0.35 m on the X axis. This ensures that the algorithm to determine the new waypoints is valid for this scenario.

Table 7.6 False positive error test results

	Localization (m)	**New waypoint (m)**
Test nº1	(0, 0.26)	(0.58, 1.16)
Test nº2	(0, 0.20)	(0.37, 1.54)
Test nº3	(0, 0.20)	(0.46, 1.19)
Test nº4	(0, 0.19)	(0,57, 1.25)
Test nº5	(0, 0.20)	(0.44, 1.51)

CHAPTER 8. CONCLUSIONS

The main objective of this project, which has been evolving an autonomous rover with 3D obstacle avoidance capability, has been accomplished. Even tough, the whole system does not work correctly because of problems with the localization module. The requirements (3.1) have been fulfilled:

- The system obstacle avoidance module has been upgraded to detect 3D obstacles.
- The system has a new 3D obstacle avoidance module.
- Obstacle detection, obstacle avoidance and path planning modules are now based on ROS. Localization and control modules are not ROS based (because they are located on the Arduino), but are compatible with the ROS message system.
- The total cost of this prototype is 473 €.

The system has been endowed with new **hardware** needed to achieve the stated requirements. The RGB-D camera Kinect 2 has given good results, providing depth information below the centimetre in point cloud form. Even tough, the frame rate of the device has proven much slower than expected, 0.5 FPS. The reason is that the library that converts images to point cloud data, OpenGL, is not supported by Odroid's GPU drivers, and it has to rely on Odroid's CPU [54]. This conversion requires a lot of image processing capacity, a type of process that the CPU does not perform as fast as the GPU would. Kinect 2 depth error has been characterized and the sensor has been calibrated.

The **ROS** framework has been installed and configured in all the system. This framework, although hard to understand at first, has allowed for faster development of the software architecture. The most notable example is *iai_kinect2* module, which acquires image and data from Kinect 2. Data acquisition from Kinect 2 is not trivial, and would have cost a lot of time of the project. The same goes for the calibration of the sensors. This package, which contained the nodes *kinect2_bridge* and *kinect2_calibration*, has reduced the workload and the complexity of the project significantly. Another advantage of having ROS running on all the system is that now it is standardized. This means, for instance, that if in the future it is required to change Kinect 2 for another RGB-D sensor, the messages that this new sensor will be delivering will be the same used by Kinect 2. This saves a lot of future developing time and simplifies the upgrading process of the platform.

Tests have been performed to evaluate the path planning, obstacle detection and obstacle avoidance modules.

The **path planning tests** have determined that the module works properly. The path planner computes the correct orders and drives the robot to the required destination. Even tough, the path planning module is affected by the localization module, which has to be improved because once in a while it provides wrong positioning of the robot. It has been noticed that sometimes encoder data provided by the motor controller is corrupted or it is simply not provided while in motion. This causes a general failure of the system at some of its operations.

The **obstacle detection tests** have proven that the reaction time is 3.43 s. This value is valid for the current robot motion speed, 0.1 m/s. In order to increase the speed of the robot, the reaction time should be decreased. Otherwise, the robot will collide with the obstacle.

The **obstacle avoidance tests** show that the implemented algorithm provide good results for the proposed scenario. Even tough, to characterize the algorithm general efficiency, more tests need to be performed in more complex scenarios (for example, with more obstacles). In addition, the algorithm has room to be improved. For instance, the current algorithm forces the robot to turn to search for obstacle free areas. An improvement would be to use all the data from the Kinect point cloud, and not only data inside the obstacle detection area. Having more information of the environment ahead of the robot will help to determine a suitable obstacle free area without turning the robot.

As can be seen, the robot performs all the desired tasks, but there is always room to improve the system. Some proposals have been made to **future developing**.

The erratic behaviour of the **localization** has to be solved. More research should be performed on this issue, because it harms the global operation of the system.

The usage of Kinect 2 to perform **SLAM** would improve the general system operation. SLAM, as has been defined in the State of the Art (2.2.1), consists on trying to simultaneously localize the sensor with respect to its surroundings, while at the same time mapping the structure of that environment. This would help solving the localization issues. In addition, generating a map of the environment would help immensely the obstacle avoidance module, because map based algorithms such as A* or D* could be used.

The **low frame rate** of Kinect should be improved. Right now Odroid is using CPU to process point clouds, but with compatible drivers, it could rely on its GPU. This would increase the frame rate.

The **IR sensors** used by *Gauss* should be integrated. This would provide the system detection capabilities under 0.5 m. They have not been implemented in the new design because the new *Rosserial* library increased considerably the usage of the Arduino memory (the code occupies 76% of flash memory and an 80% of the dynamic memory, which produces stability issues). A ROS message containing IR data cannot be defined in AROA's Arduino, without overloading the system. The solution would be to use a new microcontroller with more memory. For instance, Arduino Due would increase the flash memory from 32 KB to 512 KB and the dynamic memory from 2 Kb to 96 KB [55] [56].

The **ROS navigation stack** (2.1) could be implemented. This tool has not been considered in this project for two main reasons. First, it does not allow an easy implementation of a new obstacle avoidance algorithm. Second, it requires full message standardization of ROS, which with the current Arduino is impossible because of the lack of memory issue. Even tough, if a new microprocessor is used, it is an interesting option that might be explored, because it integrates the five main modules that AROA has (localization, control, path planning, obstacle detection and avoidance) and introduces the usage of cost maps.

Finally, the **interaction between user and platform** can be enhanced. Right now, the initialization of the robot requires four different terminals to be opened. The system could be controlled by a single computer program or a cell phone app with graphical interface that allows to initiate all the subsystems at the same time. This program could also be useful to modify the waypoint set of the robot dynamically. Also, to read data in real time from the robot, an Ethernet cable is required. In the future, a Bluetooth or Wi-Fi communication may be integrated.

