BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

MOBILE ROBOT LOCALIZATION IN INDOOR SCENARIOS

LOKALIZACE POZEMNÍHO ROBOTA VE VNITŘNÍM PROSTŘEDÍ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **TOMÁŠ SÝKORA**
AUTOR PRÁCE

SUPERVISOR **Ing. VÍTĚZSLAV BERAN, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2017

Abstract

The goal of this project was to create a software for an autonomous mobile robot capable of navigation in indoor scenarios and of detection and recognition of the objects on its way. An important condition that the final software had to meet was to find the cheapest solution possible. I achieved the given goal and its conditions using an open source software framework Robotic Operating System (ROS) and its tools. The final system is a set of ROS modules communicating with each other, using a single sensor as an input data stream and a small motor which is able to send information about its velocity and rotations to the control part of the system. This mobile robot can, without any significant problems, navigate through a mapped area while trying to find trained objects. Such a robot could be easily used (after adding specific mechanical parts) to help retired or disabled people, to cooperate with industry workers or in many other fields.

Abstrakt

Cieľom tohto projektu bolo vytvoriť softvér pre autonómneho pozemného robota, ktorý bude schopný navigovať sa vo vnútornom prostredí a detekovať a rozpoznávať objekty na svojej ceste. Dôležitou podmienkou, ktorú musel výsledný softvér spĺňať, bolo nájsť čo najmenej nákladné riešenie. Daný cieľ som splnil s použitím frameworku Robotic Operating System (ROS) a nástrojov, ktoré poskytuje. Výsledný softvér je skupina ROS modulov komunikujúcich medzi sebou, používajúcich jeden hĺbkový senzor ako vstupný dátový prúd a malý motor, ktorý je schopný poskytovať informácie o pohyboch robota do riadiaceho systému. Tento robot sa dokáže bez výrazných problémov navigovať zmapovaným prostredím, zatiaľ čo sa snaží vyhľadávať známe objekty. Podobný robot by mohol pomôcť (po doplnení špecifických mechanických častí) starým alebo hendikepovaným ľuďom, pracovníkom v priemysle či v iných oblastiach.

Keywords

robot navigation, robot localization, objects detection, objects seeking, ROS

Klíčová slova

navigácia robota, lokalizácia robota, detekcia objektov, vyhľadávanie objektov, ROS

Reference

SÝKORA, Tomáš. *Mobile robot localization in indoor scenarios.* Brno, 2017. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Beran Vítězslav.

Mobile robot localization in indoor scenarios

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Ing. Vítězslav Beran, Ph.D. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Tomáš Sýkora
May 17, 2017

Acknowledgements

I would like to thank my supervisor Ing. Vítězslav Beran, Ph.D. for his help and motivation.

Contents

Chapter 1

Introduction

Autonomous and smart machines are an interesting technology area which is being used more and more in many fields. Its potential is undoubtedly huge and we are still extremely far from using any significant percentage of it. Getting these technologies closer to the ordinary people and people in need is what I was trying to achieve with my work. The result is an open source software which can, placed to a mobile robot, autonomously localize itself within a room, navigate through it and seek known objects within it. This can be, with additional mechanical equipment, used to fulfill many different tasks to make human life easier.

An important fact here is the amount of used resources. While the common robots developed by researchers can cost thousands of dollars, the components needed by the robot system described in this bachelor thesis will not be more expensive than several hundreds. Because of the simpler equipment I had to choose such algorithms and techniques which would not be very compute intensive. With these basic capabilities such as the sense of direction and vision and with its low-cost requirements, many people could find this platform helpful in lots of different areas. It can be used as a corner stone of bigger projects (e.g. after installing a mechanical arm on the robot, so it could manipulate with found objects).

The initial idea of this project was to bring up a robot which can determine where it is according to a saved map (it can localize itself) and can move from this location to a given destination on the map (can navigate itself through the map). Further, an interesting extensions were added to the original idea. That means abilities to explore indoor areas, detect objects on its way and recognize the known (trained) ones.

This work describes how to bring up a robot software to fulfill the mentioned tasks using mostly existing tools. It is divided into three main chapters. The chapter 2 is a short theoretical introduction to robotics and computer vision techniques I used. The chapter 3 explains what parts does the whole system consists of and what is the job of each individual module. The last chapter 4 deals with the final system implementation and its realization. That means configuration of the existing packages and the implementation of my packages are described there. This chapter also presents actual results of the implemented software.

Chapter 2

Robotics and computer vision

Basic autonomous robot abilities are knowing how the world around the robot looks like so the robot can move through it and the ability to manipulate with objects within it. To do that the robot has to *see* its surroundings to be able to create a world model. To create the world model the robot takes different types of input data which are processed later and useful information is obtained from it. Besides the vision part robots also has to be able to control its movements to navigate itself to a given area and manipulate with objects. Because robotics and computer vision are huge areas with great amounts of different techniques, methodologies and approaches to solve the mentioned tasks, this chapter presents a brief overview of them, especially the ones I used in this project.

2.1 Mobile robot approaches

Mobile robots (or autonomous navigation systems in general) have found their place in many different areas such as automation of industry, autonomous cars, army or service robots helping humans in complex environments. However the result of navigation is always similar (moving through the area to a given destination) robots use different approaches depending on possibilities of the specific environment. That means specific methodologies are used in specific situations. For example the methodologies for navigation in indoor scenarios have to be different from the ones in outdoor scenarios, although this is just a broad division. Some outdoor areas have more difficult terrain than others, suitable techniques have to be chosen for every concrete assignment.

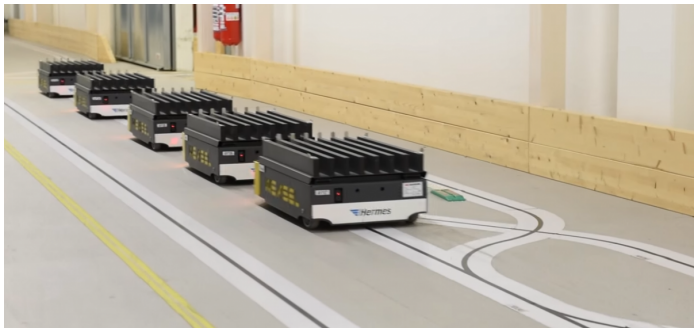
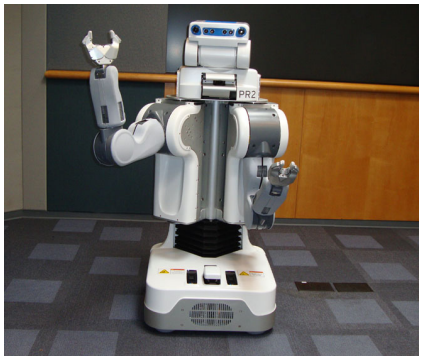


Figure 2.1: Automated Guided Vehicles (AGV) using printed lines on the floor to navigate in indoor scenarios.

An interesting approach to navigate the robot in outdoor areas is described in [5]. In this work specific characteristics of the terrain are taken into account. The existence of several regions with different navigation costs are considered. The costs are determined experimentally by navigating the robot through the regions and measuring the influence of the terrain on its motion. The authors measured the robot vertical acceleration, which reflected the terrain roughness.

Navigation in indoor scenarios is slightly different. There is usually no rough terrain, just a straight floor plane. Knowing that the terrain is a simple floor makes navigation a bit easier. In the industry (but also in other areas) mobile robots often use various kinds of markers which define the robot's path. Markers can be lines on the floor, wires, buried inductive loops, magnets or different indicators. These robots are called Automated Guided Vehicles (AGVs). An example can be seen in the figure 2.1.

Different approaches in navigation problematics use depth data which allows them to measure distances between obstacles and to build maps of the environment. Having a map robots can localize themselves within it. I used this approach in this work as it results in probably the most autonomous behavior as no markers are required. There are lots of robots from different research groups capable of solving vision and navigation tasks using depth data. They are usually equipped with combinations of several types of sensors, such as cameras, depth sensors and lidar scanners and also with other mechanical parts. Various softwares can be run on these robotic platforms, including the one from this bachelor thesis.



(a) PR2.

(b) Care-o-bot-4.

(c) Turtlebot.

Figure 2.2: Robots examples.

A well known robot from this area is PR2[1] which is quite an expensive robot used mostly in research areas. This robot is equipped with several lidar scanners, depth sensors

[1] http://www.willowgarage.com/pages/pr2/overview

and cameras. It also has two arms with lots of joints so it is capable of very interesting things. Sadly, having so many devices it is an opposite of the low-cost robot that I was trying to build. Another interesting robot meant to be used as a servant in ordinary human environments such as households or hospitals is Care-O-Bot[2]. Although it was meant to be used by ordinary people not just by researchers it is still not affordable for everyone and its cost is several thousand dollars. The most similar solution to the one I was trying to achieve is Turtlebot[3]. It is an open source robotic platform based on ROS framework using just one depth sensor and although it is very similar to my solution, its software uses slightly different tools than the one I built. Mentioned robots can be seen in the figure 2.2.

