

České učení technické v Praze
Fakulta strojní
Ústav přístrojové a řídicí techniky



Algoritmy mapování v neznámém prostředí pro mobilního robota TurtleBot3

Diplomová práce

Kirill Rassudikhin

Magisterský program: Automatizační a přístrojová technika

Magisterský obor: Automatizace a průmyslová informatika

Vedoucí práce: Ing. Jaroslav Bušek, Ph.D.

Praha, srpen 2022

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Rassudikhin** Jméno: **Kirill** Osobní číslo: **466560**
Fakulta/ústav: **Fakulta strojní**
Zadávající katedra/ústav: **Ústav přístrojové a řídicí techniky**
Studijní program: **Automatizační a přístrojová technika**
Specializace: **Automatizace a průmyslová informatika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Algoritmy mapování neznámém prostředí pro mobilního robota TurtleBot3

Název diplomové práce anglicky:

Autonomous navigation in an unknown environment for the mobile robot TurtleBot3

Pokyny pro vypracování:

1. Seznamte se s robotem TurtleBot3 a jeho funkcionalitou.
2. Proveďte rešerši na téma algoritmy mapování prostředí s přihlédnutím k možnostem platformy TurtleBot3.
3. Zvolte alespoň tři vhodné algoritmy mapování neznámého prostředí pro mobilního robota TurtleBot3 a popište je.
4. Implementujte zvolené algoritmy v simulačním prostředí s mobilním robotem TurtleBot3.
5. Nasadte zvolené algoritmy v reálném prostředí na fyzickém mobilním robotovi TurtleBot3.
6. Zhodnoťte dosažené výsledky v simulaci a při reálném měření.

Seznam doporučené literatury:

- [1] Z. Sun, B. Wu, C. -Z. Xu, S. E. Sarma, J. Yang and H. Kong, "Frontier Detection and Reachability Analysis for Efficient 2D Graph-SLAM Based Active Exploration," 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2020, pp. 2051-2058, doi: 10.1109/IROS45743.2020.9341735.
[2] LAVALLE, Steven Michael. Planning algorithms. New York: Cambridge University Press, 2006. ISBN 978-0521862059.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Jaroslav Bušek, Ph.D. U12110.3

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **29.04.2022**

Termín odevzdání diplomové práce: **12.08.2022**

Platnost zadání diplomové práce: _____

Ing. Jaroslav Bušek, Ph.D.
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

doc. Ing. Miroslav Španiel, CSc.
podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

Datum převzetí zadání

Podpis studenta

Vedoucí práce:

Ing. Jaroslav Bušek, Ph.D.
Ústav přístrojové a řídicí techniky
Fakulta strojní
České vysoké učení technické v Praze
Technická 2
160 00 Praha 6
Česká republika

Prohlášení

Prohlašuji, že jsem tuto magisterskou práci vypracoval samostatně, s tím, že její výsledky mohou být dále použity podle uvážení vedoucího diplomové práce jako jejího spoluautora. Souhlasím také s případnou publikací výsledků diplomové práce nebo její podstatné části, pokud bude uveden jako její spoluautor.

V Praze 2022

.....
Kirill Rassudikhin

Abstract

Abstract

The main goal of this master thesis is to research, implement, test and compare at least three algorithms for autonomous navigation in unknown environment for Turtlebot3 robot. The algorithms should ensure autonomous navigation. Theoretical part of this thesis describes chosen algorithms and helping toolboxes, their mathematical essence and common problems obstructing their correct work. Practical part of this thesis describes the developed software for implementation of chosen algorithms and the testing of chosen algorithms. The chosen algorithms are: naive frontier detection method, Yamauchi's frontier detection method and fast frontier detection method. Those algorithms search the map and provide the goals on unknown or partially known map for helping toolboxes to move the robot to desired positions. While moving, the robot gradually discovers new unknown areas on the map. Algorithms were developed in python language. Toolboxes that move the robot to the desired location are Actionlib, slam_toolbox, tf2. Algorithms were tested in the simulation environment on two maps and in real room with obstacles. The main tool for making these algorithms and toolboxes work together to autonomously map unknown environment is ROS.

Keywords: Autonomous navigation, Autonomous mapping, ROS, SLAM, Actionlib, slam_toolbox, Yamauchi's Frontier Detection, Fast Frontier Detection

Abstrakt Hlavním cílem této diplomové práce je vyzkoumat, implementovat, otestovat a porovnat alespoň 3 algoritmy mapování v neznámém prostředí pro robota Turtlebot3. Algoritmy musí zajistit autonomní mapování. Teoretická část této práce je věnovaná popisu třech zvolených algoritmů a popisu pomocných toolboxů. Praktická část této práce popisuje vyvinutý program, implementující zvolené metody mapování a testování zvolených algoritmů mapování. Tyto algoritmy jsou: naive frontier detection method, Yamauchi's frontier detection method, fast frontier detection method. Tyto algoritmy hledají na neznámé nebo částečně známé mapě cíle pro pomocné toolboxy, které řídí robota, aby se dostal do cíle. Zatímco jede, robot postupně objevuje a mapuje nové neznámé části mapy. Algoritmy byly vyvinuté v programovacím jazyce Python. Pomocné toolboxy jsou: Actionlib, slam_toolbox, tf2. Algoritmy byly otestované v simulačním prostředí ve dvou mapách a také v reálné testovací místnosti s překážkami. Hlavním nástrojem, který zajišťuje spolupráci algoritmů a toolboxů, je ROS.

Klíčová slova: Autonomní navigace, autonomní mapování, ROS, SLAM, Actionlib, slam_toolbox, Yamauchi's Frontier Detection, Fast Frontier Detection.

Poděkování

Děkuji moji rodině a kamarádům za podporu během psání diplomové práce. Taky děkuji mému vedoucímu Ing. Jaroslavu Buškovi, Ph.D za rady a pomoc.

List of Tables

| | | |
|------|---|----|
| 2.1 | Turtlebot Hardware Units, [2] | 3 |
| 9.1 | Results of naive frontier detection method on the small map, part 1 | 74 |
| 9.2 | Results of naive frontier detection method on the small map, part 2 | 74 |
| 9.3 | Results of YFD method on the small map, part 1 | 76 |
| 9.4 | Results of YFD method on the small map, part 2 | 76 |
| 9.5 | Results of FFD method on the small map, part 1 | 77 |
| 9.6 | Results of FFD method on the small map, part 2 | 77 |
| 9.7 | Overall results method on the small map, part 1 | 79 |
| 9.8 | Overall results on the small map, part 2 | 79 |
| 9.9 | Results of naive method on the big map, part 1 | 83 |
| 9.10 | Results of naive method on the big map, part 2 | 83 |
| 9.11 | Results of YFD method on the small map, part 1 | 85 |
| 9.12 | Results of YFD method on the small map, part 2 | 85 |
| 9.13 | Results of FFD method on the small map, part 1 | 88 |
| 9.14 | Results of FFD method on the small map, part 2 | 88 |
| 9.15 | Overall results method on the small map, part 1 | 88 |
| 9.16 | Overall results on the small map, part 2 | 89 |
| 10.1 | Overall results of naive method in the real room, part 1 | 92 |
| 10.2 | Overall results of naive method in the real room, part 2 | 93 |
| 10.3 | Overall results of YFD method in the real room, part 1 | 95 |
| 10.4 | Overall results of YFD method in the real room, part 2 | 96 |
| 10.5 | Overall results of FFD method in the real room, part 1 | 98 |
| 10.6 | Overall results of FFD method in the real room, part 2 | 98 |

List of Figures

| | | |
|-----|--|----|
| 2.1 | Image of turtlebot3, [5] | 5 |
| 3.1 | Scheme ROS connection, [7] | 6 |
| 3.2 | actionlib client-server communication, [9] | 8 |
| 3.3 | slam map example, [10] | 8 |
| 3.4 | Relation between map frame and odometry frame caused by accumulated odometry errors on servos, [13] | 9 |
| 4.1 | Illustration of GraphSLAM algorithm, [14] | 13 |
| 4.2 | SLAM Processing flow, [17] | 14 |
| 4.3 | Understanding SLAM Using Pose Graph, [18] | 15 |
| 4.4 | Particle Filter example | 16 |
| 5.1 | Occupancy Grid map illustration | 17 |
| 5.2 | Time complexity of naive approach, [25] | 18 |
| 5.3 | Nomad 200 robot, [20] | 19 |
| 5.4 | Abstraction of an exploration mission in terms of frontier generation operations and map updates, [23] | 22 |
| 5.5 | Frontier progression : (a) Deleting an entire frontier-contour and creating a new one (b)Extending a frontier-contour from one corner (c)Extending a frontier-contour from the center. Dark Lines indicate current frontier-contours and dashed lines indicate deleted frontier-contours, [23] | 22 |
| 5.6 | Example of RRT, used for path planning of autonomous vehicle | 23 |
| 6.1 | Uniform distribution, used for naive finding frontiers | 25 |
| 6.2 | Example of YFD frontier detection, (a) evidence grid, (b) frontier edge segments, (c) frontier regions, [20] | 26 |
| 6.3 | Obtaining laser sensors readings | 27 |
| 6.4 | Connected readings from laser sensors | 27 |
| 6.5 | Obtaining laser sensors readings | 28 |
| 8.1 | Conceptual class diagram of the code | 30 |
| 8.2 | TurtleBotSlamExplorer class diagram | 31 |
| 8.3 | State machine within TurtleBotSlamExplorer.explore() function | 34 |
| 8.4 | List of typical frames, present in exploration project, developed with ROS | 37 |
| 8.5 | Typical frames, present in exploration project, developed with ROS | 37 |

| | | |
|------|--|----|
| 8.6 | Procedure of obtaining the row and column number of 2D array, where the desired cell is situated. | |
| | a) Map frame coordinates | |
| | b) Origin frame coordinates | |
| | c) Desired row and column in map 2D array | 38 |
| 8.7 | diagram of DFDdetectorClass, implementing Yamauchi's Frontier Detection | 43 |
| 8.8 | Example of applying Sobel's kernel on the image for edge detection, [30] . . | 48 |
| 8.9 | Breadth-First search, [32] | 49 |
| 8.10 | Example of clustering with my implementation of recursive BFS, clusters are found, written in the cluster objects and deleted from gradient magnitude map: | |
| | a) Obtained map of estimate of the magnitude of the gradient, not clustered yet | |
| | b) Magnitude map after several iterations of recursive BFS | |
| | c) Magnitude map after more iterations of recursive BFS | |
| | d) Magnitude map after more iterations of recursive BFS | 51 |
| 8.11 | 4 and 8 connectivity kernels, used in CCL, [35] | 51 |
| 8.12 | a) binary image b) CCL processed image, [36] | 52 |
| 8.13 | UML class diagram of Cluster class | 54 |
| 8.14 | UML class diagram of FFDdetectorClass class | 55 |
| 8.15 | The process of remapping scan readings: | |
| | a) Original scan message | |
| | b) Remapped scan message | 61 |
| 8.16 | Result of working of the DDA line algorithm and the connect laser readings with lines algorithm: | |
| | a) Before connecting the laser readings | |
| | b) After connecting the laser readings | 64 |
| 8.17 | Example of working of the method i.e. classifying points of lines: | |
| | a) Red cells are frontier points | |
| | b) Violet cells contain obstacles in their neighbourhood | |
| | c) Gray do not have unknown cells around them, they are not objects of FFD method | 66 |
| 8.18 | Check neighbourhood method, forming neighbourhood, depending on the position of the checked cell | 67 |
| 8.19 | drift of the coordinates of the frontier's centroid with map growth | 67 |
| 8.20 | Map to origin transformation method description | 69 |
| 8.21 | UML class diagram for FFDfrontier class | 69 |
| 9.1 | Small map for testing the turtlebot3 burger in simulation environment . . | 71 |
| 9.2 | Big map for testing the turtlebot3 burger in simulation environment | 72 |
| 9.3 | Number of discovered cells with iterations and time graphs | 73 |
| 9.4 | Velocity of discovering cells with iterations and time graphs | 73 |
| 9.5 | Total path of the robot, shown on the map | 74 |
| 9.6 | Number of discovered cells with iterations and time graphs | 75 |
| 9.7 | Velocity of discovering cells with iterations and time graphs | 75 |
| 9.8 | Total path of the robot, shown on the map | 76 |
| 9.9 | Number of discovered cells with iterations and time graphs | 78 |
| 9.10 | Velocity of discovering cells with iterations and time graphs | 78 |

| | | |
|-------|---|-----|
| 9.11 | Total path of the robot, shown on the map | 79 |
| 9.12 | Evaluated and compared paths of all of the three methods | 80 |
| 9.13 | Number of discovered cells with iterations and time graphs | 81 |
| 9.14 | Velocity of discovering cells with iterations and time graphs | 82 |
| 9.15 | Total path of the robot, shown on the map | 82 |
| 9.16 | Number of discovered cells with iterations and time graphs | 84 |
| 9.17 | Velocity of discovering cells with iterations and time graphs | 84 |
| 9.18 | Total path of the robot, shown on the map | 85 |
| 9.19 | Number of discovered cells with iterations and time graphs | 87 |
| 9.20 | Velocity of discovering cells with iterations and time graphs | 87 |
| 9.21 | Total path of the robot, shown on the map | 88 |
| 9.22 | Evaluated and compared paths of all of three methods on the big map . . . | 90 |
| 10.1 | Photo of testing room №1 | 91 |
| 10.2 | Photo of testing room №2 | 91 |
| 10.3 | Photo of testing room №3, turtlebot in comparison with obstacles | 92 |
| 10.4 | Best result for naive frontier detection, path of the robot | 94 |
| 10.5 | Best result for naive frontier detection, number of discovered cells with time and iterations graph | 94 |
| 10.6 | Best result for naive frontier detection, velocity of discovering cells with time and iterations graph | 95 |
| 10.7 | Best result for YFD frontier detection, path of the robot | 96 |
| 10.8 | Best result for YFD frontier detection, number of discovered cells with time and iterations graph | 97 |
| 10.9 | Best result for YFD frontier detection, velocity of discovering cells with time and iterations graph | 97 |
| 10.10 | Best result for FFD frontier detection, path of the robot | 99 |
| 10.11 | Best result for FFD frontier detection, number of discovered cells with time and iterations graph | 99 |
| 10.12 | Best result for FFD frontier detection, velocity of discovering cells with time and iterations graph | 100 |

Acronyms

- API** Application Programming Interface. 6
- BFS** Breadth-First Search. ix, 20, 21, 49–51, 53
- CCL** Connected-component labeling. ix, 49, 51–53
- CPU** Central Processing Unit. 16
- DFD** Direct Frontier Detection (the same as YFD). xiii, 25, 44, 45, 59, 86
- EKF** Extended Kalman Filter. 10, 29
- FFD** Fast Frontier Detection. vii, x, xiii, 18, 20, 21, 26, 30, 31, 54, 56–59, 67–70, 77, 79, 80, 85, 86, 88, 89, 98–101, 103
- IMU** Inertial Measurement Unit. 3
- LDS** Laser distance sensor. 3, 4, 8, 14–16, 20, 33, 55, 59, 86, 89, 98, 101, 103
- MCB** Main Controlling Board. 3
- OBB** Oriented Bounded Box algorithm. xi, 18, 21, 22
- OFD** OBB Based Frontier Detection. 18, 21, 22
- ROS** Robot Operating System. 5–7, 29, 35, 58, 103
- RPi** Raspberry Pi. 3, 4, 16
- RRT** Rapidly exploring Random Trees. 23
- SBC** Single Board Computer. 3
- SLAM** Simultaneous localization and mapping. xii, 1, 8, 10–16, 24, 29, 102, 103
- UML** Unified Modeling Language. ix, 30, 69, 70
- WFD** Wavefront Frontier Detection. 18, 20, 21, 53
- WFD-INC** Incremental Wavefront Frontier Detection. 18, 21
- WFD-IP** Incremental-Parallel Frontier Detection. 18, 21
- YFD** Yamauchi’s Frontier Detection. vii, viii, x, xi, 18–20, 25, 26, 30–33, 53, 76, 77, 79, 80, 83, 85, 86, 88, 89, 95–98, 100, 101, 103

Contents

| | |
|---|-------------|
| Prohlášení | iv |
| Abstract | v |
| Poděkování | vi |
| List of Tables | vii |
| List of Figures | viii |
| Acronyms | xi |
| 1 Introduction | 1 |
| 2 Description of the robot | 3 |
| 3 Robot Operating System | 6 |
| 3.1 Actionlib toolbox | 7 |
| 3.2 slam_toolbox toolbox | 8 |
| 3.3 tf2 toolbox | 8 |
| 4 SLAM and slam_toolbox | 10 |
| 4.1 Basic types of SLAM algorithm | 10 |
| 4.2 Extended Kalman Filter for Simultaneous Localization and Mapping . . . | 10 |
| 4.2.1 Mathematics of Extended Kalman Filter Algorithms | 11 |
| 4.3 Graph Based Algorithms for Simultaneous Localization and Mapping . . . | 12 |
| 4.3.1 Mathematics of Graph Based Algorithms | 13 |
| 4.4 Common SLAM problems | 14 |
| 5 Frontier finding algorithms in general | 17 |
| 5.1 Yamauchi's Frontier Detection | 18 |
| 5.2 Fast Frontier Detection | 20 |
| 5.3 Wavefront Frontier Detection and its Incremental and Parallel modifications | 20 |
| 5.4 Oriented Bounded Box algorithm based Frontier Detection | 21 |
| 5.5 Rapidly-exploring Random Tree Frontier Detection | 23 |
| 6 Chosen frontier finding algorithms | 24 |
| 6.1 Naive Frontier Detection Algorithm | 24 |
| 6.2 Yamauchi's Frontier Detection Algorithm | 25 |
| 6.3 Fast Frontier Detection Algorithm | 26 |

| | | |
|-----------|---|------------|
| 7 | Conclusion of the theoretical part | 29 |
| 8 | Developed software and its structure | 30 |
| 8.1 | Turtlebot class | 31 |
| 8.1.1 | Explore method | 33 |
| 8.1.2 | Get robot's pose method | 36 |
| 8.1.3 | Get robot's yaw rotation method | 39 |
| 8.1.4 | Add new transformed frame method | 40 |
| 8.1.5 | Set zeroed map method | 40 |
| 8.1.6 | Naive frontier detection method | 42 |
| 8.2 | Yamauchi's Frontier Detector class | 43 |
| 8.2.1 | Frontier detectionDFDmethod | 44 |
| 8.2.2 | Map gradient method | 46 |
| 8.2.3 | Clustering method | 48 |
| 8.2.4 | Cluster class | 53 |
| 8.3 | Fast Frontier Detector class | 54 |
| 8.3.1 | Do FFD method | 56 |
| 8.3.2 | Map laser readings method | 59 |
| 8.3.3 | Get lines from laser readings method | 62 |
| 8.3.4 | Find frontiers in lines method | 64 |
| 8.3.5 | Transform map frame to origin frame and back method | 67 |
| 8.3.6 | FFD frontier class | 69 |
| 9 | Testing with simulation software: | 71 |
| 9.1 | Testing on small map | 72 |
| 9.1.1 | Exploration with naive Frontier Detection method: | 72 |
| 9.1.2 | Exploration with Yamauchi's frontier detection method | 74 |
| 9.1.3 | Exploration with fast frontier detection method | 77 |
| 9.1.4 | Overall comparing of the performance on the small map | 79 |
| 9.2 | Testing on the big map | 80 |
| 9.2.1 | Exploration with naive frontier detection method | 80 |
| 9.2.2 | Exploration with Yamauchi's frontier detection method | 83 |
| 9.2.3 | Exploration with fast frontier detection method | 85 |
| 9.2.4 | Overall comparing of the performance on the big map | 88 |
| 10 | Testing on turtlebot3 burger: | 91 |
| 10.1 | Exploration with naive frontier detection method | 92 |
| 10.2 | Exploration with Yamauchi's frontier detection | 95 |
| 10.3 | Exploration with fast frontier detection | 98 |
| 10.4 | Overall comparison of methods in real life | 100 |
| 11 | Conclusion | 102 |
| | Bibliografie | 106 |

Chapter 1

Introduction

Autonomous navigation and mapping are very trending topics today. More and more domains are getting involved with autonomous navigation: self-driving cars, self-driving robots in warehouses, search and rescue operations and working in dangerous for humans environment. All of this requires a technology for autonomous navigation and mapping.

The goal of this thesis is to research, implement, test and compare at least three algorithms for autonomous navigation. The first thing appearing if "autonomous navigation robot" query is entered into a search engine is SLAM or Simultaneous Localisation And Mapping. This method makes a map of an unknown environment, while the robot is moving to the known borders of the map. Half of the videos showing SLAM performance show people controlling robots, with the robot moving to where the operator requested it to move and robot gradually composing the map of a room, an office or a test space.

But what if someone was to make this process autonomous? They would need to replace the operator's actions that cause the robot to go to the borders of the known map. Algorithms capable of doing this already exist. These algorithms are called Frontier Finding algorithms.

Frontier finding algorithms are algorithms, which process a map and compute a point on a known map, where the robot with working SLAM should go to discover more cells. The more sophisticated frontier finding algorithm is, the faster, better and more efficient the autonomous navigation and mapping will be.

The theoretical part of this thesis will begin by describing the turtlebot, its hardware, and tools such as Robot Operating System, which is a main tool in terms of this thesis. It will describe helping toolboxes like `slam_toolbox` and others, which will help to implement autonomous navigation and mapping algorithms. Moreover, it will describe the SLAM algorithm, its mathematics and common problems for this algorithm. And in the end, it will describe frontier finding algorithms in general and the three chosen frontier finding algorithms.

The practical part of this thesis is dedicated to the description of the developed software, that implements the chosen frontier finding methods. Lastly, there will be the evaluation of the results of experiments, which test implemented methods in simulation environment and in real life.

Chapter 2

Description of the robot

Turtlebot3 is small programmable personal robot with open-source software [1]. It can be used for variety of applications. Its sensors allow it to be used for mapping, autonomous navigation, localization and many more. It consists of chassis, which has four layers. First layer has LiPo battery, DYNAMIXEL servo motors with wheels and its connectors on it, second layer has OpenCR controller and its connectors on it, third layer has RPi and board usb2lds, which has interfaces for using LDS. And forth layer has LDS.

| Hardware unit | Full name |
|---------------------|---------------------------------------|
| SBC | Raspberry Pi 3B+ |
| Servo motors | DYNAMIXEL XL430-W250 |
| MCB | OpenCR 1.0 board |
| LDS | 360 Laser Distance Sensor LDS-01 |
| IMU | Gyroscope 3 Axis Accelerometer 3 Axis |

Table 2.1: Turtlebot Hardware Units, [2]

Now moving on to the more specific description of hardware units.

The Raspberry Pi 3, with a quad-core ARM Cortex-A53 processor, is described as having ten times the performance of a Raspberry Pi 1. Benchmarks showed the Raspberry Pi 3 to be approximately 80 faster [3] than the Raspberry Pi 2 in parallel tasks. On the Model B and B+, the Ethernet port is provided by a built-in USB Ethernet adapter. Although often pre-configured to operate as a headless computer, the Raspberry Pi may also optionally be operated with any generic USB computer keyboard and mouse. It may also be used with USB storage, USB to MIDI converters, and virtually any other device/component with USB capabilities, depending on the installed device drivers. None of the Raspberry Pi models have a built-in real-time clock. When booting, the time is set either manually or configured from a previously saved state at shutdown to provide relative consistency for the file system. The Network Time Protocol is used to update the system time when connected to a network [3].

DYNAMIXEL XL430-W250 is a robot exclusive smart actuator with fully integrated DC Motor + Controller + Driver + Sensor + Reduction Gear + Network in one DC servo module. The DYNAMIXEL XL series adopts new features that allow 360 degrees control mode with its contactless magnetic encoder and hollow back case assembly structure. The XL series has the same mechanical structure as the XM430 and XH430 and is compatible with the respective models [4]. Because of this smart actuator robot knows the exact distance it went and the angle, it rotated to. In other words robot's odometry is based on DYNAMIXEL. This is possible due to 12 bit encoder [4], which is ensuring 0.087° precision on the actuators.

OpenCR1.0 is main controlling board of Turtlebot3, which provides connection from RPi to the rest of hardware (LDS, Servos ...). OpenCR; Open-source Control module for ROS, is developed for ROS embedded systems to provide completely open-source hardware and software. Everything about the board; Schematics, PCB Gerber, BOM and the firmware source code for the TurtleBot3 are free to distribute under open-source licenses for users and the ROS community. The STM32F7 series is a main chip inside the OpenCR board which is based on a very powerful ARM Cortex-M7 with floating point unit. The development environment for OpenCR is wide open from Arduino IDE and Scratch for young students to traditional firmware development for the expert. OpenCR provides digital and analog input/output pins that can interface with extension board or various sensors. Also, OpenCR features various communication interfaces: USB for connecting to PC, UART, SPI, I2C, CAN for other embedded devices [2].

360 Laser Distance Sensor LDS-01 is a 2D laser scanner capable of sensing 360 degrees that collects a set of data around the robot to use for SLAM (Simultaneous Localization and Mapping) and Navigation. The LDS-01 is used for TurtleBot3 Burger, Waffle and Waffle Pi models. It supports USB interface(USB2LDS) and is easy to install on a PC. It supports UART interface for embedded board. It has distance range: $120 \sim 3500$ [mm], rotation rate 300 ± 10 [rpm], sampling rate: 1.8 [kHz], distance precision: ± 10 [mm] for range $120 \sim 499$ [mm] and ± 3.5 [%] for range $500 \sim 3500$ [mm] [2]. It provides laser scans in 2D. The data LDS returns are distances linked to the angle, where angle is rotation angel in the moment, when distance was measured. It could make measurement 1800 times per second, but because of lower rotation rate LDS makes more measurement for the same angle. LDS's rotation rate is controlled by electric motor and sampling rate is controlled by software.

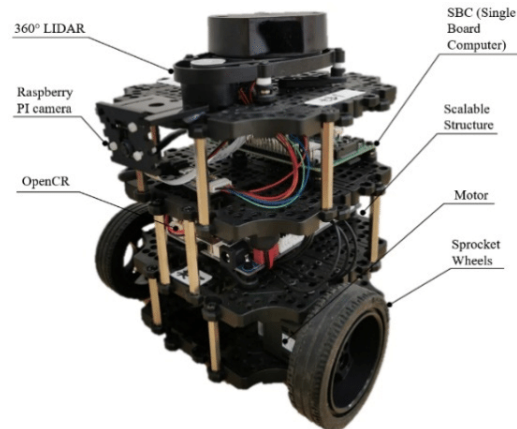


Figure 2.1: Image of turtlebot3, [5]

Now about software side of this robot. Turtlebot runs under Robot Operating System or ROS. It is service, which connects turtlebot with master server (in my case, it was my computer). Master server runs program. ROS runs on both master server and turtlebot. It runs under Ubuntu 20.04. But more on that in the next section.

Chapter 3

Robot Operating System

ROS is Robot Operating System. It is a multiple software libraries and software tools for realising robotic applications. ROS is middleware, it resembles API, because technically it is middle layer between firmware of individual actuators and your robotic application, because actuators could have completely different interfaces for interaction with it. The goal of ROS is to provide a standard for writing a robotic application. Main advantage of the ROS - its software. It runs and communicates, allowing you to design complex software without knowing how certain hardware works [6]. Also ROS despite its name is not operating system, it is extension above the OS. Usually it runs on top of Ubuntu Linux.

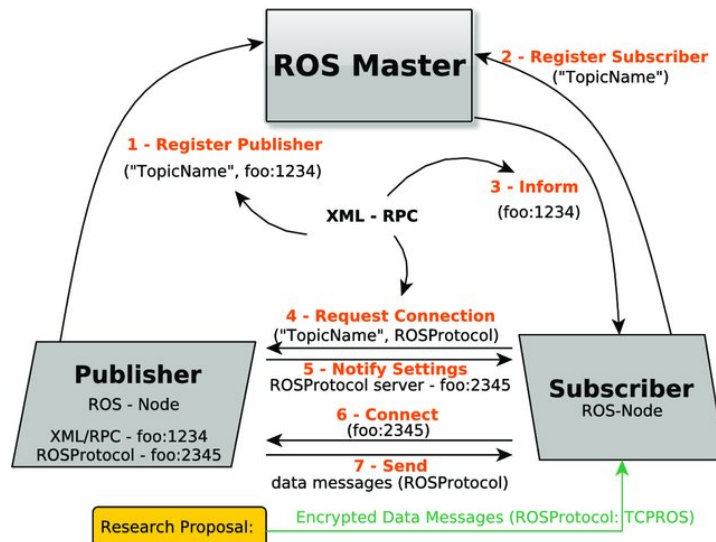


Figure 3.1: Scheme ROS connection, [7]

Now moving on to the structure of ROS and its communication. Typical ROS workspace structure is projects separated into packages. In each package there are nodes (small program), which handles its tasks. Those programs communicate through topics. Those will be used mainly for sending data streams between nodes. It is extremely useful, for exam-

ple if it is necessary to monitor the temperature of a motor on the robot, the solution is to create a node, which will send a data stream with the temperature. Now, any other node can subscribe to this topic and get the data [8]. Node, which is working with topics can be either a Publisher or Subscriber. Publisher publishes data and subscriber listens to it. You can see the scheme of connection on fig 3.1. Next part of ROS is Services and Actions. Services comes handy, when it is necessary to create a simple synchronous client/server communication between nodes. Very useful for changing a setting on your robot, or ask for a specific action: enable freedrive mode, ask for specific data, etc [8]. Actions (Action servers) are in fact based on topics. They are something like publishers with asynchronous client/server architecture. The client can send a request that takes a long time (ex: asking to move the robot to a new location). The client can asynchronously monitor the state of the server, and cancel the request anytime [8]. The great advantage of ROS is that it has logging system to keep track of all communication between all nodes and even to keep track of actions on action servers.

ROS nodes can be written in C++ or Python. You can use both languages for any application it does not matter which language you use. It can be done because of the communication layer is below the “language level”. As it is working as a middleware, ROS uses standard TCP/IP sockets to communicate between nodes [8].

The other significant advantage of ROS is, that it is widespread. Because of it major of actuators, cameras, sensors has package for ROS. And some robots are directly developed for using with ROS, such as turtlebot.

3.1 Actionlib toolbox

Actionlib is very important tool in terms of this thesis, implemented as ROS package. It serves purpose of moving robot to the stated goal, sent by user. More specifically Actionlib is toolbox, that provides an interface to setup a server-client communication for moving robot to the target location [9]. The client application runs on user’s machine and from client’s application server gets target location. Next server application are communicating with other more low-level packages to make robot move to the target. The diagram of client-server application is shown on fig 3.2. The target location is always being tracked. The goal is being tracked and executed, while the terminal state of the goal is being reached. The terminal states can be: goal reached, goal canceled, goal lost.