BIBLIOGRAPHY

[1] H. Ruslan, “https://www.aivoke.com/news/state-of-the-art-robotics/” [Online].
[2] “http://www.robotpark.com/All-Types-Of-Robots” [Online].
[3] “https://en.wikibooks.org/wiki/Robotics/Types_of_Robots/Wheeled” [Online].
[4] “http://www.irobot.co.uk/” [Online].
[5] «https://mars.nasa.gov/msl/» [Online].
[6] “https://www.amazonrobotics.com/” [Online].
[7] “https://www.clearpathrobotics.com/” [Online].
[8] “https://waymo.com/” [Online].
[9] “https://www.engineersgarage.com/tutorials/difference-between-microprocessor-and-microcontroller” [Online].
[10] “http://www.infineon.com/cms/en/about-infineon/press/market-news/2011/INFATV201110-003.html” [Online].
[11] “https://www.arduino.cc/” [Online].
[12] “https://www.raspberrypi.org/” [Online].
[13] “https://www.xilinx.com/products/silicon-devices/fpga/spartan-6.html” [Online].
[14] “http://www.ros.org/about-ros/” [Online].
[15] “http://wiki.ros.org/navigation” [Online].
[16] “http://rosindustrial.org/” [Online].
[17] “http://www.pbs.org/wgbh/nova/next/body/bioinspired-assistive-devices/” [Online].
[18] “https://ultrasonicsensor.wordpress.com/,” [Online].
[19] I. R. Nourbakhsh and R. Siegwart, Introduction to Autonomous Mobile Robots, Cambridge, Massachusets: The MIT Press, 2004.
[20] “http://www.wisegeek.org/what-is-an-infrared-sensor.html” [Online].
[21] K. Litomisky, “Consumer RGB-D Cameras and their applications”, 2012.
[22] “https://en.wikipedia.org/wiki/Point_cloud” [Online].
[23] “https://www.skybrary.aero/index.php/Dead_Reckoning” [Online].
[24] “https://www.dimensionengineering.com/info/accelerometers” [Online].
[25] “https://www.kudan.eu/kudan-news/an-introduction-to-slam/” [Online].
[26] R. D. Khan , M. Zohaib, M. Pasha, R. A. Riaz, N. Javaid and M. Ilahi, “Control Strategies for Mobile Robot with Obstacle Avoidance”.
[27] “http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html
#dijkstras-algorithm-and-best-first-search,” [Online].
[28] “http://www.frc.ri.cmu.edu/~axs/dynamic_plan.html,” [Online].
[29] A. M. Javier Arribas, “OpenIn Location Lab”, 2016.
[30] “http://www.robotplatform.com/knowledge/Classification_of_Robots/wheel_control_theory.html” [Online].

[31] "http://www.robotshop.com/en/lidar.html" [Online].
[32] "https://www.microsoft.com/en-us/store/d/kinect-sensor-for-xbox-one/91hq5578vksc" [Online].
[33] "http://wiki.ros.org/Sensors" [Online].
[34] "http://rosindustrial.org/news/2016/1/13/3d-camera-survey" [Online].
[35] "https://msdn.microsoft.com/en-us/library/jj131033.aspx" [Online].
[36] "https://click.intel.com/realsense.html" [Online].
[37] "https://www.amazon.es/Microsoft-Sensor-Kinect-Xbox-360/dp/B004RFC94O/ref=sr_1_3/262-9485731-7350541?ie=UTF8&qid=1496179453&sr=8-3&keywords=sensor+kinect+xbox+360" [Online].
[38] "https://www.microsoft.com/en-us/store/d/kinect-sensor-for-xbox-one/91hq5578vksc" [Online].
[39] "http://support.xbox.com/en-US/xbox-on-windows/accessories/kinect-for-windows-v2-setup" [Online].
[40] "http://blogs.microsoft.co.il/msdn/2015/01/07/kinect-for-windows-v2/," [Online].
[41] HardKernel, "https://magazine.odroid.com/wp-content/uploads/odroid-xu4-user-manual.pdf" [Online].
[42] "https://www.raspberrypi.org/blog/raspberry-pi-3-on-sale/" [Online].
[43] «http://www.hardkernel.com/main/products/prdt_info.php» [Online].
[44] "http://ep.yimg.com/ay/yhst-137411274456301/gens-ace-lipo-battery-4400mah-60c-6-cells-4.gif" [Online].
[45] "https://www.diamondsystems.com/files/binaries/HE104+DX%20User%20Manual.pdf" [Online].
[46] "http://www.wdlsystems.com/Power-Solutions/HE104DX/HE104-DX-108-watt-3-3V5A-5V12A-12V2-5A-5V400mA-.html" [Online].
[47] "http://www.dpie.com/pc104/psu/tri-m-he104-dx" [Online].
[48] «https://www.arduino.cc/en/Guide/Introduction» [Online].
[49] «http://wiki.ros.org/ROS/Concepts» [Online].
[50] L. Yang, L. Zhang, A. Alelaiwi and A. El Saddik, "ResearchGate", https://www.researchgate.net/publication/277564106_Evaluating_and_Improving_the_Depth_Accuracy_of_Kinect_for_Windows_v2.
[51] "https://www.mathworks.com/discovery/stereo-vision.html" [Online].
[52] "http://docs.opencv.org/2.4/modules/core/doc/intro.html" [Online].
[53] "https://www.slideshare.net/JoaquimSalvi/lecture-2-camera-calibration" [Online].
[54] https://github.com/OpenKinect/libfreenect2/blob/master/README.md
[55] https://www.arduino.cc/en/Main/ArduinoBoardDuemilanove [Online].
[56] https://store.arduino.cc/arduino-due [Online].
[57] «https://link.springer.com/referenceworkentry/10.1007%2F978-0-387-31439-6_167» [Online].
[58] "https://github.com/code-iai/iai_kinect2/issues/143" [Online].

[59] "http://www.robotshop.com/en/lynxmotion-a4wd1-autonomous-rover-kit-botboarduino.html" [Online]

[60] “http://www.robotshop.com/en/12vdc-200rpm-ghm-16-w-rear-shaft.html” [Online].

[61] “https://www.pololu.com/product/2392” [Online].

[62] “http://cdn.usdigital.com/assets/datasheets/E4P_datasheet.pdf ?k=636201882756427345” [Online].

[63] “https://www.pololu.com/product/1269/specs” [Online].

[64] “https://www.todoelectronica.com/es/2050-sensores” [Online].

[65] “http://www.robotshop.com/en/lynxmotion-botboarduino-robot-controller.html” [Online].

[66] “https://link.springer.com/referenceworkentry/10.1007%2F978-0-387-31439-6_167” [Online].

APENDIX A. GAUSS HARDWARE COMPONENTS

Robot Platform

The platform used by *Gauss* is a *Lynxmotion A4WD1* rover kit [56]. It is made of aluminium and has more than enough payload and space to carry all the sensors and electronics, 2.2 Kg. Also, it has four wheels with independent traction, which allows to perform skid steer motion. In addition to the wheels, the rover kit also includes four encoders and motors, one for each wheel.

Here is more detailed information about the platform:

- Measures
  - Length: 12.00" (30.5 cm)
  - Width: 13.50" (34.3 cm)
  - Length Chassis: 9.75" (24,8 cm)
  - Width Chassis: 8.00" (20,3 cm)
  - Height Chassis: 4.00" (20,1 cm)

- Wheels
  - Wheel radius: 4.75" (12 cm)
  - Tires of 4.75”

- Other information
  - Payload: 2.2 Kg
  - Weight: 2 Kg
  - Maximum speed: 0.9 m/s
  -

Figure 43. Lynxmotion A4WD1 chassis and wheels mounted kit [56]

Motors

The four motors are included in the rover kit, and are called GHM-16 [57]. This 12V DC motors provide 200 rpm and a 0.78Kg/cm torque, features suitable for an in-door vehicle that does not have to face difficult terrain.

Figure 44. GHM-16 motor [57]

PID Motor Controller

The PID is a RoboClaw 2x15A motor controller. This device is able to control and decode from two quadrature encoders and transmit via serial bus, and to supply energy to two brushed DC motors and 15A for each of the motors (with 30A peaks) and from 6V to 34V.