2.2 Depth data

One of the commonly used input data types in robotics and computer vision are depth data. Its added value in comparison with standard video data is that while a video stores just information about the 2D scene (a column and a row of the pixel with its color), the depth data or point clouds consist of 3D points (column, row and distance of the point from the sensor). That means although the depth data are more memory and computational power consuming they offer greater possibilities in solving computer vision and robotics tasks such as navigation, filtration, segmentation, objects detection and others.

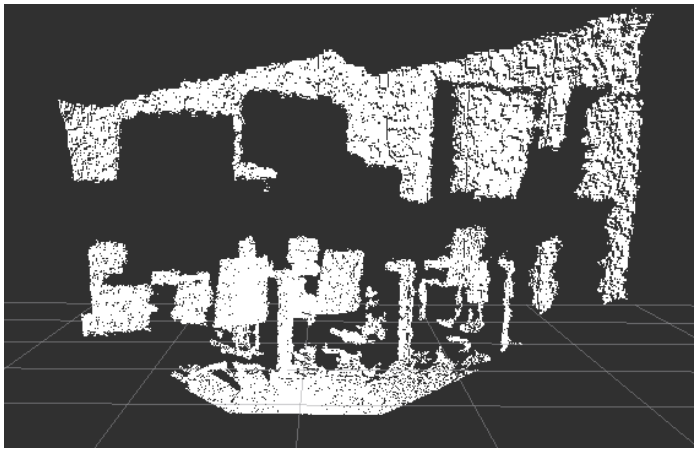


Figure 2.3: An example of a point cloud captured with an ASUS Xtion sensor.

Processing depth data

A point cloud from a depth sensor is usually processed by several modules which perform different operations with the received point clouds. A typical point cloud processing pipeline consists of severals steps depending on a specific goal which is being accomplished. The first step is collecting the depth data from a sensor. There are lots of kinds of depth sensors. They differ in quality of their data, resolution, frequency of published frames, format of the published data, price and in other parameters. A typical and well known sensor is a Kinect from Microsoft or an ASUS Xtion which I used to run my software with. The mentioned sensors capture the whole scene with a specific resolution with a number of

[2] http://www.care-o-bot-4.de/
[3] http://www.turtlebot.com/

rows and columns, where every point provides also a depth information. Other types of depth sensors are lidar scanners that do not create the whole scene point clouds. They create horizontal scans of its surroundings, typically with a higher frequency than Kinect like sensors. Their product is one or more rows containing depth points saying how far the nearest obstacles are in a certain height.

After receiving point clouds from a sensor, data are usually not suitable for immediate processing by complex algorithms (e.g. because of the noise and outlier points), the preprocessing part must come before that. Preprocessing usually means removing points that will not be used in the next steps of the pipeline or they could make these next steps more difficult to accomplish. They are usually noise or lie too far from the sensor or there are other reasons depending on a specific task. Another typical step in point cloud processing is segmentation. It divides the points of the point cloud into several clusters each representing a real object of the scene or they are part of a connected area. Created clusters can be used in other processing parts such as object recognition.

Point Cloud Library

A powerful library for processing depth data is a Point Cloud Library (PCL)[4]. Using C++ language it implements plenty of algorithms solving every part of the point cloud processing such as I/O operations, filtration, segmentation, recognition, registration and many others. It also provides tools for visualizing point cloud data. As ROS (a robotic framework I used, it is explained in the section 2.5) packages can be implemented in both python and C++, PCL is a great choice. A brief introduction to PCL tools can be found in [7].

2.3 Segmentation

Before further processing and analysis of an image or a point cloud it is usually divided into multiple segments. Each segment should represent a group of pixels or points which are somehow connected to each other. The process is called segmentation and it can segment areas of an image with the same color, points of the point cloud belonging to the same real world object or find a certain model in the image/point cloud such as a plane etc.

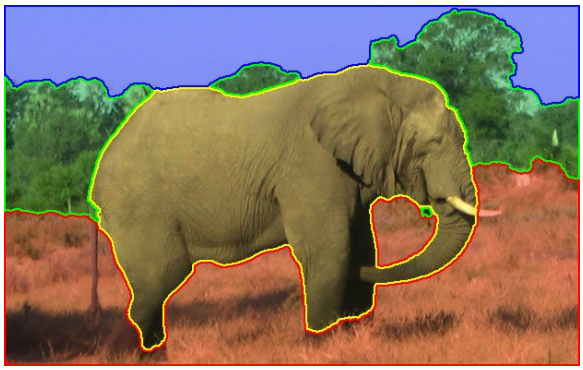


Figure 2.4: Result of segmentation.

Segmented areas are usually used in the following image analysis and other computations (e.g. detection). An example of the segmentation can be seen in the figure 2.4, with four

[4] http://pointclouds.org/

areas segmented. As the segmentation is a big part of this work (the finding/detecting objects problem is solved with segmentation techniques), some of the segmentation methods will be described in this section.

Random sample consensus

Random sample consensus (RANSAC) is a method for solving a problem of fitting a model to experimental data [2]. It is capable of interpreting/smoothing data containing a significant percentage of gross errors, and is thus ideally suited for applications in automated image analysis where interpretation is based on the data provided by error-prone feature detectors. In other words RANSAC tries to determine which of the given points are part of a certain model such as a line, a plane or more complicated ones. In the figure 2.5, RANSAC tries to choose points lying in circles (with a given tolerance/threshold).

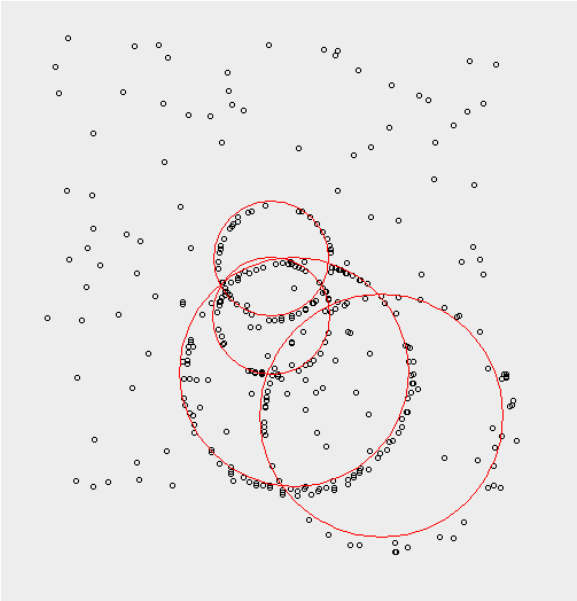


Figure 2.5: The RANSAC method tries to find points lying in a circle.

The algorithm behind this randomly selects the minimal subset of points to define the searched model (e.g. three points in the case of a circle, two points for a line) and estimates parameters for this model according to the selected points. Then the algorithm finds out how many of the points from the whole set fit the defined model. If the number is large enough the fit is accepted. If not, the process of selecting the subset of points and estimating parameters is repeated until a suited solution is found or a given number of iterations finished (then it is considered as a failure). Although the RANSAC technique is not usually considered as a segmentation technique it can be used to segment a model such as a plane which was the use case in this project.

Region growing

The purpose of the region growing segmentation algorithm is to merge the points of the point cloud that are close enough in terms of the smoothness constraint. Thereby, the output of this algorithm is a set of clusters, were each cluster is a set of points that are considered to be a part of the same smooth surface. The work of this algorithm is based on

the comparison of the angles between the points normals. The algorithm works as follows: First of all it sorts the points by their curvature value. It needs to be done because the region begins its growth from the point that has minimum curvature value. The reason for this is that the point with the minimum curvature is located in the flat area (growth from the flattest area allows to reduce the total number of segments). Having the sorted cloud, algorithm picks up the point with a minimum curvature value and starts the growth of the region. It is repeated until there are no unlabeled points in the cloud. More about the region growing algorithm can be found in [4].

2.4 Navigation prerequisites

To navigate a robot it has to know its position at first. It means that the robot has to be able to process visual input data to find itself in the world scene and to use this information to find an appropriate path to move to a specific destination. In this project I used several computer vision (robotics in general) techniques and algorithms to solve the navigation and localization problems. They will be described in this chapter. One of them is SLAM (Simultaneous Localization And Mapping) for a map creation, the other one is a Monte Carlo localization algorithm for localizing the robot in the map.

Simultaneous localization and mapping

Simultaneous localization and mapping (SLAM) solves a problem of acquiring a spatial map of a mobile robot environment while simultaneously localizing the robot relative to this model [10]. Formally, the SLAM problem is best described in a probabilistic terminology. Let's assume we have a set of robot's positions (two-dimensional coordinates of the robot's position in the map and its orientation), a set of odometry values such that from one saved odometry value the past position of the robot can be recovered and a set of measurement values used to determine the relationship between one position value and the features stored in the map (a map feature describes how the map looks like in the corresponding place).