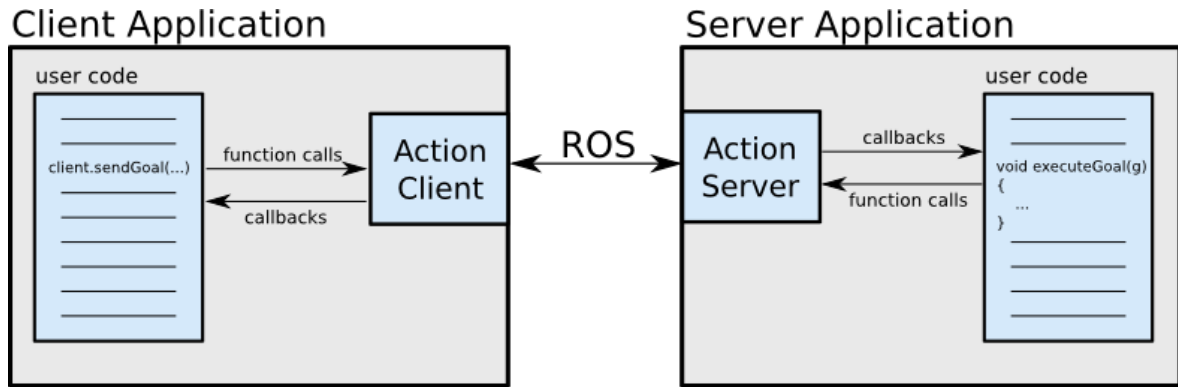


Figure 3.2: actionlib client-server communication, [9]

3.2 slam_toolbox toolbox

Slam_toolbox is second but most important toolbox used in this thesis. It implements SLAM algorithm, which will be described in the next chapter. Slam_toolbox processes information from LDS of the robot, transforms those readings from robot coordinate frame to the map coordinate frame and creates a map, where unknown, known and obstacle cells are all mapped. The example of map generated by slam_toolbox is on the fig 3.3

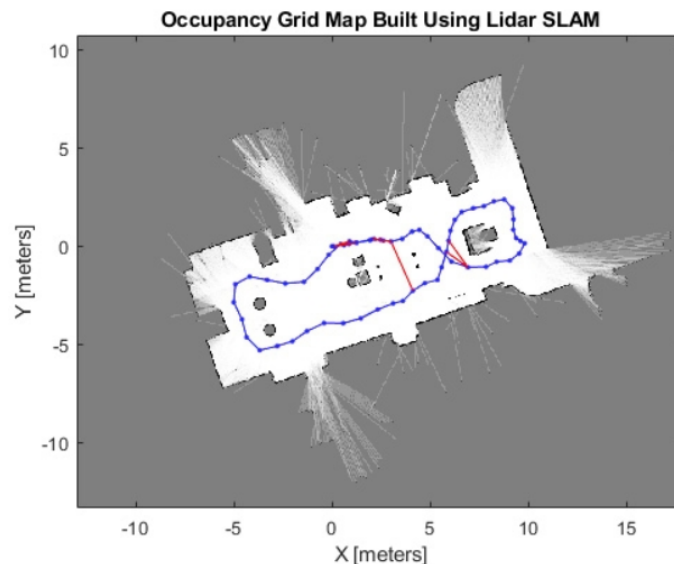


Figure 3.3: slam map example, [10]

3.3 tf2 toolbox

Tf2 is another important toolbox in the terms of my thesis. This toolbox is designed and used to keep track of multiple coordinate systems. Also it provides the interface to get the transformation between any two frames in any moments of time in form of

two vectors, which represents displacement and angular displacement between two frames [11]. Because of robotic system has a lot of coordinate systems within itself and a lot of coordinate systems on the map (map coordinate system, robot's main coordinate system, origin of the map coordinate system, odometry coordinate system and etc.), to access point's coordinates in one frame, knowing point's coordinate in other frame, user can use tf2 toolbox. "When doing tasks with a robot it is crucial that the robot be aware of where it is itself as well as where the rest of the world is in relation to itself. A simple example which demonstrates this well is a mobile robot finding a red ball and touching with it's gripper. The challenge is simply to move the gripper toward the ball. However, to do this simple task the relationship between the ball and the gripper must be known" [12]. Mainly usages of this toolbox in my code are limited to obtaining relation between map coordinate frame and odometry frame. This relation shows the drifting of map frame from its initial position and helps compensate odometric errors.

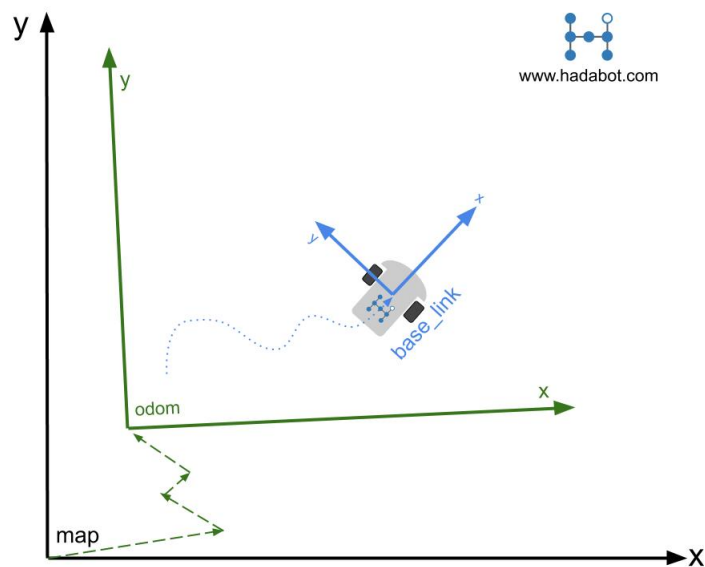


Figure 3.4: Relation between map frame and odometry frame caused by accumulated odometry errors on servos, [13]

Chapter 4

SLAM and slam_toolbox

Simultaneous Localization and Mapping or for short SLAM is method for autonomous vehicles and robots to navigate in the unknown environment. SLAM allows autonomous vehicle to get data from sensors and simultaneously navigate knowing only part of the map gradually discovering map and adding discovered pieces to known ones. The common problems, which SLAM is trying to solve is: accumulating of localization errors, caused by noisy measurement in odometry and laser distance sensors; failing of the localization and losing the position of the robot on the map; High computational cost for image processing, point cloud processing and optimization.

4.1 Basic types of SLAM algorithm

SLAM problem solutions are divided on EKF-SLAM algorithms and Graph-Based algorithms. The „slam_toolbox“ algorithm is belongs to the Graph-Based algorithms.

4.2 Extended Kalman Filter for Simultaneous Localization and Mapping

The slam_toolbox, that was used in terms of this thesis, uses graph SLAM algorithm, although I think that it is important to describe both existing SLAM algorithms to understand how SLAM works in both, because those algorithms have common traits. EKF-SLAM or Extended Kalman Filter for Simultaneous Localization and Mapping is the first approach made for solution of the SLAM problem. Any EKF algorithm makes assumption, that the noise has Gaussian Distribution and the model is linear. This algorithm depends on landmarks on the map. Algorithm remembers, where the landmarks were in time “k-1”. After that the algorithm makes an assumption, where the robot will be at time “k”, based on current state of the robot and control input. After input at time “k”

was made, the algorithm evaluates where the robot is, based on location of the features on the map. It's also possible, because location of the features on the map are included in the state vector of the robot [14]. Unfortunately EKF-SLAM solves only localization part of the SLAM problem.

4.2.1 Mathematics of Extended Kalman Filter Algorithms

This section will begin with definition of the model of robot. Considering a mobile robot as plant. Robot is moving through unknown environment taking relative observations of using sensor, located on robot. At the time k the following variables are defined [15]:

- x_k - the state vector, containing position and orientation of the robot
- u_k - control vector, applied at the time k
- m_i - location of i -th landmark. Those locations do not change in time
- z_{ik} - an observation of the i -th landmark, taken from the vehicle at time k
- $X_{0:k} = [x_0, x_1, \dots, x_k] = [X_{0:k-1}; x_k]$ - vector of previous vehicle locations
- $U_{0:k} = [u_0, u_1, \dots, u_k] = [U_{0:k-1}; u_k]$ - vector of previous vehicle control inputs
- $Z_{0:k} = [z_0, z_1, \dots, z_k] = [Z_{0:k-1}; z_k]$ - vector of previous landmark observations

That brings it to the determination of the probabilistic SLAM problem:

$$P(x_k, m | Z_{0:k}, U_{0:k}, x_0) = \frac{P(z_k | x_k, m) P(x_k, m | Z_{0:k-1}, U_{0:k}, x_0)}{P(z_k | Z_{0:k-1}, U_{0:k})}$$

Localization and mapping algorithm requires the probability $P(x_k, m | Z_{0:k}, U_{0:k}, x_0)$ to be computed for every k -time. The resulting formula is derived using Bayes rule. The description of the formula follows like this: on the left side of the formula there is joint probability of state vector (position + orientation) and vector of the location of the landmarks given the recorded observations and control inputs up to k time together with the initial state of the vehicle. In numerator there are two probabilities multiplied. The first member of numerator is probability of observation vector at k -time given joint distribution of state vector at k -time and location of the landmarks. The second member of numerator is joint probability of state vector at k -time and position of landmarks given the recorded observations and control inputs up to $k - 1$ time together with the initial state of the vehicle. In the denominator there is probability of observations of landmark given the given the recorded observations and control inputs up to $k - 1$ time. In short that

would mean that this formula computes probability of robot's and landmarks coordinates on all of the map given all previous observations, control commands and initial coordinates of the robot.

Localization and mapping algorithm requires the probability $P(x_k, m | Z_{0:k}, U_{0:k}, x_0)$ to be computed for every k -time. The resulting formula is derived using Bayes rule.

This recursive algorithm uses vectors of previous landmark observations, control inputs to calculate joint probability distribution of state vector (robot's position + orientation) and locations of the landmarks.

Also this formula requires observation model and motion model, which looks like this:

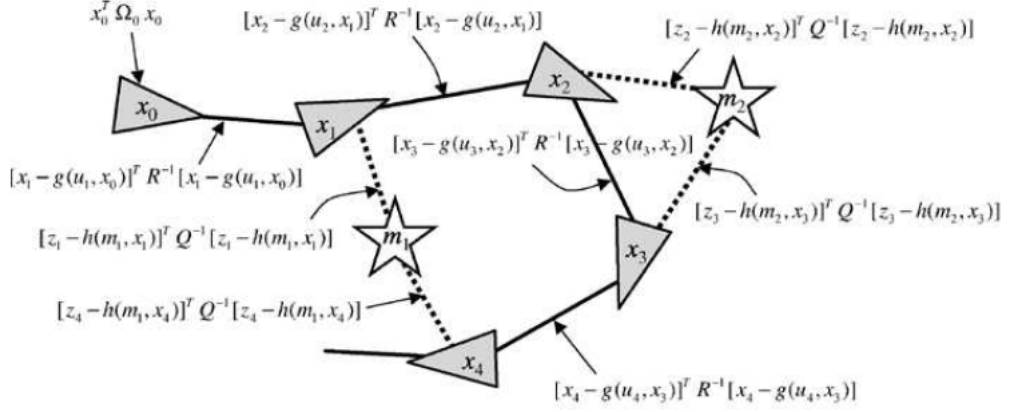
- $P(z_k | x_k, m)$ - observation model
- $P(x_k | x_{k-1}, u_k)$ - motion model

The observation model describes the probability of making an observation z_k when the vehicle location and landmark locations are known. And motion model can be imagined as state space model only in terms of probability.

This formulation of the problem has been proved, that correlations between landmark estimates increase monotonically as more and more observations are made. That means that the more measurement robot does, the more concrete positions of landmarks become. This convergence occurs, because robot's measurement are considered nearly independent [15].

4.3 Graph Based Algorithms for Simultaneous Localization and Mapping

In my implementation of SLAM + Frontier detecting algorithm there was used Graph Based Algorithm in „slam_toolbox“, so this type of algorithms are more important in terms of this thesis. GraphSLAM solves localization and mapping problem. It works that way. As robot moves it remembers all its poses and inputs, that led to them (x, u) , where x – state vector and u – inputs in the system. Also if at any of remembered poses any features of the map were observed, they are remembered and saved as well. This can be represented in a graph, where will be two types of edges: motion and measurement. Motion represents evaluated state of system at different poses, measurement represents evaluated pose of features on the map. After some steps algorithm minimises error between evaluated values and real one and determines most probabilistic path of the robot and poses of map features [14].



Sum of all constraints:

$$J_{\text{GraphSLAM}} = x_0^T \Omega_0 x_0 + \sum [x_i - g(u_i, x_{i-1})]^T R^{-1} [x_i - g(u_i, x_{i-1})] + \sum [z_i - h(m_i, x_j)]^T Q^{-1} [z_i - h(m_i, x_j)]$$

Figure 4.1: Illustration of GraphSLAM algorithm, [14]

4.3.1 Mathematics of Graph Based Algorithms

The mathematics of Graph based slam algorithms is based on Linear-Quadratic Optimization of function J . Dynamic of the system is represented by linear relation $A \cdot x = b$, where x is the state vector, representing robot's poses [16]. And J is the cost function, which is quadratic. Typically J function looks like this:

- J - the cost function
- x_t - state vector at time t
- Q - state cost matrix
- u_t - inputs at time t
- R - input cost matrix
- N - cross weight matrix

$$J = \int_{t_0}^{t_1} (x_t^T Q_t x_t + u_t^T R_t u_t + 2x_t^T N_t u_t) dt$$

With linear equations of any system $\dot{x} = A \cdot x + B \cdot u$ it is possible to calculate the most efficient coefficients K for state controller $u = Kx \rightarrow \dot{x} = A \cdot x + B \cdot Kx \rightarrow \dot{x} = x \cdot (A + BK)$. And this is how with its help we can compute optimal coefficients for state controller for linear time invariant system. State vector does not change its size with the time at that

case, optimal coefficients (result of optimisation) do not change either. With graph slam everything is a little bit trickier. Robot remembers its poses in u vector and relations between its poses and poses of local map features (unusual objects on the map) as state vector x . And it has linear relation between probability matrix A , state vector with all map features and relation of map features to robot poses x and right sides of equations b [16]. With those equations it is possible to compute optimised relations between robot's poses and map features with respect to minimal error. And it is like computing optimal coefficient for state controller, only not once, but every time when new map feature is discovered and added to state vector.

4.4 Common SLAM problems

To better explain SLAM the following example is given: "Consider a home robot vacuum. Without SLAM, it will just move randomly within a room and may not be able to clean the entire floor surface. In addition, this approach uses excessive power, so the battery will run out more quickly. On the other hand, robots with SLAM can use information such as the number of wheel revolutions and data from cameras and other imaging sensors to determine the amount of movement needed. This is called localization. The robot can also simultaneously use the camera and other sensors to create a map of the obstacles in its surroundings and avoid cleaning the same area twice. This is called mapping" [17].

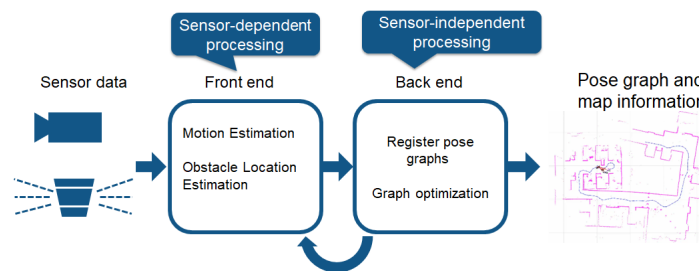


Figure 4.2: SLAM Processing flow, [17]

The common problems, which SLAM is trying to solve is:

1) Accumulating of localization errors, caused by noisy measurement in odometry and laser distance sensors:

Accumulation of such types of errors usually causing major deviation from actual values (position of the map, position of obstacles on the map etc.)

What SLAM does is decreasing the error in the measurement of distance that robot went or distance between the poses of robot, during which the measurement by LDS was made and the position of measurements of obstacle at every pose of robot. This error accumulates over time and making the map of measured obstacles not usable. Also the

position on the maps with accumulated error differs very much from real pose of the robot. The example of map with errors is on the fig. 4.3.

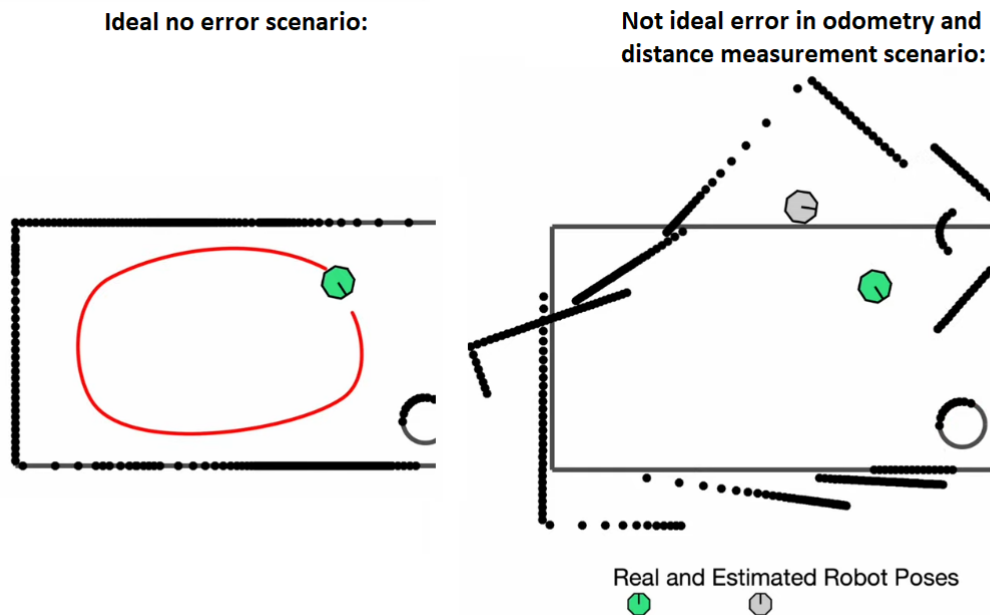


Figure 4.3: Understanding SLAM Using Pose Graph, [18]

The problem on the fig 4.3. is called "loop closure problem" because in the end as robot should drive to its initial position, on the map - it is a completely wrong position. Pose estimation errors like these are unavoidable. It is important to detect loop closures and determine how to correct or cancel out the accumulated error. The SLAM solution to this problem is to remember places on the map, where LDS measurement was conducted. After that the search for unique landscape features is conducted in every separate measurement. If unique features are found in separate measurements, but their positions on the global map do not line up with each other, then the poses of robot (where the similar features was measured) are shifted, so the features would line up. After this correction of the map, it corresponds more to the actual terrain. The separate positions and corresponding measurements are remembered through pose graphs, where edge is distance between the poses and node is position of the robot with corresponding measurement from LDS. By solving error minimization as an optimization problem, more accurate map data can be generated.

The second common problem is:

2) Localization fails and the position on the map is lost:

In earliest localization methods like triangular localization of robot, this type of problem occurred, causing random discontinuities in robot moves on the map. Robot could be moving with 2 [m/s] speed and then suddenly teleport forward. When such thing occurs

it is necessary to initiate recovery behaviour and fuse multiple sensors and robot motion model to make calculations based on the sensor data. This is done through Kalman Filters or Particle Filters.

In `slam_toolbox` usually localization on the map is conducted through particle filter. The particle filter is recursive Bayesian filter by Monte Carlo sampling [19]. For our application it means that given known map, SLAM algorithm will derive all current possible poses of robot based on current measurement from LDS. On the fig 4.4 the green is current robot pose and red dots are possible poses of robot on the given map of room, which is black.

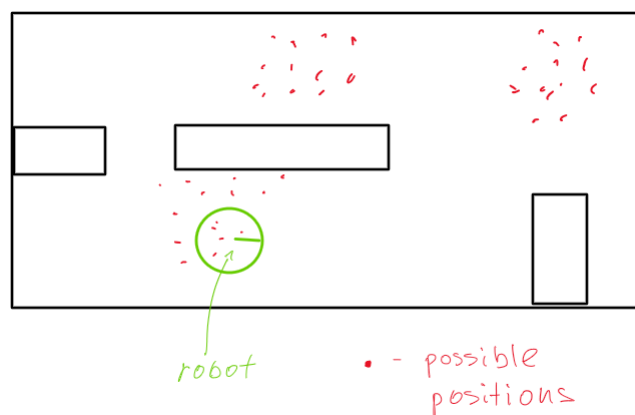


Figure 4.4: Particle Filter example

When localization still fails, a countermeasure to recover is by remembering a landmark as a key-frame from a previously visited place. When searching for a landmark, a feature extraction process is applied in a way that it can scan at high speeds [17].

3) High computational cost for image processing, point cloud processing, and optimization:

The complexity of all algorithms required for SLAM method is a problem, when it should run on compact low-energy embedded microprocessors (like in RPi). To get SLAM working accurately, it is necessary to conduct image processing, point cloud processing and correcting the map (loop closure problem) at high frequencies. One solution is to run all those processes in parallel. With multicore CPUs it is quite possible [17].

Chapter 5

Frontier finding algorithms in general

Slam_toolbox and Actionlib toolboxes, described in chapter three cannot move robot and make it map the environment on their own. This is where Frontier Finding algorithms come in handy. They process map and compute the most efficient place for robot to go to discover the most new cells and expand map.

Frontiers is very important concept in the autonomous navigation first proposed by Brian Yamauchi in 1997. According to his definition frontiers are: "regions on the boundary between open space and unexplored space." [20] And by moving constantly to the frontiers the robot gets the biggest amount of information about new unknown environment. The frontier approach to the exploration is to detect new frontiers over and over and reach them until there is no frontier left. So detecting new frontiers is critical part of autonomous exploration and speeding up the search of frontiers will speed up the whole algorithm of autonomous navigation.

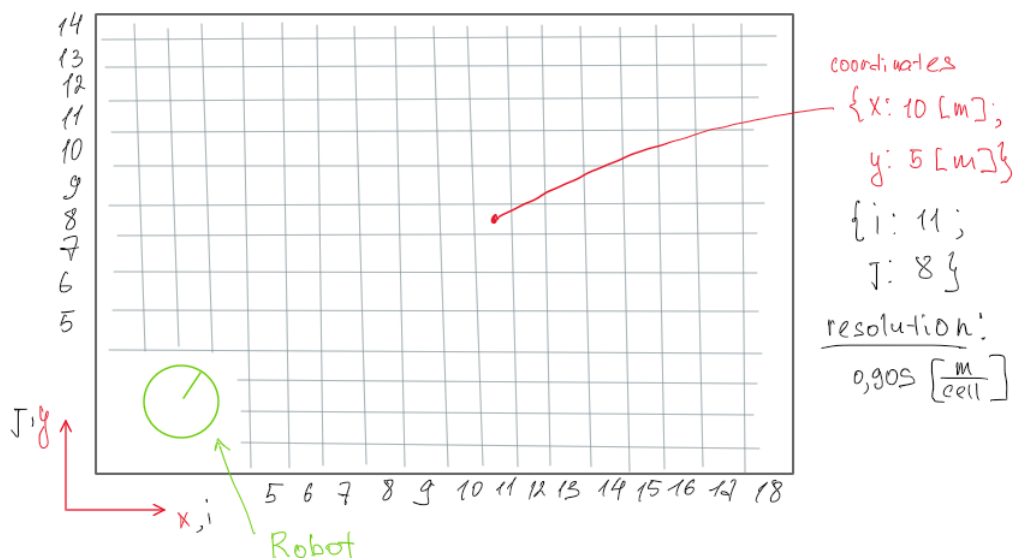


Figure 5.1: Occupancy Grid map illustration

For understanding the concept of frontiers it is important to understand the Occupancy Grid. Occupancy grid is format for storage of the map. Occupancy grid map consists of the cells, which represent physical place on the real map. The real place depends on resolution of the map. For example on fig 5.1 there are coordinates on the map $[x : 10, y : 5]$ [m] and the corresponding coordinates in Occupancy Grid frame is $[i : 11, j : 8]$. So the resolution of the map is $\frac{10}{11} = 0.909[\frac{m}{cell}]$, which defines how much space one cell occupies. Occupancy grid cells can have discrete values for each cell. Depending on application, cells have different range of values, but in case of my thesis each cell could have 3 different values: "-1" - cell is unknown, "0" - cell is unoccupied, "100" - cell is occupied by obstacle. So in current case frontiers are known free cells with value "0", which have at least one unknown cells with value "-1" near them.

Currently there are a lot of algorithms to detect frontiers: Yamauchi's Frontier Detection - YFD [20]; Fast Frontier Detection - FFD, [21]; Wavefront Frontier Detection - WFD, [21]; Incremental Wavefront Frontier Detector - WFD-INC, [22]; Incremental-Parallel Frontier Detector - WFD-IP, [22]; OBB Based Frontier Detection - OFD, [23]; frontier detection utilizing the idea behind a randomly exploring random tree [24]. The text, following after this is explanation of those algorithms, it will start with - YFD.

5.1 Yamauchi's Frontier Detection

The easy way to search for frontiers is to evaluate every cell on the map and to look if it has unknown neighbours. But with bigger maps the time complexity of this algorithm will grow exponentially, here on the fig 5.2. it can be seen, that time complexity of the algorithm for pretty realistic building is so big, that it is not possible to accomplish the task.

| Dimensions (m) | cm ² Cells/cm ³ Voxels | 1 ms | 1us (s) | 1 ns (s) |
|-------------------|--|------------|---------|----------|
| 43.8 × 18.2 | 8.79 × 10 ⁶ | 2.44 h s | 8.79 | 0.01 |
| 292 × 167 | 4.88 × 10 ⁸ | 5.6days | 8.13 | 0.49 |
| 43.8 × 18.2 × 3.3 | 2.90 × 10 ⁹ | 4.8weeks | 48.35 | 2.90 |
| 292 × 167 × 28 | 1.37 × 10 ¹² | 43.41years | 2.25 | 22.76 |

Figure 5.2: Time complexity of naive approach, [25]

So due to the time complexity of the naive approach Brian Yamauchi suggested YFD. The YFD algorithm has 7 steps:

- 1) Get the map in the format of Occupancy Grid
- 2) Conduct a process analogous to edge detection to find the boundaries between open space and unknown space

- 3) Leave only frontiers which have size bigger, than minimum frontier size
- 4) Find centroids of those frontiers
- 5) Move robot to the closest centroid
- 6) If robot safely arrived to the centroid, then add the centroid to the list of accessible centroids. If robot cannot arrive to the centroid, then add the centroid to the list of inaccessible centroids.
- 7) rotate to gather more information
- 8) Repeat until no frontiers are left

This method was successfully implemented on Nomad 200 mobile robot [20]. It had laser rangefinder sensor and 16 sonar sensors, which were equipped around the robot. The data from laser sensor were combined with data from sonar sensors. And also all 16 sonar sensors were used for avoiding obstacles as well.

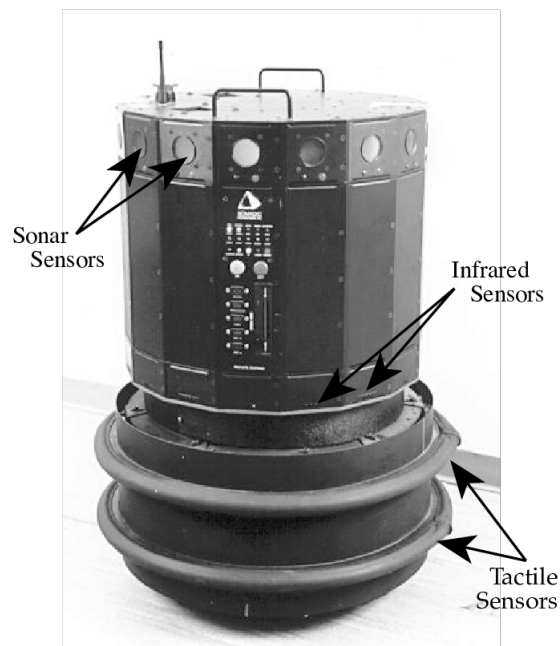


Figure 5.3: Nomad 200 robot, [20]

YFD method was breakthrough at the time, but it is a rather slow algorithm nowadays. The main reason for this is YFD is working with all of the map and it slowing down the whole autonomous navigation process [26].

5.2 Fast Frontier Detection

The difference of FFD from previously presented algorithms is, that FFD evaluates only new data from LDS. Frontiers by definition are borders between known areas and unknown areas, therefore the scan of fully unknown areas is waste of computational power. FFD algorithm is being called after every new scan. FFD algorithm contains 4 steps [21]:

- 1) Sorting the Laser Reading from LDS in polar coordinate system
- 2) Get contour of newly sorted laser readings by computing the line, which lies between each two neighbouring points from laser readings from previous step
- 3) Detecting frontiers. Algorithm goes through points one by one and all points from contour can be either:
 - a) not part of the frontier
 - b) part of the frontier, so were previous points
 - c) part of the frontier, but previous points were not in the frontier
 In case of option "a" algorithm ignores point. In case "b" point is added to the current frontier. In case "c" new frontier is initiated and point is added to it.
- 4) Maintaining previously detected frontiers. FFD can detect frontiers only from new scan readings, so the previously detected frontiers will be lost, if not stored.

The advantages of FFD is, that it is significantly faster than YFD and WFD because of it is processing only new readings. But the main disadvantage is the same. It cannot perform frontier detection on processed part of the map, therefore it cannot expand existing obstacles on the map and cannot functionally work in sensor fusion [26].

5.3 Wavefront Frontier Detection and its Incremental and Parallel modifications

Firstly usual WFD algorithm will be described here. WFD the basis of this algorithm is Breadth-First Search or BFS. BFS is graph based search algorithm, which has a starting point and will search the graph in the layers. At first step algorithm goes to all neighbouring nodes of the starting point, that is first layer. After that it goes to neighbours of the neighbours of the starting point, this is second layer. And it continues to search until it reaches the goal or reaches the end.

Now back to the WFD. The WFD has 4 steps:

- 1) First the position of the robot is established as starting point for BFS

- 2) Initial BFS is conducted on points, which have not been scanned yet. The scan is searching for first points of new frontiers (first unknown points)
- 3) If frontier point is found (unknown point, which went after known and free point) the second BFS is conducted to extract whole points of the frontier. This BFS searches for frontier points only.
- 4) To avoid detecting the same frontiers, corresponding cells are marked as discovered

The new thing in the WFD is that it limits the search area from all cells on the map to the cells, which were not previously scanned. But with growing global map, the area of searching grows too, so the performance time increases. This can be avoided with reducing the resolution of the map, so the cells will cover bigger area, but it can be a risky decision if there could be obstacles, which are smaller, than cell area [26].

Next in this text follow WFD-INC and WFD-IP. Those algorithms are WFD algorithms, but with maintenance algorithm from FFD implemented in it. Maintenance algorithm was described in step 4 of FFD's sequence. WFD-INC is the version of algorithm, who searches for new frontiers only in "active" area [22]. Active area is rectangle, that is defined by most distanced laser readings, conducted during last search. This was made, because active area will contain changes with high probability. And new frontiers can be discovered only in the changing area, or "active" area.