Quoting the manufacturer, the controller has a built-in PID routine that can be used for closed-loop speed control and the encoder counts and speeds can be read directly from the RoboClaw for use with an external control system. The main advantage of this PID is that integrates libraries for Arduino development, making it easier to program.

Figure 45. RoboClaw 2x15A motor controller [58]

Encoders

Two E4P-100 quadrature encoders, included in the rover kit, are the included in the design. They require a 5V supply and provide 400 pulses per revolution at 30 KHz.

Figure 46. E4P-100 encoder [59]

Gyroscope

The gyroscope is a L3GD20 three-axes gyro. This device has selectable measurement range and excellent sensitivity; for instance, the most precise mode allows for readings up to ±250 dps (degrees per second) and 0.00875 dps/digit sensibility.

With regards to connection, it uses an I2C bus but each sensor has its own 7-bit address, allowing direct connection to the required sensor. Besides technical characteristics, it is a very small board (25 mm x 12 mm) that can be attached anywhere in the chassis of the robot.

Figure 47. L3GD20 [60]

IR Sensors

The IR sensors are four Sharp GP2D12, which are able to detect and determine the distance to objects between 10cm and 80cm, with a 3cm tolerance.

Figure 48. Sharp GP2D12 IR sensor **[61]**

Microcontroller

The microcontroller is a BotArduino, an upgrade of Arduino Duemilanove for robotics. The main advantage of the platform in front of Duemilanove is its smaller size and the fact that all I/O have the standard servo three pin connection, allowing for direct connection to the servos and encoder directly without a protoboard.