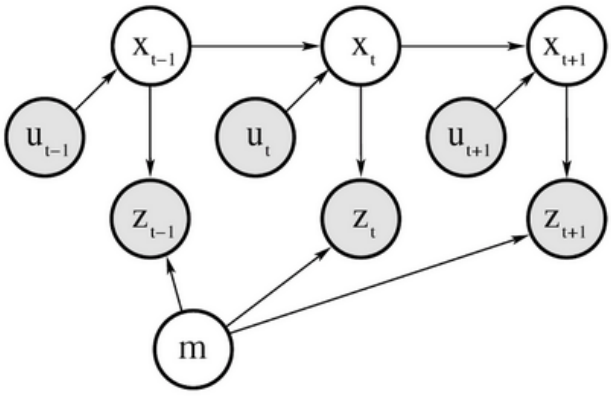


Figure 2.6: Model of the SLAM problem. x values represent robot's positions, u values represent odometry information, z values are measurements determining the relationship between the position value and features stored in the map and m stands for the map data. Shaded nodes are directly observable to the robot. In SLAM the robot seeks to recover the unobservable variables.

Relationships between these data sets are shown in the figure 2.6. The SLAM problem is now a problem of recovering a model of the world map (the map features) from the odometry and measurements data. In this project I used SLAM methods just for building a static map which is used later for localization and navigation where SLAM is not used anymore for its high computational requirements.

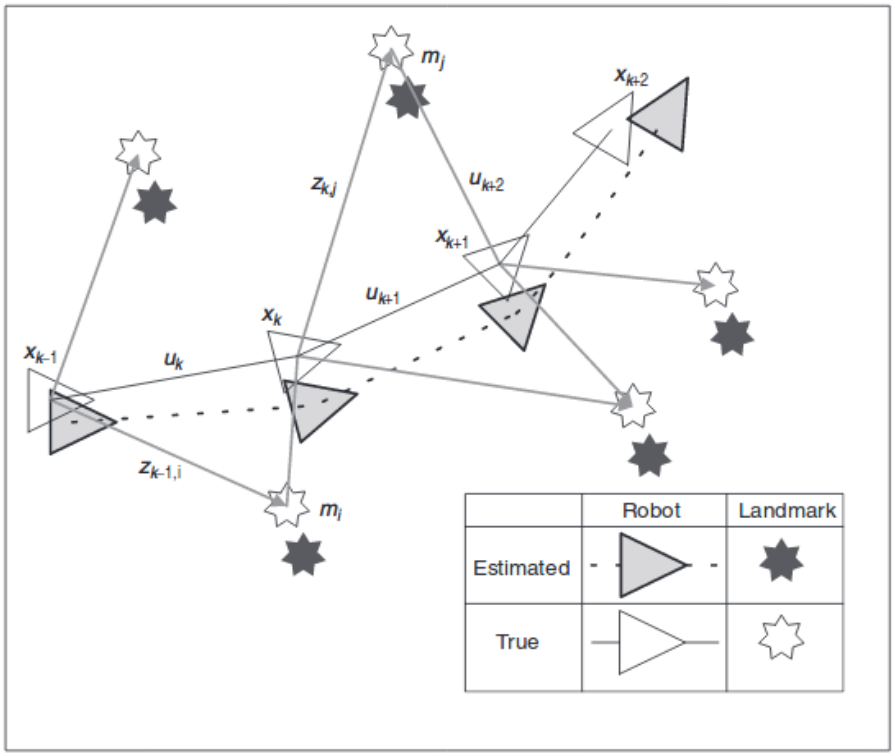


Figure 2.7: The essential SLAM problem. A simultaneous estimate of both robot and landmark location is required. The true locations are never known or measured directly. Observations are made between true robot and landmark locations. [1]

Monte Carlo localization

Monte Carlo localization is also known as a particle filter localization. Obviously it is an algorithm solving the localization problem using particle filters. The basic idea of the particle filters is that the robot has a map and through a sensor it sees the scene around itself (objects, walls...). With this information possible positions of the robot in the map and likelihoods of these positions are computed. As the robot moves, the data from the sensor are slightly changed and the new likelihoods of those positions are computed according to how the new data from the sensor match the new position in the map. After a few steps, robot should, with some likelihood, know were it is within the saved static map. Detailed mathematical explanations of Monte Carlo localization and particle filters can be found in [9].

2.5 Robotic Operating System

A typical robot consists of multiple hardware parts such as sensors and motors. Each of them works with different data representations. Data received from a sensor can affect actions taken by the motor and so on. To do that, the robot parts have to communicate with each other. To make the manipulation with robots easier various tools are available. One o them is Robotic Operating System (ROS)[5]. ROS is an open source framework offering system environment and hardware interfaces for packages implementing solutions for different robotics tasks. ROS' work is to make manipulation with robot's hardware parts as easy and effective as possible. It creates an abstract layer above the robot hardware so the programmer does not have to care whether he or she works with this or that kind of a sensor. The sensor can be just plugged to the computer (e.g. via USB cable), then a corresponding ROS node (module) processes its data.

ROS nodes are running processes, each solving its own task of the system and performing some computations. A running ROS system can manipulate with many nodes simultaneously. It provides communication methods for them and offers tools to control them. The nodes usually communicate with each other using messages. The basic communication consists of a publisher and a subscriber. The publisher is a node which publishes data (messages) on a topic which is a defined channel or a stream for sending certain types of messages. An example of such a publisher can be a video camera publishing video data. The subscriber is a node which can receive data from a publisher. The process is called listening to a topic. If the subscriber listens to a topic it can read data from it and do specific operations with them. It can send the new processed data to another topic where it will be available for other nodes listening to this topic.

Besides the mentioned messages and topics ROS also provides a server-client kinds of communication methods. One of them is an actionlib server and client communication. The actionlib server is a node which continuously works on some task which can be interrupted by the client. The server changes (or not) its behavior accordingly to a specific command received from the client.

Several visualization and simulation tools can be also found among the ROS tools. The most interesting and useful ones are *rviz* and *gazebo*. *Rviz* is used mostly to visualize data from the published topics which is sometimes necessary when one wants to check whether the incoming data looks correct. *Gazebo* is a tool to simulate the robot and the environment around it so the programmer does not have to work with an actual robot (which can be big or expensive or just difficult to manipulate with) just with the simulated one in the *gazebo* simulator.

All ROS tools provided by some open source packages are configurable with their launch files. Launch files are xml files containing specific parameters for the nodes to be run with. A substantial part of this work consists of such launch files. More about ROS framework and its tools can be found in [3].

[5] http://www.ros.org/

Chapter 3

Design of a mobile robot

The whole system consists of several modules, each performing a different task of the data processing pipeline. In this chapter I will describe how the final system architecture looks like, what parts does it consist of and what modules and algorithms were used to fulfill the assignment.

3.1 Assignment analysis

Let's assume we have a situation in which a disabled man on a wheelchair drops an object which falls some distance from him. As moving the wheelchair to the object and bending down to it could be a difficult task for the disabled man, a robot capable of finding the object and bringing it to the man would be definitely useful. Another possible use case could be a factory worker working on some manual task. A robot could save him some time by bringing him tools he asks for. To fulfill such tasks the robot has to be able to do following subtasks:

- localize itself in the area (knowing where is the position of a user, the position of an object and the position of the robot itself in the map)
- navigate through the area (exploring the area, moving between the found objects and the user)
- detect objects (selecting potentially known objects from received point clouds)
- recognize found objects (learning which object is which)
- speech recognition (communication with the user, this part is not solved by this work)
- audio localization (to localize speakers from their voices, this part is also not involved in this work)

The initial assignment was to bring up a robot capable of localization and navigation in indoor scenarios using existing methods, techniques and tools. An extension was added to the initial assignment later. Robot's navigation skills should be used to explore its surrounding areas and to find certain objects lying on the ground within the area. In the localization and navigation part of the assignment my task was to choose suitable ROS packages solving given problems, design a complex system built on them and make the individual nodes work with each other. In the extended part of the assignment (detection

and recognition) I had to implement a package detecting objects in the view field of the sensor and bring up a specific existing ROS package for image recognition (it had to be a package developed by a certain research group from another university). The last part was to connect these modules with each other using a suitable exploration package so it could compute a path of the robot and send navigation goals to the navigation module.

As the target users group may be disabled people and common households, the added value of the final software solution must be the low-budget hardware requirements. Used devices should be as cheap and simple as possible. That means robot equipped just with one depth sensor and a simple mobile base with a motor. All needed input data must be obtained from these two devices.

System architecture

I designed a system architecture consisting of several modules. The whole system schema is displayed in the figure 3.1. It shows how the input data from the depth sensor and the mobile base flow through the individual nodes. One data stream is published by the sensor. Data from it are received by an object detector module and a module creating laser scan data which are used by a map creation module and localization and navigation modules. These modules process received data and send new data to other nodes in the chain. The figure also shows how the navigation module receives laser scan data, map data and mobile base odometry data and sends new data (velocity commands) back to the mobile base. All of these parts will be described in the following sections.