WFD-IP is a lot of WFD-INC instances, which run in parallel for separate areas of the map [22]. The main advantage of WFD-IP in comparison with WFD-INC is, when certain area of map changes to often. In that case the performance of WFD-INC is the same as WFD and WFD-IP runs faster [22].

5.4 Oriented Bounded Box algorithm based Frontier Detection

OBB based Frontier Detection or OFD is solving the problem, where WFD-based methods generate a lot of frontiers, which will be cancelled during next update of the map, therefore a computational power, used for their detection was wasted. OFD separates exploration mission on epochs, where epoch is ξ -time generation of the map. The duration between two epochs is called "update window" [23]. The epochs are illustrated on fig 5.4.

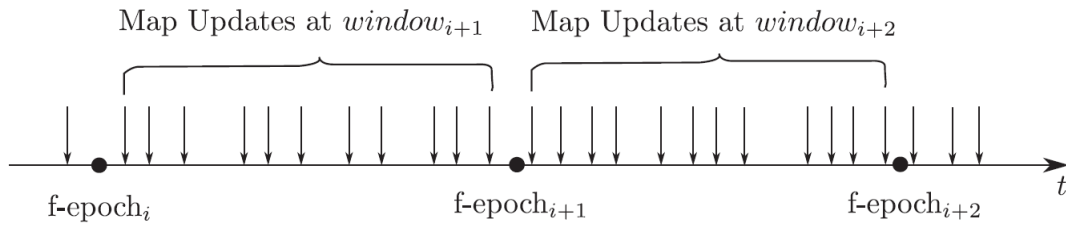


Figure 5.4: Abstraction of an exploration mission in terms of frontier generation operations and map updates, [23]

OFD is updating frontiers contours while moving from one epoch to another. During one epoch robot moves and gathers more information. But instead of cancelling old frontiers OFD updates old frontiers. The way it happens is showed on pic 5.5.

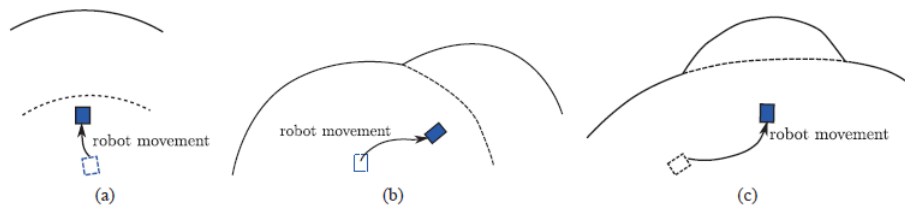


Figure 5.5: Frontier progression : (a) Deleting an entire frontier-contour and creating a new one (b) Extending a frontier-contour from one corner (c) Extending a frontier-contour from the center. Dark Lines indicate current frontier-contours and dashed lines indicate deleted frontier-contours, [23]

Three main ways are illustrated on fig 5.5. On fig 5.5a OFD will update frontier by deleting old-one and creating one, which is further from old position. But at fig. 5.5b and c frontiers are not being deleted, but updated instead. This saves computational power, because robot remembers unchanged line of frontiers and does not need to scan and discover it again. The way all frontiers are being extracted is with help of OBB algorithm [23].

5.5 Rapidly-exploring Random Tree Frontier Detection

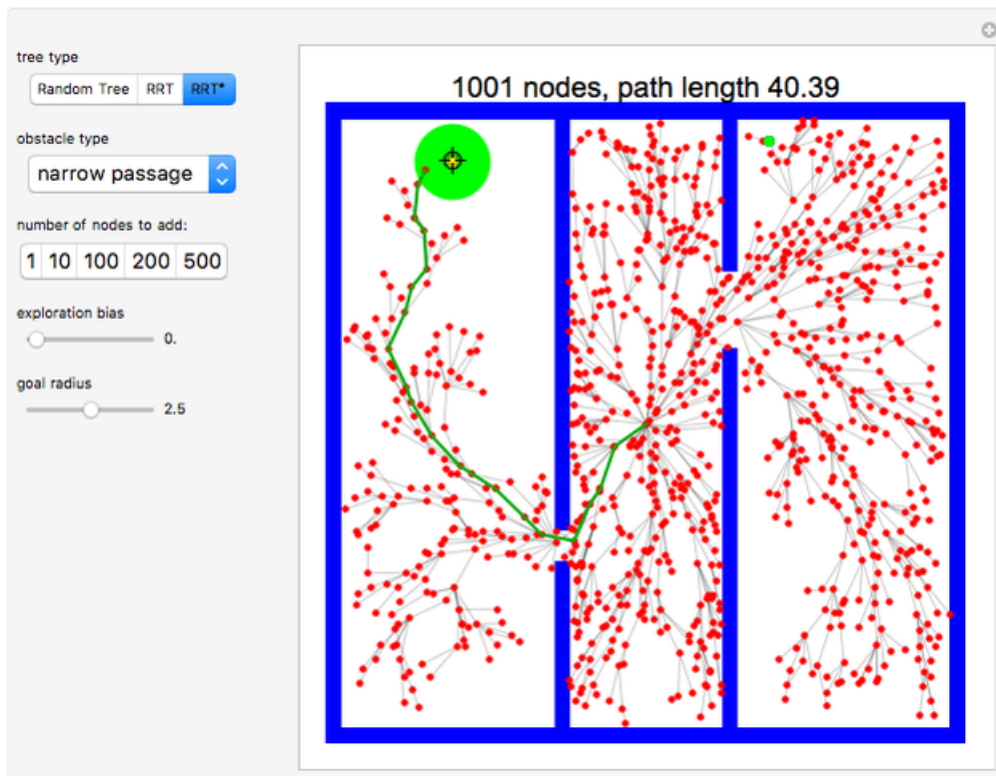


Figure 5.6: Example of RRT, used for path planning of autonomous vehicle

Rapidly exploring Random Tree algorithm or RRT is designed for efficient search by building a sparse-filling tree, which stores the position of those random nodes on the map and distance to neighbouring nodes, strictly speaking it rapidly creates a graph, which fills the empty unknown spaces on the map, making them known.

Conventionally this algorithm was more used for generating paths for exploration, but it can be also used for detecting frontiers in the same way. The authors Quio, Fang and Si proposed to modify RRT. Algorithm has a set of valid nodes from the tree, which are accessible (not obstacles, not in close areas). And every time a new random point is proposed to become a node, it does not add it to the tree, it finds the nearest node from accessible list and only after that a new edge of the graph is generated, therefore a new random point becomes a node. And the way it detects frontiers follows like this: if node and edge falls in known area, but edge falls on unknown and known area at the same time, then the unknown point on this edge adjacent to the free area is a frontier point [24].

Chapter 6

Chosen frontier finding algorithms

Following chosen algorithms are described vaguely and only in the written form by their creators, sometimes general principle is also described with pseudo code, but some steps, for example "maintaining previously frontiers" are not described at all. So the main requirement for the implementation of those algorithms is to fulfill prescribed steps. My implementation of those algorithms and fulfilling prescribed steps is coming up with new algorithms, which will not only do prescribed operations, but will find frontiers, store, extract them, manipulate with them to store them right way, extract goal from frontiers and deploy it to the robot right way. All of the above steps will be described in practical part of the thesis for each chosen algorithm.

Standalone Frontier Finding Algorithm cannot map unknown environment and localize in it. Frontier Finding Algorithm, which I will be implementing in practical part of this thesis is working with help of other algorithms. I will use toolboxes with those already implemented algorithms. Those main toolboxes are: `slam_toolbox` and `actionlib`. `Slam_toolbox` renews the map and detects newly introduced obstacles or free space, when robot moves to unknown space, `slam_toolbox` is implementation of SLAM. `Actionlib` are ensuring work of action server and action client, client is computer which sends goal and server, which runs on the robot, ensures, that robot will move towards the goal and will reach it.

6.1 Naive Frontier Detection Algorithm

The first chosen algorithm is not very sophisticated, but it is the first solution, which would come to anybody's mind and it is most met solution online, when it comes to autonomous navigation and mapping. First algorithm receives map, searches for all unknown cells on this map and chooses one of all unknown cells with uniform "equal" distribution.

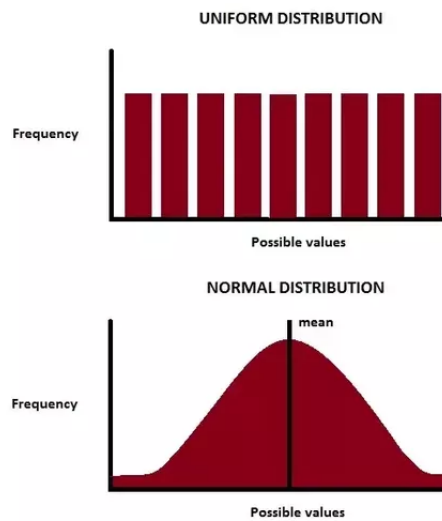


Figure 6.1: Uniform distribution, used for naive finding frontiers

As it can be seen from fig 6.1. any unknown point can be selected to be frontier with the same probability. This method does not define frontier the same way as Brian Yamauchi does in his article [20]. Instead, here frontier is all the unknown cells around robot and the algorithm decides to witch part of the frontier to go. It has advantages, such as: it is easy to implement, it is not computationally hard and its dependence on the size of the map is very little. Its main disadvantage is, that this algorithm cannot be really controlled. It is just decides to go to the cell witch was chosen from all unknown cells with normal distribution

6.2 Yamauchi's Frontier Detection Algorithm

When I began to write this thesis, I used DFD abbreviation for Yamauchi's Frontier Detection Algorithm, as in Direct Frontier Detection. But then I found out that DFD exists and translated as in Dynamic Frontier Detection and I stopped to use this abbreviation, despite this in developed software, some objects are named with DFD abbreviation and in terms of this thesis DFD abbreviation is the same as YFD or Yamauchi's Frontier Detection. The next chosen method is invented by Brian Yamauchi and described in his article [20]. This method defines frontier as boundary between known cells and unknown cells and suggest to find that boundary by obtaining the map and pass it through the "process analogous to edge detection and region extraction in computer vision is used" [20]. After boundaries are obtained the next step of the algorithm follows. The next step is sort out algorithm by its closeness to the position of the robot and filter them out by minimal size of the frontier. Then move to the centroid of nearest frontier. The example

of the frontier detection with YFD can be seen on fig 6.2.

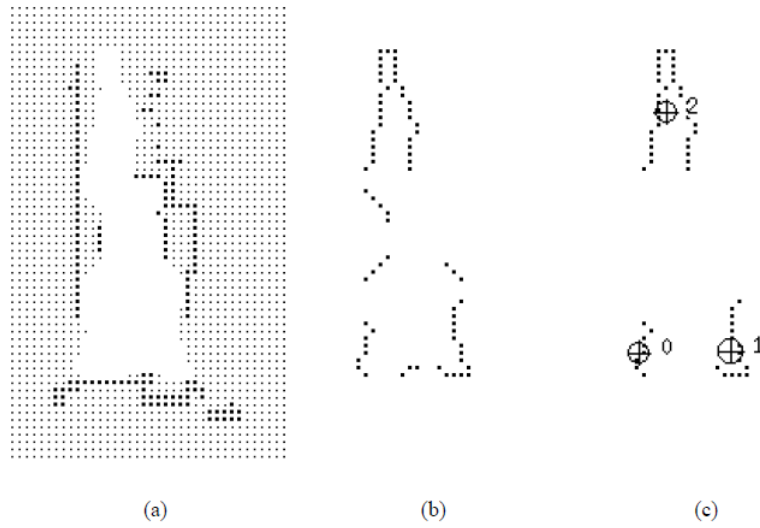


Figure 6.2: Example of YFD frontier detection, (a) evidence grid, (b) frontier edge segments, (c) frontier regions, [20]

The advantages of this method is that it is very consistent. It always works with full map and no unknown areas should not be missed by this algorithm. The robot should be always moving very tightly with discovered frontiers, gradually discovering new parts of map and expanding boundary between known and unknown. But this method has disadvantages, the main one is than as map grows, algorithm requires more and more computational power and can become quite slow with big maps [26].

6.3 Fast Frontier Detection Algorithm

This chosen algorithm was recently introduced by M. Keidar and GA Kaminka in their thesis [21]. This algorithm rethinks the concept of the frontier. In YFD Yamauchi finds frontier on all the map in every iteration of the loop and can be working with old data, for example part of the map, which was last updated couple of iterations ago. FFD on the other hand assuming that frontiers can be found only in newly obtained data from laser sensors. So first step of algorithm is obtaining the measurements from laser sensors, it can be seen on fig 6.3.

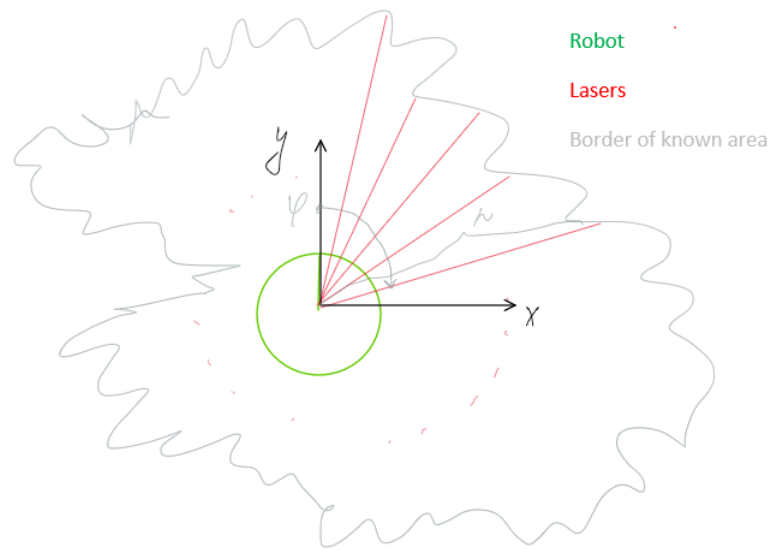


Figure 6.3: Obtaining laser sensors readings

Laser sensors return information about distance to the obstacles around robot, so on the map it can be displayed as points around robot as it can be seen on the fig 6.4.

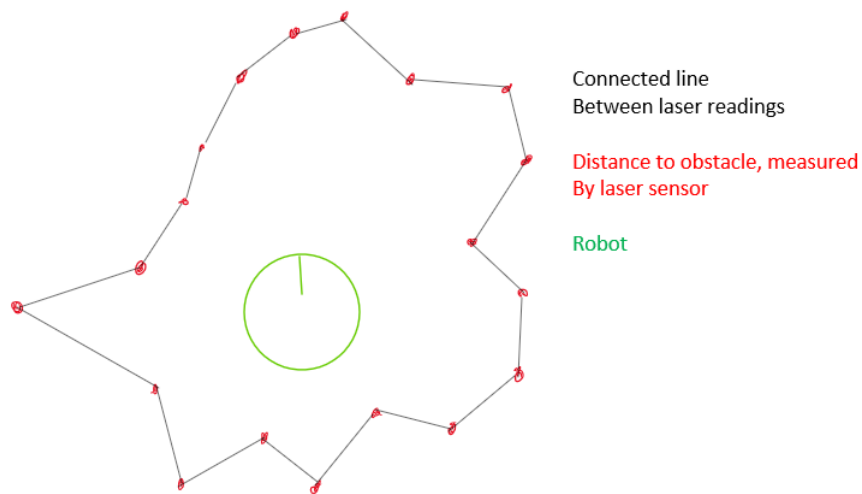


Figure 6.4: Connected readings from laser sensors

After that dots will be connected by algorithm, forming the border between known and unknown. The line connecting the dots will be inspected. If point in the line has unknown cells as neighbours and does not have obstacles as neighbours it will initiate new frontier or will be included in already existing frontier: 6.5. So as it can be seen, frontiers here are not always boundary between known and unknown cells, but the line between laser measurements, that has unknown area near it. The conception of frontiers has been changed in this method.

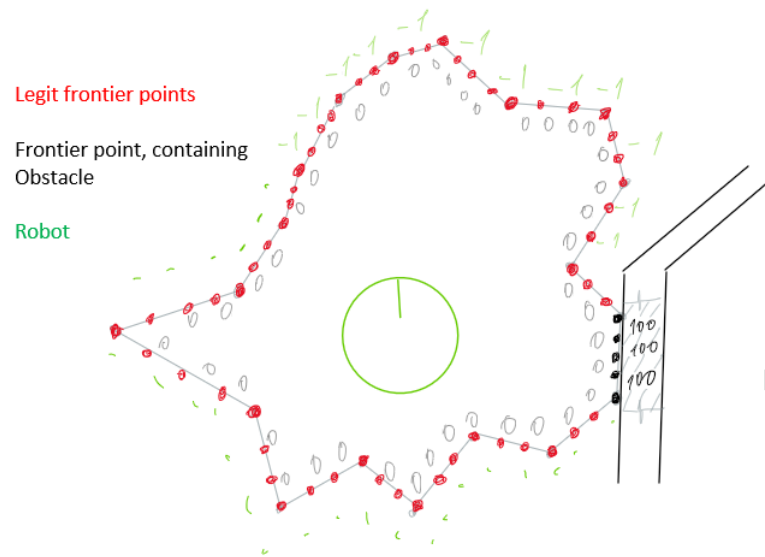


Figure 6.5: Obtaining laser sensors readings

Chapter 7

Conclusion of the theoretical part

The theoretical part of this masters thesis serves a purpose to be explanation and preparation for the practical part. So in this part of masters thesis I have explained important working principles of the methods, I chose to implement and working principles of toolboxes, used in a bundle with Frontier Finding Algorithms to make efficient program for autonomous mapping of the environment and localizing in it.

In chapter 2, I described robot and cited its important technical characteristics.

In chapter 3, I described principle of working of the main tool: ROS. I have described the architecture of this tool. The principles of two programs communicating with help of ROS and cited important technical information like supported languages for developing with this tool, protocols used for communication between software and hardware units with this tool and types of developed useful tools.

In chapter 4, I have defined second most important algorithm in terms of this thesis: SLAM. I have written about two basic groups of SLAM algorithms: EKF-SLAM and GraphSLAM. I have explained its mathematical essence and gave the example of using SLAM. Also I have listed main challenges of this algorithm.

In chapter 5, the examples of different Frontier Detection Algorithms were given. In this chapter the most important concept in terms of this thesis were introduced. Frontier Finding Algorithms are main focus of this thesis. Basically those algorithms are processing the known part of the map and they decide, where robot should go to discover the biggest new part of the map.

In chapter 6, I have stated, that Frontier Finding Algorithms are useless standalone without SLAM algorithm and algorithm, that moves robot to the goal working along. I have listed toolboxes witch I used to perform SLAM and moving robot to the goal. But most importantly I have listed chosen frontier finding algorithms and described more specifically the steps of those algorithms.

Chapter 8

Developed software and its structure

Practical part of this thesis will start with the explaining of the structure of the developed software. It is critical, that before explaining implementation of separate objects, the whole structure would be explained. Written code has 1500 lines, so describing every line would take a lot of space and it would be very confusing. So in order to explain the structure of the developed software I will use the concept of UML.

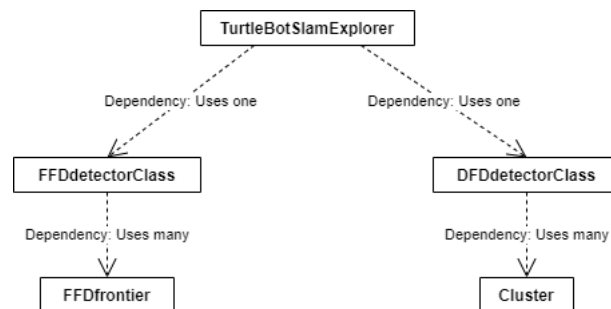


Figure 8.1: Conceptual class diagram of the code

As it can be seen of fig 8.1. there are 5 classes in my code. The structure of class diagram is vertical. Also the diagram does not contain attributes and methods of the classes. The attributes and methods will be shown in later sections of this chapter for each class individually.

The main class in my code is **TurtleBotSlamExplorer**. This class represent the robot itself and implements all technical features, needed for correct functioning of robot and correct autonomous navigating, mapping. Also first method - naive detection method, which I described in chapter 6.1. is implemented as method for this class.

Two classes on the second level of this diagram are **FFDdetectorClass** and **DFDdetectorClass**. They implement FFD and YFD algorithms respectfully. Those algorithms were mentioned in chapter 6.3. and 6.2. Depending on which method user chooses to use for autonomous mapping of the unknown environment: FFD or YFD, TutleBotSlamEx-

plorer class initialises FFDdetectorClass or DFDdetectorClass and use the same initialised object in all subsequent loops. Those classes are used to process the map and to return target coordinates for robot to go to discover biggest new part of the map most efficiently.

Left two classes on the third level serves as datatypes for FFDdetectorClass and DFDdetectorClass. If user chooses to use FFD detector class then all frontiers, found by that class are stored, trasformed to another coordinate system and etc. in **FFDfrontier** class datatype. Else if user chooses to use YFD for autonomous mapping, then clusters, which can either be frontiers or not and should be controlled for matching the minimal requirements to be frontier, are stored in the **Cluster** class datatype.

All of the above mentioned classes are fully developed by me and implement code that fully capable of turtlebot to do autonomous navigation and mapping with all three chosen methods of frontier detection.

In the next chapters the more deeper explanation of how each class works follows.

8.1 Turtlebot class

This section will begin with full class diagram for TurtleBotSlamExplorer with all its methods and attributes on fig 8.2

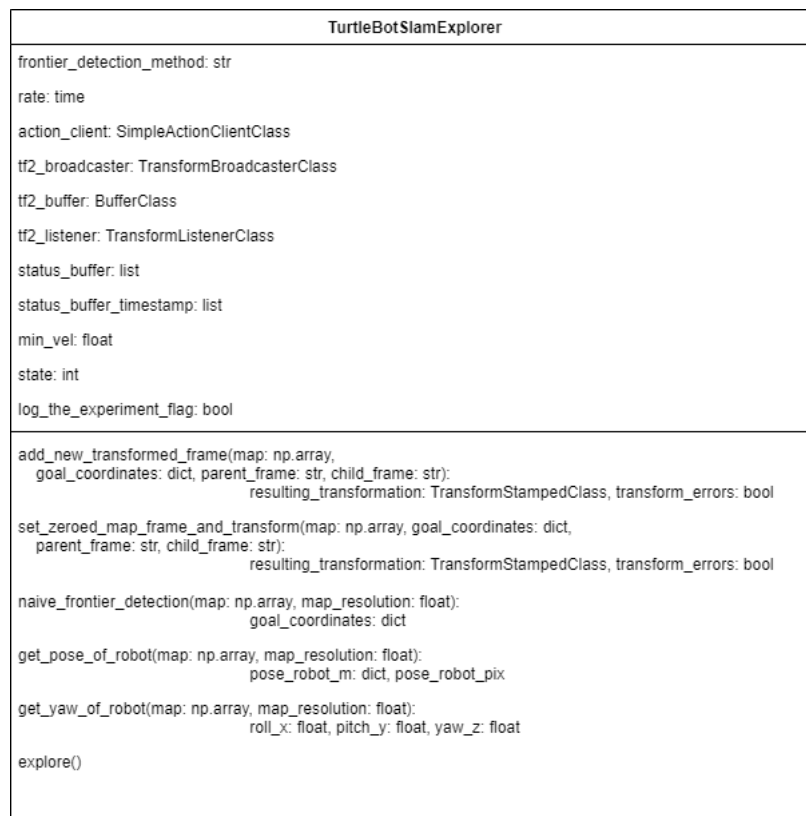


Figure 8.2: TurtleBotSlamExplorer class diagram

In the upper box there are listed all attributes of TurtleBotSlamExplorer class and in the lower box there are all methods of this class. The full description will begin with its attributes.

The first attribute is **frontier_detection_method**, it is the string, which is passed to the object in time of its initialisation to define which method will be used: FFD, YFD or naive.

Rate attribute is time in seconds to define the cycle time for the ROS. During that cycle time all operations, included in

```
while not rospy.is_shutdown():
```

loop should be conducted.

action_client attribute is initialised SimpleActionClient object from Actionlib toolbox, which I have mentioned in chapter 3.1. SimpleActionClient is interface to communicate with action server, which runs in turtlebot and makes it to go to the specified location on the map [9].

tf2_broadcaster attribute is TransformBroadcaster class initiated with the initialising of TurtleBotSlamExplorer class. This is class from tf2 toolbox, I have described this toolbox in chapter 3.3. This class sends the newly added frames with its transformation from known frames.

tf2_buffer attribute is also initialised BufferClass from tf2 toolbox, it stores the requested transformations.

tf2_listener attribute is TransformListener class from tf2 toolbox, with its help user can extract requested transformation from previously initialised Buffer class object from tf2 toolbox

StatusBuffer is list, that stores terminal statuses of last 10 goals. SimpleAction server sends statuses after goal has been reached, canceled or etc. This attribute serves as buffer for last 10 of those statuses.

StatusBufferTimestamp stores timestamps of last 10 goals, when they reached their terminal state.

min_vel attribute stores velocity with which turtlebot moves. It is set as maximal allowed velocity for turtlebot. It is used for calculating the approximate time of turtlebot approaching his goal.

State attribute is current state of state machine operating in explore() function of TurtlebotSlamExplorer class.

And finally last attribute of this class is **log_the_experiment_flag**, it is boolean value, that indicates, whether some values of current exploration should be recorded and saved to evaluate them later.

The other things, that should be explained next is the methods of this class. The main function of this class is `explore()`, so the explanation of methods will start with this function.

8.1.1 Explore method

This function works by the principle of finite state machines, this is why the explanation will start with the state diagram of this function. This function is mean to be triggered right after initialisation of `TurtleBotSlamExplorer` class, so the diagram will start with triggering of the function, but the only thing that was before triggering that function is initialising of `TurtleBotSlamExplorer` class, so basically the diagram will start from the moment my whole program is launched from terminal. Diagram is on the fig 8.3. All further described functions and classes will take its place inside of this state machine. So this diagram is base of my whole project.

The first **INIT** state is quite small. Depending on the user's choice of Frontier Finding Algorithm there will be initialised `DFDdetectorClass` object or `FFDdetectorClass` object or no object if naive method is chosen. Also map will be fetched from `slam_toolbox`, which sends it under topic name `"/map"` and saved to the variable `"previous_map"` and compare it with newly fetched map in second state to estimate if robot discovered new parts of map or not i.e. map changed or not. After all operations conducted state machine moves to the next state without any conditions

The second state **SET UP GOAL IN MAP COORDINATE FRAME** is beginning with again fetching the current map. After that the position of origin in the map coordinate frame i.e. position of cell `[0, 0]` in the map 2D array and map resolution is being obtained from message under `"/map"` topic. After that the message after topic `"/scan"` which contains all laser readings for 360° from LDS is being fetched from `turtlebot`. After that the pose of the robot is being calculated from `TurtleBotSlamExplorer` class method `"get_robot_pose()"`, the principle of this method will be explained in following chapters. Next move is to obtain yaw rotation of robot i.e. his angle of rotation around z-axis of the map coordinate frame. Next follows the implemented by me methods of frontier detection, depending on which method user preferred to choose, respected object is processing data, obtained from `turtlebot` directly and through `slam_toolbox` and `tf2` toolbox. Different methods require different quantity of information. Naive method requires only map and its resolution. `YFD` method requires map, previous map, current position of the robot and resolution of the map. `FFD` requires all of the above and yaw rotation angle of robot on top of it. The last operation of this state is writing current map into the variable `"previous_map"`.

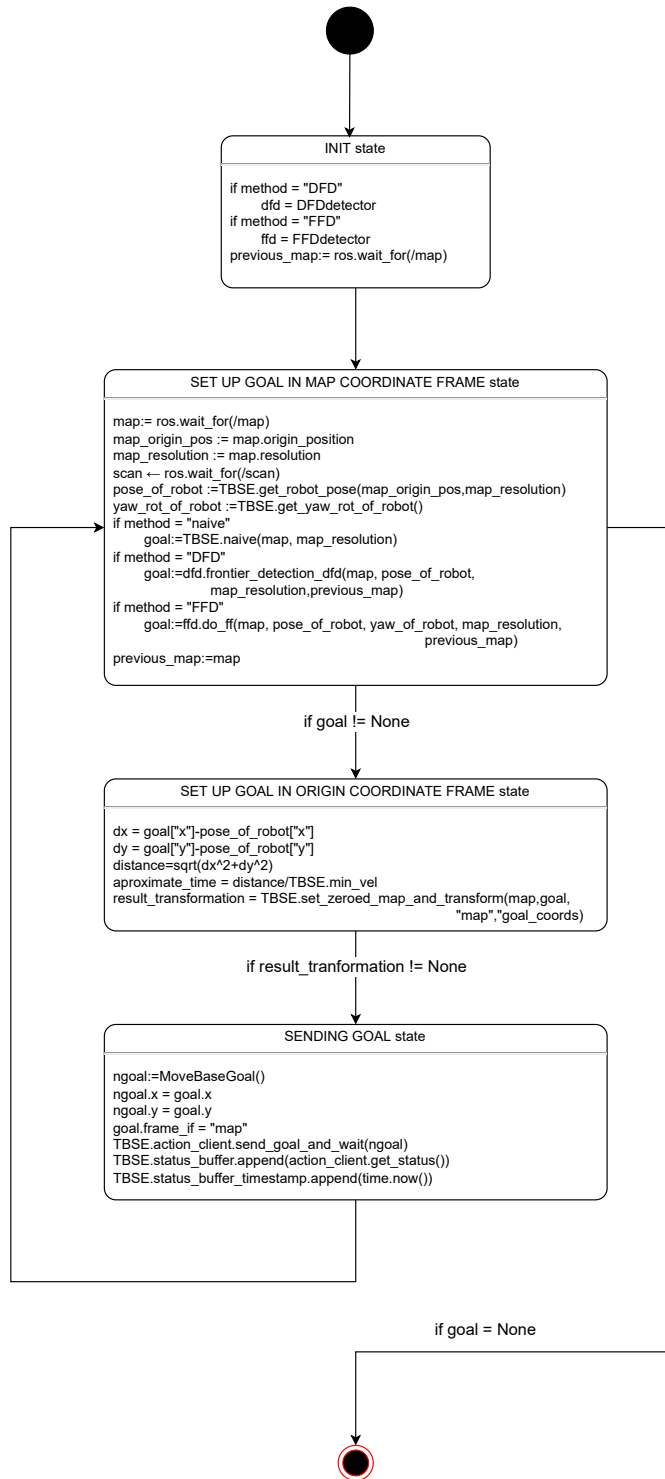


Figure 8.3: State machine within `TurtleBotSlamExplorer.explore()` function

After all operations conducted state machine moves to the next state only if frontier finding method found the goal. If frontier finding method could not find the goal the state machine terminates itself. That should happen only if there if no unknown area left

Moving on to the third state: **SET UP GOAL IN ORIGIN COORDINATE FRAME**. This state is beginning with approximate calculation of time which should take turtlebot to get to the goal. We know position of the robot and goal coordinates in the same coordinate frame and, we know maximum allowed speed for turtlebot, from that we can calculate the approximate travel time and later to set the deadline for the turtlebot to get to the goal. After that the method `TurtleBotSlamExplorer` class is being called. `Set_zeroed_map_and_transform` method will be fully explained in following chapters. For now the definition of this function goes like this: `Set_zeroed_map_and_transform` method takes goal coordinates in origin coordinate frame, makes a new coordinate frame called "goal coordinates", derived from map coordinate frame and searches for transformation between those two frames (meaning map coordinate frame and "goal coordinates" coordinate frame). That helps to find goal coordinates in map frame (originally they were in origin coordinate frame) and sets a tool for debug, because in RViz program, which is used to visualize slam mapping process, coordinate frames can be easily added to the shown objects. That way goal coordinates can be easily shown in RViz debugging tool. Resulting transformation is saved in "result_transformation" variable and in essence this resulting transformation between map coordinate frame and "goal coords" coordinate frame is goal coordinates in map coordinate frame. The transformation is conducted by asking the tf2 buffer if he knows the transformation. Sometimes buffer is overloaded and sometimes it just cannot find transformation. So if resulting transformation was not found - state machine stays in this state and waits until tf2 buffer catches up with request. If transformation is found, state machine moves to the next state. Basically `set_zeroed_map_and_transform` method transforms goal coordinates from origin coordinate frame to map frame (more on coordinate frames list 8.4, fig 8.5.).