Figure 49. Picture of a BotArduino **[62]**

APENDIX B. ARDUINO CODE

```
// MARC CHESA @ CTTC 2016

//**********************************************************
// Comments:
// Version Arduino+motors++Comms controled with ROS.
// Robot movement is controlled with waypoints. Obstacle
detection and avoidance.
// Speed: MOVE_SPEED=3000 encoder counts per second
//**********************************************************


// ######## INCLUDE ##########
  #include <L3G.h>
  #include <BMSerial.h>
  #include <RoboClaw.h>
  #include <math.h>
  #include <Wire.h>
  #include <TimerOne.h>
  #include <ros.h>
  #include <geometry_msgs/Pose2D.h>
  #include <std_msgs/Int8.h>
  #include <std_msgs/Float32.h>
  #include <std_msgs/Bool.h>

// ######## DEFINE ##########
  // ROBOCLAW PID PARAMETERS
  #define address 0x80

  #define Kp 0x00010000
  #define Ki 0x00008000
  #define Kd 0x00004000
  #define qpps 44000

  // COMS PARAMETERS
  #define SERIAL_SPEED_BPS 115200

  // NAVIGATION PARAMETERS
  #define MAX_WAYPOINTS 50
  #define METERS_TO_ENCODER_COUNTS 27700
  #define TURN_SPEED 2500
  #define MOVE_SPEED 3000
  #define MOVE_ACCEL 3000

  // GYROSCOPE PARAMETERS
  #define GYRO_CAL_MEASUREMENTS 100
  #define HEADING_ERROR_THRESHOLD_DEG 1.1
  #define GYRO_SCALE 1.01
  #define GYRO_READING_INTERVAL_MS 40
```

```
  #define GYRO_READING_INTERVAL_CAL_MS 10

  // DISTANCE SENSOR PARAMETERS
  #define FRONT_OBSTACLE_THRESHOLD_CM 10 //20
  #define DOWN_OBSTACLE_THRESHOLD_CM 20
  #define OBSTACLE_DETECTOR_DELAx_mS 50

  //MOTOR STATE PARAMETERS
  #define STOP 1
  #define FORWARD 2
  #define TURN_RIGHT 3
  #define TURN_LEFT 4
  #define STOPED_BY_USER 5

// ######## ENABLES ##########
  #define Wild_Rover_en //Enables rover control via ROS
commands and data; coodependant with ROS_Comms
  #define ROS_Comms //Enables ROS Comms; coodependant with
Wild_Rover

// ######## GLOBAL VARIABLES ##########

  RoboClaw roboclaw(12,13); // MOTOR CONTROLLER
  L3G gyro; // 3 AXIS GYROSCOPE

  // IMU calibration
  float KALMAN_PROCESS_NOISE_Z=0.05;
  float KALMAN_MEASURES_NOISE_Z;
  float BIAS_Z;
  float VAR_Z;

  // Kalman filter
  float data_gyro_z_pri;
  float P_z_pri;
  float P_z;
  float K_z;
  float A_z=1;
  float data_gyro_kalman_z;
  float data_gyro_z_acc_kalman_deg=0.0;
  boolean initial_loop=true;

  // Position
  float y_m=0;
  float x_m=0;
  float x_pre_m=0;//Initial position of robot is (0,0)
  float y_pre_m=0;
  float heading_deg = 0.0;
  float heading_rad = 0.0;
  float error_angle_rad_prev;//Correction of angle with gyro

  // Time measuring
```

```
  unsigned long time_counter_ms = 0;
  unsigned long elapsed_time_ms = 0;
  float elapsed_time_s = 0.0;
  long tlast = 0;
  long tdelay = 500;
  long tdelayenc = 1000;

  // Sensors global parameters
   bool turning_robot = false;
   int8_t Rover_Command = STOP;
   int8_t Rover_State = STOP;
   int8_t Prev_Rover_State = STOP;
   uint32_t prev_enc1 = 0;
   uint32_t prev_enc2 = 0;
   float dist_real_m = 0;
   float rover_parameter;

  // ROS Comms
   #ifdef ROS_Comms
   //ROS node
   ros::NodeHandle_<ArduinoHardware, 2, 2, 96, 96> nh;

   //messages
   std_msgs::Int8 Rvr_Cmd;
   std_msgs::Bool Ardu_Turn;
   std_msgs::Float32 Param;
   geometry_msgs::Pose2D pose2d_sms;

   //Callbacks
   void Rover_Command_Cb(const std_msgs::Int8& Rvr_Cmd);
   void Parameter_Cb(const std_msgs::Float32& Param);

   //Publisher and subscribers
   ros::Publisher Pose_Data("Pose_Data", &pose2d_sms);
   ros::Publisher Arduino_Turn("Arduino_Turn", &Ardu_Turn);
   ros::Subscriber<std_msgs::Int8>       Command("Command",
Rover_Command_Cb);
   ros::Subscriber<std_msgs::Float32>
Parameter("Parameter", Parameter_Cb);
   #endif


// ######## FUNCTIONS ##########

void setup()
{
  #ifdef ROS_Comms
  nh.loginfo("###### SETUP STARTED######");
  #endif
  // Comms initialization
    Serial.begin(57600);
```

```
    Wire.begin();
    roboclaw.begin(38400);

    // Roboclaw motor controller Setup
    roboclaw.SetM1Constants(address,Kd,Kp,Ki,qpps);
    roboclaw.SetM2Constants(address,Kd,Kp,Ki,qpps);

    #ifdef ROS_Comms
    // ROS comms init
    nh.initNode();
    nh.subscribe(Command);
    nh.subscribe(Parameter);
    nh.advertise(Pose_Data);
    nh.advertise(Arduino_Turn);
    #endif

  // Reset encoders
    roboclaw.ResetEncoders(address);

  // Gyro setup
    // Initialization
    if (!gyro.init())
    {
      delay(1000);
      while (1);
    }
    gyro.enableDefault();

    // Calibration
    calibrate_gyro(BIAS_Z, VAR_Z);
    KALMAN_MEASURES_NOISE_Z=VAR_Z;

    // Stop
    roboclaw.SpeedM1(address,0);
    roboclaw.SpeedM2(address,0);

    #ifdef ROS_Comms
    nh.loginfo("###### SETUP FINISHED######");
    #endif
}

#ifdef ROS_Comms

void pose2D_fill(void)
{
  pose2d_sms.x = x_m;
  pose2d_sms.y = y_m;
  pose2d_sms.theta = heading_rad;
}

void publish_msgs(void)
```

```
{
  //fills a ROS geometry pose2D message with x,y and heading
obtained in this cycle
  pose2D_fill();
  Ardu_Turn.data = turning_robot;
  Pose_Data.publish( &pose2d_sms );
}

void Parameter_Cb(const std_msgs::Float32& Param)
{
  rover_parameter = Param.data;
}

void Rover_Command_Cb(const std_msgs::Int8& Rvr_Cmd)
{
  if   ((Rover_State   !=   TURN_RIGHT)&&(Rover_State   !=
TURN_LEFT))
  {
    Rover_Command = Rvr_Cmd.data;
  }
}
#endif //ROS_Comms


// Estimates the Gyroscope mean and variance (channel Z only)
void calibrate_gyro(float &mean_z, float &var_z)
{
  float M2_z=0.0;
  mean_z=0.0;
  var_z=0.0;
  float delta_z;
 for (int n=1; n<(GYRO_CAL_MEASUREMENTS+1); n++)
 {
     gyro.read();
     delta_z = gyro.g.z - mean_z;
     mean_z = mean_z + delta_z/n;
     M2_z = M2_z + delta_z*(gyro.g.z - mean_z);
     delay(GYRO_READING_INTERVAL_MS);
  }
  var_z = M2_z/(GYRO_CAL_MEASUREMENTS - 1);
}

void get_gyro_heading(void)
{
 //KALMAN FILTER
    // prediction
    if (initial_loop==true)
    {
      // reset heading
      data_gyro_z_acc_kalman_deg=0.0;
      // Gyro calibration
```

```
      calibrate_gyro(BIAS_Z, VAR_Z);
      KALMAN_MEASURES_NOISE_Z=VAR_Z;

      //initial reading
      time_counter_ms = millis();
      //read gyro rate
       gyro.read();
      //read gyro temperature
      //gyro.read_temp(); //todo: for calibration table...
      //delay in initial reading
      delay(20);
      elapsed_time_ms=millis()-time_counter_ms;
      time_counter_ms = millis();
      //compute elapsed time from last reading
      elapsed_time_s=((float)elapsed_time_ms)/1000.0;

      P_z=1;

      //prediction
      P_z_pri=A_z*A_z*P_z+KALMAN_PROCESS_NOISE_Z;
      data_gyro_z_pri=A_z*(gyro.g.z-BIAS_Z);
      //update
      K_z=P_z_pri/(P_z_pri+KALMAN_MEASURES_NOISE_Z);

data_gyro_kalman_z=data_gyro_z_pri+K_z*(GYRO_SCALE*(gyro.g.
z-BIAS_Z)-data_gyro_z_pri);
      P_z=(1-K_z)*P_z_pri;
      initial_loop=false;

    }else{

      //Get loop execution time and reset timer
      elapsed_time_ms=millis()-time_counter_ms;
      time_counter_ms = millis();
      //read gyro rate
      gyro.read();
      //read gyro temperature
      //gyro.read_temp();
      //compute elapsed time from last reading
      elapsed_time_s=((float)elapsed_time_ms)/1000.0;

      //prediction
      data_gyro_z_pri=A_z*data_gyro_kalman_z;
      P_z_pri=A_z*A_z*P_z+KALMAN_PROCESS_NOISE_Z;
      //update
      K_z=P_z_pri/(P_z_pri+KALMAN_MEASURES_NOISE_Z);

data_gyro_kalman_z=data_gyro_z_pri+K_z*(GYRO_SCALE*(gyro.g.
z-BIAS_Z)-data_gyro_z_pri);
      P_z=(1-K_z)*P_z_pri;
    }
```

```
    // compute the heading change from reset [deg]

data_gyro_z_acc_kalman_deg=data_gyro_z_acc_kalman_deg+data_
gyro_kalman_z*elapsed_time_s;
    // wrap degrees to [0,360] Z
}

bool coordinates_and_heading(void)
{
    uint8_t status1, status2;
    bool valid1 = true;
    bool valid2 = true;
    uint32_t enc1=0;
    uint32_t enc2=0;
     //we need to declare this variables to avoid problems
when computing absolute value of enc1-prev_enc1;