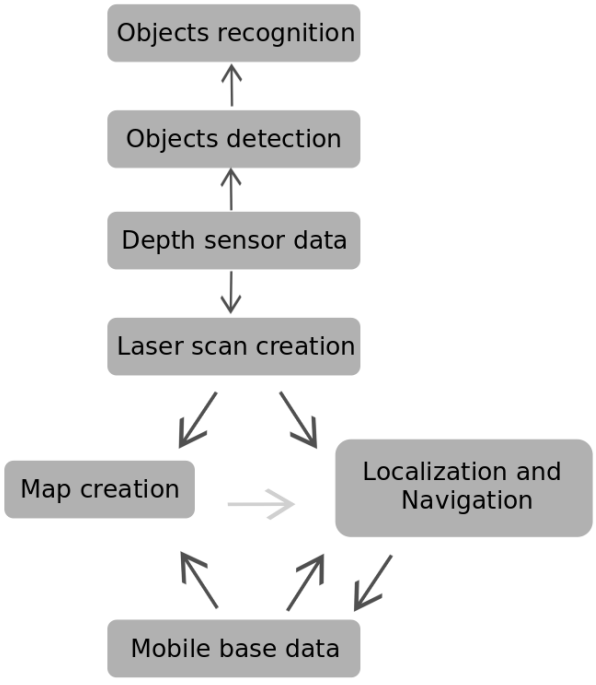


Figure 3.1: All parts of the system and how the input data from the sensor and the mobile base flow through other nodes are shown in this schema.

3.2 Building a static map using SLAM

As I mentioned earlier this solution has to use algorithms and techniques with low computational requirements. Although the SLAM methodology is capable of solving the whole localization problem including building a map, it would be compute intensive. That is why I chose a different approach to solve this problem. SLAM is used only to build an initial static map which is later used for localization and navigation. The localization and navigation parts are fulfilled by other techniques and SLAM is used no more in the system. This implies that building a static map is a prerequisite for the later functioning of the system.

Laser scan

Before building a map, input data streams must be handled (specifically laser scan and odometry data are required by the map building module) and a proper robot setup is needed. As the used robot is equipped just with a depth sensor there must be a module which can receive a point cloud and create laser scan data from it. The laser scan is a horizontal scan of the area. It can be obtained by selecting depth points with a certain height in the point cloud which can be seen in the figure 3.2. They are then reordered to a different data structure (a laser scan) which stores data in a single array (e.g. one index of the array for one column of the point cloud).

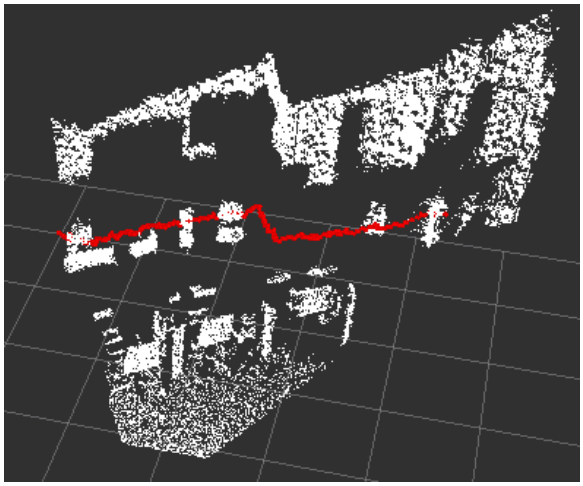


Figure 3.2: A laser scan (red) retrieved from a point cloud.

Transformations between coordinate frames

The second required input data type is odometry, which is provided by the packages controlling the mobile base. It gives information about the robot movements (e.g. velocities, rotations...) to the system. Having a fake laser scan data and odometry information, there is one last thing to do before starting building a map. Every data publishing component in the ROS system publishes them in its own coordinate system or frame. ROS system has to know the relative position of each of these frames so their real spatial relationship is obvious to other running nodes.

There are nodes in the ROS that can transform data from one coordinate frame to another. This operation is called transformation between the frames. An example of

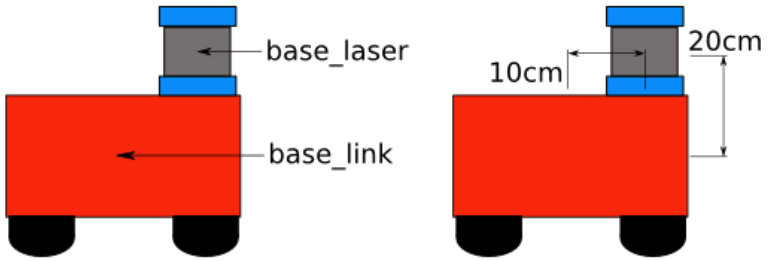


Figure 3.3: Data published by devices like a sensor or a motor have their own coordinate systems. Transformations between them are needed so the other nodes in the running ROS system know the relationships between the positions of these devices.

possible robot's frames is shown in the figure 3.3 where the transformation node would require information about the base_laser frame being 20cm above and 10cm in front of the base_link frame. In case of my system transformations between the depth sensor and the mobile base frames are needed. With laser scan data, odometry data from the mobile base and the transformations between them, the map of the environment can be created. While building the map the robot is not autonomous. The map building module takes odometry and laser scan data as an input. Thus, the user has to move with the robot manually and take it (using a keyboard or a joystick) around the room(s). The result is an image with pixels having three values: free, occupied and unknown. An example can be seen in the figure 3.4.

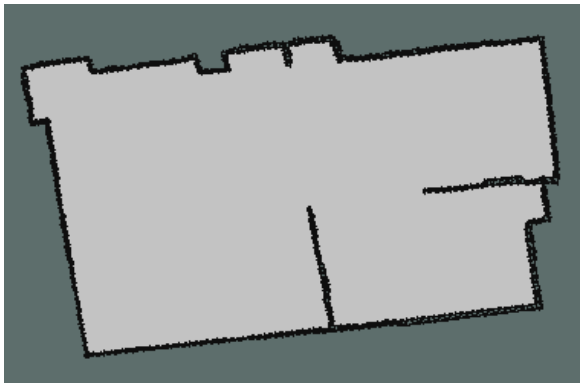


Figure 3.4: Example of a static map. Black pixels stand for occupied area, light gray for free and dark gray for unknown area.

3.3 Localization and navigation

Before the system is able to navigate itself to a given destination using a map it has to determine, where it is within the given map. In other words, the system has to be able to localize itself within the map. The solution described in this work uses fast particle filters implementation and sensor models. The principle of the particle filters methods was described in the section 2.4. The localization is accomplished by a single node that subscribes to the laser scan and map topics. It tries to match an incoming laser scan to a given map. As a result the module publishes the robot's pose.

The navigation techniques I used are based on dividing the navigation problem into two parts, specifically local and global planning of the path. Global planning means planning the path from the initial position to the final destination avoiding the obstacles on the saved static map. The local planner tries to avoid nearest obstacles (also the ones that are not in the static map) while keeping the robot as close to the original global path as possible.

To achieve this both global and local planners create supplementary maps called costmaps. The difference between a normal map created in the previous section and a costmap is that the costmap assigns one of many (e.g. 255) values to a pixel instead of three values for free, occupied and unknown. That provides an information about the distance of the specific point in the map to a nearest obstacle. This is used by the path planners to find the best path considering a given configuration which specifies how the values are assigned to points of the costmap. A costmap based on the static map from the previous section is shown in the figure 3.5.

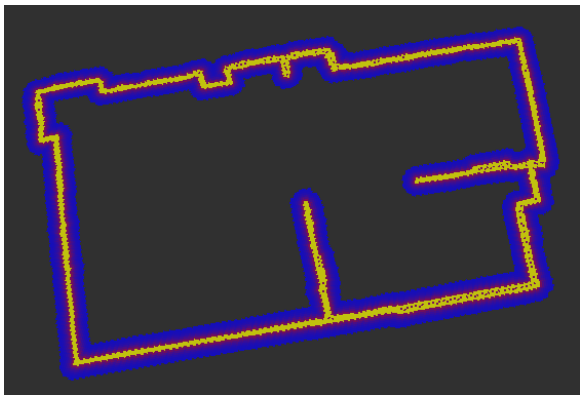


Figure 3.5: Layers of a costmap.

Having all the necessary input data the navigation module can receive navigation goal messages and publish velocity messages to the mobile base which results in the robot moving to the given destination.

3.4 Objects detection

While the robot is moving through the area it simultaneously detects objects in its field of view. The detection is done by a module which was implemented by myself. It receives point clouds from the sensor. In every received point cloud it removes the floor plane so the points remaining in the point cloud belong to the detected objects (or just an empty point cloud remains, which indicates no objects are in the robot's view field). Elementary steps of the detection algorithm are shown in the figure 3.6.