The forth and the last **SEND GOAL** state. Starts by defining the goal. Actionlib does not accept goals in any suitable for user format. It accepts goal strictly as `MoveBaseGoal()` object from ROS external package. So at the beginning of the state the properties of initialised `MoveBaseGoal` object should be changed to correspond found goal coordinates. The main properties that should be changed from default are `MoveBaseGoal.x`, `MoveBaseGoal.y` and `MoveBaseGoal.map_id`; x, y are the coordinates in `map_id` coordinate frame. After it is changed goal can be send by `action_client` attribute object. Method "`send_goal_and_wait()`" of `SimpleActionClient` class sends the goal and waits till the goal reaches its terminal state. After that status buffers fetching new status and timestamp of latest goal and state machine goes to the next iteration of


```
while not rospy.is_shutdown():
```

loop.

At the end of explanation of this base function for my thesis, the problems and my solutions to them will be listed. The state machine is not usual way to implement the main function for robotic exploration, but due to problems I have encountered with Actionlib server client communication (goal statuses are not renewing in time, sometimes server confuses two commands etc.) And decided that the best way to implement base function is to make it state machine. That way different part of code would be separated from each other and problems such as goal slippage due to delay in change of goal status would occur less. Delay in changing the goal statuses causes that next 3 or 4 found frontiers are marked as reached, but they not actually reached and new parts of the map stay undiscovered. The control of the rising edge of the status or callback features for goal status change are not implemented either. Rising edge control also works badly due to possible fluctuations of status. So in this situation the best solution to prevent goal slippage I have found is implementing the state machine.

8.1.2 Get robot's pose method

Get robot's pose method is not so trivial as it may seem. This method is not just obtaining message from turtlebot, which contain the position of the robot in any frame you like. Typically in robotics project there are several frames existing for the comfort of computation [12]. My thesis is the same. On the map there are several coordinate frames present and different messages from turtlebot come from different frames. So before beginning of explaining this method the explanation of relevant frames and relation between them will follow.

On the fig 8.5. all frames, present in typical "exploration with robot" project, developed with ROS present. And on the fig 8.4. those typical frames are fully described.

The best frame for conducting all the frontier detecting operations is origin frame. Because frontier detecting operations are conducted with array, containing the map, and to refer to any cell in this array, it needs to be done in terms of origin frame. For example if it needs to be referred to cell, which lies in coordinates

```
{"x": 0,52 [m], "y": 1,49 [m]}
```

in map frame coordinate system. Firstly its position in origin frame should be found and then it should be discredited (devided by map resolution), After that we will get the number of row and number of column of map 2D array of the desired cell, this procedure is illustrated on fig 8.6.

Figure 8.4: List of typical frames, present in exploration project, developed with ROS

- **base link frame** - is basic coordinate system of the robot, all measurements from sensors, mounted on the robot come in respect to base link frame
- **map frame** - coordinate frame that spawns randomly on the specific place on the map and stays on this place on the map despite odometry errors. All the information about position of something on the map, that needs to be stored and used after some time - needs to be transformed to the map frame, as this frame is consistently stays on its origin spawn place
- **odom frame** - at the start of exploration mission is identical with map frame, but as cumulative odometry errors increase - it drifts from the map and transformation between map frame and odometry frame shows the cumulative odometry error turtlebot collected during the exploration mission. The message about position of the robot is obtained with respect to odometry coordinate frame
- **origin frame** - origin frame is coordinate frame, which origin lies at $[0, 0]$ cell in the map array message. Simply speaking, transformation between map frame and origin frame shows how far is cell $[0, 0]$ in the map array from the map coordinate frame origin.

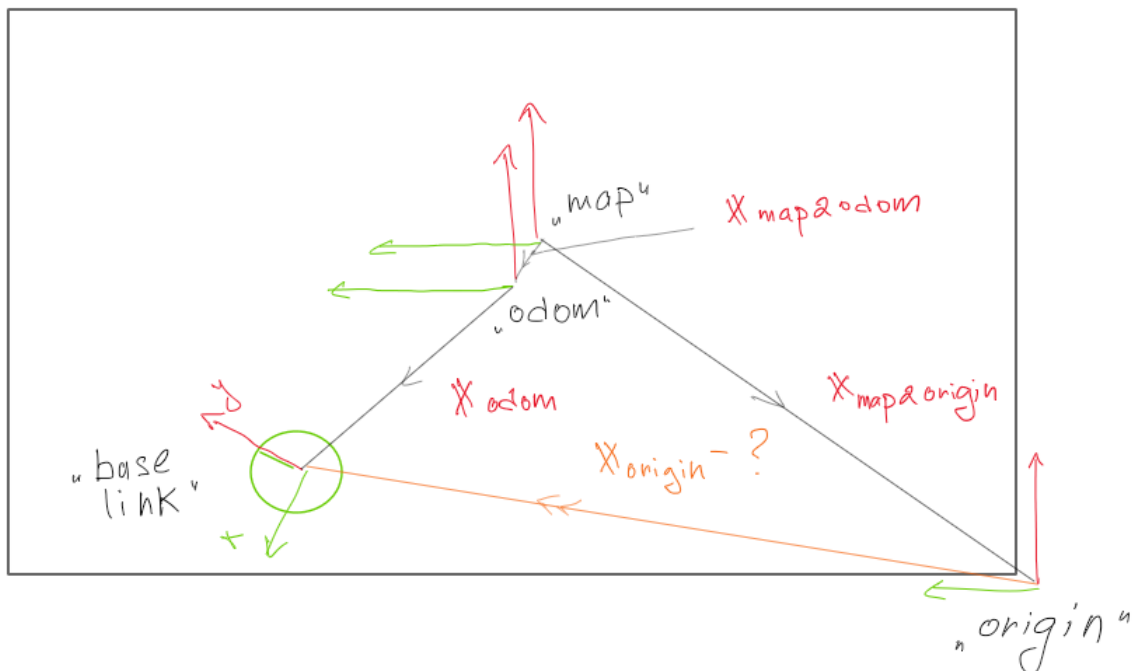


Figure 8.5: Typical frames, present in exploration project, developed with ROS

So the position of the robot must be obtained in origin frame, it should be done for the comfort of the calculations: calculation of distance of robot from frontier, calculation of distance to the obstacles and etc. But turtlebot sends the message about its

position based in the odom map frame. So the position of robot in the origin frame should be calculated, and that is the purpose of the `get_pose_of_robot()` method of the `TurtleBotSlamExplorer` class.

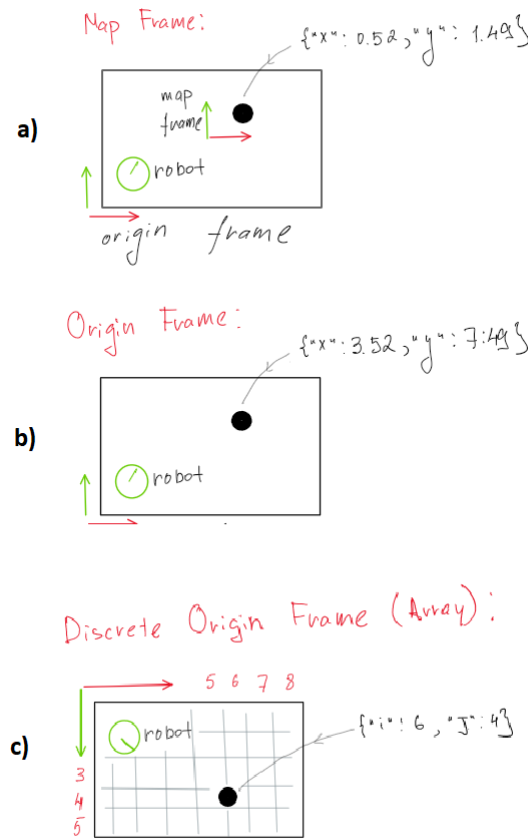


Figure 8.6: Procedure of obtaining the row and column number of 2D array, where the desired cell is situated.

- Map frame coordinates
- Origin frame coordinates
- Desired row and column in map 2D array

Now keeping in mind all of the above mentioned principles, the explanation of the code follows. If all of the missing vectors like: $\vec{x}_{map2odom}$, $\vec{x}_{map2orig}$ from fig 8.5. could be obtained or calculated, then the position of robot in origin frame would be just the matter of vector composition. The \vec{x}_{odom} is known and obtained from message of turtlebot, where it deploys its location in odom frame.

So next in my thesis follows the code of `get_pose_of_robot()` method on the figure "Algorithm 1". As it can be seen, all vectors from fig 8.5. can be obtained. Algorithm begins with obtaining \vec{x}_{odom} from message under topic `"/odom"`. This vector represents position of the robot in the odom coordinate frame, after this algorithm tries to obtain vector $\vec{x}_{map2odom}$ with help of the `tf2` toolbox (more: chapter 3.3.), this vector represents

transformation to odom frame coordinates from map frame coordinates. Actually this transformation can be obtained only with help of tf2 toolbox, as it internally monitor accumulated odometry error and moves odom frame accordingly. And the most effective and useful function of tf2 in terms of the task of my thesis is to get this transformation.

Algorithm 1 An algorithm of getting robot's pose in origin frame

```

raw_odom_message ← ros.wait_for('/odom')
 $\vec{x}_{odom}$  ← raw_odom_message.position
Try:
     $\vec{x}_{map2odom}$  ← self.tf2_buffer.lookup_transform(mapframe, odomframe)
Catch tf2_connectivity_error:
     $\vec{x}_{map2odom}$  ← ('x' : 0, 'y' : 0)
raw_map_message ← ros.wait_for('/map')
 $\vec{x}_{map2origin}$  ← raw_map_message.origin_pose_map_frame
 $\vec{x}_{origin\_m}$  ←  $\vec{x}_{map2odom}$  +  $\vec{x}_{odom}$  -  $\vec{x}_{map2origin}$ 
 $\vec{x}_{origin\_pix}$  ← ('x' :  $\vec{x}_{origin\_m}[x]/map\_resolution$ , 'y' :  $\vec{x}_{origin\_m}[y]/map\_resolution$ )
return  $\vec{x}_{origin\_m}$ ,  $\vec{x}_{origin\_pix}$ 

```

After that algorithm obtains last vector, needed for get the position of robot in the origin frame: $\vec{x}_{map2origin}$, this vector represents the position of cell [0, 0], the full description of this vector is on the fig 8.4.

The last step of the algorithm is to add those vector and obtain the desired \vec{x}_{origin_pix} vector. So the algorithm conduct vector composition accordingly fig 8.5. and returns the position of robot in origin coordinate frame in two units: meters in the orig frame and row and table in the discredited map 2D array.

In the end of this section the description of problems and my solutions will follow. Typical way to obtain robot's position is to ask tf2_buffer object for all above mentioned transformations, but sometimes buffer is overloaded, so it does not give answer for request or simply cannot calculate some transformations, so my idea for implementation of this function was to obtain all of transformations not via tf2_buffer, but to collect them from different messages and restrict usage of tf2_buffer only for obtaining the transformation between map frame and odom frame, as this transformation can only be gotten from td2_buffer. And my idea worked! The usage of calculations instead of tf2_buffer services made overall code run faster and smoother.

8.1.3 Get robot's yaw rotation method

Robot's rotation yaw angle is necessary for FFD detection method, which will be described in following chapters. Yaw rotation angle can be obtained in a similar way as robot's position is obtained, by singling out rotation part of the message under "/odom" topic, which turtlebot sends. Only problem is that information about roll, pitch and yaw angles

are encrypted in the quaternion format instead of usual rotation matrix. I did not have enough time to fully understand the conception of quaternion, so **I took the code for transformation from quaternion to roll, pitch and yaw angles from this source: [27]**. So basically this function is taking message from turtlebot under "/odom" topic, singles out information about robot's overall rotation in form of quaternion from this message and uses code, I copied from this [27] source to convert quaternion to euler angles: pitch, roll and yaw.

8.1.4 Add new transformed frame method

This method allows to add new frames derived from old ones to `tfw_buffer` attribute of `TurtleBotSlamExplorer` class, for `tf2` toolbox to keep track of them.

Algorithm 2 An algorithm of adding new coordinate frame

```

procedure ADDNEWTRANSFORMEDFRAME(parent_frame_name,
                                   child_frame_name,translation,rotation)
  new_transform ← TransformStamped()
  new_transform.stamp ← time.now
  new_transform.parent_name ← parent_frame_name
  new_transform.child_name ← child_frame_name
  new_transform.rotation ← rotation
  new_transform.translation ← translation
  self.tf2_broadcaster.send(new_transform)
end procedure
  =0

```

As it can be seen from "Algorithm 2", this function just simplifies working with the already existing interface of `tf2` toolbox. At the beginning it initialises class "TransformStamped()" to the variable "new_transform", then adds required metadata about new frame and then sends it to the `tf2_buffer` with the help of `tf2_broadcaster`.

8.1.5 Set zeroed map method

This method was written in the beginning of the development of this program and of the whole `TurtleBotSlamExplorer` class, so it contains solutions, that, as I thought, were good at the time. Looking back at this code, I know, that I should have redesigned it if I had had more time. Essentially, this method does the following: it takes goal coordinates that are in the origin frame (more on frames: fig 8.4, 8.5, chapter 8.1.2.) makes a new coordinate frame called "goal coordinates" with origin in goal coordinates, situated in the origin frame and searches for transformation between goal coordinates and the map frame. That needed to be done to later input this transformation as the goal, because the

goal should be input as coordinates on the map coordinate frame. And basically: goal coordinates in the map coordinate frame = transformation between "goal coordinates" frame and map coordinate frame. This also helps to visualise the goal coordinates in RViz visualisation program, as any coordinate frame can be easily added to the list of the shown objects. That is why it is called "set zeroed map" method. Because it basically allows to input goal in the map coordinate frame, which origin is located in the cell [0, 0].

The pseudo code of this method is as follows:

Algorithm 3 An algorithm of getting robot's pose in origin frame

```

procedure SETZEROEDMAP(map, goal_coordinates,
                        parent_frame_name, child_frame_name)
  transformation_errors  $\leftarrow$  False
  self.add_new_transformed_frame(parent_frame_name,
                                "origin", pose_of_origin_in_map_frame)
  self.add_new_transformed_frame("origin",
                                child_frame_name, goal_coordinates)

  Try:
    resulting_transformation  $\leftarrow$  TBSE.tf2_buffer.lookup_transformation(
                                      parent_frame_name, child_frame_name)

  Catch tf2_connectivity_error:
    transformation_errors  $\leftarrow$  True

  return resulting_transformation, transformation_errors
end procedure

```

This function is used in my code like this:

```

1  resulting_transformation, transform_errors = self.set_zeroed_map_frame_and_transform(
     $\leftrightarrow$  raw_map, goal_coords, "map", "goal coordinates")

```

And in this case, as it can be seen from "Algorithm 4", this function simply transforms goal coordinates from origin frame to map frame, to later send this goal to action server. This is to be done the following way, firstly algorithm adds origin coordinate frame to *tf2_buffer* derived from map coordinate frame with the help of *add_new_transformed_frame()* method of the *TurtleBotSlamExplorer* class. After that, algorithm adds new "goal coordinates" frame, derived from origin coordinate frame with the help of the same *add_new_transformed_* method. Later on it asks *tf2_buffer* to find a transformation between map coordinate frame and "goal coordinates" frame, which is equivalent to simply transforming goal coordinates from origin coordinate frame to map coordinate frame. The variable "*resulting_transformation*" contains the goal coordinates, transformed from origin coordinate frame to map coordinate frame.

I am going to present the critique of the method that I developed below. I think it should be realised with less usage of *tf2* toolbox, more like "*get_robots_pose()*" method was implemented. The reason for this is that *tf2* toolbox cannot lookup for transformation

in loop every time, sometimes it overloads, cannot find transformation or cannot even push the new frame to the `tf2_buffer`. But unfortunately I thought about it too late and there was no time left for redesigning this method of `TurtleBotSlamExplorer` class.

8.1.6 Naive frontier detection method

This is the first of three frontier detection methods chosen for implementation, that were described in the chapter 6. It is the only one out of the three, which is implemented as method for `TurtleBotSlamExplorer` class and does not have its own class. The algorithm for frontier detection with this method looks like this:

Algorithm 4 Naive frontier detection algorithm

```

procedure NAIVEFRONTIERDETECTION(map, map_resolution)
  frontier_indices  $\leftarrow$  np.where(map == -1)
  IF frontier_indices is empty :
    print("Error finding frontiers!")
  END_IF
  chosen_frontier_idx  $\leftarrow$  uniform_distribution_random(
    0, frontier_indices.size[1])

  goal_coords[x]  $\leftarrow$  map[1][chosen_frontier_idx]
  goal_coords[y]  $\leftarrow$  map[0][chosen_frontier_idx]
  return goal_coords
end procedure

```

The argument of the method is a map in format of numpy array. This format was chosen because of the quickness of the library and because of the numpy interface convenience for working with big arrays. At the start, algorithm searches for indexes of unknown cells in the map 2D array. `Slam_toolbox` is publishing map of the area with only three values present in the map 2D array: "-1" for unknown cells, "0" for known cells without obstacles and "100" for known cells with obstacles within them. If algorithm cannot find any unknown frontiers, it writes Error in the console. The next step of this method is using the function, which chooses a number in range from 0 to the biggest size of the indexes array. The smallest size of array is 2, because it finds indexes of unknown cells from array and writes it in two rows, first row is unknown indices along y-axis and second row is unknown indices along x-axis. This is why `uniform_distribution_random()` function chooses values from range from 0 to the biggest size of unknown indices array. After this, algorithm writes down obtained indices to the variable `goal_coords` and returns it as the frontier detection result.

8.2 Yamauchi's Frontier Detector class

This is class that implements the second chosen method for frontier detection. This class was described in the chapter 5.1. This method was developed by Brian Yamauchi in 1997 [20]. The working principle of this algorithm goes like this: when map is fetched, algorithm processes it with edge-detection algorithm obtaining edges between known and unknown areas of the map, those boundaries between known and unknown areas are frontiers [20]. All obtained boundaries are sorted by their size and controlled for meeting the requirements to be classified as frontiers. Boundary needs to be bigger than the minimal size of the frontier, established by the user. After that, frontiers are sorted by their closeness to the current position of the robot. The robot will visit the closest frontier, other frontiers will be maintained to be used later, if the later scans do not find any new frontiers.

I will begin the explanation of this class with UML class diagram, containing all method and attributes of DFDdetector.

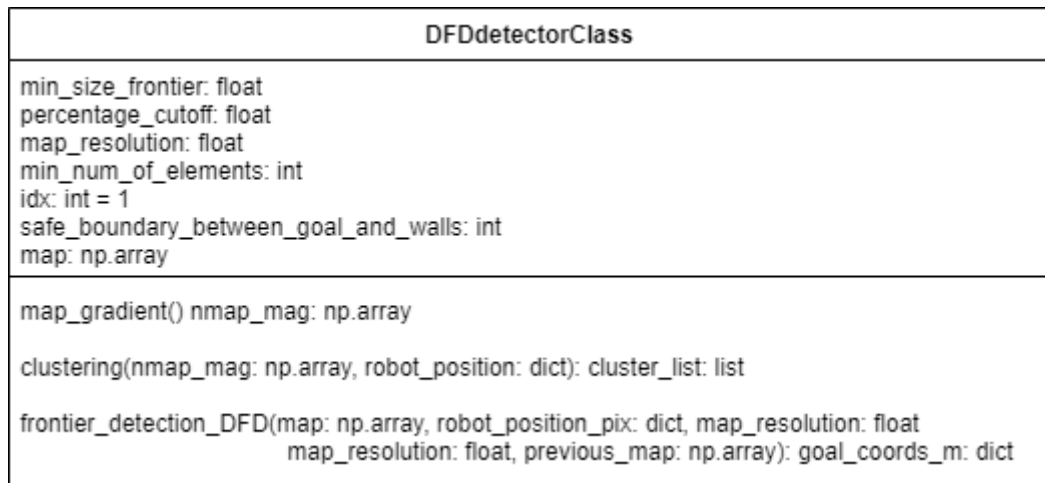


Figure 8.7: diagram of DFDdetectorClass, implementing Yamauchi's Frontier Detection

As it can be seen from fig 8.7. DFDdetectorClass has 6 attributes. The explanation of UML class diagram will begin with them.

Min_size_frontier attribute stores user defined minimal size of frontier in [m]. All found boundaries will be checked for exceeding the minimal size. Smaller ones will be erased and the bigger will be visited or maintained.

Percentage_cutoff attribute defines the border for map_gradient() method for DFDdetector class. It will be fully explained later, but just the important details will be explained now. When the map is fetched, it needs to be processed with map gradient algorithm, after which the estimate of magnitude of the gradient on the whole map will be received. But magnitude does not always signal that there is a border in that place

on the map, so only the highest gradient values need to be left and they would signal that there are borders between the known and the unknown in the place of high gradient magnitude. Percentage cutoff defines how much will be erased from gradient magnitude map and how much will be left.

Map_resolution stores the current resolution of the map in the [m/cell] units.

Min_number_of_elements attribute is derived from `min_size_frontier` attribute, stored in [m] and `map_resolution` attribute, stored in [m/cell]. This attribute is just `min_size_frontier` divided by `map_resolution` to more comfortably control frontiers on matching the requirements

idx is global index of frontier the robot is going to visit next. It initialises with value 1.

Safe_boundary_between_goal_and_walls attribute defines, how many cells should lie between the goal and the closest obstacle.

Map attribute is map obtained from turtlebot under topic `"/map"`. It is stored as a map attribute only to be global variable for `DFDdetector` class.

The explanation of the UML `DFDdetector` class diagram will continue with the description of the methods of the class. The main method of this class is `"frontier_detection_DFD()"`. This method contains and uses all of the other methods of this class and the object dependent on this class within itself.

8.2.1 Frontier detectionDFDmethod

`Frontier detectionDFDmethod` is the main method of `DFDdetectorClass`, and incorporates all other methods of this class. The description of this method will start with its algorithm, the pseudo code of the algorithm can be seen on the "Algorithm 5".

Algorithm begins by writing down relevant metadata in the `DFDdetectorClass` attributes to be accessible for all function, i.e. to be global for class. It starts with fetching actual map and map resolution. After that, the other methods of `DFDdetectorClass` enter. Algorithm continues with obtaining the estimate of gradient magnitude from method `"DFDdetector.map_gradient()"`. Later on, algorithm obtains the cluster list, containing all the clusters, from estimate of gradient magnitude of the map. It writes it to the variable `"cluster_list"` from method `"clustering()"` of the `DFDdetectorClass`. The working principle of those methods will be explained in the following chapters. After that, the empty list `"distance_from_centroid"` is allocated. This list will contain the distance from current position of robot to each centroid of clusters from `"cluster_list"` variable.

Following for loop essentially controls cluster centroid's neighbourhood. If cluster centroid has obstacles in its neighbourhood, then the whole cluster is being removed from `"cluster_list"`. Alternatively, if cluster's centroid does not have obstacles in its

neighbourhood, the distance between centroid and the robot will be calculated and added to the "distance_from_centroid" list.

Algorithm 5 Frontier detectionDFDalgorithm

```

procedure FRONTIER_DETECTION_DFD(map, robot_position_pix,
                                     map_resolution, previous_map)

  self.map ← map
  self.resolution ← map_resolution
  gradient ← self.map_gradient()
  cluster_list ← self.clustering(gradient, robot_position_pix)
  distance_from_centroid ← [ ]
  for each cluster ∈ cluster_list do:
    c_j = cluster.cluster_centroid[j]
    c_i = cluster.cluster_centroid[i]
    centroid_neighbourhood = map[c_j − self.safe_boundary :
                                   c_j + self.safe_boundary,
                                   c_i − self.safe_boundary :
                                   c_i + self.safe_boundary]
    if all cells in centroid_neighbourhood ≠ 100 do:
      distance_from_centroid.append(
         $\sqrt{(\text{robot\_position\_pix}[i] - c_i)^2 + (\text{robot\_position\_pix}[j] - c_j)^2}$ )
    else:
      cluster_list.remove(cluster)
    end_if
  end_for
  distance_from_centroid_sorted ← distance_from_centroid.sort()
  if previous_map = map do:
    curr_distance ← distance_from_centroid_sorted[self.idx]
    curr_distance_idx ← distance_from_centroid.index(curr_distance)
    goal_coords ← cluster_list[curr_distance_idx].cluster_centroid
    self.idx ← self.idx + 1
  else:
    min_distance ← distance_from_centroid_sorted[0]
    min_distance_idx ← distance_from_centroid.index(min_distance)
    goal_coords ← cluster_list[min_distance_idx].cluster_centroid
    self.idx ← 1
  end_if
  return goal_coords
end procedure

```

Next "distance_from_centroid" list is being sorted and written to the "distance_from_centroid_sorted" list, which leaves original list unsorted and creates its sorted copy.

Next condition is extremely important for this method. Condition "**if previous_map = map**" evaluates if robot is stuck to the same frontier or not. This derives from the statement that the robot has to go to the closest discovered frontier. The problem is if the robot arrived to the closest frontier and would not have discovered any new parts

of the map, robot would be stuck to this same frontier it arrived to. Because the map did not change, neither did the list of frontiers, so without this condition robot will be forever stuck at this last closest frontier and will not go any further. What this condition essentially does is it sends robot to the more distant discovered frontiers until it discovers new parts of the map. After it discovers new parts of the map, the robot will continue to go to the closest frontier again.

Algorithm ends with returning goal coordinates in format of row and column of corresponding cell from 2D array "map". This frontier can be closest to the robot or can be more distant. Its distance is regulated by attribute of DFDdetectorClass "idx". If the map does not change, then cluster with index 1 from cluster_list is chosen to be the goal, and index is incremented with 1. If map still does not change, then cluster with index 2 from cluster_list is chosen to be the goal, and so on. If the map finally changes, attribute "idx" returns to its original value of 1.

At the end of this chapter the appeared problems and my solution for them will be listed. The first problem was mentioned above, it is the problem with the robot sticking to the closest frontier if new parts of map were not discovered in the last loop iteration. My solution of the issue was to compare the current map with previous map and it made a significant difference in efficiency of this algorithm. Before this solution was implemented, the robot was stuck after two or three iterations. After implementing it, the robot was able to map the whole room. The other minor problem for me was to sort separate list of cluster objects, written in "cluster_list" by the value of their distance to the robot's position, written in the other list "distance_from_centroid". During the for loop, two lists were formed in such a way that the cluster[i] in the "cluster_list" corresponded to its distance[i] in the "distance_from_centroid". But if just the "distance_from_centroid" list was sorted, the correspondence between two list would disappear. So my solution was to make new "distance_from_centroid_sorted" list, which would be sorted from the smallest values to the biggest ones and then search index of corresponding distance and cluster by value in the original "distance_from_centroid" list.

8.2.2 Map gradient method

Map gradient method is important method for Yamauchi's Frontier Detection. From Yamauchi's paper it can be read, that frontiers are obtained by the application of computer vision like edge-detection method [20]. And the well-known technique for edge detection, that was introduced by Güner S.Robinson in 1977 proposing to apply 3x3 masks on the image to later obtain gradient, which will indicate image's edges [28]. This method "map_gradient()" implements part of the method introduced in the source [28], specifically implementing the part of obtaining gradient magnitude by applying 3x3 pixel mask

on the image. The kernel this method is going to use was proposed by Irwin Sobel [29]. The meaning of this kernel will be explained later. Now the explanation will continue with pseudo code for this method ("Algorithm 6").

Algorithm 6 Map gradient extraction algorithm

```

procedure MAP_GRADIENT
  map_I_copy ← self.map
  nmap_mag ← np.zeros(map_I_copy.shape)
  map_I_copy ← np.where(map_I_copy = 100, 50)
  map_I_copy ← np.where(map_I_copy = -1, 100)

   $Gx \leftarrow \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$ 

   $Gy \leftarrow \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$ 

  for each pixel ∈ map_I_copy do:
    i ← pixel[i]
    j ← pixel[j]
    submap ← map_I_copy[j - 1 : j + 1, i - 1, i + 1]
    ncell_I_x ←  $\sum_{j=0, i=0}^{j=2, i=2} (\textit{submap} \times Gx)$ 
    ncell_I_y ←  $\sum_{j=0, i=0}^{j=2, i=2} (\textit{submap} \times Gy)$ 
    nmap_mag[j][i] ←  $\sqrt{\textit{ncell\_I\_x}^2 + \textit{ncell\_I\_y}^2}$ 
  end for
  max_gradient = np.amax(nmap_mag)
  nmap_mag = np.where(nmap_mag >= self.percentage_cutoff * max_gradient, 255, 0)
  return nmap_mag
end procedure

```

The algorithm begins with copying current map obtained from turtlebot. This is why the "Frontier detection DFD" method was writing a map to the attribute of the DFDdetectorClass, as after that any method of the class can get access to it, since it became global variable for the class. And no values needs to be specifically passed to the function. It continues by allocating 2D array "*nmap_mag*" of the same size as 2D array "*map*".

Next operation needs to be explained more in-detail. Here the values on the map were changed. It was done to get higher contrast between unknown and known "without obstacles" cells, and as a consequence to get bigger gradient between areas with unknown cells and areas known cells without obstacles.

Next Sobel's kernel is being written to the "*Gx*" and "*Gy*" variables. Sobel's kernel works that way, by applying 3x3 mask on the neighbourhood of every pixel, estimate of

the magnitude of the gradient can be calculated. By applying " G_x ", algorithm calculates gradient along x-axis i.e. derivative along x-axis. And by applying " G_y ", algorithm calculates gradient along y-axis. And by calculating the length of the sum of vectors of gradient along x-axis and y-axis, the estimate of the magnitude of the gradient is calculated. That estimate shows where gradient is the biggest, and it can be assumed that the edges are there on the map array [29].