    uint32_t enc1_filter=0;
    uint32_t enc2_filter=0;
    uint32_t encoder_counts;
    long prev_dist_real_m;

    if (Rover_State == FORWARD)
    {
     enc1 = roboclaw.ReadEncM1(address, &status1, &valid1);
     enc2 = roboclaw.ReadEncM2(address, &status2, &valid2);
     delay(20);
     enc1_filter = enc1 - prev_enc1;
     enc2_filter = enc2 - prev_enc2;

     //This if works a filter for encorder characteristic
error (enc + (2^8/2, 2^16/2, 2^24/2 & -2^32/2)),
     //2^8 is not filtered because its a posible encoder
value, and it gives only a 5mm error.
     if (enc1_filter > 15000)
      {
        //nh.loginfo("ENCODER1 INVALID");
        valid1=false;
      }
     else if (enc2_filter > 15000)
      {
        //nh.loginfo("ENCODER2 INVALID");
        valid2=false;
      }

     //if one encoder gives wrong readings, we will rely in
the other one; if both send wrong readings, we won't send
any data.
      if ((valid1==true) && (valid2==true))
       {
        encoder_counts =(enc1+enc2);
        dist_real_m = encoder_counts / 2;
```

```
        dist_real_m         =         dist_real_m         /
METERS_TO_ENCODER_COUNTS;
        y_m = dist_real_m*cos(heading_rad)+ y_pre_m;
        x_m = dist_real_m*sin(heading_rad) + x_pre_m;
        prev_enc1 = enc1;
        prev_enc2 = enc2;
       }
       else if (valid2==true)
       {
        encoder_counts = enc2;
        dist_real_m                                       =
encoder_counts/METERS_TO_ENCODER_COUNTS;
        y_m = dist_real_m*cos(heading_rad)+ y_pre_m;
        x_m = dist_real_m*sin(heading_rad) + x_pre_m;
        prev_enc2 = enc2;
       }
       else if (valid1==true)
       {
        encoder_counts = enc1;
        dist_real_m                                       =
encoder_counts/METERS_TO_ENCODER_COUNTS;
        y_m = dist_real_m*cos(heading_rad)+ y_pre_m;
        x_m = dist_real_m*sin(heading_rad) + x_pre_m;
        prev_enc1 = enc1;
       }
       else
       {
        return false; //If false, the data is corrupted;
       }
     }
     return true;
}

boolean move_robot(float dist_m)
{
  uint8_t depth1, depth2;
  uint32_t
target_counts=round(METERS_TO_ENCODER_COUNTS*dist_m);
  uint8_t status;

  bool valid;
  bool stop_valid;
  bool valid_data = true;
  uint32_t enc1_correct;
  uint32_t enc2_correct;
  uint32_t enc1_stop;
  uint32_t enc2_stop;
  uint32_t counter_pos=0;
  uint32_t enc1_temp;
  uint32_t enc2_temp;
  uint32_t move_speed=MOVE_SPEED;
```

```
  uint32_t move_accel=MOVE_ACCEL;
  uint32_t counter_send=0;
  uint8_t counter_send2=0;
  float dist_real_m;
  uint32_t enc11;
  uint32_t enc22;
  uint32_t target_counts_correct;

  //Motor instructions to move
  roboclaw.ResetEncoders(address);

roboclaw.SpeedAccelDistanceM1M2_2(address,MOVE_ACCEL,move_s
peed,target_counts,MOVE_ACCEL,move_speed,target_counts,1);
  roboclaw.ReadBuffers(address,depth1,depth2);

  do{
    roboclaw.ReadBuffers(address,depth1,depth2);
    long tcurrent = millis();
    if( tcurrent - tlast > tdelayenc)
    {
      valid_data = coordinates_and_heading();
      if (valid_data == true)
      {
          publish_msgs();
      }
      #ifdef ROS_Comms
        nh.spinOnce();
      #endif
      tlast = millis();
    }

    //Obstacle detection
      if    ((Rover_State    ==    STOP)||(Rover_State    ==
STOPED_BY_USER))
      {
          //stop
          roboclaw.SpeedM1(address, 0);
          roboclaw.SpeedM2(address, 0);
          uint32_t   enc1   =   roboclaw.ReadEncM1(address,
&status, &valid);
          uint32_t   enc2   =   roboclaw.ReadEncM2(address,
&status, &valid);
          return true;
      }
    }while((depth1!=128) || (depth2!=128));
    return true;
}

boolean turn_robot(float target_heading)
{
  boolean turning_R=false;
```

```
  boolean turning_L=false;
  float heading_error = 0.0;
  float heading_turn = 0.0;
  initial_loop=true;
  boolean end_turn=false;

  String myString = String(target_heading);
  char character[6];
  myString.toCharArray(character,6);

  turning_robot = true;
  heading_turn = heading_deg;

  if (target_heading > 180)
  {
    target_heading = target_heading - 360;
  }
  else if (target_heading < -180)
  {
    target_heading = target_heading + 360;
  }
  #ifdef ROS_Comms
    nh.loginfo("TARGET HEADING");
    nh.loginfo(character);
  #endif

  do
  {
    get_gyro_heading();
    //displayspeed();
    heading_error       =       data_gyro_z_acc_kalman_deg-
target_heading;
    heading_deg = heading_turn + data_gyro_z_acc_kalman_deg;

    if (heading_deg>180)
    {
      heading_deg = heading_deg - 360;
    }
    else if (heading_deg<-180)
    {
      heading_deg = heading_deg + 360;
    }
    heading_rad = (heading_deg*PI)/180;

    //decide turning direction
    if (heading_error>HEADING_ERROR_THRESHOLD_DEG)
    {
      if (turning_L==false)
      {
        //turn left
        roboclaw.SpeedM1(address, TURN_SPEED);
```

```
        roboclaw.SpeedM2(address, -TURN_SPEED);
        turning_L=true;
        turning_R=false;
      }
    }
    else if (heading_error<-HEADING_ERROR_THRESHOLD_DEG)
    {
       if (turning_R==false)
        {
          //turn right
          roboclaw.SpeedM1(address, -TURN_SPEED);
          roboclaw.SpeedM2(address, TURN_SPEED);
          turning_L=false;
          turning_R=true;
        }
    }
    else
    {
        //stop and beep
        roboclaw.SpeedM1(address, 0);
        roboclaw.SpeedM2(address, 0);

        Prev_Rover_State = Rover_State;
        Rover_State = STOP;
       // Rover_Command = STOP;
        y_pre_m = y_m;
        x_pre_m = x_m;
        roboclaw.ResetEncoders(address);
        turning_robot = false;
        end_turn=true;
    }
    delay(GYRO_READING_INTERVAL_MS);
    long tcurrent = millis();
    if(tcurrent - tlast > tdelay)
    {
      publish_msgs();
      tlast = millis();
    }
    #ifdef ROS_Comms
      nh.spinOnce();
    #endif
  }while(end_turn==false);
 #ifdef ROS_Comms
  nh.loginfo("END TURN");
 #endif
 return true;
}

void move_and_turn(void)
{
  float relative_angle;
```

```
  float distance;
  uint8_t depth1,depth2;

  if ((Rover_Command == STOP) && (Rover_State != STOP))
  {
    roboclaw.SpeedM1(address,0);
    roboclaw.SpeedM2(address,0);
    Prev_Rover_State = Rover_State;
    Rover_State = STOP;
    do{
      roboclaw.ReadBuffers(address,depth1,depth2);
    }while(depth1!=0x80  &&  depth2!=0x80);    //loop  until
buffer is empty command completes
  }
  else  if  ((Rover_Command  ==  FORWARD)  &&  (Rover_State  !=
FORWARD))
  {
    roboclaw.SpeedM1(address,3000);
    roboclaw.SpeedM2(address,3000);
    Prev_Rover_State = Rover_State;
    Rover_State = FORWARD;
  }
  else  if  ((Rover_Command  ==  TURN_RIGHT)&&  (Rover_State  !=
TURN_RIGHT))
  {
     relative_angle = rover_parameter - heading_deg;
     if (relative_angle != 0)
     {
          Prev_Rover_State = Rover_State;
          Rover_State = TURN_RIGHT;
          turn_robot(relative_angle);
     }
  }
  else  if  ((Rover_Command  ==  TURN_LEFT)&&  (Rover_State  !=
TURN_LEFT))
     relative_angle = rover_parameter - heading_deg;
     if (relative_angle != 0)
     {
          Prev_Rover_State = Rover_State;
          Rover_State = TURN_LEFT;
          turn_robot(relative_angle);
     }
  }
  else if (Rover_Command == STOPED_BY_USER)
  {
    roboclaw.SpeedM1(address, 0);
    roboclaw.SpeedM2(address, 0);
    Prev_Rover_State = Rover_State;
    Rover_State = STOPED_BY_USER;

    roboclaw.ResetEncoders(address);
```

```
    do{
      roboclaw.ReadBuffers(address,depth1,depth2);
    }while(depth1!=0x80 && depth2!=0x80);
   }
}

########## MAIN LOOP ##########

void loop()
{
 long tcurrent = millis();
 bool valid_data = true;

  if( tcurrent - tlast > tdelay)
  {
    valid_data = coordinates_and_heading();
    if (valid_data == true)
    {
      publish_msgs();
    }
    tlast = millis();
  }
  #ifdef ROS_Comms
    nh.spinOnce();
  #endif
  move_and_turn();
}
```