As the floor removal is a frequently performed operation it has to be done effectively. The floor plane segmentation in every received point cloud using a complicated algorithm could be compute intensive. That is why I used the following optimization: The segmentation of the floor plane is required just after some longer time intervals (assuming that the sensor angle will not change very often). After each plane segmentation is done the angle between the sensor and the floor plane is computed. The angle computation uses the 3D plane coefficients a, b, c obtained by the segmentation. As the coefficients are values of the

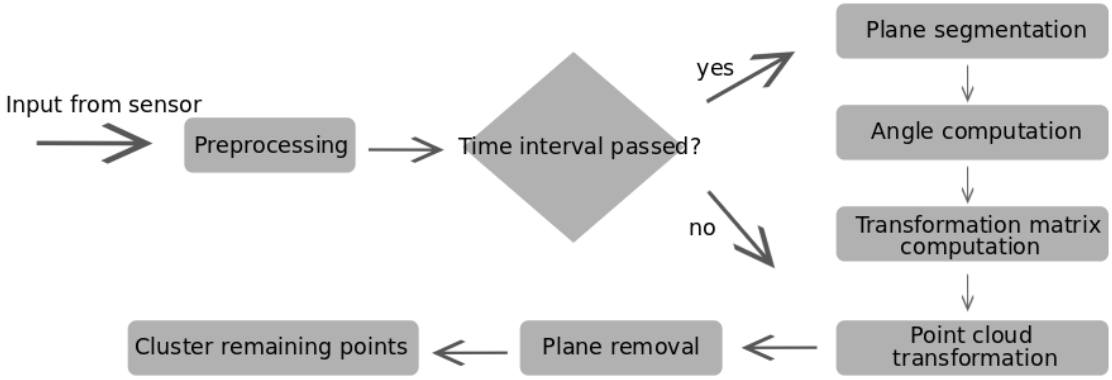


Figure 3.6: A schema showing steps of the objects detector algorithm.

normal vector of the plane the angle can be obtained using the equation 3.1. Where the computed angle is the angle between the floor plane normal and the xy axis plane normal.

$$\cos\theta = \frac{a_1 * a_2 + b_1 * b_2 + c_1 * c_2}{\sqrt{a_1^2 + b_1^2 + c_1^2} + \sqrt{a_2^2 + b_2^2 + c_2^2}} \tag{3.1}$$

Using this angle every received point cloud can be rotated so the floor plane in the rotated point cloud is parallel to the xz coordinate axis plane. Knowing the distance between the floor plane and the parallel xz plane, all points with a certain y coordinate value can be removed. The equation 3.2 computes the exact y coordinate value.

$$y_{new} = y_{original} * \cos\theta \tag{3.2}$$

As the transformation is just a multiplication of all points by some value and the floor removal is just a simple filter removing points with certain values, this is definitely more efficient solution than performing a segmentation with every received point cloud. The last thing to do is to assign the remaining points in the output point cloud to individual clusters, each cluster representing a corresponding real world object. Specific segmentation methods which I used are described in the following chapter. The results of the described algorithm are demonstrated by the figure 3.7.

3.5 Objects recognition

The objects recognition module uses a neural network to recognize and classify the trained objects. Specifically a type of a convolutional neural network is used in this work. Its exact architecture is specified in [8]. As training a neural network needs a big data set, this neural network is trained on an academic data set *ImageNet*[1]. The user of the robot system has to only retrain the top layers, the rest of the network remains untouched. The pretraining

[1] http://image-net.org/

Figure 3.7: Objects detection is done by removing the floor plane points from every received point cloud. Remaining points are points belonging to the detected objects. Segmentation algorithm assigns these points to clusters representing the real detected objects.

of the network makes things much more easier as the user now needs just several images of the objects to be recognized by the robot.

With a trained neural network the module can start receiving images of the detected objects from the object detector node. The recognition module returns likelihoods of the possible classes which were assigned to the detected object. The class with the best likelihood wins or the detected object is considered to be an unknown object. Implementation details about the process of training and using the neural network module are described in the next chapter.

Exploration

During the detection and recognition fazes the exploration module sends navigation goals to the navigation module which tries to accomplish them. The module controlling detection and recognition also controls the exploration by sending messages about whether to continue with exploring (when there are no objects detected) or to stop for a moment (while the detected objects are recognized). If a known object is found the detection module stops the exploration and publishes a navigation goal message which results in the robot moving to the recognized object.

Chapter 4

Objects finder implementation

To run and implement the mentioned modules I used the Robotic Operating System and the Point Cloud Library. Existing packages and also packages implemented by myself will be described in this chapter, together with their implementation, configuration and usage.

4.1 Robot setup

I used a very simple robot to work with. It consists of several hardware components which need to be configured properly. This section presents a description of how to bring up these robot's hardware components and how to connect them with ROS to allow other nodes to receive required input data and send them commands.

Mobile base

As a robot mobile base I used *kobuki*[1] which is a remodeled vacuum cleaner (with the cleaning part removed). It provides a motor, wheels and several ports to be connected with a computer or other devices (e.g. Raspberry Pi).

Figure 4.1: My robot for the test purposes, constructed from a *kobuki* mobile base and an ASUS Xtion depth sensor.

[1] http://kobuki.yujinrobot.com/

The research group which developed *kobuki* also implemented ROS packages to control it. That means ROS nodes publishing odomentry data and subscribing to velocity topics so other nodes can send signals to move with the *kobuki* mobile base. Specifically, I used two nodes to control *kobuki*. The first one is *kobuki_node*. It is the main interface between ROS and *kobuki's* hardware. It subscribes to velocity topics and publishes data on an odometry topic. I run it with its default launch file *minimal.launch* which is installed with the package. Another *kobuki* ROS node I used (while building the map) is *kobuki_keyop*. After launching it user can control *kobuki* with a keyboard. I also used its default launch file *keyop.launch*. No special customizations of launch files were needed to bring up this part.

Depth sensor

As a depth sensor I used an ASUS Xtion device. It provides depth and RGB data. A ROS package *openni2_launch* provides an interface between the sensor and ROS and it publishes point cloud data into the system. It is important to use the version 2 as the version 1 does not work with this device. This package can be launched with its default launch file *openni2.launch*. I only lowered the publishing frequency so the computational requirements are lower. This can be done because the recognition module is not so fast so there is no point in publishing data with a high frequency. In this part transformations between the frames are done. A ROS package *tf* takes care of this via its node *static_transform_publisher*. It publishes transformations between the *base_footprint* frame which is the frame in which *kobuki* publishes odometry and the *camera_link* frame which is the frame of the sensor. The exact values (distances) in meters are set in *static_transform_publisher* launch file. With these transformations published, the system knows what is the relationship between the position of the sensor and the position of the mobile base.

Fake laser scan

The last part of setting up the robot is creation of the fake laser scan data. There are two ROS packages available capable of fulfilling this task, each of them has its own way of achieving it. The first is *pointcloud_to_laserscan*. As the name indicates it creates the laser scan data from a received point cloud. It iterates through all depth points of the point cloud and selects a point with the closest distance from every column of the cloud.

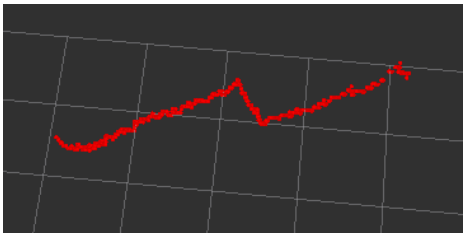


Figure 4.2: The fake laser scan data visualized in *rviz*.

The result is an array of the selected points - the fake laser scan. It is visualized in the figure 4.2. The other ROS package creating laser scans is *depthimage_to_laserscan*. The first difference between the previous package and this one is that *depthimage_to_laserscan* does not use point clouds to create laser scans. It takes a depth image as an input. A depth image is a grayscale image where pixels with the same distance from the sensor have the

same shade of gray color. The *depthimage_to_laserscan* package does not iterate through the whole image instead it takes one row (or more, depending on the configuration) of the image with its distance values and that is the output. I chose the second package as its output is more similar to the one a lidar scanner would give. Another advantage of the *depthimage_to_laserscan* package is that after a little code customization the specific row can be selected. It would be useful in a situation, when the angle between the sensor and the floor would not be zero. In such case we do not want to take the middle (default) row of an image as a laser scan. Instead some of the higher rows would be better, depending on the sensor angle.

4.2 Building a map using SLAM

Before the system can start with the localization a static map has to be prepared. Building the map is done by a ROS package *gmapping* which implements SLAM algorithms. This package requires laser scan and odometry data which are provided by the packages described in the previous sections. The node is launched with a launch file containing required parameters. The most important step here is to set the correct names of the topics publishing the data and the frame names they are published in. Specifically it is the scan topic name - „scan“, mobile base frame - „base_footprint“ and odometry frame - „odom“. After launching the configured package the user has to manually walk the robot through the area using keyboard. It can be done with the *kobuki_keyop* package. The process of building a map can be seen in the figure 4.3.