Following for cycle applies " G_x ", " G_y " mask on the 3x3 neighbourhood of every pixel and writes an estimate of the magnitude of gradient in the " $nmap_mag$ " 2D array for every cell correspondingly.

After that the maximum gradient is found and " $nmap_mag$ " image's values are assigned as 255, if they are bigger than " $max_gradient * self.percent_age_cutoff$ ", otherwise they are assigned 0. That makes output magnitude image " $nmap_mag$ " a binary image with values 0 or 255.

The last operation is returning of the map of the estimate of the magnitude of the gradient. The results of applying Sobel's kernel on image for edge detection can be seen on fig 8.8.

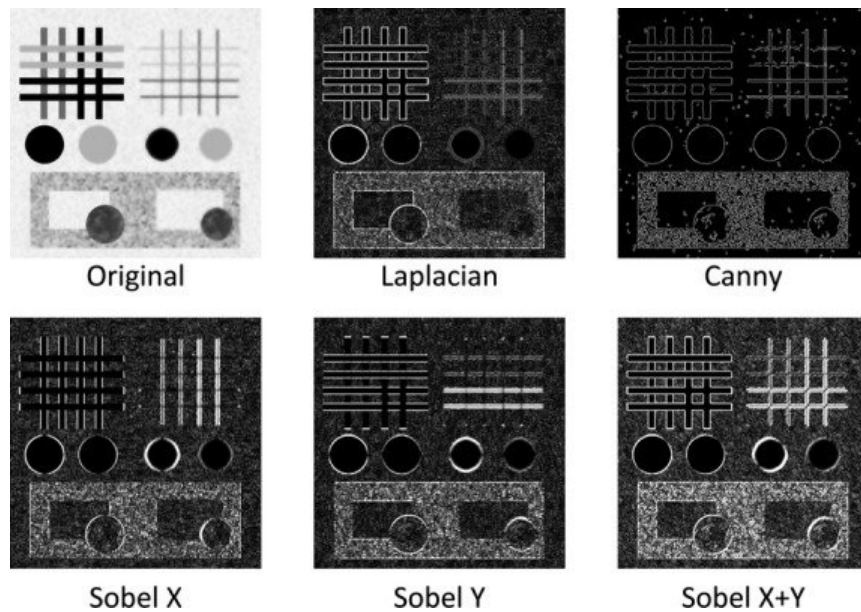


Figure 8.8: Example of applying Sobel's kernel on the image for edge detection, [30]

8.2.3 Clustering method

After a solid image containing the magnitude of the gradient is obtained, it needs to be clustered, so the cells with highest gradient i.e. with value "255" would be divided by groups. Pixels are grouped together if they have common borders. Simply speaking, pixels with value "255" which has no space between themselves are grouped in one cluster.

Clustering was implemented with two methods: Breadth-first search or BFS algorithm and connected-component labeling algorithm or CCL.

Breadth-first search clustering

BFS is algorithm for searching the path in a tree data structure. The input data for the algorithm are: starting point in the graph and condition under which the search is over. The algorithm begins its search from the neighbours of the starting point in the graph, when it comes to the neighbours of the neighbours of the starting point on the graph and so on until the search conditions are met [31].

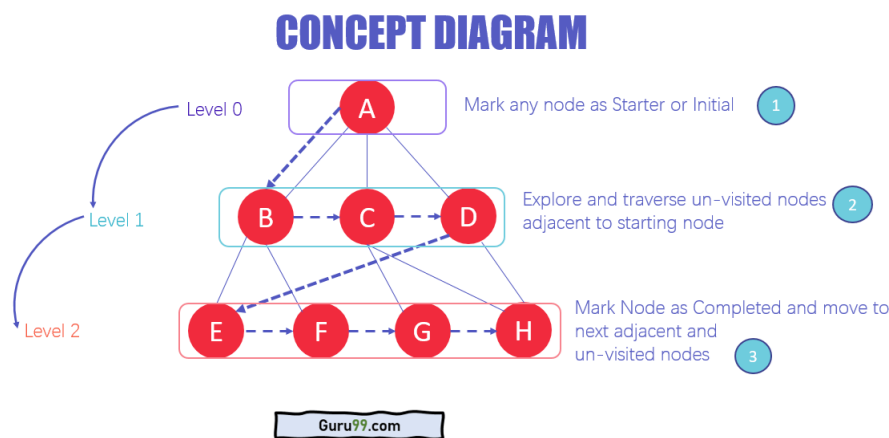


Figure 8.9: Breadth-First search, [32]

Since the magnitude map is a binary image, it can be clustered in the same way. My implementation of BFS was recursive. The pseudo code of my implementation follows on the "Algorithm 8". This algorithm is called with the first found cell on "*nmap_mag*". The row "*j*" and column "*i*" is passed to the function as well as "*nmap_mag*" binary image copy and a cluster object with class "cluster". Class "cluster" is class, developed by me to store indexes of all the cells from each cluster, obtained from "*nmap_mag*" binary image. It will be fully described in later chapters. After all of the arguments are passed to the function the search for cluster cells begins. The algorithm checks every neighbour of starting cell, if it has value "255", if so it adds this pixel to the cluster, checks pixel for being an obstacle on map, marks cell as visited and recursively calls itself, but with new row and column arguments, cluster object and "*nmap_mag*" binary image remains, so new cells could be assigned to the same cluster and be marked as visited on the same "*nmap_mag*" binary image.

On the "Algorithm 7." you can see the usage of recursive BFS in function "ClusterWithRecursiveBFS". The result of this function would be a "*cluster_list*", which is

list, containing all "Cluster" class objects, and they themselves contain all cells of each cluster.

Algorithm 7 Clustering with recursive BFS

```

procedure CLUSTERING(nmap_mag)
  nmap_mag_copy ← nmap_mag
  frontier_indices ← np.where(nmap_mag = 255)
  while frontier_indices are not empty do:
    starting_point = ["j" : frontier_indices[j][0],
                      "i" : frontier_indices[i][0]]
    nmap_mag_copy[starting_point] = 0
    new_cluster = Cluster(starting_point)
    self.RecursiveBFS(nmap_mag_copy, starting_point[j],
                     starting_point[i], new_cluster)
    cluster_list.append(new_cluster)
  end_while
  frontier_indices ← np.where(nmap_mag = 255)
  return cluster_list
end procedure

```

Algorithm 8 Recursive BFS

```

procedure CHECK_NEIGHBOURS(nmap_mag, j, i, cluster)
  if nmap_mag[j][i - 1] = 255 do:
    cluster.add_pixel(j, i - 1)
    if map[j][i - 1] = 100 do:
      cluster.has_walls_inside_flag ← True
    end_if
    nmap_mag[j][i - 1] = 0
    self.RecursiveBFS(nmap_mag, j, i - 1, cluster)
  end_if

  if nmap_mag[j - 1][i] = 255 do:
    cluster.add_pixel(j - 1, i)
    if map[j - 1][i] = 100 do:
      cluster.has_walls_inside_flag ← True
    end_if
    nmap_mag[j - 1][i] = 0
    self.RecursiveBFS(nmap_mag, j - 1, i, cluster)
  end_if

  ...
  And so on for all the neighbours of nmap_mag[i][j]
  ...
  return True
end procedure

```

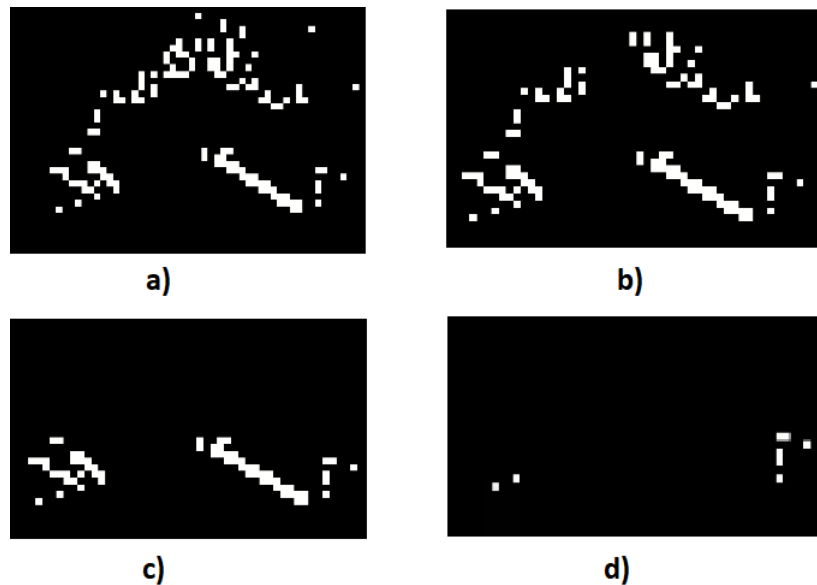


Figure 8.10: Example of clustering with my implementation of recursive BFS, clusters are found, written in the cluster objects and deleted from gradient magnitude map:

- a) Obtained map of estimate of the magnitude of the gradient, not clustered yet
- b) Magnitude map after several iterations of recursive BFS
- c) Magnitude map after more iterations of recursive BFS
- d) Magnitude map after more iterations of recursive BFS

Connected-component labeling clustering

CCL based image clustering was firstly introduced by H. Samet in 1988 [33]. This image processing technique can uniquely label connected components on binary image. Input of CCL is binary image and output is image, where different objects, including background, are labeled uniquely. It has common traits with map gradient method by Sobel [29]. In this method two pass scan of image happens. Firstly there is scan, applying the kernel on the image. You can see the kernel on the fig 8.11. , the 4-connectivity kernel was applied in my code. After first pass, the temporary labels and relation table between labels are calculated. The second pass applies relation table on the image. The best way to understand this algorithm is by looking at it: [34].

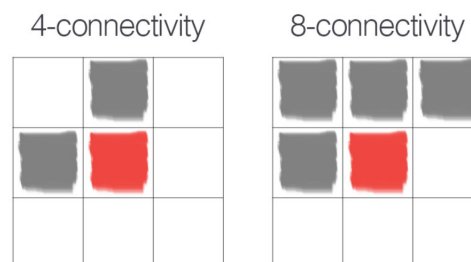


FIGURE 8: EXAMPLES OF 4-CONNECTIVITY (LEFT) AND 8-CONNECTIVITY (RIGHT) FOR CONNECTED-COMPONENT LABELING.

Figure 8.11: 4 and 8 connectivity kernels, used in CCL, [35]

Labeled image converts from binary image (a 2D array, which has only 2 values in it) to a labeled image (a 2D array which has every pixel, labeled as index of object, to which this pixel belongs). The example is on the fig 8.12

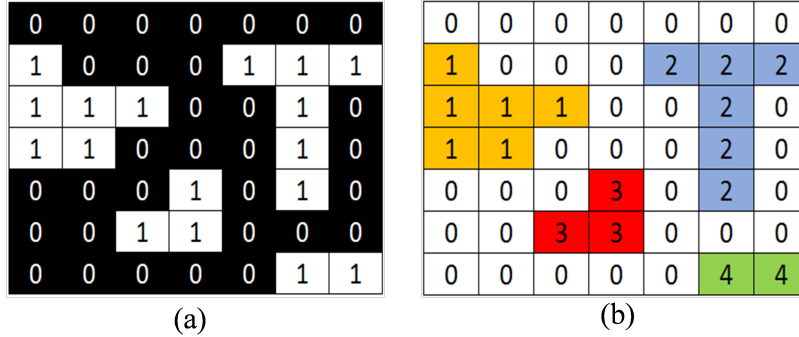


Figure 8.12: a) binary image b) CCL processed image, [36]

Now the pseudocode "*clustering()*" function with CCL algorithm will follow.

Algorithm 9 Clustering with CCL

```

procedure CLUSTERING(nmap_mag)
  cluster_list ← [ ]
  nmap_mag_copy ← nmap_mag
  labeled_nmap_mag_copy = skimage.measure.label(nmap_mag_copy)
  n_clusters = np.amax(labeled_nmap_mag_copy)
  for i = 1 to n_clusters + 1 do:
    cluster_indices = np.where(labeled_nmap_mag_copy == i)
    if np.any(self.map[cluster_indices]) = 100 do:
      nmap_mag_copy[cluster_indices] = 0
    else:
      new_cluster = Cluster()
      for cell ∈ cluster_indices do:
        new_cluster.add_pixel(cell)
      end_for
      cluster_list.append(new_cluster)
    end_if
  end_for
  return cluster_list
end procedure

```

The algorithm starts with allocating an empty list "*cluster_list*" for storing cluster objects. After that algorithm makes a copy of magnitude map "*nmap_mag*". After that the labeled copy of the magnitude map is made with help of *skimage* library throughout an CCL algorithm. Because of CCL labels objects uniquely, incrementing id label every time the new object is found on the image, as it can be seen from fig 8.12., the number of objects on the image can be found as id of the last object. This is why the "*n_clusters*"

- number of clusters variable is found as biggest value on labeled image. Following for loop checks all clusters, loop is starting with 1, because 0-th object on the labeled image is background. Firstly it finds all indexes of current cluster. Then checks if relevant map has obstacles on the places, where current cluster is situated. If cluster has obstacles then algorithm erases it. If algorithm does not find obstacles in the cluster it initialises new Cluster class object, which will store all the cells of current cluster and adds all of the indices from "*cluster_indices*" to the "*new_cluster*" object. After that algorithm adds "*new_cluster*" to the "*cluster_list*" and returns it.

chosen clustering algorithm

The recursive BFS algorithm, implemented by me performed well, was fully working and capable of clustering (the example of its work can be found on the fig 8.10.) and I was happy with it, but later I discovered, that BFS algorithm is main part of another frontier detection method WFD, which was described in chapter 5.3. Because of comparison nature of this thesis, the usage of this algorithm was not able and I decided to use CCL algorithm instead.

problems, implementing this method and their solutions

The most of the problems occurred, while developing recursive BFS. It was not an typical solution for the clustering, but my implementation showed its reliability and good performance. Unfortunately due to the comparing essence of this thesis I could not use this ready solution in the YFD frontier detecting method as it was part of the WFD method. But for real usage of this algorithm this could be done easily and would be a good merge of two algorithms.

8.2.4 Cluster class

Cluster class is the class dependant from DFDdetectorClass as it can be seen from fig. 8.1. Essentially its datatype class and DFDdetectorClass uses it to store data about cluster objects, which are found from gradient magnitude map in the "*clustering()*" method of the DFDdetectorClass. I will begin description of this class from Cluster UML class diagram. It can be seen on fig 8.13.

Cluster class has 7 attributes. **starting_point** attribute is the starting point, from which the BFS was initiated.

list_of_cells is 2D array, which contains j indexes of the cluster cells in the 0th row and i indexes on the 1st row.

map_id is the string, naming the map on which cluster is situated.

cluster_id is the int value, which signals, in what order this cluster was found on the map with `map_id`.

number_of_elements is how many cells this cluster has.

cluster_centroid is mean coordinate of all cells, which are contained in the `list_of_cells` attribute.

has_walls_inside_flag is boolean value, which signals if cells with obstacles were found within the cluster.

Speaking about the methods of this class. This class has 3 methods and its names pretty much speaks for themselves. **Calculate_number_of_elements()** fetches length of the "`list_of_cells`" attribute of the class.

Calculate_centroid_method() calculates mean `j` and `i` coordinates from all coordinates, contained in "`list_of_cells`" attribute.

add_pixel() method appends new `j,i` coordinates to the "`list_of_cells`" attribute.

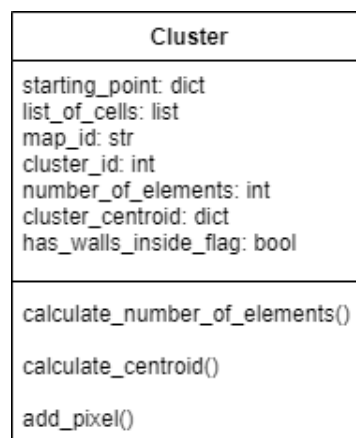


Figure 8.13: UML class diagram of Cluster class

This datatype was developed by me as the solution for storing all the cells of each frontier. Storing those cells just in list would not be comfortable for calculations and iteration over list of frontiers. This solution showed itself as very reliable and comfortable.

8.3 Fast Frontier Detector class

This is class, that implements Fast Frontier Finding algorithm or FFD for short. This algorithm was described in chapters 5.2. and 6.3. This algorithm was introduced by M.Keidar and G.A.Kaminka in 2012 [21]. So it is relatively new algorithm. The principle of this algorithm was mentioned in above chapters, but the essential working mechanisms will be explained now. This algorithm assumes, that it is not necessary to scan all the map to obtain frontiers, it is working on principle, that all frontiers on current iteration

of the map are situated in the current scan of the LDS and can be obtained by processing scan message from the sensor [21]. This algorithm starts with mapping the distances, obtained from LDS to the current iteration on the map, then it proceeds with connecting the separate laser scans points with lines. After all lines are connected algorithm obtains the the borders between known space and unknown space, which is frontiers. After that frontiers need to be singled out from border. Next step for algorithm is to check every point in the border and if it has unknown neighbouring cells and does not have cells, containing the obstacles, then it is assigned to the frontier. The visualisation of this method can be seen on figures: 6.3., 6.4. and 6.5.

Next follows the UML class diagram of the FFDdetectorClass on the fig 8.14.



Figure 8.14: UML class diagram of FFDdetectorClass class

The description of this class will begin from the attributes of it. First attribute **laser_readings_polar** is numpy array for storing actual laser readings message, which comes from turtlebot under topic `"/scan"`. The message is passed to the class, when the laser readings need to be processed and is immediately assigned to this attribute and became global variable for the class, so all functions could get access to it. It is called that way because laser readings scans are obtained in the polar coordinates in robot coordinates `"base_link"` coordinate frame (more on fig 8.5.), and to map those readings they are need to be transformed to the origin coordinate frame.

laser_readings_car_orig_pix is the attribute of the robot, which stores laser

readings, converted to the Cartesian format and transformed to the origin coordinate frame. This is also the global variable for FFDdetectorClass, so every function could get access to those data.

Map is the current iteration of map, **map_resolution** is current resolution of the map, **pose_of_robot** is current pose of the robot, **yaw_rot_of_robot** is current yaw rotation of robot to the map coordinate frame.

lines_list is the list for containing lines, connecting the laser readings

frontier_list is the list for containing frontiers, sigled out from lines, written in the lines_list.

frontier_idx is the index of frontier from global pool of frontiers. FFD is processing only part of the map, this is why frontiers from other part of the map need to be stored. My code stores frontiers from other part of the map in the global_pool_of_frontiers attribute of FFDdetectorClass. And when current scan cannot find any new frontiers, frontier from global_pool_of_frontiers attribute with frontier_idx gets extracted and acts as goal.

debug_flag is Boolean value, that signals, that experiment is being conducted and some values of exploration mission need to be recorded and saved later.

Map_origin_position is attribute, that stores the position of origin coordinate frame, obtained from "/map" message from slam_toolbox.

global_pool_of_frontiers is list that stores found frontiers, to extract them later, check their relevancy and send them as goal.

num_of_iter_map_unchnged is int value, that shows how many times map remained the same during last iterations.

Moving on to the methods of the FFDdetectorClass. The main method here is do_ffd(), so the explanation of principle of the working of the class will continue with it.

8.3.1 Do FFD method

Do FFD method is the main method of FFDdetectorClass it contains and uses all other methods of this class. The input of this method as it can be seen from fig 8.14 is map, map metadata and all metadata about robot's position, output is goal coordinates for turtlebot in origin coordinate frame. Next goes the algorithm of this function in pseudo code format on the "Algorithm 10". The algorithm starts with fetching actual data about map and robot's position and writes it from arguments of the function to the attributes of the FFDdetectorClass. After that class writes the actual laser readings in polar format in the "base_link" coordinate frame from the argument to the "laser_readings_polar" attribute of the class.

Next if conditions checks for how many iterations the map remained the same and writes that number to the "*num_of_iter_map_unchnged*" variable.

Algorithm 10 Frontier detection FFD algorithm

```

procedure DO_FFD(pos_of_robot, yaw_rot_of_robot, laser_readings_polar,
                 map, previous_map, map_resolution, map_origin_position)
  self.pos_of_robot ← pos_of_robot
  self.yaw_rot_of_robot ← yaw_rot_of_robot
  self.map ← map
  self.map_res ← map_res
  self.map_origin_pos ← map_origin_position
  self.laser_readings_polar ← laser_readings_polar
  if self.map = previous_map do:
    self.num_of_iter_map_unchnged + = 1
  else:
    self.num_of_iter_map_unchnged ← 0
  end if
  self.laser_readings_car_orig ← self.map_laser_readings(
                                self.laser_readings_polar)
  self.lines_list ← self.get_lines_from_laser_readings(
                    self.laser_readings_car_origin)
  self.frontier_list ← self.find_frontiers_in_lines(self.lines_list)
  for frontier ∈ self.frontier_list do:
    frontier.calculate_centroid()
    frontier.calculate_num_of_elements()
  end for
  self.frontier_list.sort_by(x.num_of_elements)
  if self.frontier_list is not empty and
    self.num_of_iter_map_unchnged < 2 do:
    goal_coords ← self.frontier_list[0].centroid
    self.frontier_list.remove_by_index(0)
    for frontier ∈ self.frontier_list do:
      frontier.centroid ← self.transf_map2odom_and_back("orig", "map", frontier.centroid)
      frontier.delete_all_points()
      self.global_pool_of_frontiers.append(frontier)
    end for
  else if self.frontier_list is empty or
    self.num_of_iter_map_unchnged ≥ 2 do:
    obst_flag ← False
    unkn_flag ← False
    while self.global_pool_of_frontiers is not empty do:
      goal_coords ← self.global_pool_of_frontiers[0]
      goal_coords ← self.transf_map2odom_and_back("map", "origin", goal_coords)
      self.global_pool_of_frontiers.remove_by_index(0)
      [unkn_flag, obst_flag] ← self.check_neighbours(goal_coords)
      if unkn_flag and not obst_flag do:
        break while loop
      else:
        goal_coords ← 0
      end if
    end while
  else:
    rospy.signal_shutdown()
  end if
  return goal_coords
end procedure

```

▷ Exploration mission is going normal condition

▷ FFD method could not find any frontiers or robot is stuck

▷ the exploration mission is over

After that follows the processing of laser readings by "*map_laser_readings*()" method of the FFDdetectorClass. Method transforms the laser readings from polar base_link coordinate frame to Cartesian origin map coordinate frame. Then follows the processing of mapped laser readings by "*get_lines_from_laser_readings*()". Method connects mapped laser readings with line and returns *line_list*, containing all that lines. Follows the procedure of processing the lines by method "*find_frontiers_in_lines*()". Method analyses every point in every line in "*lines_list*" and checks if point has unknown cells

around it and if point has cells, that contains obstacles around it, method chooses only points, that have unknown points and does not have obstacles around it, after that it assigns points to the frontier.

Following for loop calculates centroid and number of elements for every frontier in "*frontier_list*". After that "*frontier_list*" sorts itself by the number of elements, each frontier contains. The resulting list has the biggest frontier on the place with index "0".

Following **if condition** is very important to the FFD detection method, this condition signals **if** exploration mission is going normal, **else if** FFD method could not find any frontiers in the current iteration of the map or robot is stuck and map remains unchanged for more than 2 iterations, **else** the exploration mission is over and there are no frontiers left on the map. I will describe all three possibilities more in detail.

- **Exploration mission is going normal** - if exploration mission is going normal , then "*do_FFD()*" method will just send the biggest frontier from "*self.frontiers_list*" attribute and will write the remaining frontiers from the "*self.frontiers_list*" list to the list "*self.global_pool_of_frontiers*", but before writing them to the new list, algorithm will transform the position of the frontier from origin coordinate frame to the map coordinate frame, so the frontiers could be extracted later, after map will change.
- **FFD method could not find any frontiers or robot is stuck** - "*self.frontier_list* is empty" condition signals that FFD method could not find any frontiers on the current map and "*self.num_of_iter_map_unchnged* \geq 2" condition signals, that robot is stuck near the biggest frontier and could not find any frontier, that is bigger than current. In both cases robot will go to the second biggest frontier. As map did not change and frontiers in the last "Exploration mission is going normal" scenario were written in the "*global_pool_of_frontiers*" variable, so the second biggest frontier will be written in the "*global_pool_of_frontiers*" list under "0" index. Before sending the frontier as goal, algorithm will extract it and transform back from map coordinate frame to the origin coordinate frame with help of function "*goal_coords* \leftarrow *self.transf_map2odom_and_back("map", "origin", goal_coords)*". While loop in that scenario serves as check of the actuality of the frontier. If "*goal_coords*" still have unknown cells around it and does not have obstacles, then "*goal_coords*" are remain actual and will be sent as the goal.
- **The exploration mission is over** - the algorithm simply sends the signal to ROS to shut down the loop.

The last step for algorithm is to return "*goal_coordinates*" and finish its work. Next

the problems, I have encountered during the developing of this algorithm, and their solutions will follow.

I guess the biggest challenge was to develop an algorithm, that maintains already founded frontiers and extracts them, when FFD method cannot find any new frontiers. Because of DFD method always scans the whole map, the same unvisited frontiers can be found at any iteration and practically do not require maintaining them. (this is of course assuming smaller or middle-sized maps as in my case, because with big maps frontiers need to be stored to save computational power). And in FFD method algorithm processes only that part of the map, on which the robot is situated on, so found frontiers should be saved. And my solution of using code with three scenarios cases with first "good" scenario just sending the biggest currently found frontier, "not so good" scenario with using global pool of frontiers list to save them and extract them within the right coordinate frame (because map is constantly changing and origin coordinate frame changes its location but map coordinate frame does not) was fully functional and showed good performance with FFD method.

8.3.2 Map laser readings method

This is method of FFDdetectorClass serves a purpose to remap laser readings from polar coordinate system based in the origin of base_link coordinate frame to the cartesian coordinate system based in the origin coordinate frame (more on coordinate frames fig 8.5.). The example of work of map laser readings method can be seen on fig 8.15.

The description of this method will continue with pseudo code of this algorithm on the "Algorithm 11". The algorithm starts off with allocating of the variable "*angle_of_reading*" for storing of the current angle. There are 360 readings in "*laser_readings_polar*" - one reading for each angle, containing the distance from robot to the obstacle, laying at the certain angle from the robot. Next two arrays with the size of "*laser_reading_polar*" are allocated to store remapped readings from polar coordinate system to Cartesian coordinate system.

Following for loop checks every laser reading for being relevant and remaps it from polar coordinate system to Cartesian coordinate system if they are relevant, if they are not it assigns them with zeros. If laser reading are assigned with $float(\infty)$ value, then it means, that LDS does not have enough range to capture the obstacle laying at that angle i.e. that reading is irrelevant and should be deleted. After the end of the for loop algorithm searches for the irrelevant reading and deletes them.

Next for loop transfers laser readings from base_link coordinate frame to the origin coordinate frame. It creates vector "*LR*", containing the laser readings in the base_link frame, creates transformation matrix "*TM*" with the current position and yaw rotation

angle of the robot in the origin coordinate frame and multiplies them to get the laser readings in the origin coordinate frame. After that it leaves only unique laser readings in the origin coordinate frame and returns "*laser_readings_car_origin*" array, containing remapped to the origin frame laser readings.

Algorithm 11 Map laser readings algorithm

```

procedure MAP_LASER_READINGS(laser_readings_polar)
  angle_of_reading  $\leftarrow$  0
  laser_readings_car_x  $\leftarrow$  np.zeros(laser_readings_polar.shape)
  laser_readings_car_y  $\leftarrow$  np.zeros(laser_readings_polar.shape)
  for laser_reading  $\in$  laser_readings_polar do:            $\triangleright$  From polar to cartesian
    if laser_reading  $\neq \infty$  do:
      laser_readings_car_x  $\leftarrow$  laser_reading  $\cdot$  cos(angle_of_reading)
      laser_readings_car_y  $\leftarrow$  laser_reading  $\cdot$  sin(angle_of_reading)
    else:
      laser_readings_car_x  $\leftarrow$  0
      laser_readings_car_y  $\leftarrow$  0
    end_if
    angle_of_reading  $\leftarrow$  angle_of_reading + 1
  end_for
  inf_idxs  $\leftarrow$  np.where(np.logical_and(laser_readings_car_x = 0,
                                     laser_readings_car_y = 0))
  laser_readings_car_x.delete(inf_idx)
  laser_readings_car_y.delete(inf_idx)
  laser_readings_car  $\leftarrow$  np.vstack(laser_readings_car_x,
                                     laser_readings_car_y)
  laser_readings_car_orig  $\leftarrow$  np.zeros(laser_readings_car.shape)
  for laser_reading  $\in$  laser_readings_car do:            $\triangleright$  From base_link to origin
    x_laser  $\leftarrow$  laser_reading[x]
    y_laser  $\leftarrow$  laser_reading[y]
     $\gamma_{robot\_orig}$   $\leftarrow$  self.yaw_rot_of_robot
    x_robot_orig  $\leftarrow$  self.pos_of_robot[x]
    y_robot_orig  $\leftarrow$  self.pos_of_robot[y]

     $LR \leftarrow \begin{bmatrix} x_{laser} \\ y_{laser} \\ 1 \end{bmatrix}$ 

     $TM \leftarrow \begin{bmatrix} \cos(\gamma_{robot\_orig}) & -\sin(\gamma_{robot\_orig}) & x_{robot\_orig} \\ \sin(\gamma_{robot\_orig}) & \cos(\gamma_{robot\_orig}) & y_{robot\_orig} \\ 0 & 0 & 1 \end{bmatrix}$ 

    origin_map_frame_coords  $\leftarrow$  TM  $\times$  LR
    laser_readings_car_orig[laser_reading]  $\leftarrow$  origin_map_frame_coords
  end_for
  np.unique(laser_readings_car_orig)
  return laser_readings_car_orig
end procedure

```

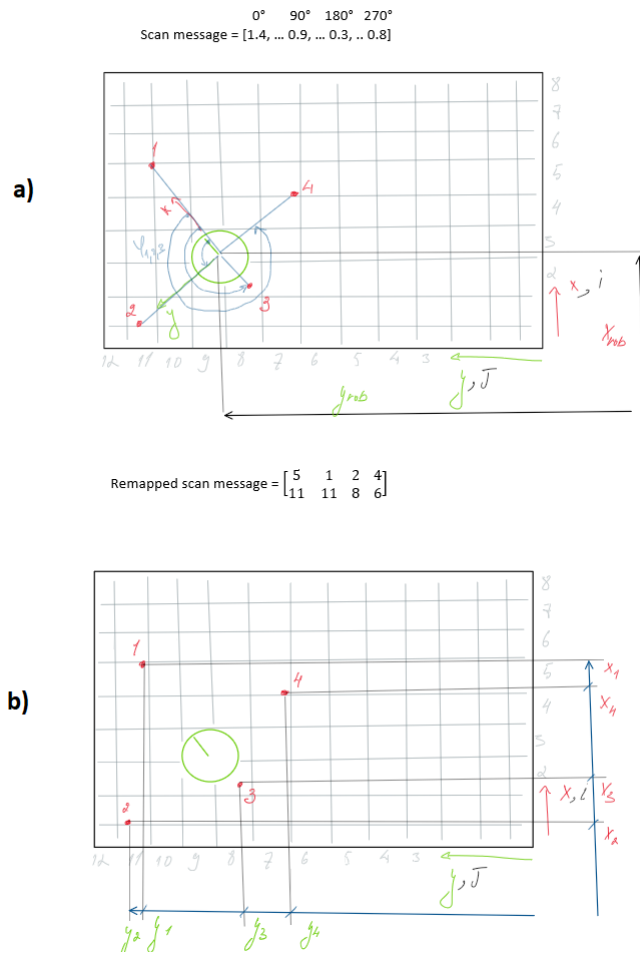


Figure 8.15: The process of remapping scan readings:

- a) Original scan message
- b) Remapped scan message

Next follows, the problems I have encountered and their solutions. The biggest problem with mapping the laser readings was: how to get robot's pose, yaw in map coordinate frame and how to transform all 360 readings without using the tf2 toolbox. Despite the toolbox is extremely useful, sometimes the communication between client and tf2 buffer can be bad, sometimes with even one request server could not find the transformation, so with 360 requests there would be the problem. So my solution of this problem, was to implement function that fetches robot's yaw and position ("*get_robot_yaw()*", "*get_robot_position()*") and use the transformation matrix instead of using the functionalities of tf2 toolbox. This solution were expected to work faster, more reliable and does not require more than 1 connection (get robot's position and yaw angle from message under/odom topic) with external services, which keep track about turtlebot. This mine solution, implemented by me, indeed showed good performance and is good alternative to fully trust tf2 with transformation.