APENDIX C. ODROID CODE

Main

```
/*main.cpp
 *
 *Created on: Jan 2, 2017
 *Author: Marc Chesa
 */

/**CTRL+C**/
#include <signal.h>

/**FILE READER**/
#include <sstream>
#include <iostream>
#include <fstream>

/**VECTOR AND MATH VARIABLES**/
#include <vector>
#include <string>
#include "math.h"
#include <chrono>

/**ROS**/
#include "ros/ros.h"
#include <ros/spinner.h>
#include "geometry_msgs/Pose2D.h"
#include "std_msgs/Bool.h"
#include "std_msgs/Int8.h"
#include "std_msgs/Float32.h"
#include "sensor_msgs/PointCloud2.h"

/**OUR CLASES & FUNCTIONS**/
#include "pathfinder.h"
#include "Point2D.h"
#include "file_reader.h"
#include "rover_command.h"

using namespace std;

/**Parameters**//
#define Ymax -0.10
#define Ymin 0.19
#define Zmax 5.5
#define Zmin 0.5
#define Xlever_arm -0.03
#define Zlever_arm -0.10
#define grid_size 0.01
#define Xdetection_max 0.13
#define Xdetection_min -0.18
#define centre_turn_criteria -0.025

/**STATES**/
```

```
#define STOP 1
#define FORWARD 2
#define RIGHT 3
#define LEFT 4
#define STOPED_BY_USER 5

//control variables
int num_obstacles = 0;
int num_kinect_it = 0;

//variables
int rover_state = STOP;
int prev_rover_state;
int IR_Right;
int IR_Left;
int turn_criteria = 3;
int current_turn_criteria;
int IR_turn_criteria = 3;
int robot_speed = 0;

float Xrover = 0;
float Yrover = 0;
float current_heading_rad = 0;
float current_heading_deg = 0;
float prev_heading_deg = -1;
float rover_parameter = 0;
float obstacle_mean_distance;
float obstacle_min_distance;
float Zdetection_max;

//bool turn_published = false;
bool obstacle_detected = false;
bool IR_obstacle_detected = false;
bool OD_enabled = true;
bool OA_enabled = false; //OA = Obstacle Avoidance
bool Decide_turn_criteria = true;
bool Kinect_Callback = false;
bool waiting_turn = false;
bool turning_robot = false;

Point2D wp;
vector<Point2D> waypoints;

chrono::time_point<chrono::system_clock> start_t, end_t,
publish_t, turn_t;

ros::Publisher Command;
ros::Publisher Parameter;
std_msgs::Int8 Motor_Command;
std_msgs::Float32 Rover_Parameter;

//***FUNCTIONS & CALLBACKS***\\

/**CTRL+C**/
```

```
void publish_command (int cmd, float param)
{

      Rover_Parameter.data = param;
      Parameter.publish(Rover_Parameter);

      Motor_Command.data = cmd;
      Command.publish(Motor_Command);

      printf("published: %d %f\n", cmd, param);

}

void my_handler(int s)
{
      rover_state = STOPED_BY_USER;
      rover_parameter = 0;
      publish_command_5(rover_state,rover_parameter);
      printf("Stopped by user\n");
      exit(1);
}


bool compare_Point2D_fields(Point2D a, Point2D b)
{
      return (a.x) < (b.x);
}
/*
void Turning_Callback(const std_msgs::Bool Ardu_Turn)
{
      turning_robot = Ardu_Turn.data;
}
*/
void Pose_Data_Callback(const geometry_msgs::Pose2D Pose_Data)
{
      Xrover = Pose_Data.x;
      Yrover = Pose_Data.y;
      current_heading_rad = Pose_Data.theta;
      current_heading_deg = ((current_heading_rad*180)/M_PI);
      //odometry_data = true;
      //printf("Pose Data\n");
}

void Point_Cloud_Callback(const sensor_msgs::PointCloud2 pCloud)
{
      //printf("KINECT CALLBACK\n");
      int height = pCloud.height;
      int width = pCloud.width;
      float X;
      float Y;
      float Z;

      Point2D kinect_point;
      Point2D kinect_prev_point;

      vector<Point2D> kinect_matrix;
```

```
vector<Point2D> kinect_vector_local;
vector<Point2D> potential_obstacles_vector;
vector<Point2D> kinect_vector_global;

if (OD_enabled)
{
      Kinect_Callback == true
      for (int v = 0 ; v < height; v++)
      {
            for (int u = 0 ; u < width; u++)
            {
                  int arrayPosition = v*pCloud.row_step +
                  u*pCloud.point_step;

                  int arrayPosX = arrayPosition +
                  pCloud.fields[0].offset;

                  int arrayPosY = arrayPosition +
                  pCloud.fields[1].offset;

                  int arrayPosZ = arrayPosition +
                  pCloud.fields[2].offset;

                  memcpy(&X, &pCloud.data[arrayPosX],
                  sizeof(float));
                  memcpy(&Y, &pCloud.data[arrayPosY],
                  sizeof(float));
                  memcpy(&Z, &pCloud.data[arrayPosZ],
                  sizeof(float));

                  if ((isnan(X) == false)&&(Z < Zmax)&&(Z >
                  Zmin))
                  {
                        if ((Y < Ymin)&&(Y > Ymax))
                        {
                              //*** Vector filler
                              kinect_point.x = roundf(X *
                              100) / 100;

                              kinect_point.y = roundf(Z *
                              100) / 100;

                              kinect_matrix.push_back
                              (kinect_point);
                        }
                  }
            }
      }
      // kinect_matrix vector values are sorted from lower
      to higher X values
      sort(kinect_matrix.begin(), kinect_matrix.end(),
            compare_Point2D_fields);

      // A new vector is created giving a 2D aproximation
      of the enviroment in front the rover
      kinect_vector_local.push_back(kinect_matrix.at(0));
```

```
for (int it = 0 ; it < kinect_matrix.size() ; it++)
{
     kinect_point = kinect_matrix.at(it);

     if (kinect_point.x == kinect_prev_point.x)
     {
          if (kinect_point.y < kinect_prev_point.y)
          {
               kinect_vector_local.pop_back();
       kinect_vector_local.push_back(kinect_point);
               kinect_prev_point = kinect_point;
          }
     }
     else
     {

       kinect_vector_local.push_back(kinect_point);
       kinect_prev_point = kinect_point;
     }
}
// Creation of the "security area" for Obstacle
Detection
float Dist_X = wp.x-Xrover;
float Dist_Y = wp.y-Yrover;

Point2D most_relevant_obstacle;
float turn_left = 0;
float turn_rigth = 0;
int mean_counter = 0;
obstacle_mean_distance = 0;

if (!OA_enabled)
     Zdetection_max = sqrt(pow(Dist_X, 2) +
     pow(Dist_Y, 2));

bool local_obstacle_detected = false;

most_relevant_obstacle = kinect_point;

for (int it = 0; it < kinect_vector_local.size() ;
it++)
{
     kinect_point = kinect_vector_local.at(it);

     if((kinect_point.x <=
     Xdetection_max)&&(kinect_point.x >=
     Xdetection_min)&&(kinect_point.y <=
     Zdetection_max))
     {
          local_obstacle_detected = true;

          if (kinect_point.x <=
          centre_turn_criteria
          {
               turn_rigth = turn_rigth +
               (1/kinect_point.y);
```

```
                }
                else if (kinect_point.x >
                centre_turn_criteria)
                {
                      turn_left = turn_left +
                      (1/kinect_point.y);
                }

                if (kinect_point.y
                < most_relevant_obstacle.y)
                {
                      most_relevant_obstacle =
                      kinect_point;
                }

                obstacle_mean_distance =
                obstacle_mean_distance + kinect_point.y;

                mean_counter++;
          }
    }
    obstacle_min_distance = most_relevant_obstacle.y;
    obstacle_min_distance );
    obstacle_detected = local_obstacle_detected;

    if ((obstacle_detected)&&(!OA_enabled))
    {
          obstacle_mean_distance =
          (obstacle_mean_distance/mean_counter);
          if (most_relevant_obstacle.y <= 0.75)
          {
                if (most_relevant_obstacle.x <=
                centre_turn_criteria)
                {