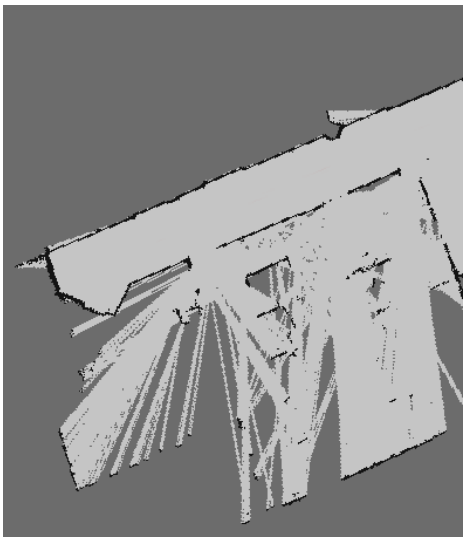


Figure 4.3: The process of building a map using a *gmapping* package. The black pixels mean the area is occupied, the light gray pixels mean it is a free space, the rest is an unknown area.

The map created with the *gmapping* package is usually not perfect, especially when using fake laser scan instead of a real lidar scanner. Also the odometry is not always a hundred percent accurate because the low-cost mobile base motor and its wheels does not work with such an accuracy. These can result in irregular or broken walls or other objects in the map. It can be solved by editing the map a little bit in an image editor. Although it is not necessary, it helps later in the localization part.

4.3 Localization

I tried two approaches while solving the localization part of the problem. The first one was a package *amcl* (which stands for adaptive Monte Carlo localization) which is a widely used tool among the ROS community. The *turtlebot* robot mentioned in the section 2.1 also uses this package to localize itself. The other approach I tried uses a module which is part of a bigger project solving robotics tasks called *Environment Descriptor*. The package from this project I used is called *ed_localization* and it also implements particle filters. The difference between these two packages is that while the *amcl* package works only with a static map provided by *gmapping* (or a similar package), *ed_localization* and *Environment Descriptor* in general offer an interesting extension. Besides the classic static map user can provide models of the real world objects by defining their positions and shapes in a yaml file. By that the system can work with a much more accurate world model which leads to more accurate localization results. Such an accuracy could be hardly achieved with a simple map from the *gmapping* package. I chose the *ed_localization* approach exactly because of this feature. Although I did not really use it, it is good to have that in the final solution in case of some future improvements.

The whole *ed_localization* package is configured in a specific yaml file. Most of the parameter values in this configure file I let set up to default values. Setting up the proper topics and frames names is important here and a path to the static map created in the previous step is also required. After the correct *ed_localization* configuration the module can be launched. The output of localization can be checked in the ROS *rviz* visualizer. The localization node should match the laser scan data to the walls and other objects in the map. It can be seen in the figure 4.4.

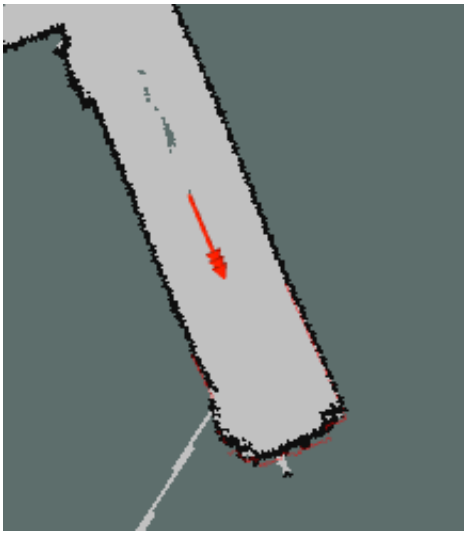


Figure 4.4: The localization module tries to match laser scan data (the red dots) to the objects in the map (the black pixels) using particle filters. The red arrow is a position and orientation of the localized robot.

4.4 Navigation

The navigation module in ROS is in general called a navigation stack. There is a package taking care of its functionalities, *move_base* and it has several files to be configured. The main launch file just loads other important files plus remaps the velocity topic name from *cmd_vel* to *mobile_base/commands/velocity* which is the topic name used by the *kobuki* package. The remaining configure files are for global costmap creation, local costmap creation and for configuring the parameters of the path planner itself like minimum and maximum velocities, allowed distances from the obstacles and other specific parameters for the algorithm. An example of a local costmap placed to the global one is shown in the figure 4.5. When the navigation stack is running, navigation goals can be sent to it through the *move_base/goal* topic. It can be done in the *rviz* visualizer by clicking somewhere on the map or in the code by sending a message to the mentioned topic.

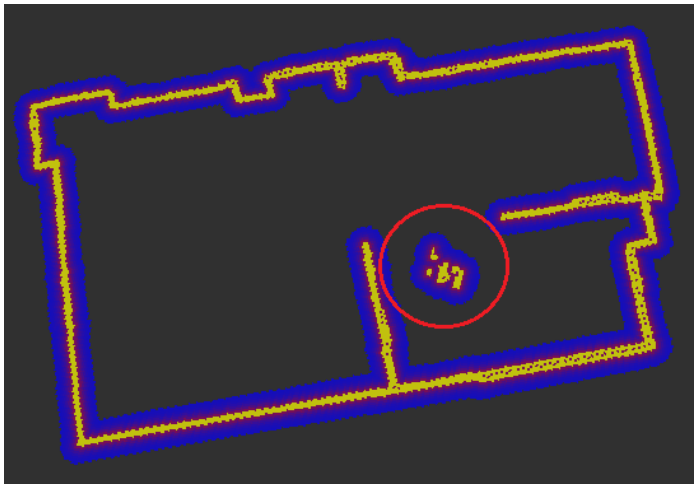


Figure 4.5: An example of a local costmap placed to a global costmap created by the navigation stack.

4.5 Objects detection

The detector module is implemented with the C++ library PCL. It is implemented as a subscriber to the point cloud topics from the depth sensor. The first step of the algorithm is preprocessing. That means noise and outliers are removed from the incoming point cloud using a *Passthrough* filter class from the PCL library. The preprocessing part also crops the point cloud so the points which are higher than a certain threshold or their distance is bigger than a certain threshold are removed. With the preprocessed point cloud next steps follow.

After receiving the first point cloud the floor plane is segmented. To do this I chose a PCL class *SACSegmentation*. It takes a segmentation method name as an argument plus a searched model type (which is a plane in this case) and the threshold values, which I set to 1 centimeter. As a segmentation method I used RANSAC. I chose the RANSAC algorithm as the robust estimator of choice which was motivated by its simplicity. More about RANSAC can be found in the section 2.3 or in [6]. Additional optimizations are often used with RANSAC, but the basic algorithm was sufficient enough in this case. The PCL

method returns coefficients of the segmented plane (which are values of the normal vector of the plane) and with them the angle between the sensor and the floor can be computed. Details about the angle computation were explained in the section 3.4. Having the angle, a transformation matrix can be prepared. With this transformation matrix every received point cloud is transformed until the next plane segmentation is done in which a new angle is computed and a corresponding transformation matrix is prepared. The time interval is controlled by a ROS *Time* class.

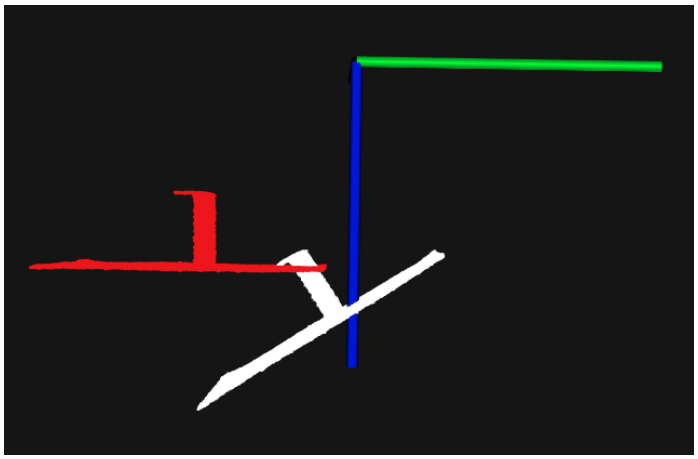


Figure 4.6: White point cloud is the original one, the red one is the point cloud after the rotation. Its floor plane has y coordinate with the same value everywhere so it can be easily removed.

The input point cloud and the point cloud after the transformation are captured in the figure 4.6. The white point cloud is the input one. Every y coordinate of the floor in the red point cloud (the point cloud after the rotation) has the same value. Using a simple PCL *Passthrough* filter method every point of the point cloud which has y coordinate in a given interval can be removed.

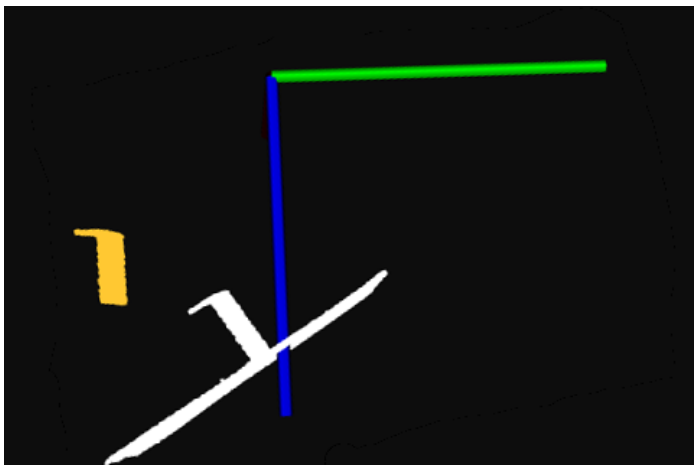


Figure 4.7: The result of the detection algorithm. The point cloud is rotated, the floor plane is removed and remaining objects are clustered.