8.3.3 Get lines from laser readings method

This method connects laser readings, remapped to the origin coordinate frame, to later find frontiers in those lines. The pseudo code of this algorithm follows on "Algorithm 12".

Algorithm 12 Connect laser readings with lines

```

procedure GET_LINES_FROM_LASER_READINGS(laser_readings_car_orig)
  previous_point("i" : laser_readings_car_orig[0][0],
                "j" : laser_readings_car_orig[0][1])
  lines_list ← [ ]
  for (i = 1, i < laser_readings_car_orig.length, i++) do:
    curr_point("i" : laser_readings_car_orig[i][0],
              "j" : laser_readings_car_orig[i][1])
    lines_list.append(self.DDALine(curr_point, previous_point))
    previous_point = curr_point
  end_for
  curr_point("i" : laser_readings_car_orig[0][0],
            "j" : laser_readings_car_orig[0][1])
  lines_list.append(self.DDALine(curr_point, previous_point))
  return lines_list
end procedure

```

Algorithm starts with writing down first laser reading to the variable "*previous_point*" then creates empty list to store lines, connecting the two neighbouring laser readings. Following for loop connects next laser readings with previous using DDA line algorithm, which will be explained later, resulting the formation of the boundaries between known and unknown. Last lines of the algorithm connects last reading with the first reading to close the boundary between known and unknown and then algorithm returns the *lines_list*

DDA line method

The DDA line algorithm was introduced by McCrea and PW Baker in 1975 [37]. This algorithm solves a problem of connecting the two dots with raster line. The pseudo code of this algorithm follows on "Algorithm 13". The algorithm begins with calculating the difference between points along i and j axes. Then max difference is chosen to be denominator for calculating the steps and incrementations along i and j axes. Following if conditions define to which quadrant the line will be interpolate from starting point. After that the "*points_list*" variable is being allocated to store cells of the line. Following for cycle calculates the cells of line, their number will be equal to the "*steps*" variable and for each step the cell will be incremented with "*i_inc*" and "*j_inc*" variables to calculate new cell of the line. After that the resulted line will be returned as the result of the function.

Algorithm 13 DDA line

```

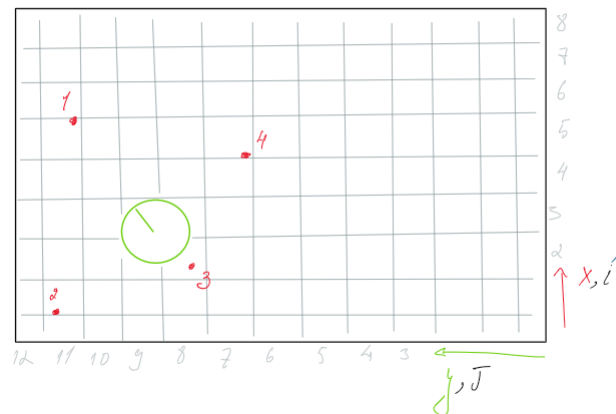
procedure DDALINE(curr_point, previous_point)
   $di \leftarrow |curr\_point[i] - previous\_point[i]|$ 
   $dj \leftarrow |curr\_point[j] - previous\_point[j]|$ 
   $steps \leftarrow \max(di, dj)$ 
  if  $curr\_point[i] < previous\_point[i]$  do:
     $i\_inc \leftarrow -di/steps$ 
  else:
     $i\_inc \leftarrow di/steps$ 
  end_if
  if  $curr\_point[j] < previous\_point[j]$  do:
     $j\_inc \leftarrow -dj/steps$ 
  else:
     $j\_inc \leftarrow dj/steps$ 
  end_if
   $points\_list \leftarrow [ ]$ 
   $new\_point \leftarrow ("i" : previous\_point[i],$ 
     $"j" : previous\_point[j])$ 
  for ( $i = 0, i < steps, i++$ ) do:
     $new\_point \leftarrow ("i" : previous\_point[i] + i\_inc,$ 
       $"j" : previous\_point[j] + j\_inc)$ 
     $points\_list.append(new\_point)$ 
  end_for
  return  $points\_list$ 
end procedure

```

problems and their solutions

Mine implementation of DDA line algorithm and connect laser readings with lines algorithm showed good performance and perfectly do their thing. The result of working of this algorithm can be seen on fig 8.16.

a) Before connecting lines



b) After connecting lines

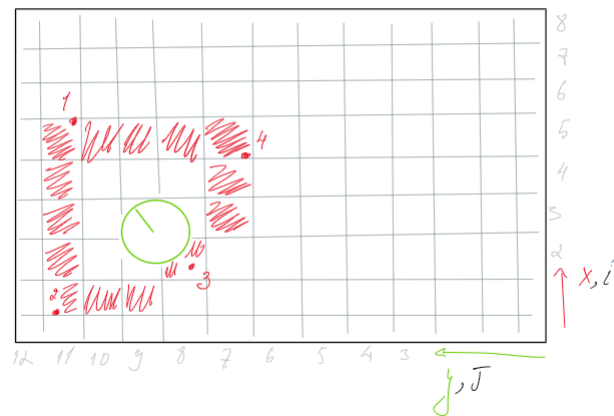


Figure 8.16: Result of working of the DDA line algorithm and the connect laser readings with lines algorithm:

- Before connecting the laser readings
- After connecting the laser readings

8.3.4 Find frontiers in lines method

Find frontiers in lines method serves purpose of finding the frontier points in borders, formed by "*get_lines_from_laser_readings()*" method of FFD detector class. The requirements for a point to be a part of the frontier are:

- to have at least one unknown cell in their neighbourhood
- to not have cells, that contains obstacles in their neighbourhood

If suitable cell is found, then it will be added to already existed frontier or new frontier will be initiated. Frontier is closed, when the first cell, containing obstacle, appears after a row of cells, which have at least one unknown cell as neighbour. I have described

that method in the chapters: 5.2. and 6.3. The frontiers are initiated and maintained as FFDfrontiers class, which acts like datatype for storing frontier cells and will be described later. Next follows the pseudo code of the find frontiers in lines method of FFDdetector class on the "Algorithm 14".

Algorithm 14 Find frontiers in lines method

```

procedure FIND_FRONTIERS_IN_LINES(lines_list)
  frontiers_list ← [ ]
  new_frontier ← FFDfrontier()
  for line ∈ lines_list do:
    for point ∈ line do:
      unkn_flag, obst_flag ← self.check_neighbours(point)
      if unkn_flag and not obst_flag do:
        new_frontier.add_point(point)
      else if obst_flag and new_frontier.list_of_points is not empty do:
        frontiers_list.append(new_frontier)
        new_frontier.delete_all_points()
      end_if
    end_for
  end_for
  return frontiers_list
end procedure

```

The algorithm begins with allocating "*frontiers_list*", which will store all frontiers objects. Then a new "*FFDfrontier*" object is initialised, which will act as a buffer, storing all points of the current frontier. Following for loop checks every point for being suitable for being the frontier's point with help of "*check_neighbours()*" method of the FFDdetector class. Next two scenarios could happen:

- The current point is suitable for the frontier and it will be added to the buffer
- Current point contains obstacle and is not suitable for the frontier and buffer is not empty i.e. frontier needs to be closed to initiate new frontier, as the cells, containing obstacles, will end. In this case, current buffer will be added to the "*frontiers_list*" and then will be erased, to store cells of the new frontier.

In every other case, algorithm just continues checking next cells. At the end method returns "*frontiers_list*" containing all "*FFDfrontier()*" objects, containing the cells of all frontiers. The result of the processing the lines by Find frontiers method can be seen on the fig 8.17.



Figure 8.17: Example of working of the method i.e. classifying points of lines:

- a) Red cells are frontier points
- b) Violet cells contain obstacles in their neighbourhood
- c) Gray do not have unknown cells around them, they are not objects of FFD method

Check neighbours method

Check neighbours method is method of FFDdetector class, which are used in algorithm 14 i.e. find frontiers in lines algorithm, to classify cells if they are suitable for being frontier's points. Method forms a 3x3 neighbourhood of the checked point and evaluates if neighbourhood has obstacles or unknown cells in it. Method forms neighbourhood depending on checked cell:

- **Cell is in center of the map** - normal 3x3 neighbourhood
- **Cell is on the edge of the map** - 3x2 neighbourhood
- **Cell is in the corner** - 2x2 neighbourhood

The example of forming the neighbourhoods can be seen on the fig 8.18.



Figure 8.18: Check neighbourhood method, forming neighbourhood, depending on the position of the checked cell

Problems and their solutions

During the developing of those methods I have encountered the problem that the neighbourhood of the cell, which lies on the border or in the edge is not being checked for having unknown cells or obstacles and my solution was to form the neighbourhood individually for cells laying in the different places. That solution works great, because now FFD method is picking frontiers, laying in the corners of the map or on the edges.

8.3.5 Transform map frame to origin frame and back method

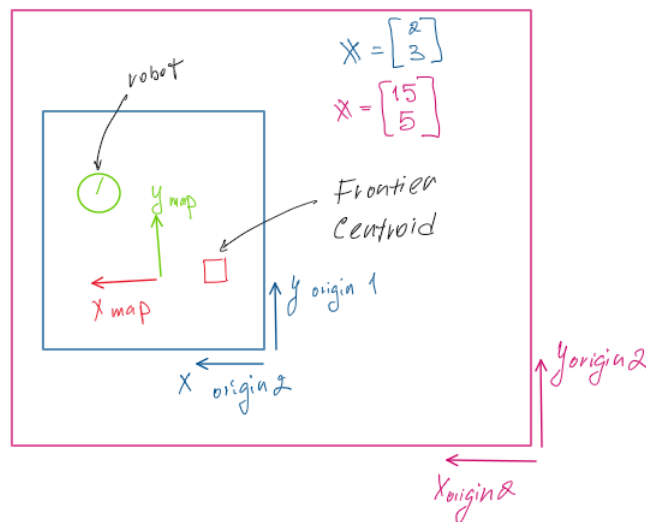


Figure 8.19: drift of the coordinates of the frontier's centroid with map growth

"Transform map to orig and back" method of FFDdetector class, serves a purpose of transforming points from map coordinate frame and back, depending on user input. It used for transferring frontier's centroids from origin coordinate frame to map coordinate

frame before storing them in the "*global_pool_of_frontiers*" list attribute of FFDdetector class and translate frontier's centroid from map coordinate frame to origin coordinate frame before sending it as goal.

The storage of frontier's centroid is conducted in the map coordinate frame, because that way frontier's centroid would not change as the map would grow, so this function is necessary for FFD method to maintain found frontiers. The drift of the coordinates of frontier's centroid coordinates in origin map frame with growth of the map is shown on fig 8.19. and as it can be seen from this same figure, the drift of the coordinates of frontier's centroid in map coordinates frame with map growth is zero. This is why the frontier's centroids are stored in the map coordinate frame and this is why the "transform map frame to odom frame and back" method is necessary to FFD method.

Next follows the pseudo code of the algorithm on the figure "Algorithm 15"

Algorithm 15 Transform point from map coordinate frame to origin coordinate frame

```

procedure TRANSF_MAP2ODOM_AND_BACK(from_which_frame,
to_which_frame, point)
  if from_which_frame = "orig" and to_which_frame = "map" do:
     $\vec{x}_{origin} \leftarrow (x : point[x],$ 
       $y : point[y])$ 
     $\vec{x}_{map2origin} \leftarrow self.map\_origin\_position$ 
     $\vec{x}_{map} \leftarrow \vec{x}_{origin} + \vec{x}_{map2origin}$ 
    return  $\vec{x}_{map}$ 
  end_if
  if from_which_frame = "map" and to_which_frame = "orig" do:
     $\vec{x}_{map} \leftarrow (x : point[x],$ 
       $y : point[y])$ 
     $\vec{x}_{map2origin} \leftarrow self.map\_origin\_position$ 
     $\vec{x}_{origin} \leftarrow \vec{x}_{map} - \vec{x}_{map2origin}$ 
    return  $\vec{x}_{origin}$ 
  end_if
end procedure

```

So, the the algorithm can be described by the figure 8.20. If the *from_which_frame* = "orig" **and** *to_which_frame* = "map", then to get \vec{x}_{map} method need to add \vec{x}_{origin} vector to the $\vec{x}_{map2origin}$ vector, which is obtained from *self.map_origin_position* attribute of the FFD detector class. If arguments are backwards, then method need to subtract $\vec{x}_{map2origin}$ vector from \vec{x}_{map} vector.

FFDfrontier class serves the same purposes as Cluster class served to the DFDfrontierDetector, it stores the frontier cells and has methods for calculating their number and calculating the centroid of frontier i.e. mean coordinate of those cells. Next comes the UML class diagram of the FFDfrontier class on the fig 8.21. Class has 3 attributes. **List_of_points** attribute is list, belonging to the frontier. **Centroid** attribute stores the row and the column of the calculated centroid i.e. mean coordinate. **num_of_elements** is integer, that stores the length of the "*list_of_points*" attribute. Moving on to the methods. **add_point()** method adds a point to the "*list_of_points*" attribute. **Delete_all_points()** method deletes all the points from "*list_of_points*" attribute, this method was implemented to erase all of the points after "*find_frontiers_in_lines()*" method will encounter the cell with obstacle and the buffer, to which all previous points, suitable for frontier were assigned, will need to be closed and appended to the "*frontiers_list*". All of this were described in the chapter 8.3.4.

This datatype was developed by me as the solution for storing all the cell of each frontier, found by FFD method. Storing those frontiers as list of cells was not comfortable for the computation and storing them. This solution showed a good performance and served its purpose well.

Chapter 9

Testing with simulation software:

This chapter describes the testing of the developed software with simulation software. Simulation of turtlebot3 is executed in the Gazebo software and visualisation of the process of the simulation is executed in RViz software. The testing takes place on two maps. First small map is a hexagon map with 9 round obstacles, situated in the centre of the map and it looks like this: fig 9.1. Its area is roughly $25 [m^2]$.

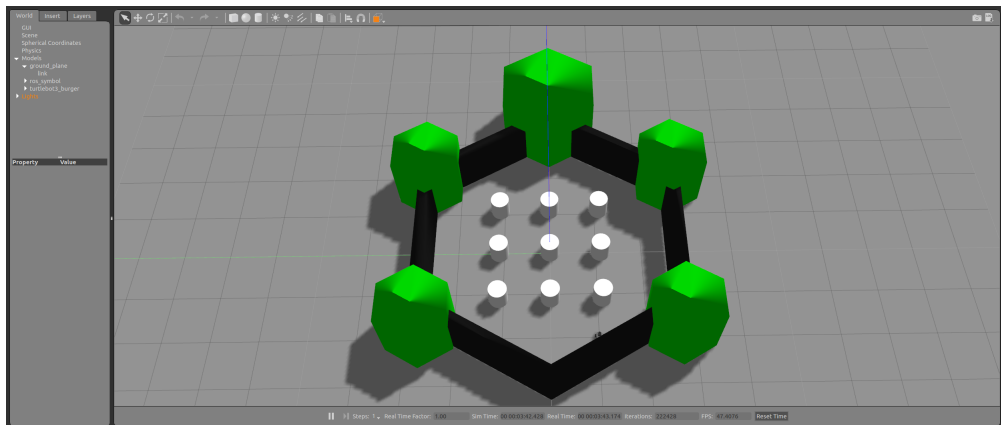


Figure 9.1: Small map for testing the turtlebot3 burger in simulation environment

As for the second map, the file containing the map model is corrupted, so the photo of 3d model can not be shown, but in 2D map looks like this fig 9.2. Area of this map is $153 [m^2]$ and it has 4 obstacles: two tables, a ladder and a bookshelf.

As the simulation environment is limited in the scenarios for the simulations to run (that means that methods are relatively stable and results are approximately the same). Only one run for each method will be presented in the simulation environment for each map. More experiments will be presented in the section where the real turtlebot3 is tested.



Figure 9.2: Big map for testing the turtlebot3 burger in simulation environment

9.1 Testing on small map

9.1.1 Exploration with naive Frontier Detection method:

The time it took for the robot to open all the cells with naive frontier detection method is 7:19 [mm:ss]. As it can be seen from graphs 9.3. and 9.4., the method is inconsistent. The number of opened cells has a sharp increase approximately in the area of 2 minutes or the 4-th iteration of loop into the exploration mission. The same trend can be seen on the graph of velocity of opened cells. Velocity has a sharp increase near the forth iteration of loop and near the 14th iteration of loop. The rest of the time robot was not discovering cells, and was standing in the same place because of wrong inputted goal, or was driving on the already discovered area of the map.

Path shown on the fig 9.5. is 11 [m] and is consistent and has acceptable length for this type of map.

The discovered map has slight unknown area, but discretized map corresponds to the map in the simulation environment. The bad side of this method is the time. It took the robot 7:19 [mm:ss] to compose a map, which is too much time to map the 25 square meter area.

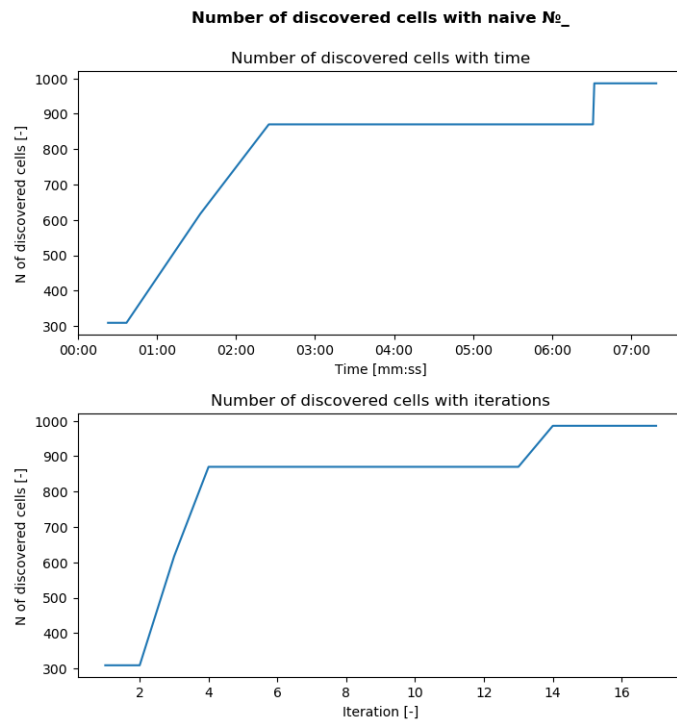


Figure 9.3: Number of discovered cells with iterations and time graphs

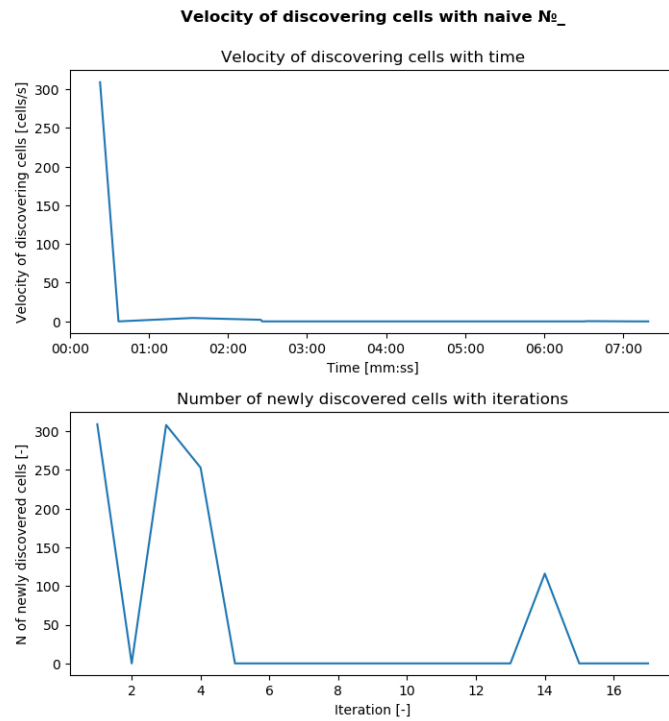


Figure 9.4: Velocity of discovering cells with iterations and time graphs

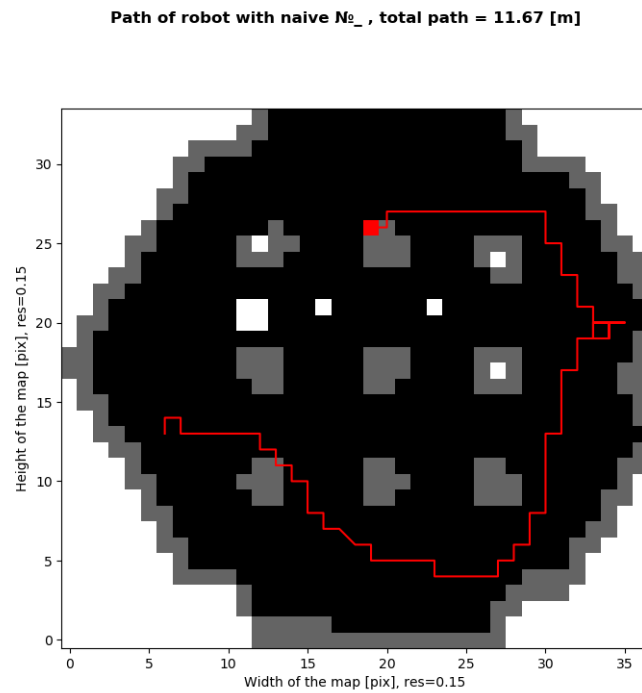


Figure 9.5: Total path of the robot, shown on the map

| Number of oppened cells | Length of path [m] | Number of iterations |
|-------------------------|--------------------|----------------------|
| 986 | 11.67 | 17 |

Table 9.1: Results of naive frontier detection method on the small map, part 1

| Time | Mean velocity [cells/s] | Median velocity [cells/s] |
|-------|-------------------------|---------------------------|
| 07:19 | 18.58 | 0.0 |

Table 9.2: Results of naive frontier detection method on the small map, part 2

Velocities of opening of new cells, shown on tables: 9.1 and 9.2. are quite small. The higher mean velocity shows that there were big increases in the opening of new cells but again, they were inconsistent.

9.1.2 Exploration with Yamauchi's frontier detection method

Next method for evaluation is Yamauchi's frontier detection method. The results from graphs 9.6. and 9.7. show that this method is far more consistent than previous one. From the "numbers of discovered cells with iterations" graph can be seen that this method opens cells at almost every iteration.

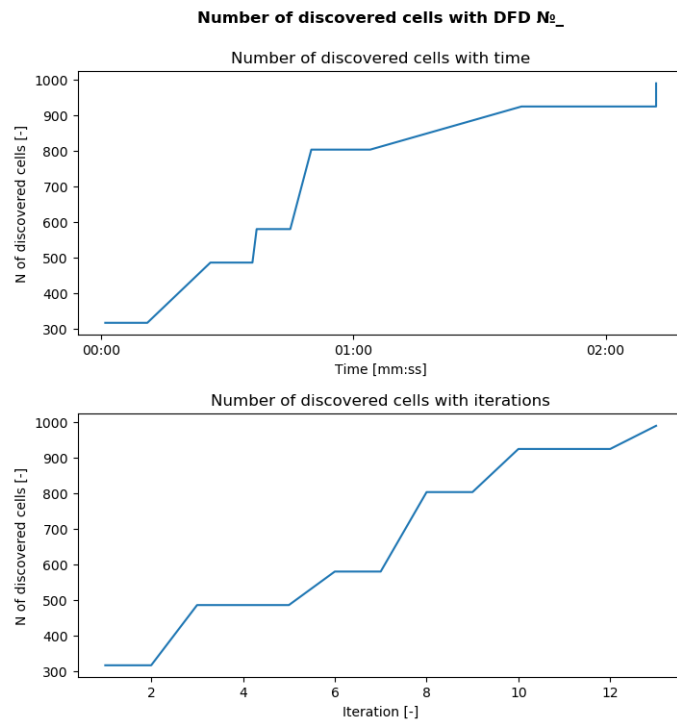


Figure 9.6: Number of discovered cells with iterations and time graphs

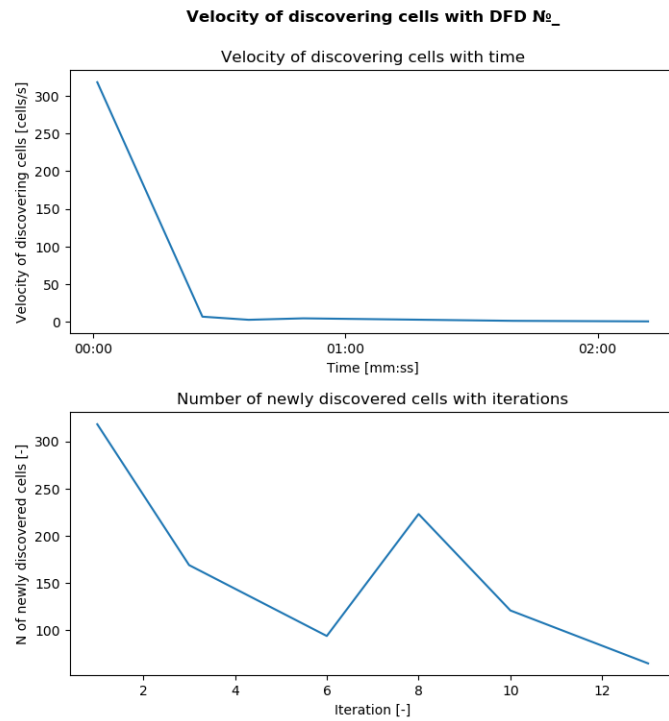


Figure 9.7: Velocity of discovering cells with iterations and time graphs

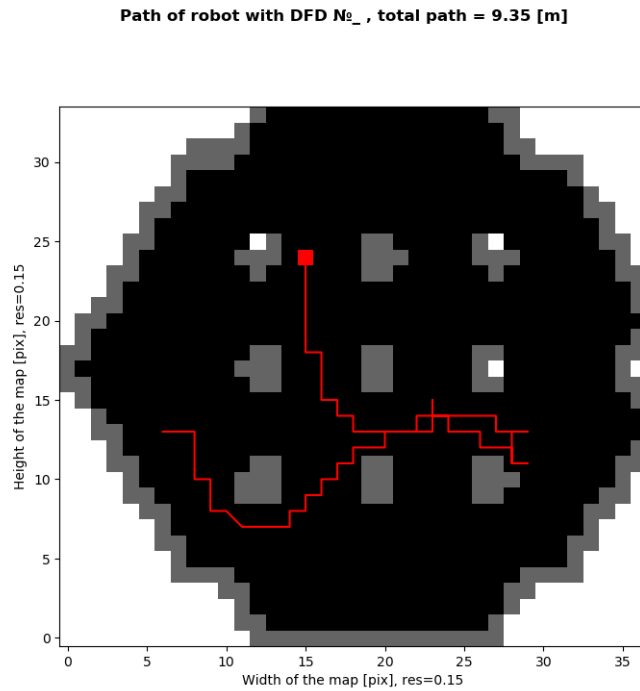


Figure 9.8: Total path of the robot, shown on the map

| Number of oppened cells | Length of path [m] | Number of iterations |
|-------------------------|--------------------|----------------------|
| 990 | 9.35 | 13 |

Table 9.3: Results of YFD method on the small map, part 1

| Time | Mean velocity [cells/s] | Median velocity [cells/s] |
|-------|-------------------------|---------------------------|
| 02:12 | 55.61 | 3.58 |

Table 9.4: Results of YFD method on the small map, part 2

More consistent functioning of YFD method can also confirm the results from the table 9.4. Mean velocity is almost 3 times higher, than mean velocity from the previous method. However, mean velocity does not actually tell us that the robot is discovering 50 cells per second. The more objective result is the median velocity and it tells us that the robot discovers 3.6 cells per second, which is a more accurate result.

Path of the robot shown on the fig 9.8. is slightly shorter, but the difference is not significant. The significant change is the time of the mapping with this method, which is 02:12 [mm:ss]. This is a substantial decrease in the time and it clearly demonstrates the difference between naive frontier detection and not naive.

Map obtained from the method corresponds with the map in simulation environment.

9.1.3 Exploration with fast frontier detection method

FFD method surpasses the results reached by YFD method. The time of the exploration mission is now two times shorter, the graph 9.9. looks less consistent, but it is because the exploration mission took less time. The graph is only flat during the 4-6 loop iterations. During all the other iterations, the robot tends to discover new cells. This is also confirmed by the graph on the fig 9.10. The number of newly discovered cells gradually decreases instead of dropping or behaving inconsistently, as the previous methods were. Velocity is also gradually decreasing instead of dropping. This concludes the effectiveness and consistency of the method in discovering new cells. The time of the exploration mission is two times lower than in the previous method. At the same time, the iterations are not, that confirms, that FFD method is faster and more aggressive in the exploration of the map.

| Number of opened cells | Length of path [m] | Number of iterations |
|------------------------|--------------------|----------------------|
| 989 | 8.11 | 9 |

Table 9.5: Results of FFD method on the small map, part 1

| Time | Mean velocity [cells/s] | Median velocity [cells/s] |
|-------|-------------------------|---------------------------|
| 01:05 | 64.23 | 5.27 |

Table 9.6: Results of FFD method on the small map, part 2

As for the results in the tables 9.5 and 9.6, they confirm the improvement of this method over YFD. The length of the path is slightly smaller. Mean and median velocity are bigger, which concludes the effectiveness of the method.