                      turn_criteria = RIGHT;
                }
                else if (most_relevant_obstacle.x >
                centre_turn_criteria)
                {
                      turn_criteria = LEFT;
                }
          }
          else
          {
                if (turn_rigth > turn_left)
                {
                      turn_criteria = RIGHT;
                }
                else
                {
                      turn_criteria = LEFT;
                }
          }
    }
}
Kinect_Callback == false;
```

```
}

int main(int argc, char **argv)
{
	printf ("START\n");

	// Variables Setup
	int rover_state = STOP;
	float initial_angle = 0;
	Point2D new_OA_waypoint;
	rover_command cmd;
	float prev_heading_deg = -1;

	// CTRL+C
	struct sigaction sigIntHandler;
	sigIntHandler.sa_handler = my_handler;
	sigemptyset(&sigIntHandler.sa_mask);
	sigIntHandler.sa_flags = 0;
	sigaction(SIGINT, &sigIntHandler, NULL);

	// ROS Setup
	ros::init(argc, argv,
	"rover_navigation_and_obstacle_detection",
	ros::init_options::NoSigintHandler);
	ros::NodeHandle nh;
	ros::Subscriber sub = nh.subscribe("Pose_Data", 1,
	Pose_Data_Callback);
	ros::Subscriber sub2 = nh.subscribe("/kinect2/sd/points",
	1, Point_Cloud_Callback);
	Command = nh.advertise<std_msgs::Int8>("Command", 1);
	Parameter = nh.advertise<std_msgs::Float32>("Parameter",
	1);

	//Function to read the waypoints dataset
	waypoints = file_reader();
	publish_t = chrono::system_clock::now();
	turn_t = publish_t;

	// LOOP

	while(waypoints.size() > 0)
	{
		if (OA_enabled)
		{
			//printf("OA Iteration\n");
			if (waiting_turn)
			{
				chrono::duration<double> diff_time =
				chrono::system_clock::now()-turn_t;

				if (fabs(current_heading_deg -
				rover_parameter) < 3.0)
					rover_state = STOP;
					rover_parameter = 0;
					usleep(3500000);
					waiting_turn = false;
```

```
                        }
                        prev_heading_deg = current_heading_deg;
                        ros::spinOnce();
                }
                else if (!obstacle_detected) //Obstacle
                avoidance module introduces the new waypoint
                {
                        rover_state = STOP;
                        rover_parameter = 0;

        publish_command(rover_state,rover_parameter);

        //rotation
        float Xrot = (*Zdetection_max)*sin(current_heading_rad);
        float Yrot = (2*Zdetection_max)*cos(current_heading_rad);

        //translation
        float Xglobal = Xrover + Xrot + Xlever_arm;
        float Yglobal = Yrover + Yrot + Zlever_arm;

        new_OA_waypoint.x = Xglobal;
        new_OA_waypoint.y = Yglobal;
        waypoints.push_back(new_OA_waypoint);

        printf("NEw OA WAYPOINT %f %f\n", Xglobal, Yglobal);
        OA_enabled = false;
                }
                else
                {
                if(Decide_turn_criteria)
                {
                        if (turn_criteria == RIGHT)
                        {
                                rover_state = RIGHT;

                                rover_parameter =
                                current_heading_deg + 10;

                                }
                                else if (turn_criteria == LEFT)
                                {
                                        rover_state = LEFT;
                                        rover_parameter =
                                        current_heading_deg - 10;
                                }
                                current_turn_criteria =
                                turn_criteria;
                                publish_command(rover_state,
                                rover_parameter);
                                printf("first turn criteria: %d\n
                                ", current_turn_criteria);
                                Decide_turn_criteria = false;
                                waiting_turn = true;
                        }
                        else
                        {
```

```
                    if (current_turn_criteria == RIGHT)
                    {
                         rover_state = RIGHT;
                         rover_parameter =
                         current_heading_deg + 10;

                    }
                    else if (current_turn_criteria ==
                    LEFT)
                    {
                         rover_state = LEFT;
                         rover_parameter =
                         current_heading_deg - 10;
                    }
                    publish_command(rover_state,
                    rover_parameter);

                    printf("turn criteria:
                    %d\n ", current_turn_criteria);
                    waiting_turn = true;
               }

          }
     }
     else if (obstacle_detected == true)
     {
          rover_state = STOP;
          rover_parameter = 0.0;
          publish_command(rover_state,rover_parameter);
          printf("Potential Obstacle detected\n");
          usleep (2000000);
          ros::spinOnce();

          if (obstacle_detected == true)
          {
               OA_enabled = true;
               Zdetection_max = obstacle_min_distance;
               Decide_turn_criteria = true;
          }
     }

     wp = waypoints.back();
     if (!OA_enabled)
     {
          prev_rover_state = rover_state;
          cmd = pathfinder (Xrover, Yrover, wp.x, wp.y,
          current_heading_rad);
          rover_state = cmd.command;
          rover_parameter = cmd.parameter;
     if ((rover_state == RIGHT)||(rover_state == LEFT))
          {
               OD_enabled = false;
          }
          else
          {
               OD_enabled = true;
```

```
                }

                // If the robot is in STOP state, but has not
                detected an obstacle, the waypoint has been
                reached and it is eliminated from the list of
                waypoints
                if (rover_state == STOP)
                {
                     speed1 = 0;
                     speed2 = 0;
                     rover_state = STOP;
                     rover_parameter = 0;
                     publish_command(rover_state,
                     rover_parameter);

                     waypoints.pop_back();

printf ("%f %f %f %f %f %f %f %d\n", time_count, Xrover, Yrover,
current_heading_deg, speed1, speed2, rover_parameter,
rover_state);

                }
                //pathfinder returns -1 if the code has failed.
This will stop the robot and the code.
                else if (rover_state == -1)
                {
                     rover_state = STOP;
                     rover_parameter = 0;
                     publish_command(rover_state,
                     rover_parameter);

                     return 0;
                }
           }
           chrono::duration<double> publisher_time =
           chrono::system_clock::now()-publish_t;

           if ((!waiting_turn)&&(publisher_time.count() >=
           0.5)&&(rover_state != STOP))
           {
                publish_command(rover_state, rover_parameter);
                //Print pose on screen
                time_count = time_count + 0.5;
printf ("%f %f %f %f %f %f %f %d\n", time_count, Xrover, Yrover,
current_heading_deg, speed1, speed2, rover_parameter,
rover_state);
                publish_t = chrono::system_clock::now();
           }
           ros::spinOnce();
      }
      printf("Final destination reached!\n");
      printf("#########################\n");
      return 0;
}
```