I used interval 2 centimeters under and above the floor in case of some unexpected noise in the point cloud. The exact y value is computed as shown in the equation 3.2.

After the floor is removed the point cloud either remains empty or some objects are detected in it. The last step of the algorithm is to assign the remaining points to clusters representing the real world objects. My choice how to do that was a PCL class implementing the region growing algorithm mentioned in the section 2.3. This class provides options to set thresholds for the minimal and maximal number of points in the segmented area. This can be used to configure the size of the objects the robot will seek. The specific configured size depends on the specific type of the work which will be done by the robot. The result of the detection process (the transformation and the floor removal) are demonstrated in the figure 4.7.

4.6 Objects recognition

When the detected objects are saved in the clusters their images have to be sent to the image recognition module. The module I used is an *image_recognition* ROS package developed by the same research group as the *Environment Descriptor*. It is built on a *TensorFlow*[2] library and its neural network can classify trained objects.

The package nodes can be used either with its GUI plugins or by calling a corresponding ROS service from the code. The services are implemented in an *image_recognition* package. For the annotation of the training images I used the GUI plugin. It is shown in the figure 4.8.

Figure 4.8: ROS image recognition package with a GUI plugin. Annotating of objects is shown in this figure.

Annotating a few dozens of images (I used around 50) for every object was enough (although there were just little improvements after using more than 20 images). When having a training data set the neural network top layers can be retrained. The *tensorflow_ros_rqt* package provides the GUI plugin for the training process and a *tensorflow_ros* package does all the necessary computations using an *object_recognition_node* node. The node is a ROS service server, it can receive images of objects in service calls and return their likelihoods.

[2]https://www.tensorflow.org/

The paths to the training data sets and the training output graph are passed as arguments while launching the node.

The recognition itself is done in the objects detector node. Because the detector node creates clusters from depth data and the image recognition node takes a simple RGB image as an input, a transformation between the point cloud points coordinates and the corresponding image pixels has to be done. The detector is implemented as a subscriber to the point cloud topic as well as to the image topic (the *ASUS Xtion* sensor provides both). It processes the received messages synchronously, each point cloud and a corresponding image with the same timestamp. The synchronous subscription is achieved with a ROS *Synchronizer* class. As the detector has each object's points saved in a cluster, it can compute objects centroids, where a centroid is an average point of the whole cluster. Having the 3D coordinates of the centroid, the detector node iterates through all the points of the point cloud (which are implemented as a vector) finding the point with the most similar distance values to the centroid point. Knowing the resolution of the point cloud (*ASUS Xtion* has a resolution 640 x 480 px) the index of the found point can be easily assigned to the corresponding 2D pixel. A slightly different position of the image camera and the depth camera on the sensor (cca 3 centimeters) has to be also considered in the transformation from the depth point to the pixel coordinates.

After obtaining the position of the detected object in the 2D image a window or a region of interest is selected from the image. The window size is constant for every object now, but it would be worth considering to add a functionality for detecting the object size, so the region of interest would be automatically adjusted. For the image processing I used tools from the *OpenCV*[3] library. The selected window is sent via a ROS service call to the recognition node. The service call returns a vector of likelihoods for every object in the training data set, where one of the returned likelihoods is the likelihood for that the object is unknown. The best likelihood is chosen. If it is something else than an unknown object, the detector node publishes a navigation goal for the navigation stack. The position of the navigation goal is basically the same as the coordinates of the object centroid point. The robot's task ends here, the robot accomplished it by finding a known object.

4.7 Area exploration

I have not mentioned yet how the exploration node sends navigation goals to the *move_base* node. For this purpose I implemented another ROS node called *kobuki_exploration*. It is an actionlib client communicating with an actionlib server provided by a *frontier_exploration* package. Before launching the exploration the package configuration is required. This node creates a costmap similar to those created by the navigation stack. Also its configuration is very similar to the one from the navigation stack. The configuration file loads several plugins to build the map. Configuration of the individual plugins consists of setting map layers parameters, obstacle distances, sensor topics and frames names, robot footprint size, map updating frequency and parameters to compute cost of the area in the costmap.

The client part is implemented in python. It defines a polygon area which it tries to explore. Exploring the area means moving through the selected area until all of it is considered as seen. The client node connects to the actionlib server *frontier_exploration* and starts to send actionlib goals to explore the area. It continues exploring until the detector node publishes a message about detecting some objects. The exploration client

[3] http://opencv.org/

subscribes to this topic and after receiving such a message it stops exploration. The client does that by sending a cancel goal command to the actionlib server. The client renews the exploration again when it receives a new message about no objects detected. The detector node publishes the messages about stopping the exploration after some objects are detected. The exploration is stopped during the whole process of communication between the detector node and the image recognition node. If no objects were classified as known, the detector node publishes a message about not finding objects and the exploration client starts to send actionlib goals to the *frontier_exploration* server again.

4.8 Evaluation

The implemented software was used in a real world environment. This section demonstrates the results of the individual modules and the system as a whole in this environment.

Mapping

As a test room a chose the school robot laboratory. This room was appropriate because there is a big space with an empty floor but also space with lots of obstacles to avoid. The laboratory with its map is shown in the figure 4.14. In this image some of the characteristic features of the room shape can be seen. Same features are stored in the map. I edited the map in an editor a little bit, so the *Environment Descriptor* world model is more accurate. I also removed objects from the map (tables, chairs...), so just walls and cabinets remained there.

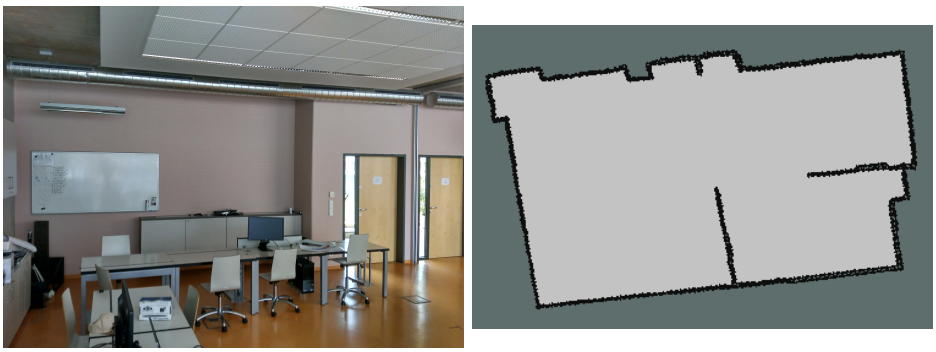


Figure 4.9: The map of the test environment.

Localization and navigation

The localization module was tested using the *rviz* visualizer. The accuracy of the localization is easily checked after displaying the map and the laser scan topics. If the localization is correct the laser scan should match the shape of the room in the map. After launching the localization module, the initial pose of the robot has to be set (*rviz* provides a tool for that). Without that the robot (usually) will not localize itself at all. It is maybe due to the fake laser scan, which is only less than a 180 degree scan, while the real scans (provided by lidar scanners) are usually 360 degree scans. A 360 degree scan has a greater power in particle filters localization as there is more data to match with the map.

After setting the initial pose (which do not have to be very accurate, it is enough to set the pose more than half a meter next to the real robot position) the robot could localize itself after it moved by a few centimeters. The result is demonstrated by the figure 4.10. As we can see it is not a hundred percent accurate, but this is definitely enough for our navigation purposes. There were also objects which are not captured by the map which is why the localization is not a hundred percent accurate.

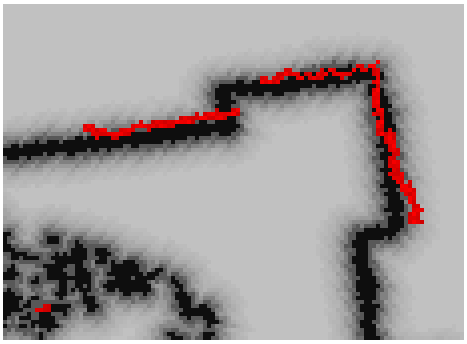


Figure 4.10: Accurate localization result.

When the robot moves too fast the localization is less accurate as it is shown in the figure 4.11. Although it may seem as a problem, it is not an actual issue at all. I set the robot speed to lower values so it usually does not come to that and if it does it does not matter in navigation as it gets back to the accurate state when the robot position is fixed or its speed is lowered.

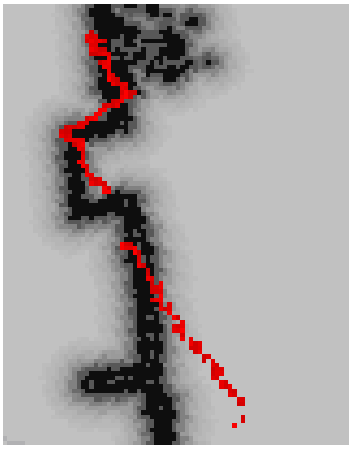


Figure 4.11: Inaccurate localization result. It happens when the robot moved too fast, it is repaired in a fraction of a second.