Map obtained by the exploration with FFD method corresponds to the simulation environment.

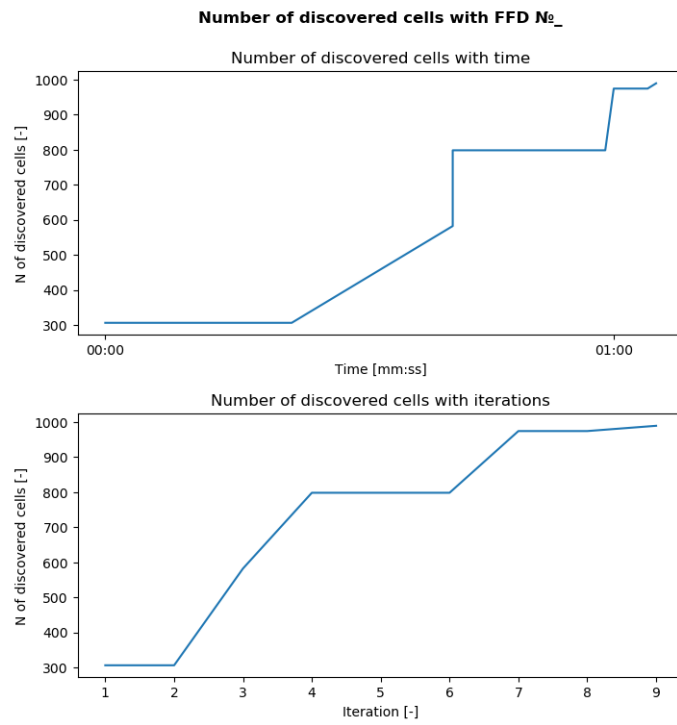


Figure 9.9: Number of discovered cells with iterations and time graphs

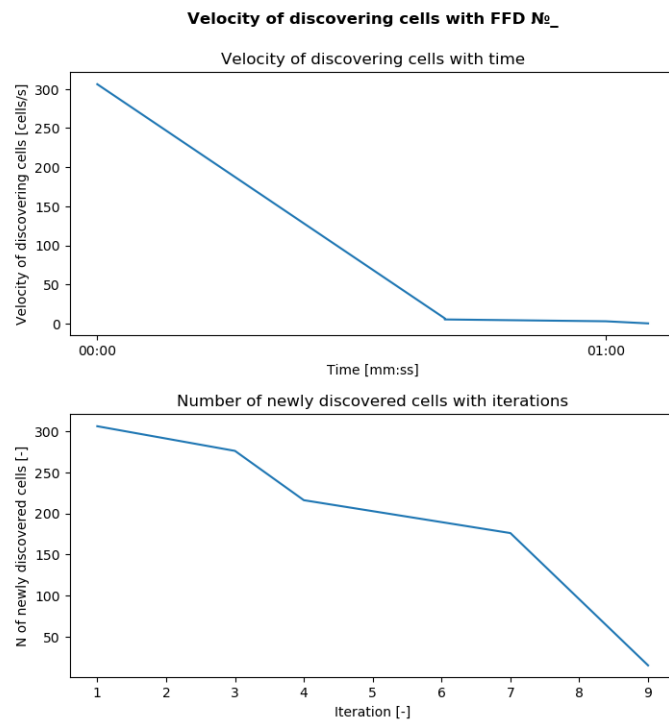


Figure 9.10: Velocity of discovering cells with iterations and time graphs

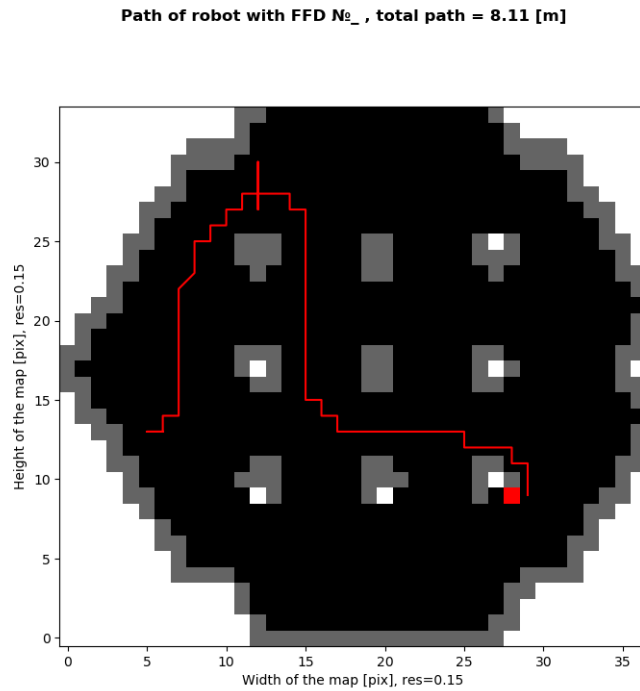


Figure 9.11: Total path of the robot, shown on the map

9.1.4 Overall comparing of the performance on the small map

| Name | Number of oppened cells | Length of path [m] | Number of iterations |
|-------|-------------------------|--------------------|----------------------|
| naive | 986 | 11.67 | 17 |
| YFD | 990 | 9.35 | 13 |
| FFD | 989 | 8.11 | 9 |

Table 9.7: Overall results method on the small map, part 1

| Name | Time | Mean velocity [cells/s] | Median velocity [cells/s] |
|-------|-------|-------------------------|---------------------------|
| naive | 07:19 | 18.58 | 0.0 |
| YFD | 02:12 | 55.61 | 3.58 |
| FFD | 01:05 | 64.23 | 5.27 |

Table 9.8: Overall results on the small map, part 2

From tables 9.7. and 9.8. can be seen, that the results are getting better with the more sophisticated methods applied. The number of the opened cells does not change, which is logical because each method proved itself to be able to obtain the map of small room.

The length of the path improves slightly but not significantly. In contrast, the number of iterations to map the room falls sharply. By comparing 17 loop iterations per 7 minutes of naive method to 9 iterations per 1 minute of FFD method, it can be deduced that FFD

method is fastest and the most aggressive, at least on small maps, which is also confirmed by a dramatic decrease in the time of the exploration mission.

With the application of more sophisticated method, the mean velocity increases as well as the median velocity, which is a good sign of the methods working as expected.

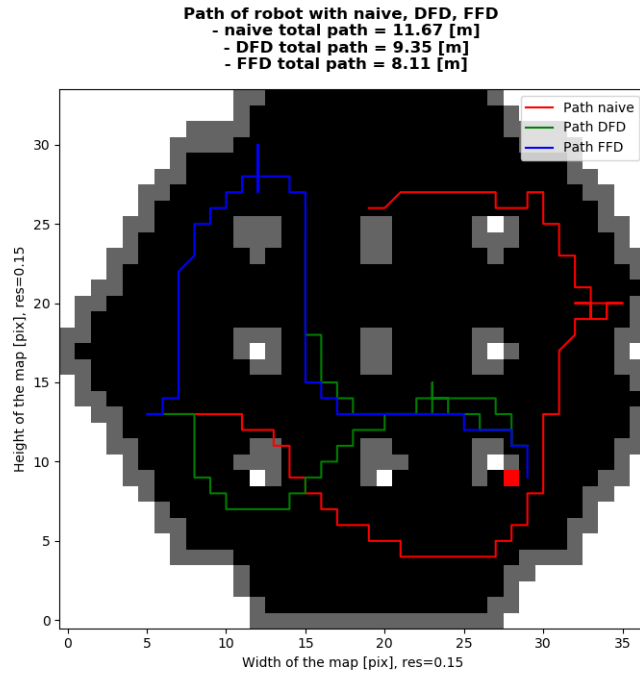


Figure 9.12: Evaluated and compared paths of all of the three methods

From the comparison of the paths, conducted on the fig 9.12. It can be seen, that the path of FFD is most optimised on this map, after driving from starting point, the method makes robot to move to the left and discover left part of the map, then makes it to move to the right. Also it can be seen that the actions of the turtlebot under FFD method are more aggressive.

The YFD method's path makes robot to move on the left, but then makes him return to the center. This path is not optimised but it is way more efficient, than naive method's path as it can be seen from tables 9.7. and 9.8.

9.2 Testing on the big map

9.2.1 Exploration with naive frontier detection method

As it can be seen from figures 9.13., 9.14 the naive method repeats the pattern from the small map. It has inconstant results during exploration mission. "Number of discovered cells with iterations" graph looks more like ladder, than like linear or some other depen-

dency. The inconstant results are showed good also on the "Number of newly discovered cells with iterations" graph, as there can be seen, that there are clearly more successful iterations, thanks to which the robot discovered more new cells then in other loop iterations.

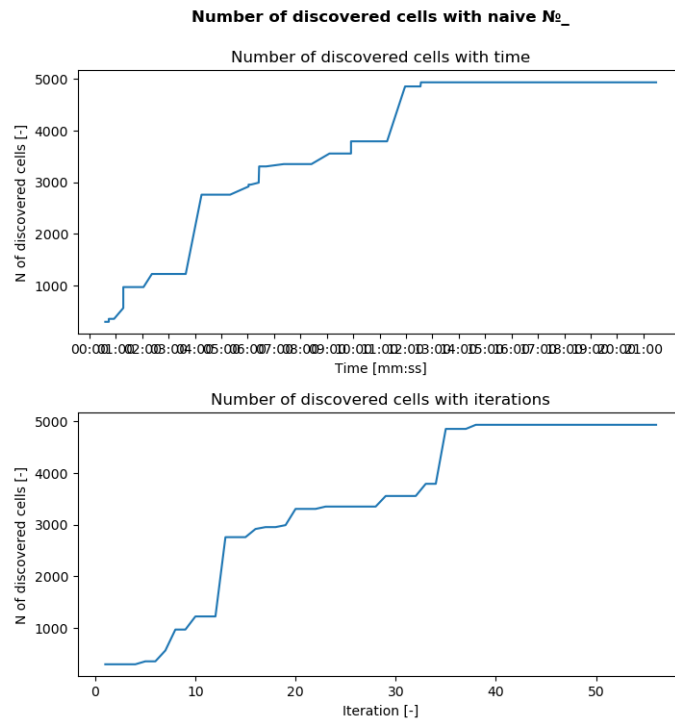


Figure 9.13: Number of discovered cells with iterations and time graphs

Map, obtained by naive method resembles the map in simulation environment (fig 9.15.), but has imperfections. It still has areas, which are not explored and filled and this method doubtfully could send robot there. As all unknown cells have equal probability to be chosen as the goal and on the current map there are more unknown cells outside the walls, the chances are higher, that robot would be sent outside the walls and action lib could not get a path for that goal and robot would initialize a recovery behaviour. The path of the robot, which equals 69.3 [m] is adequate for map with big size, such as this one.

About overall results for this particular method and this particular map, shown on the tables 9.9. and 9.10. The number of opened cells are slightly less, than with others methods. Length of the path is less than with other methods, but that means, that robot was staying one one place because of invalid goal outside the walls. Time is extremely big, it is 1.5x or even 2x bigger, than with other methods. Median and mean velocities are just confirming, that the method is inconstant in its results during the exploration mission

| Number of oppened cells | Length of path [m] | Number of iterations |
|-------------------------|--------------------|----------------------|
| 4925 | 69.3 | 56 |

Table 9.9: Results of naive method on the big map, part 1

| Time | Mean velocity [cells/s] | Median velocity [cells/s] |
|-------|-------------------------|---------------------------|
| 21:29 | 6.1 | 0.0 |

Table 9.10: Results of naive method on the big map, part 2

9.2.2 Exploration with Yamauchi's frontier detection method

Immediately from the results shown on the figures 9.16. and 9.14 we can see the difference between naive method and more sophisticated method. The "Number of discovered cells with time and iterations" graph shows nearly linear(!) dependency for increasing of the number of opened cells with time. The number of the newly discovered cells with iterations is still a little bit inconstant towards to the end of exploration mission, but the value of number of newly discovered cells do not fall to the zero almost never.

About path shown on the figure 9.18. The path really shows, that robot knows, what he does under YFD method in comparison with naive method shown on the fig 9.15. On YFD graph you can see, that robot drives along some contour, which resembles map itself. The contour on the figure 9.18 is closed and includes in it the hard-to-reach places with unknown cells (for example behind the shelf), that means that robot purposefully goes to the place with unknown cells and this is great result. The map, obtained from the robot with YFD method resembles the original map from simulation environment more than map, obtained with naive method. Still the map has imperfections.

About overall results with YFD particular method and on this particular map shown on the tables 9.11. and 9.12. The number of opened cells is slightly higher, than with naive method, which shows, that the map was composed more fully. Length of path is 106 [m], which is normal for map with big size. The YFD's path is longer than the naive's path is only because the robot under YFD method was constantly moving, that confirms the 147 iterations per 12 minutes against 56 iterations per 21 minutes with the naive method. Mean and median velocities are higher than with last method, which shows the effectiveness of this method with comparison of naive method.

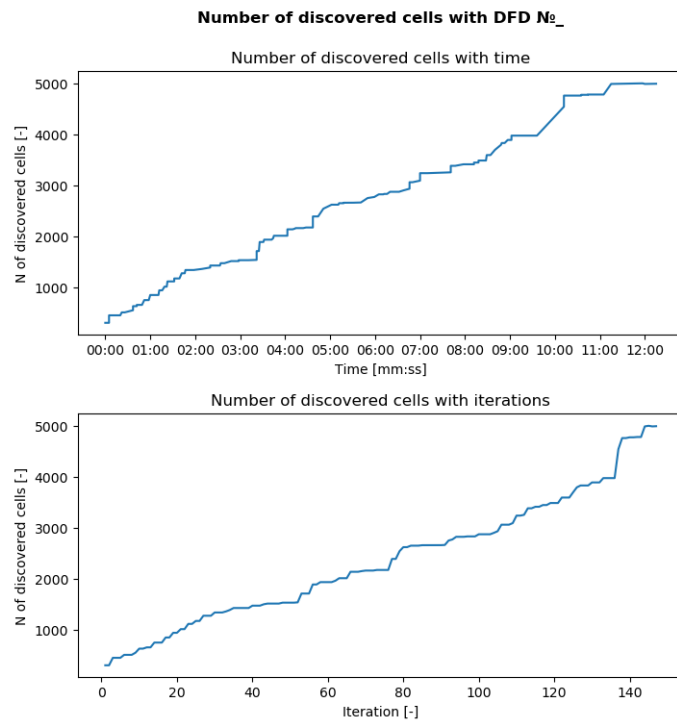


Figure 9.16: Number of discovered cells with iterations and time graphs

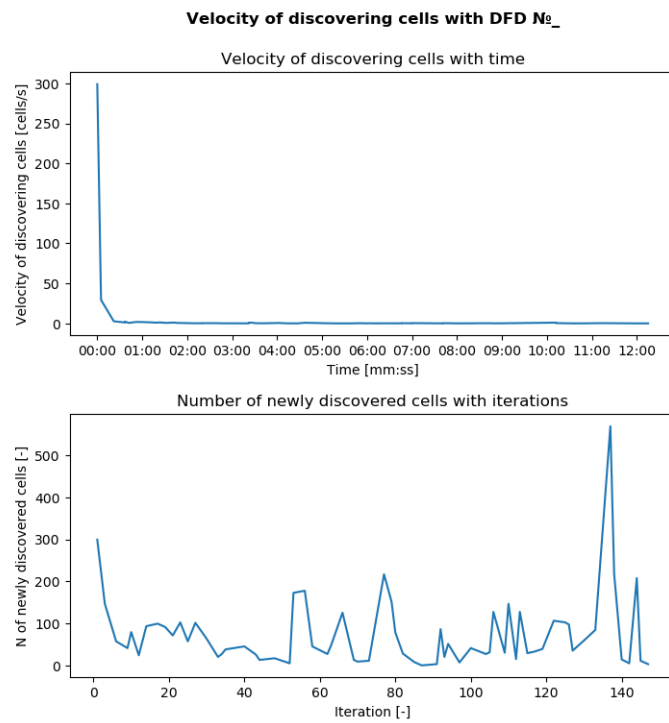


Figure 9.17: Velocity of discovering cells with iterations and time graphs

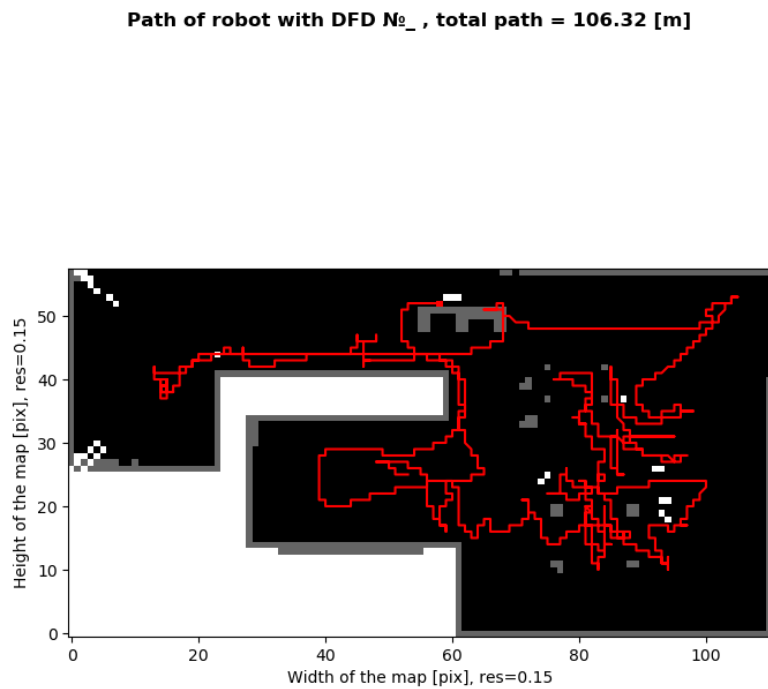


Figure 9.18: Total path of the robot, shown on the map

| Number of oppened cells | Length of path [m] | Number of iterations |
|-------------------------|--------------------|----------------------|
| 4994 | 106.32 | 147 |

Table 9.11: Results of YFD method on the small map, part 1

| Time | Mean velocity [cells/s] | Median velocity [cells/s] |
|-------|-------------------------|---------------------------|
| 12:15 | 5.37 | 0.2 |

Table 9.12: Results of YFD method on the small map, part 2

9.2.3 Exploration with fast frontier detection method

Fast frontier detection copy the pattern from the small map. It is way more better handles the exploration mission than the naive method as well as FFD method, but it also improves in comparison with FFD method too.

As it can be seen from the graphs 9.19. and 9.20 the "Number of discovered cells with iterations and time" resembles the exponential dependence, which is better than

linear dependence from graph 9.16. with YFD method. The better efficiency also confirms "Number of newly discovered cells with iterations" graph, where numbers of newly discovered cells are big and not equal 0 all the time thought first 50 loop iterations.

The map obtained from the robot with the FFD method, shown on the figure 9.21., has a better resemblance, than with previous methods, it still has minor imperfections, but overall quality is better. The path, which robot drove under the FFD method still looks like a closed contour of the map, but with more aggressive moves towards the unknown regions of the map. That is good for the exploration mission.

About overall results, shown on the tables 9.13 and 9.14. The number of opened cells is almost 200 cells higher, which confirms the better quality of the map and more explored map as the result of the exploration mission. Length of the path is also 10 meters shorter, which is good. Numbers of iteration is 69 per 12 minutes. This tells us that FFD method has about 8.5 [s] for iteration, where DFD method has 5 [s] for iteration. This can tell us, that the robot was stuck at some loop iterations, which can be caused by more aggressive manner of exploration. That is something, that is inherent for this method, because of noise on the LDS sensor the measurement can be shown as measurement outside the wall, and when method connects this measurement with neighbouring ones, the contour between known and unknown can pass outside the wall, where unknown cells are present, but they are not achievable, as the robot conducts the exploration mission in the closed room. So the 69 iterations per 9 minutes is lower result, than

Although time is improved with FFD method comparing to the YFD method. Mean and median velocities has slight improvements too.

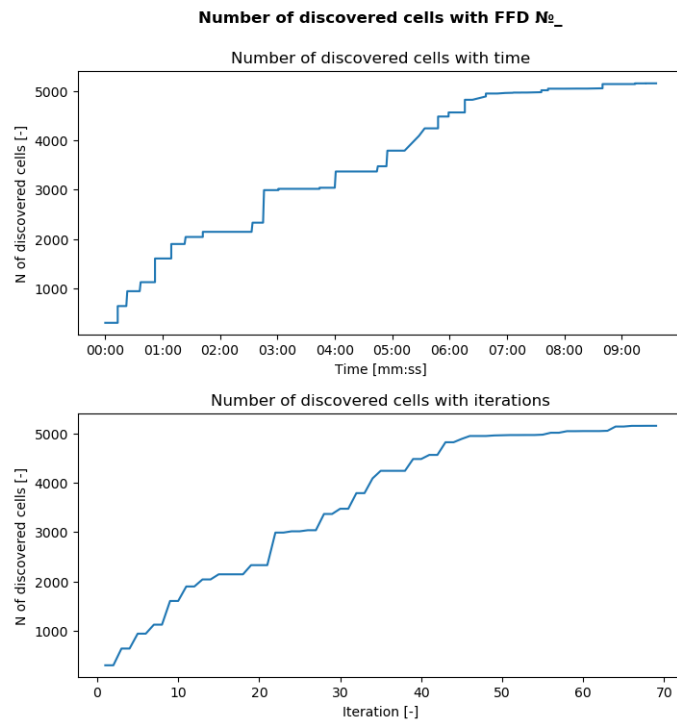


Figure 9.19: Number of discovered cells with iterations and time graphs

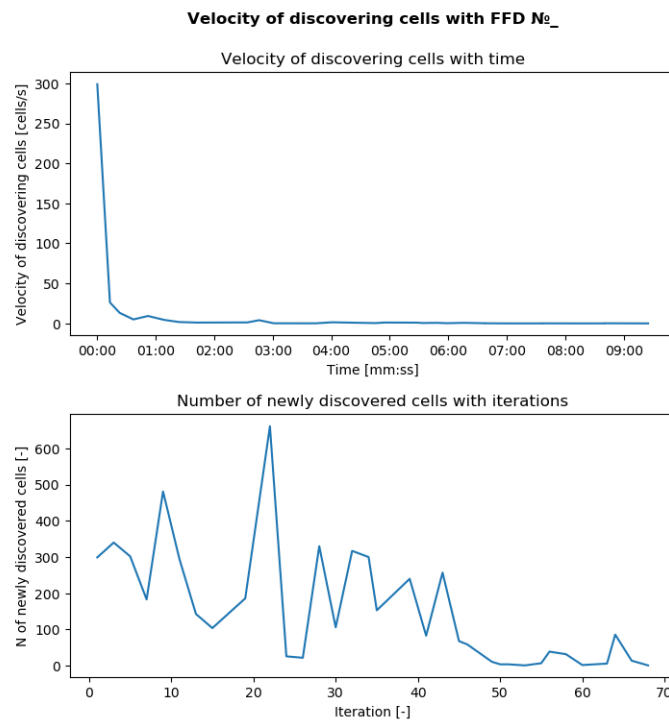


Figure 9.20: Velocity of discovering cells with iterations and time graphs

Path of robot with FFD №_ , total path = 97.66 [m]

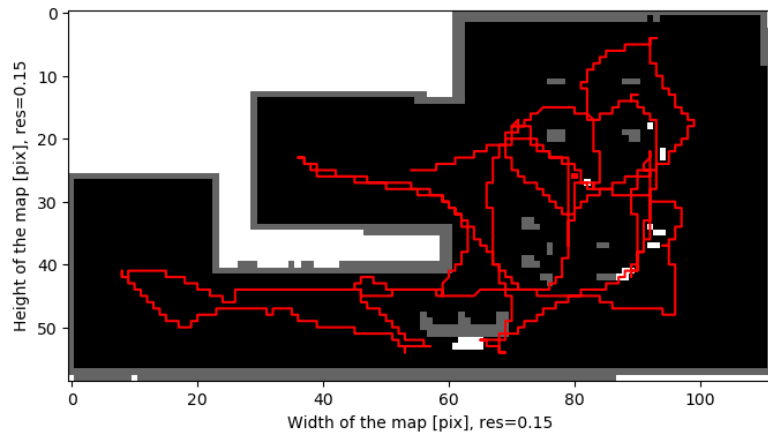


Figure 9.21: Total path of the robot, shown on the map

| Number of oppened cells | Length of path [m] | Number of iterations |
|-------------------------|--------------------|----------------------|
| 5161 | 97.66 | 69 |

Table 9.13: Results of FFD method on the small map, part 1

| Time | Mean velocity [cells/s] | Median velocity [cells/s] |
|------|-------------------------|---------------------------|
| 9:36 | 10.92 | 0.3 |

Table 9.14: Results of FFD method on the small map, part 2

9.2.4 Overall comparing of the performance on the big map

| Name | Number of oppened cells | Length of path [m] | Number of iterations |
|-------|-------------------------|--------------------|----------------------|
| naive | 4925 | 69.3 | 56 |
| YFD | 4994 | 106.9 | 147 |
| FFD | 5161 | 97.66 | 69 |

Table 9.15: Overall results method on the small map, part 1

| Name | Time | Mean velocity [cells/s] | Median velocity [cells/s] |
|-------|-------|-------------------------|---------------------------|
| naive | 21:29 | 6.1 | 0.0 |
| YFD | 12:15 | 5.37 | 0.2 |
| FFD | 9:36 | 10.92 | 0.3 |

Table 9.16: Overall results on the small map, part 2

As it can be seen from comparison tables 9.15 and 9.16 the results gets better because of applying the more and more sophisticated methods. The map becomes more and more explored, that confirms the increasing of the number of opened cells. Mean and median velocity are increased too, which is a good sign. The time dramatically decreases two times, which is a great sign. The "Number of discovered cells with time" graph goes from inconstant increasing with naive method, to linearly increasing with YFD method, to almost exponentially increasing with FFD. This is a sign of a progress and that results of comparing three methods in the simulation environment on the small map, keeps its relevancy on the big map.

So FFD method seems to be the best at all. However there are some slight inconveniences, FFD method showed better time per loop iteration on the small map, but on the big map it is the second best time. And iterations decreased in number. This means that some loops, robot was stuck because of invalid goal. And invalid goal was caused by noise from LDS measurement, leading the algorithm occasionally to send goals outside the walls. This is the weak spot of this method, but even on the big map in simulation, this did not prevent the FFD method from coming as the most efficient method.

As it can be seen from path graph on the fig 9.22. The paths show better patterns with applying more sophisticated methods. Naive method's path (it also can be seen on the fig 9.15.). It is not optimised in any way, it is not closed, it does not purposely comes to the undiscovered places (for example behind the bookshelf, the map with all the obstacles marked can be seen on fig 9.2).

YFD's method (also fig 9.18.) has shown the optimisation. The path of the robot under YFD method showed the pursuit for the unknown places. Its path resembles the contour of the map, which means, that the robot had driven along some frontier of known and unknown and this is the exact result, expected from this method

FFD's method showed the same results in path (pursuit for the unknown, contour of path copies contour of the room), but with more aggressive moves toward the unknown. And this result was also expected from YFD.

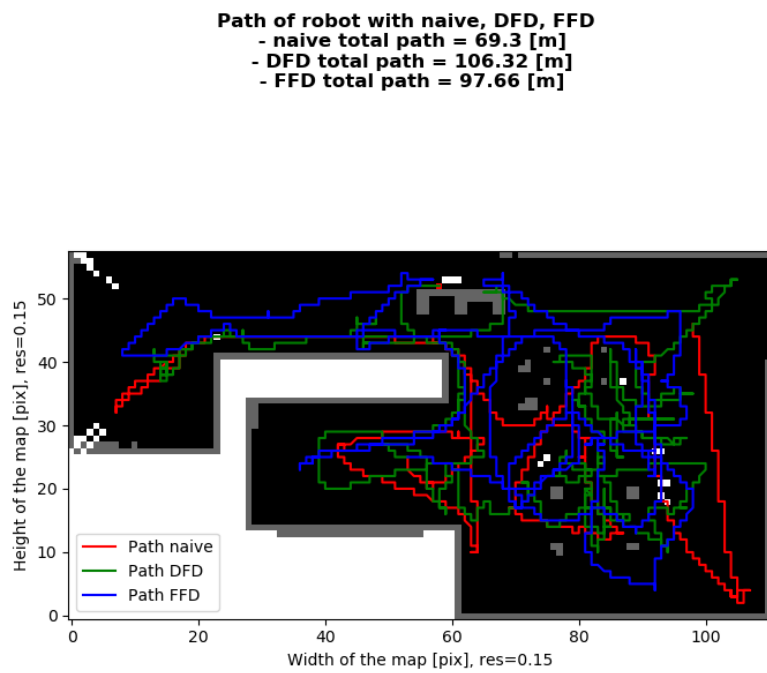


Figure 9.22: Evaluated and compared paths of all of three methods on the big map

Chapter 10

Testing on turtlebot3 burger:

This chapter will describe the testing of the developed methods on real turtlebot3 in the real environment, with real obstacles. The testing room has similar size as small map in the simulation environment. It is 5x4 [m] room with 5 different obstacles. The obstacles are: three big boxes and two round buckets. The room looks like this

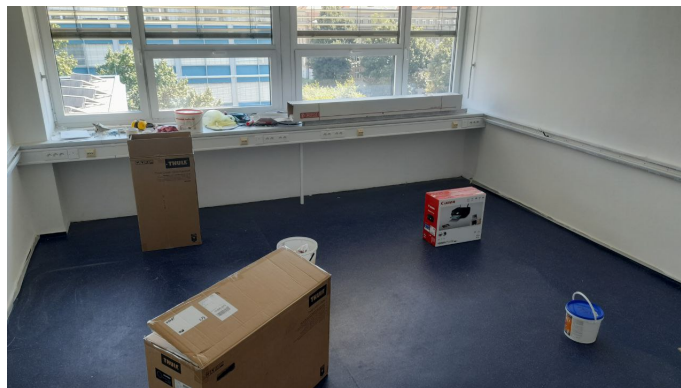


Figure 10.1: Photo of testing room №1



Figure 10.2: Photo of testing room №2

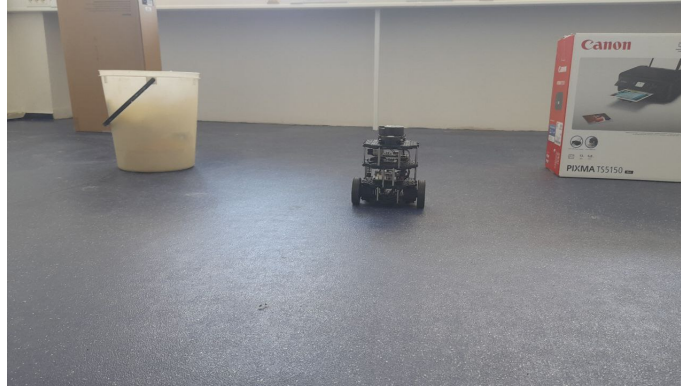


Figure 10.3: Photo of testing room №3, turtlebot in comparison with obstacles

There are 5 measurements of exploration mission for each method. It is possible to ensure the same starting position for robot at the beginning of each exploration mission, but it is impossible to ensure the same orientation of map towards the robot, as it may initialise differently. Due to above mentioned reasons, the experiments in real life were conducted from the different starting points and orientations on the map. For each method there are also "Number of discovered cells with iterations and time", "Velocity of discovering cells with iterations and time" and "Total path" graphs, recorded for best out of five exploration missions.