Path planner

```
/*  pathfinder.cpp
 *
 *  Created on: Jan 2, 2017
 *      Author: Marc Chesa
 */


#include "math.h"
#include <stdio.h>
#include "pathfinder.h"

#define waypoint_radius 0.10 //10 cm
#define max_heading_error 2 //The parameter sets the maximum error on
the right and the left as 2°, a total of 4°

using namespace std;

// Pathfinder function decides what the rover must do according to
current position and heading and destination.
// To do so, returns an rover command structure, containing a command
(in form of integer number) and a heading.
// Command can have the following values (stop (1), forward (2), right
(3), left(4), error (-1)).
rover_command pathfinder(float Xrover, float Yrover, float Xtarget,
float Ytarget, float current_heading_rad)
{
    rover_command cmd;
    float absolute_heading_rad;
    float absolute_heading_deg;
    float target_heading_deg;
    float current_heading_deg = ((current_heading_rad*180)/M_PI);
    float Dist_X = Xtarget - Xrover;
    float Dist_Y = Ytarget - Yrover;
    float Real_Distance = sqrt(pow(Dist_X, 2) + pow(Dist_Y, 2));

    // If the Real_Distance is different to 0, the relative angle
between the current
    // and destination waypoints is computed taking into account the
current heading.
    if ((Real_Distance > waypoint_radius)||(Real_Distance < -
waypoint_radius))
    {
        //Angle is obtained by computing the arcsin.
        absolute_heading_rad = asin(Dist_X/Real_Distance);
        absolute_heading_deg = ((absolute_heading_rad*180)/M_PI);

        if (Dist_Y < 0)
        {
            if (Dist_X > 0)
            {
                absolute_heading_deg = 180 - absolute_heading_deg;
                }
            else if (Dist_X <= 0)
            {
                absolute_heading_deg = fabs(absolute_heading_deg) -
180;
            }
        }
```

```
        //Finally, we compute the difference between the absolute and
the current angle;
        target_heading_deg =  absolute_heading_deg -
current_heading_deg;

    }
    // If the Real_Distance is 0, the destination has been reached, so
the rover must stop.
    else
    {
        cmd.command = 1;
        cmd.parameter = 0;
        return cmd; //Stop
    }


    //2. The angle between waypoints, target_heading_deg, is used to
determine the motion.

    //   An angle of 0 means that the heading is already correct, and
a forward order is given.
    //   An angle of 180 or -180 means the same, the destination is
straight behind. In this case,
    //   the rule is that always turn to the right.
    if ((target_heading_deg <= max_heading_error)&&(target_heading_deg
>= -max_heading_error))
    {
        cmd.command = 2; //Move forward
        //cmd.parameter = Real_Distance;
    }
    else if (0 < target_heading_deg)
    {
        cmd.command = 3; //Turn right, also for a 180 degrees
        //cmd.parameter = absolute_heading_deg;
    }
    else if (0 > target_heading_deg)
    {
        cmd.command = 4; //Turn left
        //cmd.parameter = absolute_heading_deg;
    }
    else
    {
        cmd.command = -1;
        cmd.parameter = 0;
        printf ("error %d\n", cmd.command); //Error
        return cmd;
    }
    cmd.parameter = absolute_heading_deg;
    return cmd;
}
```

File reader

```
/*  file_reader.cpp
 *
 *  Created on: Jan 2, 2017
 *      Author: Marc Chesa
 */

#include <sstream>
#include <iostream>
#include <fstream>

#include <vector>
#include <string>
#include <iterator>
#include "Point2D.h"

using namespace std;

vector<Point2D> file_reader (void)
{
    Point2D wp;
    vector<Point2D> waypoints;
    vector<Point2D>::iterator it;
    ifstream file
("/home/odroid/ros_workspace/src/rover_navigation_and_obstacle_detecti
on/Rover_Path.txt");
    string line;

    if (file.is_open())
    {
        it = waypoints.begin();
        while (getline(file,line))
        {
            stringstream ssin(line);

            //cout << line << endl;
            //cout << ssin << endl;
            ssin >> wp.x; //x
            ssin >> wp.y; //y
            //cout << wp.x << endl;
            //cout << wp.y << endl;
            waypoints.insert(waypoints.begin(), wp);
        }
        file.close();
    }
    else
    {
        cout << "Unable to open the file";
    }

    for (it = waypoints.begin(); it != waypoints.end() ; it++)
    {
        int index = distance( waypoints.begin(), it );
        wp = waypoints.at(index);
        //cout << wp.x << endl;
        //cout << wp.y << endl;
    }
    return waypoints;
}
```

Classes

Point2D

```
/*
 * Point2D.h
 *
 *  Created on: Jan 2, 2017
 *  Author: Marc Chesa Roldan
 */

#ifndef POINT2D_H_
#define POINT2D_H_

class Point2D
{
  public:
    double x, y;

  private:

};

class Pose2D
{
  public:
    double x, y, theta;

  private:

};

#endif /* POINT2D_H_ */
```

Rover command

```
/*
 * rover_command.h
 *
 *  Created on: May 31, 2017
 *  Author: Marc Chesa Roldan
 */

#ifndef ROVER_COMMAND_H_
#define ROVER_COMMAND_H_

class rover_command
{
  public:
    int command;
    float parameter;

  private:

};

#endif /* ROVER_COMMAND_H_ */
```