The navigation as a single module was tested with the *rviz* visualizer. It is done by setting a navigation goal in the map. The robot then tries to get to the given destination. The robot was tested in an area with no obstacles, a few obstacles and a lot of obstacles. If there are no obstacles or just a small amount of them the robot has no problem to navigate around them. The areas with lots of obstacles sometimes confused the robot and it finished with a failure. This result is not usually an actual issue (although it is sometimes and the robot fails), because the navigation module is controlled with the exploration module, which keeps sending new navigation goals although the previous one was not accomplished

(every time when the detector module stops the navigation after detecting objects as it was mentioned in the previous chapters).

Detection and recognition

The detection module was easily tested by placing some objects on the floor and capturing the scenery with the sensor. The detected objects were visualized as in the figure 4.12. The module itself had no problems with detecting objects in different situations and with different angles between the sensor and the floor plane. However the angle computation could not be used in the final system due to problems with transformations which I have not solve so far, so the angle has to be 90 degrees during the whole navigation process.

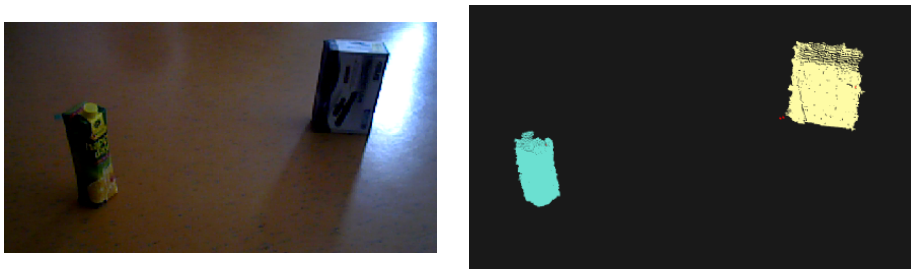


Figure 4.12: The result of the objects detection module. The right image shows clustered points of the point cloud representing the objects from the left image.

After computing a centroid of the detected object a region of interest for the recognition module is selected from the image. The result of its creation is shown in the figure 4.13. Also this part introduced no problems.

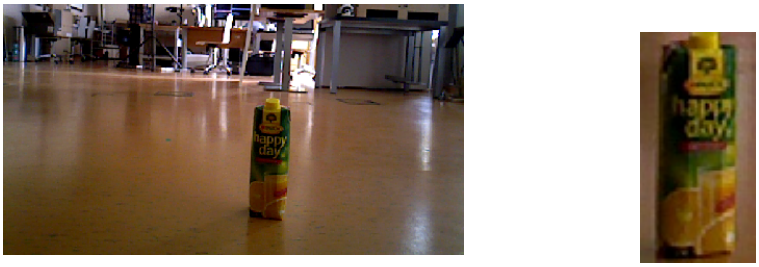


Figure 4.13: Region of interest sent to the recognition module.

The recognition module was trained to recognize four objects. For retraining the top layers of the neural network approximately fifty images for every object were annotated. If the region of interest sent to the module was accurate enough, the module recognized the objects correctly in most cases. In some cases it returned wrong likelihoods so an unknown object was considered to be a known one. Another issue (probably the worst one of the whole system) was that the recognizing of a single image takes quite a long time (cca 2 seconds).

Objects finding

The whole system was able to find a known object in most cases except the ones where the described problems occurred. Due to the low speed of the recognition module the whole

process of finding objects is a bit slow (if there are lots of objects detected in the robot's surroundings). Using a different recognition module would rapidly speed up the process.

Figure 4.14: The robot accomplishing a goal of finding an object (a juice box in this case).

Chapter 5

Conclusion

The goal of this work was to bring up a software for a mobile robot capable of localization, navigation and objects recognition in indoor scenarios using existing tools. The solution had to be as cheap as possible so the robot could be affordable by ordinary people. The goal was accomplished using a Robotic Operating System framework, a single depth sensor and a mobile base with a motor. The whole process of finding objects consists of several modules. Most of them are existing ROS packages, two modules were implemented by myself. The main parts of the system are a module for the map creation, localization module, navigation module, objects detector module, recognition module and explorer module. This work described how the individual modules work, how to use them and what results they gave.

The result of the work is an autonomous mobile robot that can find known objects in indoor scenarios. Such a robot could help people in many different areas where this functionality is required. That could be service robots in households or in industry (after equipping the robot with specific mechanical components). The final solution has several issues such as a slow recognition module and a transformation problem which makes it unable to change the angle between the sensor and the floor plane. The navigation often fails in difficult situations. The recognition module could be completely replaced by a better one in the future, which will require better knowledge of the machine learning problematics. Another important step in the future will be running the software on a hardware like Raspberry Pi instead of a notebook. Also implementing an autonomous map building module would be worth considering.

Bibliography

[1] Durrant-Whyte, H.; Bailey, T.: Simultaneous localization and mapping: part I. *IEEE Robotics Automation Magazine*. vol. 13, no. 2. June 2006: pp. 99–110. ISSN 1070-9932. doi:10.1109/MRA.2006.1638022.

[2] Fischler, M. A.; Bolles, R. C.: Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. *Commun. ACM*. vol. 24, no. 6. June 1981: pp. 381–395. ISSN 0001-0782. doi:10.1145/358669.358692.
Retrieved from: http://doi.acm.org/10.1145/358669.358692

[3] O'Kane, J. M.: *A Gentle Introduction to ROS*. Independently published. October 2013. ISBN 978-1492143239. available at http://www.cse.sc.edu/~jokane/agitr/.

[4] Pavlidis, T.; Liow, Y. T.: Integrating region growing and edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. vol. 12, no. 3. Mar 1990: pp. 225–233. ISSN 0162-8828. doi:10.1109/34.49050.

[5] Pereira, G. A. S.; Pimenta, L. C. A.; Fonseca, A. R.; et al.: Robot Navigation in Multi-terrain Outdoor Environments. *The International Journal of Robotics Research*. vol. 28, no. 6. 2009: pp. 685–700. doi:10.1177/0278364908097578. http://dx.doi.org/10.1177/0278364908097578.
Retrieved from: http://dx.doi.org/10.1177/0278364908097578

[6] Raguram, R.; Frahm, J.-M.; Pollefeys, M.: *A Comparative Analysis of RANSAC Techniques Leading to Adaptive Real-Time Random Sample Consensus*. Berlin, Heidelberg: Springer Berlin Heidelberg. 2008. ISBN 978-3-540-88688-4. pp. 500–513. doi:10.1007/978-3-540-88688-4_37.
Retrieved from: http://dx.doi.org/10.1007/978-3-540-88688-4_37

[7] Rusu, R. B.; Cousins, S.: 3D is here: Point Cloud Library (PCL). In *2011 IEEE International Conference on Robotics and Automation*. May 2011. ISSN 1050-4729. pp. 1–4. doi:10.1109/ICRA.2011.5980567.

[8] Szegedy, C.; Vanhoucke, V.; Ioffe, S.; et al.: Rethinking the Inception Architecture for Computer Vision. *CoRR*. vol. abs/1512.00567. 2015.
Retrieved from: http://arxiv.org/abs/1512.00567

[9] Thrun, S.; Fox, D.; Burgard, W.; et al.: Robust Monte Carlo localization for mobile robots. *Artificial Intelligence*. vol. 128, no. 1. 2001: pp. 99 – 141. ISSN 0004-3702. doi:http://dx.doi.org/10.1016/S0004-3702(01)00069-8.
Retrieved from:
http://www.sciencedirect.com/science/article/pii/S0004370201000698

[10] Thrun, S.; Leonard, J. J.: *Simultaneous Localization and Mapping*. Berlin, Heidelberg: Springer Berlin Heidelberg. 2008. ISBN 978-3-540-30301-5. pp. 871–889. doi:10.1007/978-3-540-30301-5_38.
Retrieved from: http://dx.doi.org/10.1007/978-3-540-30301-5_38

Appendices

Appendix A

Table of content of the attached CD

The attached CD consists of these folders:

```
/
├── poster
│   └── poster.pdf
├── demo
│   └── video.mp4
├── src
│   ├── detector_pkg
│   ├── kobuki_exploration
│   └── launch_files
│       ├── environment_descriptor
│       │   └── ed_config.yaml
│       ├── frontier_exploration
│       │   └── frontier_global_map.launch
│       ├── move_base
│       │   ├── bale_local_planner_params.yaml
│       │   ├── costmap_common_params.yaml
│       │   ├── global_costmap_params.yaml
│       │   ├── local_costmap_params.yaml
│       │   └── move_base.launch
│       └── sensor
│           └── depthimg_to_laser.launch
├── doc
│   └── usage_guide.pdf
└── tech_report
    ├── xsykor25_Mobile_robot_localization.pdf
    └── latex_src
```