10.1 Exploration with naive frontier detection method

| № | Number of opened cells | Length of path [m] | Number of iterations |
|--------|------------------------|--------------------|----------------------|
| 1 | 1230 | 12,37 | 20 |
| 2 | 1340 | 10.79 | 32 |
| 3 | 844 | 10.91 | 5 |
| 4 | 1135 | 6.73 | 23 |
| 5 | 1335 | 14.06 | 16 |
| mean | 1176.8 | 10.97 | 19.2 |
| median | 1230 | 10.91.66 | 20 |

Table 10.1: Overall results of naive method in the real room, part 1

The results from real room from robot with naive frontier finding exploration came pretty good in comparison with simulation results, tables 10.1 and 10.2. But one measurement is stands out from all of the measurements. During the third exploration mission, while mapping, robot drove too close to the obstacle and could not free himself, so the exploration mission was technically over. I decided to include this measurement, but it lasted only one minute and during that minute the method send robot to a good position and because of that robot has one or two measurements and the good one opened 800 cells

| № | Time | Mean velocity [cells/s] | Median velocity [cells/s] |
|----------|-------------|--------------------------------|----------------------------------|
| 1 | 05:48 | 200.13 | 2.59 |
| 2 | 07:14 | 12.45 | 0.0 |
| 3 | 01:03 | 313.71 | 313.71 |
| 4 | 04:40 | 32.61 | 0.0 |
| 5 | 03:17 | 39.93 | 0.0 |
| mean | 04:24 | 119.76 | 63.26 |
| median | 04:40 | 39.93 | 0.0 |

Table 10.2: Overall results of naive method in the real room, part 2

and this is why the mean velocity of this measurement is so high. For obvious reasons I did not use this method for evaluation with graphs as it is technically the best result, but exploration mission was not completed, so I treat this measurement as anomaly.

Moving on to the results in the table. Mean and median values are pretty close to each other and that signals, that measurements are valid, so on average naive method opened 1230 cells, which is less than other methods and this tells us that the quality of the map is less than with other methods.

Mean and median lengths of the path are also close to each other this method resulted approximately the same length of path as others methods

Number of iterations are little higher, than number of iterations from other methods.

On average it took turtlebot 4 minutes 30 seconds to map the room, which is a lot longer, than exploration mission took with other methods.

In the mean and median velocity values there is more trust to the median value of mean and median velocity. As one higher value in dataset can bring down the correctness of the mean value, the median value will be taken to evaluate turtlebot. Mean values of mean and median velocities are lower, than mean and median values from other methods.

The 10.5 and 10.6 graphs tell us about inconstant results with opening new cells in exploration mission, this confirm the results from the tables 10.1 and 10.2.

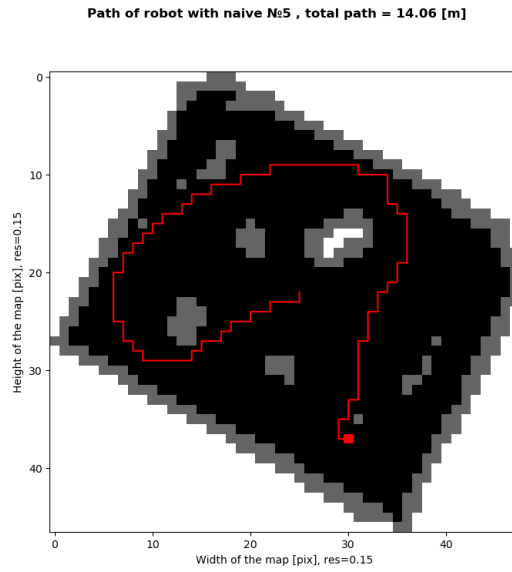


Figure 10.4: Best result for naive frontier detection, path of the robot

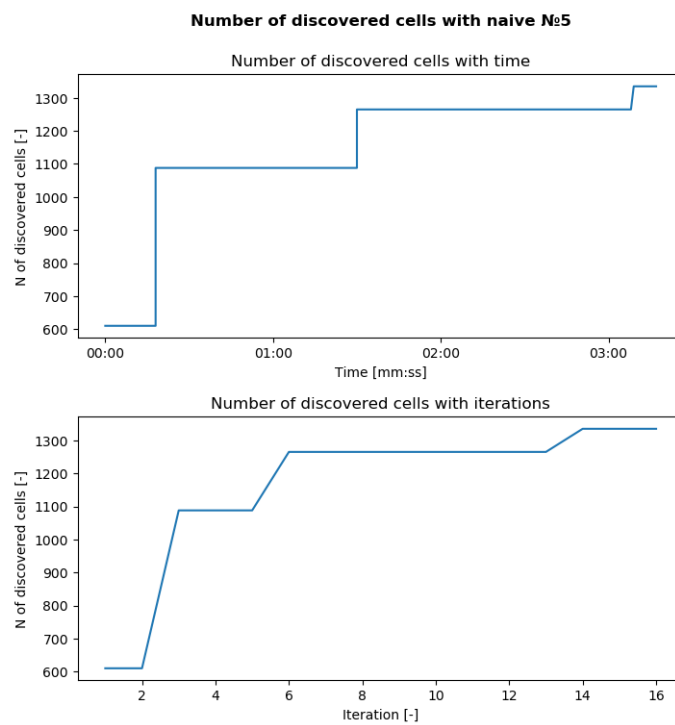


Figure 10.5: Best result for naive frontier detection, number of discovered cells with time and iterations graph

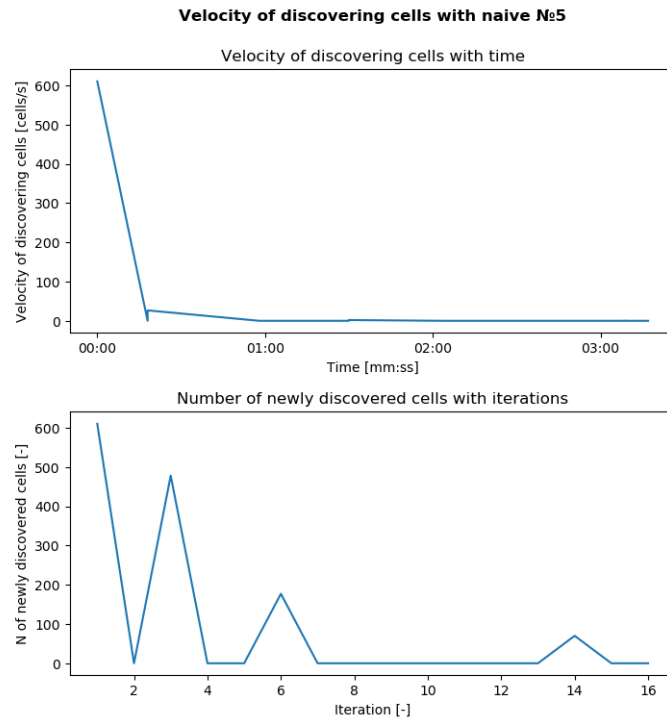


Figure 10.6: Best result for naive frontier detection, velocity of discovering cells with time and iterations graph

10.2 Exploration with Yamauchi's frontier detection

| № | Number of oppened cells | Length of path [m] | Number of iterations |
|----------|--------------------------------|---------------------------|-----------------------------|
| 1 | 1306 | 12.15 | 16 |
| 2 | 1304 | 8.82 | 11 |
| 3 | 1329 | 10.05 | 13 |
| 4 | 1344 | 15.22 | 24 |
| 5 | 1021 | 5.52 | 8 |
| mean | 1260.8 | 10.35 | 14.4 |
| median | 1306 | 10.05 | 13 |

Table 10.3: Overall results of YFD method in the real room, part 1

As it can be seen from tables 10.3 and 10.4 the YFD method showed itself as very efficient and very consistent. The mean values and median values in last two rows of the tables are very close to each other, this confirms the consistency. Mean, median velocity, length of the path show very big improvement over the naive method. And most importantly - time of the exploration mission is 4 times less, than naive's time. This method has shown itself as the most efficient.

The map of the best result for YFD method looks good, but with imperfections. Never the less all 5 obstacles can be singled out from the map.

| № | Time | Mean velocity [cells/s] | Median velocity [cells/s] |
|--------|-------|-------------------------|---------------------------|
| 1 | 01:56 | 29.47 | 6.73 |
| 2 | 01:12 | 79.69 | 12 |
| 3 | 01:27 | 46.58 | 7.89 |
| 4 | 02:05 | 32.36 | 1.79 |
| 5 | 00:35 | 194.38 | 18.08 |
| mean | 01:27 | 76.496 | 9.298 |
| median | 01:27 | 46.58 | 7.89 |

Table 10.4: Overall results of YFD method in the real room, part 2

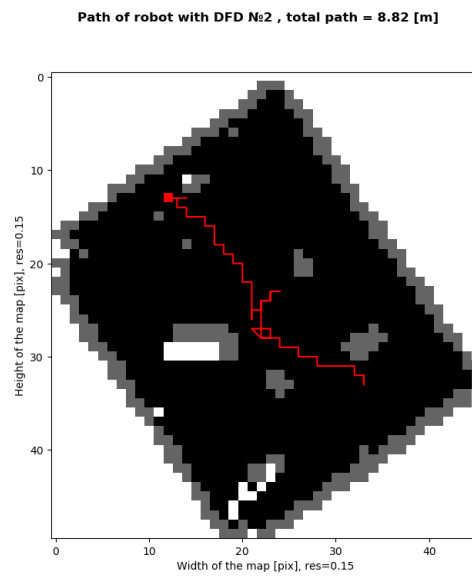


Figure 10.7: Best result for YFD frontier detection, path of the robot

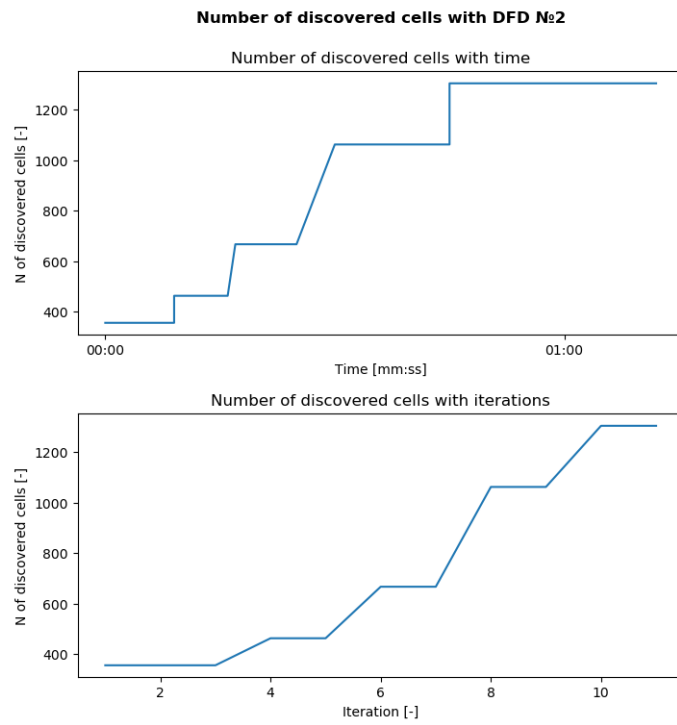


Figure 10.8: Best result for YFD frontier detection, number of discovered cells with time and iterations graph

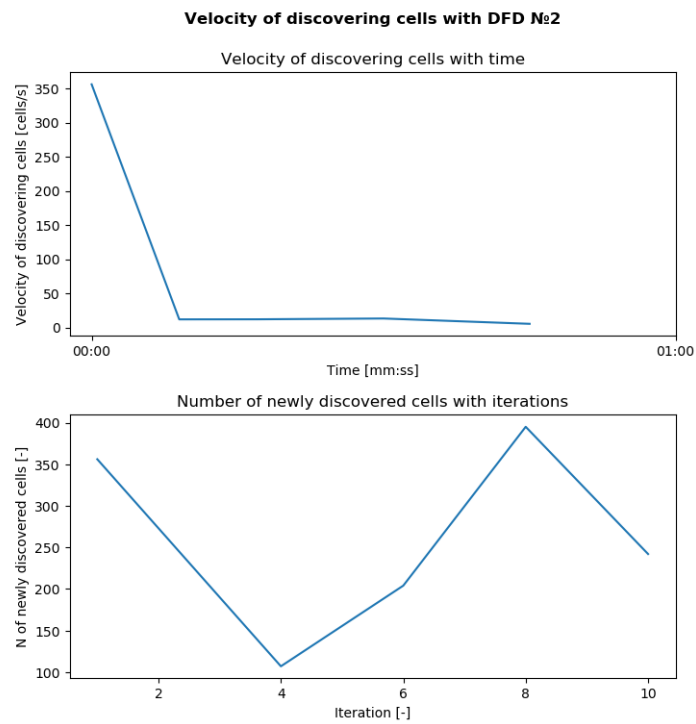


Figure 10.9: Best result for YFD frontier detection, velocity of discovering cells with time and iterations graph

10.3 Exploration with fast frontier detection

| № | Number of opened cells | Length of path [m] | Number of iterations |
|----------|-------------------------------|---------------------------|-----------------------------|
| 1 | 1326 | 8.94 | 12 |
| 2 | 1332 | 12.77 | 24 |
| 3 | 1332 | 11.34 | 17 |
| 4 | 1330 | 12.65 | 12 |
| 5 | 1336 | 16.85 | 17 |
| mean | 1331.2 | 12.51 | 16.4 |
| median | 1332 | 12.65 | 17 |

Table 10.5: Overall results of FFD method in the real room, part 1

As it can be seen from tables 10.5 and 10.6, the FFD method did not show itself as the best on the real map. Its time is almost one minute longer than YFD method, still its two times faster, than naive method. This might not be the fastest method but in terms of opening new cells it has slightly better results.

The problem caused the slowing of FFD method could be the bigger noise, than in simulation. As was mentioned above, the weak spot of this method is noise from LDS, cause it could cause the method to send goal for robot outside the walls. When method is connecting laser readings (more on that in chapter 6.3.), if it connects the reading outside the wall, where unknown cells are present -> frontier would be found in the line.

The graphs 10.11. and 10.12. showing us the consistency as well. In most iterations method finds new cells.

Map, obtained from the method, shown on fig 10.10. is good, almost without imperfections.

| № | Time | Mean velocity [cells/s] | Median velocity [cells/s] |
|----------|-------------|--------------------------------|----------------------------------|
| 1 | 01:01 | 74.06 | 24.35 |
| 2 | 04:45 | 112.65 | 16.16 |
| 3 | 02:53 | 40.8 | 16.13 |
| 4 | 01:23 | 90.35 | 10.87 |
| 5 | 02:21 | 75.78 | 11.47 |
| mean | 02:29 | 78.728 | 15.796 |
| median | 02:21 | 75.78 | 16.13 |

Table 10.6: Overall results of FFD method in the real room, part 2

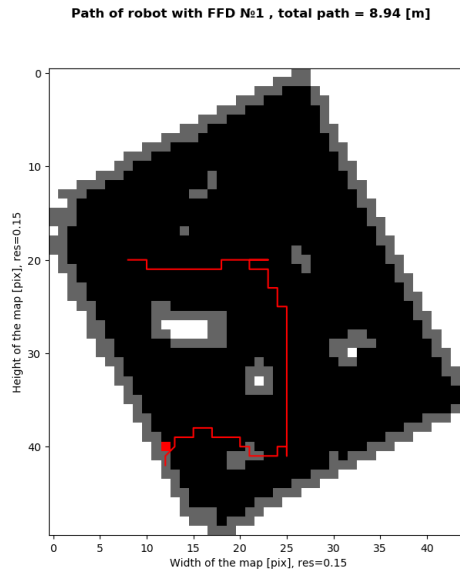


Figure 10.10: Best result for FFD frontier detection, path of the robot

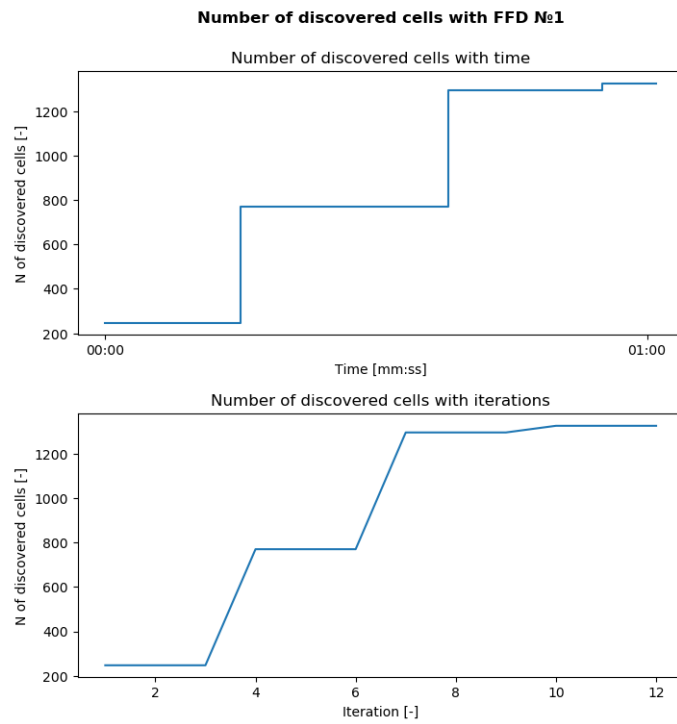


Figure 10.11: Best result for FFD frontier detection, number of discovered cells with time and iterations graph

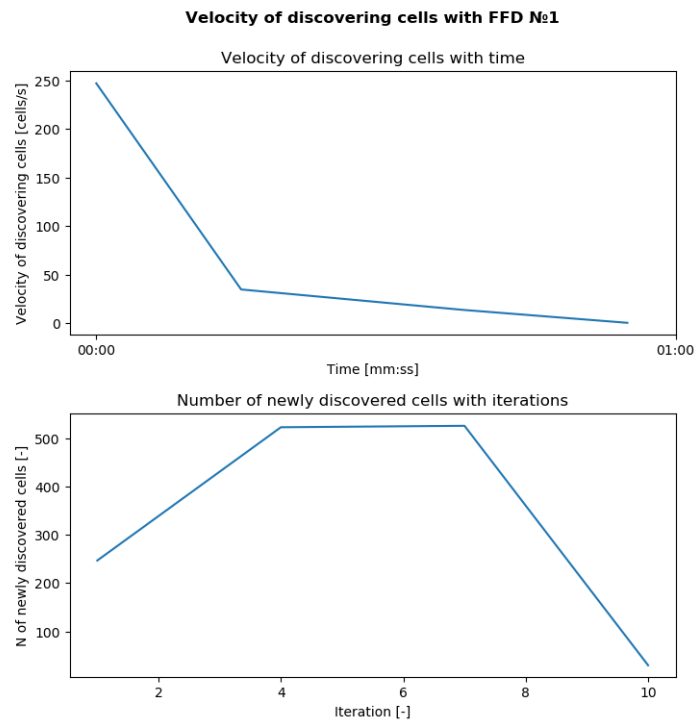


Figure 10.12: Best result for FFD frontier detection, velocity of discovering cells with time and iterations graph

10.4 Overall comparison of methods in real life

Results of testing developed methods on turtlebot3 in real life differ from the results of testing developed methods on turtlebot3 in simulation environment.

Naive method showed itself as working, but very inconsistent method, which confirms tables 10.1 and 10.2 and graphs 10.5, 10.6. It still shows, that occasionally, when point within walls is chosen by the method and it is in perfect position for robot to come and discover half of the map, method would show good results and tends to do it on the small maps, where the probability to chose unknown cell within walls is bigger, then to chose unknown cell outside the walls (because the number of unknown cells inside the walls is bigger, as it is small map, just rectangle). That shows method as inconsistent in results during the exploration mission and quite slow, as the time, that took robot to compose the map with naive method is 2x or 4x slower than other methods.

YFD method showed itself as very fast and very consistent. The number of discovered cells with time and iteration tends to have the linear dependency as it can be seen on the graph 10.8. The time it took to map the room is 4 times faster, than time with naive method. The consistency is not only shown on the graphs, but also in the tables 10.3. and 10.4. The mean value and the median value for each column in those tables are very

close to each other, which indicates the consistency of results through all 5 exploration missions.

FFD method showed different results, from results in simulation environment. Its mean and median time is still two times better than naive's time, but two times worse, than YFD's time. The reason for this was. in my opinion, bigger noise, than in simulation environment. As the noise would show some laser reading from LDS, that the reading is outside the wall, the lines, which connect this laser readings with its neighbours (more on that in the chapter 6.3.) would go outside wall and detect frontiers there, as unknown cells are present in the area outside the walls. Never the less FFD showed good results in consistency, which confirms the tables 10.5 and 10.6, even slightly better in the mean, median velocity of opening cells.

Chapter 11

Conclusion

The task of this masters thesis was:

- To get acquainted with turtlebot3 robot and its functionality
- Research the theme of algorithms for mapping the environment with respect to the Turtlebot3 platform
- Choose at least three suitable algorithms and describe them
- Implement the chosen algorithms for turtlebot3 in simulation environment
- Apply the implemented algorithms on the turtlebot3 in real environment
- Evaluate the results from the simulation environment and from the real environment

The theoretical part began with the description of turtlebot robot, its hardware units and their specification in chapter 2.

Theoretical part of this thesis continued with the description of SLAM, `slam_toolbox` and other algorithms, which would be participating in the autonomous navigation and mapping with turtlebot in the chapter 3. This was followed by the explicit description of the SLAM, `slam_toolbox` and its mathematics in chapter 4. This algorithm was to ensure the mapping of an unknown environment, as the robot would gradually move to the borders of the known map. The moving to the borders of known map was ensured by Frontier Finding algorithms described in the chapter 5.

In the end of the theoretical part three algorithms (naive frontier detection, Yamauchi's frontier detection and fast frontier detection) were chosen and described in the chapter 6.

This was followed by the practical part of this thesis, which began with the description of the developed software implementing the chosen frontier finding methods. The naive frontier detection was described in the chapter 8.1.6, the Implementation of Yamauchi's

frontier detection was described in the chapter 8.2 and implementation of the fast frontier detection method was described in the chapter 8.3.

The implemented algorithms were written, while testing in the simulation environment without requiring any modifications to be made to apply to the real turtlebot and to test them out in the real environment. This was proven by testing the implemented algorithms in the real environment in chapter 10.

Because of the development of software, implementing the methods for autonomous navigation and mapping, stable working package in ROS was developed. Mine package implements all three chosen methods, which are working stable both in simulation environment and in real life. This package could be used for further experiments with different type of SLAM algorithms and different type of frontier finding algorithms. But developing of the package was not my main goal in terms of this thesis. Main goal was to evaluate the performance of chosen algorithms in real life and in simulation environment.

The evaluation of every method's separate results from big map, small map in simulation environment and from real life were conducted in the chapters 9.2.4, 9.2.4 and 10.4 respectively. The testing in the simulation environment was conducted once for every method for both maps, 6 testings in total. Testing in real environment was conducted differently, in that there were 5 measurements of exploration mission for each method. After that, the mean and median values for all measured characteristics were calculated.

The results of testing are as follows: in simulation the FFD method showed itself as the most rapid, accurate and consistent, the next was YFD method. The naive method was the last and least efficient, even though it was capable of constructing the map of small map in simulation environment and almost always was able to construct the map of big map. The real life experiment showed different results. Since the FFD was developed and tested in the simulation environment, it was not designed to interact with big noise from the LDS sensor present in real life testing. The presence of big noise caused FFD to occasionally send goals outside the walls, which slowed down the FFD method and took it to the second place. The YFD appeared to be the most efficient method in real environment. Nevertheless, FFD was still slightly more consistent than YFD and two times faster than the naive method, but two times slower than YFD method.

This concludes the task for this master thesis. All the separate points and the whole task were completed.

All the developed program's code and results of all experiments are also available in the source: https://github.com/kipariss1/warehouse_robot

Bibliography

- [1] “Turtlebot3 documentation.” (2004), [Online]. Available: <https://www.turtlebot.com/> (visited on 04/09/2022).
- [2] “Turtlebot3 documentation.” (2017), [Online]. Available: <https://emanual.robotis.com/docs/en/platform/turtlebot3/features/> (visited on 04/09/2022).
- [3] “Raspberry pi documentation.” (2014), [Online]. Available: <https://www.raspberrypi.org/documentation/usage/gpio/RsEADME.md> (visited on 04/13/2020).
- [4] “Robotis e-shop.” (2007), [Online]. Available: <https://www.robotis.us/dynamixel-x1430-w250-t/> (visited on 04/09/2022).
- [5] X. Sun, Y. Zhang, and J. Chen, “Rtpto: A domain knowledge base for robot task planning,” *Electronics*, vol. 8, no. 10, p. 1105, 2019.
- [6] “Toptal.” (2019), [Online]. Available: <https://www.toptal.com/robotics/introduction-to-robot-operating-system> (visited on 04/20/2022).
- [7] F. Rodríguez Lera, V. Matellán, J. Balsa-Comerón, Guerrero-Higueras, and C. Fernández, “Message encryption in robot operating system: Collateral effects of hardening mobile robots,” *Frontiers in ICT*, vol. 5, Mar. 2018. DOI: 10.3389/fict.2018.00002.
- [8] “What is ros?” (2020), [Online]. Available: <https://roboticsbackend.com/what-is-ros/> (visited on 04/20/2022).
- [9] Wiki. [Online]. Available: <http://wiki.ros.org/actionlib>.
- [10] What is slam algorithm and why slam matters? 2020. [Online]. Available: <https://gisresources.com/what-is-slam-algorithm-and-why-slam-matters/>.
- [11] Tf2 package summary, 2019. [Online]. Available: <http://wiki.ros.org/tf2>.
- [12] T. Foote, “Tf: The transform library,” in *Technologies for Practical Robot Applications*, ser. Open-Source Software workshop, 2013, pp. 1–6. DOI: 10.1109/TePRA.2013.6556373.
- [13] Hadabot blog - a ros2 nav2 navigation tf2 tutorial using turtlesim, 2020. [Online]. Available: <https://blog.hadabot.com/ros2-navigation-tf2-tutorial-using-turtlesim.html>.
- [14] S. Thrun, “Probabilistic robotics,” *Communications of the ACM*, vol. 45, no. 3, pp. 52–57, 2002.
- [15] H. Durrant-Whyte and T. Bailey, “Simultaneous localization and mapping: Part i,” *IEEE robotics & automation magazine*, vol. 13, no. 2, pp. 99–110, 2006.
- [16] E. Olson, J. Leonard, and S. Teller, “Fast iterative optimization of pose graphs with poor initial estimates,” 2006, pp. 2262–2269.

- [17] “Slam (simultaneous localization and mapping).” (2020), [Online]. Available: <https://www.mathworks.com/discovery/slam.html> (visited on 04/20/2022).
- [18] M. Brian Douglas, Understanding slam, 2020. [Online]. Available: https://www.youtube.com/watch?v=saVZtgPyyJQ&t=733s&ab_channel=MATLAB (visited on 05/05/2022).
- [19] M. Isard and A. Blake, “Condensation—conditional density propagation for visual tracking,” International journal of computer vision, vol. 29, no. 1, pp. 5–28, 1998.
- [20] B. Yamauchi, “A frontier-based approach for autonomous exploration,” pp. 146–151, 1997.
- [21] M. Keidar and G. A. Kaminka, “Robot exploration with fast frontier detection: Theory and experiments,” pp. 113–120, 2012.
- [22] ———, “Efficient frontier detection for robot exploration,” The IJRR, vol. 33, no. 2, pp. 215–236, 2014.
- [23] P. Senarathne, D. Wang, Z. Wang, and Q. Chen, “Efficient frontier detection and management for robot exploration,” pp. 114–119, 2013.
- [24] W. Qiao, Z. Fang, and B. Si, “Sample-based frontier detection for autonomous robot exploration,” pp. 1165–1170, 2018.
- [25] P. Quin, D. D. K. Nguyen, T. L. Vu, A. Alempijevic, and G. Paul, “Approaches for efficiently detecting frontier cells in robotics exploration,” Frontiers in Robotics and AI, vol. 8, 2021, ISSN: 2296-9144. DOI: 10.3389/frobt.2021.616470. [Online]. Available: <https://www.frontiersin.org/article/10.3389/frobt.2021.616470>.
- [26] Z. Sun, B. Wu, C.-Z. Xu, S. E. Sarma, J. Yang, and H. Kong, “Frontier detection and reachability analysis for efficient 2d graph-slam based active exploration,” pp. 2051–2058, 2020.
- [27] A. Sears-Collins, How to convert a quaternion into euler angles in python, 2020. [Online]. Available: <https://automaticaddison.com/how-to-convert-a-quaternion-into-euler-angles-in-python/>.
- [28] G. S. Robinson, “Edge detection by compass gradient masks,” vol. 6, no. 5, pp. 492–501, 1977.
- [29] I. Sobel, “An isotropic 3x3 image gradient operator,” Feb. 2014.
- [30] S. Uchida, “Image processing and recognition for biological images,” vol. 55, Apr. 2013. DOI: 10.1111/dgd.12054.
- [31] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to algorithms. 2001, 2009.
- [32] A. Walker, Breadth first search (bfs) algorithm with example, 2022. [Online]. Available: <https://www.guru99.com/breadth-first-search-bfs-graph-example.html>.
- [33] H. Samet and M. Tamminen, “Efficient component labeling of images of arbitrary dimension represented by linear bintrees,” IEEE, vol. 10, no. 4, pp. 579–586, 1988.
- [34] aabecker5, Intro2robotics: Connected components in a binary image, 2017. [Online]. Available: https://www.youtube.com/watch?v=ticZclUYy88&ab_channel=AaronBecker.

- [35] R. Dayala and R. Dayala, Connected-component labeling, 2020. [Online]. Available: <https://cvexplained.wordpress.com/2020/06/17/connected-component-labeling/>.
- [36] C. C. L. is one of available technique used to classify object or region on digital imaging research field (...), Object counting using connected component labelling, 2022. [Online]. Available: <http://www.inforbes.com/2017/06/object-counting-using-connected.html>.
- [37] P. McCrea and P. Baker, "On digital differential analyzer (dda) circle generation for computer graphics," IEEE Transactions on Computers, vol. 24, no. 11, pp. 1109–1110, 1975.